Karlsruher Institut für Technologie

Institut für Theoretische Informatik
Logik und Formale Methoden
Prof. Dr. P. H. Schmitt, C. Engel, D. Faragó, R. Krenický, M. Ulbrich, B. Weiß

02.12.2009

Praktikum
Formale Entwicklung objektorientierter Software
Übungsblatt 4: RAC und JMLUnit

Aufgabe 9 — Einstiegsaufgabe zu RAC

Der JML Runtime Assertion Checker (RAC) erzeugt mit Hilfe des Compilers `jmlc` Bytecode, der mit Assertions für die JML-Spezifikationen angereichert ist. Der Bytecode kann dann am einfachsten mit `jmlrac` ausgeführt werden.

(a) Benutzen Sie RAC für Ihr Gegenbeispiel aus Aufgabe 7 (oder ein beliebiges Beispiel, dass sich von der `main`-Methode der Aufgabe 10 unterscheidet).

(b) Zeigt die Fehlermeldung von RAC, dass die Implementierung in `Klausuren.java` zu Aufgabe 7 (a) ihre Spezifikation nicht erfüllt? Erläutern Sie die Ausgabe von RAC. Eine grobe Beschreibung von RAC finden Sie im zweiten Kapitel des JMLUnit-Papers auf der Praktikums-Webseite.

Beachten Sie, dass der RAC von JML5.5 Zeilennummern in den Fehlermeldungen leider häufig falsch angibt und Spezifikationen mit bestimmten Allquantoren (z.B. Summe und Produkt) ignoriert. Kommentieren Sie die JML-Spezifikationsfälle mit diesen Allquantoren aus, damit RAC nicht den komplette JML-Kontrakt ignoriert.

Aufgabe 10 — Arbeitsprozess mit RAC

Häufig sind in der Implementierung als auch in der Spezifikation mehrere Fehler vorhanden, so dass mehrere Zyklen bis zum gewünschten Ergebnis nötig sind. Finden Sie Fehler mit RAC und beseitigen Sie diese. Benutzen Sie dazu die Methode `main` der Klasse `Klausuren.java` von der Praktikums-Webseite für diese Aufgabe zusammen mit Ihrer Lösung aus Aufgabe 7 (a). Am Ende soll die `main`-Methode fehlerfrei mit RAC durchläuft.

Bei der Fehlerbehebung sollen sowohl Implementierungs- als auch Spezifikations-Fehler korrigiert und ggf. JML-Spezifikationen ergänzt werden, z.B. `nullable` statt der impliziten Spezifikation `non_null`. Implementierungen und Spezifikationen dürfen aber nicht so stark vereinfacht werden, dass die ursprüngliche Intention nicht mehr gilt.

Bitte geben Sie folgendes ab:

- Ihre korrigierte Datei `Klausuren.java` mit der neuen `main`-Methode.
- Eine knappe Beschreibung Ihrer Vorgehensweise mit den aufgetretenen RAC-Fehlermeldungen und Ihren entsprechenden Korrekturen.

Aufgabe 11 — Einstiegsaufgabe zu JMLUnit

JMLUnit vereint die Werkzeuge RAC und junit: Die Testdaten und Testabwicklung übernimmt junit. RAC erzeugt die Testorakel, d.h. die Überprüfung, ob ein Test erfolgreich oder fehlerhaft (oder nutzlos wegen geworfener JMLEntryPreconditionError) ist. Der größte Vorteil von JMLUnit ist eine bessere Handhabung und Pflege, insbesondere der Konsistenz zwischen Spezifikation, Quelltext und Tests.

(a) Nutzen Sie JMLUnit, um eine Klasse aus Aufgabe 7 zu testen, z.B. `Student`:

Speichern Sie einfachheitshalber alle Klassen aus `Klausuren.java` in einzelnen, gleichnamigen Dateien. Kompilieren Sie nun die zu testende Klasse (in den folgenden Beispielen `Student`) mit `jmlc Student.java`.

Danach erzeugen Sie mit `jmlunit Student.java` zwei Dateien:

- `Student_JML_Test.java`, die Testfixture und Testorakel enthält
- `Student_JML_TestData.java` (falls diese Datei noch nicht existiert), die Generatoren für die Testdaten und Testfälle. enthält.

Nun können Sie die Datei `Student_JML_TestDaten.java` editieren, um für alle Typen einige Werte hinzuzufügen. Erklärungen finden Sie als Kommentare in der Datei selbst (z.B. `//replace this comment with test data if desired`). Details sind im JMLUnit-Paper (siehe Praktikums-Webseite) beschrieben, insbesondere Seite 12 und Kapitel 5.1 könnten hilfreich sein.

Führen Sie danach `javac Student_JML_Test*.java` aus (bei Rechnern nicht aus dem i12-Poolraum ggf. `CLASSPATH` erweitern, Beispiel siehe Praktikums-Webseite).

Nun können Sie die Tests mit `jmlrac Student_JML_Test` (oder mit `jml-junit Student_JML_Test`) ausführen. Wieviele Tests sind erfolgreich, wieviele fehlerhaft?

(b) Die in Aufgabe 10 gefundenen Fehler sollen nun auch mit JMLUnit entdeckt werden. Führen Sie dazu die Schritte aus (a) für alle notwendigen Klassen durch und erweitern Sie die Test-Generatoren hinreichend. Schreiben Sie keine Testmethoden (`test`X) selbst.

Überprüfen Sie, dass die resultierenden Tests alle Fehler aus Aufgabe 10 entdecken, indem sie diese jeweils einzeln wieder einbauen.

Aufgabe 12 — Testabdeckung mit JMLUnit

Benutzen Sie nun JMLUnit um Ihre Aufgabe 8 zu testen:

(a) Erweitern Sie die `JML_TestDaten`-Klassen so, dass möglichst viele Fälle möglichst vieler Requirements aus Aufgabe 8 abgedeckt werden. Sie dürfen nun auch eigene Testmethoden (`test`X) schreiben. Nennen Sie den Grund, falls Sie manche Requirements nicht abgedeckt haben.

(b) Beheben Sie alle Fehler, die Sie entdecken. Wie viele waren es? Wie viele Tests werden nun erfolgreich ausgeführt?

Abgabe bis Mittwoch, 09.12.
Die Abgabe erfolgt über die Praktikums-Webseite. Es braucht pro Gruppe und Aufgabe nur *eine* Lösung abgegeben werden. Bitte dokumentieren Sie Ihre Lösungen ausreichend und seien Sie darauf vorbereitet, sie auf Nachfrage zu erklären.

Praktikums-Webseite: `http://lfm.iti.kit.edu/keyprakt0910.php`

Christian Engel:	R. 106,	Tel. 608-4338,	E-Mail: `engelc@ira.uka.de`
David Faragó:	R. 308,	Tel. 608-7322,	E-Mail: `farago@ira.uka.de`
Roman Krenický:			E-Mail: `krenicky@ira.uka.de`
Mattias Ulbrich:	R. 106,	Tel. 608-4338,	E-Mail: `mulbrich@ira.uka.de`
Benjamin Weiß:	R. 309,	Tel. 608-4324,	E-Mail: `bweiss@ira.uka.de`