Testing object Interactions

Grüner, A.

Summary

In most of today's software development projects, testing is still the only feasible instrument for assuring the quality of a software product. Rigorous testing is necessary during all stages of the software development life cycle. While system and acceptance tests deal with the complete system, unit testing aims at the smallest building blocks of the software. Not only due to the growing popularity of agile software development methodologies, the responsibility for unit tests is more and more shifted from software testers to software developers. Besides the entailed advantages, like increasing quality awareness and immediate feedback to the programmers, this development also demands for novel testing frameworks accounting for the different qualifications and expectations of software developers. At the same time, the rampant use of object-oriented programming languages equally necessitates a change regarding the unit testing approaches. Specifically, instead of traditional *state-based* tests in terms of an input-output comparison, in an object-oriented context it is more useful to execute *interaction-* or *behavior-based* tests considering the *sequence of interactions* between the unit under test and its environment.

In this thesis we provide a unit testing approach for multi-purposes object-oriented programming languages in the style of *Java* and C♯. To meet the above indicated requirements our approach includes the definition of a test specification language which results from extending the programming language with new designated specification constructs. In this way, the software developer does not need to learn a completely new language. At the same time, adding new constructs allows to increase the abstraction of the language regarding the specification of interaction-based tests. In order to execute a specified test, programming language code is automatically generated from a test specification.

Our testing approach is presented in terms of a formal framework. On the one hand, this enables us to identify and analyze the requirements on the design of the specification language in a formal way. On the other hand, on the formal basis we can, likewise, give a formal definition of the code generation algorithm which, in turn, allows us to formally prove its correctness.

The development of the testing framework goes through several stages: First, we introduce a programming language that captures a reasonable subset of features many modern object-oriented general-purpose programming languages like *Java* and C♯ have in common. In particular, the language comes with a formal

operational semantics giving a precise meaning to programs without ambiguities.

After that, we discuss and justify the new specification constructs which we use in order to define the test specification language. A crucial aspect is that the specification language is also equipped with an operational semantics giving a precise meaning to the test specifications, as well.

Next, we present the code generation algorithm. To cope with the complexity, the code generation is divided into two parts. First, a preprocessing step enriches the original specification with additional code such that the new specification has some useful features. Second, the actual code generation step transforms the new test specification into a test program of the programming language. Finally, we provide a correctness proof regarding the code generation algorithm. In particular, we show that the resulting test program indeed tests for the specified interactions.

As mentioned above, the programming language represents only a small subset of languages like *Java* or C♯. Also the corresponding test specification language lacks of several construct that would facilitate the writing of specifications. On account of this, we discuss some possible language extensions regarding, both, the programming language and the specification language.

An important feature of modern programming language is their support for *concurrency*. Therefore, we propose the introduction of concurrency into the programming language by means of *thread classes* which, in particular, allow to create new threads, dynamically. Finally we suggest a corresponding extension of the specification language and sketch a modification of the code generation algorithm.

Samenvatting

In de meeste software-ontwikkelingsprojecten van vandaag is testen nog steeds de enige bruikbare manier om de kwaliteit van een softwareproject te waarborgen. Rigoureus testen is nodig tijdens alle stages van de levenscyclus van de software-ontwikkeling. Terwijl systeem- en acceptatie-testen over het complete systeem gaan, is unit-testen gericht op de kleinste bouwstenen van de software. De verantwoordelijkheid voor de unit-testen is meer en meer verschoven van de softwaretesters naar de softwareontwikkelaars. Dit komt niet alleen door de groeiende populariteit van agile-software-ontwikkelingsmethodologieën. Naast de daaruit volgende voordelen voor de programmeurs, zoals het verhogen van hun kwaliteitsbewustzijn en onmiddellijke feedback, eist deze ontwikkeling ook nieuwe testraamwerken die rekening houden met de veranderde kwalificaties en verwachtingen van de softwareontwikkelaars. Tevens is door het sterk stijgende gebruik van object-georiënteerde programmeertalen ook een wijziging nodig wat betreft de unit-test benaderingen. In het bijzonder, in plaats van de traditionele toestand-gebaseerde tests door middel van een input-output vergelijking, is het in een object-georiënteerde context nuttiger om op interactie of gedrag gebaseerde tests uit te voeren, die op de volgorde van de interacties tussen de unit onder test en diens omgeving letten.

In dit proefschrift geven we een aanpak van unit-testen voor multi-purpose object-georiënteerde programmeertalen in de stijl van *Java* en C♯. Om aan de bovengenoemde eisen te voldoen, bevat onze aanpak de definitie van een taal voor het specificeren van de tests. Die is het resultaat van de uitbreiding van de eerdergenoemde programmeertaal met nieuwe taalconstructies voor de specificatie van de tests. Op deze manier hoeft de softwareontwikkelaar niet een compleet nieuwe taal te leren. Tegelijkertijd maakt het toevoegen van nieuwe constructies het mogelijk om de abstractie van de taal te verhogen ten opzichte van de specificatie van interactie-gebaseerde tests. Om een bepaalde test uit te voeren wordt de programma-code automatisch van een test-specificatie gegenereerd.

Onze testaanpak wordt gepresenteerd op basis van een formeel raamwerk. Dit stelt ons enerzijds in staat om de eisen aan het ontwerp van de specificatietaal op een formele manier te identificeren en te analyseren. Anderzijds kunnen we op deze formele basis eveneens een formele definitie geven van het code-generatiealgoritme, die ons zijnerzijds in staat stelt om de correctheid ervan formeel te bewijzen.

De ontwikkeling van het testraamwerk gaat door verschillende fasen: ten eerste

introduceren wij een programmeertaal, die een redelijke deelverzameling van features omvat die veel moderne general-purpose object-georiënteerde programmeertalen, zoals *Java* en C♯, gemeen hebben. In het bijzonder komt de taal met een formele operationele semantiek, die een precieze betekenis aan programma's geeft, zonder dubbelzinnigheden.

Daarna bespreken en rechtvaardigen wij de nieuwe specificatie constructies die wij gebruiken om de test-specificatietaal te definiëren. Een belangrijk punt is dat de specificatietaal eveneens uitgerust is met een operationele semantiek die een precieze betekenis geeft aan de test specificaties.

Vervolgens presenteren wij het code-generatiealgoritme. Om de complexiteit het hoofd te bieden is de code-generatie gesplitst in twee delen. Het eerste gedeelte is een voorverwerkingsstap die extra code aan de oorspronkelijke specificatie toevoegt, zodat de nieuwe specificatie een aantal handige features heeft. De tweede stap is de werkelijke code-generatie, die de nieuwe test-specificatie transformeert naar een testprogramma van de programmeertaal. Tot slot presenteren we een correctheidsbewijs van het code-generatiealgoritme. In het bijzonder tonen we aan dat het resulterende testprogramma inderdaad voor de opgegeven interacties test.

Zoals hierboven vermeld, vertegenwoordigt de programmeertaal slechts een kleine deelverzameling van talen zoals *Java* of C♯. Eveneens ontbreken een aantal constructies in de test-specificatietaal die het schrijven van bepaalde specificaties makkelijker zouden maken. Op grond daarvan bespreken wij een aantal mogelijke taaluitbreidingen voor zowel de programmeertaal als de specificatietaal.

Een belangrijke feature van de moderne programmeertalen is hun ondersteuning voor concurrency. Daarom stellen we de invoering voor van concurrency in de programmeertaal door middel van threadklassen die, in het bijzonder, het mogelijk maken om nieuwe threads dynamisch te creëren. Tot slot stellen we daarbij een uitbreiding voor van de specificatietaal en schetsen we een wijziging van het code-generatiealgoritme.