Modellbasiertes Testen

Jens Kosiol

Überblick

- Was ist modellbasiertes Testen und wofür braucht man es?
- Modellbasiertes Testen ermöglicht verschiedene Formen der Testautomatisierung.
  - *Generierung von Testfällen entlang eines Auswahlkriteriums*
  - *Erzeugung von Testorakeln zur Vorhersage von Testergebnissen*

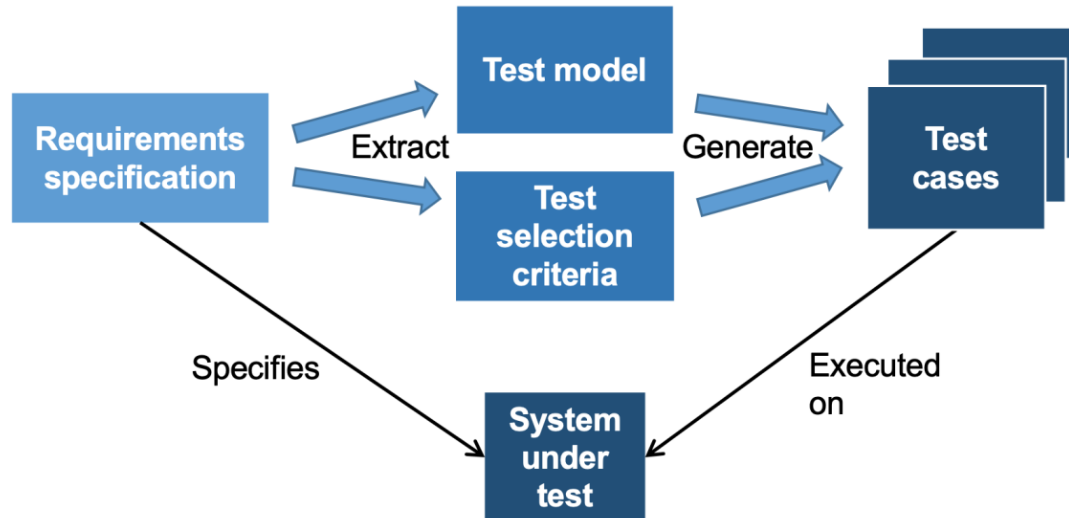
Motivation

- Das Testen ist die wichtigste Methode, die Korrektheit von Softwaresystemen zu prüfen.
- In der Praxis ist das Testen von Software
  - *häufig unstrukturiert*
  - *nicht nachvollziehbar*
- Ein systematischer Ansatz zum Testen:
  - *Wie kann man die Qualität und Vollständigkeit von Testsuites prüfen? → Testabdeckung*
  - *Wie kann man die Korrektheit von Testergebnissen prüfen? → Testorakel*
  - *Wie kann man das Testen automatisieren? → Testgenerierung*

Modellbasiertes Testen

- Was ist modellbasiertes Testen?
  - *Testmodell: Ein Modell des zu testenden Softwaresystems*
  - *Kriterien für die Auswahl repräsentativer Tests*
    - Codeabdeckung
    - Abdeckung von Eingabe- und Umgebungsdaten
    - Abdeckung möglicher Abhängigkeiten funktionaler Einheiten
  - *Automatische Generierung von Testfällen*
  - *Testorakel zur Vorhersage von Testergebnissen*
    - Durch Ausführung des Testmodells

Generelles Vorgehen



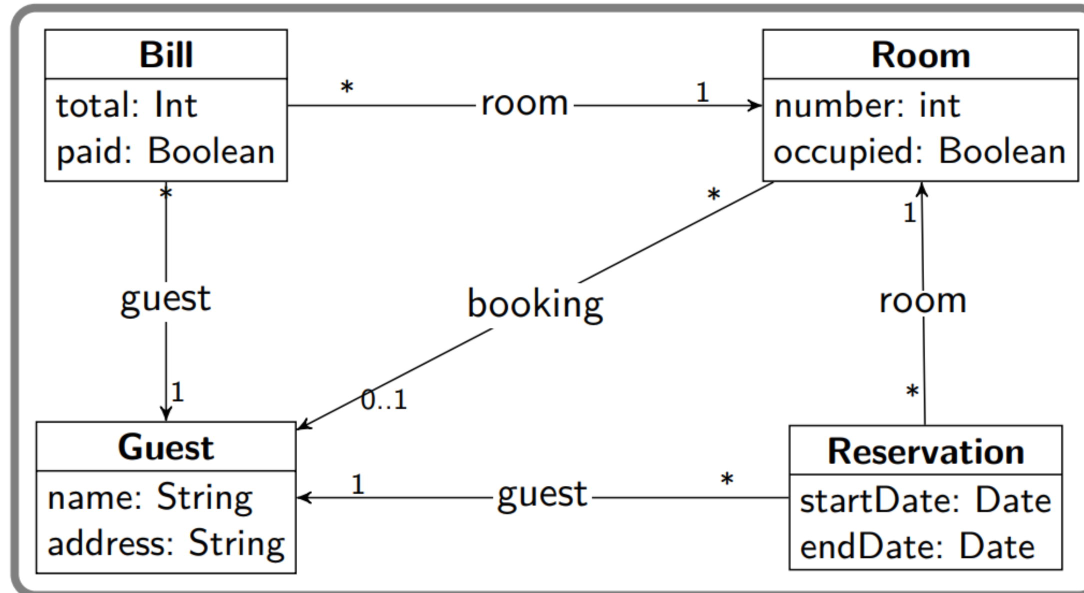
- Extrahiere Testmodell und Auswahlkriterien für Tests aus der Anforderungsspezifikation für das zu testende Softwaresystem (Software under test (SUT))
- Generiere Testfälle für SUT aus dem Testmodell und für die aktuellen Auswahlkriterien

Beispiel: Ein Service für Zimmerverwaltung im Hotel

```
//...
public interface Hotel {
    // ...
    public Guest findGuest(String name);
    public String bookRoom(Room room, Guest guest, s:Date, e:Date);
    public String occupyRoom(Room room, Guest guest, Bill bill);
    public boolean updateBill( Bill bill , int amount);
    public boolean clearBill ( Bill bill ) throws Exception;
    public boolean checkout(Guest guest, Room room, Bill bill );
    // ...
}
```

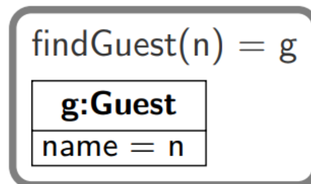
- IHotel ist eine Fassade zum Service.

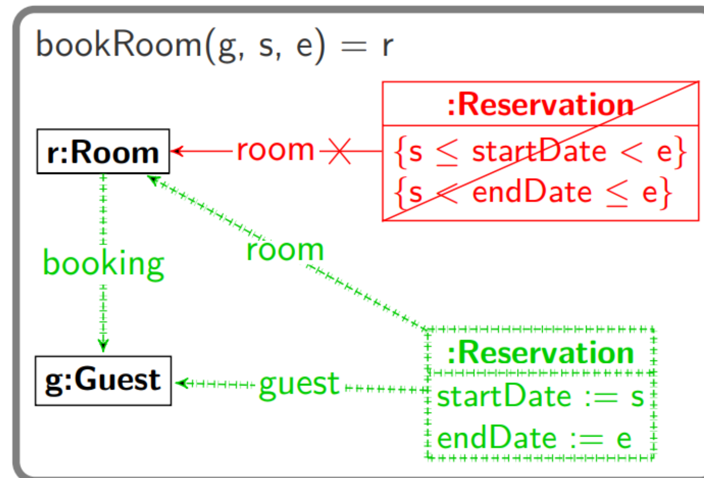
Beispiel: Klassenmodell für einen einfachen Hotel-Service

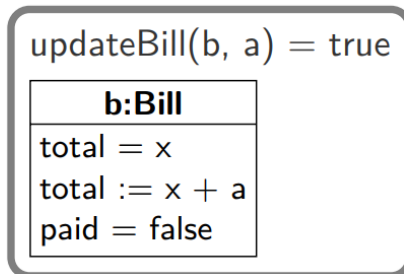


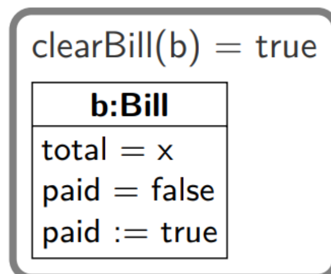
- Das Klassenmodell enthält alle vom Service benötigten Klassen und Felder.

Beispiel: Vor- und Nachbedingungen der Service-Operationen



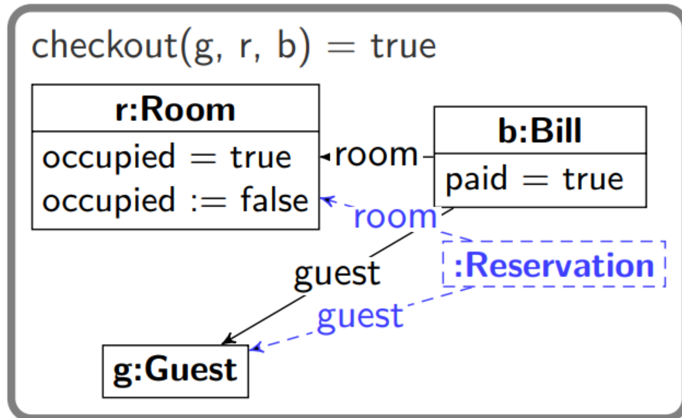


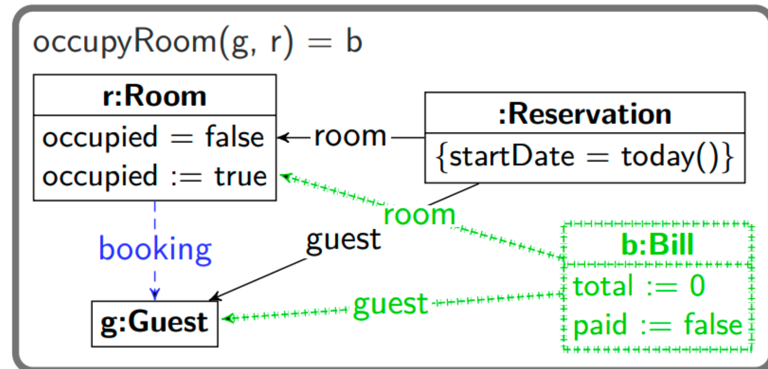


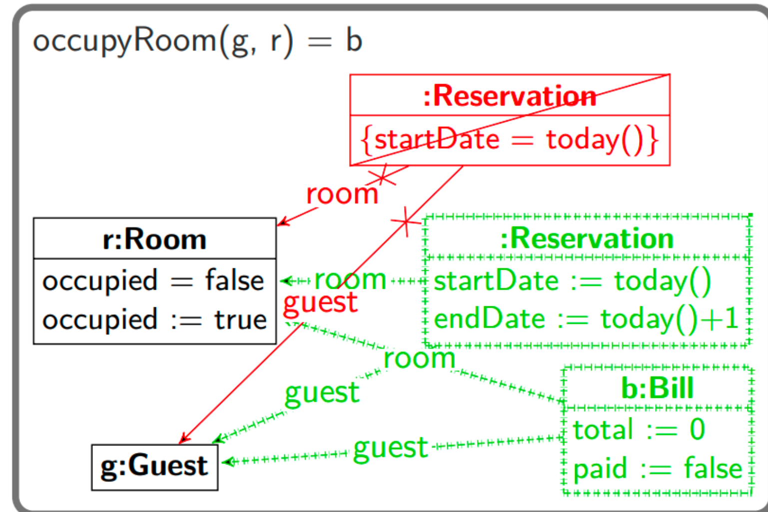


- Modellierung der Vor- und Nachbedingungen durch Visual Contracts (Regeln)

Beispiel: Vor- und Nachbedingungen der Service-Operationen







- Eine Operation kann durch mehrere Visual Contracts modelliert sein. Sie spezifizieren unterschiedliches Verhalten der Operation.

Einfacher Testfall im Modell

- Ein einfacher Testfall im Modell besteht aus
  - *einem initialen Modell,*
  - *einer zu testenden Regel mit belegten Parametern und*
  - *einer Graphformel zur Prüfung der Testbedingung (Test-Constraint).*
- Ein einfacher Testfall entspricht einem Unit-Test.
- Ein einfacher Testfall ist ausführbar im Modell, wenn
  - *die Regel mit ihrer Parameterbelegung auf dem initialen Modell anwendbar ist.*
- Ein einfacher Testfall ist erfolgreich, wenn
  - *der Ergebnisgraph der Regelanwendung den Test-Constraint erfüllt.*

Sequentieller Testfall im Modell

- Ein sequentieller Testfall im Modell besteht aus
  - *einem initialen Modell und*
  - *einer Sequenz von Regeln mit Parameterbelegungen.*
  - *Dient zum Testen von Abhängigkeiten zw. Visual Contracts*
- Ein Testfall ist ausführbar im Modell, wenn
  - *die Regelsequenz auf dem initialen Modell anwendbar ist und zu einer Sequenz von Regelaufrufen führt.*
- Beispiel:

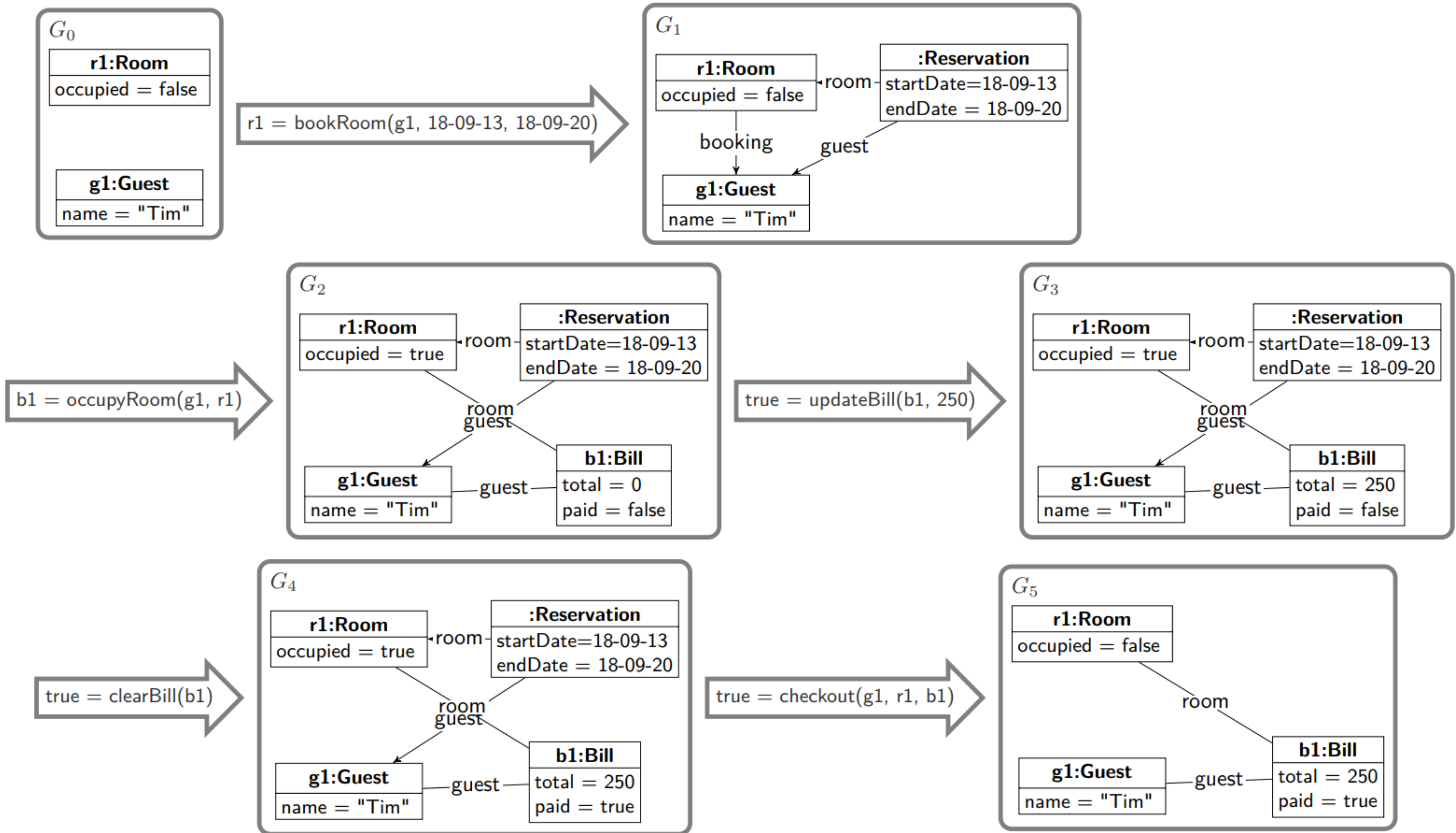
Regelsequenz:

is : r = bookRoom(g, s, e);
b = occupyRoom(g, r);
updateBill(b, a);
clearBill(b);
checkout(g, r, b).

Aufrufsequenz:

cs : r1 = bookRoom(g1, 18-09-13, 18-09-20);
b1 = occupyRoom(g1, r1);
updateBill(b1, 250);
clearBill(b1);
checkout(g1, r1, b1).

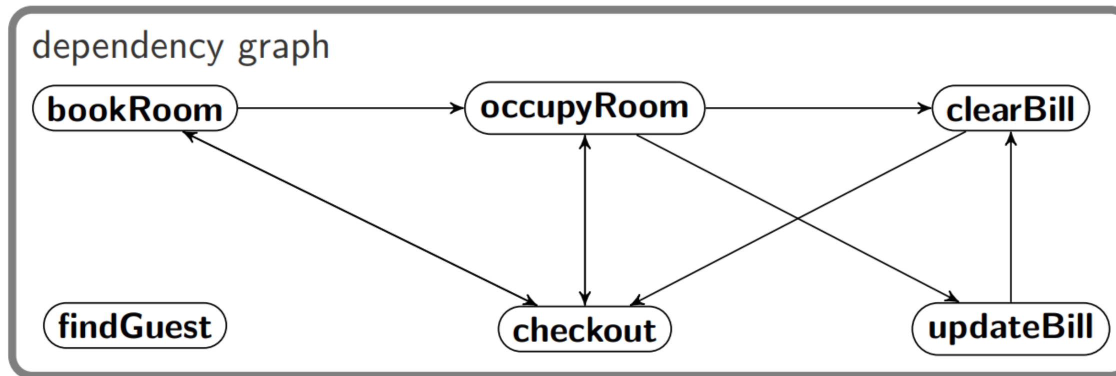
Beispiel: Eine Transformationssequenz



Testauswahlkriterien

- Überdeckung der Vorbedingungen v. Visual Contracts:
  - *Auswahl von Zustandsmodellen und Argumenten so, dass alle Kombinationen von Vorbedingungen geprüft werden.*
  - *Entspricht der Abdeckung von Datenkategorien oder Code*
  - *Testfall im Modell: Anwendung einer Regel mit Argumenten auf ein passendes Zustandsmodell + Test-Constraints*
- Konflikt- und Abhängigkeitsüberdeckung:
  - *Auswahl von Aufrufsequenzen, die alle potentiellen Konflikte/ Abhängigkeiten überdecken*
  - *Testfall im Modell: Anwendung einer Aufrufsequenz auf ein passendes Zustandsmodell → Transformationssequenz*

Beispiel: Abhängigkeitsüberdeckung

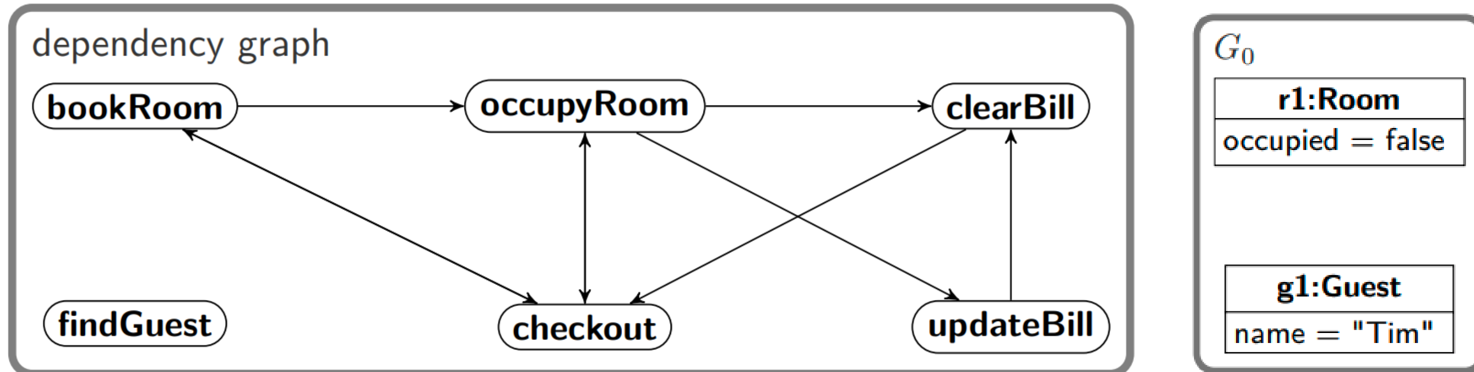


- Basisüberdeckung:
  - *Menge von Aufrufsequenzen, die alle Abhängigkeits-relationen überdecken*
- Differenziertere Überdeckung:
  - *Menge von Aufrufsequenzen, die alle kritischen Paare (bzgl. Abhängigkeit) überdecken*

Algorithmus zur Testfallgenerierung für Abhängigkeitsüberdeckung

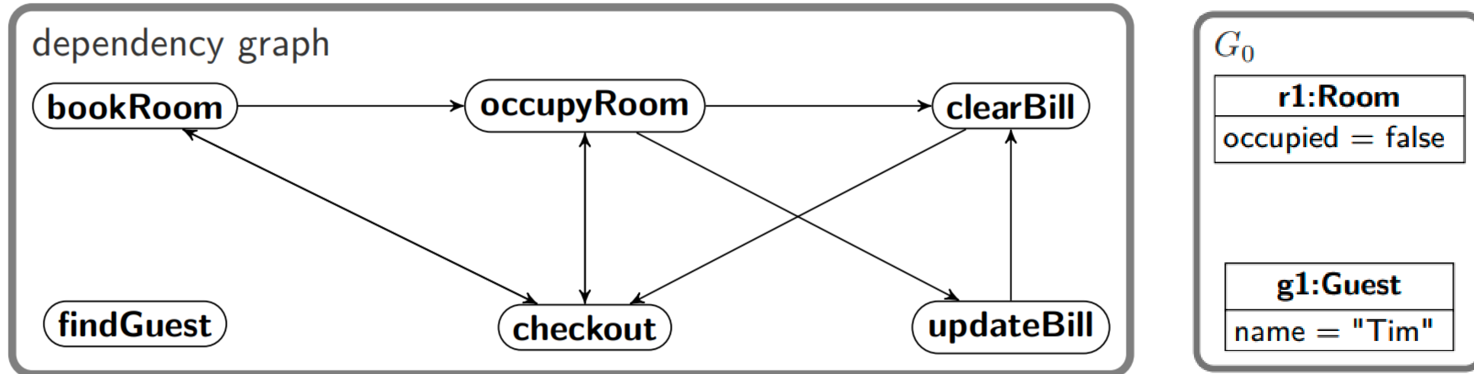
- Geg.: Menge von Regeln R, Abhängigkeitsgraph DG und initiales Modell G_0
- Wähle eine Regel, die auf G_0 anwendbar ist, und bilde Regel-sequenzen, die Abhängigkeiten abdecken.
- Wenn es noch unabgedeckte Abhängigkeiten gibt:
  - *Erweitere Sequenzen, falls Regeln von mehreren anderen abhängig sind. (Z.B: Geg. Abhängigkeiten (p, r) und (q, r): Sequenz … $p; r$ … wird zu … $p; q; r$ … erweitert.)*
  - *Redundante Subsequenzen können gelöscht werden.*
  - *Prüfe, welche Sequenzen auf G_0 anwendbar sind und gib passende Argumente so an, dass die anvisierten Abhängigkeiten auftreten.*
  - *Iteriere durch weitere mögliche Startregeln, bis die Überdeckung vollständig ist oder nicht mehr verbessert werden kann.*
- Ausgabe: Alle berechneten Aufrufsequenzen

Beispiel: Testfallgenerierung zu Abhängigkeitsüberdeckung



- Regeln *findGuest* und *bookRoom* sind auf G_0 anwendbar.
- $S = \{s_1 = \text{bookRoom; occupyRoom}, s_2 = \text{bookRoom; checkout}\}$
- $s_2' = \text{bookRoom; occupyRoom; checkout}$, weil *checkout* auch abhängig von *occupyRoom*
- Da s_1 in s_2' enthalten, brauchen wir nur s_2'
- Weitere nötige Sequenz: $s3 = \text{bookRoom; occupyRoom; updateBill; clearBill; checkout; bookRoom; occupyRoom}$

Beispiel: Testfallgenerierung zur Abhängigkeitsüberdeckung



- *Differenzierte Überdeckung von Abhängigkeiten durch die folgenden Aufrufsequenzen:*

```
r1 = bookRoom(g1, 18-09-13, 18-09-20);
b1 = occupyRoom(g1, r1);
checkout(g1, r1, b1).
```

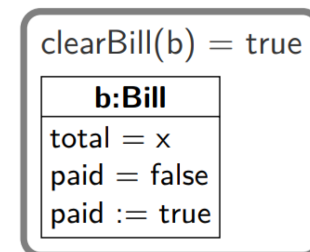
```
r1 = bookRoom(g1, 18-09-13, 18-09-20);
b1 = occupyRoom(g1, r1);
updateBill(b1, 250);
clearBill(b1);
checkout(g1, r1, b1);
bookRoom(g1, 18-09-13, 18-09-20).
```

Testorakel

- Gegeben eine Testspezifikation für eine implementierte Testsuite.
  - *Das Ergebnis einer Ausführung im Testmodell ist ein Testorakel.*
  - *Das Ergebnis eines implementierten Tests soll mit dem entsprechenden Testorakel verglichen werden.*
- Testspezifikation:
  - *Graphtransformationssystem mit Visual Contracts für alle Operationsaufrufe*
  - *Testorakel: Ergebnisgraph einer Transformationssequenz*

Partialität von Testspezifikationen

- Eine Testspezifikation ist üblicherweise unvollständig:
  - *Fehlerfälle sind häufig nicht spezifiziert.*
  - *Detaildaten wurden weggelassen.*
- Beispiel: partiell modelliertes Verhalten
  - *Implementierung von clearBill wirft eine Exception, wenn die Bezahlung nicht durchgeführt werden kann.*
  - *Da die Bezahlung nicht modelliert wird, liefert das Testorakel einen Ergebnisgraphen.*

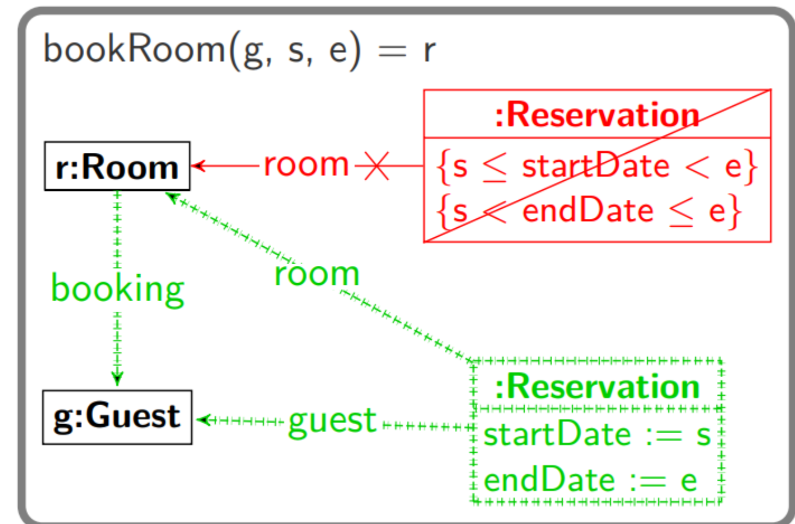


Einsatz von Testorakel

	Testausführung erfolgreich	**Testausführung nicht erfolgreich**
Testorakel erfolgreich	(1) Beide liefern dasselbe Ergebnis. / (2) Beide liefern unter-schiedliche Ergebnisse.	(4) Eine fehlerhafte Implementierung / eine zu schwache Vorbedingung im Visual Contract
Testorakel nicht erfolgreich	(5) Fehler: Dieser Fall darf nicht vorkommen.	(3) Beide liefern Fehler.

Beispiel: Testorakel

- bookRoom:
  - *Fall (1) oder (2), falls z.B. eine Anzahlung für die Reservierung gefordert ist.*
  - *Fall (5), falls die Implement. eine Änderung einer Reservierung durch denselben Gast zulässt.*
  - *Fall (3), falls solch eine Änderung nicht zulässig ist.*



Zusammenfassung

- Modellbasiertes Testen kann zur automatischen Testfallgenerierung und zur Überprüfung von Testergebnissen verwendet werden.
- Testfallgenerierung:
  - *Für ein Graphtransformationssystem und einen Abhängigkeitsgraphen wird eine Menge von Regelsequenzen so generiert, dass alle potentiellen Abhängigkeiten abgedeckt sind.*
- Überprüfung von Testergebnissen:
  - *Eine Graphtransformationssequenz liefert einen Ergebnisgraphen, der mit dem entsprechenden implementierten Test verglichen wird.*