Reasoning about object-oriented programs: from classes to interfaces

Bian, J.

Nederlandse samenvatting

Software speelt een essentiële rol in hoe we met de echte wereld interacteren en is ingebed in enkele van de meest cruciale systemen die we dagelijks gebruiken. Het verzekeren dat deze software foutloos is en naar behoren functioneert, vormt een aanzienlijke uitdaging in de wereld van softwareontwikkeling. Het programmeringsparadigma van objectgeoriënteerd programmeren, dat breed wordt toegepast in de ontwikkeling van talrijke software systemen, staat centraal in deze discussie. Het inzetten van formele specificaties om de correctheid van objectgeoriënteerde programma's te verifiëren, biedt aanzienlijke voordelen. Zelfs geringe fouten in veelgebruikte programma's kunnen immers leiden tot grote problemen, zoals uitval en storingen van systemen. Dit proefschrift illustreert hoe formele methoden kunnen worden aangewend voor de systematische verificatie van geavanceerde, daadwerkelijk gebruikte objectgeoriënteerde programma's.

In het derde hoofdstuk richten we ons op de formele verificatie van objectgeoriënteerde klassen. We behandelen de specificatie en verificatie van een verbeterde versie van de implementatie van de gelinkte lijst uit het Java Collection Framework, die oorspronkelijk een bug gerelateerd aan geheugenoverloop bevatte. Onze formele specificatie streefde naar twee doelstellingen: ten eerste het aantonen van de afwezigheid van deze overflow bug en ten tweede het nauwkeurig beschrijven van het essentiële gedrag van de methoden, in relatie tot de structurele eigenschappen van de gelinkte lijst. We hebben met succes aangetoond, door het gebruik van de KeY theorem prover, dat de gecorrigeerde versie van de kernmethoden van de gelinkte lijst implementatie in Java formeel correct is.

In ons onderzoek naar gelinkte lijsten hebben we de meeste methoden geverifieerd, maar sommigen, die interfaces gebruiken, overgeslagen. Interfaces abstraheren van toestand en implementatiedetails, wat modulaire softwareontwikkeling bevordert. Echter, veel programmeerlogica en specificatietalen zijn toestandgebaseerd en niet direct geschikt voor interfaces. In Hoofdstuk 4 introduceren we een nieuwe methode die 'geschiedenissen' gebruikt voor het vastleggen van interface-interacties. Deze methode, via 'attributen', beschrijft objectgedrag onafhankelijk van implementatie. Hiermee kunnen in JML geschreven interfacespecificaties verwijzen naar deze 'geschiedenissen' en attributen om implementatiegedrag te definiëren.

Om de toepasbaarheid van de benadering gebaseerd op geschiedenisredenering te demonstreren, hebben we in Hoofdstuk 5 enkele kernmethoden van de Java Collection interfaces beschreven met behulp van de *uitvoerbare* geschiedenisgebaseerde methode (de EHB-methode). Deze aanpak hanteert een weergave van geschiedenis-

sen als Java-objecten in het heapgeheugen. Deze representatie maakte echter gebruik van zogenaamde pure methoden in de specificatie. Hoewel deze methodologie in theorie functioneert, bleken de pure methoden in de praktijk, vooral bij geavanceerd gebruik, te leiden tot aanzienlijke overhead en complexiteit in het bewijsproces.

Om de geschiedenisgebaseerde methode verder te verfijnen, behandelt Hoofdstuk 6 de integratie van abstracte datatypes (ADT's) in de KeY theorem prover. Dit gebeurt via een innovatieve aanpak die datatypes modelleert met behulp van Isabelle/HOL als een interactieve back-end, en Isabelle-theorema's vertaalt naar door gebruikers gedefinieerde taclets in KeY. We conceptualiseren geschiedenissen als elementen van een ADT, onderscheiden van de door Java gebruikte types in de EHB-benadering (Hoofdstuk 5). Hierdoor kunnen de Java-programma's die geverifieerd worden de geschiedenissen zelf niet wijzigen. We duiden dit aan als de *logische* geschiedenisgebaseerde benadering (LHB-benadering). We hebben aangetoond hoe extern in Isabelle/HOL gedefinieerde ADT's ingezet kunnen worden binnen JML-specificaties en KeY-bewijzen voor Java-programma's. Als een diepgaandere casestudie hebben we de methode addAll gespecificeerd en de correctheid van de eigenschappen van zijn cliënten geverifieerd. Bovendien hebben we geavanceerde, realistische scenario's onderzocht waarbij meerdere instanties van dezelfde interface betrokken zijn.

Hoofdstuk 7 focust op het gebruik van hiërarchie binnen objectgeoriënteerde programma's. Deze hiërarchie volgt natuurlijk uit gedragsafhankelijke subtyperingen, waarin de subtypen niet alleen moeten overeenkomen met de methodesignaturen zoals gedefinieerd door hun supertypen, maar ook het beoogde gedrag moeten volgen. Voor dit doeleinde ontwikkelen we een algemene theorie voor op geschiedenis gebaseerde verfijning, die ingezet wordt om de relatie tussen subtypen te verifiëren. Elke interface en klasse koppelen we aan een geschiedenis die de reeks methodenaanroepen weergeeft die op het object zijn uitgevoerd sinds zijn creatie. De relatie tussen subtypen wordt uitgedrukt in termen van een projectierelatie, die de connectie tussen subtypen en supertypen baseert op de projectie tussen hun respectievelijke geschiedenissen, waarbij elk type zijn eigen unieke geschiedenis heeft. Door middel van een doorlopend voorbeeld tonen we de praktische bruikbaarheid van onze aanpak aan.

Het programmeren op basis van interfaces vormt een fundamenteel principe binnen objectgeoriënteerd programmeren en speelt een centrale rol in de meeste standaardbibliotheken, die een hiërarchie van interfaces en klassen bieden ter vertegenwoordiging van objectcontainers. We stellen technieken voor die in staat zijn om de structuur van klassen, interfaces en hun hiërarchie te specificeren en verifiëren. Aan de hand van het Java Collection Framework als casestudie, demonstreren we de effectiviteit van onze technieken. Hiermee biedt dit proefschrift significante bijdragen en nieuwe inzichten die van waarde zijn voor de onderzoeksgemeenschap en toekomstige softwareverificatieprojecten.