Report

on the implementation of the linear CTL model checking algorithm in Go

by

Luis Thiele

for the lecture Computer Aided Verification

August, 2024

Contents

1 Introduction

This project is about the implementation of the linear CTL model checking algorithm in the Go programming language discussed in the lecture Computer Aided Verification (2024) at the University of Salzburg, Austria [1][2][3]. The algorithm uses set arithmetic ($\cup, \cap, \setminus$) and as such a proper set type had to be implemented as well. As a result you can now define Kripke structures in `.txt` files together with CTL formulas and the program is able to parse them, detect errors, and display the formula results. For better overview all figures which are primarily code snippets have been moved to the end of the document.

1.1 About Go

Go is a programming language made by Google in 2007. The main goals were high performance, readability, and usability and establishing it as an alternative to other languages in the C family [4].

The most important aspect about Go for this project is that it is an object-oriented language without supporting inheritance. Rather you can define structs and methods on those structs (receiver functions). Additionally you can define interfaces which require certain functions. If the receiver functions of a struct cover all functions required by an interface then this struct implements the inteface. This goes against other languages like Java where you have to explicitly state that a class implements an interface.

2 Implementation

The entire implementation is made of 2 layers - so to say - interfaces and the actual implementation of those interfaces. The implementation of every single interface can be easily exchanged with another one without breaking the system (assuming said implementation is correct).

2.1 Overview

All Go code is inside the `golang` folder as shown in figure 1. The `types` folder contains the algorithm itself and all types defined for it:

- `set.go` contains the set arithmetic definition `ISet` implementation `Set`.
- `formula.go` contains the recursive formula definition `IFormula` and implementation `Formula` and uses `IKripkeStructure` and `ISet`.
- `label.go` contains the label definition `ILabel` and implementation `Label` and uses `IKripkeStructure` and `IFormula`.
- `state.go` contains the state definition `IState` and implementation `State` and uses `IKripkeStructure`, `ILabel`, and `ISet`.

- `kripkestructure.go` contains the definition `IKripkeStructure` and implementation `KripkeStructure` for Kripke Structures and uses `ISet`, `IState`, `ILabel`, and `IFormula`.

2.2 Sets

As a basis, the `ISet[T comparable]` interface (a `comparable` is a primitive, including `string`, a pointer or interface, or a struct made of primitives) requires to implement the methods

- `Add(T)` to fill the set with values,
- `Contains(T) bool` to check whether or not an object is an element of the set,
- `ForEach(f func(T))` to iterate over the set,
- and `Copy() ISet[T]` to copy the set such that the copy contains the same elements.

Of course, it requires you to also implement the arithmetic methods

- `Union(other ISet[T]) ISet[T]`,
- `Intersect(other ISet[T]) ISet[T]`,
- `Minus(other ISet[T]) ISet[T]`,
- and `Equals(other ISet[T]) bool`,

where `Union`, `Intersect`, and `Minus` should not edit any set object but rather always return a new one.

Since the backbone of the algorithm is based on set arithmetic it is very important that it is as performant as possible. Go provides a native map (hash table [5]) implementation which can be used for sets. Using this native implementation lets us do hash table checks "as low as possible" in hardware. Basically we use the key-set of the map as our set and set empty structs as values by defining the `Set` type with `type Set[T comparable] map[T]struct{}` which defines `T` as key type and `struct{}` (the empty struct) as value type. Such a set can be initialized by calling the static function `MakeSet` as shown in figure 2.

Now, we can simply add values to our set and implement `Add` as shown in figure 3 by putting an initialized empty struct into the map using the object to be added to the sat as map-key. If an element is added twice then the same key is used both times so it effectively is still only once in the set. `Contains` and `ForEach` are trivial as shown in figure 4 and figure 5. Finally, `Copy` can be implemented using the other methods already implemented as shown in figure 6.

These function implementations can now be used to implement the functions representing set arithmetic in a way such that even different implementations of

`ISet` can interact without problems on the interface layer. For `Union` we copy the first set, iterate over the second one using `ForEach`, add all these elements to the copy, and return the copy as shown in figure 7. For `Intersect` we make a new empty set, then iterate over the first set, and for each element which is in the second set (using `Contains`) we add it to the new empty set which is then returned as shown in figure 8. `Minus` is now trivial as we do the same thing again but only add elements to the new set that are not contained in the second set as shown in figure 9. Finally, for `Equals` we iterate over the first set and if any element is not contained in the second set we return false, then we repeat it the other way around as shown in figure 10.

2.3 Formulas

The `IFormula` interface requires you to implement a `Check() ISet[IState]` function which as the name suggests requires you to return the set of states for which the formula holds. Formulas are bound to Kripke structures (see section 3.2) and follow a recursive structure, i.e. a lot of implementations require an `IFormula` parameter, some require two, and some require none as shown in figures 11, 12, 13, 14, and 15 which show the structs all formula implementations (except label formulas) are based on.

2.3.1 Label formulas

Formulas of struct type `LabelFormula` are the only special kind of formulas which in this implementation can only be constructed in a certain way using the `ILabel` object (see section 3.2). The struct and implementation of label formulas is shown in figure 16 and figure 17. Basically, the struct contains the label to check for and the implementation simply initializes a new empty set, iterates over all states of the kripke structure, and adds the states which are labeled as such to the set, which is then returned.

2.3.2 "`true`" and "`false`" formulas

The `TrueFormula` and `FalseFormula` types are both of type `emptyFormula`. The implementation of the former simply returns all states of the kripke structure, while the later simply returns an empty set as shown in figure 18 and figure 19.

2.3.3 "$\neg$" formula

Such formulas are represented by type `NotFormula` which is of type `subFormula` and uses simple set arithmetic where $\neg\varphi$ is satisfied by the states

$$\{s \in S \mid s \not\models \varphi\} = S \setminus \{s \in S \mid s \models \varphi\}$$

as shown in figure 20.

2.3.4 "$\wedge$" and "$\vee$" formulas

Conjunctions and disjunctions are represented by types `AndFormula` and `OrFormula` which are of type `biSubFormula`. The implementation simply gets the results of both sub-formulas and then uses set arithmetic where

$$\{s \in S \mid s \models \varphi \wedge \psi\} = \{s \in S \mid s \models \varphi\} \cap \{s \in S \mid s \models \psi\}$$

and, of course,

$$\{s \in S \mid s \models \varphi \vee \psi\} = \{s \in S \mid s \models \varphi\} \cup \{s \in S \mid s \models \psi\}$$

as shown in figure 21 and figure 22.

2.3.5 "EX" formulas

The `EXFormula` type is of type `subFormula` and represents formulas of type $EX\varphi$. Figure 23 shows the implementation where we first get the set of states of the sub formula φ, then initialize an empty set, iterate over all states s_1 of the kripke structure and for each of those states we iterate over the result states s_2 of φ. Whenever s_1 has a transition to s_2 we add s_1 to the empty previously initialized set and finally in the end we return said set. This is done because

$$\{s \in S \mid s \models EX\varphi\} = \{s_1 \in S \mid s_2 \in S \wedge s_2 \models \varphi \wedge s_1 \longrightarrow s_2\}$$

by definition.

2.3.6 "EG" formulas

The `EGFormula` type is also of type `subFormula` and uses its fixpoint algorithm discussed in the lecture and shown in figure 24.

2.3.7 "EF" formulas

The `EFFormula` type is the first formula of type `equivalencyFormula`. This means that by equivalency

$$EF\varphi \equiv E[trueU\varphi]$$

the `EUFormula` type is being initialized in Go (such that the equivalency holds), passed as paremeter `equivalenceFormula` to this struct, and any `Check` call is simply passed to it instead and the result is returned.

2.3.8 "EU" formulas

The `EUFormula` type represents formulas of the form $E[\varphi\, U\, \psi]$ is of type `biSubFormula`, and also uses its fixpoint algorithm presented in the lecture and shown in figure 25.

2.3.9 "ER" formulas

The ERFormula type also represents formulas which can be represented by an equivalency and as such it is of type biEquivalencyFormula with

$$E[\varphi \, R \, \psi] \equiv \neg A[\neg\varphi \, U \, \neg\psi]$$

used for the implementation.

2.3.10 "AX" formulas

Type AXFormula, once again, is of type equivalencyFormula with

$$AX\varphi \equiv \neg EX\neg\varphi$$

used for the implementation.

2.3.11 "AG" formulas

Type AGFormula is of type equivalencyFormula with

$$AG\varphi \equiv \neg EF\neg\varphi$$

used for the implementation.

2.3.12 "AF" formulas

Type AFFormula is of type equivalencyFormula with

$$AF\varphi \equiv \neg EG\neg\varphi$$

used for the implementation.

2.3.13 "AU" formulas

Type AUFormula is of type biEquivalencyFormula with

$$A[\varphi \, U \, \psi] \equiv \neg E[\neg\psi \, U \, \neg\varphi \wedge \neg\psi] \wedge \neg EG\neg\psi$$

used for the implementation.

2.3.14 "AR" formulas

Finally, the type ARFormula is also of type biEquivalencyFormula with

$$A[\varphi \, R \, \psi] \equiv \neg E[\neg\varphi \, U \, \neg\psi]$$

used for the implementation.

2.4 Kripke structures, states, and labels

These are implemented in the structs `KripkeStructure`, `State`, and `Label`. Their implementation is trivial and does not need very deep explaning (see their usage in section 3). The only thing worth mentioning is that every `State` object contains a set of children which are the states it has a transition to.

3 Usage

Their usage designed to be intuitive and very easy. Next, it will now be shown how to represent the Kripke structure shown in figure 26.

3.1 Creating a Kripke structure

Initially, you initialize a new Kripke structure as shown in figure 27 and then everything else is created and initialized by using this `IKripkeStructure` object `ks`. So, first we create the labels like shown in figure 28, then we create our states as shown in figure 29 where we can conveniently pass the labels we want to assign to each state as dynamic parameters (you can also assign labels to each state by using `IState#AddLabel`). Next, we connect the states by assigning children to each state (a parent has a transition to each of its children) as shown in figure 31 which finishes our Kripke structure.

3.2 Creating and testing formulas

Once again, formulas are initialized by using our `IKripkeStructure` object `ks` with the exception of label formulas. We create these directly off our labels as shown in figure 33 where `fla_p` represents the formula p with

$$\{s \in S \mid s \models p\} = \{s_1, s_2, s_3, s_6, s_7, s_8\}$$

as result, `fla_q` represents q with

$$\{s \in S \mid s \models q\} = \{s_5\}$$

as result, and `fla_r` represents r with

$$\{s \in S \mid s \models r\} = \{s_4\}$$

(all of these are of type `IFormula`). We can now use the formula-builder methods required by `IKripkeStructure` and shown in figure 32 on the `ks` object to construct any formula.

Figure 33 shows examples how to construct more formulas. The object `fla1` represents formula EXp, `fla2` represents EGp, and `fla3` represents $E[p\,U\,q]$. The figure 34 shows these three formulas being checked and the results getting printed is shown in figure 35.

3.3 Using the Parser

The program can be used by passing the path of a `.txt` file as parameter which follows a certain format and defines a Kripke structure and formulas to run on it. Figure 36 shows a file where the Kripke structure of figure 26 is defined together with the formulas of section 3.2 which are then getting checked and the results being printed as shown in figure 37.

This file is pretty intuitive; You start with the "`states`" keyword and simply define a new state per line until the keyword "`transitions`" is used where you connect states by putting "`->`" (or "`<-`") between the states. Next, you define labels by using the keyword "`labels`" followd by a new label per line by first defining the label, then the "`:`" character, and then the list of states you want to assign the label to (use "`,`" as delimeter). Finally, you use the keyword "`formulas`" to define a formula per line to check.

The parser is very lenient when it comes to extra spaces and empty lines. Formulas are parsed such that you can freely use semantically unnecessary parentheses "`(`" and "`)`" or brackets "`[`" and "`]`" (an example can be found in the repository in file `kripkestructure_test.txt`). Additionally, it supports comments which can be introduced with classical "`//`" syntax.

4 Comparison to NuSMV

Finally, the runtime of the entire implementation has been compared to the program NuSMV where the above used Kripke structure has been translated into the format used by it together with a lot of formulas to check for and then ran 1000 times by using the `Measure-Command` Powershell command. To be more precise, the pre-compiled `NuSMV-2.6.0-win64` program has been run using the `-dcx` flag (disable computation of counter-examples) and compared to a compiled version of this project. The result is NuSMV requiring about 12.5 seconds for completion and the project about 9.5 seconds. There are, of course, some differences which may explain this time difference:

- NuSMV requires you to declare which states are initial states and returns whether or not all of these initial states hold for each formula. This project does not let you declare initial states and returns a list of states that hold for each formula.
- This project only allows you to assign labels to states whereas NuSMV allows you to declare variables, check asynchronous systems, and in general has a much bigger scope and many more capabilities.
- The formulas `ER` and `AR` are not supported by NuSMV. You muse use the previously mentioned equivalencies instead.

5 References

[1] Ana Sokolova. "Computer Aided Verification" Lecture at the University of Salzburg, Austria.
https://online.uni-salzburg.at/plus_online/ee/ui/ca2/app/desktop/#/slc.tm.cp/student/courses/664996?$ctx=lang=en, 2024.

[2] Edmund M. Clarke Jr., Helmuth Veith, Doron Peled, Daniel Kroenig, and Orna Grumberg. *Model Checking.* The MIT Press, second edition, 2018.

[3] Christel Bayer and Joost-Pieter Katoen. *Principles of Model Checking.* The MIT Press, 2008.

[4] The Go Authors. Frequently Asked Questions (FAQ) - The Go Programming Language.
https://go.dev/doc/faq#change_from_c, 2024.

[5] Andrew Gerrand. Go maps in action - The Go Programming Language.
https://go.dev/blog/maps, 2013.

A Go Code

A.1 Repository

The repository containing the source code can be found online at:
`https://github.com/CAS-ual-TY/CAV-CTL-MC`

A.2 Figures

This section contains all the figures (see the following pages). For better readability all tabulator characters (`'\t'`) have been replaced with single spaces.

```
CAV
├── golang
│   ├── parser
│   │   └── parser.go
│   ├── test
│   │   └── cav_test.go
│   ├── types
│   │   ├── formula.go
│   │   ├── kripkestructure.go
│   │   ├── label.go
│   │   ├── set.go
│   │   └── state.go
│   └── main.go
├── go.mod
├── kripkestructure_exam.txt
├── kripkestructure_report.txt
└── kripkestructure_test.txt
```

Figure 1: Go Code Structure

```go
func MakeSet[T comparable]() ISet[T] {
 return Set[T]{}
}
```

Figure 2: `MakeSet` function

```go
func (s Set[T]) Add(value T) {
 s[value] = struct{}{}
}
```

Figure 3: `Set#Add` method

```go
func (s Set[T]) Contains(value T) bool {
 _, ok := s[value]
 return ok
}
```

Figure 4: `Set#Contains` method

```go
func (s Set[T]) ForEach(f func(T)) {
 for value := range s {
  f(value)
 }
}
```

Figure 5: `Set#ForEach` method

```go
func (s Set[T]) Copy() ISet[T] {
 result := MakeSet[T]()
 s.ForEach(func(value T) {
  result.Add(value)
 })
 return result
}
```

Figure 6: `Set#Copy` method

```go
func (s Set[T]) Union(other ISet[T]) ISet[T] {
 result := s.Copy()
 other.ForEach(func(value T) {
  result.Add(value)
 })
 return result
}
```

Figure 7: `Set#Union` method

```go
func (s Set[T]) Intersect(other ISet[T]) ISet[T] {
 result := MakeSet[T]()
 s.ForEach(func(value T) {
  if other.Contains(value) {
   result.Add(value)
  }
 })
 return result
}
```

Figure 8: `Set#Intersect` method

```go
func (s Set[T]) Minus(other ISet[T]) ISet[T] {
 result := MakeSet[T]()
 s.ForEach(func(value T) {
  if !other.Contains(value) {
   result.Add(value)
  }
 })
 return result
}
```

Figure 9: `Set#Minus` method

```go
func (s Set[T]) Equals(other ISet[T]) bool {
 if other == nil {
  return false
 }
 result := true
 s.ForEach(func(value T) {
  if !other.Contains(value) {
   result = false
  }
 })
 other.ForEach(func(value T) {
  if !s.Contains(value) {
   result = false
  }
 })
 return result
}
```

Figure 10: `Set#Equals` method

```go
type emptyFormula struct {
 kripkeStructure IKripkeStructure
}
```

Figure 11: `emptyFormula` struct

```go
type subFormula struct {
 kripkeStructure IKripkeStructure
 formula         IFormula
}
```

Figure 12: `subFormula` struct

```go
type biSubFormula struct {
 kripkeStructure IKripkeStructure
 formula1        IFormula
 formula2        IFormula
}
```

Figure 13: `biSubFormula` struct

```go
type equivalencyFormula struct {
 kripkeStructure    IKripkeStructure
 formula            IFormula
 equivalenceFormula IFormula
}
```

Figure 14: `equivalencyFormula` struct

```go
type biEquivalencyFormula struct {
 kripkeStructure    IKripkeStructure
 formula1           IFormula
 formula2           IFormula
 equivalenceFormula IFormula
}
```

Figure 15: `biEquivalencyFormula` struct

```go
type LabelFormula struct {
 kripkeStructure IKripkeStructure
 label           ILabel
}
```

Figure 16: `LabelFormula` struct

```go
func (f *LabelFormula) Check() ISet[IState] {
 result := MakeSet[IState]()
 f.kripkeStructure.GetStates().ForEach(func(state IState)
 ↪  {
  if state.HasLabel(f.label) {
   result.Add(state)
  }
 })
 return result
}
```

Figure 17: `LabelFormula#Check` method

```go
func (f *TrueFormula) Check() ISet[IState] {
 return f.kripkeStructure.GetStates()
}
```

Figure 18: `TrueFormula#Check` method

```go
func (f *FalseFormula) Check() ISet[IState] {
 return MakeSet[IState]()
}
```

Figure 19: `FalseFormula#Check` method

```
func (f *NotFormula) Check() ISet[IState] {
 return f.kripkeStructure.GetStates().Minus(f.formula.Check())
}
```

Figure 20: `NotFormula#Check` method

```
func (f *AndFormula) Check() ISet[IState] {
 return f.formula1.Check().Intersect(f.formula2.Check())
}
```

Figure 21: `AndFormula#Check` method

```
func (f *OrFormula) Check() ISet[IState] {
 return f.formula1.Check().Union(f.formula2.Check())
}
```

Figure 22: `OrFormula#Check` method

```
func (f *EXFormula) Check() ISet[IState] {
 check := f.formula.Check()
 result := MakeSet[IState]()
 f.kripkeStructure.GetStates().ForEach(func(state IState) {
  check.ForEach(func(nextState IState) {
   if state.HasChild(nextState) {
    result.Add(state)
   }
  })
 })
 return result
}
```

Figure 23: `EXFormula#Check` method

```go
func (f *EGFormula) Check() ISet[IState] {
 p := f.formula.Check()

 var prevZ ISet[IState]
 var nextZ ISet[IState] = f.kripkeStructure.GetStates()

 for !nextZ.Equals(prevZ) {
  prevZ = nextZ

  exz := MakeSet[IState]()
  f.kripkeStructure.GetStates().ForEach(func(state IState) {
   prevZ.ForEach(func(nextState IState) {
    if state.HasChild(nextState) {
     exz.Add(state)
    }
   })
  })

  nextZ = p.Intersect(exz)
 }
 return prevZ
}
```

Figure 24: `EGFormula#Check` method

```
func (f *EUFormula) Check() ISet[IState] {
 p := f.formula1.Check()
 q := f.formula2.Check()

 var prevZ ISet[IState]
 var nextZ ISet[IState] = MakeSet[IState]()

 for !nextZ.Equals(prevZ) {
  prevZ = nextZ

  exz := MakeSet[IState]()
  f.kripkeStructure.GetStates().ForEach(func(state
↪  IState) {
   prevZ.ForEach(func(nextState IState) {
    if state.HasChild(nextState) {
     exz.Add(state)
    }
   })
  })

  nextZ = q.Union(p.Intersect(exz))
 }
 return prevZ
}
```

Figure 25: `EUFormula#Check` method

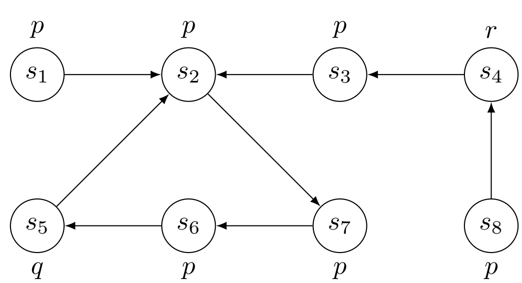

Figure 26: Example Kripke Structure

```
ks := cav.MakeKripkeStructure()
```

Figure 27: `IKripkeStructure` initialization

```
p := ks.NewLabel("p")
q := ks.NewLabel("q")
r := ks.NewLabel("r")
```

Figure 28: `ILabel` initialization

```
s1 := ks.NewState("s1", p)
s2 := ks.NewState("s2", p)
s3 := ks.NewState("s3", p)
s4 := ks.NewState("s4", r)
s5 := ks.NewState("s5", q)
s6 := ks.NewState("s6", p)
s7 := ks.NewState("s7", p)
s8 := ks.NewState("s8", p)
```

Figure 29: `IState` initializations with label assignments

```
s1.AddChildren(s2)
s2.AddChildren(s7)
s3.AddChildren(s2)
s4.AddChildren(s3)
s5.AddChildren(s2)
s6.AddChildren(s5)
s7.AddChildren(s6)
s8.AddChildren(s4)
```

Figure 30: `IState` child assignments

```
fla_p := p.MakeLabelFormula()
fla_q := q.MakeLabelFormula()
fla_r := r.MakeLabelFormula()
```

Figure 31: Label formulas

```
MakeTrueFormula() IFormula
MakeFalseFormula() IFormula
MakeNotFormula(formula IFormula) IFormula
MakeAndFormula(formula1 IFormula, formula2 IFormula)
 ↪  IFormula
MakeOrFormula(formula1 IFormula, formula2 IFormula)
 ↪  IFormula
MakeEXFormula(formula IFormula) IFormula
MakeEGFormula(formula IFormula) IFormula
MakeEFFormula(formula IFormula) IFormula
MakeEUFormula(formula1 IFormula, formula2 IFormula)
 ↪  IFormula
MakeERFormula(formula1 IFormula, formula2 IFormula)
 ↪  IFormula
MakeAXFormula(formula IFormula) IFormula
MakeAGFormula(formula IFormula) IFormula
MakeAFFormula(formula IFormula) IFormula
MakeAUFormula(formula1 IFormula, formula2 IFormula)
 ↪  IFormula
MakeARFormula(formula1 IFormula, formula2 IFormula)
 ↪  IFormula
```

Figure 32: `IKripkeStructure` methods

```
fla1 := ks.MakeEXFormula(fla_p)
fla2 := ks.MakeEGFormula(fla_p)
fla3 := ks.MakeEUFormula(fla_p, fla_q)
```

Figure 33: Example formulas

```
fmt.Println(fla1.Check())
fmt.Println(fla2.Check())
fmt.Println(fla3.Check())
```

Figure 34: Checking results of the formulas in figure 33

```
{s3, s4, s5, s7, s1, s2}
{}
{s3, s6, s5, s7, s1, s2}
```

Figure 35: Results from figure 34

```
states
s1
s2
s3
s4
s5
s6
s7
s8

transitions
s1 -> s2 -> s7 -> s6 -> s5 -> s2
s8 -> s4 -> s3
s3 -> s2

labels
p: s1, s2, s3, s6, s7, s8
q: s5
r: s4

formulas
EX p
EG p
E[p U q]
```

Figure 36: `kripkestructure_report.txt`

```
Formula Results:
EXp:
{s2, s3, s4, s5, s7, s1}
EGp:
{}
E[p U q]:
{s3, s6, s7, s5, s1, s2}
```

Figure 37: Results from figure 36