University of Zurich[UZH]

Department of Informatics

Martin Glinz

Software Quality

Chapter 2

Model Checking

2.1 Motivation

2.2 Temporal Logic

2.3 Principles of Model Checking with LTL

2.4 Model Checking in Practice

Proving programs and properties

When developing critical software, we are interested in formally proving that

- A program is correct (i.e., it satisfies its specification)
- A model actually has certain required properties

- First case: Classical program proofs, i.e. proving $P \vdash S$ for a program P and its specification S
- Second case: This kind of proof is called Model Checking: Let M be a model and Φ a required property (typically specified as a formula in temporal logic). We have to prove that $M \models \Phi$, i.e., M satisfies Φ.

[Clarke and Emerson 1981, Queille and Sifakis 1982]

Ways of using Model Checking

Model Checking is typically used in two ways:

- Partial verification of programs:

  Let M be a program and Φ some critical part of its specification. $M \models \Phi$ means proving the correctness of program M with respect to the part Φ of its specification

- Proving properties of a specification:

  Let M be a specification and Φ a property that this specification is required to have. $M \models \Phi$ means proving that the property Φ actually holds for this specification

Classes of properties to be proven

- There are two classes of required properties
- Safety properties: unwanted/forbidden/dangerous states shall never be reached
- Liveness properties: desired states shall always be reached sometimes

[Lamport 1977; Owicki and Lamport 1982]

- Typical safety properties: impossibility of deadlock, guaranteed mutual exclusion
- Typical liveness properties: eventual termination of a program, impossibility of starvation or livelock

2.1 Motivation

2.2 Temporal Logic

2.3 Principles of Model Checking with LTL

2.4 Model Checking in Practice

Expressing time in logic formulae

[Pnueli 1977]

- Safety and liveness properties imply a notion of time
- However: no notion of state or time in propositional logic and predicate logic
- Extension needed for state or time dependent statements
- Various potential forms of temporal and modal logic
- We use Linear temporal logic (LTL) here

Linear time logic (LTL)

- Time is modeled as an ordered sequence of discrete states
- The existential and universal quantifiers of predicate logic are generalized to four temporal quantifiers:
  - S holds forever from now
  - S will hold sometimes in the future
  - S will hold in the next state
  - S holds until T becomes true
- LTL formulae are interpreted over so-called Kripke structures

Kripke structures

[Kripke 1963]

Let S be a finite set of states and P a finite set of atomic propositions

A System (S, I, R, L) consisting of

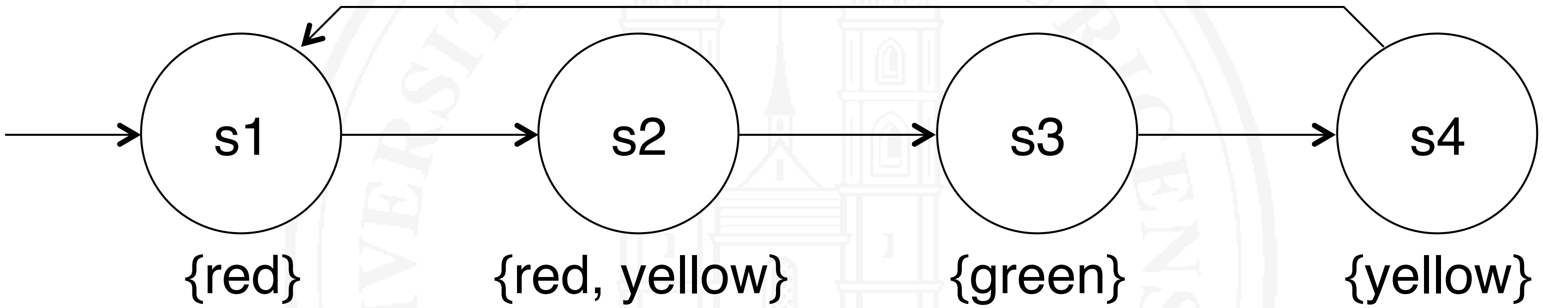
- the set S of states,
- a set I of initial states, $I \subseteq S$
- a transition relation $R \subseteq S \times S$, such that there is no terminal state in S
- a labeling function $L: S \rightarrow IP(P)$, mapping every state $s \in S$ to a subset of propositions which are true in state s

is called a Kripke structure (or Kripke transition system)

$IP(P)$ denotes the power set of P, i.e., the set of all subsets of P

Example: a traffic light

Let P = {off, red, yellow, green}



Exercise: Modify the given Kripke structure such that it also models a yellow flashing light.

Formulae in LTL

- Formulae in LTL are constructed from
  - atomic propositions
  - the Boolean operators ¬, ∧, ∨, →
  - the temporal quantifiers
    - X (next)
    - G (globally)
    - F (finally)
    - U (until)

Alternate Notation:

○ f	for X f
□ f	for G f
◇ f	for F f

- Interpretation: always on a path in a Kripke structure
- Example: For any path s2 → s3 → s4 → ... in our traffic light model, we have: X green, G ¬off, F (red ∧ ¬yellow)

2.1 Motivation

2.2 Temporal Logic

2.3 Principles of Model Checking with LTL

2.4 Model Checking in Practice

Model Checking with LTL

- A Kripke structure *M* satisfies the LTL formula Φ, formally speaking $M \models \Phi$, iff Φ is true for all paths in *M*.
- Now we can precisely define Model Checking with LTL as follows:
  - Let *M* be a model, expressed as a Kripke structure and Φ a formula in LTL that we want to prove
  - Model Checking is an algorithmic procedure for proving $M \models \Phi$
  - If the proof fails, i.e., $M \models \neg\Phi$, holds, the procedure yields a counter example: a concrete path in *M* for which Φ is false

Example: mutual exclusion

We consider the problem of two processes p_1 and p_2 and a critical region c which must not be used by more than one process at every point in time.

Let $c_i \equiv p_i$ uses the critical region c
$t_i \equiv p_i$ tries to enter the critical region c
$n_i \equiv p_i$ does something else

Now we can state the mutual exclusion problem formally as

(1) $G \neg(c_1 \wedge c_2)$

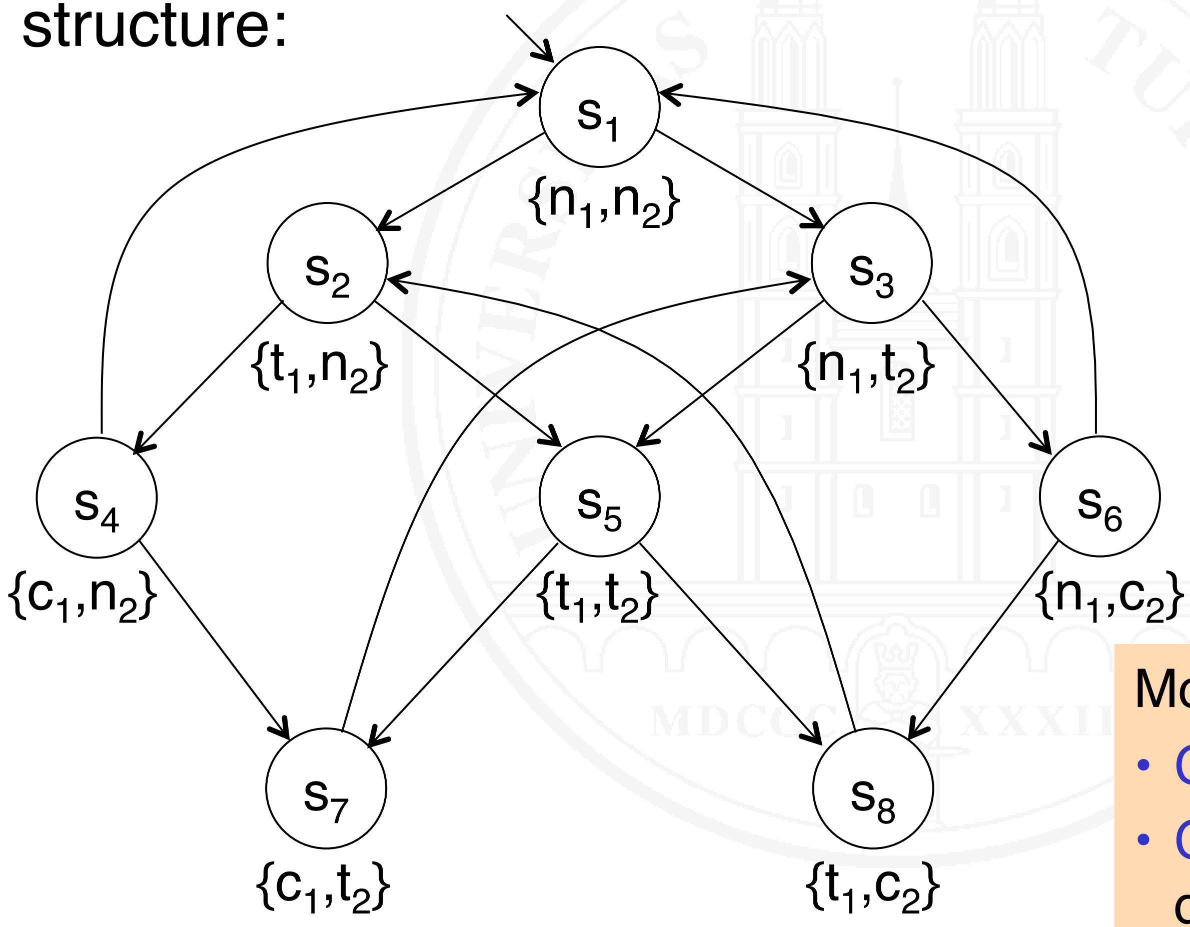
Further, we want the following property to hold:

(2) $G ((t_1 \rightarrow F\, c_1) \wedge (t_2 \rightarrow F\, c_2))$

Explain why we state property (2).
What kind of property is this?

Example: mutual exclusion – 2

Now we model a simple mutual exclusion protocol as a Kripke structure:



Model Checking proves:

- $G \neg(c_1 \wedge c_2)$ holds
- $G ((t_1 \rightarrow F c_1) \wedge (t_2 \rightarrow F c_2))$ does not hold

Example: mutual exclusion – 3

Exercise:
Give a counter example showing that
(2) $G\ ((t_1 \rightarrow F\ c_1) \wedge (t_2 \rightarrow F\ c_2))$
does not hold.

Modify the model such that property (2) holds on all paths.

A simple Model Checking algorithm

Given a model *M* as a Kripke structure and a LTL formula Φ

Parse the formula Φ

WHILE not done, traverse the parse tree in *post-order* sequence

    Take the sub-formula ρ represented by the currently visited node of the parse tree

    Label all nodes of *M* for which ρ is true[1)] with ρ

ENDWHILE

IF all nodes of *M* have been labeled with Φ[2)]
  THEN success
  ELSE fail
ENDIF

1) Due to the order of traversal, all terms needed for evaluating ρ are already present as labels

2) The root of the parse tree represents the full formula Φ

Tractability of Model Checking

- The computational complexity of efficient model checking algorithms is O(n), with n being the number of states
- However, the number of states grows exponentially with the number of variables in the model:
  - n binary variables: 2^n states
  - n variables of m Bit each: 2^{nm} states
- Even with the fastest algorithms, Model Checking is intractable for programs / models of real-world size

➪ Simplification required

Lossless simplification of Model Checking

Representing models and formulae with so-called ordered binary decision diagrams

- allows significantly faster algorithms
- is called symbolic Model Checking
- Still proves $M \models \Phi$ or $M \models \neg\Phi$

Simplification by abstracting the state space

Deliberate simplification of the model (to be performed manually)

- The full domain of a variable is replaced by a few representative values
  (for example, an Integer with 2^{32} states is replaced by a small set of representative values, e.g., {-4, 0, 1, 13}
- A successful Model Checking run is no longer a proof of $M \models \Phi$. It only provides strong evidence for $M \models \Phi$.
- A failing run still proves $M \models \neg\Phi$

➪ Model Checking a simplified state space constitutes a systematic automated test

2.1 Motivation

2.2 Temporal Logic

2.3 Principles of Model Checking with LTL

2.4 Model Checking in Practice

Practical application

- Regularly used in industry for verifying
  - electronic circuit designs
  - safety-critical components of software systems, particularly in avionics
  - security-critical software components, particularly in communication systems
- Models can be created in a notation resembling a programming language; no need to build actual Kripke structures

Tools

Two well-known tools in the public domain

- SPIN [Holzmann 1991, 1997, 2003]
  - Available at: http://spinroot.com
  - Uses LTL
  - Models are written in the Promela language
- SMV [McMillan 1993]
  - Available at: http://www.cs.cmu.edu/~modelcheck/
  - Uses CTL (computation tree logic)

Many other model checking tools available

References

E.M. Clarke, E.A. Emerson (1981). Design and Synthesis of Synchronization Skeletons Using Branching Time Temporal Logic. In: D. Kozen (ed.), *Logics of Programs, Workshop*, Yorktown Heights, NY. Lecture Notes in Computer Science Volume 131. Berlin-Heidelberg: Springer. 52–71.

E.M. Clarke, E.A. Emerson and A.P. Sistla (1986). Automatic Verification of Finite-State Concurrent Systems Using Temporal Logic Specifications. *ACM Transactions on Programming Languages and Systems* **8**(2):244–263.

G.J. Holzmann (1991). *Design and Validation of Computer Protocols*. Englewood Cliffs, N.J.: Prentice Hall.

G.J. Holzmann (1997). The Model Checker SPIN. *IEEE Transactions on Software Engineering* **23**(5):279–295.

G.J. Holzmann (2003). *The Spin Model Checker: Primer and Reference Manual.* Addison-Wesley.

M.R.A. Huth, M.D. Ryan (2000). *Logic in Computer Science: Modelling and Reasoning about Systems*. Cambridge: Cambridge University Press.

S.A. Kripke (1963). Semantic Considerations on Modal Logic. *Acta Philosphica Fennica* **16**:83-94.

L. Lamport (1977). Proving the Correctness of Multiprocess Programs. *IEEE Transactions on Software Engineering* **SE-3**(2):125–143.

K.L. McMillan (1993). *Symbolic Model Checking*. Kluwer Academic Publishers.

References – 2

S. Owicki, L. Lamport (1982). Proving Liveness Properties of Concurrent Programs. *ACM Transactions on Programming Languages and Systems* **4**(3):455–495.

A. Pnueli (1977). The Temporal Logic of Programs. *Proc. 18th IEEE Symposium on the Foundations of Computer Science*, Providence, R.I. 46–57.

J.P. Queille, J. Sifakis (1982). Specification and Verification of Concurrent Systems in CESAR. In: M. Dezani-Ciancaglini, U. Montanari (eds.), *International Symposium on Programming, 5th Colloquium, Turin, April 6-8, 1982. Proceedings*. Lecture Notes in Computer Science vol. 137. Berlin-Heidelberg: Springer. 337–351.