Universität Zürich[UZH]

Institut für Informatik

Martin Glinz

Software-Qualität – Ausgewählte Kapitel

Kapitel 2

Model Checking

2.1 Motivation

- Vor allem bei sicherheitskritischer Software interessiert man sich für formale Beweise, dass
  - ein Programm korrekt ist (d.h. seine Spezifikation erfüllt)
  - ein Modell wichtige geforderte Eigenschaften aufweist

- Im ersten Fall handelt es sich um den klassischen Programmbeweis, d.h. der Beweis, dass ein Programm P die gegebene Spezifikation S erfüllt, d.h. logisch gesehen $P \vdash S$
- Den zweiten Fall bezeichnet man als *Model Checking*
  Sei M das gegebene Modell und Φ die zu beweisende Eigenschaft (typisch formuliert als Formel in temporaler Logik), so ist zu zeigen, dass das Modell M die Formel Φ erfüllt, d.h. $M \models \Phi$

[Clarke and Emerson 1981, Queille and Sifakis 1982]

Formen des Model Checkings

Model Checking wird typisch für zwei Situationen verwendet:

- Partielle Programmverifikation:
  Ist *M* ein Programm und Φ ein Teil der Spezifikation dieses Programms, so bedeutet $M \models \Phi$, dass die Korrektheit von M bezüglich des Teils Φ der Spezifikation bewiesen wird

- Beweis von Eigenschaften einer Spezifikation:
  Ist *M* eine Spezifikation und Φ eine geforderte Eigenschaft, so bedeutet $M \models \Phi$ den Beweis, dass die Spezifikation die Eigenschaft Φ tatsächlich hat

Zu beweisende Eigenschaften

- Bei der Untersuchung geforderter Eigenschaften eines Modells gibt es zwei häufig auftretende Klassen
  - Sicherheitseigenschaften (safety properties) besagen, dass unerwünschte/verbotene/gefährliche Zustände *niemals* erreicht werden.
  - Lebendigkeitseigenschaften (liveness properties) besagen, dass erwünschte Zustände garantiert *irgendwann* erreicht werden.

    [Lamport 1977; Owicki and Lamport 1982]

- Typische Sicherheitseigenschaften: Unmöglichkeit einer Verklemmung (deadlock), garantierter wechselseitiger Ausschluss
- Typische Lebendigkeitseigenschaften: Terminierung eines Programms, Unmöglichkeit von Verhungern (livelock, starvation)

2.2 Exkurs: Temporale Logik

[Pnueli 1977]

- Formeln in Aussagenlogik und Prädikatenlogik: Wahrheitswert zustandsfrei und zeitlich unbegrenzt: keine zeitlichen Aussagen möglich
- Für zustands- und zeitabhängige Aussagen: temporale Logik
- Verschiedene Formen möglich, hier: LTL (Linear-time Temporal Logic)
- Formeln in LTL erlauben Aussagen, dass etwas
  - von nun an immer gilt
  - irgendwann einmal gilt
  - im unmittelbar nächsten Schritt gilt
  - so lange gilt, bis eine andere Aussage gilt
- Zeit wird als eine diskrete Folge von Zuständen modelliert
- Das System, über das zeitliche Aussagen gemacht werden sollen, wird als so genannte Kripke-Struktur modelliert

Kripke-Strukturen

[Kripke 1963]

Gegeben seien eine endliche* Menge S von Zuständen und eine endliche Menge P von atomaren logischen Aussagen

Ein System (S, I, R, B), welches besteht aus

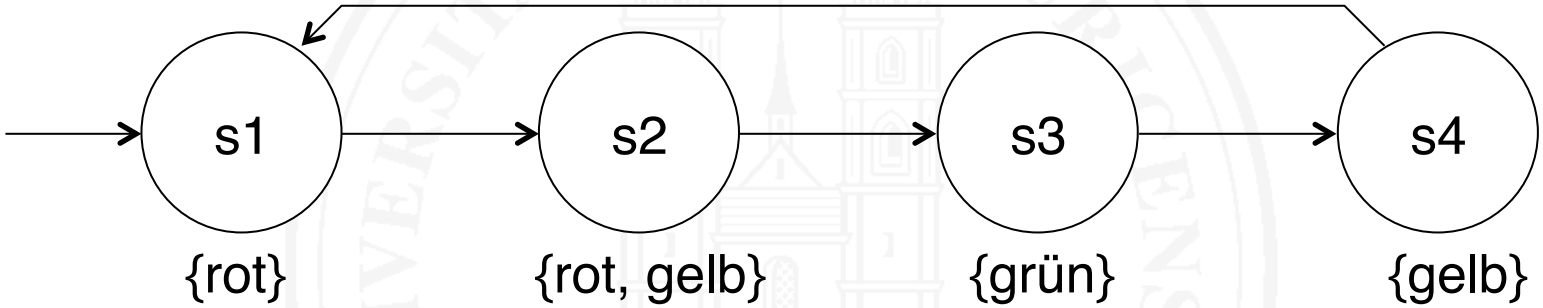
- der Zustandsmenge S,
- einer Menge I von Anfangszuständen, $I \subseteq S$
- einer Zustandsübergangsrelation $R \subseteq S \times S$, so dass es in S keinen Endzustand gibt,
- einer Bewertungsfunktion B, welche jedem Zustand $s \in S$ eine Teilmenge von Aussagen aus der Menge P zuordnet, die im Zustand s wahr sind,

wird Kripke-Struktur (oder Kripke-Transitionssystem) genannt.

*Kripke-Strukturen sind im Prinzip auf abzählbar unendlichen Zustandsmengen definiert. Für Model Checking können wir uns aber auf endliche Mengen beschränken

Beispiel: Modell einer Ampel als Kripke-Struktur

Sei P = {aus, rot, gelb, grün}



Aufgabe: Ergänzen Sie dieses Modell so, dass die Ampel auch gelb blinken kann.

Formeln in LTL

- Eine LTL-Formel ist aufgebaut aus
  - atomaren logischen Aussagen,
  - den aussagenlogischen Operatoren ¬, ∧, ∨, →,
  - den Zeitoperatoren
    - X (next)
    - G (globally)
    - F (finally)
    - U (until)

Alternative Notation:
○ f für X f
□ f für G f
◇ f für F f

- Interpretation: immer auf einem Pfad in einer Kripke-Struktur
- Beispiel: sei s1 → s2 → s3 → s4 → ... ein Pfad im Ampel-System. Für diesen Pfad sind
  - Wahr: X gelb ∨ grün, rot U grün, G ¬aus, F grün
  - Falsch: X grün, G(rot U grün), F aus

Begründen Sie, warum.

2.3 Model Checking mit LTL

- Ein als Kripke-Struktur gegebenes Modell M erfüllt die LTL-Formel Φ, formal $M \models \Phi$, genau dann, wenn Φ für alle Pfade in M wahr ist.
- Damit lässt sich Model Checking mit LTL wie folgt präzise fassen:
  - Sei M ein gegebenes Modell in Form einer Kripke-Struktur
  - Seien ferner die zu beweisenden Eigenschaften als Formel Φ in LTL formuliert
  - Model Checking ist ein algorithmisches Verfahren zum Beweis von $M \models \Phi$. Wenn der Beweis scheitert, d.h es gilt $M \models \neg\Phi$, so liefert das Verfahren ein Gegenbeispiel in Form eines Pfads in M, für den Φ falsch ist.

Beispiel: Wechselseitiger Ausschluss

Gegeben sei das Problem des wechselseitigen Ausschlusses von zwei Prozessen p_1 und p_2. Es gebe einen kritischen Bereich c, in dem sich zu jedem Zeitpunkt höchstens ein Prozess befinden darf.

Sei $c_i \equiv p_i$ befindet sich im kritischen Bereich
$t_i \equiv p_i$ versucht, den kritischen Bereich zu betreten
$n_i \equiv p_i$ tut etwas Anderes

In LTL können wir den wechselseitigen Ausschluss formulieren als:

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

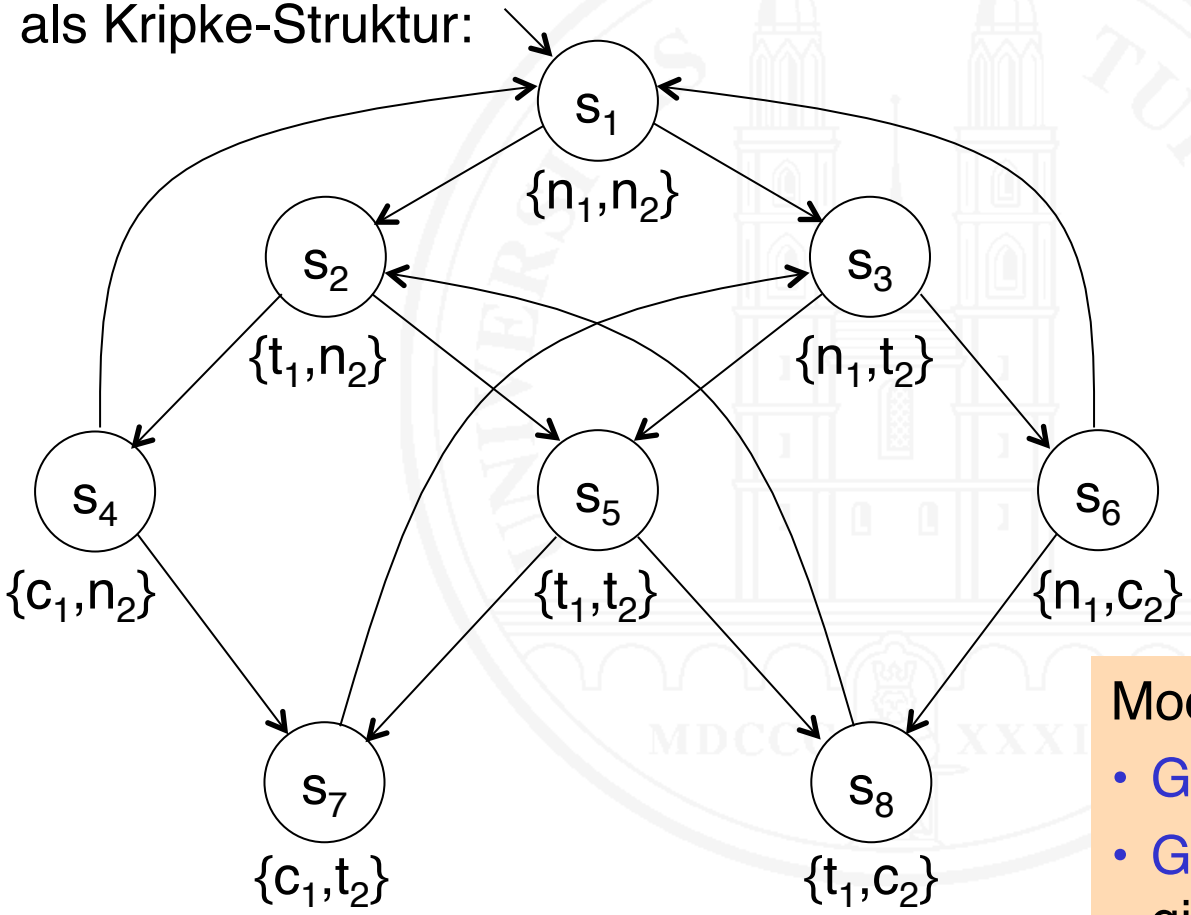
Ferner formulieren wir die Eigenschaft, dass jeder Prozess, welcher versucht, den kritischen Bereich zu betreten, ihn auch irgendwann betritt.

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

Aufgabe: Von welcher Art sind die Eigenschaften (1) und (2)?

Beispiel: Wechselseitiger Ausschluss – 2

Nun modellieren wir ein einfaches wechselseitiges Ausschlussprotokoll als Kripke-Struktur:



Model Checking beweist:

- $G\ \neg(c_1 \wedge c_2)$ gilt
- $G\ ((t_1 \rightarrow F\ c_1) \wedge (t_2 \rightarrow F\ c_2))$ gilt nicht

Beispiel: Wechselseitiger Ausschluss – 3

- Aufgabe:
  - Geben Sie ein Gegenbeispiel an, welches zeigt, dass
    (2) $G((t_1 \rightarrow F\, c_1) \wedge (t_2 \rightarrow F\, c_2))$
    nicht gilt
  - Wie müsste das Modell modifiziert werden, damit Formel (2) auf allen Pfaden wahr ist?

Aufwand für Model Checking

- Die Komplexität effizienter Model Checking Algorithmen ist O(n), wobei n die Anzahl der Zustände ist
- Aber: die Anzahl der Zustände wächst exponentiell mit der Anzahl der Variablen im Modell:
  - n binäre Variablen 2^n Zustände
  - n Variablen à m Bit: 2^{nm} Zustände
- Die Zustandsräume realer Programme bzw. Modelle sind auch für schnelle Algorithmen zu groß

➪ Vereinfachung notwendig

Vereinfachung

- Verlustlose Vereinfachung
  - Repräsentation von Modell und Formeln mit Hilfe so genannter geordneter binärer Entscheidungsdiagramme (ordered binary decision diagrams)
  - beschleunigt die Algorithmen signifikant
  - Wird symbolisches Model Checking genannt
  - Beweist entweder $M \models \Phi$ oder $M \models \neg\Phi$
- Vereinfachung durch Abstraktion
  - Gezielte Vereinfachung des Modells durch Modellierer
  - Beispielsweise nur drei Werte für eine Integer-Variable: -3, 0, 3
  - Erfolg ist kein Beweis mehr, sondern liefert nur noch Evidenz
  - Ein Gegenbeispiel beweist nach wie vor $M \models \neg\Phi$
  - ➪ Wird von einem Beweisverfahren zu einem automatisierten Testverfahren

Model Checking in der Praxis

- Im Schaltkreisentwurf (Hardwarebau)
- Für kritische Teile realer großer Software, v.a. in den Domänen Telekommunikation und Avionik
- Modelle müssen nicht als Kripke-Strukturen dargestellt werden, sondern können in einer programmiersprachen-ähnlichen Notation geschrieben werden
- Bekannte Model Checker sind
  - SPIN [Holzmann 1991, 1997, 2003]
    - Frei verfügbar, große Benutzergemeinde: http://spinroot.com
  - SMV [McMillan 1993]
    - Frei verfügbar: http://www.cs.cmu.edu/~modelcheck/
    - Verwendet CTL (computation tree logic) als Logik

Literatur

Clarke, E.M., 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.

Clarke, E.M., 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.

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

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

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

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

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

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

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

Literatur – 2

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

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

Queille, J.P., 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.