Model Checking

Baher S. Salama

May 28, 2013

Outline

- Overview
- Kripke Structures
- Temporal Logics (CTL*, CTL, LTL)
- Model Checking Problem
- Büchi Automata
- Solution algorithm
- State explosion problem
- Model-checking in practice

Model Checking

- An automated technique for formal software verification

Model Checking

- An automated technique for formal software verification
- Introduced in 1981 by Clarke and Emerson (USA) and Sifakis (France)

Model Checking

- An automated technique for formal software verification
- Introduced in 1981 by Clarke and Emerson (USA) and Sifakis (France)
- Uses temporal logic to reason about the correctness of a system

Model Checking

- An automated technique for formal software verification
- Introduced in 1981 by Clarke and Emerson (USA) and Sifakis (France)
- Uses temporal logic to reason about the correctness of a system
- Works with finite-state concurrent system.

Advantages of Model Checking

- Advantages over other formal methods:

Advantages of Model Checking

- Advantages over other formal methods:
    - Requires minimal human intervention (and less experience)

Advantages of Model Checking

- Advantages over other formal methods:
    - Requires minimal human intervention (and less experience)
    - Applies to systems with realistic properties (concurrent interactive/event-based systems)

Advantages of Model Checking

- Advantages over other formal methods:
    - Requires minimal human intervention (and less experience)
    - Applies to systems with realistic properties (concurrent interactive/event-based systems)
    - Not restricted to input-processing-output paradigm.

Advantages of Model Checking

- Advantages over other formal methods:
  - Requires minimal human intervention (and less experience)
  - Applies to systems with realistic properties (concurrent interactive/event-based systems)
  - Not restricted to input-processing-output paradigm.
  - Produces a *counter-example*, in case of failure.

Advantages of Model Checking

- Advantages over other formal methods:
  - Requires minimal human intervention (and less experience)
  - Applies to systems with realistic properties (concurrent interactive/event-based systems)
  - Not restricted to input-processing-output paradigm.
  - Produces a *counter-example*, in case of failure.
- Advantages over testing/simulation techniques:

Advantages of Model Checking

- Advantages over other formal methods:
  - Requires minimal human intervention (and less experience)
  - Applies to systems with realistic properties (concurrent interactive/event-based systems)
  - Not restricted to input-processing-output paradigm.
  - Produces a *counter-example*, in case of failure.
- Advantages over testing/simulation techniques:
  - Testing cannot cover all the possible cases.

Steps of Model Checking

1. Modeling
   Converting the system to a formalism accepted by the model checker. (Kripke Structure)

Steps of Model Checking

1. Modeling
   Converting the system to a formalism accepted by the model checker. (Kripke Structure)
2. Specification
   Specifying the desired properties in a formal language. (Temporal Logic)

Steps of Model Checking

1. Modeling
   Converting the system to a formalism accepted by the model checker. (Kripke Structure)
2. Specification
   Specifying the desired properties in a formal language. (Temporal Logic)
3. Verification
   Running the model checking algorithm.

Steps of Model Checking

1. Modeling
   Converting the system to a formalism accepted by the model checker. (Kripke Structure)
2. Specification
   Specifying the desired properties in a formal language. (Temporal Logic)
3. Verification
   Running the model checking algorithm.
4. Analysis

Steps of Model Checking

1. Modeling
   Converting the system to a formalism accepted by the model checker. (Kripke Structure)
2. Specification
   Specifying the desired properties in a formal language. (Temporal Logic)
3. Verification
   Running the model checking algorithm.
4. Analysis
   - If the result is **yes**, no analysis is required.

Steps of Model Checking

1. Modeling

   Converting the system to a formalism accepted by the model checker. (Kripke Structure)

2. Specification

   Specifying the desired properties in a formal language. (Temporal Logic)

3. Verification

   Running the model checking algorithm.

4. Analysis

   - If the result is **yes**, no analysis is required.
   - If the result is **no**, counter-example needs to be analyzed to discover the source of the bug.

Kripke Structure

A formalism for specifying the possible states of a system and their transition relations.

Definition

A *Kripke Structure* M over a set of atomic propositions AP is a 4-tuple:

$$M = \langle S, S_0, R, L \rangle$$

where:

1. S is a finite set of states.
2. $S_0 \subseteq S$ is the set of starting states.
3. $R \subseteq S \times S$ is a transition relation.
4. $L : S \rightarrow \mathcal{P}(AP)$ is function that labels each state with the set of propositions that are true in that state.

Alternative definition for $L : S \rightarrow (AP \rightarrow \{\top, \bot\})$

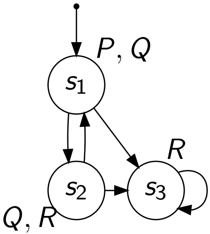
Kripke Structure Example

Define Kripke Structure M_1 over the atomic propositions $AP = \{P, Q, R\}$ as follows:

$$M_1 = \langle \{s_1, s_2, s_3\}, \{s_1\}, R_1, L_1 \rangle$$

where:

- $R_1 = \{(s_1, s_2), (s_2, s_1), (s_1, s_3), (s_2, s_3), (s_3, s_3)\}$
- $L_1 = \{(s_1 \to \{P, Q\}), (s_2 \to \{Q, R\}), (s_3 \to \{R\})\}$

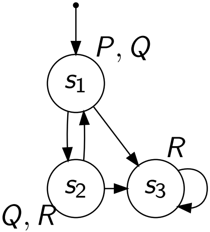


Paths

Definition (Path)

A *path* π in a Kripke Structure $M = \langle S, S_0, R, L \rangle$ is an infinite sequence of states $s_0, s1, \ldots$ such that for each $i \geq 0$, $(s_i, s_{i+1}) \in R$.

- The notation π^i refers to the subsequence of π starting at s_i (i.e. $s_i, s_{i+1}, \ldots$)
- Kripke Structure unwinding



Temporal Logics

- Temporal logics: logics of time

Temporal Logics

- Temporal logics: logics of time
- Two major classes of temporal logics:

Temporal Logics

- Temporal logics: logics of time
- Two major classes of temporal logics:
  1. First-order: times are treated as first-order objects
     E.g.: Situation Calculus, Interval Calculus

Temporal Logics

- Temporal logics: logics of time
- Two major classes of temporal logics:
  1. First-order: times are treated as first-order objects
     E.g.: Situation Calculus, Interval Calculus
  2. Modal: uses states or *possible worlds*

Temporal Logics

- Temporal logics: logics of time
- Two major classes of temporal logics:
  1. First-order: times are treated as first-order objects
     E.g.: Situation Calculus, Interval Calculus
  2. Modal: uses states or *possible worlds*
- In model-checking, temporal modal logics are used to specify the desirable properties of the system.

Temporal Logics

- Temporal logics: logics of time
- Two major classes of temporal logics:
  1. First-order: times are treated as first-order objects
     E.g.: Situation Calculus, Interval Calculus
  2. Modal: uses states or *possible worlds*
- In model-checking, temporal modal logics are used to specify the desirable properties of the system.
- Commonly used TLs are CTL*, CTL, and LTL.

Temporal Logics

- Temporal logics: logics of time
- Two major classes of temporal logics:
    1. First-order: times are treated as first-order objects
       E.g.: Situation Calculus, Interval Calculus
    2. Modal: uses states or *possible worlds*
- In model-checking, temporal modal logics are used to specify the desirable properties of the system.
- Commonly used TLs are CTL*, CTL, and LTL.
- LTL is a *linear-time logic*

Temporal Logics

- Temporal logics: logics of time
- Two major classes of temporal logics:
  1. First-order: times are treated as first-order objects
     E.g.: Situation Calculus, Interval Calculus
  2. Modal: uses states or *possible worlds*
- In model-checking, temporal modal logics are used to specify the desirable properties of the system.
- Commonly used TLs are CTL*, CTL, and LTL.
- LTL is a *linear-time logic*
- CTL and CTL* are *branching-time logics*

CTL*

- stands for "Computational Tree Logic*"
- is a superset of LTL and CTL.
- CTL* has 2 types of formulas:
    1. Path formulas: specify properties of a given path.
    2. State formulas: specify properties of a given state.

CTL* Path Operators

- **X** f ("Next"): The property f holds in the *next state* of the given path.
- **F** f ("future"): The property f holds *finally* (eventually).
- **G** f ("globally"): The property f holds *globally* (in all future states of the path).
- f **U** g ("until"): Property f must hold *until* g holds. g is required to become true eventually.
- f **R** g ("release"): Property g must hold up-to and including the first state in which f holds. g is *released* by f.
- **Examples:** P **U** Q, P **R** Q.

CTL* Syntax

- Given a set of atomic propositions AP,
- the syntax of **state formulas** is defined as follows:
  1. every proposition $p \in AP$ is a state formula. (Holds if p is true in the given state)
  2. If f and g are state formulas, then $\neg f$, $f \wedge g$, $f \vee g$ are state formulas.
  3. If f is a *path formula*, the $\mathbf{A} f$ and $\mathbf{E} f$ are state formulas.
- **A** and **E** are path quantifiers.
- The syntax of **path formulas** is defined as follows:
  1. If f is a *state formula* then f is also a path formula. (Holds if f is true in the first state of the path)
  2. If f and g are *path formulas* then $\neg f$, $f \wedge g$, $f \vee g$.
  3. If f and g are *path formulas* then $\mathbf{X}\, f$, $\mathbf{F}\, f$, $\mathbf{G}\, f$, $f \mathbf{U}\, g$, and $f \mathbf{R}\, g$.

CTL* Formal Semantics

- CTL* semantics are defined in terms of a Kripke structure.
- Given a Kripke structure $M = \langle S, S_0, R, L \rangle$, a state s in M and a state formula f, the notation:

  $$M, s \models f$$

  means that f in true in M at state s.
- Given a path π through M, and a path formula g, the notation:

  $$M, \pi \models g$$

  means that g is true in M over path π.
- Also referred to as M, s *models* f, or M, s *satisfies* f.

CTL* Formal Semantics

Given a Kripke structure $M = \langle S, S_0, R, L \rangle$. Let $p \in AP$ be an atomic proposition, f_1 and f_2 be *state formulas*, g_1 and g_2 be *path formulas*:

1. $M, s \models p$ iff $p \in L(s)$
2. $M, s \models \neg f_1$ iff $M, s \not\models f_1$
3. $M, s \models f_1 \vee f_2$ iff $M, s \models f_1$ or $M, s \models f_2$.
4. $M, s \models f_1 \wedge f_2$ iff $M, s \models f_1$ and $M, s \models f_2$.
5. $M, s \models \mathbf{E} g_1$ iff there is a path π starting at s such that $M, \pi \models g_1$.
6. $M, s \models \mathbf{A} g_1$ iff for every path π starting at s, $M, \pi \models g_1$.

CTL* Formal Semantics

Given a Kripke structure $M = \langle S, S_0, R, L\rangle$. Let $p \in AP$ be an atomic proposition, f_1 and f_2 be *state formulas*, g_1 and g_2 be *path formulas*:

1. $M, \pi \models f_1$ iff s is the first state in π and $M, s \models f_1$.
2. $M, \pi \models \neg g_1$ iff $M, \pi \not\models g_1$
3. $M, \pi \models g_1 \vee g_2$ iff $M, \pi \models g_1$ or $M, \pi \models g_2$.
4. $M, \pi \models g_1 \wedge g_2$ iff $M, \pi \models g_1$ and $M, \pi \models g_2$.
5. $M, \pi \models \mathbf{X}\, g_1$ iff $M, \pi^1 \models g_1$.
6. $M, \pi \models \mathbf{F}\, g_1$ iff there exists a $k \geq 0$ such that $M, \pi^k \models g_1$.
7. $M, \pi \models \mathbf{G}\, g_1$ iff for all $k \geq 0$, $M, \pi^k \models g_1$.
8. $M, \pi \models g_1 \mathbf{U}\, g_2$ iff there exists a $k \geq 0$ such that $M, \pi^k \models g_2$ and for all $0 \leq i < k$, $M, \pi^i \models g_1$.
9. $M, \pi \models g_1 \mathbf{R}\, g_2$ iff for all $j \geq 0$, if for every $i < j$ $M, \pi^i \not\models g_1$ then $M, \pi^j \models g_2$.

CTL* Examples

Examples:

- $M, s \models \mathbf{EF}\, p$
- $M, s \models \mathbf{AF}\, p$
- $M, s \models \mathbf{EG}\, p$
- $M, s \models \mathbf{AG}\, p$

LTL

- stands for "Linear-Time Logic"
- is a subset of CTL*
- all formulas are (implicitly) universally quantified
- no explicit path quantifiers are used in state formulas (i.e. all state formulas are atomic)
- Provides operators for describing events along a *single* path.
- Example: **F G** p
  At some point in the future, all the following states will have the property p.

CTL

- stands for "Computational-Tree Logic"
- subset CTL* where only *state* formulas are allowed.
- every temporal operator (**F** , **G** , **X** , **U** , **R**) must be quantified.
- Example: **EF AG** p
- CTL operators:
  1. **AX** and **EX**
  2. **AF** and **EF**
  3. **AG** and **EG**
  4. **AU** and **EU**
  5. **AR** and **ER**

The Model Checking Problem

- Using the previous definitions, the Model-Checking problem can be defined as follows:

$$M \models \phi$$

- Given:
  1. a finite model M represented as a Kripke structure, and
  2. a specification formula ϕ specified in TL,

  check whether the model satisfies the given formula.

Frequently-Used Properties

- Safety: "Something bad will *never* happen"

$$M \models \mathbf{G}\,\neg p$$

- Liveness: "Something good will *eventually* happen"

$$M \models \mathbf{F}\,p$$

Finite State Machines

Definition (Finite State Machine)

A Finite State Machine (FSM) $\mathcal{A}$ is defined as a 5-tuple:

$$\mathcal{A} = \langle Q, \Sigma, \Delta, Q_0, F \rangle$$

where:

- Q is a finite set of states,
- Σ is a finite *alphabet*,
- $\Delta \subseteq Q \times \Sigma \times Q$ is a *transition relation*,
- $Q_0 \subseteq Q$ is a set of *initial states*,
- $F \subseteq Q$ is a set of *final states*.

FSM Acceptance

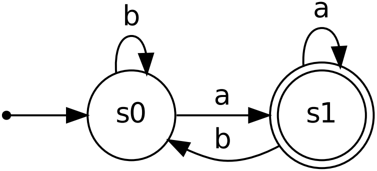
- A FSM accepts a word $w \in \Sigma^*$ if there is a sequence of states $s_0, s_1, \ldots, s_n$ such that:
  1. $s_0 \in Q_0$,
  2. $s_n \in F$,
  3. for each $1 \leq i \leq n$, $(s_{i-1}, w_i, s_i) \in \Delta$, where w_i is the i-th character of w.
- The language of a FSM $\mathcal{A}$, denoted $\mathcal{L}(\mathcal{A})$, is the set of all words accepted by $\mathcal{A}$.

FSM Example

Example:

$$\mathcal{A}_1 = \langle \{s_0, s_1\}, \{\mathtt{a}, \mathtt{b}\}, \Delta, \{s_0\}, \{s_1\} \rangle$$

where: $\Delta = \{(s_0, \mathtt{b}, s_0), (s_0, \mathtt{a}, s_1), (s_1, \mathtt{a}, s_1), (s_1, \mathtt{b}, s_0)\}$



This FSM accepts all words that end with an a.

Büchi Automata

- A Büchi Automaton is a FSM that recognizes *infinite* words.
- This concept is called ω-acceptance.

Definition (Büchi Automaton)

A Büchi Automaton $\mathcal{B}$ is defined as a 5-tuple:

$$\mathcal{B} = \langle Q, \Sigma, \Delta, Q_0, F \rangle$$

where:

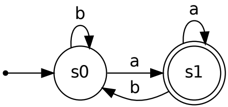
- Q is a finite set of states,
- Σ is a finite *alphabet*,
- $\Delta \subseteq Q \times \Sigma \times Q$ is a *transition relation*,
- $Q_0 \subseteq Q$ is a set of *initial states*,
- $F \subseteq Q$ is a set of *final states*.

Büchi Automaton Acceptance (ω-acceptance)

- A Büchi Automaton has a finite number of states.
- However, it recognizes infinite words.
- Therefore, some of the states have to be visited *infinitely many* times.
- A Büchi Automaton accepts a word w if there is an infinite path $\rho = s_0, s_1, \ldots$ such that:
  1. $s_0 \in Q_0$,
  2. For all $i \geq 1$, (s_{i-1}, w_i, s_i),
  3. If $inf(\rho)$ denotes the set of states visited *infinitely-many* times in ρ, then $inf(\rho) \cap F \neq \emptyset$.
- A Büchi Automaton accepts a word if at least one of the final states is visited infinitely-many times.
- The language of a Büchi Automaton $\mathcal{B}$, denoted $\mathcal{L}(\mathcal{B})$ is the set of all (infinite) words it accepts.
- Note that $\mathcal{L}(\mathcal{B}) \subseteq \Sigma^\omega$, where Σ^ω is the set of infinite words over Σ.

Büchi Automaton Example

The following Büchi Automaton accepts all words that have infinitely-many a's:



- For example, it accepts the word $(\mathtt{ab})^\omega = ababab\ldots$.
- In general, it accepts words described by the follows ω-regular expression $(\mathtt{b}^*\mathtt{a})^\omega$.

From Kripke to Büchi

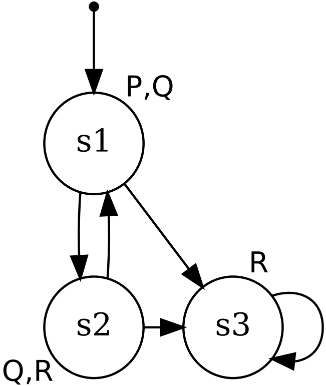
Convert a Kripke structure $M = \langle S, S_0, R, L \rangle$ over atomic propositions AP to a Büchi automaton $\mathcal{B} = \langle Q, \Sigma, \Delta, Q_0, F \rangle$ such that:

1. $Q = S \cup \{i\}$,
2. $\Sigma = \mathcal{P}(AP)$, (i.e. each transition is labeled with a subset of AP)
3. Same transitions as the Kripke structure in addition to:
    1. Transitions going from i to each of the start states in S_0.
    2. Each transition is *labeled* with the set of predicates of the *target state*.
4. $Q_0 = \{i\}$
5. $F = Q$, (All states are accept states)

The resulting Büchi Automaton accepts words equivalent to possible state sequences in the Kripke structure.

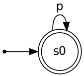
From Kripke to Büchi (Example)

Example:
Convert the following Kripke structure, defined over $AP = \{P, Q, R\}$, to a Büchi automaton:

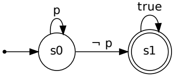


Modeling LTL Properties with Büchi Automata

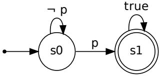
- Every LTL formula over *AP* can be modeled as a Büchi automaton with alphabet $\Sigma = \mathcal{P}(AP)$.
- The language of the Büchi automaton is the set of *paths* that *satisfy* the LTL formula.
- Examples:
  - **G** p



  - $\neg$**G** p



  - **F** p



LTL Model Checking with Büchi Automata

Given a model M represented as a Kripke structure, and an LTL formula ϕ, the following algorithm decides whether $M \models \phi$:

1. Convert M to a Büchi Automaton $\mathcal{B}_1$.
2. Construct a Büchi Automaton $\mathcal{B}_2$ equivalent to the *negation* of ϕ ($\neg\phi$).
3. Construct a Büchi Automaton $\mathcal{B}_3$ that recognizes the language $\mathcal{L}(\mathcal{B}_3) = \mathcal{L}(\mathcal{B}_1) \cap \mathcal{L}(\mathcal{B}_2)$, by calculating the *cross-product* for $\mathcal{B}_1 \times \mathcal{B}_2$.
4. Check the language of $\mathcal{B}_3$ for emptiness:
    - If the language is empty, then ϕ *holds* in M.
    - If not, then ϕ *does not hold* in M. Any word $w \in \mathcal{L}(\mathcal{B}_3)$ is a counter-example.

Emptiness Check for Büchi Automata

Given a Büchi Automaton $\mathcal{B}_3$, the following algorithm determines whether its language is empty.

1. Determine the *strongly-connected components* (SCC) in $\mathcal{B}_3$.
2. If there is a *reachable*, *non-trivial* strongly-connected component that *contains a final state*, then the language is *not* empty. Otherwise, the language is empty.

Notes:

- A *trivial* SCC, is one that contains only 1 state without a self-transition.
- A *reachable* SCC, is one that can be reached from a start state.

Time Complexity of Model Checking

- There exist several model checking algorithms.
- The best ones currently have the following upper-bound time-complexities for a formula ϕ and model M:
  - LTL: $O(|M| \cdot 2^{|\phi|})$
  - CTL: $O(|M| \cdot |\phi|)$
  - CTL*: $O(|M| \cdot 2^{|\phi|})$

  $|M| = n + m$, where n in the no. of states, and m is the no. of transitions.
- The following lower-bounds have also been proven for model-checking:
  - LTL: PSPACE-Complete
  - CTL: P-Complete
  - CTL*: PSPACE-Complete

State-Explosion Problem and Solutions

- One of the major-challenges facing model checking.

State-Explosion Problem and Solutions

- One of the major-challenges facing model checking.
- Refers to the exponential increase in the number of possible states with processes and data.

State-Explosion Problem and Solutions

- One of the major-challenges facing model checking.
- Refers to the exponential increase in the number of possible states with processes and data.
- A system with n asynchronous processes, each having m states has up-to m^n states.

State-Explosion Problem and Solutions

- One of the major-challenges facing model checking.
- Refers to the exponential increase in the number of possible states with processes and data.
- A system with n asynchronous processes, each having m states has up-to m^n states.
- State transition system for n-bits of data has 2^n states.

State-Explosion Problem and Solutions

- One of the major-challenges facing model checking.
- Refers to the exponential increase in the number of possible states with processes and data.
- A system with n asynchronous processes, each having m states has up-to m^n states.
- State transition system for n-bits of data has 2^n states.
- A lot of research has been (and is being) done on the state-explosion problem.

State-Explosion Problem and Solutions

- One of the major-challenges facing model checking.
- Refers to the exponential increase in the number of possible states with processes and data.
- A system with n asynchronous processes, each having m states has up-to m^n states.
- State transition system for n-bits of data has 2^n states.
- A lot of research has been (and is being) done on the state-explosion problem.
- The following are the major results:

State-Explosion Problem and Solutions

- One of the major-challenges facing model checking.
- Refers to the exponential increase in the number of possible states with processes and data.
- A system with n asynchronous processes, each having m states has up-to m^n states.
- State transition system for n-bits of data has 2^n states.
- A lot of research has been (and is being) done on the state-explosion problem.
- The following are the major results:
  - Ordered binary decision diagrams (OBDDs):
    Works on synchronous systems and has been used for systems with up-to 10^{120} states.

State-Explosion Problem and Solutions

- One of the major-challenges facing model checking.
- Refers to the exponential increase in the number of possible states with processes and data.
- A system with n asynchronous processes, each having m states has up-to m^n states.
- State transition system for n-bits of data has 2^n states.
- A lot of research has been (and is being) done on the state-explosion problem.
- The following are the major results:
  - Ordered binary decision diagrams (OBDDs): Works on synchronous systems and has been used for systems with up-to 10^{120} states.
  - Partial order reduction: Works on asynchronous systems and exploits certain *mutual-independence* properties of parallel processes.

SPIN and Promela

- LTL model checker.
- SPIN stands for "**S**imple **P**romela **In**terpreter"
- Model is specified in **Promela**
- Promela stands for "**Pro**cess **Me**ta **La**nguage"
- Supports parallel synchronous or asynchronous processes that communicate using global variables or message passing.

Structure of a Promela Model Specification

- A Promela specification consists of:
    - type declarations
    - channel declarations
    - variable declarations
    - process declarations
    - Optionally: `init` process
- since the model needs to be finite, data, channels and processes must be bounded.

Process Declaration in Promela

- A process is declared using the `proctype` keyword.
- Process declaration consists of:
  1. process name
  2. list of parameters
  3. local variable declaration
  4. body

Promela Statements

- Promela statements can be either *executable* or *blocked*
- A blocked statement blocks the execution until the statement becomes *unblocked*
- statements:
  - `skip`: always executable
  - `assert`(<expr>): asserts that <expr> should always be true. always executable.
  - expression: executable if not zero.
  - assignment: always executable.
  - `if :: fi`: Provides non-deterministic choice. Executable if at least one choice is executable.
  - `do :: od`: Like if but repeats. Executable if at least one choice is executable.
  - `break`: Exits a `do` statement. Always executable.

Mutual Exclusion Problem

- Organizing access to a shared resource such that:
  1. At most 1 process uses the resource at any given time.
  2. Every interested process can eventually get access to the resource.
- The program part that accesses a shared resource is called the *critical region*.

Phony Mutual Exclusion Algorithm

```
int flag = 0;

void enter_critical() {
    while(flag != 0);
    flag = 1;
    critical_region();
    flag = 0;
}
```

- Flaw: If process 2 reads the flag before process 1 sets it to 1, both processes will enter critical region at the same time.

Using SPIN to Discover the Bug

References

- Clarke, Edmund M., Orna Grumberg, and Doron Peled. *Model checking*. The MIT press, 1999.
- Clarke, Edmund M., E. Allen Emerson, and Joseph Sifakis. "Model checking: algorithmic verification and debugging." *Communications of the ACM* 52.11 (2009): 74-84.
- Holzmann, Gerard J. "The model checker SPIN." *Software Engineering, IEEE Transactions on 23.5* (1997): 279-295.

Thank you