Vorlesung vom 20.06.06: Werkzeugunterstützung für JML

Wo sind wir?

- Hoare-Kalkül
- Jenseits von Hoare:
  - Einführung
  - Reasoning in JML — Ausnahmen
  - *Werkzeugunterstützung*
- Korrekte Programmentwicklung

Werkzeuge für JML

- Zur *Laufzeit*:
  - *Prüfung* von *Zusicherungen*
  - Generierung von *Testfällen*
- Zur *Übersetzungszeit*:
  - *Statische Überprüfung*,
  - *Animation*,
  - *Theorembeweisen*

Der JML-Übersetzer `jmlc`

- Übersetzt *JML-Programm* in *Bytecode* (spezielle JVM)
- Zur Laufzeit *Überprüfung* der *Zusicherungen*, *Invarianten*, *Vor/-Nachbedingungen*
- Verletzte Bedinungen: *Fehlermeldung*
- Nur für *ausführbare Untermenge* von JML
  - *Konservativ* (meldet nur *echte* Fehler)
- Erzeugt *Fehlermeldung* und *backtrace*

Unit Testing: `jmlunit` = JML + JUnit

- `jmlunit` generiert *Unit Tests* aus JML
- Tests prüfen *Zusicherungen*, *Vor-/Nachbedingungen*, *Invarianten* für *Methode*
- Beispiel:
  - `Purse.java`
  - `jmlunit` findet Ausnahme in `debit`: Ausgabe
  - Verstärkte Vorbedingung behebt Fehler:
    `amount >= 0 && amount <= balance`

Unit Testing: `jmlunit` = JML + JUnit

- Vorteile:
  - Prüft auch *Spezifikation*
  - *Abdeckung* der Zusicherungen (dazu nötig: vollständige Spezifikation)
- *Nachteile:*
  - Kann nur Fehler in *einzelnen* Methoden finden
  - Testdaten müssen gut *ausgewählt* werden
  - Testdaten werden nicht aus *Spezifikation generiert*

Extended Static Checking: ESC/Java

- *Vollautomatische* Programmverifikation:
  - Einfache Zusicherungen
  - Dereferenzierung von `null`
  - Feldzugriff außerhalb der Indexgrenze
  - *Cast* auf unzulässigen Typ
- Weder *vollständig* noch *konsistent*:
  - Findet nicht *alle Fehler*
  - Meldet *nicht-existente Fehler*
  - Pragmatik: es findet *genug richtige Fehler!*
- Benutzt *externen*, automatischen Beweiser
- Beispiel: Ausgabe

Exkurs: Abstrakte Interpretation

- *Analyse* durch *Interpretation* in *einfacherem Wertebereich*
- Beispiel:
  - $\mathbb{Z}$ interpretiert als $\mathbb{Z}^{\#} \stackrel{def}{=} \{plus, zerominus\}$
  - Abstrakte Version $*^{\#}$ der Multiplikation $*$
- *Sicherheitskriterium*:

$$\begin{tikzcd} D \arrow[r, dash, "f"] \arrow[d, dash, "abs"'] \arrow[dr, phantom, "\supseteq"] & D \arrow[d, dash, "abs"] \\ D^{\#} \arrow[r, dash, "f^{\#}"'] & D^{\#} \end{tikzcd}$$

- Anwendung: *Striktheitsanalyse*

LOOP

- Echtes *Theorembeweisen*
- *Coalgebraische* Semantik (siehe letzte Woche)
- *Flache Einbettung* von Java in HOL
  - Garantierte *Korrektheit*
- LOOP:
  - Liest JML (`Java`)
  - Erzeugt *Beweisverpflichtungen* in *Isabelle/PVS*
- Nicht frei verfügbar — *Forschungsprototyp*

JACK

- Schwächste *Vorbedingungen* (wp) in *Java* für *Java*
- Implementation: *Eclipse* plug-in
- Ziel: *specification for the working programmer*
- Keine Formalisierung des wp-Kalküls, aber soll *vollständig* und *konsistent* sein
- Erzeugt *Beweisverpflichtungen* für *automatische Theorembeweiser*
  - Atelier B, Simplify, PVS
- Nicht frei verfügbar — *lizenziert*

WHY: multi-prover program verification

- `WHY`: schwächste Vorbedingung
- Sprach- und Beweiserunabhänging
- Liest *annotierten Quelltext*, produziert *Beweisverpflichtungen*
- Eingabesprachen: C, ML
- Beweiser: Coq, PVS, Isabelle, HOL Light, Mizar, Simplify, haRVey
- Zusätzliches Werkzeug für Java: *Krakatoa*
  - Liest JML, erzeugt Eingabe für `model` und Eingabe für WHY
- Frei verfügbar (GPL)

Zusammenfassung

- `jmlc`: *Prüfung* von *Zusicherungen* zur *Laufzeit*
- `jmlunit`: Generierung von *Testfällen*
- `ESC/Java2`: *Statische Überprüfung*; vollautomatisch, weder *vollständig* noch *konsistent*
- `LOOP`: Flache Einbettung der Semantik, Isabelle und PVS als Beweiser, interaktiv
- `JACK`: wp-Kalkül, verschiedene Beweiser (vollautomatisch)
- `WHY`: verschiedene Eingabesprachen (C, ML, JML `Krakatoa`), verschiedene Beweiser, interaktiv
- Altes *Dilemma*: *Automatisierung* vs. *Mächtigkeit*
- Nächste Woche: wie *funktioniert* `WHY`?