Forschungszentrum Informatik
an der
Universität Karlsruhe
Forschungsbereich Softwaretechnik

Protocol analysis in distributed applications with model checking of timed automata

Cand. Inform. Piotr Serwa

Forschungszentrum Informatik
Universität Karlsruhe (TH)
Forschungsbereich Softwaretechnik
Prof. Dr. Walter F. Tichy
Dipl. Inform. Marc Schanne

Inhalt

1. Einführung, Analyse, State-of-Art
2. Umsetzung
3. Anwendungsfallgetriebene Spezifikation
4. Validierungsbeispiel
5. Zusammenfassung und Bewertung

Ziel der Diplomarbeit und Plan

Ziel – Spezifieren eine Lösung die ermöglicht:

1. **effizient, einfach und möglichst viel Designdefekte identifizieren**
2. **in einer frühen Phase des Entwicklungszyklus**
3. **bei verteilten Java Anwendungen**

Plan um dieses Ziel zu erreichen:

1. **Ziel-orientierte und durch den Entwickler kontrollierte Extraktion**
2. **Spezifikation einer Java-Annotationssprache und Definition von Transformationsregeln**
3. **Extraktion der Programmeigenschaften und des Modells aus dem annotierten *Programmcode* mit einem Protokoll-Verifier-Werkzeug (PV-Tool)**
4. **Verifikation des Modells gegen Programmeigenschaften mit UPPAAL**
5. **Simulation des Modells um das Programm zu verstehen**

Ergebnisse der Diplomarbeit

Ergebnisse:

1. **Anwendungsfallgetriebene Spezifikation:**
   a. Annotationssprache für Java
   b. Regeln der Abstraktion (Java + Annotationen) → (Modell + Programmeigenschaften)
2. **Proof-of-concept Design und Implementierung**
3. **Validierung durch eine existierende Ziel-Anwendung**

Kontext der Diplomarbeit:

1. **Durchgeführt im Kontext des EU-Forschungsprojekt HIJA – Weiterbenutzung der Ergebnisse**
2. **PV-Tool als Open-Source-Projekt (sf.net)**

Existierende Lösungen

Design-basierte Formale Werkzeuge: SPIN, UPPAAL, SMV

1. Modell: graphisch oder textuell, mit oder ohne Zeitangaben
2. Programmeigenschaften: Linear-Logic oder Branching-Logic
3. Werkzeug: GUI oder Kommandozeile

Hautprobleme:

1. Notwendigkeit von Experten
2. Nicht-existierendes Design
3. Synchronisierung zwischen Design, Modell, Code

Modelextraktionswerkzeuge für Java: JavaPath, Bandera

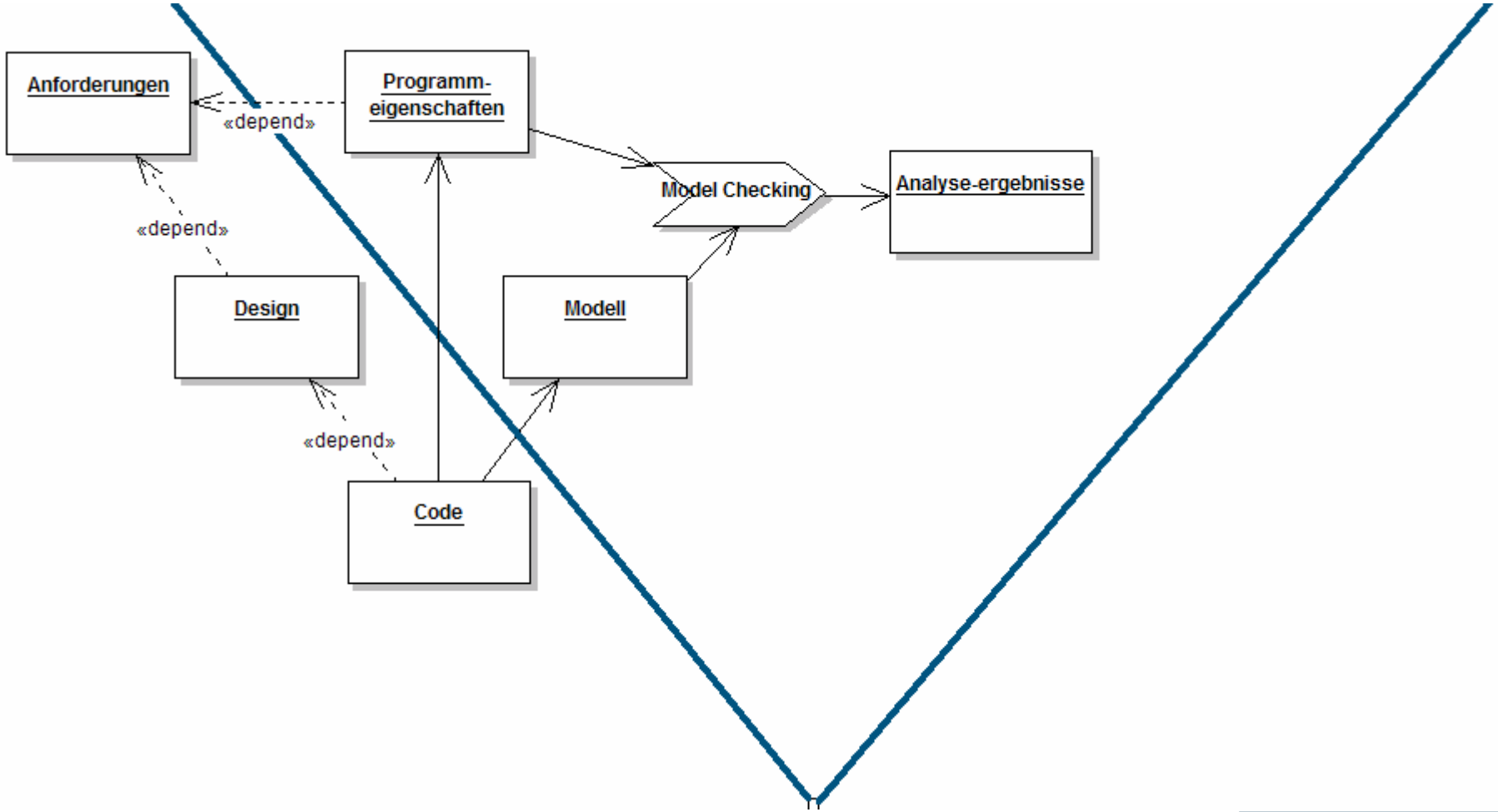
Hautprobleme:

1. keine Skalierbarkeit („state explosion“), große Modelle als Ausgang, unmöglich zu berechnen
2. gehen zu tief, verlorene Verständlichkeit für Java-Entwickler

Einführung

1. **Ziel-orientierte und durch den Entwickler kontrollierte Extraktion:**
   a. Entwickler definiert die Programmeigenschaften
   b. Entwickler kontrolliert PV-Tool: WELCHE Systemteile sollen extrahiert und und WIE abstrahiert werden
   c. PV-Tool extrahiert das Modell und die Programmeigenschaften
   d. UPPAAL überprüft Modell gegen Programmeigenschaften
2. **Skalierbar: nur relevante Information werden genommen und abstrahiert**
3. **Standard-konform:**
   a. Eingaben: Java, Javadoc, JML
   b. Ausgaben: UPPAAL-Sprache
   c. Modellprüfer: UPPAAL
4. **Einfach für Java-Entwickler (einfache Syntax und Regeln)**
5. **Einfach zu pflegen: eine Quelle für Informationen (Programmcode)**

Integration in V-Modell-Entwicklugsprozess



Überblick

1. **Die Spezifikation der Abstraktionsregeln ist der Kern der Diplomarbeit**
2. **Kategorien für Anwendungsfälle:**
   a) Basisstruktur (Prozesse, Schleifen, Methoden, Klassen)
   b) Kommunikation (RMI, Nachrichtenversand)
   c) Programmeigenschaften (generisch, benutzerspezifisch)
3. **Hauptelemente für *Spezifikation* aus Anwendungsfällen:**
   a) Funktionalität
   b) Eingaben: Java Code und Annotationen
   c) Ausgaben: Modellteile, Programmeingenschaften
   d) Transformationsregeln/Logik
4. **26 Annotation-Keys**

Eingabe

Property	Description		
Annotation	@rmiLookup		
Location	In front of Naming.lookup()		
Exclusivity	No (before and after there can be transitions, declarations.		
Multiplicity	There can be only one annotation per Java class declaration.		
Key	**Meaning**	**Value constraints**	**Assumed value if not specified**
expression	Class name and initial object reference	Naming like for classes and object references	Java expression For example: Cricle c = Naming.lookup(s) has default expression Circle c

RMI Lookup:

```
2        /**
3         * @rmiLookup expression = "Account account2",
4         * serverObject = "$(i)";
5         */
6        account2 = (Account) Naming.lookup("//127.0.0.1/" + i);
```

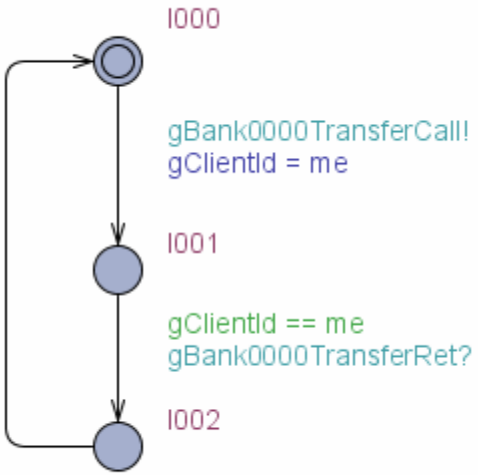
Property	Description		
Annotation	@callNew		
Location	In front of invocation/call of a constructor		
Exclusivity	No (before and after there can be transitions, declarations.		
Multiplicity	There can be only one annotation per Java class declaration.		
Key	**Meaning**	**Value constraints**	**Assumed value if not specified**
parameters	List of parameters of the constructor	Naming like for classes and object references	Parameters passed to java method
expression	Class name and initial object reference	Naming like for classes and object references	Java expression For example: Cricle c = new Cicle() Has default expression Circle c

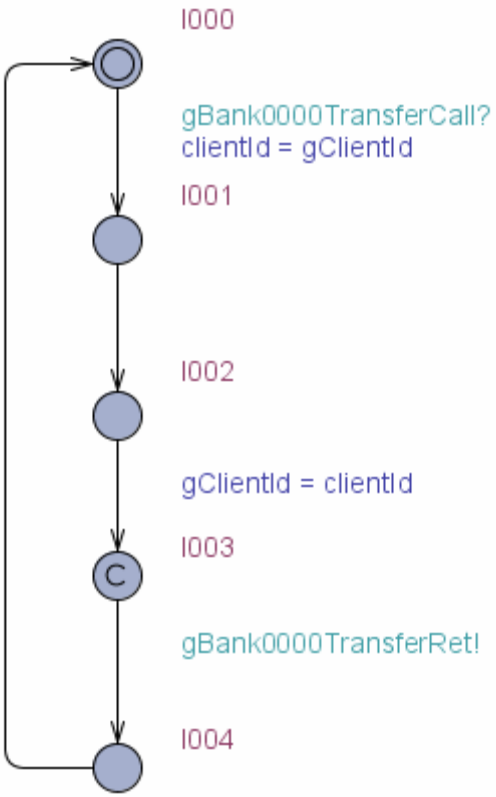
Konstruktor-Aufruf:

```
        /**
         *
         * @callNew expression = "Circle circle", params = "10, initialCenter";
         */
        Circle circle = new Circle(10, initialCenter);
```

Ausgabe

RMI Aufruf auf Dienstnehmer (links) und Dienstgeber (rechts):





Einführung

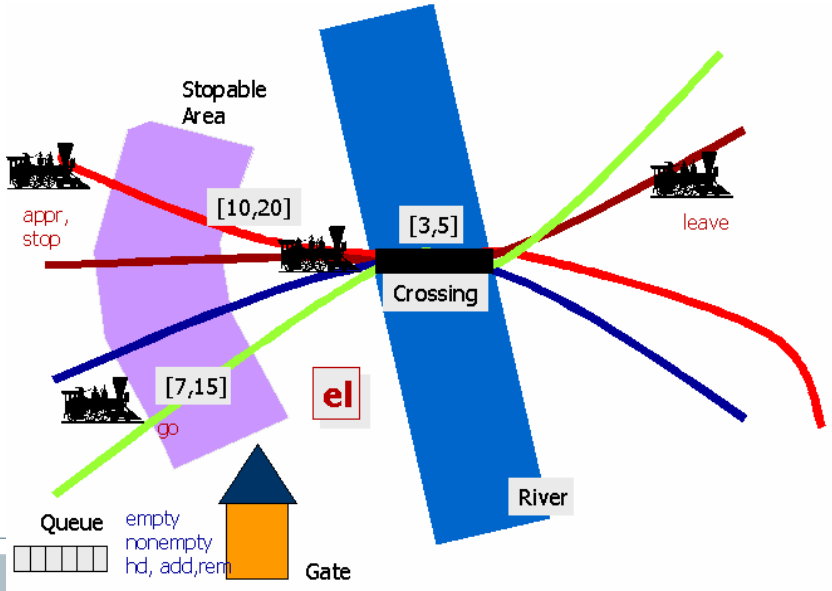
Ziel: Überprüfung und Demonstration der Nützlichkeit und Vollständigkeit des Werkzeugs und der Spezifikation

Validierungsbeispiel basiert auf “Standard-Beispiel” von UPPAAL Bahnschränke (Train-Gate):

1. **Design teilweise geändert**
2. **Implementierung in Java, voll respektierend “Java-Philosophie”**
3. **Model und Programmeigenschaften extrahiert aus Java:**
   1. Teilweise durch das Werkzeug (Kern-Funktionen sind implementiert)
   2. Meist per Hand (nach den PV-Tool-Extraktionsregeln)

Anwendungslogik

1. **Züge benutzen gemeinsame Ressource (Brücke), durch eine Weiche („Gate“) kontrolliert**
2. **Züge und Weiche haben ein Java-basiertes Kontrollsystem, kommunizieren über Nachrichten (keine Semaphore)**
3. **Wenn der Zug ankommt, fragt er um Erlaubnis der Weiche:**
   1. Wenn Erlaubnis in 10 s ankommt, passiert Zug ohne zu stoppen
   2. Ansonsten, stoppt der Zug und wartet auf Erlaubnis
4. **Nach dem Kreuzen, benachrichtigt der Zug die Weiche**



Verwendete Funktionalität

1. Programmeigenschaften
2. Klassen, Objekte, Methoden, Attribute
3. Schleifen und Bedingungen
3. Nachrichtenbasierte Kommunikation
4. Bibliotheken

Beispiel einer Programmeigenschaft

Der Zug, der die Weiche über seine Durchfahrt benachrichtigt, soll derjenige sein, der (als letzter) von der Weiche eine Erlaubnis bekommen hat

Teile von Programmcode der Weiche:

1. **Nimm nächsten wartenden Zug (Z. 90)**
2. **Schicke Erlaubnis (Z. 97)**
3. **Warte auf Durchfahrtbenachrichtigung und überprüfe (Z. 103-106)**

```
90      crossingTrainId = waitingTrains.get(0);

97      sendPermissionToCross(crossingTrainId);

103       * @assert expression =
104       * "A[] $(here) --> leavingTrainId == crossingTrainId";
105       */
106     leavingTrainId = waitForDepartureNotifiation();
```

Beispiel für Modell-Annotationen

Beispiel 1: 1-zu-1 Extraktion einer Deklaration:

```
58          /** @declaration; */
59          int crossingTrainId;
```

Beispiel 2: Nachrichtenversand

```
92              /**
93               * Now, we allow the first train to cross.
94               * @send message = "gCrossAllow", senderAddress = "me",
95               * receiverAddress = "crossingTrainId";
96               */
97              sendPermissionToCross(crossingTrainId);
```

Beispiel von PV-Tool Bibliotheken

Beispiel 3: UPPAAL-Algorithmen-Bibliothek

- Algorithmen, Datenstrukturen (wie FIFO, ArrayList) müssen nicht modelliert werden
- Sie werden jedoch gebraucht (für Liste wartender Züge)

```
50        /**
51         * @declarationLibrary class = "uppaal.util.ArrayListInteger",
52         * listName = "waitingTrains", size = 3, elementMin = 0, elementMax = 3;
53         */
54        List<Integer> waitingTrains = new ArrayList();
```

```
88            * @transition update = "waitingTrainsGet(0)";
89            */
90           crossingTrainId = waitingTrains.get(0);
```

Ausgabe

1. **UPPAAL überprüft Modell gegen Programmeigenschaften und findet Fehler (z.B. Deadlock)**
2. **Benutzer analysiert die fehlerhafte Sequenzen, um Defekte im Programm zu finden**

Ergebnis

In Java-Implementierung sind Defekte gefunden worden (ca 6 Stück)

1. **Design-Defekte**
2. **Sowie low-level Code-Defekte**

Bestätigung der Anwendungsfallsgetriebenen Spezifikation

1. **Gültigkeit einzelner Regeln**

Brauchbarkeit des allgemeinen Verfahrens

1. **Einfach zu benutzen**
2. **Nur wenige Annotationen notwendig, Nutzen von Standardwerten**

Ergebnisse

1. **Die Validierungsbeispiel zeigt, dass das Verfahren erfolgreich funktioniert**
2. **Um gute Programmeigenschaften zu definieren hilft Erfahrung**
3. **Hilfe für bessere Dokumentation / Verständnis der Anwendung**
4. **Verfahren ist mit Zeit und Kosten verbunden**
5. **Frühes Finden von Fehlern spart Kosten**

Ende der Präsentation

Fragen

Diskussion