06/02/2022

Formal Modeling & Verification of Critical Systems

Mini-Project: Implementation of a CTL Model Checker

Rémy CHERKAOUI ; Quentin FELTEN ; Daniil ROSSO
M2-SE2 EFREI PARIS, PROMOTION 2022

Table of Content

Introduction:

Model checking is the operation of verifying if a system satisfies a set of properties. It can be seen as an exhaustive exploration algorithm of the system.
It is very useful for instance in computer science to ensure a variable is never null or to verify that a system cannot reach a deadlock state.

In this project, we designed an application running a *Computation Tree Logic* (or **CTL** for short) algorithm to check if a given Kripke structure satisfies a specific CTL formula.

Using mostly vanilla JavaScript technology, we were able to create a simple graphical interface for the user and make the project easier of access to someone wanting to experiment with it. You can find all the source code at the following repository:

https://github.com/QuentinFelten/FormalModeling_MiniProjet

Input Formats:

For the general input format, we chose a structure suggested by the instructions: a Kripke structure followed by a CTL formula. Having the input as a .txt file ensures small scale maintainability, and it would probably be the simplest short-term solution thanks to its ease of use and of modification by the reviewer. Implementing a new Kripke structure is simple: the user only needs to create a new txt file or overwrite an existing one. Plus, this method would make testing during the development phase that much easier.

Note: During the development of this project, we approached you with a very peculiar bug that forbid us from dynamically parsing and using file names. As agreed, the solution is hard coded for the sake of demonstration, but it should be understood as any file of the format fileX.txt would be readable by the system.

The Kripke structure is constituted of four parts:

S: the list of states.

L: the label of each state.

T: the list of transitions.

I: the initial state.

Let us consider the following example:

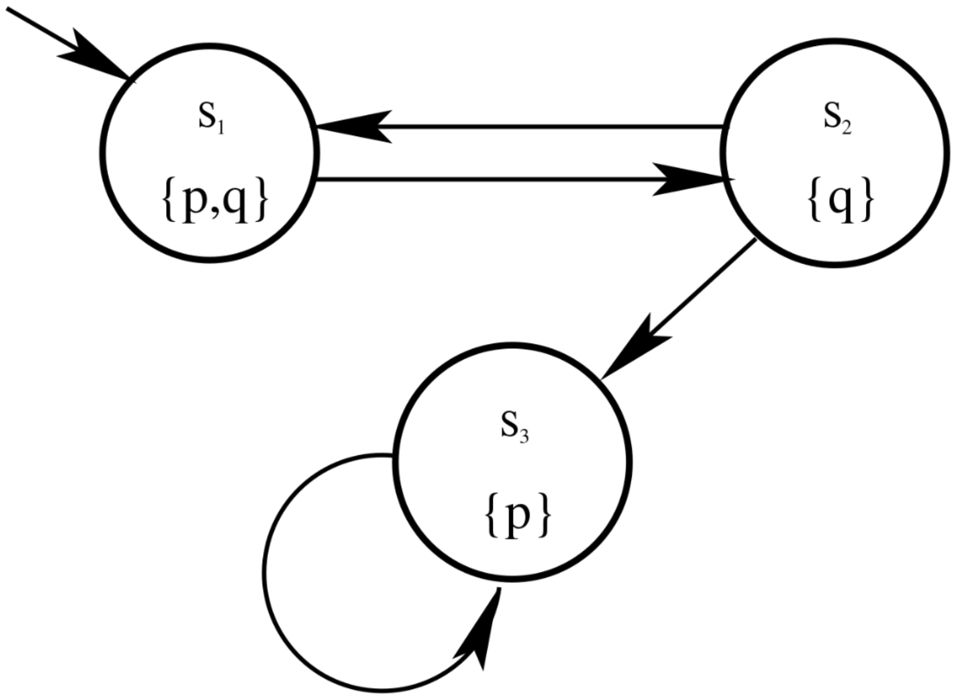

Figure 1: test file 1

- **S**: we have 3 states here (S1, S2, S3).
- **L**: the labels are p and q for S1, q for S2 and p for S3.
- **T**: there are four transitions here (the opening transition doesn't count): from S1 to S2, from S2 to S1, from S2 to S3 and from S3 to S3.
- **I**: S1 is the initial state here.

So, the formatted Kripke structure would be:

```
S:
{S1, S2, S3}
L:
{(S1,{p,q}), (S2,{q}), (S3,{p})}
T:
{(S1,S2), (S2,S1), (S2,S3), (S3,S3)}
I:
{S1}
CTL: //this part can be changed at will by the user
!(&(p,q))
```

To input a CTL proposition, You can either:

- Enter a CTL proposition on one line at the end of a test file.
- Enter the custom CTL proposition in the dedicated text filed while the application is running.

The CTL propositions must respect a strict syntax to be understood by the program:

```
NOT x      → !(x)
x AND y    → &(x,y)
x OR y     → |(x,y)
For All    → A
Exists     → E
Next x     → E(X(x)) or A(X(x))
x Then y   → E(T(x,y)) or A(T(x,y))
Finally x  → E(F(x)) or A(F(x))
x Until y  → E(U(x,y)) or A(U(x,y))
Globally x → E(G(x)) or A(G(x)) // Work In Progress
```

Here is an explanation:

Let us say that we wanted to implement the following CTL proposition:

¬ (p ^ q)

The correct input syntax for this proposition would be:

!(&(p,q))

The functions used in the application to compute such CTL propositions are listed below:

```
Implementation of the CTL proposition
{
    marking: marking(atom1),
    not: not(sub_func),
    and: and(atom1, atom2),
    or: or(atom1, atom2),
    ex: EX(sub_func),
    ax: AX(sub_func),
    eu: EU(sub_func1, sub_func2),
    au: AU(sub_func1, sub_func2),
    ef: EF(sub_func),
    af: AF(sub_func),
    et: ET(sub_func1, sub_func2),
    at: AT(sub_func1, sub_func2),
    eg: EG(sub_func), /// Work In Progress
    ex: EX(sub_func),
    eu: EU(sub_func1, sub_func2),
    au: AU(sub_func1, sub_func2),
}
```

Data Structures:

The data structure we use in our algorithm are objects representing both the Kipke structure, and the CTL proposition. This object has properties with explicit names to facilitate manipulation and data extraction down the line.
It is created when reading the text files containing the structures and are used throughout the application since then.

Below is a snippet of our code representing a Kripke structure element:

```
Implementation of the data structure as an object
{
    states: states,
    tuples: tuples,
    transitions: transitions,
    initial: initial_state,
    ctl: ctl
}
//Note: The CTL is stored as a string that will be parsed and simplified by a
//series of functions.
```

Using this very “to the point” structure makes further use of elements of this object much simpler and easier to understand for someone reading the code, since the transitions attribute designs clearly the table of state transitions for example.

Design of the Complementary Verification Algorithms:

```
OR :
marking(Ψ1);
marking(Ψ2);
forall q in Q do :
q. φ := q.Ψ1 ∨ q. Ψ2
```

```
EF :
marking(Ψ);
forall q in Q do
        q.φ :=false;
        q.seenbefore :=false
L := ∅;
```

```
AF :
marking(Ψ);
forall q in Q do
        q.φ :=false;
Q.nb := degree(q);
L := ∅;
forall q in Q do
        if q.Ψ =true then
        L :=L[{q}
while L != ∅ do
        pick q from L ; L :=L\{q};
        q.φ :=true;
        forall (q';_; q) in T do
```

```
AF (cont) :
forall q in Q do
        if q.Ψ =true then
        L :=L[{q}
while L != ∅ do
        pick q from L ; L :=L\{q};
        q.φ :=true;
        forall (q';_; q) in T do
                if  q'.seenbefore=false
                then
                        q'.seenbefore
                        :=true;
                        L :=LU{q'}
q'.nb := q'.nb - 1
if q.nb = 0 and q'.φ  = false then
                        L :=LU{q'}
```

ET and AT were way simpler to implement, as they are made of functions previously made:

```
ET:
φ := marking(Ψ1 ∧ EX(Ψ2))
```

```
AT:
φ := marking(Ψ1 ∧ AX(Ψ2))
```

Encountered Difficulties:

Kripke Parser:

One of the main difficulties we had to overcome was the implementation of our Kripke structure parser. As it is one of the core elements of our algorithm, it is a major piece of our project that required lots of time to tweak it to perfection through trials and error until it satisfied our needs. Since we got started with the idea of a simple txt data storage, we had to manage all the string processing and data purification. In hindsight, stocking this data in a json would've been much more convenient for the development.

CTL Parser:

Speaking of parser, we encountered a similar problem when working on the CTL parser that now reads CTL propositions for the rest of the algorithm to process. The challenge was figuring out a way to program a code that would *understand* the priorities of a formula – that is, for example, parentheses have priority over other computation elements. For example, there is a strict difference between algorithms requiring one argument and those requiring two. It created the need of two different splitting logics for left-hand and right-hand elements, with some healthy use of mathematic and programming tricks. The parsing is optimized to a degree with the use of regular expressions (regex) and it could be pushed even further. Again, making this parser work as intended required lots of trial and error and was a main chunk of the work volume of this project.

The two "G" functions (EG & AG):

As stated previously, the "G" functions are currently listed as "Work In Progress", as we couldn't make an algorithm that could correctly verify them. To be more precise, the problem lies in the detection of an infinite loop or an end that verifies the sub-function. Our attempt at it and main roadblock is the function named "state_explorer" in the algorithms.js file, doesn't return what it should.

General difficulties regarding NodeJS:

On a more general approach, the project was developed in JavaScript using mostly vanilla features and some NodeJS. But, while NodeJS is a very powerful tool, it proved to sometimes be erratic and caused a few problems that to our surprise, did not come from our work or the libraries we used but rather from NodeJS itself. Some inconcistencies made use hard refresh pages, or even entirely reboot the project to double check that the issue at hand was not a cache problem and truly a bug.
In the end, NodeJS turned out to be a very handy complementary tool even if it presents its own drawbacks. However, it was quite difficult to set up and maintain.

Graphical User Interface

Another part of the project which turned out to be very time consuming was implementing a GUI that would not fail in front of the user. One major issue was the output of the algorithm being display to the user every time they pressed the Launch button. Not a big issue, yes, but an issue nonetheless that we think was worth spending some time fixing it. Fixing all the various small use cases allowed us to refine our knowledge of front end developing as well as the knowledge of JQuery, which remains an exceedingly useful tool in this situation.

Possible improvements:

Graphical User Interface:

Having a user-friendly GUI was a clear and general objective since the beginning of the work on this project. As of now, we are happy with the results we have : It is simple, straight forward and explicit. It can both display all the information at once or let the user decide what information they want to be able to read.

Still our solution can be improved in many ways: The current GUI is good but in no way good looking and might have been improved to something more pleasant to use – that is something prettier. We also would have liked having as a display option the possibility to have a representation of the graph of the Kripke structure, but the timeframe was a little limited for that.

Random Generators

It would have been a great feature to have a generator to create both *Kripke Structures* and *CTL propositions* on the fly (by simply pressing a button for example).

However, such features require strong and robust algorithms that we decided to work on only after finishing more important functions – the basic CTL parser and EG & AG. Due to a lack of time, we could not work on these generators as we were too busy finishing more critical parts of the application.

In conclusion:

At the end of this project, we are happy to enjoy a better understanding of Model Checking and CTL.

On a more technical point of view, the project was the opportunity for us to refine a relatively large spectrum of programming skills. Not only did we practice JavaScript programming, but we also made this project the opportunity to learn how to program an easy-to-use web application using NodeJS and JQuery, among other more basic web technologies.

Thank you for your time reading this paper,

The team.