Allen Emerson

Emerson is co-inventor and co-developer of model checking , an algorithmic method of verifying nominally finite-state concurrent programs. The method was proposed in 1981 in a paper widely recognized as seminal for founding today's broad field of model checking and establishing its basic principles. He won the 2007 A.M. Turing Award along with Edmund M. Clarke and Joseph Sifakis for the invention and development of Model checking.

The Verification Problem

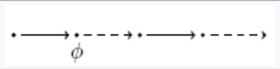
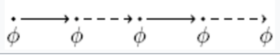
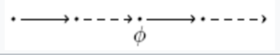
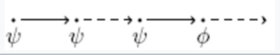
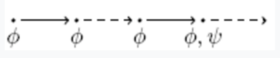
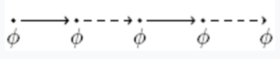
As defined in the paper the problem asks if given some program M and specification h, then is it possible to determine whether the behavior of M meets the specification h.

A famous example of this is the halting problem proposed by Turing which states that given a Turing machine M with the specification h that the machine eventually should halt. The condensed version of the proof goes something like this. Assume we have a black box program/algorithm that can tell us if a program will halt. Then we modify that program so if it does halt it loops forever and if doesn't halt then it immediately halts. So a program that never halts will halt but a program that halts infinitely loops which is a contradiction.

Model checking like symbolic execution we discussed on Tuesday was created as a result of the difficulties of manual proofs and the difficulties when reasoning about concurrent programs. I particularly liked the analogy to the task of a human adding 100,000 decimal numbers of 1,000 digits each. This is rudimentary in principle, but likely impossible in practice for any human to perform reliably. It's an example of the verification problem. For instance my research involves modeling individual instructions from the x86 architecture and verifying the specifications of the manuals and general security principles are upheld.

The first step towards model checking was recognizing temporal logic could be applied to the properties

Linear Temporal Logic

Textual	**Symbolic†**	**Explanation**	**Diagram**
Unary operators:			
X ϕ	$\bigcirc\phi$	ne**X**t: ϕ has to hold at the next state.	
G ϕ	$\Box\phi$	**G**lobally: ϕ has to hold on the entire subsequent path.	
F ϕ	$\Diamond\phi$	**F**inally: ϕ eventually has to hold (somewhere on the subsequent path).	
Binary operators:			
ψ **U** ϕ	$\psi\,\mathcal{U}\,\phi$	**U**ntil: ψ has to hold *at least* until ϕ, which holds at the current or a future position.	
ψ **R** ϕ	$\psi\,\mathcal{R}\,\phi$	**R**elease: ϕ has to be true until and including the point where ψ first becomes true; if ψ never becomes true, ϕ must remain true forever.	 

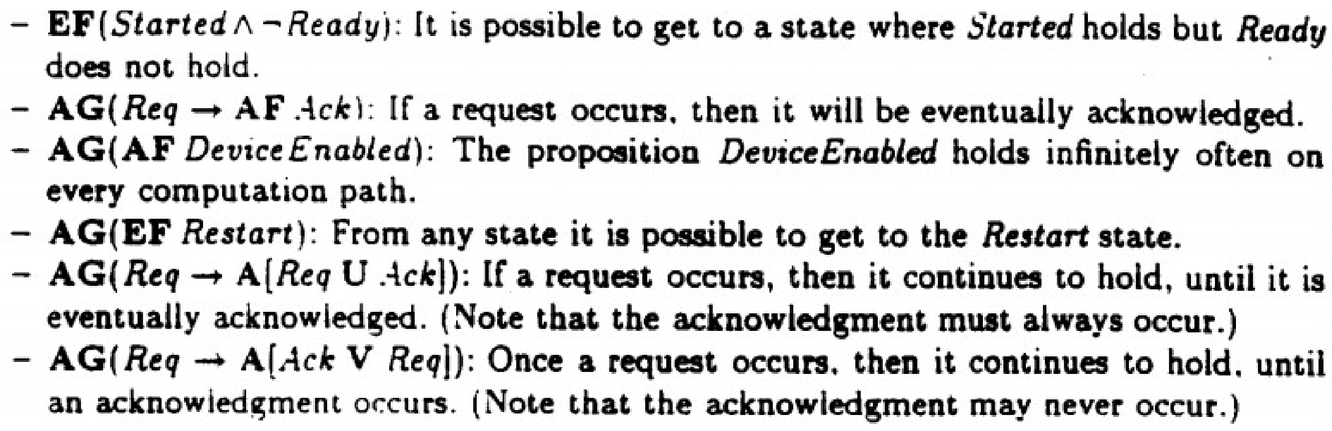
G ¬(owns1 ∧ owns2) : It is never the case that both processes own the resource.

G(req1 ⇒ F owns1) : Whenever process 1 has requested the resource, it will eventually obtain it.

G F(req1 ∧ ¬(owns1 ∨ owns2)) ⇒ G F owns1 : If it is infinitely often the case that process 1 has requested the resource when the resource is free, then process 1 infinitely often owns the resource.

Branching Temporal Logic

- Quantifiers over paths
  - **A** ϕ – **A**ll: ϕ has to hold on all paths starting from the current state.
  - **E** ϕ – **E**xists: there exists at least one path starting from the current state where ϕ holds.



- **EF**(*Started* ∧ ¬*Ready*): It is possible to get to a state where *Started* holds but *Ready* does not hold.
- **AG**(*Req* → **AF** *Ack*): If a request occurs, then it will be eventually acknowledged.
- **AG**(**AF** *DeviceEnabled*): The proposition *DeviceEnabled* holds infinitely often on every computation path.
- **AG**(**EF** *Restart*): From any state it is possible to get to the *Restart* state.
- **AG**(*Req* → **A**[*Req* **U** *Ack*]): If a request occurs, then it continues to hold, until it is eventually acknowledged. (Note that the acknowledgment must always occur.)
- **AG**(*Req* → **A**[*Ack* **V** *Req*]): Once a request occurs, then it continues to hold, until an acknowledgment occurs. (Note that the acknowledgment may never occur.)

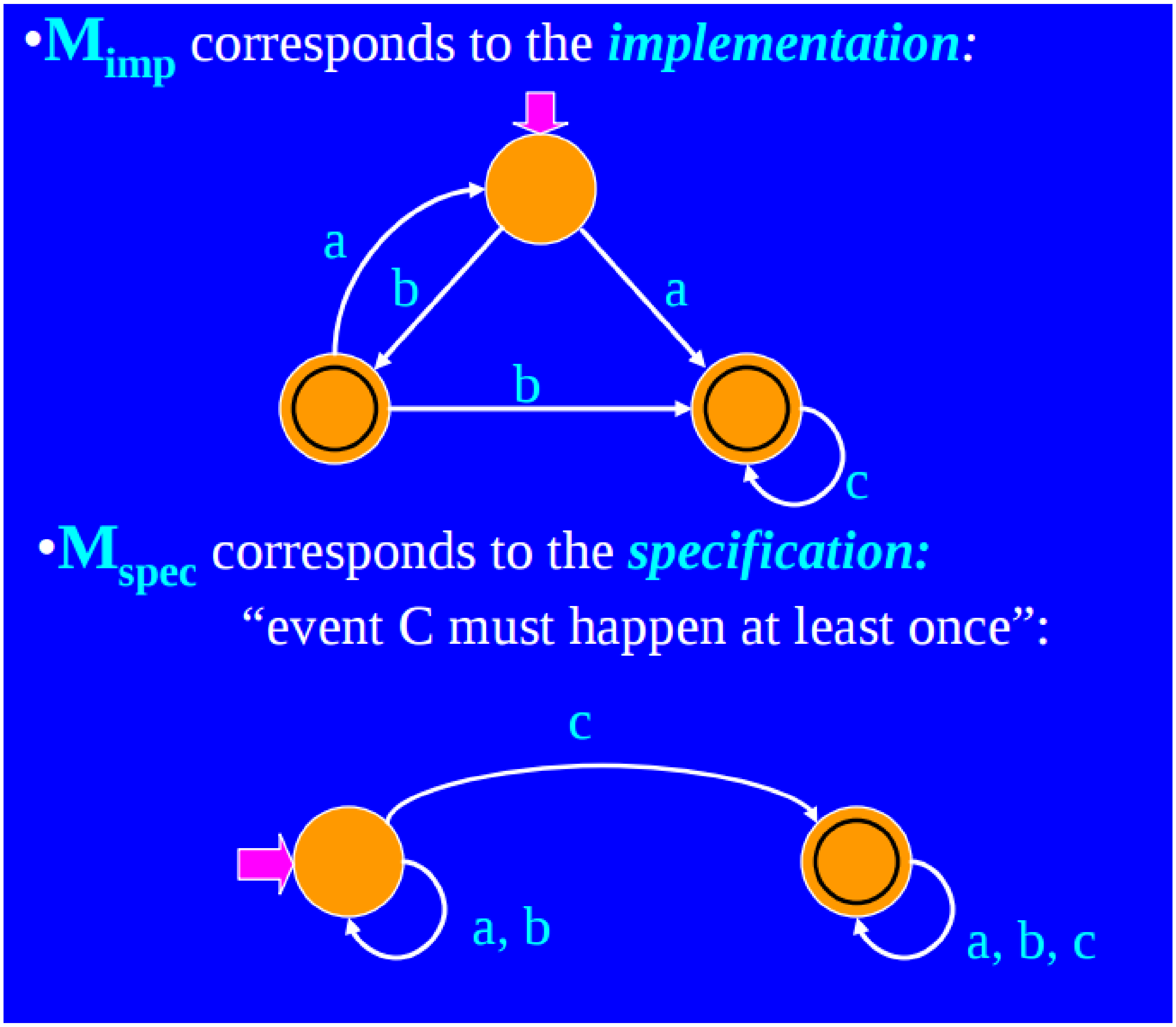
In CTL*, the temporal operators can be freely mixed. In CTL, the operator must always be grouped in two: one path operator followed by a state operator. CTL* is strictly more expressive than CTL.

The Mu-calculus may be thought of as extending CTL with a least fixpoint (u) and greatest fixpoint (v) operator. A fixpoint is where f(x) = x. So for example f(x^2) has the least fixpoint when x =0 and the greatest when x = 1. It provides a single, simple and uniform framework that characterizes modalities in terms of recursively defined tree-like patterns subsuming most other logics of interest for reasoning.

Emerson used this combination of the tree-like structures which allows for recursion and Mu-calculus' ability to express other LTL and CTL formulas to apply the Tarski-Knaster theorem which acted to verify the model.

Since this isn't the current way model checkers verify models there isn't a significant amount of information about this technique that I could find.

A more modern technique is to turn both the implementation and specification into an automata. If a sequence is accepted by M imp then it is also accepted by Mspec and only if this holds true can the specification be said to be upheld. This can be determined algorithmically



•M_{imp} corresponds to the *implementation:*

•M_{spec} corresponds to the *specification:*
"event C must happen at least once":

State Explosion Problem

One of the major problems with model checking is the more complicated the model is the number of states grows exponentially. There are a number of ways to combat this problems

Abstraction

Given original system M an abstraction is obtained by suppressing detail yielding a simpler and likely smaller system M that is, ideally, equivalent to M for purposes of verification. The precise nature of the abstraction and the correspondence between M and M can vary considerably. For example in my research there may be a property that is only considering the current privilege level (what ring the cpu is in) so even though other registers are being affected at the same time I don't specifically need to include them in my model.

Symmetry
If there are multiple instances of the same module, then it may be possible to only model one of those modules because if one fails to meet the specifications then they all will fail.

Compositional Reasoning
It may be possible to split a larger model into its individual components for modeling. In my research we accomplish this by giving the component symbolic inputs. This means any possible input given by the larger part of the model is considered.

Theoretical Improvements
Structures such as Binary Decision Diagrams can be used to represent state transition systems more efficiently. Partial order reduction is used for reducing the size of the state-space to be searched by exploiting the commutativity of concurrently executed transitions which result in the same state when executed in different orders.

Practical Improvements

The increase of computing power and memory acts in a brute force way to handle the state explosion problem

Some personal thoughts on model checking

From my personal experience sometime creating the model can be easy but it can be difficult to find the specification or combination of specifications that ensures your model is valid when the model is sufficiently complex. In regards to my research creating the models for the individual instructions is relatively simple, but finding a specification that tests the model in a nontrivial way is a lengthy, research-filled process.

Additionally the model you are making is much more abstract than you would first imagine. Often you don't have to model the minor details. For example when making sure an address in canonical you don't have to explicitly test that the 48th bit through the 64th bit are all zeros or ones

Example of Model Checking

I want to show you a simple example using UCLID, the software I use, which is slightly different as it allows for the modeling of infinite state systems.

I want to show a toy model in Uclid that shows what a model, counter-examples, and assertions look like.

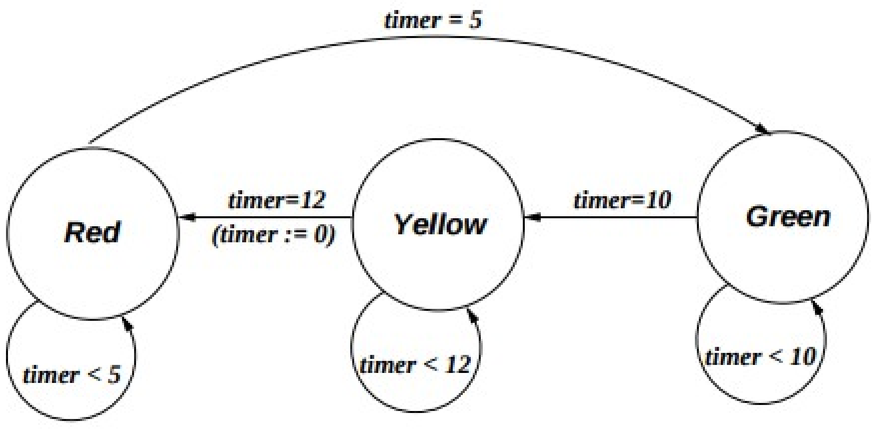


Figure 2: A timed traffic light

Thoughts about the paper

I thought this paper was a little weak in the fact it seemed to bounce around from topic to topic without really giving the depth needed for any one in particular. It was more interesting from a historical perspective than a computer science one in my opinion

Does anyone else have opinions about the paper?

What did you think of the grand challenges he presented at the end of the paper

Grand Challenge for Hardware. Hardware designs with a few hundreds to thousands of state variables can be model checked in some fashion; but not an entire microprocessor. It would be a Grand Challenge to verify an entire microprocessor with one hundred thousand state variables.

Grand Challenge for Software. Software device drivers have been shown amenable to software model checking. These are mostly sequential software with up to one hundred thousand lines of code. Of course, there is software with millions of lines of code. Windows Vista contains somewhat over 50 million lines of code, and entails concurrency as well. It would be a Grand Challenge to to verify software with millions to tens of millions lines of code