KATHOLIEKE UNIVERSITEIT LEUVEN
FACULTEIT INGENIEURSWETENSCHAPPEN
DEPARTEMENT COMPUTERWETENSCHAPPEN
AFDELING INFORMATICA
Celestijnenlaan 200 A — B-3001 Leuven

A Statically Verifiable Programming Model for Concurrent Object-Oriented Programs

Promotoren :
Prof. Dr. ir. F. PIESSENS
Prof. Dr. ir. E. STEEGMANS

Proefschrift voorgedragen tot
het behalen van het doctoraat
in de ingenieurswetenschappen

door

Bart JACOBS

februari 2007

KATHOLIEKE UNIVERSITEIT LEUVEN
FACULTEIT INGENIEURSWETENSCHAPPEN
DEPARTEMENT COMPUTERWETENSCHAPPEN
AFDELING INFORMATICA
Celestijnenlaan 200 A — B-3001 Leuven

A Statically Verifiable Programming Model for Concurrent Object-Oriented Programs

Jury :
Prof. Dr. ir. G. De Roeck, voorzitter
Prof. Dr. ir. F. Piessens, promotor
Prof. Dr. ir. E. Steegmans, promotor
Prof. Dr. B. Demoen
Prof. Dr. T. Holvoet
Prof. Dr. ir. W. Joosen
Prof. Dr. ir. E. Poll
Dr. ir. W. Schulte

Proefschrift voorgedragen tot
het behalen van het doctoraat
in de ingenieurswetenschappen

door

Bart JACOBS

U.D.C. 681.3*D24, 681.3*F31, 681.3*D1

februari 2007

©Katholieke Universiteit Leuven – Faculteit Ingenieurswetenschappen
Arenbergkasteel, B-3001 Heverlee (Belgium)

Alle rechten voorbehouden. Niets uit deze uitgave mag worden vermenigvuldigd en/of openbaar gemaakt worden door middel van druk, fotocopie, microfilm, elektronisch of op welke andere wijze ook zonder voorafgaande schriftelijke toestemming van de uitgever.

All rights reserved. No part of the publication may be reproduced in any form by print, photoprint, microfilm or any other means without written permission from the publisher.

D/2007/7515/6
ISBN 978–90–5682–772–4

Acknowledgements

This work would not have been possible without the support of several people, all of whom I would like to thank here. The following enumeration is likely to be incomplete, for which I apologize.

First and foremost, I would like to express my gratitude to my supervisor, Prof. Frank Piessens. One could not wish for a better PhD supervisor. His guidance and support, his enthusiasm, and his qualities as a scientist have all been both extraordinary and instrumental. I am greatly looking forward to continuing my research under his guidance.

Also instrumental have been my two internships with the Spec# team at Microsoft Research in Redmond, WA, USA. I would like to thank Dr. Mike Barnett, Dr. Rustan Leino, and especially Dr. Wolfram Schulte, for the fun, instructive, and fruitful time I had, and for their continuing kindness, encouragement, and scientific example and collaboration. Further thanks go to the other members of Dr. Schulte's group for their kindness, and the other researchers, engineers, and interns which I had the pleasure of working with, especially Dr. Manuel Fähndrich, Herman Venter and Bor-Yuh Chang.

I have enjoyed greatly the friendship and collaboration of my fellow PhD students at the Department. I especially thank Jan Smans, who has been a great colleague and co-author, and my other great former or current office-mates Bart De Win, Tom Mahieu, Liesje Demuynck, Bart Elen, Steven Gevers, and Dries Vanoverberghe, as well as Marko Van Dooren for kindly offering to proof-read papers and for our many fascinating discussions. Further thanks go to the Professors, esp. Prof. Pierre Verbaeten and Prof. Wouter Joosen, and the excellent administrative staff, esp. Esther Renson.

Thanks further go to the other researchers and students worldwide who enlightened, inspired, or otherwise helped me. I would especially like to thank Prof. Gary Leavens, Prof. Joe Kiniry, Prof. Peter Müller, Prof. Erik Poll, Adam Darvas, Werner Dietl, and Matthew Parkinson.

I would like to thank Prof. Eric Steegmans, Prof. Bart Demoen, and Prof. Tom Holvoet, for agreeing to be on my supervisory committee and for dedicatedly executing its duties. Further thanks go to Dr. Wolfram Schulte, Prof. Erik Poll,

and Prof. Wouter Joosen for agreeing to be part of my jury and Prof. Guido De Roeck for chairing it. I am especially grateful to Dr. Schulte and Prof. Poll for agreeing to travel to Leuven for the occasion.

I acknowledge the Flemish Fund for Scientific Research (F.W.O.-Vlaanderen) (Belgium) for funding my research.

Last but not least, I would like to express my deepest gratitude to my family and friends, and especially my parents and my sister and her partner, for their love and support.

Abstract

Computers and computer programs play an increasingly important role in almost every area of human activity. However, failures in computer systems caused by program errors are common. There is an increasing need for tools to verify the absence of errors in programs.

Many important classes of computer programs are concurrent. Furthermore, due to evolutions in hardware, there is an increasing need for additional concurrency. However, delivering correct concurrent programs is particularly difficult. Existing approaches to detect program errors, in particular testing, are ineffective for finding concurrency errors. At the same time, future increases in computer system performance depend critically on increasing the ability of programmers to deliver correct concurrent programs.

This thesis presents an approach for verifying that a concurrent program does not contain certain common programming errors related to the use of locks (the most common concurrent programming paradigm). Specifically, the approach guarantees the absence of data races and deadlocks. It also guarantees compliance of the program with programmer-provided correctness criteria such as method contracts and object invariants. Finally, it guarantees that it is sound for programmers to reason sequentially about their program, except at synchronization points. The approach is modular, to increase scalability.

Under the proposed approach, programmers write their program according to a particular regime (or *programming model*) for the correct use of locks, defined in this thesis. Amongst other things, the regime requires that the program text be annotated in certain specific ways to make the programmer's intentions explicit and to allow modular verification. When a program module is finished, the programmer may then pass the program module to a tool that verifies compliance with the programming model, as well as compliance with programmer-specified correctness criteria. Compliance with the programming model guarantees that there are no data races or deadlocks and that sequential code may be reasoned about sequentially. The tool operates by generating verification conditions in the form of formulae of first-order logic, and passing them to an automatic theorem prover.

To improve the modularity of the approach, it includes an approach for referring, in a module's specification, to the module's state abstractly, using the mechanism of inspector methods. The approach further supports immutable objects, as well as static fields, lazy class initialization, and static class invariants.

We performed a validation of the approach by creating a prototype program verifier implementation as an extension of the Boogie program verifier. We obtained indications that the approach works for useful, non-trivial programs, by using the implementation to verify a number of small concurrent programs, the largest of which is a chat server. In addition, we developed a formalization of the core of the approach and a soundness proof.

Een Statisch Verifieerbaar Programmeermodel voor Gelijktijdige Objectgeoriënteerde Programma's

Overzicht

Vele kritische computerprogramma's, met hoge betrouwbaarheidsvereisten, zijn gelijktijdig (m.a.w., de volgorde van de bewerkingen is niet volledig gespecificeerd) en zijn geschreven in op gedeeld geheugen gebaseerde gelijktijdige imperatieve programmeertalen. In de huidige stand van de technologie hebben programmeurs grote moeite met het afleveren van dergelijke programma's zonder programmeerfouten. Deze thesis stelt een aanpak voor die programmeurs kunnen gebruiken om te verifiëren dat een op gedeeld geheugen gebaseerd gelijktijdig imperatief programma, of een module van zo'n programma, vrij is van bepaalde vaak voorkomende programmeerfouten. Meer specifiek garandeert de aanpak de afwezigheid van dataraces en deadlocks en fouten tegen door de programmeur gespecificeerde correctheidscriteria. In deze aanpak annoteren programmeurs hun programma en voeren er dan een hulpprogramma op uit dat verificatiecondities genereert om te worden bewezen door een automatische stellingenbewijzer. De aanpak ondersteunt toestandsabstractie door middel van inspectormethodes, alsook onwijzigbare objecten en luie klasse-initialisatie. We hebben een prototype-implementatie ontwikkeld en met succes een aantal kleine programma's geverifieerd, en het verder werk nodig om de aanpak toepasbaar te maken in grote projecten geïdentificeerd.

Deze samenvatting begint met een inleiding tot de thesis. Dan volgen twee secties die de kernidee schetsen van de aanpak voor toestandsabstractie, resp. verificatie van gelijktijdige programma's. De samenvatting eindigt met een besluit

voor de thesis.

1 Inleiding

Deze inleidende sectie beschrijft eerst het probleem dat door deze thesis behandeld wordt, alsook de voorgestelde oplossing. Ze lijst dan de onderzoeksbijdragen op die beschreven worden in de thesis en de andere behaalde onderzoeksresultaten.

1.1 Probleembeschrijving

Computers en computerprogramma's spelen een steeds belangrijkere rol op bijna elk gebied van menselijke activiteit. Falingen in computersystemen veroorzaakt door programmeerfouten komen echter vaak voor. Er is een toenemende nood aan hulpmiddelen om de afwezigheid van fouten in programma's te verifiëren.

Veel computerprogramma's zijn *gelijktijdig*, wat betekent dat de volgorde waarin de operaties plaatsvinden niet volledig gespecificeerd is. Dit weerspiegelt ofwel inherente gelijktijdigheid in de invoer vanuit de omgeving van het computersysteem, ofwel een keuze van de programmeur om de vereisten betreffende de volgorde te verzwakken teneinde de mogelijkheid tot parallelle uitvoering en de performantie te verhogen.

Gelijktijdige computerprogramma's worden typisch geschreven gebruikmakend van het concept van een *uitvoeringsdraad* (of kortweg *draad*). Een draad is een entiteit die de instructies van een programma leest en ze een voor een uitvoert. Er is geen gelijktijdigheid tussen de operaties van een gegeven draad. Er ontstaat gelijktijdigheid wanneer hetzij de omgeving hetzij het programma zelf bijkomende draden creëert. Wanneer meerdere draden bestaan, opereren ze allemaal gelijktijdig.

Wanneer twee draden gelijktijdig opereren kan het gebeuren dat ze tegelijk eenzelfde hulpbron proberen te benaderen. De hulpbronnen die het vaakst benaderd worden door programma's zijn geheugenlocaties. Indien beide draden eenvoudigweg de huidige waarde opgeslagen in de geheugenlocatie uitlezen, is er geen probleem. Echter, indien een of beide draden de bestaande waarde vervangen door een nieuwe waarde, spreekt men van een datarace.

Indien een datarace plaatsvindt, gaat het meestal om een programmeerfout. De intentie van de programmeur is doorgaans dat de verschillende draden atomaire (m.a.w. effectief ogenblikkelijke) bijwerkingen doen van de gegevensverzamelingen die het programma bijhoudt. Een gegevensverzameling is doorgaans te groot om in een enkele geheugenlocatie opgeslagen te worden. Bovendien moeten de verschillende geheugenlocaties die samen de gegevensverzameling bevatten doorgaans zowel gelezen als geschreven worden als onderdeel van een gegeven bijwerking. Indien een draad probeert een gegevensverzameling bij te werken zonder te wachten tot andere draden klaar zijn met het bijwerken van dezelfde gegevensverzameling,

m.a.w. zonder te *synchroniseren*, dan kunnen de bijwerkingen met elkaar interfereren en dan kan het resultaat incorrect zijn. Dataraces zijn een symptoom van ongesynchroniseerde bijwerkingen.

Programmeurs realiseren synchronisatie tussen gelijktijdige bijwerkingen doorgaans door middel van het mechanisme genaamd *grendels*. De programmeur associeert met elke gegevensverzameling een grendel. Het programma wordt zodanig geschreven dat een draad, voorafgaand aan een bijwerking van een gegevensverzameling, eerst de grendel geassocieerd met de gegevensverzameling probeert te sluiten. Slechts één draad mag een gegeven grendel op een gegeven ogenblik gesloten houden. Als een andere draad de grendel al gesloten houdt, moet de draad wachten. De draad opent de grendel pas dan weer, wanneer de bijwerking voltooid is. Dit zorgt ervoor dat de geheugenoperaties waaruit twee gelijktijdige bijwerkingen van eenzelfde gegevensverzameling bestaan, niet door elkaar plaatsvinden.

Grendels voorkomen dataraces indien ze correct gebruikt worden, maar het correct gebruik van grendels is moeilijk. Als een programmeur vergeet een grendel te sluiten, of de verkeerde grendel sluit bij het benaderen van een gegevensverzameling, dan is een datarace het gevolg. Bovendien bestaat er gevaar op *deadlocks*. Een groep draden bevindt zich in een deadlock indien elke draad aan het wachten is tot een andere draad van de groep een grendel weer opent. Bijgevolg leveren de draden in de groep geen nuttig werk meer en het programma “hangt”.

Dat correcte meerdradige programma’s schrijven zo moeilijk is, is een belangrijk probleem want vele belangrijke klassen van programma’s zijn gelijktijdig. Bijvoorbeeld, de meeste systeemprogrammatuur zoals besturingssystemen, gedistribueerde programmatuur zoals de programma’s die het internet doen draaien, of andere klant-leverancier-programmatuur zoals gegevensbanken en toepassingsleveranciers, en ingebedde programmatuur zoals programmatuur voor mobiele telefoons, is gelijktijdig. Ook vele toepassingsprogramma’s zijn gelijktijdig. Bijvoorbeeld, recente computerspellen zijn gelijktijdig om meerdere processoren uit te buiten, en toepassingen met een grafische gebruikersinterface zijn gelijktijdig om te voorkomen dat de gebruikersinterface te traag reageert ten gevolge van lang lopende bewerkingen.

Bovendien is er steeds meer vraag naar het gelijktijdig maken of meer gelijktijdig maken van programma’s, om redenen van performantie. De producenten van microprocessoren komen steeds dichter bij het uiterste aantal instructies dat door een enkele microprocessor uitgevoerd kan worden per tijdseenheid. Bijgevolg moeten programma’s, om de performantie te verhogen, zo geschreven worden dat het te verrichten werk verdeeld wordt over meerdere microprocessoren. Dergelijke programma’s zijn noodzakelijkerwijze gelijktijdig. Om in de toekomst de performantie van computersystemen nog te kunnen blijven verhogen, zal het cruciaal zijn om programmeurs beter in staat te stellen om correcte gelijktijdige programma’s af te leveren.

Ten slotte moet erop gewezen worden dat fouten veroorzaakt door de gelijk-

tijdigheid van programma's moeilijk op te sporen zijn met bestaande aanpakken om programmeerfouten te vinden, en in het bijzonder door het testen ervan. Om een programma te testen voert men het uit en kijkt men na dat de uitvoer correct is. Echter, een gelijktijdig programma kan voor gegeven invoer een zeer groot aantal verschillende uitvoeringen hebben, ten gevolge van de verschillende volgordes waarin de operaties van de verschillende draden plaatsvinden. Het testen van programma's kan slechts een klein deel van de mogelijke uitvoeringen van een gelijktijdig programma nakijken. Aangezien fouten gerelateerd aan de gelijktijdigheid van een programma dikwijls slechts leiden tot een faling in heel specifieke ordeningen van de operaties, is de kans op het vinden van dergelijke fouten door de programma's te testen, klein.

1.2 Voorgestelde oplossing

Deze thesis stelt een aanpak voor om na te gaan dat bepaalde vaak voorkomende programmeerfouten verbonden met het gebruik van grendels niet voorkomen in een gelijktijdig programma. Meer bepaald garandeert de aanpak de afwezigheid van dataraces en deadlocks. Ze garandeert ook dat het programma voldoet aan door de programmeur gespecificeerde correctheidscriteria, zoals methodecontracten en objectinvarianten. Ten slotte garandeert ze dat programmeurs veilig sequentieel kunnen redeneren over hun programma, tenzij op synchronisatiepunten. De aanpak is modulair, wat de schaalbaarheid verhoogt.

Onder de voorgestelde aanpak schrijven programmeurs hun programma volgens een bepaald regime (of *programmeermodel*) voor het correct gebruik van grendels, gedefinieerd in deze thesis. Onder andere vereist het regime dat de programmatekst geannoteerd wordt op welbepaalde manieren, om de intenties van de programmeur expliciet te maken en om modulaire verificatie mogelijk te maken. Eens een programmamodule af is, kan de programmeur haar doorgeven aan een hulpprogramma dat verifieert dat het programma voldoet aan het programmeermodel alsook met de door de programmeur gespecificeerde correctheidscriteria. Als een programma voldoet aan het programmeermodel, dan bevat het gegarandeerd geen dataraces of deadlocks en dan mag de programmeur sequentieel redeneren over sequentiële code. Het hulpprogramma gaat te werk door verificatiecondities te genereren in de vorm van formules in eerste-orde-logica, en die door te geven aan een automatische stellingenbewijzer.

Om concreet te kunnen zijn, formuleert de thesis de aanpak in termen van de populaire objectgeoriënteerde imperatieve programmeertaal Java (behalve Hoofdstuk 2, dat geformuleerd is in termen van C#, een taal die in essentie een uitbreiding is van Java). Zoals in de conclusie van deze thesis verdedigd zal worden, zijn de kernideeën van de aanpak toepasbaar op op gedeeld geheugen gebaseerde imperatieve programmeertalen in het algemeen. De thesis maakt gebruik van het betere ontwerp van objectgeoriënteerde talen in vergelijking met hun niet-objectgeoriënteerde voorgangers om tot een elegantere verificatie-aanpak te komen. Bij-

voorbeeld, het is niet alleen gemakkelijker om elegante programma's te schrijven met virtuele methodes dan met functiewijzers, dergelijke programma's zijn ook gemakkelijker te verifiëren.

Het programmeermodel is gebaseerd op de notie van *objecttoegankelijkheid*, gedefinieerd in de thesis. (Toegankelijkheid is niet hetzelfde als *bereikbaarheid*. Het programmeermodel legt geen beperkingen op aan het doorgeven van objectreferenties; het beperkt enkel het gebruik ervan.) Op elk ogenblik tijdens de uitvoering van een programma is een gegeven object *toegankelijk* voor ten hoogste één draad. Een draad mag een veld lezen of schrijven enkel indien het object toegankelijk is voor de draad. Initieel is een object toegankelijk voor de draad die het aanmaakte. Daarnaast kan elke draad toegang krijgen tot het object door de grendel van het object te sluiten. Echter, om een race te voorkomen tussen de draad die het object creëerde en een draad die de grendel gesloten houdt, maakt het model onderscheid tussen *gedeelde* en *niet-gedeelde* objecten. Een object is initieel niet-gedeeld. Een draad mag de grendel van een object proberen te sluiten enkel indien het object gedeeld is. Een draad die toegang heeft tot een niet-gedeeld object kan het delen door er een *deel-operatie* op uit te voeren, zoals gespecificeerd door de programmeur door middel van een programma-annotatie. Vanaf dit ogenblik is het object niet langer toegankelijk voor deze of andere draden, totdat een draad de grendel van het object sluit.

Om deadlocks te voorkomen vereist het programmeermodel dat de programmeur een partiële orde definieert op de gedeelde objecten. Een draad mag een grendel enkel proberen te sluiten indien de grendel kleiner is in de partiële orde dan de grendels die de draad al gesloten houdt.

Er bestaan verschillende andere aanpakken om de afwezigheid van dataraces en deadlocks in gelijktijdige programma's te verifiëren. Het betreft aanpakken gebaseerd op het genereren van verificatiecondities [23, 32, 36, 77, 1, 79], typesystemen [16, 34], en andere. (Zie Sectie 4.8 van de thesis voor een overzicht van gerelateerd werk.) Echter, de aanpak van deze thesis is de eerste veilige aanpak die garandeert dat het veilig is om sequentieel te redeneren over sequentiële code, die objectinvarianten verifieert die betrekking hebben op gegevensverzamelingen bestaande uit meerdere objecten, en die de overdracht van eigendom ondersteunt van objecten waarnaar meerdere referenties bestaan.

We hebben een validatie van de aanpak uitgevoerd door een prototype-implementatie te creëren als een uitbreiding van het programmaverificatieprogramma voor sequentiële programma's genaamd Boogie. We hebben indicaties verkregen dat de aanpak werkt voor bruikbare, niet-triviale programma's, door de implementatie te gebruiken om een aantal kleine gelijktijdige programma's te verifiëren, waaronder een babbeldienstprogramma.

De aanpak van deze thesis richt zich op op gedeeld geheugen gebaseerde imperatieve gelijktijdige programma's, waarin wijzigbare gegevensstructuren vrij gedeeld worden tussen uitvoeringsdraden. Veel, zo niet de meeste, gelijktijdige program-

ma's vallen in deze categorie. Echter, er bestaan alternatieve paradigma's voor gelijktijdig programmeren waarin wijzigbare gegevensstructuren nooit gedeeld worden en dataraces bijgevolg nooit kunnen voorkomen. Voorbeelden van programmeertalen die dergelijke paradigma's ondersteunen zijn de functionele talen Erlang en Concurrent Haskell, en de voorgestelde uitbreiding genaamd SCOOP voor de imperatieve programmeertaal Eiffel.

Indien draden geen wijzigbare gegevensstructuren delen, voorkomt dit dataraces, maar het zorgt ook voor een betere foutisolatie en bijgevolg voor robuustere systemen. Bijvoorbeeld, Erlang wordt met succes gebruikt door Ericsson in telecomapparatuur met hoge beschikbaarheidsvereisten. Niettemin hebben talen die gedeelde gegevensstructuren verbieden geen brede ingang gevonden, vermoedelijk omdat programmeurs ze beschouwen als te beperkend. Het valt buiten de opzet van deze thesis, te oordelen of deze perceptie gerechtvaardigd is.

1.3 Bijdragen

De bijdragen van de thesis zijn de volgende:

- Een uitbreiding van de Boogie-aanpak [9] met betere ondersteuning voor modulariteit door toestandsabstractie mogelijk te maken door middel van inspectormethodes. Inspectormethodes mogen afhangen van de velden van het ontvangerobject alsook de toestand van de objecten waarvan de ontvanger eigenaar is. Ze mogen abstract zijn en ze mogen overschreven worden. Ze mogen parameters hebben en ze mogen afhangen van de toestand van een parameter indien aangegeven.

- Een uitbreiding van de Boogie-aanpak met ondersteuning voor gelijktijdige programma's die grendels gebruiken. De aanpak voorkomt dataraces en garandeert dat veilig sequentieel geredeneerd kan worden over sequentiële code. De aanpak ondersteunt eigendomsoverdracht van objecten waarnaar meerdere referenties bestaan.

- Een aanpak voor het verifiëren van de afwezigheid van deadlocks. De partiële orde op de objecten kan worden gespecificeerd door een *grendelniveau* toe te kennen aan een object op het ogenblik dat het gedeeld wordt. Grendelniveaus zijn waarden die gecreëerd, gemanipuleerd, en opgeslagen kunnen worden net zoals andere waarden.

- Ondersteuning voor onwijzigbare objecten. Om onwijzigbare objecten te ondersteunen, wordt onderscheid gemaakt tussen toegankelijkheid voor lezen en toegankelijkheid voor schrijven. Elk niet-gedeeld object kan gedeeld worden als een door een grendel beschermd object, of als een onwijzigbaar object.

- Ondersteuning voor statische velden, statische initialisatieroutines, en statische klasse-invarianten. De aanpak is veilig met betrekking tot de luie klasse-initialisatie-semantiek van Java en C#. Door de aanpak te integreren met de aanpak van gelijktijdigheid kan men veilig sequentieel redeneren over sequentiële code en worden deadlocks bij klasse-initialisatie voorkomen.
- Een prototype-implementatie van de aanpak als een uitbreiding van het programmaverificatieprogramma Boogie. De implementatie is beschikbaar op het web in binaire vorm.
- Initiële ervaring met het toepassen van de aanpak op niet-triviale programma's, inclusief een babbeldienst. De ervaring levert voorlopige indicaties dat de aanpak voldoende flexibel en expressief is om toegepast te worden op niet-triviale, bruikbare gelijktijdige programma's.
- Een formalisatie en bewijs van de kern van de aanpak. De annotatiesyntax en de semantiek van het programmeermodel voor het voorkomen van dataraces werden geformaliseerd en correct bewezen. Verder werden de regels voor het genereren van verificatiecondities geformaliseerd en correct bewezen.

1.4 Ander geleverd werk

Naast het onderzoek beschreven in deze thesis, zijn de belangrijkste behaalde onderzoeksresultaten de volgende:

- Een programmeermodel voor programma's die het *Iterator*-ontwerppatroon implementeren door middel van de *iterators* en *foreach-lussen* van C# 2.0.
- De auteur droeg bij aan een aanpak voor het verifiëren dat een programma voldoet aan een op stapelinspectie gebaseerd toegangscontrolebeleid.

2 Inspectormethodes

Deze sectie geeft een overzicht van onze aanpak voor toestandsabstractie met behulp van inspectormethodes. We geven eerst een overzicht. Dan schetsen we de kernidee van de aanpak aan de hand van een voorbeeld. De sectie eindigt met een besluit.

2.1 Overzicht

In een objectgeoriënteerd programma levert een klasse typisch toegang tot de toestand van een object via methodes in plaats van door rechtstreeks toegang te leveren tot de velden en interne hulpobjecten van het object. Dit zorgt ervoor dat cliëntcode niet afhangt van en niet kan interfereren met de interne voorstelling

van het object. Methodes die hiervoor gebruikt worden, worden soms *inspectormethodes* genoemd. Om de voordelen van inspectormethodes uit te breiden naar de specificaties, kan men de methodecontracten van niet-inspectormethodes uitdrukken met behulp van inspectormethodes, en op die manier toestandsabstractie in specificaties realiseren.

We stellen een aanpak voor voor het verifiëren van programma's die inspectormethodes gebruiken in methodecontracten en objectinvarianten. Inspectormethodes mogen parameters hebben en ze mogen afhangen van de toestand van objecten die als argumenten meegegeven worden. Onze aanpak bouwt verder op de Boogie-methodologie voor objectinvarianten en eigendom.

Toestandsabstractie realiseren in een programmeertaal die toelaat dat er meerdere referenties zijn naar eenzelfde object, zorgt voor problemen bij het specificeren van het effect van een methode. Meer bepaald moet een cliënt te weten kunnen komen of het wijzigen van een gegeven object of het oproepen van een gegeven methode de waarde van een gegeven inspectormethode-oproep kan beïnvloeden. We lossen dit op door inspectormethodes te modelleren als functies die als argumenten enkel die delen van het objectgeheugen nemen waarvan ze afhangen. Dankzij een nieuwe logische encodering van het objectgeheugen kunnen we dit doen zonder implementatiedetails prijs te geven, zelfs in gevallen waar inspectormethodes afhangen van interne hulpobjecten.

De kern van onze aanpak is geïmplementeerd in een aangepaste versie van het programmaverificatieprogramma Spec#.

2.2 Voorbeeld

Beschouw het programma in Figuur 1. Het bestaat uit een klasse *IntList* en een stukje cliëntcode. Een object van de klasse *IntList* stelt een lijst van gehele getallen voor. Het object houdt het aantal elementen bij in het veld *count* en de elementen zelf in een rij-object. Een verwijzing naar het rij-object wordt bijgehouden in het veld *elems*.

De klasse *IntList* stelt twee methodes ter beschikking van cliënten om de toestand van een *IntList*-object te inspecteren: de methodes *getCount* en *getItem*. Daarnaast gebruikt de klasse deze methodes ook zelf, in de precondities en postcondities van haar methodes, om ze te kunnen formuleren zonder implementatiedetails prijs te geven.

Het gebruik van inspectormethodes in methodecontracten leidt ertoe dat cliënten moeten kunnen redeneren over deze inspectormethodes. Meer bepaald moeten cliënten voldoende informatie hebben om te kunnen afleiden of een bepaalde wijziging in de programmatoestand al dan niet een invloed heeft op de resultaatwaarde van een gegeven inspectormethode-oproep.

Dit wordt geïllustreerd door het stukje cliëntcode onderaan Figuur 1. Om af te kunnen leiden of de **assert**-opdracht slaagt of faalt, moet de cliënt weten of

```
final class IntList {
  rep int[] elems;
  int count;

  invariant 0 ≤ count ∧ count ≤ elems.length;

  inspector int getCount() { return count; }
  inspector int getItem(int index)
    requires 0 ≤ index ∧ index < getCount();
  { return elems[index]; }

  derived_invariant 0 ≤ getCount();

  IntList(int[] xs)
    requires ¬xs.committed;
    ensures ¬this.committed ∧ this.inv;
    ensures getCount() = xs.length;
    ensures
      forall{int i in (0 : getCount()); getItem(i) = xs[i]};
  { . . . }

  void add(int x)
    requires ¬this.committed ∧ this.inv;
    modifies this.∗;
    ensures ¬this.committed ∧ this.inv;
    ensures getCount() = old(getCount()) + 1;
    ensures
      forall{int i in old((0 : getCount())); getItem(i) = old(getItem(i))};
    ensures getItem(old(getCount())) = x;
  {
    unpack this;
    count++;
    ensureCapacity(count);
    elems[count − 1] := x;
    pack this;
  }

  . . .
}

int[] xs := {1, 2, 3};
IntList list := new IntList(xs);
xs[0] := 5;
assert list.getItem(0) = 1;
```

Figuur 1: Deze klasse illustreert inspectormethodes, objectinvarianten, eigendom, geparametriseerde inspectormethodes, inspectormethodeprecondities, en afgeleide invarianten. (Let wel: de vermelde referentietypes moeten gelezen worden als niet-null-types.)

de toekenning aan het rij-element een invloed heeft op de resultaatwaarde van $list.getItem(0)$ of niet.

Onze aanpak lost dit probleem op als volgt. We leggen de regel op dat een inspectormethode-oproep enkel mag afhangen van de velden van het ontvangerobject en van de objecten die transitief *eigendom* zijn van het ontvangerobject. Een object p is *eigendom* van een object o in een gegeven programmatoestand als en slechts als o een veld f heeft dat gemarkeerd is met het sleutelwoord **rep** en $o.f = p$ en o is *ingepakt*. Het ingepakt zijn van een object wordt aangegeven door de waarde van het speciale booleaanse veld *inv* dat in onze aanpak toegevoegd wordt aan elk object: o is ingepakt als $o.inv$ gelijk is aan *true* en *uitgepakt* in het andere geval. Een object is initieel uitgepakt. Een object o kan ingepakt worden met de speciale opdracht **pack** o; en weer uitgepakt met **unpack** o;.

We zeggen dat een object *in eigendom* is in een gegeven programmatoestand als en slechts als het object eigendom is van een ander object. Dit is een belangrijke eigenschap en daarom geven we dit ten overvloede aan met een speciaal booleaans veld *committed* dat, net zoals *inv*, in onze aanpak toegevoegd wordt aan elk object. De opdrachten **pack** en **unpack** zorgen ervoor dat steeds de eigenschap geldt dat het *committed*-veld van een object o gelijk is aan *true* als en slechts als o in eigendom is.

Dankzij deze aanpak is het mogelijk voor de cliënt in ons voorbeeld, af te leiden dat de inspectormethode-oproep $list.getItem(0)$ niet beïnvloed wordt door de toekenning aan het rij-element. Immers, het rij-object is geen transitief eigendom van het object *list* op het ogenblik van de toekenning. Dit volgt uit het feit dat het rij-object op dat ogenblik niet in eigendom is. En dit volgt dan weer uit het feit dat het rij-object niet in eigendom is net nadat het gecreëerd is, plus het feit dat het contract van de *IntList*-constructor garandeert dat de constructor het rij-object, en dus ook haar *committed*-veld, niet wijzigt.

2.3 Besluit

We toonden in deze sectie met een voorbeeldje aan hoe onze aanpak het mogelijk maakt cliëntcode te verifiëren van klassen die toestandsabstractie realiseren door middel van het gebruik van inspectormethodes in methodecontracten. De thesis gaat hierop dieper in en legt ook uit hoe de aanpak inspectormethodes mogelijk maakt die afhangen van de toestand van argumenten, en hoe de aanpak toegepast kan worden op programma's waarin sommige klassen subklassen zijn van andere klassen.

3 Gelijktijdigheid

In deze sectie schetsen we de kernidee van onze aanpak voor de modulaire statische verificatie van de afwezigheid van dataraces in Java-achtige programma's. Meer

bepaald leggen we het programmeermodel uit. De thesis beschrijft ook hoe vanuit dit programmeermodel gekomen wordt tot een aanpak voor modulaire statische verificatie, alsook uitbreidingen voor het voorkomen van deadlocks en hoog-niveauraces op gegevensstructuren bestaande uit meerdere objecten.

3.1 Programmeermodel voor afwezigheid van dataraces

We beschrijven het programmeermodel in de context van Java, maar het is evenzeer toepasbaar op C# en andere gelijkaardige talen.

Voordat we het programmeermodel uitleggen, bekijken we kort Java's ingebouwd synchronisatiemechanisme. In Java kunnen draden met elkaar synchroniseren door middel van de **synchronized**-opdracht. Een draad kan een **synchronized** (o)-blok alleen binnengaan als geen andere draad zich binnen een **synchronized** (o)-blok bevindt; anders wacht de draad. We gebruiken de volgende terminologie om naar Java's synchronisatiemechanisme te verwijzen: wanneer een draad een **synchronized** (o)-blok binnengaat, zeggen we dat het *de grendel van o sluit* of, korter, dat het *o vergrendelt*; zolang ze zich in het blok bevindt, zeggen we dat het *de grendel van o gesloten houdt*; en wanneer ze het blok verlaat, zeggen we dat het *de grendel van o opent*, of korter, dat het *o ontgrendelt*. Merk op dat, in tegenstelling tot wat de terminologie misschien suggereert, wanneer een draad een object vergrendelt, Java andere draden verhindert het object te vergrendelen maar niet verhindert dat andere draden de velden van het object benaderen. Dit is in essentie het probleem dat door de voorgestelde methodologie opgelost wordt. Zolang een draad een object vergrendeld houdt, zeggen we ook dat het object *vergrendeld is* door de draad.

Een belangrijk terminologisch punt is het volgende: wanneer een draad t een **synchronized** (o)-blok bereikt, zeggen we dat de draad *o probeert te vergrendelen*. Er kan enige tijd verstrijken voordat de draad *o vergrendelt*, meer bepaald indien een andere draad o vergrendeld houdt. Inderdaad, indien de andere draad o nooit ontgrendelt, dan vergrendelt t o nooit. Het onderscheid is belangrijk want ons programmeermodel legt beperkingen op aan het proberen te vergrendelen van een object.

We leggen nu ons programmeermodel uit. Met *programmeermodel* bedoelen we een opdeling van de verzameling van de Java-programma's in *toegelaten programma's* (die programma's die voldoen aan het programmeermodel) en de *verboden programma's* (die niet voldoen). Ons programmeermodel definieert de verzameling van toegelaten programma's door een dynamische semantiek te definiëren voor Java-programma's die de dynamische semantiek van Java zelf uitbreidt met bijkomende toestandsvariabelen, een bijkomende operatie, en het concept *verboden operatie*. De toegelaten programma's zijn die die geen verboden operaties uitvoeren onder deze semantiek. Ons programmeermodel heeft de eigenschap dat toegelaten programma's vrij zijn van dataraces. Let wel: de uitgebreide dynamische semantiek dient enkel om de verzameling van toegelaten programma's te definiëren; onze

aanpak verifieert uiteindelijk de afwezigheid van dataraces in Java-programma's bij uitvoering onder de originele Java-semantiek.

Het programmeermodel is een eerste stap in de richting van een aanpak voor het modulair statisch verifiëren van de afwezigheid van dataraces in Java-programma's, want het transformeert een globale dynamische eigenschap (afwezigheid van dataraces) naar een per-draad dynamische eigenschap (het toegelaten zijn van een operatie). Secties 4.2.2 en 4.2.3 in de thesis transformeren deze per-draad dynamische eigenschap verder naar een per-methode statische eigenschap, namelijk geldigheid van een operatie.

Het programmeermodel zorgt ervoor dat toegelaten programma's vrij zijn van dataraces als volgt. In de uitgebreide dynamische semantiek wordt een bijkomende toestandsvariabele geassocieerd met elke draad t, namelijk de *toegangsverzameling* $t.A$ van de draad, die een verzameling is van objectidentiteiten. Een lees- of schrijfoperatie op een veld $o.f$ door een draad t is alleen toegelaten onder het programmeermodel indien het doelobject o behoort tot de toegangsverzameling $t.A$ van t. Onder de uitgebreide regels voor uitvoeringsstappen evolueren de toegangsverzamelingen op zo'n manier dat ze steeds disjunct zijn. Bijgevolg benaderen twee draden nooit gelijktijdig hetzelfde veld, en het programma is vrij van dataraces.

De toegangsverzameling van de hoofddraad is initieel leeg. Wanneer een draad een object creëert of de grendel van een object sluit, wordt het object toegevoegd aan de toegangsverzameling van de draad. Wanneer een draad de grendel van een object opent, wordt het object verwijderd uit de toegangsverzameling van de draad. Om een datarace te voorkomen tussen de draad die een object creëert en een draad die de grendel van het object sluit, maakt het programmeermodel onderscheid tussen *gedeelde* en *niet-gedeelde* objecten; objecten zijn initieel niet-gedeeld, en een poging om de grendel van een object te sluiten is enkel toegelaten indien het object gedeeld is. Formeel wordt het onderscheid tussen gedeelde en niet-gedeelde objecten gemaakt in de dynamische semantiek door de programmatoestand uit te breiden met een globale toestandsvariabele S, genaamd de *gedeelde verzameling*, die een verzameling is van objectidentiteiten. Een object wordt beschouwd als *gedeeld* in een gegeven toestand als en slechts als het behoort tot de gedeelde verzameling.

Een niet-gedeeld object wordt een gedeeld object wanneer een draad een *deel-operatie* uitvoert op het object. Een deel-operatie is een operatie in de uitgebreide dynamische semantiek die niet overeenkomt met een Java-opdracht. Een deel-operatie wijzigt enkel de bijkomende toestandsvariabelen; meer bepaald: een deel-operatie uitgevoerd door een draad t op een niet-gedeeld object o dat behoort tot $t.A$ verwijdert o uit $t.A$ en voegt het toe aan de gedeelde verzameling. Het is een draad verboden een deel-operatie uit te voeren op een object dat niet behoort tot haar toegangsverzameling.

Het is belangrijk te wijzen op het volgende: de bijkomende toestandsvariabe-

len worden gebruikt bij het definiëren van welke operaties verboden zijn, maar ze hebben geen invloed op het gedrag van het programma. Meer bepaald, als een Java-programma een gegeven operatie uitvoert onder de Java-semantiek (in een gegeven uitvoering), dan voert het deze operatie ook uit in de uitgebreide semantiek (in een gegeven uitvoering). Met andere woorden, de bijkomende toestand zorgt er niet voor dat operaties blokkeren die anders niet zouden blokkeren.

Merk ook op:

- Wanneer een draad een nieuw object creëert, wordt het object toegevoegd aan de toegangsverzameling van de deze draad. Dit maakt het mogelijk voor de constructor om de velden van het object te initialiseren zonder eerst een grendel te moeten sluiten. Dit betekent ook dat alle enkeldradige programma's toegelaten zijn onder het programmeermodel: als er slechts één draad is, creëert zij alle objecten, en kan zij ze benaderen zonder grendeloperaties.

- Eens een object gedeeld is, kan het nooit terugkeren naar de niet-gedeelde toestand.

- Wanneer een draad t een nieuwe draad opstart, wordt de toegankelijkheid van het ontvangerobject van de hoofdmethode van de draad (dit is het *Runnable*-object in Java, of het doelobject van het *ThreadStart*-delegeerobject in het .NET Framework) overgedragen van t naar de nieuwe draad. Dit is noodzakelijk om de hoofdmethode toe te laten haar ontvangerobject te benaderen.

Een object bevindt zich dus op elk ogenblik in één van de volgende drie toestanden: *niet-gedeeld*, *vrij* (gedeeld maar niet vergrendeld), of *vergrendeld* (gedeeld en vergrendeld door een van de actieve draden). Een object is initieel niet-gedeeld. Niet-gedeelde objecten waarop een deel-operatie uitgevoerd wordt, gaan over naar de toestand van gedeeld object. Na deze overgang behoort het object tot de toegangsverzameling van geen enkele draad, en wordt het *vrij* genoemd. Om een vrij object te benaderen moet het eerst vergrendeld worden. Dit brengt het object in de vergrendelde toestand en voegt het toe aan de toegangsverzameling van de vergrendelende draad. Het ontgrendelen van het object verwijdert het uit de toegangsverzameling en maakt het weer vrij.

We kunnen het bovenstaande samenvatten als volgt. Draden mogen enkel object benaderen die behoren tot hun toegangsverzameling. De toegangsverzameling van een draad op een gegeven ogenblik bestaat uit alle objecten die het creëerde of die het vergrendeld houdt, plus desgevallend het ontvangerobject van haar hoofdmethode, min de objecten waarop het een deel-operatie uitvoerde of die het als doelobject gebruikte bij het starten van een nieuwe draad. Toegelaten programma's zijn vrij van dataraces aangezien bij uitvoeringen van toegelaten programma's de toegangsverzamelingen van de verschillende draden nooit overlappen.

Meervoudig vergrendelen

De **synchronized**-opdracht van Java laat meervoudige vergrendeling toe; met andere woorden, wanneer een draad een object o al vergrendeld houdt, dan slaagt een poging om opnieuw een **synchronized** (o)-blok binnen te gaan onmiddellijk. Dit heeft gevolgen voor het programmeermodel. Meer bepaald moeten we het scenario voorkomen waarbij een draad t_1 een object o vergrendelt, het dan gebruikt als het doelobject om een draad t_2 te starten, en dan o opnieuw vergrendelt. Op dit ogenblik bevatten de toegangsverzamelingen van zowel t_1 als t_2 het object o en dataraces zijn mogelijk.

Een ander probleem met meervoudige vergrendeling is dat het modulaire verificatie moeilijker maakt. Daarom verbieden we meervoudige vergrendeling in het programmeermodel. We realiseren dit als volgt:

- We vereisen dat het doelobject bij het starten van een nieuwe draad niet-gedeeld is. Bijgevolg hebben we de eigenschap dat in elke programmatoestand, een gedeeld object behoort tot de toegangsverzameling van een draad als en slechts als de draad het object vergrendeld houdt.
- We vereisen dat op het ogenblik waarop een draad een object probeert te vergrendelen, het object nog niet behoort tot de toegangsverzameling van de draad. Hieruit volgt dat de draad het object nog niet vergrendeld houdt.

3.2 Besluit

In deze sectie schetsten we het programmeermodel dat de basis vormt voor de aanpak voor statische verificatie van de afwezigheid van dataraces. We verwijzen naar de thesis voor een beschrijving van hoe we vanuit dit programmeermodel komen tot de aanpak voor statische verificatie, alsook van de uitbreidingen die deadlocks en hoog-niveau races op structuren bestaande uit meerdere objecten voorkomen.

4 Besluit van de thesis

In deze sectie formuleren we een besluit voor deze thesis.

In deze thesis stellen we een aanpak voor voor de formele verificatie van de afwezigheid van bepaalde fouten in gelijktijdige objectgeoriënteerde programma's. De thesis draagt bij tot het onderzoek naar manieren om het gemakkelijker te maken voor ontwikkelaars van programmatuur om correcte gelijktijdige programma's af te leveren, en in het bijzonder naar hulpmiddelen voor het verifiëren dat een op gedeeld geheugen gebaseerd gelijktijdig imperatief programma vrij is van dataraces en deadlocks. De prototype-implementatie van het hulpprogramma werd met succes gebruikt om een aantal kleine gelijktijdige programma's te verifiëren; niettemin

is verder werk nodig om te komen tot een hulpmiddel dat met positief resultaat ingezet kan worden in een software-ontwikkelingsproject van normale grootte. De belangrijkste problemen zijn de volgende:

- Het programmeermodel moet uitgebreid worden om meer programmeerpatronen toe te laten. Om de toepasbaarheid van het programmeermodel in te schatten, heeft de auteur een cursorisch onderzoek gedaan van het vrij beschikbare gelijktijdige Java-programma Azureus. Het bleek dat dit programma in zijn bestaande toestand niet voldoet aan het programmeermodel. Bijvoorbeeld, sommige objecten worden beschermd door zelf geïmplementeerde grendels in plaats van de in Java ingebouwde grendels. Daarnaast worden sommige statische velden geïnitialiseerd in de hoofdmethode in plaats van in de statische initialisatieroutine. Het is weliswaar duidelijk dat het programmeermodel uitgebreid moet worden; echter, de patronen opgemerkt door de auteur lijken slechts vrij beperkte uitbreidingen te vereisen. Niettemin is het mogelijk dat er andere vaak voorkomende patronen zijn die meer uitgebreide wijzigingen aan het programmeermodel vereisen, of, het zou kunnen blijken dat geen enkel bevattelijk programmeermodel alle patronen kan vatten die door programmeurs gebruikt worden.

- De annotatielast moet verminderd worden. De aanpak vereist dat programmeurs hun programma annoteren om het programmeermodel te instantiëren en om modulaire verificatie mogelijk te maken. Het annoteren van een programma is momenteel te arbeidsintensief. Vaak voorkomende patronen moeten geïdentificeerd worden en voor die patronen moeten beknopte notaties voorzien worden. Daarnaast zijn hulpmiddelen nodig om annotaties automatisch af te leiden uit de code, om het mogelijk te maken de aanpak te introduceren in een bestaand project.

- Het vermogen en de snelheid van de stellingenbewijzer moet waarschijnlijk verhoogd worden. Ervaring toont aan dat voor complexe methodes met complexe annotaties de stellingenbewijzer er soms niet in slaagt een geldige verificatieconditie te bewijzen. In de voorgekomen gevallen was het steeds mogelijk dit op te lossen door bijkomende annotaties toe te voegen, die als lemma's dienen voor de stellingenbewijzer. Bovendien duurt verificatie van een enkele methode soms een halfuur. Dit maakt het waarschijnlijk onmogelijk een realistisch programma dagelijks 's nachts te verifiëren. Anderzijds is het dankzij de modulariteit van de aanpak voldoende na een wijziging die methodes opnieuw te verifiëren waarop de wijziging impact had. In een goed gestructureerd programma met lage koppeling zou dit de verificatielast beduidend moeten verminderen.

We hebben onze aanpak gebaseerd op de huidige stand van de technologie wat betreft modulaire verificatie van sequentiële programma's met behulp van een

automatische stellingenbewijzer. Dit is één verificatie-aanpak naast vele andere: het interactief bewijzen van stellingen, abstracte interpretatie, modelcontrole, typecontrole, testen, en hybride aanpakken. Deze maken allemaal verschillende compromissen volgens de verschillende assen: snelheid van verificatie, graad van zekerheid, annotatielast, en verscheidenheid aan eigenschappen die geverifieerd kunnen worden. Automatische stellingenbewijzers scoren goed op het vlak van graad van zekerheid en verscheidenheid aan eigenschappen, die het meest interessant zijn vanuit een onderzoeksoogpunt. Vanuit een praktisch oogpunt zijn de andere assen minstens even belangrijk. Er wordt belangrijke vooruitgang geboekt in hybride technologie die modelcontrole, het bewijzen van stellingen, en/of abstracte interpretatie combineren, met technieken zoals abstractieverfijning en invarianten-op-aanvraag, waarbij de sterktes van elke aanpak gecombineerd worden.

Naast de onderliggende verificatietechnologie is de andere belangrijke component van een programmaverificatie-aanpak het programmeermodel. Het bepaalt welke ontwerp- en programmeerpatronen ondersteund worden door de aanpak. Er is recentelijk grote vooruitgang geboekt op het vlak van programmeermodellen. Eigendomssystemen maken abstractie mogelijk in het geval dat er meerdere referenties kunnen zijn naar gegeven objecten; op zichtbaarheid gebaseerde invariantsystemen maken niet-hiërarchische afhankelijkheden mogelijk. Ondersteuning voor toestandsabstractie in de vorm van modelvelden en gegevensgroepen, of dynamische kaders, is verschenen. Gelijkaardige ontwikkelingen gebeuren in de scheidingslogica. We hebben een eigendomssysteem gebruikt in onze aanpak, en een toestandsabstractie-benadering gebaseerd op inspectormethodes bijgedragen.

Op het gebied van programmeermodellen voor de verificatie van gelijktijdige imperatieve programma's worden eigendomssystemen gecombineerd met neem-aan-garandeer-redeneerpatronen om zowel modulariteit als redeneervermogen te realiseren.

Deze thesis spitst zich toe op objectgeoriënteerde programma's; echter, de kernideeën zijn toepasbaar op op gedeeld geheugen gebaseerde imperatieve programma's in het algemeen. Bijvoorbeeld, de aanpak voorgesteld voor Java kan waarschijnlijk vrij gemakkelijk overgedragen worden naar een Java-achtige deeltaal van C. Het volgen van een wijzer zou vereisen dat de wijzer behoort tot de toegangsverzameling van de draad. Het vrijgeven van een wijzer zou haar uit de toegangsverzameling verwijderen. Een dynamisch aangemaakte instantie van een structtype zou gedeeld kunnen worden met bescherming door een grendel op voorwaarde dat de struct een veld heeft dat een grendel bevat.

Nog een (misschien vooral historisch) belangrijke op gedeeld geheugen gebaseerde imperatieve programmeertaal is Ada. De benadering van de thesis kan waarschijnlijk vrij eenvoudig toegepast worden op Ada om dataraces en deadlocks tussen draden (*taken* genoemd in Ada) te voorkomen. Een kleine complicatie is dat de voornaamste synchronisatieconstructie in Ada niet de grendelconstructie is

maar het zogenaamde *rendezvous*. Een rendezvous vindt plaats wanneer een taak een *ingang* van een andere taak oproept en de andere taak een *aanvaarding* van de ingang uitvoert. De oproepende taak wordt geblokkeerd tot de aanvaardende taak klaar is met het uitvoeren van het lichaam van het *aanvaardingsblok*. Rendezvous kan gemakkelijk geverifieerd worden door elke ingang te annoteren met een preconditie en een postconditie om op te geven welke overdracht van objecttoegankelijkheid plaatsvindt in het begin, resp. aan het einde van een rendezvous.

Het gelijktijdigheidsparadigma dat momenteel gebruikt wordt in systeemprogrammatuur, zoals ondersteund door de populairste programmeertalen, is op gedeeld geheugen gebaseerde gelijktijdigheid, met gebruikmaking van grendels voor synchronisatie. Dit beperkt de abstractiekloof tussen de programmeertaal en de apparatuur tot een minimum en maakt daarom maximale performantie en flexibiliteit mogelijk. Anderzijds is het programmeren in dit paradigma moeilijk en leidt het gemakkelijk tot fouten. Deze thesis draagt bij tot het verbeteren van deze situatie, maar meer werk is nodig, zowel op het gebied van het ondersteunen van het programmeren met grendels, en op het gebied van het vinden van alternatieve of complementaire gelijktijdigheidsparadigma's. Sommige van de paradigma's die overwogen worden, zijn: geprogrammeerd transactioneel geheugen, functioneel programmeren, en aanpakken gebaseerd op het uitwisselen van boodschappen zoals de actoren-aanpak. Sommige hiervan verbieden het delen van objecten in zijn geheel; andere vereisen nog steeds een programmeermodel zoals het model voorgesteld in deze thesis. Een belangrijke vraag is of het opgeven van gedeeld geheugen een meer economische oplossing zou zijn voor het dataracesprobleem dan het proberen te verifiëren van de afwezigheid van dataraces met een aanpak zoals die van deze thesis. Het antwoord op deze vraag hangt af van de kost van het verlies aan programmeergemak en performantie bij uitvoering dat desgevallend het gevolg is van het opgeven van gedeeld geheugen. Bijgevolg is er waarschijnlijk geen eenduidig antwoord op deze vraag.

Contents

1 Introduction **1**
1.1 Problem description 1
1.2 Proposed solution 3
1.3 Contributions 5
1.4 Other work done 6
1.5 Structure of the thesis 6

2 The Boogie Program Verifier **9**
2.1 Introduction 11
2.2 Overview 12
2.3 Spec# 15
2.4 BoogiePL 16
2.5 Translating CIL to BoogiePL programs 20
2.6 Verification condition generation 24
2.7 The Boogie programming methodology 25
2.7.1 Object invariants 26
2.7.2 Ownership 27
2.7.3 Method framing 28
2.7.4 Subclassing 28
2.7.5 Method contracts for virtual methods 29
2.8 Related work 30
2.9 Conclusion 31

3 Inspector Methods **33**
3.1 Introduction 35
3.2 Framing 36
3.3 Object invariants and ownership 37
3.3.1 Object invariants 38
3.3.2 Ownership 41
3.4 Multi-dependent inspector methods 44
3.5 Formal details 47

3.6 Subclassing 49
3.7 Related work 54
3.8 Future work 57
3.9 Conclusion 57

4 Verifying Concurrent Programs 59
4.1 Introduction 61
4.2 Preventing data races 63
4.2.1 Programming model 64
4.2.2 Program annotations 66
4.2.3 Static verification 69
4.3 Lock levels for deadlock prevention 71
4.3.1 Programming model 72
4.3.2 Program annotations 72
4.3.3 Static verification 75
4.4 Invariants and ownership 75
4.4.1 Programming model 76
4.4.2 Program annotations 77
4.4.3 Static verification 80
4.5 Immutable objects 80
4.6 Static fields and static initializers 82
4.6.1 Class initialization in Java 82
4.6.2 Programming model 84
4.6.3 **lock_before** acyclicity 86
4.6.4 Immutable class objects 87
4.7 Experience 87
4.8 Related work 88
4.9 Conclusion 90

5 Formal Development 91
5.1 Programming model 92
5.1.1 Programs 92
5.1.2 Program executions 95
5.1.3 Data-race-freedom 99
5.1.4 Non-interference 103
5.2 Static verification 104
5.2.1 Modular verification 104
5.2.2 Verification logic 105
5.2.3 Valid programs 107
5.2.4 Soundness 112

6 Conclusion 117
6.1 Future work 120

Chapter 1

Introduction

This introductory chapter first describes the problem that is addressed by this thesis, as well as the proposed solution. It then lists the research contributions described in this thesis and other research results obtained, and outlines the structure of the thesis.

1.1 Problem description

Computers and computer programs play an increasingly important role in almost every area of human activity. However, failures in computer systems caused by program errors are common. There is an increasing need for tools to verify the absence of errors in programs.

Many computer programs are *concurrent*, which means that the order in which the operations occur is not specified fully. This reflects either inherent concurrency in the inputs provided by the computer system's environment, or a choice by the programmer to relax the ordering requirements to increase parallelism and performance.

Concurrent computer programs are typically written using the concept of a *thread of control* (or *thread* for short). A thread is an entity that reads the program's instructions and executes them one by one. There is no concurrency among the operations performed by a single thread. Concurrency is introduced when either the environment or the program itself creates additional threads. When multiple threads exist, they all execute concurrently.

When two threads execute concurrently, it may happen that they attempt to access the same resource concurrently. The resources most often accessed by programs are memory locations. If both threads are simply reading the current value stored in the memory location, there is no problem. However, if one or both threads are replacing the existing value with a new value, this is called a data race.

If a data race occurs, this is most likely a programmer error. The programmer's intention is usually that the various threads perform atomic (that is, as-if-instantaneous) updates to the data sets stored by the program. A data set is typically too large to be stored in a single memory location. Also, the various memory locations that store the data set typically need to be both read and written as part of the same update. If a thread attempts to update a data set without waiting for other threads to finish updating the same data set, that is, without *synchronizing*, then the updates might interfere and the result is incorrect. Data races are a symptom of unsynchronized updates.

Programmers typically achieve synchronization between concurrent updates using the mechanism of *locks*. The programmer associates a lock with each data set. The program is written such that before performing an update on a data set, a thread first attempts to acquire the lock associated with the data set. Only one thread may hold a given lock at any one time. If another thread already holds the lock, the thread waits. The thread only releases the lock after the update is complete. This ensures that the memory accesses that constitute two concurrent updates of the same data set are never interleaved.

Locks prevent races if used correctly, but the correct use of locks is difficult. If a programmer forgets to acquire a lock, or acquires the wrong lock when accessing a data set, or if data sets are incorrectly identified and different locks are associated with different parts of a data set, a race ensues. Also, there is the potential of deadlocks. A group of threads are in a deadlock when each thread is waiting for another thread to release a lock. It follows that the deadlocked threads stop performing useful work and the program hangs.

The difficulty of writing correct concurrent programs is an important problem because many important classes of programs are concurrent. For example, most systems software such as operating systems, distributed software such as the software that runs the internet or other client-server software such as databases and application servers, and embedded software such as cellphone software are concurrent. Often, application programs, too, are concurrent. For example, recent games are concurrent to exploit multiple processors, and applications with a graphical user interface are concurrent to prevent the user interface from becoming unresponsive due to long-running operations.

Furthermore, there is an increasing requirement that programs be concurrent or more highly concurrent, for performance reasons. Computer microprocessor manufacturers are reaching the limit of the number of instructions that a single microprocessor can execute per unit of time. Therefore, to increase performance, programs must be written such that the work to be done is split across multiple microprocessors. Such programs must necessarily be concurrent. Future increases in computer performance depend critically on increasing the ability of programmers to deliver correct concurrent programs.

Finally, concurrency-related errors are difficult to find using existing approaches

to detect program errors, in particular testing. Testing means executing the program and checking that the results are correct. However, in the presence of concurrency, a single program may have very many different executions for given inputs, as a result of different thread interleavings. Testing can only check a small subset of the possible executions of a concurrent program. Since often concurrency-related errors cause failures only under very specific interleavings, the likelihood of finding such errors using testing is low.

1.2 Proposed solution

This thesis proposes an approach for verifying that a concurrent program does not contain certain common programming errors related to the use of locks. Specifically, the approach guarantees the absence of data races and deadlocks. It also guarantees compliance of the program with programmer-provided correctness criteria such as method contracts and object invariants. Finally, it guarantees that it is sound for programmers to reason sequentially about their program, except at synchronization points. The approach is modular, to increase scalability.

Under the proposed approach, programmers write their program according to a particular regime (or *programming model*) for the correct use of locks, defined in this thesis. Amongst other things, the regime requires that the program text be annotated in certain specific ways to make the programmer's intentions explicit and to allow modular verification. When a program module is finished, the programmer may then pass the program module to a tool that verifies compliance with the programming model, as well as compliance with programmer-specified correctness criteria. Compliance with the programming model guarantees that there are no data races or deadlocks and that sequential code may be reasoned about sequentially. The tool operates by generating verification conditions in the form of formulae of first-order logic, and passing them to an automatic theorem prover.

For concreteness, the approach is formulated in terms of the popular object-oriented concurrent imperative programming language Java (except for Chapter 2, which is formulated in terms of C#, which is essentially a superset of Java). As will be argued in the conclusion of this thesis (Chapter 6), the core ideas of the approach are applicable to shared-memory imperative programming languages in general. However, this thesis leverages the improved design of object-oriented languages compared to their non-object-oriented predecessors to achieve a more elegant verification approach. For example, not only is it easier to write elegant programs using virtual methods than using function pointers, such programs are also easier to verify.

The programming model is based on the notion of *object accessibility*, defined in the thesis. (Accessibility is distinct from *reachability*. The programming model does not impose restrictions on the flow of object references; it only restricts the

use of those object references.) At any point during program execution, each object is *accessible* to at most one thread. A thread may read or write a field of an object only if the object is accessible to the thread. Initially, an object is accessible to the thread that created it. Additionally, any thread may gain access to the object by locking it. However, to prevent a race between the creating thread and the thread that holds the lock, the model distinguishes between *shared* and *unshared* objects. An object is initially unshared. A thread may attempt to lock an object only if the object is shared. A thread that has access to an unshared object may share it by performing a *share operation* on it, as specified by the programmer in a program annotation. At this point, the object is no longer accessible to any thread until it is locked.

To prevent deadlocks, the programming model requires the programmer to define a partial order among shared objects. Each thread may acquire locks only in accordance with this partial order.

Several other approaches exist to verify race- and deadlock-freedom for multithreaded code. They range from generating verification conditions [23, 32, 36, 77, 1, 79], to type systems [16, 34]. (See Section 4.8 for an overview of related work.) However, this is the first sound approach that guarantees the soundness of sequential reasoning about sequential code, that verifies object invariants over multi-object data sets, and that supports ownership transfer of aliased objects.

We performed a validation of the approach by creating a prototype implementation as an extension of the Boogie program verifier [9] for sequential (i.e., non-concurrent) programs. We obtained indications that the approach works for useful, non-trivial programs, by using the implementation to verify a number of small concurrent programs, the largest of which is a chat server.

The approach of this thesis targets shared-memory imperative concurrent programs, where mutable data structures are shared freely between threads of control. Many, if not most, concurrent programs fall in this category. However, concurrent programming paradigms have been proposed where mutable data structures are never shared and hence data races cannot occur. Notable languages that support such paradigms are the functional languages Erlang and Concurrent Haskell, and the proposed SCOOP (Simple Concurrent Object-Oriented Programming) extension for the imperative programming language Eiffel.

In addition to preventing data races, the absence of shared mutable data structures between threads of control improves fault isolation and therefore robustness. For example, Erlang is being used successfully by Ericsson in telecom equipment with high availability requirements. However, languages that prevent shared data structures have not achieved mainstream status, presumably because programmers perceive them as too restrictive. To judge whether this perception is justified is beyond the scope of this thesis.

1.3 Contributions

The contributions of the thesis are the following:

- An extension of the Boogie approach [10] with improved support for modularity by allowing state abstraction using inspector methods. Inspector methods may depend on the fields of the receiver object as well as the state of owned objects. They may be abstract and may be overridden. They may have parameters and they may depend on the state of a parameter if specially marked.

- An extension of the Boogie approach with support for concurrent programs that use mutual exclusion locks. The approach prevents data races and guarantees the soundness of sequential reasoning about sequential code. The approach supports ownership transfer of aliased objects.

- Support for deadlock prevention. The partial order among objects is specified by assigning a *lock level* to an object when it is shared. Lock levels are values which may be created, manipulated and stored like other program values, except that statements that manipulate lock levels and variables that hold lock levels must be inside annotations.

- Support for immutable objects. To support immutable objects, a distinction is made between accessibility for reading and accessibility for writing. Any unshared object may be shared as lock-protected or as immutable.

- Support for static fields, static initializers, and static class invariants. The approach is sound with respect to Java and C#'s lazy class initialization semantics. By integrating the approach with the concurrency approach, sequential code can be reasoned about sequentially and class initialization deadlocks are prevented.

- A prototype implementation of the approach as an extension of the Boogie program verifier. The implementation is available on-line in binary form.

- Initial experience with the implementation on non-trivial programs, including a chat server. The experience provides preliminary indications that the approach is sufficiently flexible and expressive to be applied to non-trivial, useful concurrent programs.

- A formalization, including a soundness proof, of the core of the approach. The annotation syntax and the dynamics of the programming model for data race prevention are formalized and proven sound. Further, the verification condition generation rules are formalized and proven sound.

1.4 Other work done

Besides the research described in this thesis, the main research results obtained are:

- A programming model for programs that implement the *Iterator* design pattern using C# 2.0's *iterators* and *foreach loops*. See *Bart Jacobs, Erik Meijer, Frank Piessens, and Wolfram Schulte. Iterators revisited: Proof rules and implementation. In Jan Vitek and Francesco Logozzo, editors, Seventh International Workshop on Formal Techniques for Java-like Programs (FTfJP 2005), 2005* [44].

- The author contributed to an approach for verification of compliance of a program with a stack walking-based sandboxing policy. See *Jan Smans, Bart Jacobs, and Frank Piessens. Static verification of Code Access Security policy compliance of .NET applications. Journal of Object Technology, 5(3):35–58, April 2006* [82].

1.5 Structure of the thesis

The remainder of the thesis consists of a background chapter, two core chapters, a formal development chapter, and a concluding chapter. The background chapter and each of the core chapters contain a mostly verbatim inclusion of a paper that is put into context by a preamble.

Chapter 2, *The Boogie Program Verifier*, introduces the general program verification framework within which we integrated our contributions. The specific program verifier described operates on programs written in *Spec#*, which is an extension of the Java-like object-oriented programming language C# with syntax for annotations that specify programmer-declared correctness criteria and that aid verification. The chapter describes the architecture of the verifier, which is two-tiered: the Spec# input program is first translated to an intermediate programming language called *BoogiePL*, and in a second stage the BoogiePL program is translated to first-order logic verification conditions. This chapter was derived from *Mike Barnett, Bor-Yuh Evan Chang, Robert DeLine, Bart Jacobs, and K. Rustan M. Leino. Boogie: A modular reusable verifier for object-oriented programs. In Proceedings of the Fourth International Symposium on Formal Methods for Components and Objects (FMCO 2005), volume 4111 of LNCS. Springer, 2006* [9].

Chapter 3, *Inspector Methods*, describes the state abstraction approach. The approach of this chapter allows specially marked methods, called *inspector methods*, to be used in method contracts and elsewhere. Restrictions are imposed to ensure that inspector methods can be interpreted as mathematical functions soundly. The issue of interaction with method effect framing is described and a

solution presented. Under certain conditions, inspector methods may themselves call other inspector methods. Also, inspector methods may depend on the state of multiple objects. Inheritance is supported fully. The approach is integrated with the Boogie object invariant and ownership approach. The chapter concludes with a description of related work. This chapter was presented at FTfJP 2006 (*Bart Jacobs and Frank Piessens. Verification of programs with inspector methods. In Elena Zucca and Davide Ancona, editors, Eighth International Workshop on Formal Techniques for Java-like Programs (FTfJP 2006), 2006* [45]).

Chapter 4, *Verifying Concurrent Programs*, describes the approach for verification of concurrent programs. Data races are prevented using access sets and by distinguishing between shared and unshared objects. Deadlocks are prevented by allowing the programmer to create partially ordered lock levels and to assign a lock level to each shared object. The approach is integrated with the Boogie object invariant and ownership system. Immutable objects need not be protected by a lock. Static class invariants and class initialization are supported. The chapter concludes with a description of experience and related work. This chapter was originally presented at ICFEM 2006 and published as *Bart Jacobs, Jan Smans, Frank Piessens, and Wolfram Schulte. A statically verifiable programming model for concurrent object-oriented programs. In Proceedings of the Eighth International Conference on Formal Engineering Methods (ICFEM2006), volume 4260 of LNCS. Springer, 2006* [48]. It was later presented as an invited talk by Wolfram Schulte and the author at the *First Workshop on Multithreading in Hardware and Software: Formal Approaches to Design and Verification (TV 06), Seattle, August 21-22, 2006.*

Chapter 5, *Formal Development*, provides a formalization of the core of the approach for verification of concurrent programs. A syntax for annotated programs is defined, and an operational semantics is given for execution of well-formed programs. To formalize the programming model, the notion of *legal program states* that comply with the programming model is defined and absence of data races is proven for programs that reach only legal states. Then, to formalize the static verification approach, a *verification logic* is introduced as well as a notion of *program validity* based on verification conditions in the verification logic. It is shown that valid programs reach only legal states and therefore are data-race-free.

Chapter 6 offers concluding remarks, summarizes the contributions of the thesis and identifies a number of avenues for future work.

Chapter 2

The Boogie Program Verifier

Preamble

This chapter describes the architecture of Boogie, a state-of-the-art program verifier for object-oriented programs. Whereas the ideas in this chapter are not contributions of this thesis, they establish the framework that forms the context for the contributions of this thesis.

This chapter contains the following paper: *Mike Barnett, Bor-Yuh Evan Chang, Robert DeLine, Bart Jacobs, and K. Rustan M. Leino. Boogie: a modular reusable verifier for object-oriented programs. In de Boer, F.S., Bonsangue, M.M., Graf, S., de Roever, W.-P. (eds.), Proceedings of the Fourth International Symposium on Formal Methods for Components and Objects (FMCO 2005), volume 4111 of LNCS, Springer-Verlag, 2006* [9].

For this chapter, a section on the Boogie programming methodology was added. Also, some text that is less relevant to the thesis and to which the author did not contribute, such as text on the invariant inference approach or the theorem proving mechanics, was dropped from the paper, and a few other editorial changes were made.

The author co-authored the paper. Specifically, he was the main author of the BoogiePL section (Section 2.4) and the bytecode translation section (Section 2.5).

Also, during his two internships at Microsoft Research in Redmond, Washington, United States, the author contributed to the implementation. His contributions include:

- Implementation of run-time checking of object invariants, ownership, and safe concurrency (see Chapter 4), including immutable objects, reader-writer locks, and conditional critical regions.

- Implementation of Spec# loop invariants support (parsing, type checking,

encoding in bytecode, emission of run-time checks, decoding from bytecode, emitting to BoogiePL).

- Design and implementation of parsing, type checking, and run-time checking of loop invariants for **foreach** loops. See [44].
- Design and implementation of [*StrictReadonly*] fields support. A [*StrictReadonly*] field can be seen as a function of object identity rather than of object state (i.e. the heap). This allows more programs to be verified.
- Improved error reporting when verifying BoogiePL programs directly.
- Participation in design discussions.
- Many bug fixes.

See also the discussion of SpecLeuven in the preamble of Chapter 4.

The Boogie Program Verifier

Abstract

A program verifier is a complex system that uses compiler technology, program semantics, property inference, verification-condition generation, automatic decision procedures, and a user interface. This chapter describes the architecture of a state-of-the-art program verifier for object-oriented programs.

2.1 Introduction

A program verifier is built from a number of complex pieces of technology: a source programming language, its usage rules and formal semantics, a logical encoding suitable for automatic reasoning, abstract domains for program analysis and property inference, decision procedures for discharging proof obligations, and a user interface that lets a user understand the results of the verification process. Dealing with these complexities, like other software engineering problems, requires a modular architecture with well established interface boundaries.

In this paper, we describe the architecture of Boogie, a state-of-the-art program verifier for verifying Spec# programs in the object-oriented .NET framework. Internally, Boogie is structured as a pipeline performing a series of transformations from the source program to a verification condition (VC) (see Fig. 2.1). The novel aspects of the Boogie architecture include the following:

1. *Design-Time Feedback.* Boogie (together with the Spec# compiler) is integrated with Microsoft Visual Studio to provide design-time feedback in the form of red underlinings that highlight not only syntax and typing errors but also semantic errors like precondition violations.

2. *Distinct Proof Obligation Generation and Verification Phases.* The Boogie pipeline is centered around intermediate representations in BoogiePL [21] (Sec. 2.4), a language tailored for expressing proof obligations and assumptions. BoogiePL serves a critical role in separating the generation of proof

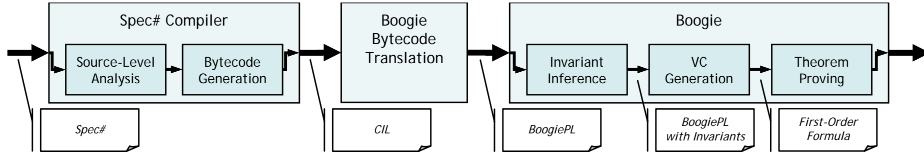


Figure 2.1: The Boogie pipeline.

obligations from the semantic encoding of the source program and the proving of those obligations. This separation has been critical in the simultaneous development of the object-oriented program verification methodology and the core verification technology.

3. *Abstract Interpretation and Verification Condition Generation.* Boogie performs loop-invariant inference using abstract interpretation and generates verification conditions to be passed (Sec. 2.6) to an automatic theorem prover. This combination allows Boogie to utilize both the precision of verification condition generation (that must necessarily be lost in an abstraction) and the inductive invariant inference of abstract interpretation (that simply cannot be obtained with a concrete model).

2.2 Overview

In this introductory section, we give an overview of Boogie's architecture. The rest of the paper provides more details of each architectural component.

Source language.

The Spec# language is a superset of C#, adding specification features (i.e., *contracts*) such as pre- and postconditions and object invariants [12]. Spec# prescribes static type checks beyond those prescribed by C# and introduces dynamic checks for the specified contracts. The compiler performs the static type checking and emits the dynamic checks as part of the target code. Boogie makes use of the type properties enforced by the compiler and attempts statically to prove that the dynamic checks will always succeed. Boogie thus checks for error conditions defined by the virtual machine, such as array bounds errors and type cast errors, and error conditions specified by user supplied contracts, such as precondition violations. To ensure soundness of the verification, Boogie additionally checks for error conditions defined by the programming methodology[1] [8, 61, 14, 62, 64].

[1]This thesis presents a version of the Boogie programming methodology extended with state abstraction in Chapter 3, and a version extended with concurrency support in Chapter 4.

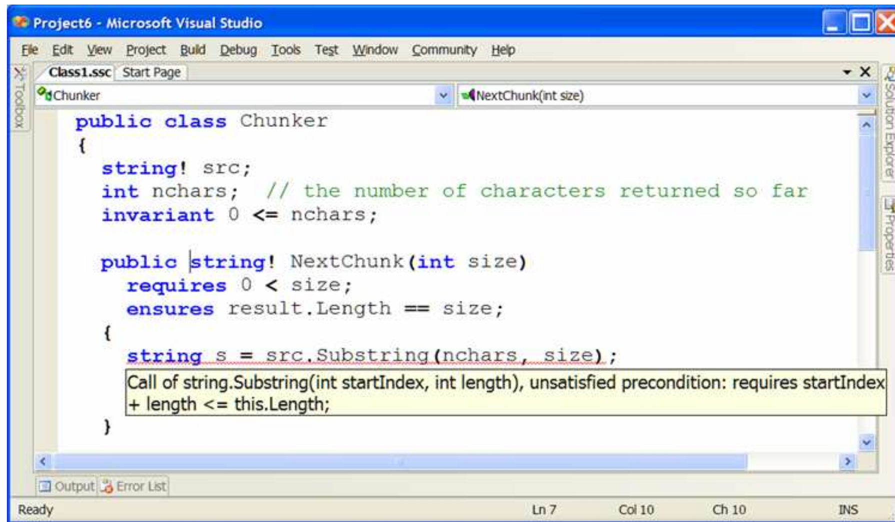


Figure 2.2: Design time feedback of verification errors within Microsoft Visual Studio 2005. The red squiggly under the call to *Substring* indicates an error. The hover text shows the error to be a precondition violation.

As depicted in Fig. 2.1, Spec# programs are compiled into CIL, the executable format of the .NET virtual machine. Boogie starts with an abstract syntax tree (AST) for this CIL, which it either gets directly from the compiler or reconstructs from reading a compiled `.dll` or `.exe` file. The latter is the more conventional mode of a program verifier and allows batch processing. The former allows Boogie to run as part of compilation, which enables a clean integration with Microsoft Visual Studio to provide design time feedback. This feedback shows up as red underlinings in the program text, and the user can get further information by rolling the cursor over these underlinings, which brings up some hover text that explains the problem (as shown in Fig. 2.2). To the authors' knowledge, Boogie is the first program verifier to provide such interactive design-time feedback.

Intermediate language.

The generation of verification conditions from source code involves a great number of verifier design decisions. By staging this process by first translating CIL into BoogiePL, the Boogie architecture separates the concerns of deciding how to encode source language features and their usage rules from the concerns of how to reason about control flow in the program. BoogiePL provides **assert** statements that encode proof obligations stemming from the source program, to be checked by the program verifier, and **assume** statements that encode properties

guaranteed by the source language and verification process, available for use in the proof by the program verifier. The architectural layering of verification condition generation via an intermediate program notation was used by ESC/Modula-3 [24] (cf. [57]), which made use of *guarded commands* whose semantics is given by *weakest preconditions* [25]. This architecture was sharpened by ESC/Java [33], which defined and staged the translation further [67].

Verification condition generation involves not just the executable program statements in the source language, but also other declarations of the source program and properties guaranteed by the source language. In the aforementioned ESC tools, the logical encoding of these additional properties, called the *background predicate*, was produced separately from the intermediate program notation and fed directly to the theorem prover. BoogiePL innovates further by allowing the background predicate to be encoded as part of the intermediate program. That is, BoogiePL includes declarations for mathematical functions and axioms. Consequently, the translation of CIL culminates in a BoogiePL program that encodes the entire proof task—in fact, properties of Spec# and the source program are no longer used after this point in Boogie's pipeline, except when mapping errors back to line numbers in the source text.

Like other languages, BoogiePL can be printed as and parsed from a textual representation. This feature has been the most important vehicle in debugging and experimenting with the semantic encoding of Spec#. For example, it is often convenient to manually perform small changes to the BoogiePL program without having to modify the Spec# compiler and/or the bytecode translator.

This strong interface boundary between the bytecode translation and the rest of Boogie's pipeline also makes it possible for other program verifiers to reuse Boogie's VC generation, simply by encoding their proof obligations as BoogiePL programs. As such, BoogiePL can also be viewed as a high-level front-end to a theorem prover.

Verification conditions.

From the BoogiePL program, Boogie then generates verification conditions. There are many logically equivalent ways of expressing the verification conditions, and which way is chosen can have a dramatic impact on the performance of the underlying theorem prover. Boogie performs a series of transformations on the program, essentially producing one snippet of the verification condition from each basic block of the BoogiePL procedure implementation being verified [11]. The verification condition is represented as a formula in first-order logic and arithmetic. It is then passed to a first-order automated theorem prover to determine the validity of the verification condition (and thus the correctness of the program).

The verification conditions are encoded in such a way as to make it possible to reconstruct from a failed proof an error trace (i.e., an execution path through the procedure leading to a proof obligation that the theorem prover is unable to

```
public class Example {
  int x;
  string! s;
  invariant s.Length >= 12;
  public Example(int y) requires y > 0; { ... }
  public static void M(int n) {
    Example e = new Example(100/n);
    int k = e.s.Length;
    for (int i = 0; i < n; i++) { e.x += i; }
    assert k == e.s.Length;
  }
}
```

Figure 2.3: An example Spec# class. Its BoogiePL translation is shown in Fig. 2.4.

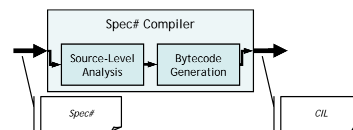
establish [59]). The bytecode translator performs enough bookkeeping to map the BoogiePL error trace back into a Spec# error trace, much like a compiler performs enough bookkeeping for a source-level debugger to operate from compiled code. Note: typically, a failed proof indicates an error in the program or some missing condition in a contract, but due to incompleteness in the theorem prover, there is also the possibility of spurious error reports. For example, an error may be reported if the correctness of an execution path is based on some fact of integer arithmetic that the theorem prover is unable to prove. Also, if the theorem prover runs out of some limited resource, such as the allotted time or space, that event is reported.

Theorem prover.

Boogie can generate verification conditions for the off-the-shelf theorem prover Simplify [22], as well as for Zap [6], a set of decision procedures being developed at Microsoft Research. Most of the Spec# team's Boogie experience has been with Simplify, but they expect to shift their use toward Zap if and when it outperforms Simplify for their purposes. The Boogie architecture makes it fairly easy to retarget the final step of the VC generation to a new theorem prover.

2.3 Spec#

Figure 2.3 shows a synthetic example program to highlight some of the features of Spec#; a more detailed introduction is found in the Spec# overview paper [12]. The example shows one class, called *Example*, which contains two fields, an object invariant, a constructor, and a method.



The body of the method allocates a new *Example* object. The actual argument to the constructor, $100/n$, contains a potential division-by-zero error. The loop repeatedly increments the x field of the newly allocated object by various amounts. The code also saves the length of the string $e.s$ and later checks, using an assert statement, that it is unchanged by the loop.

Spec# incorporates a non-null type system [27]; the type `string!` means the field s can never hold the value `null`. The authors have found this to be the most common specification in object-oriented programming.

The Spec# compiler generates standard .NET assemblies. A .NET assembly contains bytecode in the form of method bodies within type definitions and *meta-data* for describing extra-runtime features of the types and their members. The meta-data format allows *custom attributes*, which are arbitrary user-defined data. The Spec# compiler uses these to encode specifications. Due to the limitations of the meta-data format, specifications are persisted as *serialized ASTs*. The same meta-data format is used to store *out-of-band* specifications. These are Spec# specifications for types and methods that are already defined in third-party assemblies. For instance, the Spec# distribution [83] provides out-of-band contracts for the two most central assemblies in the .NET Base Class Library (BCL), `mscorlib.dll` and `System.dll`. Both the Spec# compiler and Boogie have the ability to weave together an assembly and its out-of-band specification so that it appears as if the contracts were natively present in the original assembly. One of the key benefits is that client code, which generally is heavily dependent on the BCL, receives warnings/errors related to incorrect usage of the library APIs. This feature has been critical in obtaining a usable development system with contracts.

2.4 BoogiePL

BoogiePL [21] is an effective intermediate language for verification condition generation for object-oriented programs because it lacks the complexities of a full-featured object-oriented programming language, while also introducing features of the target logic. As a result, it distributes the complexity of verification condition generation over two well-defined phases, each of which is significantly less complex than the whole. Compared with Spec#, BoogiePL retains the following features: procedures (but not methods), mutable variables, and pre- and postconditions. On the other hand, it lacks the following complications: expressions with side effects, a heap with objects, classes and interfaces, call-by-reference parameter passing, and structured control-flow. It introduces the following features for modeling: constants, function symbols, axioms, non-deterministic control-flow, and the notion of “going wrong”.

Figure 2.4 shows the translation of the Spec# example given in Fig. 2.3. While we give details on how this translation is obtained in Sec. 2.5, we observe some salient features of BoogiePL here. BoogiePL looks somewhat like a high-

level assembly language in that the control-flow is unstructured but the notions of statically-scoped locals and procedural abstraction are retained; however, intraprocedural control-flow is given by a non-deterministic **goto**. Also, observe that the heap has been made explicit with the global variable *Heap* and similarly the implicit receiver object of the method is now an explicit parameter *this*.

A BoogiePL *program* consists of a *theory* that is used to encode the semantics of the source language, followed by an *imperative part*. We show the abstract syntax for BoogiePL; for punctuation and other concrete details, see [21].

$$\mathit{program} ::= \mathit{typedecl}^* \ \mathit{symboldecl}^* \ \mathit{axiom}^* \ \mathit{vardeclstmt}^* \ \mathit{proc}^* \ \mathit{impl}^*$$

We use the meta-level symbols *, $^+$, $^?$ to indicate a sequence, a nonempty sequence, and an optional syntactic entity, respectively, use $|$ for alternatives, and use $\langle\cdot\rangle$ for grouping.

A theory consists of *type declarations*, *symbol declarations*, and *axioms*.

$$\begin{aligned}
&\mathit{typedecl} ::= \textbf{type}\ \mathit{typename}\ ;\\
&\mathit{symboldecl} ::= \mathit{constdecl} \mid \mathit{functiondecl}\\
&\mathit{constdecl} ::= \textbf{const}\ \mathit{var} : \mathit{type}\ ;\\
&\mathit{type} ::= \textbf{bool} \mid \textbf{int} \mid \textbf{ref} \mid \textbf{name} \mid \textbf{any} \mid \mathit{typename} \mid \mathit{arraytype}\\
&\mathit{arraytype} ::= [\ \mathit{type}\ ,\ \mathit{type}\ ]\ \mathit{type}\\
&\mathit{functiondecl} ::= \textbf{function}\ \mathit{function}\ (\ \mathit{type}^*\ )\ \textbf{returns}\ (\ \mathit{type}\ )\ ;\\
&\mathit{axiom} ::= \textbf{axiom}\ \mathit{expr}\ ;
\end{aligned}$$

BoogiePL has types and the type checker enforces that every expression is properly typed. However, all type information is erased during the translation into verification conditions. The reason for having the types is to improve readability by expressing intent and to catch simple errors. However, any expression may be cast to type **any** and thence to any other type; just as types are erased in verification conditions, so are casts. In addition to built-in types like **bool** and **int**, BoogiePL supports user-defined types (*typename*) and arrays (*arraytype*). The types used to index into arrays can be any types, not just integers. For brevity, we show only 2-dimensional arrays.

BoogiePL's expressions include boolean, reference, and integer literals and arithmetic and first-order logical operators:

$$\begin{aligned}
&\mathit{expr} ::= \mathit{literal} \mid \mathit{var} \mid \mathit{unop}\ \mathit{expr} \mid \mathit{expr}\ \mathit{binop}\ \mathit{expr} \mid \mathit{expr}\ [\ \mathit{expr}\ ,\ \mathit{expr}\ ]\\
&\quad \mid \mathit{funapp} \mid \mathit{quant} \mid \textbf{cast}\ (\ \mathit{expr}\ ,\ \mathit{type}\ ) \mid \textbf{old}\ (\ \mathit{expr}\ )\\
&\mathit{literal} ::= \textbf{false} \mid \textbf{true} \mid \textbf{null} \mid \mathit{integer}\\
&\mathit{binop} ::= \Leftrightarrow \mid \Rightarrow \mid \vee \mid \wedge \mid <: \mid \leqslant \mid < \mid \neq \mid = \mid + \mid - \mid * \mid / \mid \%\\
&\mathit{unop} ::= - \mid \neg\\
&\mathit{funapp} ::= \mathit{function}\ (\ \mathit{expr}^*\ )\\
&\mathit{quant} ::= (\forall\ \mathit{vardecl}^*\ \mathit{trigger}^*\ \bullet\ \mathit{expr}\ ) \mid (\exists\ \mathit{vardecl}^*\ \mathit{trigger}^*\ \bullet\ \mathit{expr}\ )\\
&\mathit{vardecl} ::= \mathit{var} : \mathit{type}\ \langle \textbf{where}\ \mathit{expr}\rangle^?\\
&\mathit{trigger} ::= \{\ \mathit{expr}^+\ \}
\end{aligned}$$

```
const System.Object : name;
const Example : name;
axiom Example <: System.Object;
function typeof(obj : ref) returns (class : name);

const allocated : name;
const Example.x : name;
const Example.s : name;

var Heap : [ref, name]any;

function StringLength(s : ref) returns (len : int);

procedure Example..ctor(this : ref, y : int);
  requires ... ∧ y > 0;   modifies Heap;   ensures ...;

procedure Example.M(n : int);
  requires ...;   modifies Heap;   ensures ...;

implementation Example.M(n : int)
{
  var e : ref where e = null ∨ typeof(e) <: Example;
  var k : int, i : int, tmp : int;

  Start :
    assert n ≠ 0;
    tmp := 100/n;
    havoc e;
    assume e ≠ null ∧ typeof(e) = Example ∧ Heap[e, allocated] = false;
    Heap[e, allocated] := true;
    call Example..ctor(e, tmp);

    assert e ≠ null;   k := StringLength(cast(Heap[e, Example.s], ref));
    i := 0;
    goto LoopHead;

  LoopHead :
    goto LoopBody, AfterLoop :

  LoopBody :
    assume i < n;
    assert e ≠ null;
    Heap[e, Example.x] := cast(Heap[e, Example.x], int) + i;
    i := i + 1;
    goto LoopHead;

  AfterLoop :
    assume ¬(i < n);
    assert e ≠ null;   assert k = StringLength(cast(Heap[e, Example.s], ref));
    return;
}
```

Figure 2.4: A simplified version of the BoogiePL resulting from translation of the *Example* class in Fig. 2.3.

For use in procedure postconditions and implementations, the expression **old**(E) refers to the value of E in the procedure's pre-state. The **where** clause in a variable declaration postulates a unary constraint on the variable's value (like a type qualifier). Triggers are for use by the underlying theorem prover in deciding how to instantiate universal quantifiers [22].

The imperative part of a BoogiePL program consists of global variable declarations, procedure headers (*procedures*), and procedure implementations (*implementations*).

vardeclstmt ::= **var** *vardecl** ;
proc ::= **procedure** *procname* (*vardecl**) $\langle$**returns** (*vardecl**)$\rangle^?$
$\langle$**free**$^?$ **requires** *expr* ;$\rangle^*$ $\langle$**modifies** *var** ;$\rangle^*$ $\langle$**free**$^?$ **ensures** *expr* ;$\rangle^*$
implbody$^?$
impl ::= **implementation** *procname* (*vardecl**) $\langle$**returns** (*vardecl**)$\rangle^?$
implbody
block ::= *label*: *cmd** *transfercmd*
implbody ::= { *vardeclstmt** *block*$^+$ }

As a syntactic sugar, a procedure header can have an optional implementation body, which has the same effect as an implementation declaration with the same name. While many languages have named out-parameters and an anonymous return value, BoogiePL simply allows multiple return values; they are all named as out-parameters in the **returns** clause.

An implementation body consists of a sequence of local variable declarations, followed by a sequence of *blocks*. An implementation starts at the block listed first, in a state where the procedure's preconditions hold and where global and local variables and in-parameters have values that satisfy their respective **where** clauses.

A block has a label and a sequence of commands, followed by a control transfer command.

cmd ::= *passive* | *assign* | *call*
passive ::= **assert** *expr* ; | **assume** *expr* ;
assign ::= *var* $\langle$[*expr* , *expr*]$\rangle^?$:= *expr* ; | **havoc** *var*$^+$;
call ::= **call** *var** := *procname* (*expr**) ;
transfercmd ::= **goto** *label*$^+$; | **return** ;

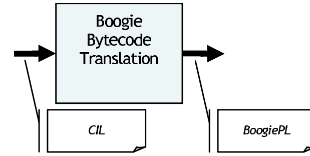
The **assert** and **assume** commands indicate conditions to be checked or used, respectively, in the verification. If the given expression evaluates to **true**, then each of these commands proceeds like a no-op. If the condition evaluates to **false**, the **assert** command *goes wrong*, which is a terminal failure. For the **assume** command, if the condition evaluates to **false**, one is freed of all subsequent proof obligations, thus indicating a terminal success. The **assume** command, which is

known as a *partial command* [72] or *miracle* (cf. [5]), is a crucial ingredient when encoding verification problems as programs (cf. [57]). The **havoc** command assigns an arbitrary value to each indicated variable; when present, the variable's **where** clause constrains this value. The **goto** command jumps nondeterministically to one of the indicated blocks. The **return** command ends the implementation. It goes wrong if the procedure's non-**free** postconditions are not satisfied; otherwise, the procedure implementation terminates successfully.

The **call** command is defined in terms of the specification of the procedure being called. It goes wrong if the procedure's non-**free** preconditions do not hold. Otherwise, the state after the **call** command satisfies the procedure's postconditions and the **where** clauses of the procedure's out-parameters. Specifically, it is not assumed that the procedure implementation that gets executed is one of the implementations declared in the program.

2.5 Translating CIL to BoogiePL programs

In this section, we describe some key issues and design choices for the bytecode translator. These issues include encoding the heap, allocation, and fields, axiomatizing the Spec# type system, translating call-by-reference parameters, translating methods and method calls, and generating frame conditions.



The bytecode translator first transforms a method body, consisting of CIL instructions, into a *normalized AST*, which is essentially an enriched object model that augments CIL with contract features: non-null type annotations, **assert** and **assume** statements, loop invariants, method contracts, object invariants, and various custom attributes for the programming methodology. In a normalized AST, as in CIL, a method body contains no structured programming constructs such as **if** statements or **while** statements. Rather, a method body contains a sequence of labeled blocks, each of which contains a sequence of statements, some of which may be conditional or unconditional branch statements that specify the label of the target block.

One of the differences between a normalized AST and BoogiePL is that in the former, a statement may contain expressions that may have side effects and that may go wrong (such as method calls). For this reason, Boogie transforms the normalized AST into a *flattened AST*, where the values of expressions are assigned to evaluation stack slots and only evaluation stack slots appear as operands of expressions and statements. A flattened AST is suitable for walking the sequence of statements and generating BoogiePL commands based on the kind of statement, although a dataflow analysis is still needed to provide some flow-sensitive contextual information necessary for the translation of some statements.

In addition to inferring the CIL type of each local variable and evaluation stack

slot, the analysis attempts to track the following information:

- managed pointers (i.e., reference parameters or arguments for reference parameters or the pointers used when dealing with structs)
- method pointers (which appear when creating a delegate instance)
- type tokens and *System.Type* objects (which appear when reflecting over types)
- booleans (for which CIL code uses integers)

Encoding the heap, allocation, and fields.

Recall that BoogiePL has no built-in notion of a heap, object allocation, or fields. The translation models the heap as a BoogiePL global two-dimensional array, named *Heap*, that maps an object reference o and a field name f to the current value of $o.f$ (as shown in Fig. 2.4) [18]. Field names encountered during translation are emitted as unique constants, whose names are qualified by the name of the declaring class.

A 2-dimensional heap [76] is used rather than one 1-dimensional "heap" per field (cf. [24, 57]), because the encoding of the modular verification methodology quantifies over field names (see frame conditions below).

Object allocation is modeled by adding an extra boolean field called allocated to each object, which indicates whether the object has been allocated. Allocating an object consists of choosing an object that has not yet been allocated and setting its allocated bit, see Fig. 2.4 (cf. [40, 24, 57]). In managed code like Spec#, all objects reachable from the program are allocated. This property is important for proving that newly allocated objects are distinct from previously allocated objects, which is crucial for reasoning about object state updates. Allocatedness information is emitted in the form of procedure preconditions, frame conditions, and loop invariants, as well as axioms that state that objects reachable from allocated objects are allocated (cf. [66]). This statement is complicated somewhat by the fact that a path between two objects may pass through one or more `structs`.

Static fields are stored in the heap, just like instance fields. In particular, fields are translated as follows:

instance field $o.f$	translates to	$\mathit{Heap}[o, \mathsf{C_f}]$
static field $C.g$	translates to	$\mathit{Heap}[\mathsf{TypeObject}(C), \mathsf{C_g}]$

where fields f and g are declared by class C and o is an expression of type C. A major advantage of storing static fields in the heap as opposed to, for example, separate global variables, is that one frame condition can govern both static and instance fields. It also allows a more uniform treatment of both kinds of fields by the modular verification methodology.

Axiomatizing the Spec# type system.

In order to model the semantics of Spec# type tests and typecasts, an axiomatization of the subtype relation on reference types is required. This axiomatization additionally helps in deriving object distinctness results, which reduces the number of inequalities that users need to include in method contracts, object invariants, loop invariants, etc.

It turns out to be a challenge to author a type axiomatization that is queried efficiently by the theorem prover. Therefore, the Spec# team is considering adding a decision procedure specifically for this purpose. Unfortunately, this choice would probably require that BoogiePL be made aware of the Spec# type system.

Translating call-by-reference parameters.

Both C# and Spec# support call-by-reference argument passing (reference parameters are marked `ref`). A call-by-reference parameter of type T takes as an argument not a value of type T but a pointer to a variable of type T. Accesses to the parameter are thus dereferences of the pointer.

BoogiePL does not support call-by-reference parameters directly. Since it does support both in- and out-parameters, reference parameters are modeled by performing copy-in/copy-out for the purposes of verification. In the Spec# program, when a variable x is passed as an argument to a reference parameter p, then in the BoogiePL program the value of x is passed as an argument to an in-parameter. At the start of the body of the callee, the in-parameter is copied into a local variable. Accesses to p in the Spec# program are translated into accesses of the local variable in the BoogiePL program. When the procedure completes, the local variable is copied into an out-parameter and at the call site the out-parameter is copied back into x.

Translating from IL introduces a snag: accesses to reference parameters appear as pointer dereferences, and the pointers being dereferenced are read from the parameter in some preceding instruction. As a result, it is impossible to tell by looking at the instruction itself which reference parameter is being referred to. This problem is solved by the bytecode translator's dataflow analysis mentioned above.

Using copy-in/copy-out is sound only if there is no aliasing amongst the actual arguments for reference parameters. Therefore, such aliasing is disallowed in Spec#. (This is not yet implemented in the compiler, but the Spec# team intends to impose enough restrictions on actual arguments that a simple syntactic check suffices to forbid such aliasing, cf. [78].)

Translating methods and method calls.

For each declaration of a method or method override, the bytecode translation generates a BoogiePL procedure. For each method implementation, it also generates

a BoogiePL implementation. Having a separate procedure per override permits specification refinement in subclasses.

For translating method calls, two cases are distinguished. When the exact target of a call can be determined (that is, the call is statically bound, such as for a non-virtual method or a base call), it is translated into a call to the associated procedure. When the call is dynamically bound, the call is translated into a call of an additional BoogiePL procedure that is generated for virtual methods. This makes it possible to use a slightly different specification for such calls, as required by the programming methodology [8].

Method framing.

In BoogiePL, the effect of a procedure is framed by its **modifies** clause. Specifically, a procedure may assign to a global variable only if the variable appears in the procedure's **modifies** clause. As mentioned above, the Spec# heap is modeled in BoogiePL as a global variable (observe the **modifies** clauses in Fig. 2.4 for *Heap*). Since almost all methods may potentially modify the heap (by creating a new object or assigning to a field), the heap appears in almost every procedure's **modifies** clause. However, this is clearly an overapproximation. Therefore, additional framing information is encoded in the BoogiePL program in the form of an extra postcondition on the procedure. This postcondition is known as the *frame condition*. The precise form of the frame condition depends on the modular verification methodology used; for example, for the original Boogie methodology [8], each procedure gets a frame condition of the following form:

$$
\begin{aligned}
(\forall o\colon \mathbf{ref}, f\colon \mathbf{name}\ \bullet\ & \\
& (o, f) \notin W \ \wedge\ \mathbf{old}(\mathit{Heap}[o, \mathsf{allocated}] \wedge \neg \mathit{Heap}[o, \mathsf{committed}]) \\
& \Rightarrow\ \mathit{Heap}[o, f] = \mathbf{old}(\mathit{Heap}[o, f]))
\end{aligned}
$$

where W are the locations listed in the Spec# method's `modifies` clause, and $\mathit{Heap}[o, \mathsf{committed}]$ is a special field introduced by the modular verification methodology related to an ownership model [8]. Essentially, the frame condition says that unless a given location satisfies certain criteria, it is guaranteed not to incur a net modification by the method call.

Loop framing.

In order to generate verification conditions [11] for a loop, such as the following:

$$\mathit{LoopHead}\colon \mathbf{assert}\ I;\quad S;\quad \mathbf{goto}\ \mathit{LoopHead};$$

it must be transformed into acyclic control flow that abstracts the behaviors of the loop (for soundness). Specifically, the loop above is transformed into the following

sequence of statements:

$$\begin{array}{l} x_1^0 := x_1; \ \ldots \ x_n^0 := x_n; \quad \textbf{assert } I; \\ \textbf{havoc } x_1, \ldots, x_n; \quad \textbf{assume } I; \\ S; \quad \textbf{assert } I; \quad \textbf{assume false}; \end{array}$$

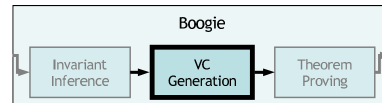
where S is the loop body (which may include commands that jump out of the loop) , $x_1, \ldots, x_n$ are the variables (global or local) updated by S, and $x_1^0, \ldots, x_n^0$ are fresh local variables. The predicate I serves as a loop invariant.

The transformation causes the loop body (as well as code paths that exit the loop) to be verified in all possible states that satisfy the loop invariant. The **assume false**; command indicates that a code path that does not exit the loop can be considered to reach terminal success at the end of the loop body, provided that the loop invariant has been re-established.

When the loop body updates the heap (which is the typical situation), the heap is **havoc**ed (i.e., assigned an arbitrary value) on entry to a loop. This abstraction results in a sound but gross overapproximation of the set of heap locations that may be modified by loop body executions. Some of the lost precision may be recovered by inference, but additionally *loop frame conditions* can be emitted that increase the precision of the verification and are guaranteed by the verification methodology. For example, one loop frame condition that is always added states that all objects that were allocated on entry to the loop are still allocated at the start of the current iteration:

$$(\forall o\colon \textbf{ref} \ \bullet \ \mathit{Heap}^0[o, \mathit{allocated}] \Rightarrow \mathit{Heap}[o, \mathit{allocated}])$$

2.6 Verification condition generation



A BoogiePL program encodes sets of program traces and proof obligations in those traces. Verification condition generation turns those proof obligations into first-order formulas. As already described, BoogiePL is intentionally *not* a structured programming language. That is, a BoogiePL program is a somewhat high-level way of specifying a control-flow graph whose nodes are *basic blocks*. In order to apply standard weakest-precondition calculus [25, 72] for verification condition generation, the Spec# team had to develop a method for computing first-order weakest preconditions of an unstructured program.

The method used, which the authors have described in detail elsewhere [11], produces one VC for every BoogiePL procedure implementation. It starts by transforming the implementation into some loop-free BoogiePL code that over-approximates the loops in the original. It then performs a *single assignment*

transformation (cf. [19, 35]), resulting in only *passive* code, that is, code without state changes. Finally, to encode the unstructured nature of the control flow, for every block A a boolean variable A_{ok} is introduced, defined to be *true* if every execution starting from A is correct (i.e., does not go wrong). For a block:

$$A\colon \mathit{PassiveCommands};\ \textbf{goto}\ B, C;$$

the *block equation* that defines A_{ok} is:

$$A_{ok} \;\Leftarrow\; \mathfrak{wp}(\mathit{PassiveCommands},\ B_{ok} \wedge C_{ok})$$

where $\mathfrak{wp}$ computes the weakest precondition of *PassiveCommands* with respect to the postcondition $B_{ok} \wedge C_{ok}$. The VC is then:

$$\mathit{Axioms} \;\wedge\; \mathit{BlockEqs} \;\;\Rightarrow\;\; \mathit{Start}_{ok}$$

where *Axioms* is the conjunction of the axioms in the BoogiePL program, *BlockEqs* is the conjunction of block equations (which thus encode the semantics of the passive BoogiePL code), and *Start* is the implementation's start block.

While translating the BoogiePL program into a verification condition, the back-end phase builds up a table mapping labeled subformulas to BoogiePL program elements. The back-end phase uses this table to translate the label output from the theorem prover into an error message in terms of the BoogiePL program [59]. Similarly, the front-end bytecode translation creates a table mapping BoogiePL program elements to Spec# program elements. This table allows it to take an error message on a BoogiePL program and generate an error message on the original Spec# program in terms that the programmer can understand.

2.7 The Boogie programming methodology

The Boogie program verifier performs *modular* verification. That is, it verifies a Spec# assembly (i.e., a set of classes and interfaces, that may refer to exported classes and interfaces of other assemblies through symbolic assembly references) using only the annotations (mainly method contracts) exported by the referenced assemblies. The validity of an assembly does not depend on the method bodies of referenced assemblies, nor on assemblies referenced transitively from the referenced assemblies, for example. Validity is compositional, in the sense that a program composed of valid assemblies does not go wrong. It follows that modifying an assembly without modifying the exported annotations does not require re-verification of client assemblies.

The degree of benefit gained from this modularity property depends on the number of degrees of freedom allowed by an assembly's exported annotations. This is related to the notions of *information hiding* and *abstraction*. The Boogie

program verifier has a number of features that help increase the level of information hiding and abstraction of an assembly's exported annotations. These are together called the Boogie *programming methodology*. In this section, we briefly introduce the original Boogie programming methodology [8], on which this thesis is based. Later versions of the methodology [61, 14, 62, 64] are not discussed or used in this thesis.

The Boogie programming methodology consists of a mechanism for abstracting over *object invariants*, and an *ownership system* for abstracting over the set of private auxiliary objects used internally by an object to help represent its state.

2.7.1 Object invariants

Often, the correctness of a method depends on *consistency properties* of the receiver object. However, in the verification condition generation strategy explained above, the only assumption emitted about the method's pre-state is the precondition. It is not desirable to explicitly state the consistency properties in the precondition, since this breaks information hiding. In the literature, the following approach has been proposed: if it is checked that a property holds in the post-state of each constructor and method of a class, then one may assume that it also holds in the pre-state of each method. However, this is unsound in the presence of re-entry: if a method call on an object o temporarily breaks the consistency of o and then performs a method call on an object p, and this call then performs a re-entrant call on o, then the re-entrant call unsoundly assumes that the receiver object is consistent.

The Boogie methodology solves this problem by allowing the programmer to mention an object's consistency properties in a method's precondition in an abstract way. Specifically, a boolean ghost field *inv* is added to each class. (A *ghost field* is a field that is used for static verification only and that is not used during execution.) Also, in each class the programmer may include a declaration of the form **invariant** I;, where I is an expression that may mention only the fields of **this**. I is called the class's *object invariant*. An object's *inv* field is initially false. The programming methodology imposes restrictions on updates of the *inv* field and other fields, such that the property holds that for any object o, if $o.inv$ is true, then $I[o/\mathbf{this}]$ holds (that is, the object invariant holds after substitution of o for **this**). It follows that if a method's precondition includes the condition $\mathbf{this}.inv$, the receiver object's invariant holds in the pre-state.

The restrictions on field updates are as follows:

- a field $o.f$, where $f \neq inv$, may be updated only if $o.inv$ is false.
- a field $o.inv$ may be updated only through special **pack** o; and **unpack** o; commands. (These are *ghost commands*, in that they update only ghost fields and have no effect on the program's behavior.) **pack** o; first checks

that o's object invariant holds. If it does not, the program goes wrong, i.e., the program is invalid. Then, $o.inv$ is set to true. **unpack** o; sets $o.inv$ to false.

2.7.2 Ownership

The object invariant abstraction approach explained in the previous subsection is sound, provided that an object invariant depends only on the fields of **this** and not on the fields of other objects. Indeed, if the object invariant of an object o depends on a field $p.f$, where $p \neq o$, then an update of $p.f$ might break the property $o.inv \Rightarrow I[o/\mathbf{this}]$.

However, it is often important to be able to mention fields of objects other than **this** in an object invariant. For example, consider a *SortedList* class that internally stores its elements in an array. The class's methods depend on the consistency property that the elements in the array are sorted. The Boogie methodology allows this type of consistency property to be expressed as an object invariant by introducing an *ownership system*. Specifically, at any time an object o may *own* some set of other objects. An object invariant is allowed to mention not just the fields of **this** but the fields of objects transitively owned by **this** as well. This is sound because the Boogie methodology imposes restrictions such that the property holds that if $o.inv$ is true, then $p.inv$ is true for all objects p transitively owned by o.

Ownership is specified as follows. The programmer may mark a subset of the fields of a class using the keyword **rep**. The objects pointed to by the **rep** fields of an object o are called o's *rep objects*; they are considered to be part of the representation of the state of o. An object o *owns* its rep objects whenever $o.inv$ is true. That is, when a **pack** o; operation is performed, o gains ownership of its rep objects, and when an **unpack** o; operation is performed, o relinquishes ownership of its rep objects.

In order to guarantee the property that if $o.inv$ is true, then $p.inv$ is true for all objects p transitively owned by o, the following restrictions are imposed on **pack** and **unpack** operations.

- When packing an object o, o's rep objects must not be owned and their inv fields must be true.
- When unpacking an object o, o must not be owned.

To simplify the encoding of these conditions in verification conditions, the Boogie methodology adds a second boolean ghost field to each class, called $committed$. This field is initially false, and it is updated only by the **pack** and **unpack** commands. It is updated in such a way that the property always holds that $o.committed$ is true if and only if there is some object p such that p owns o (i.e., p has a rep field f such that $p.f = o$ and $p.inv$ is true).

2.7.3 Method framing

As mentioned in Section 2.5, when emitting a BoogiePL procedure for a Spec# method, in addition to emitting a BoogiePL postcondition for each Spec# postcondition, an additional BoogiePL postcondition is emitted that restricts the set of heap locations (i.e., fields) modified by the method. This is called the *frame condition*.

A method may modify the fields mentioned in its modifies clause. In order to abstract over the set of fields of a given object, the notation $o.*$ may be used to denote all fields of o. Still, this level of abstraction is not sufficient. For example, if a method modifies the internal auxiliary objects of its receiver object, it should not be necessary to name the auxiliary objects in the method's modifies clause explicitly.

A good way to solve this problem would be to allow a method to modify a field $o.f$ if the method's modifies clause contains either $o.f$ or $p.*$, where p transitively owns o. However, encoding transitive ownership in verification conditions presents difficulties for the automatic theorem prover. Therefore, the Boogie methodology adopts the rule that a method may always modify all committed objects. (Recall that the committed objects are all objects that are owned by other objects.) Experience shows that the overapproximation inherent in this rule is not a problem in practice.

2.7.4 Subclassing

The system described in the previous sections is a version of the Boogie methodology for a language without subclassing. In this subsection, we adapt this system to obtain the full Boogie methodology, which supports subclassing.

In general, an object o is an instance of multiple classes. Specifically, if the class of o is C (i.e., o was created through a **new** C operation), and C extends S which extends $Object$, then o is an instance of $Object$, of S, and of C. Each of these classes may declare an object invariant.

The object invariant declared in a class C may mention fields only through expressions of the form $\textbf{this}.f_1.\dots.f_n$, where

- f_1 is declared by C or by a direct or indirect superclass of C, and
- f_1 through f_{n-1} are **rep** fields.

There is one *inv* ghost field per object. (One could consider it to be declared by class $Object$.) In the full methodology, the *inv* field holds a class name rather than a boolean. In particular, at any time, the *inv* field of an object o holds the name of one of the classes of which o is an instance. The methodology imposes restrictions on field updates such that the property holds that if $o.inv <: C$, then $I_C[o/\textbf{this}]$ holds (where $<:$ denotes the reflexive and transitive subclassing

relation and I_C is the object invariant declared by class C). The initial value for $o.inv$ is $Object$. There is also one boolean *committed* field per object.

The *inv* field may be updated only through **pack** o **as** C; and **unpack** o **from** C; commands. Their meaning is given by the following expansions:

pack o **as** C; $\equiv$
  assert $o.inv = \mathbf{super}(C)$;
  assert $I_C[o/\mathbf{this}]$;
  foreach $(p \in \mathrm{rep}(o))$
    assert $p.inv = \mathbf{classof}(p) \wedge \neg p.committed$;
  foreach $(p \in \mathrm{rep}(o))$
    $p.committed := true$;
  $o.inv := C$;

unpack o **from** C; $\equiv$
  assert $o.inv = C \wedge \neg o.committed$;
  $o.inv := \mathbf{super}(C)$;
  foreach $(p \in \mathrm{rep}(o))$
    $p.committed := false$;

$o.f := v$; $\equiv$
  assert $\mathbf{super}(\mathbf{class}(f)) <: o.inv$;
  $o.f := v$;

2.7.5 Method contracts for virtual methods

A method typically starts by unpacking its receiver object from its declaring class, so that it can update the fields of the receiver object declared in the method's declaring class. This is valid only if the receiver object's *inv* field equals the method's declaring class. Therefore, a typical precondition for non-virtual methods is that the receiver's *inv* field equals the method's declaring class.

However, in the case of dynamically-bound calls of virtual methods, such a precondition is not appropriate, since it would not allow an overriding method to unpack its receiver object from its declaring class. To support writing a precondition that allows the dynamic callee (i.e., the method to which the call is bound at run time) to unpack the receiver object from its declaring class in all cases, the Boogie methodology introduces the notation $\mathbf{1}$, which may be used only in method contracts and which denotes the dynamic callee's declaring class. Typically, an appropriate precondition for a method is $\mathbf{this}.inv = \mathbf{1}$.

A problem remains: how are callers to establish this precondition? Experience shows that callers can easily establish this precondition provided that method inheritance is disallowed. Specifically, the Boogie methodology requires each class to override each virtual superclass method. However, for convenience, if no explicit

override is given, Boogie generates a default override that unpacks the receiver from the declaring class, calls the superclass method, and packs the receiver back to the declaring class. This default override is subject to verification, like other methods.

Thanks to this restriction, the property holds that the declaring class of a dynamically bound call's dynamic callee is always the class of the receiver object. Callers may use this property as an aid in establishing a call's precondition.

2.8 Related work

The body of previous work in program verification is enormous. In this section, we mention some of the tools that are most closely related to Boogie.

Three static program verifiers for the object-oriented language Java are LOOP, Jive, and KeY. LOOP [15, 47] takes Java plus contracts written in the Java Modeling Language (JML) [53, 54]. It uses the interactive theorem prover PVS [73], for which it generates proof obligations that look like Hoare triples [38, 46]. It also provides some automation by a weakest-precondition tactic [41]. Jive [71] also uses a Hoare-like logic [76] and its custom-built interactive theorem prover operates at the level of Hoare triples (as opposed to first-order VCs generated from the programs). The KeY tool [3] offers several specification notations, including JML and dynamic logic, and targets several proof engines. The main differences between these three tools and Boogie are that they address a more limited subset of the source language and that they are not automatic.

Program verification technology has also been used in tools that find some program errors without promising to find all errors. These include the Extended Static Checkers for Modula-3 and Java [24, 33, 58, 51], JACK [17], Krakatoa [68], and Cadeuces [29]. The automation in these tools rivals that of Boogie, and they all support the Simplify [22] theorem prover. In addition, JACK supports PVS and the interactive prover of the Atelier B toolkit. Like Boogie, Krakatoa and Cadeuces generate verification conditions via an intermediate language, called Why [28]. Though developed independently, Why and BoogiePL are more similar than they are different. The Why tool currently supports six different theorem provers, both interactive and automatic, but does not support property inference like Boogie's.

A number of programming languages have built-in specifications or were designed with verification in mind. Among these are Gypsy [4], Euclid [52], APP [80], and others mentioned in the Spec# overview paper [12]. The languages SPARK Ada [7], B [2], ACL2 [50], Perfect Developer [26], and C0 [56] include verifiers with interactive theorem provers. The Eiffel language [69] is well known for its pioneering combination of object-orientation and dynamically checked contracts, but it does not yet offer static verification.

2.9 Conclusion

In summary, Boogie is an automatic program verifier for modern object-oriented programs. Its architecture helps tame the complexity of the program verification task. Providing design-time feedback, Boogie moves the program verifier closer to the developer, while still hiding the theorem prover and other verification machinery from the developer. Designed around an intermediate programming notation, BoogiePL, it separates the semantic encoding of the source program from the analysis of this encoding. Since Boogie can also read BoogiePL programs directly, it offers the possibility for others to write program verifiers by encoding their proof obligations in BoogiePL.

The Spec# team has applied Boogie to a growing number of small (300–1500 lines) programs, and is applying Boogie to parts of its own implementation (which is written in Spec#). They also are supporting an experiment in using Boogie on production code. This experience constantly demands support for more programming idioms, more targeted default specifications, better explanations of error messages (especially those having to do with violations of the ownership-based alias-confinement regime), and higher performance.

Boogie can be run as part of compilation, where the compiler provides its in-memory data structures to Boogie for verification. The verification results of Boogie could be used by the compiler's code optimizer to produce better performing code (cf. [84, 30]). However, this is not part of the current Boogie architecture. The prospects of including this feedback in the architecture seem promising, but also contains some research questions such as how and to what degree to rely on specifications of code that may not have been verified.

Chapter 3

Inspector Methods

Preamble

Chapter 2 described the architecture of a tool for performing modular static verification of object-oriented programs. The range of programs supported by such a tool, and the range of properties that it can verify, depends on the *programming methodology* supported by the tool. This chapter and the next present extensions of the Boogie program verifier's programming methodology, which provide a solution to its lack of support for state abstraction and concurrency, respectively.

This chapter describes an approach for referring to the state of a module in the module's specification, in a way that abstracts over the way the state is represented internally using fields and component objects.

This chapter contains the following paper: *Bart Jacobs and Frank Piessens. Verification of Programs with Inspector Methods. Eighth Workshop on Formal Techniques for Java-like Programs (FTfJP 2006). Nantes, July 2006* [45]. Only minor editorial changes were made.

Inspector Methods

Abstract

In an object-oriented program, a class typically provides access to an object's state through methods rather than by directly providing access to the object's fields and internal component objects. This has the benefit that client code does not depend on and cannot interfere with the object's internal representation. Methods used for this purpose are sometimes called *inspector methods*. In order to extend the benefits of inspector methods to specifications, the method contracts of non-inspector methods may be expressed using inspector methods, hence providing support for state abstraction in specifications.

In this chapter, we propose an approach to the verification of programs that use inspector methods in method contracts and object invariants. Inspector methods may have parameters, and they may depend on the state of objects passed as arguments. Our approach builds on the Boogie methodology for object invariants and ownership.

Performing state abstraction in a programming language that allows aliasing through object references poses a framing problem. Specifically, client code needs to be able to tell whether modifying a given object or calling a given method may affect the value of a given inspector method call. We solve this by modeling inspector methods as functions that take as arguments only those parts of the heap on which they depend. Thanks to a novel logical encoding of the heap, we can do this without breaking information hiding, even in cases where inspector methods depend on internal component objects.

The core of our approach has been implemented in a custom build of the Spec# program verifier.

3.1 Introduction

Consider the program in Figure 3.1. Class *Cell* provides access to the state of a *Cell* object using method *getX*. It also uses *getX* to specify the effect of the class's constructor and of the *setX* method. This makes it possible to prove the

```
class Cell {
  int x;
  inspector int getX()
  { return x; }
  Cell(int value)
    ensures getX() = value;
  { x := value; }
  void setX(int value)
    modifies this.*;
    ensures getX() = value;
  { x := value; }
}
```

```
Cell c1 := new Cell(0);
int y := c1.getX();
assert y = 0;
Cell c2 := new Cell(5);
c1.setX(10);
assert c1.getX() = 10;
assert c2.getX() = 5;
```

Figure 3.1: A class specified using an inspector method, and a client program

correctness of the client program using a proof that does not depend on the internal representation of the *Cell* object's state using field x. As a result, when class *Cell*'s internal representation is changed, only class *Cell* needs to be reverified. The client program's proof remains valid.

In this chapter, we concern ourselves with how to prove the correctness of programs such as the one in Figure 3.1.

Note that one of the key problems that needs to be solved by our verification approach is the *framing* problem. Specifically, in order to prove the assertion in Figure 3.1 that $c2.getX() = 5$, it must be encoded in the verification logic that $c2.getX()$ does not depend on any objects modified by method call $c1.setX(10)$.

The remainder of the chapter is structured as follows. We introduce our approach in five versions, each extending the previous one. In Section 3.2, we address the framing problem in its basic form (Version 1). In Section 3.3, we integrate support for object invariants (Version 2) and ownership, and we show how to solve the framing problem for inspector methods that may depend on owned objects (Version 3). In Section 3.4, we show how we allow inspector methods to depend on the state of objects passed as arguments (Version 4). In Section 3.5, we deal with a number of formal details. In Section 3.6, we describe our approach to inheritance (Version 5). In the final sections, we discuss related work and offer a conclusion.

3.2 Framing

In this section, we show how our approach addresses the framing problem in the simple setting where an inspector method may depend only on the fields of the receiver object. In later sections, we extend the approach to allow inspector methods to depend on fields of transitively owned objects and objects passed as arguments.

Our solution consists of three components: the way the heap is modeled, the way inspector method calls are modeled, and the way non-inspector method calls (which may modify the program state) are modeled.

In our verification logic, we represent the heap as a function that maps object references to *object states*. For the simple setting we discuss here, an object state is a function that maps field names to field values. Later sections extend the notion of object state to deal with transitively owned objects.

We treat inspector methods as follows. For each inspector method declared in the program, we introduce a function symbol in the verification logic, as well as an axiom, derived from the inspector method's body, that defines the function symbol's meaning. All information on which the inspector method depends is passed as arguments to the corresponding function. Hence, in the simple setting we discuss here, we pass only the receiver's reference and state. For example, the assertion

$$\mathbf{assert}\ \mathit{c2}.\mathit{getX}() = 5;$$

is modeled as

$$\mathbf{assert}\ \mathit{Cell_getX}(\mathit{c2}, \mathit{Heap}[\mathit{c2}]) = 5;$$

The final ingredient to our approach is the way non-inspector method calls are modeled in our verification logic. The post-state heap of a non-inspector method call is assumed to be an arbitrary new heap, constrained only by the method's declared postconditions and by an implicit postcondition called the *frame condition*. The frame condition says that for all objects o that were allocated in the pre-state and that are not listed by the method's modifies clause, the object's post-state is equal to its pre-state. Formally:

$$\forall o \bullet \mathbf{old}(\mathit{Alloc}[o] \wedge o \notin W) \Rightarrow \mathit{Heap}[o] = \mathbf{old}(\mathit{Heap}[o])$$

where W denotes the set of objects listed by the modifies clause. (We allow only items of the form $o.*$ in modifies clauses. Also, note that a constructor's modifies clause is always considered to implicitly include **this**.$*$.)

We can now prove the program of Figure 3.1. Let Heap and Heap' denote the heap in the pre-state and the post-state, respectively, of the setX call. We need to prove $\mathit{Cell_getX}(\mathit{c2}, \mathit{Heap}'[\mathit{c2}]) = 5$. From the postcondition of Cell's constructor, we have $\mathit{Cell_getX}(\mathit{c2}, \mathit{Heap}[\mathit{c2}]) = 5$. Finally, from the frame condition of the setX call, we have $\mathit{Heap}'[\mathit{c2}] = \mathit{Heap}[\mathit{c2}]$, which allows us to arrive at our goal by substitution.

3.3 Object invariants and ownership

In this section, we integrate inspector methods with the support for object invariants and ownership provided by the Boogie methodology [10]. The example in Figure 3.2 motivates and illustrates the approach. (Note: the expression $(a : b)$

denotes the range of integers i where $a \leq i < b$. Also, throughout this chapter, we indicate assignment using := and comparison using =.)

An object of class *IntList* in Figure 3.2 represents a container for a list of integers. Internally, the integers are stored in an array *elems*. As is common, the array may be larger than the length of the list to minimize the number of heap allocations when adding or removing elements. The actual number of elements is stored in the *count* field.

The constructor's postcondition and the postcondition of the *add* method are expressed using the *getCount* and *getItem* inspector methods. Also, the *getCount* inspector method is used in the precondition of the *getItem* inspector method.

The constructor's postcondition states that in the constructor's post-state, the return value of the *getCount* inspector method is equal to the length of the array passed as an argument, and the return value of a call of *getItem* with argument i is equal to element i of the array, for all non-negative i less than the length of the array. Note that the constructor's contract specifies that the constructor call does not modify the elements of *xs*, since the contract does not include a modifies clause. The meaning of the other clauses in the constructor's contract is introduced below. The constructor's implementation is not shown.

Method *add*'s contract similarly constrains the return values returned by the class's inspector methods in the method's post-state. The postcondition refers to the pre-state using the **old** notation: **old**(E) denotes the result of evaluation of expression E in the method's pre-state.

3.3.1 Object invariants

Methods that operate on an *IntList* object, such as the *add* method, need to know that *count* is never negative and never greater than the length of *elems*. It would be unfortunate to require the method's caller to guarantee this; this would cause the caller to depend on the internals of class *IntList*. To solve this problem, the Boogie methodology provides a mechanism called *object invariants* that allows a developer to expose conditions on internal state to clients in an *abstracted* form. Specifically, it allows a developer to declare an object invariant using the new **invariant** keyword, and in each object, it introduces a special boolean field *inv* and it restricts modifications of this field and the object's other fields in such a way that *inv* is only ever *true* at a time when the object invariant holds. Consequently, by exposing to clients only the *inv* field and requiring *inv* to be *true* on entry to a method, the method can rely on the internal object invariant without having to expose it. Note that a method is not allowed to assume without proof that object invariants hold on entry to the method; this would be unsound because of possible re-entrancy [10]. A method may assume only that its precondition holds.

An object o for which $o.inv$ is *false* is called *mutable*; if $o.inv$ is *true*, the object is called *valid*. An assignment to a field $o.f$ is allowed only when o is mutable.

```
final class IntList {
  rep int[] elems;
  int count;

  invariant 0 ≤ count ∧ count ≤ elems.length;

  inspector int getCount() { return count; }
  inspector int getItem(int index)
    requires 0 ≤ index ∧ index < getCount();
  { return elems[index]; }

  derived_invariant 0 ≤ getCount();

  IntList(int[] xs)
    requires ¬xs.committed;
    ensures ¬this.committed ∧ this.inv;
    ensures getCount() = xs.length;
    ensures
      forall{int i in (0 : getCount()); getItem(i) = xs[i]};
  { ... }

  void add(int x)
    requires ¬this.committed ∧ this.inv;
    modifies this.∗;
    ensures ¬this.committed ∧ this.inv;
    ensures getCount() = old(getCount()) + 1;
    ensures
      forall{int i in old((0 : getCount())); getItem(i) = old(getItem(i))};
    ensures getItem(old(getCount())) = x;
  {
    unpack this;
    count++;
    ensureCapacity(count);
    elems[count − 1] := x;
    pack this;
  }

  ...
}

int[] xs := {1, 2, 3};
IntList list := new IntList(xs);
xs[0] := 5;
assert list.getItem(0) = 1;
```

Figure 3.2: A class that illustrates object invariants, ownership, parameterized inspector methods, inspector method preconditions, and derived invariants. (Note: reference types are non-null types by default.)

Field inv is initially $false$. It may be read only in method contracts, not in program code. Also, it may be updated only through the special new statements **pack** o; and **unpack** o;. These statements behave as follows (where $Inv(o)$ denotes o's object invariant):

pack o; $\equiv$	**unpack** o; $\equiv$
assert $\neg o.inv$;	**assert** $o.inv$;
assert $Inv(o)$;	$o.inv := false$;
$o.inv := true$;	

That is, the **pack** o; operation checks that o is mutable, and that o's object invariant holds. It then marks the object as valid. The **unpack** o; operation checks that o is valid. It then marks the object as mutable.

Provided that an object o's object invariant depends only on the fields of o, the semantics of **pack** and **unpack** together with the restriction on field assignments guarantee that whenever the object is valid (i.e. $o.inv$ is $true$), its object invariant holds. We call this property the soundness of the object invariant methodology.

Inspector methods and object invariants Non-inspector methods that rely on an object's invariant need to require the object's validity as a precondition. Whereas for non-inspector methods we provide the option of either requiring validity of a given object or not requiring it, for inspector methods we always require validity of the receiver object. That is, each inspector method implicitly gets a precondition saying that the receiver object is valid. We made this choice because supporting inspector methods that do not require the validity of their receivers would complicate the approach, and scenarios where the abstraction provided by inspector methods is required but the abstraction provided by object invariants is not, are probably rare.

Since it needs to be possible to evaluate an object invariant even in a state where it does not hold, we do not allow inspector method calls on **this** in an object invariant. However, in addition to an object invariant, a class may declare a *derived invariant*, using one or more **derived_invariant** declarations. If a class declares a derived invariant, this implies a proof obligation that the derived invariant follows from the object invariant. Contrary to an object invariant, a derived invariant may include inspector method calls on **this**. If a class mentions private fields in its object invariant, it must declare the object invariant itself private, and as a result, other classes cannot use it in proofs. Still, the class can expose information to other classes in the form of public derived invariants, provided that the information is stated in terms of public inspector methods.

We could have allowed inspector method postconditions instead of, or in addition to, derived invariants. This would be equivalent in terms of expressiveness. However, since such postconditions would in general have to mention other inspector methods, to express relationships between inspector methods, it seems more

natural to centralize this information at the class level.

Derived invariants (or an equivalent mechanism) serve to reduce specification effort. For example, if class *IntList* did not declare a derived invariant, each method that takes an *IntList* object *list* as an argument would have to specify $0 \leq list.getCount()$ in its precondition and postcondition.

An alternative, more conceptual view of derived invariants is that they help define the abstract state space of their declaring class. One could consider the abstract state of an object as being defined by the return values of the inspector method calls on the object. For example, the abstract state of an *IntList* object o is defined by its item count (returned by $o.getCount()$) and its items (returned by the $o.getItem(i)$ calls, with $0 \leq i < o.getCount()$). Correspondingly, the abstract state space of a class may be defined as the set of abstract states that objects of the class may reach during program execution. Derived invariants help specify a class's abstract state space for the sake of client code. For example, objects of class *IntList* never reach a state where $getCount()$ returns a negative value; therefore, such abstract states are not in the class's abstract state space.

3.3.2 Ownership

The client program provided at the bottom of Figure 3.2 allocates and initializes an array, then uses this array as an argument for the creation of an *IntList* object, then sets element 0 of the array to 5, and finally asserts that item 0 in the list, as reported by the *getItem* inspector method, still equals 0. This client program, in effect, tests whether the state of the *list* object depends on the state of the array. Whether this is the case, depends on the implementation of the *IntList* constructor. If the constructor simply stores a reference to the array in the *elems* field, then the assertion is false; if, on the other hand, the constructor copies the elements of the argument array into a new array and stores a reference to the latter in the *elems* field, then the assertion is true. However, if one naively applies the approach of the preceding section for statically verifying the assertion in the client program, the assertion verifies, regardless of how the constructor is implemented. Clearly, this is unsound.

The cause of this unsoundness is the fact that the *getItem* inspector reads the elements of the *elems* array, even though, as discussed in Section 3.2, the function generated for the logical encoding of *getItem* takes only the state of **this**, not the state of the *elems* array, as a parameter. As a result, the array element update in the client program appears not to affect the *getItem* call.

In order to allow inspector methods like *getItem*, which depend on the state of objects other than the receiver object, we apply the Boogie methodology's *ownership system*, where an object can *own* other objects. Specifically, whenever an object o is valid, it *owns* the objects referred to by o's **rep** fields. For example, in Figure 3.2, whenever an *IntList* object o's *inv* field is **true**, o owns the array pointed to by $o.elems$.

We allow inspector methods to depend on their receiver object, as well as any objects directly or indirectly owned by it. However, in the verification logic, we still wish to pass just the state of the receiver object as an argument, as opposed to passing an additional argument for each owned object. The reason is that **rep** fields are typically private fields, so requiring clients to pass an additional argument for each **rep** field in inspector method function applications when verifying their code would break information hiding.

To make this work, we change the logical encoding of the heap. Object references still map to object states, but the notion of object state is extended to deal with ownership. An object o's object state is extended to contain a copy of the state of each of o's owned objects. Specifically, for each **rep** field $o.f$, we introduce an additional field $o.f_{\mathrm{state}}$ which maps to a copy of the state of the object referred to by $o.f$ whenever $o.inv$ is **true**. Of course these additional fields exist only in the logical encoding; they do not exist at run time. We do not change the program's run-time semantics.

This extended notion of object state allows inspector method results to be defined entirely in terms of the state of the receiver object. For example, the axiom that defines the function symbol for inspector method *getItem* is as follows:

$$\begin{array}{l}\forall o, o_{\mathrm{state}}, i \bullet \\ \quad \mathit{IntList_getItem}(o, o_{\mathrm{state}}, i) = o_{\mathrm{state}}[\mathit{IntList_elems}_{\mathrm{state}}][i]\end{array}$$

Clearly, using a copy of an object's state instead of the original is sound only if the copy is up-to-date whenever it is used. This is exactly what the heap consistency theorem below shows. The proof of this theorem relies on another aspect of the Boogie methodology. The methodology does not allow updates to fields of *committed* objects, i.e. objects owned by other objects.

An object p is committed if and only if there is some valid object o that has a **rep** field $o.f$ such that $o.f = p$. However, to simplify the verification logic, we track whether an object o is committed explicitly in the form of a boolean field $o.\mathit{committed}$. Like the *inv* field, this field can be read only in method contracts and cannot be assigned to explicitly in code.

To deal with ownership, we extend the meaning of the **pack** and **unpack** commands as in Figure 3.3.

Heap consistency As explained above, if the body of an inspector method dereferences an owned object **this**.f, we retrieve its state from a special field **this**.f_{state} instead of looking it up in the heap as usual. This is sound because **this** is *rep-consistent*:

Definition 3.1 (rep-Consistency)**.** *An object* o *is* rep-consistent *if, for each non-null* **rep** *field* $o.f$*, it holds that* $o.f_{\mathrm{state}} = \mathit{Heap}[o.f]$*, or, fully expanded,*

$$\mathit{Heap}[o][f_{\mathrm{state}}] = \mathit{Heap}[\mathit{Heap}[o][f]]$$

pack o; $\equiv$
  assert $\neg o.committed \wedge \neg o.inv$;
  foreach (non-null **rep** field $o.f$)
    assert $\neg o.f.committed \wedge o.f.inv$;
  assert $Inv(o)$;
  foreach (non-null **rep** field $o.f$) {
    $o.f.committed := true$;
    $o.f_{\text{state}} := Heap[o.f]$;
  }
  $o.inv := true$;

unpack o; $\equiv$
  assert $\neg o.committed \wedge o.inv$;
  foreach (non-null **rep** field $o.f$)
    $o.f.committed := false$;
  $o.inv := false$;

Figure 3.3: The **pack** and **unpack** commands in the presence of ownership

In fact, we have the following theorem:

Theorem 3.1 (rep-Consistency)**.** *In each program state of each execution of each valid program, each valid object is rep-consistent.*

Proof. Since our approach is a conservative extension of the Boogie methodology, we may assume the known properties of the Boogie methodology. In particular, we know that objects pointed to by **rep** fields of valid objects are valid and committed.

We prove the theorem by induction over the length of an execution. The theorem holds for the empty execution, since in the initial program state no object is valid. Now consider a non-empty execution. We now look at the final command performed in this execution:

- A field assignment $p.g := v;$. We know that p is mutable and rep-consistency involves only valid objects; therefore, this command does not invalidate the theorem.
- A **pack** p; operation. This operation establishes the rep-consistency of p, and it does not break the rep-consistency of the objects pointed to by the **rep** fields of p since changing $o.committed$ does not influence the rep-consistency of an object o.
- An **unpack** p; operation. Since p is made mutable, the theorem no longer applies to p. Also, since the Boogie methodology guarantees that committed objects have unique owners, changing the *committed* bits of p's owned objects does not invalidate the rep-consistency for any valid object $o \neq p$.
- No other commands modify existing objects.

□

Ownership and frame conditions Recall from Section 3.2 that each method gets an implicit postcondition, called the *frame condition*, that encodes in the verification logic that some objects are not modified by the method. The frame condition given in Section 3.2 was:

$$\forall o \bullet \mathbf{old}(Alloc[o] \wedge o \notin W) \Rightarrow Heap[o] = \mathbf{old}(Heap[o])$$

In the presence of ownership, a different frame condition is required. Specifically, we wish to allow a method that lists an object o in its modifies clause to modify not just o, but objects directly or indirectly owned by o as well. (We cannot require the method to list these owned objects in the modifies clause because of information hiding.) To achieve this, we follow the Boogie methodology in allowing the method to modify any object that is *committed* in the pre-state, in addition to the objects listed in the modifies clause:

$$\forall o \bullet \mathbf{old}(Alloc[o] \wedge o \notin W \wedge \neg o.committed) \Rightarrow Heap[o] = \mathbf{old}(Heap[o])$$

3.4 Multi-dependent inspector methods

In this section, we show how the approach supports *multi-dependent* inspector methods, i.e. inspector methods that depend on the state of objects passed as arguments, in addition to the state of the receiver object.

A motivating example is shown in Figure 3.4. It shows part of the Microsoft .NET Framework's support for restricting the access partially trusted code has to system resources.[1] A *PermissionSet* object holds the *permissions* assigned to a given piece of code. Permissions are represented using objects that implement interface *Permission*. For example, a *SecurityPermission* object may represent permission to run, or permission to skip bytecode verification (or both, or neither).

Interface *Permission* declares an inspector method *isSubsetOf* that returns whether one permission of a given type is implied by another permission of the same type. Also, class *PermissionSet* declares an inspector method *contains* that returns whether the set contains a given permission.

Importantly, in the .NET Framework *Permission* objects are mutable. That is, a given *Permission* object may be made to represent different permissions at different points in time. However, a *PermissionSet* object contains specific permissions, not *Permission* objects. Therefore, the *contains* inspector method must be allowed to depend on the state of the *Permission* object passed as an argument, not just its identity.

For example, in the piece of client code shown at the bottom of Figure 3.4, permission to execute is added to a permission set, and the *contains* method returns *true* for the *Permission* object that represents this permission. However,

[1] We changed names to conform to Java naming conventions.

```
interface Permission {
  inspector boolean isSubsetOf(state Permission other)
    requires other.getClass() = getClass();
}
final class SecurityPermission implements Permission {
  static final int EXECUTION := 1;
  static final int SKIP_VERIFICATION := 2;

  int flags;
  inspector int getFlags() { return flags; }
  inspector boolean isSubsetOf(state Permission other) {
    return (flags & ∼((SecurityPermission)other).flags) = 0;
  }
  derived_invariant
    forall{state SecurityPermission p;
      isSubsetOf(p) = ((getFlags() & ∼p.getFlags()) = 0)};

  SecurityPermission(int flags)
    ensures getFlags() = flags;
  { this.flags := flags; }

  void setFlags(int flags)
    ensures getFlags() = flags;
  { this.flags := flags; }
}
final class PermissionSet {
  import Permission;
  ...
  inspector boolean contains(state Permission p) { ... }

  PermissionSet()
    ensures forall{state Permission p; ¬contains(p)};
  { ... }

  void setPermission(Permission p)
    ensures forall{state Permission q;
      contains(q) = (q.getClass() = p.getClass() ? q.isSubsetOf(p) : old(contains(q)))};
  { ... }
}

PermissionSet s := new PermissionSet();
Permission p := new SecurityPermission(SecurityPermission.EXECUTION);
s.setPermission(p);
assert s.contains(p);
p.setFlags(SecurityPermission.EXECUTION|SecurityPermission.SKIP_VERIFICATION);
assert ¬s.contains(p);
```

Figure 3.4: An example demonstrating multi-dependent inspector methods and quantification over object states

if subsequently permission to skip verification is added to the *Permission* object, calling *contains* with the same object returns *false*. This shows the need for multi-dependent inspector methods.

The following issues arise when multi-dependent inspector methods are admitted: how are calls of such methods translated into the verification logic, how do we ensure consistency of the resulting logic, and how do we specify the abstract state of an object using multi-dependent inspector methods.

An inspector method is allowed to depend on the state of the receiver object, the objects that are passed as arguments for parameters marked **state**, and their transitively owned objects. A call of an inspector method is translated into an application of the corresponding function symbol with the following arguments:

- For each argument a to the inspector method call, there is an argument to the function symbol application that models the value a.
- In addition, for each argument a to the inspector method call for a parameter marked **state**, there is an argument to the function symbol application that models the state of the object a.

An inspector method implicitly gets a precondition that says that each argument a for a **state** parameter must be valid (i.e., $a.inv$ is *true*).

Consistency of the verification logic For each inspector method, an axiom is added to the verification logic that defines the corresponding function symbol. We restrict inspector method bodies to be of the form { **return** E; }, so that we can translate an inspector method whose body is { **return** E; } into an axiom

$$\textbf{axiom}\ (\forall x_1, \ldots, x_n \bullet C_m(x_1, \ldots, x_n) = [[E]]);$$

where $[[E]]$ denotes the translation of Java expression E into a logical term. A potential problem with this approach is that the resulting set of axioms may be inconsistent. Notice that this is the case only if there is an inspector method call that does not terminate. To ensure that inspector methods always terminate, we restrict which method calls may appear inside inspector method bodies. Specifically, if a method call $o.m_1(\ldots)$ appears in the body of an inspector method m_2, then m_1 must be an inspector method and the the declaring class of m_2 must transitively *import* the static type of o. A type may import another type explicitly using an **import** declaration. Also, when a class C implements an interface I, I implicitly imports C (sic). For example, in Figure 3.4, *PermissionSet* imports *Permission* and *Permission* imports *SecurityPermission*. (We discuss the import relation in the presence of subclassing in Section 3.6.) It is checked at load time that the import relation is acyclic. This ensures that inspector method calls always terminate.

Quantification over object states Typically, a postcondition of a non-inspector method fully specifies the post-state of each object o that is modified by the method. It does so by specifying the value of each inspector method call on o. If an inspector method has parameters, this requires quantifying over the possible argument values. For example, in the *IntList* example (Figure 3.2), both the constructor and the *Add* method have an **ensures** clause that quantifies over a range of integers, to serve as the argument to the *getItem* inspector method.

If an inspector method takes an object reference as an argument, one may quantify over object references, using the following syntax:

$$\textbf{forall}\{T\ o;\ E\}$$

Variable o ranges over both unallocated and allocated objects, so that if the inspector method whose value is being defined by the quantification is called with an argument object that is allocated after the quantification is asserted, the quantification applies to that call as well. However, expression E is not allowed to access the state of o and E cannot assume the validity of o; therefore, o cannot be passed as an argument for a **state** parameter.

To support full specification of the abstract state of an object that has inspector methods that take object states as arguments, we support quantification over object states, using the following syntax:

$$\textbf{forall}\{\textbf{state}\ T\ o;\ E\}$$

Variable o again ranges over both unallocated and allocated object references, but when the state of o is inspected, it is not looked up in the heap; rather, the state, too, is considered to be bound by the quantification, and it ranges over all possible valid object states. That is, the above quantification translates into the verification logic as follows:

$$(\forall o, o_{\text{state}} \bullet o_{\text{state}}[inv] \Rightarrow [[E]])$$

For example, referring to Figure 3.4,

$$\textbf{forall}\{\textbf{state}\ \textit{Permission}\ q;\ \textit{contains}(q) = \dots\}$$

is encoded as

$$\begin{array}{l}(\forall\ q, q_{\text{state}} \bullet q_{\text{state}}[inv] \Rightarrow \\ \quad \textit{PermissionSet_contains}(\textit{this}, \textit{Heap}[\textit{this}], q, q_{\text{state}}) = \dots)\end{array}$$

3.5 Formal details

Well-formedness conditions Method preconditions and postconditions and object invariants must be *pure expressions*, and the body of an inspector method

must be of the form { **return** E; } where E is a pure expression. A pure expression is a Java expression that does not contain object or array creations, simple or compound assignments, or increment or decrement operators, and that calls only inspector methods. This ensures that evaluation of pure expressions has no side-effects and that their translation into first-order logic terms, denoted as $[[\cdot]]$, can be defined easily.

Additionally, object invariants and derived invariants may depend only on the fields of **this** and on the fields of objects transitively owned by **this**. An object invariant cannot assume that **this** is valid; therefore, it cannot include inspector method calls on **this**. It can, however, include inspector method calls on objects pointed to by **rep** fields of **this**. Also, as stated before, inspector method bodies may depend only on the fields of the receiver object, objects passed as arguments for parameters marked **state**, or objects transitively owned by these objects.

Soundness of derived invariants The declaration of a derived invariant in a class C generates a proof obligation saying that it holds for all valid objects of class C. For discharging this obligation for an object o, one may assume that all derived invariants applicable to the owned objects of o hold for those objects (even if some of the owned objects are themselves of class C). To show that this proof rule is sound, we prove the following theorem:

Theorem 3.2 (Derived invariants)**.** *In each execution state, for each valid object, each derived invariant applicable to it holds.*

Proof. By induction on the length of the execution. The only interesting case is when the last operation is a **pack** o; operation. The induction hypothesis allows us to apply the aforementioned proof obligation to prove that all derived invariants applicable to o hold. □

Dynamically bound inspector method calls An inspector method call may be either statically or dynamically bound. For example, calls of inspector methods of final classes are statically bound, and calls of inspector methods of interfaces are dynamically bound. (In the presence of subclassing, methods may generally be called both ways. We discuss subclassing in Section 3.6.)

In contrast with a statically-bound inspector method, the function for a dynamically-bound inspector method is not defined once and for all by the method's declaration; rather, it is partially defined by each non-abstract inspector method that overrides it. Specifically, for each non-abstract inspector method m in class C, and each dynamically-bound inspector method m in class or interface T that it overrides, an axiom is generated that says that if the target object's type is C, both functions coincide. Formally:

$$\begin{array}{l}\forall o, o_{\text{state}}, a_1, \ldots, a_n \bullet \mathsf{type}(o) = C \Rightarrow \\ \quad T_m(o, o_{\text{state}}, a_1, \ldots, a_n) = C_m(o, o_{\text{state}}, a_1, \ldots, a_n)\end{array}$$

3.6 Subclassing

In this section, we describe an extension of our approach that allows a superclass to hide its internal representation not only from client code, but from code in subclasses as well.

Our approach is based on the observation that if we want to abstract superclass state from subclasses, then objects are no longer the unit of abstraction. Indeed, we must subdivide an object along the classes that contribute state to it. We call each such subdivision an *object frame* (or *frame* for short). That is, a frame (or frame reference) is a tuple (o, C) consisting of an object reference o and the name of a class C of which o is an instance. If a class D extends a class C which extends class Object, then an object whose type is D consists of three frames: (o, D), (o, C), and (o, Object).

We update our heap representation accordingly. Rather than mapping object references to object states, in our new encoding the heap maps *frames* to *frame states*. The frame state for a frame (o, C) consists of the values of the fields of o declared in C.

In previous sections, we achieved abstraction of object state from client code by introducing per-object *inv* and *committed* fields, introducing **pack** and **unpack** operations on objects, introducing an ownership relation between objects, caching owned object state in owner objects, and passing object states as arguments in inspector method function applications. Completely analogously, to achieve abstraction of superclass state from subclass code, we introduce per-*frame inv* and *committed* fields, **pack** and **unpack** operations on *frames*, and an ownership relation between *frames*, and we cache owned frame state in owner frames and we pass frame states as arguments in inspector method function applications. Note that in the absence of subclassing, we again obtain the approach of the previous sections as a special case.

Frame ownership The definition of the ownership relation between frames involves a few design issues. One is: are there ownership relationships between the frames of a given object? In order to allow the subclass object invariant and subclass inspector methods to depend on superclass state, we consider a valid subclass frame to own its superclass frame. In fact, each class other than *Object* gets a special **rep** field called *super* that refers to the superclass frame. (Note that this field is exceptional in that it contains a frame reference instead of an object reference.) It follows that packing a subclass frame requires that the superclass frame is valid and uncommitted and commits it. Also, if the most derived frame of an object (i.e., the frame corresponding to the object's run-time type) is valid, it transitively owns all of the object's frames and encapsulates the entire object's state.

There is one other design decision involving the ownership relation between

```
class Cell {
  int x;
  invariant 0 ≤ x;
  inspector int getX() { return x; }
  derived_invariant 0 ≤ getX();
  dynamic_invariant 0 ≤ getX();

  Cell(int x)
    requires 0 ≤ x;
    ensures ¬committed ∧ inv;
    ensures getX() = x;
  { this.x := x; pack this; }

  void setX(int x)
    requires ¬committed ∧ inv;
    requires 0 ≤ x;
    modifies this.∗;
    ensures ¬committed ∧ inv;
    ensures getX() = x;
  {
    unpack this;
    this.x := x;
    pack this;
  }
}
```

```
class MyCell extends Cell {
  invariant 1 ≤ super.getX();
  inspector int getX()
  { return super.getX() − 1; }

  MyCell(int x)
    requires 0 ≤ x;
    ensures ¬committed ∧ inv;
    ensures getX() = x;
  { super(x + 1); pack this; }

  void setX(int x)
    requires ¬committed ∧ inv;
    requires 0 ≤ x;
    modifies this.∗;
    ensures ¬committed ∧ inv;
    ensures getX() = x;
  {
    unpack this;
    super.setX(x + 1);
    pack this;
  }
}
```

Figure 3.5: Example illustrating the approach for verification of programs with subclassing. An instance of type *Cell* may hold an arbitrary nonnegative integer value, which may be retrieved using inspector method *getX* and set using method *setX*. Both class *Cell* and class *MyCell* implement the *Cell* type. Class *Cell* does so by storing the value in a field x, whereas *MyCell* does so by storing the value plus one in its superclass frame. While contrived, this example shows how our approach supports the clean separation of the two aspects of subclassing: interface re-implementation and implementation inclusion. There need be no relationship between how the superclass and the subclass implement the superclass's interface; in particular, if the subclass chooses to re-use the included superclass implementation, it need not do so by direct delegation. This clean separation promotes modularity, i.e. separate development and evolution of superclass and subclass code. Note: all methods in this example are virtual and subclass methods override the corresponding superclass methods.

frames. Specifically, if a **rep** field $o.f$ declared in a class C of a valid object o points to an object p, which ownership relationship does this give rise to? In order to allow the object invariant and the inspector methods of class C to perform dynamically-bound inspector method calls on p, which access p's most derived frame, we consider frame (o, C) to own p's most derived frame. Due to the previous design decision, it follows that (o, C) transitively owns all of p's frames.

Static and dynamic binding As pointed out earlier, in general inspector method calls may be statically bound or dynamically bound. Calls of abstract methods are always dynamically bound and calls of private methods are always statically bound, but for a non-abstract inspector method of a non-final class, some calls that resolve to such a method at compile time may be statically bound and some may be dynamically bound. Since at run time, these calls may bind to different methods and yield different results, we encode them differently in the verification logic. Specifically, for calls resolved at compile time to a method m of a class or interface T, we encode statically bound ones as $T_m(\dots)$ and dynamically bound ones as $T_m_{\mathrm{D}}(\dots)$. Function symbol T_m is defined fully by the body of method m in T; function symbol T_m_{D} is defined partially by each non-abstract method that overrides it, as explained earlier.

In the encoding of a statically bound call to an inspector method m of class C on an object o, the frame state passed as the state of the receiver is always the state of frame (o, C), whereas in the encoding of a dynamically bound call, the frame state passed is always the state of the most derived frame. (Also, an argument for a parameter marked **state** in a call of a multi-dependent inspector method is always interpreted as the most derived frame.)

The object invariant and the derived invariants declared in a class C apply to frames (o, C). Since they must depend only on the state of the frame to which they apply (plus transitively owned frames), they must not include dynamically bound inspector method calls on **this**. Therefore, we always interpret inspector method calls on **this** in object invariants and derived invariants as statically bound calls.

Inspector method calls may occur in preconditions of inspector methods and in preconditions and postconditions of non-inspector methods. It is crucial for the soundness of verification that there be no confusion as to whether these calls are statically or dynamically bound. Our approach is sound regardless of which choice is made, so long as the choice is the same when verifying the caller and when verifying the callee. Note in this regard that it is fine, and often appropriate, to make different choices for statically and dynamically bound calls of the method in whose contract the inspector method call appears. This is sound so long as it is verified that the contract for dynamically bound calls is implied by the contract for statically bound calls under the assumption that the run-time type of the receiver equals its static type. The method body need then only be verified against the contract for statically bound calls.

The question remains as to how the choice of static or dynamic binding of inspector method calls in method contracts is made. Java specifies that a call in program code is dynamically bound unless the call is a **super** call or the method being called is private. However, this approach is not always appropriate for inspector method calls in method contracts. For example, consider the $getX()$ call in the contract of method $setX$ of class $Cell$ in Figure 3.5. Interpreting the call as dynamically bound would be approriate for dynamically bound calls of $setX$, but not, for example, for the **super**.$setX$ call that occurs in class $MyCell$. Indeed, **super**.$setX(5)$ ensures **super**.$getX() = 5$, not $getX() = 5$.

Therefore, we interpret calls in contracts differently from calls in program code. Also, the interpretation is different depending on whether the contract is used for a statically bound call or a dynamically bound call. The rule is as follows: all calls are bound dynamically, except for **super** calls, calls of private methods, or calls on **this** in contracts for statically bound calls. Non-**super** calls of non-private methods whose target expression is not **this** (explicitly or implicitly) or calls on **this** in contracts for dynamically bound calls are bound dynamically. For example, for the purpose of verifying the **super**.$getX$ call in Figure 3.5, the $getX$ call in the contract of $Cell.setX$ is interpreted as a statically bound call. Note that this rule ensures that the contract for dynamically bound calls is equivalent to the contract for statically bound calls under the assumption that the receiver's run-time type is equal to its static type.

The import relation In the presence of subclassing, the import relation which we use to ensure termination of inspector method calls, and consistency of their axiomatization, must be refined. In particular, for each class, we introduce two nodes in the import graph, which we shall call the static node and the dynamic node. An interface only has a dynamic node. The rules are as follows:

- The dynamic node of C imports the static node of C.
- If a class C declares an **import** T; declaration, this means the static node of C imports the dynamic node of T.
- If a class D extends a class C, this means the static node of D imports the static node of C, and the dynamic node of C imports the static node of D.
- If a class C implements an interface I, this means the dynamic node of I imports the static node of C.
- If a class C declares an inspector method whose body includes a dynamically bound inspector method call resolved at compile time to a method of a type T, then the static node of C must transitively import the dynamic node of T.

- If a class C declares an inspector method whose body includes a statically bound inspector method call declared by a class D, then the static node of C must import the static node of D.

This ensures that there can be no inspector method call cycles within an inheritance hierarchy.

Dynamic invariants As stated above, a class C may declare a derived invariant, using the **derived_invariant** keyword. This invariant applies to frames (o, C) only, and inspector method calls in derived invariants are always interpreted as statically bound calls. This means that derived invariants declared by a class C are useful for a subclass of C when performing **super** calls, or if C is a **final** class, but not for client code that accesses an instance of C through dynamically bound calls.

To convey properties of and relationships between dynamically bound inspector methods, a class or interface may declare one or more *dynamic invariants*, using the **dynamic_invariant** keyword. A dynamic invariant declared by a type T is enforced against all non-abstract classes that implement or extend T. It follows that a dynamic invariant declared by a type T holds for all instances of T.

Note: dynamic invariants do not subsume derived invariants; for example, an abstract class that implements some of its inspector methods may declare a derived invariant to specify a relationship between the inspector methods it implements.

Method inheritance As in the Boogie methodology, we do not allow method inheritance as such. That is, each class must override all visible methods of its superclass. This rule is crucial for the soundness of our approach since our approach is based on the assumption that if a dynamically bound call binds to a method m declared by a class C, then the static type C of m's receiver is equal to its run-time type. This is true only if m was not inherited from C by the receiver's run-time type.

However, to reduce programming overhead, we follow the Boogie methodology in generating a default override for each method that is not overridden explicitly. The body of the default override for a method m is of the form

$$\{\ \textbf{unpack this};\ \textbf{super}.m();\ \textbf{pack this};\ \}$$

(or similar if m has parameters or a return value). Note that default overrides are subject to verification just like explicitly declared methods. If a default override fails to verify, an explicit override must be provided.

3.7 Related work

The Boogie methodology Our approach's support for object invariants, ownership, and method framing is based on the Boogie methodology [10]. In order to add support for inspector methods, we applied the following modifications:

- In the Boogie methodology, the heap is encoded in the verification logic as a function that maps an object reference and field name combination to a field value. We encode the heap as a function from frame references to frame states. (Note: the notion of object frames was introduced in [10], under the name *class frames*, but there it was not used as the basis for the encoding of the heap.)

- We extend the semantics of the **pack** command to store a copy of each owned frame's state in a special field of the owner frame. This allows inspector method calls to be encoded as function applications that take object frame states.

- In the Boogie methodology, method frame conditions are expressed as equalities between field values. We express method frame conditions as equalities between object frame states. Together with the previous modification, this enables a theorem prover to carry information regarding inspector method call return values across method calls.

- Instead of introducing a single *committed* bit and a single *inv* field per object, and storing in the *inv* field the name of the most derived type whose object frame is valid, we introduce separate *inv* and *committed* bits in each object frame. This allows object frames to be treated as independent units.

Method calls in specifications Darvas and Müller [20] identify and propose solutions for problems that arise when method calls are used in specifications. Specifically, the authors show how to deal with abrupt termination, object creation, and inconsistent axiomatization due to unsatisfiable postconditions. Methods called in specifications must be pure, which means they do not modify existing objects. Inspector methods are like pure methods, with the additional constraint that they depend only on the state of the receiver object and objects passed as arguments for parameters marked **state** (and their transitively owned objects). A minor difference is that we do not allow inspector methods to declare a postcondition and that we derive the axiom that defines the corresponding function from the inspector method's body rather than its postcondition. We avoid the abrupt termination issue by verifying that the inspector method body does not throw any exceptions.

Darvas and Müller's solution to the object creation issue is to pass the heap as an argument to the function and have the function return a new heap together

with its result value. We did not adopt this solution because it is incompatible with our approach to framing; specifically, our approach requires that only the state of the dependee objects is passed to the inspector method's function, rather than the entire heap. Therefore, we disallow object creation in inspector methods.

We avoid the problem of inconsistent axiomatization by imposing a partial order on classes and by allowing nested inspector method calls to proceed along this partial order only. This ensures that inspector method calls always terminate and have a return value under Java semantics, and the return values thus obtained always satisfy the system of equations that defines the inspector method functions of a given program.

State abstraction in ownership systems Müller [70] combines a notion of *model fields* with an ownership type system called *Universes*. Model fields are comparable with parameterless inspector methods. The Universes ownership system is less flexible than that of the Boogie methodology; for example, it does not allow ownership transfer. On the other hand, Müller allows model fields of an object o to depend on fields of *peer objects* of o, i.e. objects that have the same owner as o, provided that the model field definitions are visible to the field declarations, i.e. both are in the same module. We intend to add support for visibility-based inspector methods to our approach as future work.

Leino and Müller [65] achieve state abstraction by combining the Boogie ownership system with another notion of model fields, similar to but distinct from that of Müller [70]. Model fields in [65] are encoded in the verification logic as if they are stored in the heap along with concrete fields. Each model field declaration specifies a model field constraint that serves as an abstraction relation. The constraint for a model field $o.m$ needs to hold only when o is valid. As part of executing a **pack** statement on an object o, the constraints of the model fields of o are checked and, if they do not hold, a new value that satisfies the constraint is assigned to the model field. If no such value exists, the pack statement is considered invalid.

A constraint may underspecify a model field, and subclasses may strengthen an inherited model field's constraint. As a result, packing an object for a subclass may assign a new value to a model field declared in a direct or indirect superclass. An underspecified model field is similar to an abstract inspector method with a dynamic invariant, and a strengthening of an inherited model field's constraint by a subclass is similar to an inspector method that overrides an abstract inspector method. However, "overriding" a fully specified model field with another differently fully specified model field, similar to overriding a non-abstract inspector method, is not supported in [65]. This means that a subclass is forced to adopt fully specified public superclass model fields as part of its own abstract state, without the ability to provide a different abstraction function. Alternatively, if a class leaves a model field underspecified and does not tie it to its own concrete state, so that subclasses have maximum freedom in providing an abstraction function, then

the class cannot use the model field to abstractly expose its own state. Therefore, it also is not able to fully implement methods specified using the model field. As a result, in practice, classes with underspecified model fields are effectively abstract. In other words, [65] does not fully support specification of classes that may be used both for direct instantiation and as superclasses, while leaving subclasses free to provide their own abstraction functions for superclass model fields. (For example, see class *Cell* in Figure 3.5.)

To support the kind of specifications enabled in our approach by parameterized inspector methods, such as the *getItem* method in the *IntList* example of Figure 3.2, [65] would have to use model fields containing special-purpose immutable objects such as immutable list objects (known as *model types* in JML [55]).

Ownership-free approaches Kassios [49] also uses abstraction functions, but instead of an ownership system he proposes *dynamic frames* to abstractly specify an abstraction function's dependencies and a mutator method's effects. Dynamic frames are themselves abstraction functions that return sets of locations. A module specification may specify a frame (i.e., an upper bound on the set of locations that an abstraction function depends on) for each abstraction function separately.

The dynamic frames approach subsumes our approach on an abstract level. However, it is formulated in the context of an idealized logical framework; for example, it does not show how to apply the approach to Java-like inheritance, but this could probably be done in a way similar to what we do in Section 3.6. Also, the proposed approach has not been applied in the context of an automatic program verifier. Therefore, it is not clear whether the approach can be used as a basis for generating verification conditions suitable for automatic proof.

Parkinson and Bierman [75] and Parkinson [74] extend separation logic with *abstract predicates* and apply it to Java to achieve state abstraction for Java programs. Abstract predicates are similar to inspector methods that return a boolean value. However, as in the case of the aforementioned dynamic frames approach, Parkinson and Bierman do not restrict the set of locations that an abstract predicate may depend on. Rather, it is up to client code to track the separation between abstract predicates. Parkinson and Bierman solve the problem of well-definedness of abstract predicates by allowing abstract predicates to appear inside other abstract predicates only in positive positions, and by taking the least fixpoint of a set of abstract predicate definitions as their meaning. Subclassing is addressed by introducing *abstract predicate families*; that is, an abstract predicate name may be subscripted by a class name, and a separate definition may be given for each subscript. For example, the abstract predicate saying that a *Cell* instance o holds the value v could be written $cell_{\mathsf{type}(o)}(o, v)$.

The use of abstract predicates in method contracts typically requires the use of universally quantified logical variables whose scope extends across both the pre- and post-state. For example, the contract for a method that that increments the

value of a cell would say that for each value v, if $cell(o, v)$ holds in the pre-state, then $cell(o, v+1)$ holds in the post-state. This could be a disadvantage compared to model fields or inspector methods, in particular for run-time checking.

Redundant invariants Our derived invariants and dynamic invariants are similar to JML's [55] redundant invariants, in that an object's redundant invariants must be implied by its invariants. However, the interaction between JML's different kinds of invariants and JML model fields is not clear. Specifically, on which invariants is a JML model field's *represents clause*, which defines its abstraction relation, allowed to depend to prove absence of evaluation errors such as null dereferences or divisions by zero? And which invariants are allowed to mention model fields of **this**?

3.8 Future work

An important item of future work is to perform a rigorous and comprehensive formalization and soundness proof of the approach.

It seems promising to investigate relaxations of the requirement that nested inspector method calls follow a partial order. One relaxed rule would be that for each nested call, the callee must depend on fewer objects (or object frames) than the caller. Or alternatively, that the size of the argument list, defined as the total number of object states, including duplicates and internal owned object states, must decrease.

Another area of extension is to allow inspector methods to depend on non-owned *peer* objects of classes declared in the same module.

We implemented the core of our approach in a custom build of the Spec# program verifier [13]. It remains as future work to use this implementation to gain experience in applying the approach to realistic programs.

3.9 Conclusion

We proposed an approach to the verification of object-oriented programs that use inspector methods for state abstraction in specifications.

We solve the problem of encoding in the verification logic whether a given method call or field assignment affects a given inspector method call's return value, by

- modeling the heap as a function that maps object references to object states,
- logically (but not physically) storing a copy of the state of owned objects in special fields of the owner object, and

- encoding inspector method calls as function applications whose arguments include the object states on which the inspector method depends.

We support multi-dependent inspector methods, i.e. inspector methods that depend on the state of objects passed as arguments. To enable the specification of the return values of all possible calls of a multi-dependent inspector method, we support quantification over object states.

Our approach to subclassing is to separate its two aspects: superclass interface re-implementation and superclass implementation inclusion. Subclasses may override superclass inspector methods; our approach preserves soundness by binding inspector method calls statically or dynamically as appropriate. A distinction is made between derived invariants, which apply only to the declaring class, and dynamic invariants, which apply to subclasses as well.

We implemented the core of our approach in a custom build of the Spec# program verifier [13].

Chapter 4

Verifying Concurrent Programs

Preamble

This chapter describes the approach for verification of concurrent programs.

This chapter contains the following paper (with minor changes, as noted below): *Bart Jacobs, Jan Smans, Frank Piessens, and Wolfram Schulte. A simple sequential reasoning approach for sound modular verification of mainstream multithreaded programs. First Workshop on Multithreading in Hardware and Software: Formal Approaches to Design and Verification (TV 06). Invited Talk, Seattle, August 21-22, 2006.* This paper was presented during an invited talk by Wolfram Schulte and the author. The same paper, minus Section 4.6 on support for static fields and static initializers, was published as *Bart Jacobs, Jan Smans, Frank Piessens, and Wolfram Schulte. A statically verifiable programming model for concurrent object-oriented programs. In Proceedings of the Eighth International Conference on Formal Engineering Methods (ICFEM 2006), volume 4260 of LNCS. Springer, 2006* [48].

For this chapter the *LinkedList* example in Figure 4.9 and the accompanying discussion in Section 4.4.2 were added. Other changes are of an editorial nature.

The ideas of the approach are introduced in this chapter semi-formally. For a full formal development, see Chapter 5. In particular, a formal treatment of the method effect framing approach, as well as the details of the way thread interference is taken into account for the verification of **synchronized** blocks, may be found in Chapter 5.

The research for this approach started during an internship at Microsoft Research in Redmond, WA, U.S.A. in the summer of 2004. Initial results were presented at SAVCBS 2004 [43] and later at SEFM 2005 [42]. Further results were

presented at ICFEM 2006 [48] and TV 06.

The author developed a prototype implementation of a static program verifier for the approach of this chapter, called SpecLeuven, which is available from the author's web site, together with the sample programs verified by the tool.

The major new features of SpecLeuven compared to the Boogie program verifier on which it is based, are:

- Sound support for C#'s **lock** blocks (equivalent to Java's **synchronized** blocks)
- Sound support for C#'s *delegate types* (equivalent to single-method interfaces) and *delegate instance creation expressions* (equivalent to instance creation of an anonymous class that implements the interface by delegating to a given target method on a given target object)
- Sound support for C# thread creation
- Support for inspector methods (but not all validity constraints required for soundness are checked at this time; also, multi-dependent inspector methods are not yet supported)
- Sound support for read-only accessibility and immutable objects
- Sound support for deadlock prevention
- Sound support for loop framing (i.e., if a loop invariant does not require writability of an object, then a loop frame condition is added saying that the loop does not change the object)
- Sound support for set-valued ghost *rep* fields

Verifying Concurrent Programs

Abstract

Reasoning about multithreaded object-oriented programs is difficult, due to the non-local nature of object aliasing, data races, and deadlocks. We propose a programming model that prevents data races and deadlocks, and supports local reasoning in the presence of object aliasing and concurrency. Our programming model builds on the multi-threading and synchronization primitives as they are present in current mainstream languages. Java or C# programs developed according to our model can be annotated by means of stylized comments to make the use of the model explicit. We show that such annotated programs can be formally verified to comply with the programming model. In other words, if the annotated program verifies, the underlying Java or C# program is guaranteed to be free from data races and deadlocks, and it is sound to reason locally about program behavior. Our approach supports immutable objects as well as static fields and static initializers. We have implemented a verifier for programs developed according to our model in a custom build of the Spec# programming system, and have validated our approach on a case study.

4.1 Introduction

Writing correct multithreaded software in mainstream languages such as Java or C# is notoriously difficult. The non-local nature of object aliasing, data races, and deadlocks makes it hard to reason about the correctness of such programs. Moreover, many assumptions made by developers about concurrency are left implicit. For instance, in Java, many objects are not intended to be used by multiple threads, and hence it is not necessary to perform synchronization before accessing their fields. Other objects are intended to be shared with other threads and accesses should be synchronized, typically using locks. However, the program text does not make explicit if an object is intended to be shared, and as a consequence

it is impossible in practice for the compiler or other static analysis tools to verify if locking is performed correctly.

We propose a programming model for concurrent programming in Java-like languages, and the design of a set of program annotations that make the use of the programming model explicit. For instance, a developer can annotate his code to make explicit whether an object is intended to be shared with other threads or not. These annotations provide sufficient information to static analysis tools to verify if locking is performed correctly: shared objects must be locked before use, unshared objects can only be accessed by the creating thread. Moreover, the verification can be done modularly, hence verification scales to large programs.

Several other approaches exists to verify race- and deadlock-freedom for multi-threaded code. They range from generating verification conditions [23, 32, 36, 77, 1, 79], to type systems [16, 34]. (See Section 4.8 for an overview of related work.)

Our approach is unique, in that it builds around protecting invariants and that it allows sequential reasoning for multithreaded code. Our contributions are thus as follows:

- We present a programming model and a set of annotations for safe concurrent programming in Java-like languages.
- Following our programming model ensures absence of data races and deadlocks.
- The generated verification conditions allow sound local reasoning about program behavior. Note that in this chapter we ignore null dereference checking to avoid clutter, although our prototype implementation fully supports it.
- We have prototyped a verifier as a custom build of the Spec# programming system [13, 9], and in particular its program verifier for sequential programs.
- Through a case study we show the model supports useful, non-trivial programs and we assess the annotation overhead.

The present approach evolved from [42] and [48]. It improves upon [42] by directly supporting platform-standard locking primitives, by preventing deadlocks, by adding support for immutable objects, and by reporting on experience gained using a prototype implementation. It improves upon [48] by adding support for static fields and static initializers. As did [42] and [48], it builds on and extends the Spec# programming methodology [10] that enables sound reasoning about object invariants in sequential programs.

The rest of the chapter is structured as follows. We introduce the methodology in three steps. The model of Section 4.2 prevents low-level data races on individual fields. Section 4.3 adds deadlock prevention. The final model, which adds prevention of races on data structures consisting of multiple objects, is presented in Section 4.4. Each section consists of three subsections, that elaborate the

programming model, the program annotations, and the static verification rules, respectively. The remaining sections discuss immutable objects (Section 4.5), our approach for static fields and static initializers (Section 4.6), experience (Section 4.7), and related work (Section 4.8), and offer a conclusion.

4.2 Preventing data races

In this section, we present our approach for the modular static verification of the absence of data races in Java-like programs.

A data race occurs when multiple threads simultaneously access the same variable, and at least one of these accesses is a write access. Developers can protect data structures accessed concurrently by multiple threads by associating a mutual exclusion lock with each data structure and ensuring that a thread accesses the data structure only when it holds the associated lock. However, mainstream programming languages such as Java and C# do not force threads to acquire any locks before accessing data structures, and they do not enforce that locks are associated with data structures consistently.

A simple strategy to prevent data races is to lock every object before accessing it. Although this approach is safe, it is rarely used in practice since it incurs a major performance penalty, is verbose, and is prone to deadlocks. Instead, standard practice is to only lock the objects that are effectively shared between multiple threads. However, it is hard to distinguish shared objects (which should be locked) from unshared objects based on the program text. As a consequence, without additional annotations a compiler cannot enforce a locking discipline where shared objects can only be accessed when locked and unshared objects can be accessed without locking.

An additional complication is the fact that the implementation of a method may assume that an object is already locked by its caller. Hence, the implementation will access fields of a shared object without locking the object first. In such a case, merely indicating which objects are shared does not suffice. The implementor of a method should also make his assumptions about locks that are already held by the calling thread explicit in a method contract.

In this section, we describe a simple version of our approach that deals with data races on the fields of shared objects. Later sections develop this model further to deal with deadlocks and high-level races on multi-object data structures.

The approach is presented in three steps. First, a *programming model* is defined such that Java programs that comply with the programming model (the *legal programs*) are data-race-free. Secondly, the annotations that are required to make the programming model explicit and to enable modular static verification are presented. Thirdly, the verification approach is explained. The verification approach is such that annotated Java programs that verify (the *valid programs*) are legal, and therefore are data-race-free.

4.2.1 Programming model

We describe our programming model in the context of Java, but it applies equally to C# and other similar languages.

Before we explain the programming model, we briefly review Java's built-in synchronization feature. In Java, threads may synchronize using Java's **synchronized** statement. A thread may enter a **synchronized** (o) block only if no other thread is executing inside a **synchronized** (o) block; otherwise, the thread waits. In the remainder of the chapter, we use the following terminology to refer to Java's built-in synchronization mechanism: when a thread enters a **synchronized** (o) block, we say it *acquires* o*'s lock* or, as a shorthand, that it *locks* o; while it is inside the block, we say it *holds* o*'s lock*; and when it exits the block, we say it *releases* o*'s lock*, or, as a shorthand, that it *unlocks* o. Note that, contrary to what the terminology may suggest, when a thread locks an object, the Java language prevents other threads from locking the object but it does not prevent other threads from accessing the object's fields. This is the main problem addressed by the proposed methodology. While a thread holds an object's lock, we also say that the object *is locked* by the thread.

An important terminological point is the following: when a thread t's program counter reaches a **synchronized** (o) block, we say the thread *attempts to lock* o. Some time may pass before the thread *locks* o, specifically if another thread holds o's lock. Indeed, if the other thread never unlocks o, t never locks o. The distinction is important because our programming model imposes restrictions on attempting to lock an object.

We now explain our programming model. By *programming model*, we mean a separation of the set of Java programs into *legal programs* (the ones that comply with the programming model) and *illegal programs* (the ones that do not comply). Our programming model defines the set of legal programs by defining a dynamic semantics for Java programs that extends Java's own dynamic semantics with extra state variables, an extra operation, and a notion of *illegal operation*. The legal programs are those that perform no illegal operations under this semantics. Our programming model has the property that legal programs are data-race-free. Note that the extended dynamic semantics serves only to define the set of legal programs; our approach ultimately verifies the data-race-freedom of Java programs when executing under Java's original semantics.

The programming model is a first step toward an approach for modular static verification of the data-race-freedom of Java programs, because it transforms a non-thread-local dynamic property (data-race-freedom) to a thread-local dynamic property (legality of an operation). In Sections 4.2.2 and 4.2.3, we further transform the thread-local dynamic property of legality of an operation to a method-local static property of validity of an operation.

The programming model ensures that legal programs are data-race-free as follows. In the extended dynamic semantics, an additional state variable is associated

with each thread t, called the thread's *access set* $t.A$, which is a set of object identities. A read or write operation on a field $o.f$ by a thread t is legal under the programming model only if the target object o is in t's access set $t.A$. Under the extended step rules, the access sets evolve in such a way that they are always disjoint. As a result, two threads never access the same field concurrently, and the program is data-race-free.

The main thread's access set is initially empty. When a thread creates an object or acquires an object's lock, the object is added to the thread's access set. When a thread releases an object's lock the object is removed from the thread's access set. In order to prevent a data race between the thread that creates an object and a thread that acquires the object's lock, the programming model distinguishes between *shared* and *unshared* objects; objects are initially unshared, and attempting to acquire an object's lock is legal only if the object is shared. Formally, the dinstinction between shared and unshared objects is made in the dynamic semantics by extending the program state with a global state variable S, called the *shared set*, which is a set of object identities. An object is considered *shared* in a given state if and only if it is in the shared set.

An unshared object becomes shared when a thread performs a *share operation* on the object. A share operation is an operation of the extended dynamic semantics that does not correspond to a Java statement. A share operation modifies only the extra state variables; specifically, a share operation performed by a thread t on an unshared object o that is in $t.A$ removes o from $t.A$ and adds it to the shared set. It is illegal for a thread to perform a share operation on an object that is not in its access set.

It's important to note that the extra state variables are used in the definition of which operations are illegal, but they do not otherwise influence the behavior of the program. Specifically, if a Java program performs a given operation under Java semantics (in some execution), then it performs the same operation under the extended semantics (in some execution). Another way to put this is to say that the extra state does not cause operations to block that would not otherwise block.

Note the following:

- When a thread creates a new object, the object is added to the creating thread's access set. This means the constructor can initialize the object's fields without acquiring a lock first. This also means single-threaded programs just work: if there is only a single thread, it creates all objects, and can access them without locking.

- In our programming model, objects that are not intended to be shared are never locked.

- Once shared, an object can never revert to the unshared state.

- Starting a new thread transfers the accessibility of the receiver object of the thread's main method (i.e. the *Runnable* object in Java, or the *ThreadStart* delegate instance's target object in the .NET Framework) from the starting thread to the started thread. This is necessary since otherwise, the thread's main method would not be allowed to access its receiver.

As illustrated in Figure 4.1, an object can be in one of three states: *unshared*, *free* (not locked by any thread and shared) or *locked* (locked by some thread and shared). Initially, an object is unshared. Some objects eventually transition to the shared state (at a program point indicated by the developer). After this transition, the object is not part of any thread's access set and is said to be *free*. To access a free object, it must be locked first, changing its state to locked and adding the object to the locking thread's access set. Unlocking the object removes it from the access set and makes it free again.

Let's summarize. Threads are only allowed to access objects in their corresponding access set. A thread's access set consists of all objects it created or whose lock it holds, plus the receiver of its main method (if any), minus the objects on which it performed a share operation or which it used as the target in a thread creation. Our programming model prevents data races by ensuring that access sets never intersect.

Lock re-entry

Java's **synchronized** blocks are re-entrant; that is, if a thread already holds an object o's lock, then an attempt to enter another **synchronized** (o) block succeeds immediately. This has implications for the programming model. Specifically, we must prevent the scenario where a thread t_1 locks an object o, then uses it as the target object for creating a thread t_2, and then locks o again. At this point, the access sets of t_1 and t_2 both contain o and data races are possible.

Besides this problem, lock re-entry also complicates modular verification. Therefore, we rule out lock re-entry in the programming model. We achieve this as follows:

- We require the target object of a thread creation to be unshared. As a result, we have the property that in each program state, a shared object is in a thread's access set if and only if the thread holds the object's lock.
- We require that when a thread attempts to lock an object, the object is not yet in the thread's access set. It follows that the thread does not yet hold the lock.

4.2.2 Program annotations

In this section we elaborate on the annotations needed by our approach by means of the example shown in Figure 4.2. The example consists of a program that

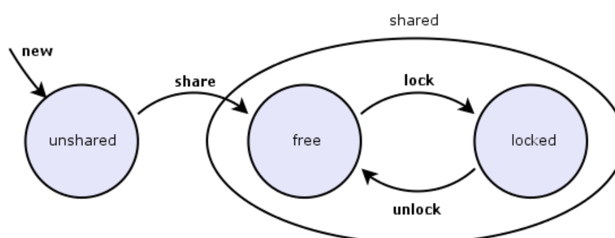


Figure 4.1: The three states of an object.

observes events from different sources and keeps a count of the total number of events observed. Since the count is updated by multiple threads, it is subject to data races unless precautionary measures are taken. Our approach ensures that it is impossible to "forget" to take such measures.

In our prototype implementation (see Section 4.7), annotations are written as stylized comments. But to improve readability, we use a language integrated syntax in this chapter.

The program shown in Figure 4.2 is a Java program augmented with a number of annotations (indicated by the gray background). More specifically, three sorts of annotations are used: **share** commands, **shared** modifiers and method contracts. Furthermore, := denotes assignment and = equality.

- The **share** command makes an unshared object available for concurrent access by multiple threads. In the example, the *counter* object is shared between all sessions.
- Fields and parameters can be annotated with a **shared** modifier, indicating they can only hold shared objects. The field *counter* of *Session* is an example of a field with a **shared** modifier.
- Method contracts are needed to make modular verification possible. They consist of preconditions and postconditions. A precondition states what the method implementation assumes about the current thread's access set (denoted as $\mathbf{tid}.A$) and about the global shared set. For instance, the precondition of the *start* method requires the access set to be empty. Postconditions state properties of the access set and the shared set guaranteed by the method upon return. For example, the postcondition of *Session*'s constructor guarantees that the new object is in the current thread's access set and unshared.

Note that our annotations are entirely erasable, i.e. they have no effect whatsoever on the execution of the program.

Note: An important purpose of method contracts is to *frame* the method's effect. The method effect framing approach used in this chapter is based not on

```
class Counter {
   int count;
   Counter()
      ensures this ∈ tid.A ∧ this ∉ S;
   {}
}
class Session implements Runnable {
   shared Counter counter;
   int sourceId;
   Session(Counter counter, int sourceId)
      requires counter ∈ S;
      ensures this ∈ tid.A ∧ this ∉ S;
   {
      this.counter := counter;
      this.sourceId := sourceId;
   }
   public void run()
      requires tid.A = {this} ∧ this ∉ S;
   {
      for (;;) {
         // Wait for event from source sourceId (not shown)
         synchronized (counter) {
            counter.count++;
         }
      }
   }
}
class Program {
   static void start()
      requires tid.A = ∅;
   {
      Counter counter := new Counter();
      share counter;
      new Thread(new Session(counter, 1)).start();
      new Thread(new Session(counter, 2)).start();
   }
}
```

Figure 4.2: Example program illustrating the approach of Section 4.2. Programmer-supplied annotations are shown on a gray background.

modifies clauses in method contracts, but on a method's *required access set*; this is the set of objects that the precondition requires to be in the access set. The frame condition is that if an object is in the pre-state access set, but not in the required access set, then it is in the post-state access set and its state has not changed. Note that if an object is in the required access set, then it is guaranteed to still be in the access set in the post-state only if the postcondition says so.

The example program is correctly synchronized (i.e., data-race-free), and the annotations enable our static verifier to prove this. We discuss in the next subsection how this is done. If the developer forgets to write the **synchronized** block in the *run* method, the program is no longer correctly synchronized. Specifically, the access of *counter.count* in method *run* violates the programming model, since object *counter* is not in the thread's access set.

Thread creation

To verify the example, we also need the method contracts of all library methods used by the program. These are shown in Figure 4.3.

The method contracts shown in Figure 4.3 encode the programming model's rules regarding thread creation.

- The *Thread* constructor requires its argument to be part of the calling thread's access set and unshared. The constructor removes the *Runnable* object from the access set and associates it with the *Thread* object. Indeed, the constructor's postcondition does not state that in the post-state, the *Runnable* object is still in the access set, and therefore the caller cannot assume this and can no longer access the *Runnable* object.

- When method *start* is called, a new thread is started and the *Runnable* object associated with the *Thread* object is inserted into the new thread's access set. Method *run*'s precondition allows the method to assume that its receiver is the only object in the access set and that this object is unshared. It follows that no shared objects are in the access set and therefore the thread is allowed to lock them.

4.2.3 Static verification

We have explained our programming model informally in the previous subsections. In this subsection we define the model more formally, and show how we can statically verify adherence to the model in a modular (i.e. per-method) way.

We proceed as follows: a program P enriched with our annotations is translated to a *verification-time* program P' enriched with assertions and classical method contracts. This translation defines the semantics of our annotations, and

```
public interface Runnable {
  void run();
    requires tid.A = {this} ∧ this ∉ S;
}
public class Thread {
  public Thread(Runnable runnable)
    requires runnable ∈ tid.A ∧ runnable ∉ S;
    ensures this ∈ tid.A ∧ this ∉ S;
  { . . . }
  public void start()
    requires this ∈ tid.A;
  { . . . }
}
```

Figure 4.3: Contracts for the library methods used by the program in Figure 4.2.

is the formal definition of our programming model: the original annotated program P is correct according to our model, if and only if the translated program P' is correct with respect to its assertions and classical method contracts. To check if the translated program P' is correct, we use an existing automatic program verifier for single-threaded programs. Our experiments show (Section 4.7) that state-of-the-art verifiers are capable of verifying non-trivial, useful programs in this way.

The contributions of this chapter are in the design of the annotation syntax (for the multithreading-specific annotations) and the translation of the annotated program; we use existing technology [9] for sequential program verification. The translation involves two things. In a first step, we insert two verification-only global variables into the program (so called *ghost variables*) to track the state necessary to do the verification. The ghost variable **tid**.A represents the current thread's access set, while S represents the set of shared objects.

Then, in a second step each method of the original program is translated in such a way that the translated method can be verified modularly. The method contracts that the developer writes in annotations are classical method contracts on the ghost state introduced in the first step. The code and other annotations written by the developer are translated into verification-time code and proof obligations (written as assertions) for the verifier. The essence of the translation of code and annotations is shown in Figure 4.4. It is a formalization of the programming model rules introduced in Section 4.2.1. We ignore the fact that object references can be null to reduce clutter. The verification-time code for a **synchronized** block includes a **havoc** operation that assigns an arbitrary value to all fields of the object being locked. This reflects the fact that other threads may have modified these

fields. Source program assignment and verification-time assignment are shown as $:=$ and $\leftarrow$, respectively.

$o := \textbf{new}\ C;\ \equiv$
  $o \leftarrow \textbf{new}\ C;$
  $\textbf{assume}\ o \notin S;$
  $\textbf{tid}.A \leftarrow \textbf{tid}.A \cup \{o\};$

$x := o.f;\ \equiv$
  $\textbf{assert}\ o \in \textbf{tid}.A;$
  $x \leftarrow o.f;$

$o.f := x;\ \equiv$
  $\textbf{assert}\ o \in \textbf{tid}.A;$
  $\textbf{if}\ (f \text{ is declared } \textbf{shared})$
    $\textbf{assert}\ x \in S;$
  $o.f \leftarrow x;$

$\textbf{share}\ o;\ \equiv$
  $\textbf{assert}\ o \in \textbf{tid}.A;$
  $\textbf{assert}\ o \notin S;$
  $\textbf{tid}.A \leftarrow \textbf{tid}.A \setminus \{o\};$
  $\textbf{tid}.S \leftarrow \textbf{tid}.S \cup \{o\};$

$\textbf{synchronized}\ (o)\ B\ \equiv$
  $\textbf{assert}\ o \in S;$
  $\textbf{assert}\ o \notin A;$
  $\textbf{havoc}\ o.*;$
  $\textbf{tid}.A \leftarrow \textbf{tid}.A \cup \{o\};$
  B
  $\textbf{tid}.A \leftarrow \textbf{tid}.A \setminus \{o\};$

Figure 4.4: Translation of source program commands to verification-time commands.

4.3 Lock levels for deadlock prevention

The approach of Section 4.2 prevents data races but it does not prevent deadlocks. In this section, we introduce our approach to deadlock prevention.

For the purpose of this chapter, we define a deadlock to be a cycle of threads such that each thread is waiting for the next thread to release some lock. Formally, a deadlock is a sequence of threads $t_0, \ldots, t_{n-1}$ and a sequence of objects $o_0, \ldots, o_{n-1}$ such that t_i holds o_i's lock and is trying to acquire $o_{(i+1) \bmod n}$'s lock. Threads involved in a deadlock are stuck forever.

The prototypical way in which a developer can avoid deadlocks is by defining a partial order over all shared objects, and by allowing a thread to attempt to acquire an object's lock only if the object is less than all objects whose lock the thread already holds.

There are different common strategies for defining such a partial order. A first one is to define the order statically. This approach is common in case the shared objects protect global resources: code has to acquire these resources in the statically defined order. A second strategy is to define the order based on some field of the objects involved. For instance to define a transfer operation between accounts, the two accounts involved can be locked in order of the account number, thus avoiding deadlocks while locking account objects.

In some cases the developer of a particular module may only wish to impose partial constraints on the locking order or may wish to abstract over a set of objects. For instance the developer of the Subject class in the Subject-Observer pattern may wish to specify that Observers should be locked before locking the Subject and not vice-versa. In other words, all Observers are above the Subject in the deadlock prevention ordering.

4.3.1 Programming model

Our programming model is designed to support all three scenarios outlined above. The developer can indicate his intended ordering through the intermediary of *lock levels*. A lock level is a value of the new primitive type (existing only for verification purposes) **locklevel**. The programming model extends program states with a new state variable called the *lock level graph*, consisting of a finite set of lock levels and a less-than ($<$) relation between lock levels. The lock level graph is initially empty. A new lock level ℓ can be created using the statement

$$\ell := \mathbf{between}(\{\ell_1^A, \ldots, \ell_m^A\}, \{\ell_1^B, \ldots, \ell_n^B\});$$

where $0 \leq m, n$. The new lock level is inserted above the existing lock levels ℓ_1^A through ℓ_m^A and below the existing lock levels ℓ_1^B through ℓ_n^B. To ensure that the lock order graph remains a partial order, each specified lower bound must be below each specified upper bound.

Note that, just like access sets and the shared set, the lock order graph is a theoretical construction which serves only to explain our approach for statically verifying properties of normal executions of standard Java programs on standard Java virtual machines.

In the model, a lock level is associated with an object the moment the object is shared. This defines the lock order: for shared objects o_1 and o_2, we have $o_1 < o_2$ iff $o_1.\mathit{lockLevel} < o_2.\mathit{lockLevel}$. A thread is only allowed to lock an object if the object is less than the objects whose lock the thread already holds.

Notice that the above rule prevents both deadlocks and lock re-entry. Therefore, in this model, the separate lock re-entry prevention approach of Section 4.2 is no longer required. The *Runnable* and *Thread* method contracts in the new model are as in Figure 4.5.

The level of indirection introduced by the lock levels provides an easy way to abstract over sets of objects. In the Subject-Observer example discussed above, all Observer objects can be given the same lock level (that should be above the Subject lock level).

4.3.2 Program annotations

In a concurrent Java or C# program, a lock ordering adopted by the developers of a program for the purpose of deadlock prevention is not explicit in the pro-

```
public interface Runnable {
  void run();
    requires this ∈ tid.A ∧ tid.lockStack.isEmpty();
}
public class Thread {
  public Thread(Runnable runnable)
    requires runnable ∈ tid.A;
    ensures this ∈ tid.A ∧ this ∉ S;
  { ... }
  public void start()
    requires this ∈ tid.A;
  { ... }
}
```

Figure 4.5: Method contracts for the thread creation API in the programming model of Section 4.3.

gram text, although it can be documented informally in comments. We propose annotations that make it possible for a developer to document the intended ordering formally. As a consequence, static verification of adherence to the ordering is possible (Section 4.3.3).

Three kinds of annotations are important. We discuss them using the example of the Dining Philosophers program in Figure 4.6. The program implements a deadlock-free solution to the Dining Philosophers problem with three philosophers. Our annotations explain formally why the program is deadlock-free.

The first kind of annotation is the creation of a lock level using the **between** constructor. The example defines the lock levels and their ordering statically in class *Program*'s *start* method. Three linearly ordered levels are defined: $\mathit{level1} < \mathit{level2} < \mathit{level3}$.

The second kind of annotation associates lock levels with shared objects. The **share** annotation is extended to accept a lock level as the second argument. Again, this happens three times in the example: each of the forks is shared with its associated lock level. As a consequence, fork objects are totally ordered, with $\mathit{fork1} < \mathit{fork2} < \mathit{fork3}$. Hence, forks can only be locked in descending order.

The third kind of annotations are the method contracts that make modular static verification possible. Method contracts make explicit what assumptions the method makes about the ordering of parameter objects, or about locks already held by the current thread. For instance the constructor of *Philosopher* expects its first argument to have a lower lock level than the second argument, and the *run* method requires that the current thread holds no locks. Another typical requirement expressed in a contract is that a given object is less than the objects

```
class Fork {
}

class Philosopher implements Runnable {
  shared Fork fork1;
  shared Fork fork2;

  Philosopher( shared Fork fork1, shared Fork fork2)
    requires fork1.lockLevel < fork2.lockLevel;
    ensures this ∈ tid.A ∧ this ∉ S
  {
    this.fork1 := fork1;
    this.fork2 := fork2;
  }

  public void run()
    requires this ∈ tid.A;
    requires tid.lockStack.isEmpty();
  {
    for (; ; ) {
      synchronized (fork2) {
        synchronized (fork1) {
          // Use the forks to eat...
        }
      }
    }
  }
}

class Program {
  static void start()
    requires tid.lockStack.isEmpty();
  {
    locklevel level1 := between({}, {});
    locklevel level2 := between({level1}, {});
    locklevel level3 := between({level2}, {});
    Fork fork1 := new Fork();
    share (fork1, level1);
    Fork fork2 := new Fork();
    share (fork2, level2);
    Fork fork3 := new Fork();
    share (fork3, level3);
    new Thread(new Philosopher(fork1, fork2)).start();
    new Thread(new Philosopher(fork2, fork3)).start();
    new Thread(new Philosopher(fork1, fork3)).start();
  }
}
```

Figure 4.6: Deadlock prevention for the Dining Philosophers

already locked by the thread. We use the following abbreviated syntax for this:

$$o < \mathbf{tid}.\mathit{lockStack} \quad \equiv \quad (\forall p \in \mathbf{tid}.\mathit{lockStack} \bullet o.\mathit{lockLevel} < p.\mathit{lockLevel})$$

which is equivalent to

$$\mathbf{tid}.\mathit{lockStack}.\mathit{isEmpty}() \vee o.\mathit{lockLevel} < \mathbf{tid}.\mathit{lockStack}.\mathit{top}()$$

These annotations enable a formal static verification of the absence of deadlocks.

4.3.3 Static verification

Static verification is again done by translating the annotated program P into a program P' enriched with proof obligations for a static verifier (in the form of classical method contracts and assertions). The translation adds ghost fields and variables to track the necessary state. To track the lock level of objects, we add to each object a ghost field called *lockLevel*, whose value is either *null* or a lock level and whose initial value is *null*. The field is written only once: when the object is shared a non-null lock level is assigned to this field. This way, each shared object has an immutable association with a lock level.

To track the locks that the current thread holds, we introduce a ghost variable $\mathbf{tid}.\mathit{lockStack}$, which is a stack containing the objects whose lock the thread holds. Whenever a thread acquires an object's lock, the object is pushed onto the stack. Note that it follows that the top of the stack is always the least of all objects on the stack. A thread is allowed to acquire an object o's lock only if the lock stack is empty or o's lock level is strictly less than the lock level of the object at the top of the stack.

The essence of the translation of an annotated program is summarized in Figure 4.7. Note that the rules for object creation and field access have been omitted since they are unchanged from the previous section.

4.4 Invariants and ownership

The approach as described in the preceding sections ensures absence of low-level data races and deadlocks. However, it does not prevent higher-level race conditions, where the programmer protects individual field accesses, but not updates involving accesses of multiple fields or objects that are part of the same data structure. As a result, accesses may be interleaved in such a way that the data structure's consistency is not maintained.

share (o, l); $\equiv$
  assert $o \in \textbf{tid}.A$;
  assert $o \notin S$;
  $\textbf{tid}.A \leftarrow \textbf{tid}.A \setminus \{o\}$;
  $\textbf{tid}.S \leftarrow \textbf{tid}.S \cup \{o\}$;
  $o.\mathit{lockLevel} \leftarrow l$;

synchronized (o) B $\equiv$
  assert $o \in S$;
  assert $o < \textbf{tid}.\mathit{lockStack}$;
  $\textbf{tid}.\mathit{lockStack}.\mathit{push}(o)$;
  havoc $o.*$;
  $\textbf{tid}.A \leftarrow \textbf{tid}.A \cup \{o\}$;
  B
  assert $o \in \textbf{tid}.A$;
  $\textbf{tid}.A \leftarrow \textbf{tid}.A \setminus \{o\}$;
  $\textbf{tid}.\mathit{lockStack}.\mathit{pop}()$;

Figure 4.7: Translation of source program commands to verification-time commands.

4.4.1 Programming model

To prevent race conditions that break the consistency of multi-object data structures, we integrate the Spec# methodology's object invariant and ownership system [10] into our approach. The model supports objects that use other objects to represent their state, and object invariants that express consistency constraints on such multi-object structures.

The programming model requires the programmer to designate a subset of each class's fields as the class's *rep fields*. The objects pointed to by an object o's non-null *rep* fields in a given program state are called o's *rep objects*. An object's *rep* objects may have *rep* objects themselves, and so on; we refer to all of these as the object's transitive *rep* objects. The fields of an object, along with those of its transitive *rep* objects, are considered in our approach to constitute the entire representation of the state of the object; hence the name. As will be explained later, a shared object o's lock protects both o and its transitive *rep* objects.

In addition to a set of *rep* fields, the programming model requires the programmer to designate, for each class C, an *object invariant*, denoted $\mathit{Inv}_C(o)$ when applied to an object o of C. $\mathit{Inv}_C(o)$ is a predicate that may depend on the state of o, i.e. the fields of o and of its transitive *rep* objects.

The object invariant for an object o need not hold in each program state; rather, the programming model associates with each object a boolean state variable called its *inv bit*.[1] The programming model requires the object invariant to hold only when the *inv* bit is *true*.

The programming model requires an object's *inv* bit to be *true* when a thread shares the object or unlocks it, i.e. when the object becomes free. It follows that each free object's *inv* bit is *true* and its object invariant holds. As a result, when

[1]The *inv* bit is not a field in the actual program; it is a variable introduced only to explain the programming model.

a thread locks an object, it may assume that the object's *inv* bit is true and its object invariant holds.

At the start of an object's constructor, its *inv* bit is *false*. The programming model requires the programmer to designate the regions of code where an object's invariant is supposed to hold by designating the points where **pack** *o*; and **unpack** *o*; operations occur. The former sets *o*'s *inv* bit to *true*, and the latter sets it to *false*. These commands are the only way to update an object's *inv* bit; it may not be assigned directly.

To ensure that whenever an object's *inv* bit is *true*, its object invariant holds, the programming model imposes the following restrictions:

- A thread may assign to an object's fields only when the object is in the thread's access set *and* the object's *inv* bit is *false*. Furthermore, the remaining restrictions ensure that whenever an object's *inv* bit is *true*, then so are those of its transitive *rep* objects. As a result, an object's state does not change while its *inv* bit is *true*.

- A thread is allowed to perform a **pack** *o*; operation only when *o*'s object invariant holds, its *inv* bit is *false*, its *rep* objects are in the thread's access set, and their *inv* bits are *true*. Furthermore, besides setting *o*'s *inv* bit to *true*, the operation removes *o*'s *rep* objects from the thread's access set.

- A thread is allowed to perform an **unpack** *o*; operation only when *o*'s *inv* bit is *true*. The operation sets *o*'s *inv* bit to *false* and adds *o*'s *rep* objects to the thread's access set.

We say that an object *owns* its *rep* objects whenever its *inv* bit is *true*. It follows from the above restrictions that an object has at most one owner.

Note that our approach supports ownership transfer; a *rep* object can be moved from one owner to another by first unpacking both owners and then simply updating the relevant *rep* fields.

4.4.2 Program annotations

The example in Figure 4.8 shows the annotations required by our final methodology. A *Rectangle* object is used to store the bounds of an application's window. The *Rectangle*'s state is represented internally using two *Point* objects, that represent the location of the upper-left and lower-right corner, respectively. If the user drags the window's title bar, the window manager moves the window, even if the application is painting the window contents. Our methodology ensures that the application sees only valid states of the *Rectangle* object.

Developers designate a class's *rep* fields using the **rep** modifier, they define a class's object invariant using **invariant** declarations, and they insert **pack** and

```
class Point {
  int x, y;
  void move(int dx, int dy)
    requires this ∈ tid.A ∧ this.inv;
    ensures this ∈ tid.A ∧ this.inv;
  {
    unpack this; x := x + dx;
    y := y + dy; pack this;
  }
}
class Rectangle {
  rep Point ul, lr;
  invariant ul.x ≤ lr.x ∧ ul.y ≤ lr.y;
  void move(int dx, int dy)
    requires this ∈ tid.A ∧ this.inv;
    ensures this ∈ tid.A ∧ this.inv;
  {
    unpack this; ul.move(dx, dy);
    lr.move(dx, dy); pack this;
  }
  int getHeight()
    requires this ∈ tid.A ∧ this.inv;
    ensures this ∈ tid.A ∧ this.inv;
    ensures 0 ≤ result;
  {
    unpack this; int h := lr.y − ul.y;
    pack this; return h;
  }
}
```

```
class Application {
  shared Rectangle windowBounds;
  void paint()
    requires tid.lockStack.isEmpty();
    requires this ∈ tid.A ∧ this.inv;
    ensures this ∈ tid.A ∧ this.inv;
  {
    int height;
    synchronized (windowBounds) {
      height := windowBounds.getHeight();
    }
    ...
  }
}
class WindowManager {
  shared Rectangle windowBounds;
  void mouseDragged(int dx, int dy)
    requires tid.lockStack.isEmpty();
    requires this ∈ tid.A ∧ this.inv;
    ensures this ∈ tid.A ∧ this.inv;
  {
    synchronized (windowBounds) {
      windowBounds.move(dx, dy);
    }
  }
}
```

Figure 4.8: An example illustrating our data race and deadlock prevention strategy, combined with object invariants and ownership.

```
class Node { Node prev, next; int value; }

class LinkedList {
  Node first, last;
  ghost rep Set⟨Node⟩ nodes;
  invariant first ≠ last;
  invariant first ∈ nodes ∧ last ∈ nodes;
  invariant (∀ n ∈ nodes • n = first ∨ n.prev ≠ null);
  invariant (∀ n ∈ nodes • n = last ∨ n.next ≠ null);
  invariant first.prev = null ∧ last.next = null;
  invariant (∀ n ∈ nodes • n.next ≠ null ⇒ n.next ∈ nodes);
  invariant (∀ n ∈ nodes • n.prev ≠ null ⇒ n.prev ∈ nodes);
  invariant (∀ n ∈ nodes • n.next = null ∨ n.next.prev = n);
  invariant (∀ n ∈ nodes • n.prev = null ∨ n.prev.next = n);

  void append(LinkedList other)
    requires other ≠ this;
    requires this ∈ tid.A ∧ this.inv ∧ other ∈ tid.A ∧ other.inv;
    ensures this ∈ tid.A ∧ this.inv;
  {
    unpack this;
    unpack other;
    if (other.first.next ≠ other.last) {
      nodes := nodes ∪ other.nodes \ {other.first, other.last};
      Node left := last.prev;
      left.next := other.first.next;
      last.prev := other.last.prev;
      left.next.prev := left;
      last.prev.next := last;
    }
    pack this;
  }

  synchronized void appendSync(LinkedList other)
    requires other ≠ this;
    requires this < tid.lockStack ∧ other ∈ tid.A ∧ other.inv;
  { this.append(other); }
}
```

Figure 4.9: An example illustrating ownership transfer of unbounded numbers of aliased objects

unpack commands in method bodies. Additionally, developers may denote an object o's *inv* bit in method contracts, using the $o.inv$ notation.

Another example is shown in Figure 4.9. It shows how an object may own an unbounded number of objects. If a class declares regular **rep** fields only, the number of owned objects is bounded by the number of **rep** fields. To remove this restriction, the approach supports *set-valued ghost rep fields*. Specifically, the approach allows the programmer to declare *ghost fields*, which are fields required for static verification but not for execution. A statement that uses a ghost field must be in an annotation (and a statement in an annotation must not have side-effects on non-ghost state). A ghost field may be of a special type $Set\langle T\rangle$, which means that instead of holding a single value of type T, it holds a set of such values. If a **rep** field is set-valued, all objects referenced by the field are considered to be *rep* objects. This allows a *LinkedList* object to own all of its nodes, even though at run time it holds direct references to the sentinel nodes only.

The example also shows how the approach supports ownership transfer. Method *append* transfers ownership of the non-sentinel nodes of *other* to **this**.

4.4.3 Static verification

Figure 4.10 shows the translation of source program commands to input for the sequential program verifier.

Note that the verification-time commands for a **synchronized** (o) block havoc all objects that are not in the thread's access set, rather than just object o. This is necessary since other threads may have modified not just o, but o's transitively owned objects as well.

The assumptions encoded by the **assume** statements are justified by the programming model.

The verifier is additionally made aware of the following properties:

$$(\forall o \bullet o.inv \Rightarrow Inv(o))$$
$$(\forall o, p \bullet o.inv \wedge p \in \text{repobjects}(o) \Rightarrow p.inv)$$

These are guaranteed to hold in each program state by the programming model, as explained above.

4.5 Immutable objects

In this section we briefly describe how the approach we implemented supports sharing immutable objects without synchronization.

If after an object is shared, it is only ever inspected and never mutated, then there's no need to synchronize accesses. Our approach supports this by replacing a thread's access set with a *read set* and a *write set*, and by replacing the shared set

$o := \textbf{new}\ C;\ \equiv$
- $o \leftarrow \textbf{new}\ C;$
- $\textbf{assume}\ o \notin S;$
- $\textbf{tid}.A \leftarrow \textbf{tid}.A \cup \{o\};$
- $o.inv \leftarrow \mathit{false};$

$\textbf{pack}\ o;\ \equiv$
- $\textbf{assert}\ o \in \textbf{tid}.A;$
- $\textbf{assert}\ \neg o.inv$
- $\textbf{assert}\ (\forall p \in \text{repobjects}(o) \bullet p \in \textbf{tid}.A \wedge p.inv);$
- $\textbf{assert}\ \mathit{Inv}(o);$
- $o.inv \leftarrow \mathit{true};$
- $\textbf{tid}.A \leftarrow \textbf{tid}.A \setminus \text{repobjects}(o);$

$\textbf{unpack}\ o;\ \equiv$
- $\textbf{assert}\ o \in \textbf{tid}.A;$
- $\textbf{assert}\ o.inv;$
- $o.inv \leftarrow \mathit{false};$
- $\textbf{tid}.A \leftarrow \textbf{tid}.A \cup \text{repobjects}(o);$

$x := o.f;\ \equiv$
- $\textbf{assert}\ o \in \textbf{tid}.A;$
- $x \leftarrow o.f;$

$o.f := x;\ \equiv$
- $\textbf{assert}\ o \in \textbf{tid}.A;$
- $\textbf{assert}\ \neg o.inv;$
- $\textbf{if}\ (f \text{ is } \textbf{shared})$
  - $\textbf{assert}\ x \in S;$
- $o.f \leftarrow x;$

$\textbf{share}\ (o, l);\ \equiv$
- $\textbf{assert}\ o \in \textbf{tid}.A;$
- $\textbf{assert}\ o.inv;$
- $\textbf{assert}\ o \notin S;$
- $o.lockLevel \leftarrow l;$
- $S \leftarrow S \cup \{o\};$
- $\textbf{tid}.A \leftarrow \textbf{tid}.A \setminus \{o\};$

$\textbf{synchronized}\ (o)\ B\ \equiv$
- $\textbf{assert}\ o \in S;$
- $\textbf{assert}\ o < \textbf{tid}.lockStack;$
- $\textbf{tid}.lockStack.push(o);$
- $\textbf{foreach}\ (p \notin \textbf{tid}.A)$
  - $\textbf{havoc}\ p.*;$
- $\textbf{tid}.A \leftarrow \textbf{tid}.A \cup \{o\};$
- $\textbf{assume}\ o.inv;$
- B
- $\textbf{assert}\ o \in \textbf{tid}.A;$
- $\textbf{assert}\ o.inv;$
- $\textbf{tid}.A \leftarrow \textbf{tid}.A \setminus \{o\};$
- $\textbf{tid}.lockStack.pop();$

Figure 4.10: Translation of source program commands to verification-time commands (with invariants and ownership).

with a *shared-as-lock-protected* set and a *shared-as-immutable* set. Correspondingly, the **share** command is replaced with a **share_lockprotected** command and a **share_immutable** command. Sharing an object as immutable requires that it is unshared and in the current thread's write set. It removes the object from the write set and adds it to each thread's read set (even if the thread has not yet been started). If the object has *rep* objects, they are recursively shared as immutable and added to all read sets.

Regardless of whether an object is shared as lock-protected or as immutable, at the point where it is shared it must be valid. As a result, an immutable object's invariant holds at all times.

Our approach supports writing classes that allow client code the freedom to use some of the class's objects as thread-local (unshared) objects, to share some and protect them by their lock, and to share some as immutable. Such a class typically provides inspector methods and mutator methods. Only inspector methods can be called on immutable objects, since mutator methods require that the receiver is in the write set.

The **unpack** o; command requires o to be in the thread's write set. To allow an inspector method to access its receiver's *rep* objects, regardless of whether the receiver is writable or only readable, our approach includes a **read** (o) block that adds o's *rep* objects to the thread's read set for the duration of the block. It also temporarily removes o itself from the write set (but not the read set); this is required for soundness.

4.6 Static fields and static initializers

In this section, we extend our programming model to also prevent data races on static fields and deadlocks involving class initialization and class lock acquisition. The extended model also enforces invariants on the static fields of a class and its transitively owned objects.

We first briefly recall the syntax and semantics of class initialization in Java. We then present the programming model extension. The remaining subsections discuss acyclicity of the **lock_before** relation and describe our support for classes whose static state does not change after initialization.

4.6.1 Class initialization in Java

In this subsection, we briefly recall the syntax and semantics of class initialization in the Java programming language.

A class may declare static field initializers and static initializer blocks (or *static initializers* for short). In the sequel, we assume that each class declares no static field initializers and exactly one static initializer. (It is always possible to rewrite

```
static_lockprotected class Counter {
  static int count;
  static invariant 0 ≤ count;

  static {
    // initialize Object;
    pack Counter.class;
    // share_lockprotected Counter.class;
  }

  static void increment()
    requires Counter.class < tid.lockStack;
  {
    // initialize Counter;
    synchronized (Counter.class) {
      unpack Counter.class;
      // initialize Counter;
      int c := count;
      // initialize Counter;
      count := c + 1;
      pack Counter.class;
    }
  }
}
```

(a)

```
static_immutable class Primes {
  static rep int[] primes;
  static invariant primes ≠ null ∧
    forall{int i in (0..primes.length − 2);
      primes[i] < primes[i + 1]};

  static {
    // initialize Object;
    primes := new int[] {2, 3, 5, 7, 11};
    pack Primes.class;
    // share_immutable Primes.class;
  }

  static int getThirdPrime()
    requires Primes.class < tid.lockStack;
  {
    // initialize Primes;
    read (Primes.class) {
      // initialize Primes;
      return primes[2];
    }
  }
}
```

(b)

Figure 4.11: Example illustrating the programming model's support for static fields and static initializers. Commands implicitly inserted by the model are shown in comments.

a class to satisfy this assumption.) A static initializer is an arbitrary sequence of statements.

Java ensures that a class's static initializer is executed at most once, at the last possible moment, and that it completes normally before any access of the class or one of its transitive subclasses (except for an access performed while the current thread is running the static initializer). The following are considered accesses of a class: reads and writes of static fields, calls of static methods, and calls of constructors.

We may think of this semantics as follows. In a preprocessing step, an **initialize** C; command is inserted before each access of class C in the program.[2] An **initialize** C; command is additionally inserted at the start of the static initializer of each direct subclass of C. A thread t executes an **initialize** C; command as follows:

- If no thread has started executing C's static initializer, thread t executes C's static initializer. If execution completes normally, the **initialize** operation completes normally. If execution completes abruptly with an exception, then so does the **initialize** operation.

- If thread t is currently executing C's static initializer, i.e. if this is a recursive **initialize** C; operation, then the operation completes normally directly, without recursively executing the static initializer.

- If some other thread is executing C's static initializer, thread t waits until the execution has completed (normally or abruptly). Once execution of C's static initializer has completed normally or abruptly, execution of the **initialize** C; operation by thread t completes normally or abruptly in the same way.

- Otherwise, if C's static initializer was already executed to completion earlier, then execution of the **initialize** C; operation by thread t completes normally or abruptly directly, in the same way that execution of the static initializer did.

Note that class initialization may deadlock, if threads are waiting for each other to finish executing a static initializer.

4.6.2 Programming model

In this section, we present a programming model that prevents data races on static fields and deadlocks involving class initialization and class lock acquisition, and that ensures invariants on the static fields of a class and its transitively owned objects.

[2]Actually, for calls of constructors and static methods, we insert the **initialize** operation at the top of the constructor or method body rather than at the call site. Both encodings are sound but the callee-side encoding yields slightly simpler method contracts.

Our general approach is to treat static fields of a class C as if they are fields of the *Class* object for C, denoted in Java as C.**class**. That is:

- We prevent data races on static fields by allowing a thread t to access a static field $C.f$ only if C.**class** is in t's access set, and by ensuring that access sets are always disjoint.
- An object C.**class** is accessible and unshared on entry to its static initializer. Upon normal completion of the static initializer, a **share** operation is implicitly performed on the object. Once C.**class** is shared, to access the static fields of C, a thread must first lock C.**class**, which adds C.**class** to the thread's access set.
- The lock acquisition deadlock prevention approach applies to *Class* objects as well. A class C may specify lower bounds for its *Class* object's lock level using **lock_before** D; declarations. C.**class**'s lock level is constructed to be above the lock levels of the classes mentioned in C's **lock_before** declarations. Cycles in the **lock_before** relation are not allowed. Also, the approach does not support specification of upper bounds for lock levels of *Class* objects.
- A class may declare some of its static fields as **rep**. The objects pointed to by a class's non-null *rep* fields are its *rep* objects. A class may declare a static invariant, which may depend on the class's static fields and the fields of its transitive *rep* objects. A *Class* object has an *inv* field, and the **pack** and **unpack** operations apply to *Class* objects as well as other objects. A *Class* object must be valid when it is shared and when it is unlocked.

We prevent deadlocks involving class initialization by applying the locking order to **initialize** operations as well, and by tracking static initializer executions in a thread's lock stack. Specifically, a thread is allowed to perform an **initialize** C; operation only if

- C.**class** is less than all objects on the thread's lock stack, or
- C.**class** is already shared (which implies the static initializer has already completed), or
- C.**class** is on the thread's lock stack (which implies that either the class is locked and therefore already shared or the thread is already executing the static initializer)

It follows that on entry to the static initializer, we have

$$C.\mathbf{class} < \mathbf{tid}.\mathit{lockStack} \quad .$$

Object C.**class** is pushed onto the lock stack for the duration of the static initializer's execution. Class C's static initializer may lock and then access classes that are less than C in the locking order.

An **initialize** C; operation ensures that if C.**class** is less than the objects on the lock stack, then in the post-state, C.**class** is shared.

An **initialize** operation's frame condition states that it does not modify any fields of any objects that are in the thread's access set in the operation's pre-state.

Notice that in this approach, a thread must trigger an **initialize** C; operation before it can acquire the lock of class C, since it needs to ensure that the class is shared. The easiest way to achieve this is by acquiring the lock inside a method of class C.

Figure 4.11 (a) illustrates the approach. It shows how an **initialize** command is inserted at the top of each static method and before each static field access. Validity of the **synchronized** command in method *increment* requires that *Counter*.**class** is shared. The preceding **initialize** command guarantees this, provided that *Counter*.**class** is not on the lock stack. This, in turn, is guaranteed by *increment*'s precondition.

4.6.3 lock_before acyclicity

The soundness of our approach requires that the **lock_before** relation is acyclic. If a run-time system ensures that the module import relation is acyclic, then acyclicity of the **lock_before** relation may be ensured by checking at compile time that the **lock_before** relation on the classes of the module being compiled is acyclic.

However, neither the Java virtual machine nor the Microsoft .NET Framework's CLR refuse to load modules that import each other. As a result, the modules themselves are responsible for detecting cycles in the locking order at run time.

In our approach, this is achieved by requiring the program to build a module lock order graph, whose nodes are modules, at run time. The graph is stored in a static field in a class called *LockOrder* in a special module that all modules of the program must import. The graph is initially empty. Whenever a thread requires a lock order edge between a class C in a module M_1 and a class D in a different module M_2, it must request it by performing a call

$$\mathit{LockOrder}.\mathit{checkEdge}(C.\mathbf{class}, D.\mathbf{class}) \quad .$$

This call first checks if a path already exists from M_1 to M_2. If so, the call returns normally. Otherwise, it checks if an edge from M_1 to M_2 would create a cycle. If so, the call throws an exception. Otherwise, the call adds the edge to the graph and returns normally.

4.6.4 Immutable class objects

The approach of the previous sections supports classes whose static fields are protected by locks. It is easy to extend the approach with more efficient support for classes whose static state is not modified after initialization, by allowing the immutable objects approach of Section 4.5 to be applied to *Class* objects as well.

A class must declare whether its *Class* object is shared as lock-protected or as immutable. Depending on this declaration, the implicit **share** operation at the end of the static initializer is either a **share_lockprotected** or a **share_immutable** operation.

An **initialize** C; operation ensures that if C.**class** is less than the objects on the lock stack, then in the post-state, C.**class** is shared as declared.

Figure 4.11 (b) illustrates the approach. Recall that validity of the array element access in method *getThirdPrime* requires that the array is in the read set. The array is inserted into the read set by the **read** command (see Section 4.5). The **read** command, in turn, requires that *Primes*.**class** is in the read set. This follows from the fact that it is immutable. The **initialize** command preceding the **read** command guarantees that *Primes*.**class** is immutable, provided that it is not on the lock stack. This, finally, is guaranteed by method *getThirdPrime*'s precondition.

4.7 Experience

To verify the applicability of our approach to realistic, useful programs, we implemented it in a custom build of the Spec# program verifier [9] and used it to verify a chat server application written in C# with annotations inserted in the form of specially marked comments. The application verifies successfully; this guarantees the following:

- The program is free from data races and deadlocks
- Object invariants, loop invariants, method preconditions and postconditions, and assert statements declared by the program hold
- The program is free from null dereferences, array index out of bounds errors, and typecasting errors
- The program is free from races on platform resources such as network sockets. This is achieved by enforcing concurrency contracts on the relevant API methods.

Table 4.1 shows the annotation overhead of four programs which we annotated and verified. Programs `chat` and `phone` were derived from the ones used in [16].

The prototype verifier and the sample programs are available at the author's home page at http://www.cs.kuleuven.be/~bartj/.

Program	Lines of Code	Lines Changed or Added	Overhead
chat	344	117	34%
phone	222	50	23%
prod-cons	84	24	29%
philosophers	64	21	33%

Table 4.1: Annotation overhead

4.8 Related work

The Extended Static Checkers for Modula-3 [23] and for Java [32] attempt to statically find errors in object-oriented programs. These tools include support for the prevention of data races and deadlocks. For each field, a programmer can designate which lock protects it. However, these two tools trade soundness for ease of use; for example, they do not take into consideration the effects of other threads between regions of exclusion. Moreover, various engineering trade-offs in the tools notwithstanding, the methodology used by the tools was never formalized enough to allow a soundness proof.

Method specifications in our methodology pertain only to the pre-state and post-state of method calls. Some systems [77, 36] additionally support specification and verification of the atomic transactions performed during a method call. We focus on verification of object invariants, which does not require such specifications. Note that [77, 36] have no support for layered object structures, such as supported by the Boogie ownership system.

A number of type systems have been proposed that prevent data races in object-oriented programs. For example, Boyapati *et al.* [16] parameterize classes by the protection mechanism that will protect their objects against data races. The type system supports thread-local objects, objects protected by a lock (its own lock or its root owner's lock), read-only objects, and unique pointers. The system does not support forms of ownership transfer other than transfer of unique pointers. For example, it cannot type the program of Figure 4.9. Also, the type system does not support object invariants.

Boyapati *et al.* prevent deadlocks by allowing the developer to declare a fixed set of lock levels. Lock levels are assigned to objects as type arguments. Additional expressiveness is gained by supporting locking the nodes of a mutable tree data structure or an immutable DAG data structure, and by ordering the objects of designated classes at run time.

We enable sequential reasoning and ensure consistency of aggregate objects by preventing data races. Some authors propose pursuing a different property, called

atomicity, either through dynamic checking [31], by way of a type system [34], or using a theorem prover [79]. An atomic method can be reasoned about sequentially. However, we enable sequential reasoning even for non-atomic methods, by assuming only the object invariant for a newly acquired object (see Figure 4.10). Also, in [34] the authors claim that data-race-freedom is unnecessary for sequential reasoning. It is true that some data races are benign, even in the Java and C# memory models; however, the data races allowed in [34] are generally not benign in these memory models; indeed, the authors prove soundness only for sequentially consistent systems, whereas we prove soundness for the Java memory model, which is considerably weaker.

Ábrahám-Mumm *et al.* [1] propose an assertional proof system for Java's reentrant monitors. It supports object invariants, but these can depend only on the fields of **this**. No claim of modular verification is made.

The rules in our methodology that an object must be valid when it is released, and that it can be assumed to be valid when it is acquired, are taken from Hoare's work on monitors and monitor invariants [39].

There are also tools that try dynamically to detect violations of safe concurrency. A notable example is Eraser [81]. It finds data races by looking for locking-discipline violations. The tool has been effective in practice, but does not come with guarantees about the completeness nor the soundness of the method.

In the straightforward implementation proposed in this chapter, mutual exclusion is achieved through coarse-grained locking. However, the methodology allows one to use other semantically equivalent techniques that may perform better under particular contention patterns, while preserving the same reasoning framework and safety guarantees. Possible alternatives include fine-grained locking of the objects within an ownership domain, or a form of optimistic concurrency, such as transactional monitors [85].

Leino and Müller [63, 60] propose an approach for verification of programs with static class invariants. Contrary to our work, they support neither multithreading nor Java and C#'s lazy class initialization semantics. Also, our approach is more flexible in terms of method effect framing w.r.t. static fields. Furthermore, our **lock_before** relation improves on the import order of [63], in that a) we do not restrict accessing classes as such, and b) contrary to the class import order, the **lock_before** order is consistent with the module import order, which is easier to understand for deployers and checking its acyclicity at run time is more efficient.

The present approach evolved from [42] and [48]. [48] improved upon [42] by supporting standard locking primitives, by preventing deadlocks, by supporting immutable objects, and by reporting on experience gained using a prototype verifier. The present text improves on [48] by adding support for static fields and static initializers.

4.9 Conclusion

We propose a programming model for concurrent programming in Java-like languages, and the design of a set of program annotations that make the use of the programming model explicit and that enable automated verification of compliance. Our programming model ensures absence of data races and deadlocks, and provides a sound approach for local reasoning about program behavior. We have prototyped the verifier as a custom build of the Spec# programming system. Through a case study we show that the model supports non-trivial, useful programs, and we assess the annotation overhead.

A number of areas of future work remain, including further extending the programming model to encompass lock re-entry and read-write locks, reducing the annotation overhead, and obtaining further experience.

Chapter 5

Formal Development

This chapter presents a formal development of the core of the approach for verification of concurrent programs; specifically, it formalizes and proves the soundness of the approach for verification of data-race-freedom proposed in Section 4.2.

The approach is formalized in two steps. First, the programming model that prevents data races is formalized without regard to modular static verification. Specifically, a syntax for a core subset of Java, augmented with the annotations required by the approach, is defined, as well as a small step semantics for execution of programs in this syntax that tracks the extra state variables (specifically, access sets and shared sets) required by the programming model. Based on this extra state, a set of *legal program states* that comply with the programming model is defined. It is proven that if a program reaches only legal states, then it is data-race-free.

The formalized programming model differs in one minor respect from the one in Section 4.2: in Section 4.2, lock re-entry is prevented by enforcing the invariant that a thread holds a shared object's lock if and only if the object is in the thread's access set; in the formalized model, this invariant is not enforced; rather, the set of objects locked by a thread is tracked separately. This results in a slightly cleaner formal development.

In the second step, the modular static verification approach is formalized. The rules for verification condition generation are given and proven sound by defining a notion of valid program states and proving that valid program states are legal and execution steps transform valid program states to valid program states. The method effect framing approach based on *required access sets* is proven using the notion of *activation record access sets*.

It is explained how the approach enables modular sequential reasoning about concurrent programs. *Modular* reasoning means that one can reason about the correctness of a method body relying only on specifications of the methods that the given method body depends on. *Sequential* reasoning means that interference

by other threads needs to be taken into account only at synchronization points.

5.1 Programming model

This section formalizes the programming model, without regard to modular static verification. Section 5.1.1 formalizes a subset of Java augmented with the annotations required by the approach. Section 5.1.2 defines a small step semantics for execution of programs in this language, which tracks the extra state variables (specifically, access sets and shared sets) required by the programming model. The section also defines a set of *legal program states* that comply with the programming model. Section 5.1.3 proves that programs that reach only legal program states are data-race-free.

5.1.1 Programs

To formalize the rules imposed by our programming model, we first define a small language consisting of a subset of Java (minus static typing) plus two kinds of annotations (indicated by the gray background): share statements and method contracts. Its syntax is shown in Figure 5.1. A rendering of the example program of Figure 4.2 in this language is shown in Figure 5.2. We discuss the language and define *well-formedness* of programs.

$$C \in \mathcal{C}, I \in \mathcal{I}, m \in \mathcal{M}, f \in \mathcal{F}, x \in \mathcal{X}, o \in \mathcal{O}$$

$$
\begin{array}{rcl}
\pi & ::= & \langle \mathit{iface} \mid \mathit{class} \rangle^* \ s^* \\
\mathit{iface} & ::= & \textbf{interface}\ I\ \{\ \mathit{mh}^*\ \} \\
\mathit{mh} & ::= & m(x^*)\ \textbf{requires}\ \mathit{pred}^*;\ \textbf{ensures}\ \mathit{pred}^*; \\
g & ::= & \textbf{this} \mid x \mid \mathsf{null} \mid \underline{o} \\
\mathit{pred} & ::= & \langle g \mid \textbf{result} \rangle.\langle \textbf{canAccess} \mid \textbf{canLock} \mid \textbf{canShare} \rangle \\
& & \mid \textbf{threadholdsnolocks} \\
\mathit{class} & ::= & \textbf{class}\ C\ \textbf{implements}\ I^*\ \{\ \mathit{field}^*\ \mathit{meth}^*\ \} \\
\mathit{field} & ::= & f; \\
\mathit{meth} & ::= & \mathit{mh}\ \{\ s^*\ \} \\
\tau & ::= & C \mid I \\
s & ::= & \textbf{if}\ (g = g)\ \{\ s^*\ \}\ \textbf{else}\ \{\ s^*\ \} \mid \textbf{assert}\ g\ \textbf{instanceof}\ \tau; \\
& & \mid x := g.f_C; \mid g.f_C := g; \mid x := \textbf{new}\ C; \mid x := g.m_I(g^*); \mid \textbf{start}\ g.m_I(); \\
& & \mid \textbf{share}\ g; \mid \textbf{synchronized}\ (g)\ \{\ s^*\ \} \\
& & \mid \textbf{return}\ g; \mid \underline{x := \textbf{receive}\ (\mathit{pred}^*, \mathit{pred}^*, o^*, o^*, o^*);} \mid \underline{\textbf{unlock}\ o;}
\end{array}
$$

Figure 5.1: Syntax of a small Java-like language without static typing, but with two kinds of annotations (indicated by the gray background): method contracts and share statements. The underlined elements appear only as part of continuations during program execution (see Section 5.1.2) and are not allowed in well-formed programs

```
class Counter { count; }

interface Runnable { run() requires threadholdsnolocks ∧ this.canAccess; }

class Session implements Runnable {
  counter;
  run()
    requires threadholdsnolocks ∧ this.canAccess;
  {
    /* wait for event */;
    c := this.counter_Session;
    assert c instanceof Counter;
    synchronized (c) { n := c.count_Counter; c.count_Counter := n + 1; }
    this.run_Runnable();
  }
}

c := new Counter;
share c;
s := new Session; s.counter_Session := c; start s.run_Runnable();
s := new Session; s.counter_Session := c; start s.run_Runnable();
```

Figure 5.2: The example program of Figure 4.2, rendered in the formal syntax of Figure 5.1. (Note: the example also uses integer values and integer operations; these are omitted from the formal development for simplicity.)

A program consists of a number of classes and interfaces and a main routine. Each method declared within such a class or interface has a corresponding method contract, consisting of a *precondition* (**requires** clause) and a *postcondition* (**ensures** clause). The precondition and postcondition specify an assertion that must hold in the pre-state and the post-state of the method call, respectively. Since method contracts are used only for verifying modularly whether a given program complies with the programming model, and are not part of the programming model itself, it is safe to ignore them in this section.

In addition to method contracts, the syntax supports a second type of annotations, namely *share statements*. Using share statements, a programmer can explicitly indicate when an object transitions from unshared to shared.

The language is not statically typed. Rather, to avoid formalizing a type system, type mismatches are considered run-time errors. The static verification approach detailed in Section 5.2 guarantees the absence of programming model violations as well as run-time errors (i.e., null dereferences and type mismatches).

Our static verification approach performs modular verification. This means that for each method, all executions of that method are considered, not just those that occur in executions of the program. For example, for method *run* of class *Session* in Figure 5.2, all executions that satisfy *run*'s precondition are considered, including those where the *count* field contains a null value, even though there is

no execution of the program of Figure 5.2 where the method is called in a state where *count* is null. It follows that in the absence of the **assert** statement, the method would be considered invalid since in an execution where *count* is null, the **synchronized** statement would cause a null dereference. This limits the usefulness of the approach, since users are not interested in errors that do not occur in program executions. To alleviate this limitation, the programmer may restrict the set of executions considered by the static verification approach, by inserting **assert** statements into the program. If an **assert** statement's condition evaluates to false, the statement is said to *fail*. If in a given execution an **assert** statement fails, the execution is *stuck* (in Java, an exception is thrown), but for purposes of static verification, the execution is considered to be valid (or in other words, the execution is not considered further), since the failure is assumed to mean that the method execution never appears in an execution of the whole program. Other verification approaches outside the scope of this thesis, such as code review, model checking, or testing, may be used to verify such assumptions. (Note that the **assert** statements of this chapter correspond to the **assume** statements of earlier chapters. In this chapter we use **assert** syntax since this is the existing syntax of the Java language.)

We only consider *well-formed* programs. Well-formed programs have no name clashes. Also, local variables need not be declared before they are assigned, but they must be assigned before they are used. By requiring that both branches of a conditional statement assign to the same variables, we can define a notion of *free variables* at each program point by considering an assignment to *bind* the variable occurrences that occur after it in the control flow and that are not hidden by a later assignment. Further, a well-formed program must not contain any of the syntactic forms that are intended to appear only during program execution. Lastly, classes must implement all methods of their declared interfaces, method and field names that appear in the program must be declared by the type with which they are annotated, and if a method is used to start a new thread, its precondition must be as prescribed.

Definition 5.1. *A program* π *is* well-formed *if all of the following hold:*

- *Interface and class names are unique. Field and method names are unique within an interface or class (no overloading). Method parameters are unique within a method.*

- *If one branch of an* **if** *statement directly or indirectly contains an assignment to a variable* x*, then so does the other branch.*

- *If a statement uses a variable* x*, then an assignment to* x *appears before the statement in the method body (ignoring the other branch of an enclosing* **if** *statement), or* x *is a parameter of the enclosing method or* **this***.*

- *The last statement of a method body and of the main routine is a* **return** *statement and a* **return** *statement does not appear anywhere else.*
- *The program does not contain any* **receive** *and* **unlock** *statements or object references.*
- *Each class implements every method of the interfaces it implements. The header of the implementing method is syntactically identical to the header of the implemented method. A class does not define additional methods.*
- *Each interface I and each class C mentioned in the program is declared by the program. If the program mentions a method m_I, then interface I declares method m. If the program mentions a field f_C, then class C declares field f.*
- *The number of arguments specified in a call equals the number of parameters declared by the corresponding method.*
- *If a method is mentioned in a* **start** *statement, then its requires clause is exactly* **threadholdsnolocks** $\wedge$ **this.canAccess**.

The example program of Figure 5.2 is well-formed.

5.1.2 Program executions

We now formalize the semantics of our annotated Java subset. While remaining faithful to Java semantics, our semantics additionally tracks the extra state variables (specifically, access sets and shared sets) required by the programming model. We also define a set of *legal program states*. A program state is legal if all thread states are legal. A thread is in a legal state if it is not about to violate the programming model or cause a run-time error (in particular, a null dereference or a type mismatch). In Section 5.1.3, we show that programs that reach only legal states are data-race-free.

We use the following notation. $\mathcal{O}$ denotes the set of object references, $\mathcal{F}$ the set of field names, $\mathcal{M}$ the set of method names, $\mathcal{T}$ the set of thread identifiers and $\mathcal{C}$ the set of class names. Furthermore, v represents a value (i.e. $v \in \mathcal{O} \cup \{\mathsf{null}\}$) and o represents an object reference (i.e., a non-null value). $\mathbf{fields}_\pi(C)$ represents the set of fields declared by class C. We use $C \prec_\pi I$ to denote that class C implements interface I. $\mathbf{mbody}_\pi(C, m)$ denotes the body of method m declared in class C. $\mathsf{pre}(I, m)$ and $\mathsf{post}(I, m)$ denote the precondition and postcondition, respectively, declared by method m in interface I. Also, we assume the existence of a function c that maps object references to class names. We assume that for each class name $C \in \mathcal{C}$, there are infinitely many object references $o \in \mathcal{O}$ such that $\mathsf{c}(o) = C$. $\mathsf{objectRefs}(\phi)$ and $\mathsf{free}(\phi)$ denote the free object references and free variables, respectively, in syntactic entity ϕ. Finally, we use $f[x \mapsto y]$ to

denote the update of the function f at argument x with value y. Specifically, $f[x \mapsto y]$ is identical to f, except that it maps x to y. Similarly, $f \setminus [x \mapsto y]$ removes the mapping of x to y from f, and thereby removes x from the domain of f. We use the notation $\overline{s}[v/x]$ to denote substitution of a value for a program variable in a thread continuation (i.e., list of statements) $\overline{s}$. This substitution replaces only the free occurrences of x, i.e., the ones that are not bound by an assignment inside $\overline{s}$.

The dynamic semantics is defined as a small step relation on *program states.*

Definition 5.2. *A* program state

$$\sigma = (\mathsf{H}, \mathsf{L}, \mathsf{S}, \mathsf{T})$$

consists of:

- *the heap* H*, a partial function that maps object references to object states. An object state is a partial function that maps field names to values.*

$$\mathsf{H} : \mathcal{O} \hookrightarrow (\mathcal{F} \hookrightarrow \mathcal{O} \cup \{\mathsf{null}\})$$

  The domain of H *consists of all allocated objects. The domain of an object state* $\mathsf{H}(o)$ *consists of the declared fields of the class* $\mathsf{c}(o)$ *of* o*.*

- *the lock map* L*, a partial function that maps a locked object to the identifier of the thread that holds the lock*

$$\mathsf{L} : \mathcal{O} \hookrightarrow \mathcal{T}$$

- *the shared set* S*, the set of shared objects*

- *the thread set* T*. Each thread state* $(\mathsf{tid}, \mathsf{A}, \mathsf{F}) \in \mathsf{T}$ *consists of a unique thread identifier* $\mathsf{tid} \in \mathcal{T}$*, an access set* $\mathsf{A} \subseteq \mathcal{O}$ *and a list of activation records* $\mathsf{F} \in (s^*)^*$*.*

Figures 5.3 and 5.4 show the definition of the small step relation $\rightarrow$ on program states, as well as the definition of *legality* $\mathsf{H}, \mathsf{L}, \mathsf{S} \vdash t : \mathsf{legal}$ of a thread state t. Legality captures the rules of the programming model, as well as absence of run-time errors (i.e., null dereferences and type mismatches).

The rule [IF] is standard. An **assert** statement that fails (either because the operand is null or because it is not of the specified type) causes the thread to block forever ([ASSERT]). For reading ([READ]) or writing ([WRITE]) a field f, the target object must be non-null, part of the current thread's access set, and of the class that declares the field. Note that field updates change the heap: the old value of the field is replaced with the new value. When creating a new object ([NEW]), an unused object reference is chosen from $\mathcal{O}$, is inserted into the heap

$$[\textsc{Legal-If}]\quad \mathsf{H},\mathsf{L},\mathsf{S} \vdash (\mathsf{tid},\mathsf{A},(\textbf{if}\ (v_1 = v_2)\ \{\ \overline{s}_1\ \}\ \textbf{else}\ \{\ \overline{s}_2\ \}\ \overline{s})\cdot \mathsf{F}) : \mathsf{legal}$$

$$[\textsc{If}]\quad \frac{v_1 = v_2 \Rightarrow \overline{s}' = \overline{s}_1 \qquad v_1 \neq v_2 \Rightarrow \overline{s}' = \overline{s}_2}{(\mathsf{H},\mathsf{L},\mathsf{S},\mathsf{T} \triangleleft (\mathsf{tid},\mathsf{A},(\textbf{if}\ (v_1 = v_2)\ \{\ \overline{s}_1\ \}\ \textbf{else}\ \{\ \overline{s}_2\ \}\ \overline{s})\cdot \mathsf{F})) \rightarrow (\mathsf{H},\mathsf{L},\mathsf{S},\mathsf{T} \triangleleft (\mathsf{tid},\mathsf{A},(\overline{s}'\ \overline{s})\cdot \mathsf{F}))}$$

$$[\textsc{Legal-Assert}]\quad \mathsf{H},\mathsf{L},\mathsf{S} \vdash (\mathsf{tid},\mathsf{A},(\textbf{assert}\ v\ \textbf{instanceof}\ \tau;\ \overline{s})\cdot \mathsf{F}) : \mathsf{legal}$$

$$[\textsc{Assert}]\quad \frac{v \neq \mathsf{null} \qquad c(v) \preceq_\pi \tau}{(\mathsf{H},\mathsf{L},\mathsf{S},\mathsf{T} \triangleleft (\mathsf{tid},\mathsf{A},(\textbf{assert}\ v\ \textbf{instanceof}\ \tau;\ \overline{s})\cdot \mathsf{F})) \rightarrow (\mathsf{H},\mathsf{L},\mathsf{S},\mathsf{T} \triangleleft (\mathsf{tid},\mathsf{A},(\overline{s})\cdot \mathsf{F}))}$$

$$[\textsc{Legal-Read}]\quad \frac{v \neq \mathsf{null} \qquad v \in \mathsf{A} \qquad c(v) = C}{\mathsf{H},\mathsf{L},\mathsf{S} \vdash (\mathsf{tid},\mathsf{A},(x := v.f_C;\ \overline{s})\cdot \mathsf{F}) : \mathsf{legal}}$$

$$[\textsc{Read}]\quad (\mathsf{H},\mathsf{L},\mathsf{S},\mathsf{T} \triangleleft (\mathsf{tid},\mathsf{A},(x := v.f_C;\ \overline{s})\cdot \mathsf{F})) \rightarrow (\mathsf{H},\mathsf{L},\mathsf{S},\mathsf{T} \triangleleft (\mathsf{tid},\mathsf{A},(\overline{s}[H(v,f)/x])\cdot \mathsf{F}))$$

$$[\textsc{Legal-Write}]\quad \frac{v_1 \neq \mathsf{null} \qquad v_1 \in \mathsf{A} \qquad c(v_1) = C}{\mathsf{H},\mathsf{L},\mathsf{S} \vdash (\mathsf{tid},\mathsf{A},(v_1.f_C := v_2;\ \overline{s})\cdot \mathsf{F}) : \mathsf{legal}}$$

$$[\textsc{Write}]\quad (\mathsf{H},\mathsf{L},\mathsf{S},\mathsf{T} \triangleleft (\mathsf{tid},\mathsf{A},(v_1.f_C := v_2;\ \overline{s})\cdot \mathsf{F})) \rightarrow (\mathsf{H}[(v_1,f) \mapsto v_2],\mathsf{L},\mathsf{S},\mathsf{T} \triangleleft (\mathsf{tid},\mathsf{A},(\overline{s})\cdot \mathsf{F}))$$

$$[\textsc{Legal-New}]\quad \mathsf{H},\mathsf{L},\mathsf{S} \vdash (\mathsf{tid},\mathsf{A},(x := \textbf{new}\ C;\ \overline{s})\cdot \mathsf{F}) : \mathsf{legal}$$

$$[\textsc{New}]\quad \frac{o \notin \mathit{dom}(\mathsf{H}) \qquad \mathsf{c}(o) = C \qquad \textbf{fields}_\pi(C) = \{f_1,...,f_n\}}{\begin{array}{c}(\mathsf{H},\mathsf{L},\mathsf{S},\mathsf{T} \triangleleft (\mathsf{tid},\mathsf{A},(x := \textbf{new}\ C;\ \overline{s})\cdot \mathsf{F}))\\ \rightarrow (\mathsf{H}[o \mapsto (f_1 \mapsto \mathsf{null},...,f_n \mapsto \mathsf{null})],\mathsf{L},\mathsf{S},\mathsf{T} \triangleleft (\mathsf{tid},\mathsf{A} \cup \{o\},(\overline{s}[o/x])\cdot \mathsf{F}))\end{array}}$$

$$[\textsc{Legal-Share}]\quad \frac{v \neq \mathsf{null} \qquad v \in \mathsf{A} \qquad v \notin \mathsf{S}}{\mathsf{H},\mathsf{L},\mathsf{S} \vdash (\mathsf{tid},\mathsf{A},(\textbf{share}\ v;\ \overline{s})\cdot \mathsf{F}) : \mathsf{legal}}$$

$$[\textsc{Share}]\quad (\mathsf{H},\mathsf{L},\mathsf{S},\mathsf{T} \triangleleft (\mathsf{tid},\mathsf{A},(\textbf{share}\ v;\ \overline{s})\cdot \mathsf{F})) \rightarrow (\mathsf{H},\mathsf{L},\mathsf{S} \cup \{v\},\mathsf{T} \triangleleft (\mathsf{tid},\mathsf{A} \setminus \{v\},(\overline{s})\cdot \mathsf{F}))$$

$$[\textsc{Legal-Synchronized}]\quad \frac{v \neq \mathsf{null} \qquad v \notin \mathsf{L}^{-1}(\mathsf{tid}) \qquad v \in \mathsf{S}}{\mathsf{H},\mathsf{L},\mathsf{S} \vdash (\mathsf{tid},\mathsf{A},(\textbf{synchronized}(v)\ \{\ \overline{s}'\ \}\ \overline{s})\cdot \mathsf{F}) : \mathsf{legal}}$$

$$[\textsc{Synchronized}]\quad \frac{v \notin \mathit{dom}(\mathsf{L})}{\begin{array}{c}(\mathsf{H},\mathsf{L},\mathsf{S},\mathsf{T} \triangleleft (\mathsf{tid},\mathsf{A},(\textbf{synchronized}(v)\ \{\ \overline{s}'\ \}\ \overline{s})\cdot \mathsf{F}))\\ \rightarrow (\mathsf{H},\mathsf{L}[v \mapsto \mathsf{tid}],\mathsf{S},\mathsf{T} \triangleleft (\mathsf{tid},\mathsf{A} \cup \{v\},(\overline{s}'\ \textbf{unlock}\ v;\ \overline{s})\cdot \mathsf{F}))\end{array}}$$

$$[\textsc{Legal-Unlock}]\quad \frac{o \in \mathsf{A} \qquad (o \mapsto \mathsf{tid}) \in L}{\mathsf{H},\mathsf{L},\mathsf{S} \vdash (\mathsf{tid},\mathsf{A},(\textbf{unlock}\ o;\ \overline{s})\cdot \mathsf{F}) : \mathsf{legal}}$$

$$[\textsc{Unlock}]\quad (\mathsf{H},\mathsf{L},\mathsf{S},\mathsf{T} \triangleleft (\mathsf{tid},\mathsf{A},(\textbf{unlock}\ o;\ \overline{s})\cdot \mathsf{F})) \rightarrow (\mathsf{H},\mathsf{L} \setminus [o \mapsto \mathsf{tid}],\mathsf{S},\mathsf{T} \triangleleft (\mathsf{tid},\mathsf{A} \setminus \{o\},(\overline{s})\cdot \mathsf{F}))$$

Figure 5.3: Legal thread states and execution steps. $(\mathsf{T} \triangleleft t = \mathsf{T} \cup \{t\})$ (*Continued in Figure 5.4.*)

$$\text{[Legal-Call]}\ \frac{v \neq \mathsf{null} \qquad \mathsf{c}(v) \prec_\pi I}{\mathsf{H}, \mathsf{L}, \mathsf{S} \vdash (\mathsf{tid}, \mathsf{A}, (x := v.m_I(\overline{v});\ \overline{s}) \cdot \mathsf{F}) : \mathsf{legal}}$$

$$\text{[Call]}\ \frac{\overline{s}' = \mathbf{mbody}_\pi(\mathsf{c}(v), m)[v/\mathbf{this}, \overline{v}/\overline{x}] \quad P' = \mathsf{pre}(I, m)[v/\mathbf{this}, \overline{v}/\overline{x}] \qquad Q' = \mathsf{post}(I, m)[v/\mathbf{this}, \overline{v}/\overline{x}]}{\begin{array}{c}(\mathsf{H}, \mathsf{L}, \mathsf{S}, \mathsf{T} \triangleleft (\mathsf{tid}, \mathsf{A}, (x := v.m_I(\overline{v});\ \overline{s}) \cdot \mathsf{F})) \\ \rightarrow (\mathsf{H}, \mathsf{L}, \mathsf{S}, \mathsf{T} \triangleleft (\mathsf{tid}, \mathsf{A}, (\overline{s}') \cdot (x := \mathsf{receive}(P', Q', A, S, L^{-1}(\mathsf{tid}));\ \overline{s}) \cdot \mathsf{F}))\end{array}}$$

$$\text{[Legal-Return]}\ \mathsf{H}, \mathsf{L}, \mathsf{S} \vdash (\mathsf{tid}, \mathsf{A}, (\mathbf{return}\ \mathsf{v};) \cdot \mathsf{F}) : \mathsf{legal}$$

$$\text{[Return]}\ \begin{array}{c}(\mathsf{H}, \mathsf{L}, \mathsf{S}, \mathsf{T} \triangleleft (\mathsf{tid}, \mathsf{A}, (\mathbf{return}\ v;) \cdot (x := \mathsf{receive}\ (_, _, _, _, _);\ \overline{s}) \cdot \mathsf{F})) \\ \rightarrow (\mathsf{H}, \mathsf{L}, \mathsf{S}, \mathsf{T} \triangleleft (\mathsf{tid}, \mathsf{A}, (\overline{s}[v/x]) \cdot \mathsf{F}))\end{array}$$

$$\text{[Legal-NewThread]}\ \frac{v \neq \mathsf{null} \qquad v \in A \qquad \mathsf{c}(v) \prec_\pi I}{\mathsf{H}, \mathsf{L}, \mathsf{S} \vdash (\mathsf{tid}, \mathsf{A}, (\mathbf{start}\ v.m_I();\ \overline{s}) \cdot \mathsf{F}) : \mathsf{legal}}$$

$$\text{[NewThread]}\ \frac{\forall (\mathsf{t}, _, _) \in \mathsf{T} \bullet \mathsf{tid}' \neq \mathsf{t} \qquad \mathsf{tid}' \neq \mathsf{tid}}{\begin{array}{c}(\mathsf{H}, \mathsf{L}, \mathsf{S}, \mathsf{T} \triangleleft (\mathsf{tid}, \mathsf{A}, (\mathbf{start}\ v.m_I();\ \overline{s}) \cdot \mathsf{F})) \\ \rightarrow (\mathsf{H}, \mathsf{L}, \mathsf{S}, \mathsf{T} \triangleleft (\mathsf{tid}', \{v\}, (\mathbf{mbody}_\pi(\mathsf{c}(v), m))) \triangleleft (\mathsf{tid}, \mathsf{A} \setminus \{v\}, (\overline{s}) \cdot \mathsf{F}))\end{array}}$$

Figure 5.4: Legal thread states and execution steps. ($\mathsf{T} \triangleleft t = \mathsf{T} \cup \{t\}$) (*Continued from Figure 5.3.*)

and all its fields are initialized to the default value null. New objects are initially only accessible to their creator and therefore the reference is added to the creating thread's access set. A thread may share ([Share]) an unshared object in its access set. By doing so, it removes the object from its access set and adds it to the global shared set S. Shared objects may be locked ([Synchronized]) provided they are not locked yet. For the duration of the synchronized block, the lock map L marks the current thread as holder of the lock. The object is also added to the access set. When the end of the synchronized block is reached ([Unlock]), the object must still be in the thread's access set. At this time, the object is removed from the access set and its corresponding lock is released. Invoking a method m ([Call]) within a thread t results in the addition of a new activation record to t's call stack. The activation record contains the body of m where all variables (**this** and parameters) are replaced with the corresponding argument values. In the caller's activation record, the method invocation is replaced by a **receive** statement, which keeps a record of the call's precondition and postcondition, the thread's access set, the global shared set and the set of objects locked by the thread. The operands of the **receive** statement are not used by the dynamic semantics; they are used only in the soundness proof in Section 5.2.4. When a method call returns ([Return]), the top activation record is popped and the return value is substituted into the caller's activation record. A new thread ([NewThread])

is started by performing a **start** $o.m_I()$; operation. Accessibility of object o is transferred from the original thread to the new one. The new thread consists of a single activation record containing the body of method m where **this** is replaced by o.

Program execution starts in an *initial state*.

Definition 5.3. *In a program's* initial state*, there is only a single thread. It has an empty access set and it executes the main routine $\overline{s}$. Moreover, the heap, the lock map, and the shared set are all empty.*

$$\mathsf{initial}_\pi((\emptyset, \emptyset, \emptyset, \{(\mathsf{tid}, \emptyset, \overline{s})\}))$$

The programs allowed by the programming model are the *legal programs*, which reach only *legal program states*.

Definition 5.4. *A program state* $(\mathsf{H}, \mathsf{L}, \mathsf{S}, \mathsf{T})$ *is* legal *if all thread states are legal as per Figures 5.3 and 5.4:*

$$\mathsf{legal}(\mathsf{H}, \mathsf{L}, \mathsf{S}, \mathsf{T}) \Leftrightarrow (\forall t \in \mathsf{T} \bullet \mathsf{H}, \mathsf{L}, \mathsf{S} \vdash t : \mathsf{legal})$$

Definition 5.5. *A program* π *is* legal *(formally* $\mathsf{legal}(\pi)$*) if it is well-formed and all program states reached by execution of the program are legal:*

$$\mathsf{legal}(\pi) \Leftrightarrow \mathsf{wf}(\pi) \wedge (\forall \sigma \bullet (\emptyset, \emptyset, \emptyset, (\mathsf{tid}, \emptyset, (\overline{s}))) \rightarrow^* \sigma \Rightarrow \mathsf{legal}(\sigma))$$

where $\overline{s}$ *is the main routine of program* π *and* tid *is an arbitrary thread identifier.*

5.1.3 Data-race-freedom

The dynamic semantics defined above is an *interleaving* semantics, which implies a *sequentially consistent* memory model. This means that there is a total order on all field accesses such that each read of a field $o.f$ yields the value most recently written to $o.f$ in this total order. However, the Java Language Specification, Third Edition (JLS3) [37] does not guarantee sequential consistency. In general, it allows threads to see writes performed by other threads *out of order*, which is necessary to allow efficient implementations involving optimizations at the level of the compiler, the memory hierarchy, and the processor. Still, JLS3 does guarantee sequential consistency for *correctly synchronized* programs. These are programs where all sequentially consistent executions are *data-race-free*.

In conclusion, in order for the dynamic semantics in this chapter to be sound with respect to Java and for the programs we care about, we need to prove that all executions of these programs under this semantics are data-race-free as per JLS3.

In this subsection, we show that legal programs are data-race-free, by proving that $\rightarrow$ maintains *well-formedness* of the program state. Specifically, each execution step maintains the property that the threads' access sets and the *free set* partition the heap.

Definition 5.6. *The* free set *of a program state* σ *consists of all shared objects that are not locked.*

$$\Phi = \mathsf{S} \setminus dom(\mathsf{L})$$

Definition 5.7. *A multiset of sets* S partitions *a set* T *(*$S \ll T$*) if all sets in* S *are disjoint and their union equals* T*.*

$$S \ll T \Leftrightarrow \forall s_1 \in S, s_2 \in S \setminus \{s_1\} \bullet s_1 \cap s_2 = \emptyset \wedge \bigcup S = T$$

Execution steps maintain well-formedness of the heap, the shared set, and the lock map.

Definition 5.8. *A heap is* well-formed *(*$\vdash \mathsf{H} : \mathsf{ok}$*) if objects referenced from fields are allocated.*

$$\forall o \in \mathrm{dom}(\mathsf{H}), f \in \mathrm{dom}(\mathsf{H}(o)) \bullet \mathsf{H}(o)(f) \in dom(\mathsf{H}) \cup \{\mathsf{null}\}$$

Definition 5.9. *A shared set is* well-formed *with respect to a heap (*$\mathsf{H} \vdash \mathsf{S} : \mathsf{ok}$*) if shared objects are allocated.*

$$\mathsf{S} \subseteq dom(\mathsf{H})$$

Definition 5.10. *A lock map is* well-formed *with respect to a heap and shared set (*$\mathsf{H}, \mathsf{S} \vdash \mathsf{L} : \mathsf{ok}$*) if locked objects are shared.*

$$dom(\mathsf{L}) \subseteq \mathsf{S}$$

Definition 5.11. *A program state*

$$\sigma = (\mathsf{H}, \mathsf{L}, \mathsf{S}, \mathsf{T})$$

is well-formed *(*$\mathsf{wf}(\sigma)$*) if the following conditions hold:*

- *The access sets and the free set partition the heap.*

$$((\biguplus_{(_,\mathsf{A},_) \in \mathsf{T}} \{\mathsf{A}\}) \uplus \{\Phi\}) \ll dom(\mathsf{H})$$

- *The heap, the shared set, and the lock map are well-formed.*

$$\vdash \mathsf{H} : \mathsf{ok} \;\wedge\; \mathsf{H} \vdash \mathsf{S} : \mathsf{ok} \;\wedge\; \mathsf{H}, \mathsf{S} \vdash \mathsf{L} : \mathsf{ok}$$

- *The continuations in each thread's call stack contain only references to allocated objects and do not contain any free variables.*

$$\forall (_, _, \mathsf{F}) \in \mathsf{T} \bullet \forall \overline{s} \in \mathsf{F} \bullet$$
$$(\mathsf{objectRefs}(\overline{s}) \subseteq dom(\mathsf{H}) \;\wedge\; \mathsf{free}(\overline{s}) = \emptyset)$$

Notice that well-formedness of a program state implies that access sets are disjoint and that accessible objects are allocated.

Theorem 5.1. *In a legal program π, the small step relation $\rightarrow$ preserves well-formedness.*

$$\mathsf{legal}(\pi) \Rightarrow (\forall \sigma_1, \sigma_2 \bullet (\mathsf{wf}(\sigma_1) \ \wedge \ \sigma_1 \rightarrow \sigma_2) \Rightarrow \mathsf{wf}(\sigma_2))$$

Proof. By case analysis on the step from σ_1 to σ_2. We consider cases [Share], [Synchronized], and [Unlock].

- **Case** [Share]. By legality, we have that the object being shared is in the access set but not in the shared set. The step adds it to the shared set (and therefore the free set since unshared objects are not locked) and removes it from the access set. It follows that the partition is maintained.
- **Case** [Synchronized]. Assume that the object being locked is o. In σ_1, o is shared and not locked, and therefore it is part of σ_1's free set. Since in a well-formed state the free set and the access sets partition the heap, o is not in any thread's access set. Adding o to a single access set and removing it from the free set (by adding an entry for o to the lock map) maintains the proper partitioning of the heap. Because locking an object modifies neither the heap nor the shared set and both are well-formed in the pre-state, they are well-formed in the post-state. Adding an entry for o (a shared object) to σ_1's well-formed lock set, preserves the fact that the lock set only contains shared objects. Finally, the continuations in σ_2 contain neither free variables nor unallocated objects, because σ_1 does not contain any and because locking did not introduce any.
- **Case** [Unlock]. By legality, the object being unlocked is in the access set and is locked by the current thread. The step removes it from the lock map (thus adding it to the free set, since locked objects are shared) and from the thread's access set. It follows that the partition is maintained.

□

Theorem 5.2. *States reached by legal programs are well-formed.*

Proof. By induction on the number of execution steps: the initial state is well-formed, and the induction step holds, as per Theorem 5.1. □

The notion of *data race* is defined in JLS3 in terms of the *happens-before* relation.

Definition 5.12 (Happens-Before)**.** *Consider an execution of a legal program. The* happens-before relation *on the execution is a partial order among the steps of the execution. Specifically, it is the smallest transitive relation such that*

- *Each step performed by a thread t happens-before each subsequent step performed by t.*
- *Each* [Unlock] *step on an object o happens-before each subsequent* [Synchronized] *step on o.*
- *Each* [NewThread] *step that creates a thread t happens-before each operation performed by t.*

Definition 5.13 (Data Race)**.** *Consider an execution of a legal program. A pair of steps of the execution constitutes a* data race *if one is a* [Write] *of a field $o.f$ and the other is a* [Read] *or* [Write] *of $o.f$ and the steps are not ordered by the happens-before relation.*

The following lemma states that in legal programs, no ordering constraints exist on execution steps beyond those imposed by the synchronization constructs (i.e., thread creation and **synchronized** blocks).

Lemma 5.1. *In an execution of a legal program π, if two consecutive steps are not ordered by happens-before, then swapping them results again in an execution of π.*

Proof. By case analysis on the steps. We detail a few cases.

- The steps are not accesses of the same field, since this would mean access sets are not disjoint, and by Theorem 5.2 we have that program states are well-formed.
- A [New] step can be moved to the right. The other step does not access the newly created object since the [New] step does not modify the thread state of the other thread and well-formedness of the latter implies it does not mention unallocated objects.

□

We are now ready to prove the main theorem of this section.

Theorem 5.3. *Executions of legal programs do not contain data races.*

Proof. By contradiction. Consider a legal program π such that at least one of its executions contains a data race on a field $o.f$. Of all the data races in all the executions of π, pick one where the number of steps that intervene between the steps that constitute the data race is minimal. Let those steps be S_i and S_{i+1+n}.

Assume first that $n = 0$. In state σ_{i-1} (the pre-state of step S_i) o is in the access set of the thread t_i that performs S_i. Since a field access does not modify any access sets, this is still true in state σ_i. However, in this state, it is also true

that o is in the access set of thread t_{i+1} that performs S_{i+1}. This contradicts Theorem 5.2.

Assume now that $n > 0$. Consider the last step preceding S_{i+1+n} that does not happen before S_{i+1+n}. Such a step exists, since S_i does not happen before S_{i+1+n}. By repeated application of Lemma 5.1, this step can be moved after S_{i+1+n}, contradicting the minimality assumption. □

5.1.4 Non-interference

In this subsection, we introduce the notion of a *non-interfering state change* and we prove that in executions of legal programs, with respect to the access set of one thread, steps of other threads are non-interfering state changes.

Definition 5.14. *Two states are related by a* non-interfering state change *with respect to a given access set if*

- *all objects that are allocated in the first state are also allocated in the second state,*
- *all objects that are shared in the first state are also shared in the second state, and*
- *if an object is in the access set, then its state in the heap is unchanged and if additionally the object is unshared in the first state, then it is unshared in the second state.*

Formally:

$$(\mathsf{H}, \mathsf{S}) \overset{\mathsf{A}}{\rightsquigarrow} (\mathsf{H}', \mathsf{S}')$$
$$\Updownarrow$$
$$\mathrm{dom}(\mathsf{H}) \subseteq \mathrm{dom}(\mathsf{H}') \wedge \mathsf{S} \subseteq \mathsf{S}' \wedge \mathsf{H}'|_{\mathsf{A}} = \mathsf{H}|_{\mathsf{A}} \wedge \mathsf{S}' \cap \mathsf{A} = \mathsf{S} \cap \mathsf{A}$$

The following theorem states a key property of the programming model. Note: we write $\sigma \overset{\mathsf{tid}}{\rightarrow} \sigma'$ to denote that program states σ and σ' are related by an execution step performed by thread tid.

Theorem 5.4 (Thread Isolation)**.** *In an execution of a legal program, with respect to the access set of one thread, a sequence of steps of other threads constitutes a non-interfering state change.*

$$\mathsf{legal}(\pi) \wedge \mathsf{initial}_\pi(\sigma^0) \wedge \sigma^0 \rightarrow^* \sigma_0 \overset{\mathsf{tid}_1}{\rightarrow} \sigma_1 \cdots \overset{\mathsf{tid}_n}{\rightarrow} \sigma_n$$
$$\wedge$$
$$(\forall i \in \{0, \ldots, n\} \bullet \sigma_i = (\mathsf{H}_i, \mathsf{L}_i, \mathsf{S}_i, \mathsf{T}_i \lhd (\mathsf{tid}, \mathsf{A}_i, \mathsf{F}_i))) \wedge \mathsf{tid} \notin \{\mathsf{tid}_1, \ldots, \mathsf{tid}_n\}$$
$$\Downarrow$$
$$(\mathsf{H}_0, \mathsf{S}_0) \overset{\mathsf{A}_0}{\rightsquigarrow} (\mathsf{H}_n, \mathsf{S}_n) \wedge \mathsf{A}_n = \mathsf{A}_0$$

Proof. Since the program is legal, by Theorem 5.2 we have that all program states reached are legal and well-formed. We prove the theorem by induction on n. The base case $n = 0$ is trivial. Assume $n > 0$. By induction we have

$$(\mathsf{H}_0, \mathsf{S}_0) \overset{\mathsf{A}_0}{\leadsto} (\mathsf{H}_{n-1}, \mathsf{S}_{n-1}) \wedge \mathsf{A}_{n-1} = \mathsf{A}_0$$

We perform case analysis on the rule used to derive the last step. We consider case [WRITE].

Suppose the step's pre-state is well-formed and that the transition corresponds to the application of the [WRITE] rule, i.e. assigning a new value v to the field f of an object o. No new objects are allocated when assigning to fields, and therefore the domain of the old and the new heap are equal. The old and the new heap differ only at a single location, namely (o, f). From legality it follows that o is an element of thread tid_n's access set, and by well-formedness it is not an element of A_0. Finally, updating a field does not modify the shared set. □

5.2 Static verification

The previous section proved that legal programs are data-race-free. However, legality of a program cannot be checked automatically. In this section, we define the notion of *valid* programs, which is a condition suitable for submission to an automatic theorem prover, and we show that valid programs are legal. It follows that valid programs are data-race-free and that they perform no null dereferences or ill-typed operations.

The validity notion is based on provability of *verification conditions* in the *verification logic*, i.e., the logic used to interpret the verification conditions.

Before we define and prove the validity notion, we discuss the modular verification approach and we establish the verification logic.

5.2.1 Modular verification

To determine validity of a method, one could take the entire program into account at once. However, this approach is not modular, and hence does not scale to large programs. Instead, we propose a modular approach where the validity of a method does not depend on the entire program, but only on its body, its method contract, and the contracts of its callees.

To verify whether a method complies with the model and respects its method contract and those of its callees, one may only assume that the method's precondition holds and that the program state is well-formed when the method is called. Furthermore, note that methods cannot query the access set in the sense that branching on whether an object is accessible or not is impossible. From this we can derive a framing property: a method call does not modify or share an

object o that is allocated and accessible before the call, if the precondition does not require o to be accessible. Thanks to this framing property, we do not need explicit modifies clauses in method contracts.

We impose the additional restriction that fields can only hold shared objects. By doing so, we must verify that any field write stores a shared object, but in return we may assume that the objects read from fields are shared. This restriction simplifies the development, but it can easily be relieved, for example by introducing a **shared** modifier, which indicates that a certain field can only hold shared objects (see Section 4.2) or by introducing invariants (see Section 4.4).

5.2.2 Verification logic

We target multi-sorted first-order predicate logic with equality. That is, a *term* t is a *logical variable* $y \in Y$ or a *function application* $f(t_1, \ldots, t_n)$ where f is a *function symbol* from the *signature* with *arity* n. A *formula* ϕ is an *equality* $t_1 = t_2$, an *inequality* $t_1 \neq t_2$, a *literal* **true** or **false**, an *atom* $P(t_1, \ldots, t_n)$ where P is a predicate symbol from the signature with arity n, a *propositional formula* using the connectives $\wedge$, $\vee$, $\Rightarrow$, and $\neg$, or a *quantification* $(\forall y \bullet \phi)$.

The sorts are the following:

- The sort of object references
- The sort of program values (the object references and the null value)
- The sort of finite sets of object references
- The sort of field names
- The sort of class names
- The sort of interface names
- The sort of *object states*, i.e., finite partial functions from field names to program values
- The sort of *heaps*, i.e., finite partial functions from object references to object states

Note:

- All sorts are countably infinite.
- We leave the sorts of quantifications, function symbols, and predicate symbols implicit when they are clear from the context.

We use the following signature, for a given program:

Predicate symbol	Syntax	Notes
$\mathsf{in}(t_1, t_2)$	$t_1 \in t_2$	
$\mathsf{prec}(t_1, t_2)$	$t_1 \prec_\pi t_2$	subtype relation

Function symbol	Syntax	Notes
$\mathsf{emptyset}$	$\emptyset$	
$\mathsf{insert}(t_1, t_2)$	$t_1 \cup \{t_2\}$	
$\mathsf{remove}(t_1, t_2)$	$t_1 \setminus \{t_2\}$	
$\mathsf{intersect}(t_1, t_2)$	$t_1 \cap t_2$	
$\mathsf{setminus}(t_1, t_2)$	$t_1 - t_2$ or $t_1 \setminus t_2$	
$\mathsf{apply}(t_1, t_2)$	$t_1(t_2)$	
$\mathsf{update}(t_1, t_2, t_3)$	$t_1[t_2 \mapsto t_3]$	function update
$\mathsf{dom}(t)$	$\mathrm{dom}(t)$	function domain
null	null	
$\mathsf{c}(t)$	$\mathsf{c}(t)$	class of an object
$\mathsf{asvalue}(t)$	t	implicit widening conversion
$\mathsf{asobjref}(t)$	t	implicit narrowing conversion

The widening and narrowing conversions are inserted implicitly to convert between program values and object references. Also, some of the function symbols are overloaded. Specifically, implicitly there is a separate $\mathsf{emptyset}$ symbol for each set or function sort, and there are separate apply, update, and dom symbols for each function sort.

Additionally, the signature contains a nullary function symbol τ for each class or interface τ declared by the program, and a nullary function symbol f for each field f declared by the program.

Note: we also use the following abbreviations:

Abbreviation	Meaning
$t_1(t_2, t_3)$	$t_1(t_2)(t_3)$
$t_1[(t_2, t_3) \mapsto t_4]$	$t_1[t_2 \mapsto (t_1(t_2)[t_3 \mapsto t_4])]$
$t_1 \subseteq t_2$	$(\forall y \bullet y \in t_1 \Rightarrow y \in t_2)$
$t_2\vert_{t_1} = t_3\vert_{t_1}$	$(\forall y \bullet y \in t_1 \Rightarrow t_2(y) = t_3(y))$

where y is a fresh logical variable.

Throughout, we interpret the logic according to an interpretation $\mathfrak{I}$ of the signature, which is as expected. Corner cases are as follows. $\mathfrak{I}(\mathsf{asobjref})(\mathfrak{I}(\mathsf{null}))$ yields some (fixed) object reference. If e_2 is not in the domain of e_1, then $\mathfrak{I}(\mathsf{apply})(e_1, e_2)$ yields some (fixed) element of the range sort. $\mathfrak{I}(\mathsf{prec})(C, I)$ holds if the program declares a class C that mentions an interface I in its **implements** clause.

Let Σ be an (incomplete) axiomatization of $\mathfrak{I}$. It follows that if a formula ϕ is provable from Σ, then ϕ is true under $\mathfrak{I}$:

$$\text{if } \Sigma \vdash \phi \text{ then } \mathfrak{I} \vDash \phi$$

Note:

- A multi-sorted first-order logic where all sorts are countably infinite can be reduced to a one-sorted first-order logic by having a universe of natural numbers and mapping it to the various sorts as appropriate in the interpretation. This allows us to use theorem provers for one-sorted logic, such as Simplify.
- We will sometimes use *partially interpreted formulae*, i.e., formulae where elements of the universe appear as terms. Also, we will sometimes use a verification logic formula directly as a meta-logic formula, assuming the interpretation $\mathfrak{I}$ and interpreting the free logical variables as the corresponding meta-logical variables.

5.2.3 Valid programs

In this subsection, we define which *verification conditions* are generated for a given program. A verification condition is a closed formula in the verification logic. A program is *valid* if its verification conditions are provable from the verification logic's theory Σ.

We define program verification conditions in terms of *continuation verification conditions*. A continuation verification condition $\mathsf{vc}(\overline{s}, Q)$ is a formula of the verification logic that expresses conditions under which a continuation $\overline{s}$ (i.e., a list of statements to be executed next) executes correctly and satisfies postcondition Q when started in a given program state. A continuation verification condition depends only on the state variables relevant to the thread that executes the continuation. These state variables appear as free logical variables in the formula.

Definition 5.15. *In a given program state σ, the* thread-relevant state $(\mathsf{H}, \mathsf{L_t}, \mathsf{S}, \mathsf{A})$ *of a certain thread consists of the heap, the objects locked by the thread, the shared set and the thread's access set.*

Definition 5.16. *A thread-relevant state $(\mathsf{H}, \mathsf{L_t}, \mathsf{S}, \mathsf{A})$ is* well-formed *($\vdash (\mathsf{H}, \mathsf{L_t}, \mathsf{S}, \mathsf{A}) : \mathsf{ok}$) if the heap is well-formed, the shared set is well-formed with respect to the heap, the access set is a subset of the heap's domain, the lock set is a subset of the shared set, and non-null field values are shared.*

$$\begin{array}{c} \vdash (\mathsf{H}, \mathsf{L_t}, \mathsf{S}, \mathsf{A}) : \mathsf{ok} \\ \Updownarrow \\ \vdash \mathsf{H} : \mathsf{ok} \ \wedge \ \mathsf{H} \vdash \mathsf{S} : \mathsf{ok} \ \wedge \ \mathsf{L_t} \subseteq \mathsf{S} \wedge \ \mathsf{A} \subseteq \mathit{dom}(\mathsf{H}) \\ \wedge \\ (\forall o \in \mathrm{dom}(\mathsf{H}), f \in \mathrm{dom}(\mathsf{H}(o)) \bullet \mathsf{H}(o, f) = \mathsf{null} \vee \mathsf{H}(o, f) \in \mathsf{S}) \end{array}$$

A continuation, as it appears in a program text, may contain free program variables. Consequently, the corresponding continuation verification condition contains these program variables as free logical variables.

Definition 5.17. *A* program variable $z \in \mathit{Var}$ *is either* **this***,* **result***, or a method parameter or local variable* $x \in \mathcal{X}$*.*

$$\mathit{Var} = \{\mathbf{this}, \mathbf{result}\} \cup \mathcal{X}$$

A continuation verification condition is a *state-based predicate*.

Definition 5.18. *A* state-based predicate Q *is a formula of the verification logic, whose free logical variables are thread-relevant state variables and program variables:*

$$\mathsf{free}(Q) \subseteq \{\mathsf{H}, \mathsf{L_t}, \mathsf{S}, \mathsf{A}\} \cup \mathit{Var}$$

A state-based predicate is program-closed *if no program variables appear free in it.*

We will use the notation $\mathfrak{I}, \mathsf{H}, \mathsf{L_t}, \mathsf{S}, \mathsf{A}, V \vDash Q$ to denote the truth of a state-based predicate in a particular thread-relevant state and under a particular valuation V that maps program variables to program values.

Where appropriate, we implicitly interpret method preconditions and postconditions as state-based predicates as follows:

$$\begin{array}{rcl} g.\mathbf{canAccess} & = & g \in \mathsf{A} \\ g.\mathbf{canLock} & = & g \in \mathsf{S} \ \wedge \ g \notin \mathsf{L_t} \\ g.\mathbf{canShare} & = & g \in \mathsf{A} \ \wedge \ g \notin \mathsf{S} \\ \mathbf{threadholdsnolocks} & = & \mathsf{L_t} = \emptyset \end{array}$$

The method framing approach used as part of the modular verification approach is based on the notion of *required access sets*. Semantically, under a given argument list, a method precondition P's required access set $\mathcal{A}(P)$ is the set of objects that are required by P to be in the access set. Syntactically, $\mathcal{A}(P)$ is defined as:

$$\begin{array}{rcl} \mathcal{A}(t.\mathbf{canAccess}) & \equiv & \{t\} \\ \mathcal{A}(t.\mathbf{canLock}) & \equiv & \emptyset \\ \mathcal{A}(t.\mathbf{canShare}) & \equiv & \emptyset \\ \mathcal{A}(\mathbf{threadholdsnolocks}) & \equiv & \emptyset \\ \mathcal{A}(\mathit{pred}_1 \ldots \mathit{pred}_n) & \equiv & \mathcal{A}(\mathit{pred}_1) \cup \cdots \cup \mathcal{A}(\mathit{pred}_n) \end{array}$$

Figure 5.5 defines the predicate transformer vc, which maps a continuation $\overline{s}$ and a state-based predicate Q to the continuation verification condition for $\overline{s}$ and Q. As noted above, $\mathsf{vc}(\overline{s}, Q)$ is a sufficient condition such that any thread in a state satisfying $\mathsf{vc}(\overline{s}, Q)$ ends up in a state satisfying Q after executing the continuation $\overline{s}$.

The intuition underlying the verification condition rules is as follows. The verification condition for [VC-If] is a standard weakest precondition. Since an

assert statement blocks forever if its condition is false, the statement's continuation is verified under the assumption that the condition is true. For a field access ([VC-READ] or [VC-WRITE]), the target of the access must be non-null and accessible. A newly created object ([VC-NEW]) was previously unallocated and is of the correct class. In order to share ([VC-SHARE]) a value, the value should be non-null, accessible and unshared. Locking ([VC-SYNCHRONIZED]) is disallowed if the object is unshared or already locked by the current thread. Moreover, since between synchronized statements other threads may modify shared objects, we should assume nothing about the fields of the newly locked object. The only property we may assume is that the old state and the new state are related by a non-interfering state change with respect to the thread's access set. Unlocking ([VC-UNLOCK]) is allowed only for locked and accessible objects. When invoking a method ([VC-CALL]), the target should not be null, the callee's precondition should hold and when returning ([VC-RECEIVE]) Q should hold. When a method call returns, we make some assumptions about the post-state. First of all, as per the method framing approach, we may assume that the post-state is related to the pre-state by a non-interfering state change with respect to the pre-state access set minus the callee's required access set. Secondly, we may assume the post-state is well-formed and satisfies the callee's postcondition. Finally, we may assume the return value is allocated. Starting a new thread via **start** ([VC-NEWTHREAD]) requires the target object to be accessible and non-null. Accessibility of the target object is transferred from the current thread to the new thread.

An important property of continuation verification conditions is that they are *local*.

Definition 5.19. *A state-based predicate Q is* local *if it is preserved by a non-interfering state change.*

$$\mathsf{local}(Q)$$
$$\Updownarrow$$
$$\forall \mathsf{H}, \mathsf{L_t}, \mathsf{S}, \mathsf{A}, \mathsf{H}', \mathsf{S}' \bullet \mathfrak{I}, \mathsf{H}, \mathsf{L_t}, \mathsf{S}, \mathsf{A}, V \vDash Q \land (\mathsf{H}, \mathsf{S}) \overset{\mathsf{A}}{\rightsquigarrow} (\mathsf{H}', \mathsf{S}') \Rightarrow \mathfrak{I}, \mathsf{H}', \mathsf{L_t}, \mathsf{S}', \mathsf{A}, V \vDash Q$$

Theorem 5.5. *If a state-based predicate Q is local, then the verification condition of a continuation $\overline{s}$ with respect to Q is local as well.*

$$\mathsf{local}(Q) \Rightarrow \mathsf{local}(\overline{s}, Q)$$

Proof. By induction on $\overline{s}$.

- **Case** $\overline{s} =$ **synchronized** (v) $\{\ \overline{s}''\ \}\ \overline{s}'$.
  1. We may assume that $\overline{s}$ is valid (i.e., $\mathsf{vc}(\overline{s}, Q)$ holds) in a state $(\mathsf{H}, \mathsf{L_t}, \mathsf{S}, \mathsf{A})$.
  2. It follows by [VC-SYNCHRONIZED] that the continuation $\overline{s}''$ **unlock** v; $\overline{s}'$ of $\overline{s}$ is valid in any state $(\mathsf{H}', \mathsf{L_t} \cup \{v\}, \mathsf{S}', \mathsf{A} \cup \{v\})$ where $(\mathsf{H}, \mathsf{S}) \overset{\mathsf{A}}{\rightsquigarrow} (\mathsf{H}', \mathsf{S}')$.

$\mathsf{vc}(\textbf{if } (v_1 = v_2) \; \{ \; \overline{s}_1 \; \} \textbf{ else } \{ \; \overline{s}_2 \; \} \; \overline{s}, Q) \equiv$ [VC-IF]
$\quad (v_1 = v_2 \Rightarrow \mathsf{vc}(\overline{s}_1 \; \overline{s}, Q)) \; \wedge \; (v_1 \neq v_2 \Rightarrow \mathsf{vc}(\overline{s}_2 \; \overline{s}, Q))$

$\mathsf{vc}(\textbf{assert } v \textbf{ instanceof } \tau; \; \overline{s}, Q) \equiv$ [VC-ASSERT]
$\quad (v \neq \mathsf{null} \wedge c(v) \preceq_\pi \tau) \Rightarrow \mathsf{vc}(\overline{s}, Q)$

$\mathsf{vc}(x := v.f_C; \; \overline{s}, Q) \equiv$ [VC-READ]
$\quad v \neq \mathsf{null} \wedge c(v) = C \wedge \; v \in \mathsf{A} \; \wedge \; \mathsf{vc}(\overline{s}, Q)[\mathsf{H}(v, f)/x]$

$\mathsf{vc}(v_1.f_C := v_2; \; \overline{s}, Q) \equiv$ [VC-WRITE]
$\quad v_1 \neq \mathsf{null} \wedge c(v_1) = C \wedge v_1 \in \mathsf{A} \wedge (v_2 = \mathsf{null} \vee v_2 \in \mathsf{S}) \wedge \mathsf{vc}(\overline{s}, Q)[\mathsf{H}[(v_1, f) \mapsto v_2]/\mathsf{H}]$

$\mathsf{vc}(x := \textbf{new } C; \; \overline{s}, Q) \equiv$ [VC-NEW]
$\quad \forall o \bullet o \notin dom(\mathsf{H}) \; \wedge c(o) = C \Rightarrow$
$\quad\quad \mathsf{vc}(\overline{s}, Q)[o/x, (\mathsf{H}[o \mapsto (f_1 \mapsto \mathsf{null}, ..., f_n \mapsto \mathsf{null})]/\mathsf{H}, (\mathsf{A} \cup \{o\})/\mathsf{A}]$
$\quad$ where $\textbf{fields}_\pi(C) = \{f_1, ..., f_n\}$

$\mathsf{vc}(\textbf{share } v; \; \overline{s}, Q) \equiv$ [VC-SHARE]
$\quad v \neq \mathsf{null} \; \wedge \; v \in \mathsf{A} \; \wedge \; v \notin \mathsf{S} \wedge \; \mathsf{vc}(\overline{s}, Q)[(\mathsf{A} \setminus \{v\})/\mathsf{A}, (\mathsf{S} \cup \{v\})/\mathsf{S}]$

$\mathsf{vc}(\textbf{synchronized}(v) \; \{ \; \overline{s}' \; \} \; \overline{s}, Q) \equiv$ [VC-SYNCHRONIZED]
$\quad v \neq \mathsf{null} \; \wedge \; v \in \mathsf{S} \; \wedge \; v \notin \mathsf{L_t}$
$\quad \wedge \; (\forall \mathsf{H}', \mathsf{S}' \bullet$
$\quad\quad ((\mathsf{H}, \mathsf{S}) \overset{\mathsf{A}}{\rightsquigarrow} (\mathsf{H}', \mathsf{S}') \; \wedge \; \vdash (\mathsf{H}', \mathsf{L_t}, \mathsf{S}', \mathsf{A}) : \mathsf{ok})$
$\quad\quad \Rightarrow$
$\quad\quad \mathsf{vc}(\overline{s}' \textbf{ unlock } v; \; \overline{s}, Q)[\mathsf{H}'/\mathsf{H}, (\mathsf{A} \cup \{v\})/\mathsf{A}, (\mathsf{L_t} \cup \{v\})/\mathsf{L_t}, \mathsf{S}'/\mathsf{S}])$

$\mathsf{vc}(\textbf{unlock } o; \; \overline{s}, Q) \equiv$ [VC-UNLOCK]
$\quad o \in \mathsf{A} \; \wedge \; o \in \mathsf{L_t} \; \wedge \; \mathsf{vc}(\overline{s}, Q)[(\mathsf{A} \setminus \{o\})/\mathsf{A}, (\mathsf{L_t} \setminus \{o\})/\mathsf{L_t}]$

$\mathsf{vc}(x := v.m_I(\overline{v}); \; \overline{s}, Q) \equiv$ [VC-CALL]
$\quad v \neq \mathsf{null} \; \wedge \; c(v) \prec_\pi I \wedge \mathsf{pre}(I, m)[v/\textbf{this}, \overline{v}/\overline{x}]$
$\quad \wedge \; \mathsf{vc}(x := \textbf{receive } (\mathsf{pre}(I, m)[v/\mathsf{this}, \overline{v}/\overline{x}], \mathsf{post}(I, m)[v/\mathsf{this}, \overline{v}/\overline{x}], A, S, L_\mathrm{t}); \; \overline{s}, Q)$

$\mathsf{vc}(\textbf{start } v.m_I(); \; \overline{s}, Q) \equiv$ [VC-NEWTHREAD]
$\quad v \neq \mathsf{null} \; \wedge c(v) \prec_\pi I \wedge \; v \in \mathsf{A} \; \wedge \; \mathsf{vc}(\overline{s}, Q)[(\mathsf{A} \setminus \{v\})/\mathsf{A}]$

$\mathsf{vc}(\textbf{return } v; , Q) \equiv$ [VC-RETURN]
$\quad Q[v/\textbf{result}]$

Figure 5.5: Continuation verification conditions *(Part 1 of 2)*

$$
\begin{array}{ll}
\mathsf{vc}(x := \textbf{receive}\ (P', Q', \mathsf{A}^{\text{old}}, \mathsf{S}^{\text{old}}, \mathsf{L}_{\mathsf{t}}^{\text{old}});\ \overline{s}, Q) \equiv & \text{[VC-RECEIVE]} \\
\quad \forall \mathsf{H}', \mathsf{S}', \mathsf{A}', v_{\mathrm{r}} \bullet & \\
\qquad ((\mathsf{H}, \mathsf{S}) \overset{\mathsf{A}^{\text{old}} - \mathcal{A}(P')}{\rightsquigarrow} (\mathsf{H}', \mathsf{S}') \wedge\ \vdash (\mathsf{H}', \mathsf{L}_{\mathsf{t}}, \mathsf{S}', \mathsf{A}') : \mathsf{ok} & \\
\qquad \wedge\ \mathsf{A}' \cap (\mathsf{A}^{\text{old}} - \mathcal{A}(P')) = \mathsf{A} \cap (\mathsf{A}^{\text{old}} - \mathcal{A}(P')) & \\
\qquad \wedge\ Q'[\mathsf{H}'/\mathsf{H}, \mathsf{S}'/\mathsf{S}, \mathsf{A}'/\mathsf{A}, \mathsf{L}_{\mathsf{t}}^{old}/\mathsf{L}_{\mathsf{t}}, v_r/\textbf{result}] & \\
\qquad \wedge\ (v_{\mathrm{r}} = \mathsf{null} \vee v_{\mathrm{r}} \in dom(\mathsf{H}')) & \\
\qquad \Rightarrow & \\
\qquad \mathsf{vc}(\overline{s}, Q)[\mathsf{H}'/\mathsf{H}, \mathsf{S}'/\mathsf{S}, \mathsf{A}'/\mathsf{A}, \mathsf{L}_{\mathsf{t}}^{old}/\mathsf{L}_{\mathsf{t}}, v_r/x] &
\end{array}
$$

Figure 5.5: Continuation verification conditions *(Part 2 of 2)*

3. We need to prove that $\overline{s}$ is valid in any state $(\mathsf{H}'', \mathsf{L}_{\mathsf{t}}, \mathsf{S}'', \mathsf{A})$ where $(\mathsf{H}, \mathsf{S}) \overset{\mathsf{A}}{\rightsquigarrow} (\mathsf{H}'', \mathsf{S}'')$.
4. This requires that we prove that the continuation $\overline{s}''$ **unlock** v; $\overline{s}'$ of $\overline{s}$ is valid in any state $(\mathsf{H}''', \mathsf{L}_{\mathsf{t}} \cup \{v\}, \mathsf{S}''', \mathsf{A} \cup \{v\})$ where $(\mathsf{H}'', \mathsf{S}'') \overset{\mathsf{A}}{\rightsquigarrow} (\mathsf{H}''', \mathsf{S}''')$.
5. It is easy to see that the non-interfering state change relation is transitive; therefore, from the assumptions in points 3 and 4 we have $(\mathsf{H}, \mathsf{S}) \overset{\mathsf{A}}{\rightsquigarrow} (\mathsf{H}''', \mathsf{S}''')$. By instantiating the rule in point 2, we obtain the goal in point 4.

- **Case receive** is analogous to case **synchronized**.
- The other cases are easy.

□

Notice that method postconditions are local.
We are now ready to define *program validity*.

Definition 5.20. *A well-formed program* π *is* valid *if all of following hold:*

- *each method* $m_I(\overline{x})$ $\{\ \overline{s}\ \}$ *in each class* C *is valid, i.e., it is provable from the verification logic that the method's precondition implies validity of the method's body.*

$$
\begin{array}{c}
\Sigma \vdash \\
\forall o, \overline{v}, \mathsf{H}, \mathsf{L}_{\mathsf{t}}, \mathsf{S}, \mathsf{A} \bullet \\
(\vdash (\mathsf{H}, \mathsf{L}_{\mathsf{t}}, \mathsf{S}, \mathsf{A}) : \mathsf{ok} \wedge \mathsf{c}(o) = C \wedge \mathsf{pre}(I, m)[o/\textbf{this}, \overline{v}/\overline{x}]) \\
\Downarrow \\
\mathsf{vc}(\overline{s}, \mathsf{post}(I, m))[o/\textbf{this}, \overline{v}/\overline{x}]
\end{array}
$$

- *the main routine* $\overline{s}$ *is valid:*

$$\Sigma \vdash \mathsf{vc}(\overline{s}, \mathbf{true})[\emptyset/\mathsf{H}, \emptyset/\mathsf{L_t}, \emptyset/\mathsf{S}, \emptyset/\mathsf{A}]$$

The example of Figure 5.2 is a valid program.

5.2.4 Soundness

In this subsection we define a notion of *valid program state* and we prove that valid states are legal states. We then prove that program states reached by valid programs are valid, by proving that the initial state is valid and that execution steps preserve validity. It follows that valid programs are legal programs and therefore they are data-race-free.

A program state is *valid* if each thread state is *consistent* and each activation record is *valid*. The latter means that the continuation verification condition of the activation record's continuation holds with respect to the activation record's postcondition. An activation record's validity is independent of actions performed by other activation records. We prove this using the notion of *activation record access sets*. We prove that an activation record's validity depends only on the state of the objects in its access set, and actions of other activation records are non-interfering with respect to this access set.

Definition 5.21. *An* activation record's access set *is recursively defined as follows:*

- *The top (i.e., active) activation record's access set is the thread's access set minus the other activation records' access sets.*
- *The access set of an activation record that is suspended while waiting for a call to return, is the pre-state thread access set, minus the access sets of transitive caller activation records, minus the call's required access set.*

Formally:

$$\begin{array}{c} t = (\mathsf{tid}, \mathsf{A}, \overline{s}_1 \cdot x_2 := \mathbf{receive}\ (P_2, _, A_2^{\mathrm{old}}, _, _);\ \overline{s}_2 \\ \cdot \ldots \cdot x_n := \mathbf{receive}\ (P_n, _, A_n^{\mathrm{old}}, _, _);\ \overline{s}_n) \\ \Downarrow \end{array}$$

$$\forall i \in \{1, \ldots, n\} \bullet \mathsf{A}_{\mathsf{ar}(i)}(t) = \begin{cases} \mathsf{A} - \bigcup_{1<j\leq n} \mathsf{A}_{\mathsf{ar}(j)}(t) & \text{if } i = 1 \\ \mathsf{A}_i^{\mathrm{old}} - \bigcup_{i<j\leq n} \mathsf{A}_{\mathsf{ar}(j)}(t) - \mathcal{A}(P_i) & \text{if } i > 1 \end{cases}$$

Definition 5.22. *A thread state is* consistent *with respect to a given lockset if*

- *The thread's lockset is equal to the set of objects that appear in* **unlock** *statements in the thread's activation records' continuations*

- *No object appears more than once in an* **unlock** *statement*
- *A caller's pre-state lockset is equal to the set of* **unlock** *statements in the call's continuation plus transitive caller continuations*
- *Each non-top activation record's required access set is included in its pre-state activation record access set*
- *Each non-top activation record's pre-state access set includes the access sets of transitive caller activation records*
- *The thread's access set includes the non-top activation records' access sets*

Formally:

$$\begin{gathered}
\mathsf{L_t} \vdash \mathsf{consistent}(t) \\
\Updownarrow \\
t \textit{ is of the form} \\
(\mathsf{tid}, \mathsf{A}, \overline{s}_1 \cdot x_2 := \textbf{receive } (P_2, _, A_2^{\text{old}}, _, L_2^{\text{old}});\ \overline{s}_2 \\
\cdot \ldots \cdot x_n := \textbf{receive } (P_n, _, A_n^{\text{old}}, _, L_n^{\text{old}});\ \overline{s}_n) \\
\textit{where} \\
\mathsf{L_t} = \{o | \exists i \in \{1, \ldots, n\} \bullet \overline{s}_i \textit{ contains } \textbf{unlock } o;\} \\
\wedge \\
\neg(\overline{s}_1 \ldots \overline{s}_n = \ldots\ \textbf{unlock } o;\ \ldots\ \textbf{unlock } o;\ \ldots) \\
\wedge \\
(\forall i \in \{2, \ldots, n\} \bullet L_i^{\text{old}} = \{o | \exists j \in \{i, \ldots, n\} \bullet \overline{s}_i = \ldots\ \textbf{unlock } o;\ \ldots\}) \\
\wedge \\
(n = 1 \\
\vee \\
(\mathcal{A}(P_n) \subseteq A_n^{\text{old}} \\
\wedge \\
(\forall i \in \{2, \ldots, n-1\} \bullet A_{i+1}^{\text{old}} - \mathcal{A}(P_{i+1}) \subseteq A_i^{\text{old}} \wedge \mathcal{A}(P_i) \subseteq A_i^{\text{old}} - (A_{i+1}^{\text{old}} - \mathcal{A}(P_{i+1}))) \\
\wedge \\
A_2^{\text{old}} - \mathcal{A}(P_2) \subseteq \mathsf{A}))
\end{gathered}$$

Definition 5.23. *An* activation record's postcondition $Q_{\mathsf{ar}(i)}(t)$ *is the postcondition stored in the* **receive** *statement in the caller's continuation, or* **true** *if the activation record has no caller.*

Formally:

$$\begin{gathered}
t = (\mathsf{tid}, \mathsf{A}, \overline{s}_1 \cdot x_2 := \textbf{receive } (_, Q_2, _, _, _);\ \overline{s}_2 \\
\cdot \ldots \cdot x_n := \textbf{receive } (_, Q_n, _, _, _);\ \overline{s}_n) \\
\Downarrow \\
\forall i \in \{1, \ldots, n\} \bullet Q_{\mathsf{ar}(i)}(t) = \begin{cases} Q_{i+1} & \textit{if } i < n \\ \textbf{true} & \textit{if } i = n \end{cases}
\end{gathered}$$

Definition 5.24. *An activation record is* valid *if the verification condition of its continuation with respect to its postcondition holds under the current heap, lock set, and shared set, and under the activation record's access set. Formally:*

$$
\begin{array}{c}
t = (\mathsf{tid}, \mathsf{A}, \overline{s}_1 \cdot \ldots \cdot \overline{s}_n) \\
\Downarrow \\
(\forall i \in \{1, \ldots, n\} \bullet \\
\mathsf{H}, \mathsf{L}, \mathsf{S} \vdash \mathsf{valid}_{\mathsf{ar}(i)}(t) \\
\Updownarrow \\
\mathfrak{I}, \mathsf{H}, \mathsf{L}^{-1}(\mathsf{tid}), \mathsf{S}, \mathsf{A}_{\mathsf{ar}(i)}(t) \vDash \mathsf{vc}(\overline{s}_i, Q_{\mathsf{ar}(i)}(t)))
\end{array}
$$

Definition 5.25. *A program state is* valid *if it is well-formed, each thread state is well-formed and consistent, and each activation record is valid.*

$$
\begin{array}{c}
\mathsf{valid}(\mathsf{H}, \mathsf{L}, \mathsf{S}, \mathsf{T}) \\
\Updownarrow \\
\mathsf{wf}(\mathsf{H}, \mathsf{L}, \mathsf{S}, \mathsf{T}) \\
\wedge \\
(\forall t \in \mathsf{T} \bullet t = (\mathsf{tid}, \mathsf{A}, r_1 \cdot \cdots \cdot r_n) \\
\Downarrow \\
\vdash (\mathsf{H}, \mathsf{L}^{-1}(\mathsf{tid}), \mathsf{S}, \mathsf{A}) : \mathsf{ok} \wedge \mathsf{L}^{-1}(\mathsf{tid}) \vdash \mathsf{consistent}(t) \\
\wedge \\
(\forall i \in \{1, \ldots, n\} \bullet \mathsf{H}, \mathsf{L}, \mathsf{S} \vdash \mathsf{valid}_{\mathsf{ar}(i)}(t)))
\end{array}
$$

Theorem 5.6. *A valid program state is a legal program state.*

Proof. One can easily prove that each thread is in a legal state by case analysis on the form of the continuation. □

Theorem 5.7. *In a valid program π, the small step relation $\rightarrow$ preserves validity.*

$$\forall \sigma_1, \sigma_2 \bullet (\mathsf{valid}(\sigma_1) \;\wedge\; \sigma_1 \rightarrow \sigma_2) \Rightarrow \mathsf{valid}(\sigma_2)$$

Proof. By case analysis on the step. Note that preservation of well-formedness is given by Theorem 5.1. We refer to the thread that performs the step as the current thread and the top activation record of the current thread as the current activation record. For each step rule, we have to prove that in σ', thread states are well-formed and consistent and activation records are valid.

Each step changes the current activation record's continuation. We only note other changes. Also, we only detail the argument for preservation of validity of the current activation record if it does not follow easily from the verification condition.

- **Case** [If], [Assert]. The step changes only the continuation of the current activation record. Therefore, thread state consistency and validity of the other activation records is preserved trivially.

- **Case** [WRITE]. The step changes the heap at some location $o.f$, and the current activation record's continuation. Thread state consistency depends on neither so it is preserved trivially. By [VC-WRITE], we have that o is in the current activation record's access set. Well-formedness of the thread-relevant state is preserved since the value written into the field is either **null** or a shared object. Since activation record access sets are disjoint and, as a result of the locality of continuation verification conditions (Theorem 5.5), activation record validity depends only on heap locations in the activation record's access set, validity of other activation records is preserved trivially.
- **Case** [SHARE]. The step changes the shared set and the current thread's access set. Since the object being shared was in the current activation record's access set, it is not in the access set of any other activation record of the current thread so the access set still contains those. This establishes thread state consistency. The validity of an activation record is preserved by sharing an object that is not in its access set, and by removing from the thread's access set an object that is not in the activation record's access set, so the validity of non-current activation records is preserved.
- **Case** [CALL]. The step changes the current activation record's continuation and adds a new activation record. Since the precondition holds, its required access set is contained in the caller's pre-state access set. Therefore, thread state consistency is preserved. Validity of existing non-current activation records is preserved trivially. Since the program is valid, the method being called is valid, and the method's body is valid in any state that satisfies the precondition. Note that the new activation record's access set is equal to the precondition's required access set.
- **Case** [RETURN]. The step replaces the caller and callee activation records with an activation record containing the continuation of the call. Validity of the caller activation record implies that the call's continuation is valid in any thread state that a) differs from the current state only as allowed by the method's frame condition and b) satisfies the postcondition. Validity of the callee implies that the postcondition holds in the current state. Therefore the call's continuation is valid in the current state.
- **Case** [NEWTHREAD]. The step adds a new thread with a single activation record, and removes the target object from the creating thread's access set. Thread state consistency of the new thread is trivial. The new activation record is valid since the method it executes is valid and its precondition, which by well-formedness of the program must be **threadholdsnolocks** $\wedge$ **this.canAccess**, is satisfied.
- **Case** [READ]. The step changes only the current activation record's continuation. Validity follows trivially from [VC-READ].

- **Case** [New]. The step adds an object to the heap domain and the thread's access set. Since by well-formedness access sets contain only allocated objects, access sets remain disjoint.
- **Case** [Synchronized]. The step adds an object to the lock set and the access set. Since by well-formedness the free set is disjoint from access sets and the object was in the free set, access sets remain disjoint.
- **Case** [Unlock]. The step removes an object from the thread's access set. Since it was in the current activation record's access sets, no other activation records are affected.

□

Theorem 5.8. *A valid program is a legal program.*

Proof. The initial state of a valid program is a valid state. It follows, by Theorem 5.7, that all reachable states are valid. Therefore, by Theorem 5.6, all reachable states are legal. Therefore, the program is legal. □

Theorem 5.9 (Main Theorem)**.** *Valid programs are data-race-free.*

Proof. By combining Theorem 5.8 and Theorem 5.3. □

Chapter 6

Conclusion

In this thesis we presented an approach for the formal verification of the absence of certain errors in concurrent object-oriented programs.

The contributions of the thesis are the following:

- An extension of the Boogie approach with improved support for modularity by allowing state abstraction using inspector methods. Inspector methods may depend on the fields of the receiver object as well as the state of owned objects. They may be abstract and may be overridden. They may have parameters and they may depend on the state of a parameter if specially marked.

- An extension of the Boogie approach with support for concurrent programs that use mutual exclusion locks. The approach prevents data races and guarantees the soundness of sequential reasoning about sequential code. The approach supports ownership transfer of aliased objects.

- Support for deadlock prevention. The partial order among objects is specified by assigning a *lock level* to an object when it is shared. Lock levels are values which may be created, manipulated and stored like other program values, except that statements that manipulate lock levels and variables that hold lock levels must be inside annotations.

- Support for immutable objects. To support immutable objects, a distinction is made between accessibility for reading and accessibility for writing. Any unshared object may be shared as lock-protected or as immutable.

- Support for static fields, static initializers, and static class invariants. The approach is sound with respect to Java and C#'s lazy class initialization semantics. By integrating the approach with the concurrency approach,

sequential code can be reasoned about sequentially and class initialization deadlocks are prevented.

- A prototype implementation of the approach as an extension of the Boogie program verifier. The implementation is available on-line in binary form.

- Initial experience with the implementation on non-trivial programs, including a chat server.

- A formalization, including a soundness proof, of the core of the approach. The annotation syntax and the dynamics of the programming model for data race prevention are formalized and proven sound. Further, the verification condition generation rules are formalized and proven sound.

This thesis contributes to the research on ways to make it easier for software developers to deliver correct concurrent programs, and in particular on tool support for verifying that a shared-memory concurrent imperative program is free from data races and deadlocks. The prototype tool implementation was used successfully to verify a number of small concurrent programs; however, further work is needed to arrive at a tool that is ready to be used beneficially in a full-scale software development project. The main issues are the following:

- The programming model needs to be extended to allow more programming patterns. To assess the applicability of the programming model, the author performed a cursory examination of the open-source concurrent Java program Azureus. It was found that this program does not comply with the programming model as-is. For example, some objects are protected by custom lock implementations rather than Java's built-in locks. Also, some static fields are initialized in the main method rather than the static initializer. Whereas it is clear that the programming model needs to be extended, the patterns noticed by the author seemed to require only fairly small extensions. Still, it is possible that other common patterns exist that would require more extensive modifications to the programming model, or, it could turn out that no tractable programming model can capture all patterns in common use.

- The annotation burden needs to be reduced. The approach requires that programmers annotate their program to instantiate the programming model and to enable modular verification. Annotating a program is currently too labor-intensive. Common patterns need to be identified and concise syntax needs to be provided for them. Also, tool support for inferring annotations is needed to make it possible to start using the tool on an existing codebase.

- The power and the speed of the theorem prover might need to be improved. Experience shows that for complex methods with complex annotations, the

theorem prover sometimes fails to prove a valid verification condition. In the cases encountered, it was always possible to fix this by adding additional annotations, which serve as lemmas for the theorem prover. Also, in some cases verification of a single method takes half an hour. This might make it impossible to verify a realistic codebase overnight. On the other hand, thanks to the modularity of the verification approach, it is sufficient to re-verify the methods that are impacted by recent changes. In a well-structured codebase with low coupling, this should reduce the verification burden significantly.

We based our approach on state-of-the-art technology for modular verification of sequential (i.e., non-concurrent) programs, based on an automatic theorem prover. This is one verification approach among many: interactive theorem proving, abstract interpretation, model checking, type checking, testing, and hybrid approaches. All of these have different trade-offs along the various axes: speed of verification, degree of assurance, annotation overhead, range of properties that can be verified. Automatic theorem proving scores well in the areas of degree of assurance and range of properties, which are most interesting from a research point of view. From a practical point of view, the other axes are at least as important. Important strides are being made in hybrid model checking-theorem proving-abstract interpretation technology, with techniques like abstraction refinement and invariants on demand, where the strengths of each approach are combined.

Next to the underlying verification technology, the other major component of a program verification approach is the programming model. It determines which design and programming patterns are supported by the approach. There has been great progress recently in programming models. Ownership systems support abstraction in the presence of object aliasing; visibility-based invariant systems support peer relationships. Support for state abstraction in the form of model fields and data groups, or dynamic frames, has appeared. Similar developments occur in separation logic. We included an ownership system in our approach, and we contributed a state abstraction approach based on inspector methods.

In the space of programming models for verification of concurrent imperative programs, ownership systems are being combined with rely-guarantee reasoning to achieve both modularity and reasoning power.

This thesis focuses on object-oriented programs; however, the core ideas are applicable to shared-memory imperative programs in general. For example, the approach presented for Java is probably fairly easy to port to a Java-like subset of C. Dereferencing a pointer would require the pointer to be in the thread's access set. Freeing a pointer would remove it from the access set. A heap-allocated struct instance could be shared as lock-protected provided it has a field holding a mutex.

Another (perhaps mainly historically) important shared-memory imperative programming language is Ada. The approach of the thesis can probably be applied to Ada fairly easily to prevent data races and deadlocks amongst threads (called *tasks* in Ada). A minor complication is that Ada's primary synchroniza-

tion construct is *rendezvous* rather than locking. A rendezvous occurs when one task calls an *entry* of another task and the other task performs an *accept* of the entry. The calling task is blocked until the accepting task finishes executing the *accept body*. Rendezvous can be verified easily by annotating each entry with a precondition and a postcondition to specify the transfer of object accessibility that occurs at the start and at the end of a rendezvous, respectively.

The concurrency paradigm in current practice in systems software, as supported by the most popular programming languages, is shared memory concurrency, using locks for synchronization. This minimizes the abstraction gap between the programming language and the hardware and therefore allows for maximum performance and flexibility. On the other hand, programming in this paradigm is difficult and error-prone. This thesis contributes to improving this situation, but more work is needed, both in the area of providing assistance for lock-based programming, and in the area of finding alternative or complementary concurrency paradigms. Some of the paradigms being considered are: software transactional memory, functional programming, and message-passing-based approaches such as the actors approach. Some of these rule out object sharing altogether; others still require a programming model such as the one proposed in this thesis. An important question is whether giving up shared memory would be a more cost-effective solution to the data races problem than attempting to verify data-race-freedom using an approach such as the one in this thesis. The answer to this question depends on the cost of the loss of programming convenience and run-time performance incurred as a result of giving up shared memory. Therefore, there is probably no single answer to this question.

6.1 Future work

This thesis is by no means the final word on verification of concurrent programs.

Some potential areas of future work:

- Integrating rely-guarantee reasoning support, so that more properties can be verified, including properties of lock-free algorithms.
- Reducing annotation overhead by inferring annotations and by identifying common patterns and introducing more concise annotations for those, and by choosing defaults more appropriately.
- Improving verification performance by fine-tuning the logical encoding, improving the theorem prover, adding special VC generation rules for special cases, or by complementing it with specialized static analysis techniques.
- Supporting additional concurrent design patterns, such as reader-writer locks, or fine-grained locking techniques such as hand-over-hand locking or hierarchical locking.

Potential areas of future work on the inspector methods approach are noted in Section 3.8.

Bibliography

[1] Erika Ábrahám-Mumm, Frank S. de Boer, Willem-Paul de Roever, and Martin Steffen. Verification for Java's reentrant multithreading concept. In *FoSSaCS 2002*, volume 2303 of *LNCS*, pages 5–20. Springer, April 2002.

[2] Jean-Raymond Abrial. *The B-Book: Assigning Programs to Meanings*. Cambridge University Press, August 1996.

[3] Wolfgang Ahrendt, Thomas Baar, Bernhard Beckert, Richard Bubel, Martin Giese, Reiner Hähnle, Wolfram Menzel, Wojciech Mostowski, Andreas Roth, Steffen Schlager, and Peter H. Schmitt. The KeY tool. *Software and System Modeling*, 4(1):32–54, February 2005.

[4] Allen L. Ambler, Donald I. Good, James C. Browne, Wilhelm F. Burger, Richard M. Cohen, Charles G. Hoch, and Robert E. Wells. GYPSY: A language for specification and implementation of verifiable programs. *SIGPLAN Notices*, 12(3):1–10, March 1977.

[5] Ralph-Johan Back and Joakim von Wright. *Refinement Calculus: A Systematic Introduction*. Graduate Texts in Computer Science. Springer-Verlag, 1998.

[6] Thomas Ball, Shuvendu Lahiri, and Madanlal Musuvathi. Zap: Automated theorem proving for software analysis. Technical Report MSR-TR-2005-137, Microsoft Research, October 2005.

[7] John Barnes. *High Integrity Software: The SPARK Approach to Safety and Security*. Addison Wesley, 2003.

[8] Michael Barnett, Robert DeLine, Manuel Fähndrich, K. Rustan M. Leino, and Wolfram Schulte. Verification of object-oriented programs with invariants. *Journal of Object Technology*, 3(6):27–56, 2004.

[9] Mike Barnett, Bor-Yuh Evan Chang, Robert DeLine, Bart Jacobs, and K. Rustan M. Leino. Boogie: A modular reusable verifier for object-oriented

programs. In *Proceedings of the Fourth International Symposium on Formal Methods for Components and Objects (FMCO 2005)*, volume 4111 of *LNCS*. Springer, 2006.

[10] Mike Barnett, Robert DeLine, Manuel Fähndrich, K. Rustan M. Leino, and Wolfram Schulte. Verification of object-oriented programs with invariants. *Journal of Object Technology*, 3(6):27–56, 2004.

[11] Mike Barnett and K. Rustan M. Leino. Weakest-precondition of unstructured programs. In *Workshop on Program Analysis for Software Tools and Engineering (PASTE)*, pages 82–87, 2005.

[12] Mike Barnett, K. Rustan M. Leino, and Wolfram Schulte. The Spec# programming system: An overview. In *Construction and Analysis of Safe, Secure, and Interoperable Smart devices (CASSIS)*, volume 3362 of *Lecture Notes in Computer Science*, pages 49–60. Springer, 2004.

[13] Mike Barnett, K. Rustan M. Leino, and Wolfram Schulte. The Spec# programming system: An overview. In *CASSIS 2004*, volume 3362 of *LNCS*. Springer, 2004.

[14] Mike Barnett and David A. Naumann. Friends need a bit more: Maintaining invariants over shared state. In Dexter Kozen and Carron Shankland, editors, *Mathematics of Program Construction (MPC)*, volume 3125 of *Lecture Notes in Computer Science*, pages 54–84. Springer, 2004.

[15] Joachim van den Berg and Bart Jacobs. The LOOP compiler for Java and JML. In Tiziana Margaria and Wang Yi, editors, *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, volume 2031 of *Lecture Notes in Computer Science*, pages 299–312. Springer, 2001.

[16] Chandrasekhar Boyapati, Robert Lee, and Martin Rinard. Ownership types for safe programming: Preventing data races and deadlocks. In *OOPSLA 2002*, volume 37 of *SIGPLAN Notices*, pages 211–230. ACM, November 2002.

[17] L. Burdy, A. Requet, and J.-L. Lanet. Java applet correctness: a developer-oriented approach. In Keijiro Araki, Stefania Gnesi, and Dino Mandrioli, editors, *FME 2003: Formal Methods, International Symposium of Formal Methods Europe*, volume 2805 of *Lecture Notes in Computer Science*, pages 422–439. Springer, September 2003.

[18] Rod M. Burstall. Some techniques for proving correctness of programs which alter data structures. *Machine Intelligence*, 7:23–50, 1972.

[19] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems*, 13(4):451–490, October 1991.

[20] Ádám Darvas and Peter Müller. Reasoning about method calls in JML specifications. In Francesco Logozzo, editor, *Proceedings of the Seventh Workshop on Formal Techniques for Java-like Programs (FTfJP 2005)*, 2005.

[21] Robert DeLine and K. Rustan M. Leino. BoogiePL: A typed procedural language for checking object-oriented programs. Technical Report MSR-TR-2005-70, Microsoft Research, March 2005.

[22] David Detlefs, Greg Nelson, and James B. Saxe. Simplify: a theorem prover for program checking. *Journal of the ACM*, 52(3):365–473, May 2005.

[23] David L. Detlefs, K. Rustan M. Leino, Greg Nelson, and James B. Saxe. Extended static checking. Research Report 159, Compaq Systems Research Center, December 1998.

[24] David L. Detlefs, K. Rustan M. Leino, Greg Nelson, and James B. Saxe. Extended static checking. Research Report 159, Compaq Systems Research Center, December 1998.

[25] Edsger W. Dijkstra. *A Discipline of Programming.* Prentice Hall, Englewood Cliffs, NJ, 1976.

[26] Escher Technologies. Perfect Developer. http://eschertech.com/, 2006.

[27] Manuel Fähndrich and K. Rustan M. Leino. Declaring and checking non-null types in an object-oriented language. In Ron Crocker and Guy L. Steele Jr., editors, *Object-Oriented Programming Systems, Languages and Applications (OOPSLA)*, pages 302–312. ACM, 2003.

[28] Jean-Christophe Filliâtre. Verification of non-functional programs using interpretations in type theory. *The Journal of Functional Programming*, 13(4):709–745, July 2003.

[29] Jean-Christophe Filliâtre and Claude Marché. Multi-prover verification of C programs. In Jim Davies, Wolfram Schulte, and Michael Barnett, editors, *Formal Engineering Methods (ICFEM)*, volume 3308 of *Lecture Notes in Computer Science*, pages 15–29. Springer, 2004.

[30] Robert Fitzgerald, Todd B. Knoblock, Erik Ruf, Bjarne Steensgaard, and David Tarditi. Marmot: An Optimizing Compiler For Java. *Software—Practice and Experience*, 30(3):199–232, 2000.

[31] Cormac Flanagan and Stephen N. Freund. Atomizer: A dynamic atomicity checker for multithreaded programs. In *POPL 2004*, volume 39 of *SIGPLAN Notices*, pages 256–267. ACM, January 2004.

[32] Cormac Flanagan, K. Rustan M. Leino, Mark Lillibridge, Greg Nelson, James B. Saxe, and Raymie Stata. Extended static checking for Java. In *PLDI 2002*, volume 37 of *SIGPLAN Notices*, pages 234–245. ACM, May 2002.

[33] Cormac Flanagan, K. Rustan M. Leino, Mark Lillibridge, Greg Nelson, James B. Saxe, and Raymie Stata. Extended static checking for Java. In *Programming Language Design and Implementation (PLDI)*, pages 234–245, 2002.

[34] Cormac Flanagan and Shaz Qadeer. A type and effect system for atomicity. In *PLDI 2003*, pages 338–349. ACM, 2003.

[35] Cormac Flanagan and James B. Saxe. Avoiding exponential explosion: Generating compact verification conditions. In *POPL 2001: The 28th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 193–205. ACM, January 2001.

[36] Stephen N. Freund and Shaz Qadeer. Checking concise specifications for multithreaded software. *Journal of Object Technology*, 3(6):81–101, June 2004.

[37] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *The Java Language Specification (3rd Edition)*. Addison-Wesley, 2005.

[38] C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580,583, October 1969.

[39] C. A. R. Hoare. Monitors: An operating system structuring concept. *Communications of the ACM*, 17(10):549–557, October 1974.

[40] C. A. R. Hoare and N. Wirth. An axiomatic definition of the programming language PASCAL. *Acta Informatica*, 2(4):335–355, 1973.

[41] Bart Jacobs. Weakest pre-condition reasoning for Java programs with JML annotations. *Journal of Logic and Algebraic Programming*, 58(1–2):61–88, January–March 2004.

[42] Bart Jacobs, K. Rustan M. Leino, Frank Piessens, and Wolfram Schulte. Safe concurrency for aggregate objects with invariants. In *Proc. Int. Conf. Software Engineering and Formal Methods (SEFM 2005)*, pages 137–146. IEEE Computer Society, sep 2005.

[43] Bart Jacobs, K. Rustan M. Leino, and Wolfram Schulte. Verification of multithreaded object-oriented programs with invariants. In Mike Barnett, Stephen H. Edwards, Dimitra Giannakopoulou, Gary T. Leavens, and Natasha Sharygina, editors, *SAVCBS 2004 Workshop Proceedings*, 2004. Technical Report 04-09, Computer Science, Iowa State University.

[44] Bart Jacobs, Erik Meijer, Frank Piessens, and Wolfram Schulte. Iterators revisited: Proof rules and implementation. In Jan Vitek and Francesco Logozzo, editors, *Seventh International Workshop on Formal Techniques for Java-like Programs (FTfJP 2005)*, 2005.

[45] Bart Jacobs and Frank Piessens. Verification of programs with inspector methods. In Elena Zucca and Davide Ancona, editors, *Eighth International Workshop on Formal Techniques for Java-like Programs (FTfJP 2006)*, 2006.

[46] Bart Jacobs and Erik Poll. A logic for the Java Modeling Language JML. In H. Hussmann, editor, *Fundamental Approaches to Software Engineering (FASE)*, volume 2029 of *Lecture Notes in Computer Science*, pages 284–299. Springer, 2001.

[47] Bart Jacobs and Erik Poll. Java program verification at Nijmegen: Developments and perspective. In *Software Security—Theories and Systems, Second Mext-NSF-JSPS International Symposium, ISSS 2003*, pages 134–153, November 2003.

[48] Bart Jacobs, Jan Smans, Frank Piessens, and Wolfram Schulte. A statically verifiable programming model for concurrent object-oriented programs. In *Proceedings of the Eighth International Conference on Formal Engineering Methods (ICFEM2006)*, volume 4260 of *LNCS*. Springer, 2006.

[49] Ioannis T. Kassios. Dynamic frames: Support for framing, dependencies and sharing without restrictions. Technical Report 528, Dept. of Computer Science, University of Toronto, July 2005.

[50] Matt Kaufmann, Panagiotis Manolios, and J Strother Moore. *Computer-Aided Reasoning: An Approach.* Kluwer Academic Publishers, June 2000.

[51] Joseph R. Kiniry and David R. Cok. ESC/Java2: Uniting ESC/Java and JML: Progress and issues in building and using ESC/Java2, including a case study involving the use of the tool to verify portions of an Internet voting tally system. In *Construction and Analysis of Safe, Secure, and Interoperable Smart devices (CASSIS)*, volume 3362 of *Lecture Notes in Computer Science*, pages 108–128. Springer, 2004.

[52] Butler W. Lampson, James J. Horning, Ralph L. London, James G. Mitchell, and Gerald J. Popek. Report on the programming language Euclid. Technical Report CSL-81-12, Xerox PARC, October 1981. An earlier version of this report appeared as volume 12, number 2 in *SIGPLAN Notices*. ACM, February 1977.

[53] Gary T. Leavens, Albert L. Baker, and Clyde Ruby. JML: A notation for detailed design. In Haim Kilov, Bernhard Rumpe, and Ian Simmonds, editors, *Behavioral Specifications of Businesses and Systems*, pages 175–188. Kluwer Academic Publishers, Boston, 1999.

[54] Gary T. Leavens, Albert L. Baker, and Clyde Ruby. Preliminary design of JML: A behavioral interface specification language for Java. Technical Report 98-06u, Iowa State University, Department of Computer Science, April 2003.

[55] Gary T. Leavens, Albert L. Baker, and Clyde Ruby. Preliminary design of JML: A behavioral interface specification language for Java. Technical Report 98-06-rev28, Department of Computer Science, Iowa State University, July 2005.

[56] Dirk Leinenbach, Wolfgang Paul, and Elena Petrova. Towards the formal verification of a C0 compiler: Code generation and implementation correctness. In Bernhard K. Aichernig and Bernhard Beckert, editors, *Third IEEE International Conference on Software Engineering and Formal Methods (SEFM 2005)*, pages 2–12. IEEE Computer Society, September 2005.

[57] K. Rustan M. Leino. *Toward Reliable Modular Programs*. PhD thesis, CalTech, 1995. Available as Technical Report Caltech-CS-TR-95-03.

[58] K. Rustan M. Leino. Extended static checking: A ten-year perspective. In Reinhard Wilhelm, editor, *Informatics—10 Years Back, 10 Years Ahead*, volume 2000 of *Lecture Notes in Computer Science*. Springer, 2000.

[59] K. Rustan M. Leino, Todd Millstein, and James B. Saxe. Generating error traces from verification-condition counterexamples. *Science of Computer Programming*, 55(1–3):209–226, March 2005.

[60] K. Rustan M. Leino and Peter Müller. Modular verification of global module invariants in object-oriented programs. Technical Report 459, ETH Zürich, 2004.

[61] K. Rustan M. Leino and Peter Müller. Object invariants in dynamic contexts. In Martin Odersky, editor, *European Conference on Object-Oriented Programming (ECOOP)*, volume 3086 of *Lecture Notes in Computer Science*, pages 491–516. Springer-Verlag, 2004.

[62] K. Rustan M. Leino and Peter Müller. Modular verification of static class invariants. In John Fitzgerald, Ian J. Hayes, and Andrzej Tarlecki, editors, *Symposium on Formal Methods Europe (FM)*, volume 3582 of *Lecture Notes in Computer Science*, pages 26–42. Springer, 2005.

[63] K. Rustan M. Leino and Peter Müller. Modular verification of static class invariants. In *Proc. Formal Methods (FM 2005)*, 2005.

[64] K. Rustan M. Leino and Peter Müller. A verification methodology for model fields. In Peter Sestoft, editor, *European Symposium on Programming (ESOP)*, volume 3924 of *Lecture Notes in Computer Science*, pages 115–130. Springer, 2006.

[65] K. Rustan M. Leino and Peter Müller. A verification methodology for model fields. In *Proc. ESOP*, volume 3924 of *LNCS*. Springer-Verlag, 2006.

[66] K. Rustan M. Leino and Greg Nelson. Data abstraction and information hiding. *ACM Transactions on Programming Languages and Systems*, 24(5):491–553, September 2002.

[67] K. Rustan M. Leino, James B. Saxe, and Raymie Stata. Checking Java programs via guarded commands. In *Formal Techniques for Java Programs*, Technical Report 251. Fernuniversität Hagen, May 1999. Also available as Technical Note 1999-002, Compaq Systems Research Center.

[68] Claude Marché, Christine Paulin-Mohring, and Xavier Urbain. The KRAKATOA tool for certification of JAVA/JAVACARD programs annotated in JML. *Journal of Logic and Algebraic Programming*, 58(1–2):89–106, January–March 2004.

[69] Bertrand Meyer. *Eiffel: The Language*. Object-Oriented Series. Prentice Hall, 1992.

[70] Peter Müller. *Modular Specification and Verification of Object-Oriented Programs*, volume 2262 of *LNCS*. Springer-Verlag, 2002.

[71] Peter Müller, Jörg Meyer, and Arnd Poetzsch-Heffter. Programming and interface specification language of JIVE—specification and design rationale. Technical Report 223, Fernuniversität Hagen, 1997.

[72] Greg Nelson. A generalization of Dijkstra's calculus. *ACM Transactions on Programming Languages and Systems*, 11(4):517–561, October 1989.

[73] Sam Owre, S. Rajan, John M. Rushby, Natarajan Shankar, and Mandayam K. Srivas. PVS: Combining specification, proof checking, and model checking. In Rajeev Alur and Thomas A. Henzinger, editors, *Computer-Aided Verification*

(CAV), volume 1102 of *Lecture Notes in Computer Science*, pages 411–414. Springer, 1996.

[74] Matthew J. Parkinson. *Local reasoning for Java*. PhD thesis, Computer Laboratory, Cambridge University, 2005.

[75] Matthew J. Parkinson and Gavin M. Bierman. Separation logic and abstraction. In *Proc. POPL*, 2005.

[76] Arnd Poetzsch-Heffter. Specification and verification of object-oriented programs. Habilitationsschrift, Technische Universität München, 1997.

[77] Shaz Qadeer, Sriram K. Rajamani, and Jakob Rehof. Summarizing procedures in concurrent programs. In *POPL 2004*, volume 39 of *SIGPLAN Notices*, pages 245–255. ACM, January 2004.

[78] John C. Reynolds. Syntactic control of interference. In *Fifth ACM Symposium on Principles of Programming Languages (POPL)*, pages 39–46, January 1978.

[79] Edwin Rodríguez, Matthew Dwyer, Cormac Flanagan, John Hatcliff, Gary T. Leavens, and Robby. Extending sequential specification techniques for modular specification and verification of multi-threaded programs. In *ECOOP 2005*, volume 3586 of *LNCS*, pages 551–576. Springer, July 2005.

[80] David S. Rosenblum. A practical approach to programming with assertions. *IEEE Transactions on Software Engineering*, 21(1):19–31, January 1995.

[81] Stefan Savage, Michael Burrows, Greg Nelson, Patrick Sobalvarro, and Thomas E. Anderson. Eraser: A dynamic data race detector for multithreaded programs. *ACM Transactions on Computer Systems*, 15(4):391–411, November 1997.

[82] Jan Smans, Bart Jacobs, and Frank Piessens. Static verification of Code Access Security policy compliance of .NET applications. *Journal of Object Technology*, 5(3):35–58, April 2006.

[83] Spec# homepage. http://research.microsoft.com/specsharp, 2006.

[84] Mark T. Vandevoorde. *Exploiting Specifications to Improve Program Performance*. PhD thesis, Massachusetts Institute of Technology, February 1994. Available as Technical Report MIT/LCS/TR-598.

[85] Adam Welc, Suresh Jagannathan, and Antony L. Hosking. Transactional monitors for concurrent objects. In *ECOOP 2004*, volume 3086 of *LNCS*. Springer, June 2004.

List of Publications

International journals

- Jan Smans, Bart Jacobs, and Frank Piessens. Static verification of Code Access Security policy compliance of .NET applications. Journal of Object Technology 5 (3), 2006.

- Frank Piessens, Bart Jacobs, Eddy Truyen, and Wouter Joosen. Support for metadata-driven selection of run-time services in .NET is promising but immature. Journal of Object Technology 3 (2), 2004.

- Frank Piessens, Bart Jacobs, and Wouter Joosen. Software security: experiments on the .NET common language run-time and the shared source common language infrastructure. IEE Proceedings - Software 150 (05), 2003.

International conferences and workshops

- Bart Jacobs, Frank Piessens, and Wolfram Schulte. VC generation for functional behavior and non-interference of iterators. In Proceedings of the Fifth Workshop on Specification and Verification of Component-Based Systems (SAVCBS 2006), ACM Press, 2006.

- Bart Jacobs, Jan Smans, Frank Piessens, and Wolfram Schulte. A statically verifiable programming model for concurrent object-oriented programs. In Proceedings of the Eighth International Conference on Formal Engineering Methods (ICFEM 2006), LNCS 4260, Springer, 2006.

- Bart Jacobs, Jan Smans, Frank Piessens, and Wolfram Schulte. A simple sequential reasoning approach for sound modular verification of mainstream multithreaded programs. In Proceedings of the First International Workshop on Multithreading in Hardware and Software: Formal Approaches to Design and Verification (TV 2006), 2006. Presented as an Invited Talk by Wolfram

Schulte and Bart Jacobs, August 21, 2006. Published on the website. Also to appear in Electronic Notes in Theoretical Computer Science, Elsevier.

- Mike Barnett, Bor-Yuh Evan Chang, Robert DeLine, Bart Jacobs, and K. Rustan M. Leino. Boogie: A modular reusable verifier for object-oriented programs. In Proceedings of the Fourth International Symposium on Formal Methods for Components and Objects (FMCO 2005), LNCS 4111, Springer, 2006.

- Bart Jacobs, and Frank Piessens. Verification of programs using inspector methods. In Proceedings of the Eighth Workshop on Formal Techniques for Java-like Programs (FTfJP 2006), 2006. Published on the website.

- Bart Jacobs, K. Rustan M. Leino, Frank Piessens, and Wolfram Schulte. Safe concurrency for aggregate objects with invariants. In Proceedings of the Third International Conference on Software Engineering and Formal Methods (SEFM 2005), IEEE Computer Society, 2005.

- Bart Jacobs, Erik Meijer, Frank Piessens, and Wolfram Schulte. Iterators revisited: proof rules and implementation. In Proceedings of the Seventh Workshop on Formal Techniques for Java-like Programs (FTfJP 2005), 2005. Published on the website.

- Mike Barnett, Rob DeLine, Bart Jacobs, Manuel Fähndrich, K. Rustan M. Leino, Wolfram Schulte, and Herman Venter. The Spec# programming system: challenges and directions. In Proceedings of the First Workshop on Verified Software: Tools, Techniques, and Experience (VSTTE 2005), 2005. Published on the website.

- Bart Jacobs, K. Rustan M. Leino, and Wolfram Schulte. Verification of multithreaded object-oriented programs with invariants. In Proceedings of the Third Workshop on Specification and Verification of Component-Based Systems (SAVCBS 2004), Technical Report 04-09, Department of Computer Science, Iowa State University, 2004.

- Lieven Desmet, Bart Jacobs, Frank Piessens, and Wouter Joosen. A generic architecture for web applications to support threat analysis of infrastructural components. In Proceedings of the Eighth IFIP TC-6 TC-11 Conference on Communications and Multimedia Security (CMS 2004), Springer, 2004.

- Lieven Desmet, Bart Jacobs, Frank Piessens, and Wouter Joosen. Threat modeling for web services based web applications. In Proceedings of the Eighth IFIP TC-6 TC-11 Conference on Communications and Multimedia Security (CMS 2004), Springer, 2004.

- Bart Jacobs, Frank Piessens, Pieter Bekaert, and Eric Steegmans. Whole-program specifications permit better abstraction and concurrent implementations. In Proceedings of the FastAbstracts Track of the Thirteenth International Symposium on Software Reliability Engineering (ISSRE 2002), 2002. Published on the website.

Technical reports

- Bart Jacobs, Frank Piessens, and Wolfram Schulte. Safe fine-grained locking for aggregate objects. Technical Report CW 444, Department of Computer Science, Katholieke Universiteit Leuven, 2006.
- Bart Jacobs and Frank Piessens. Verifying programs using inspector methods for state abstraction. Technical Report CW 432, Department of Computer Science, Katholieke Universiteit Leuven, 2005.
- Bart Jacobs, K. Rustan M. Leino, Frank Piessens, and Wolfram Schulte. Safe concurrency for aggregate objects with invariants: soundness proof. Technical Report MSR-TR-2005-85, Microsoft Research, 2005.
- Bart De Win, Bart Jacobs, Wouter Joosen, Gregory Neven, Frank Piessens, and Tine Verhanneman. Formal technologies for information and software security: An annotated bibliography. Technical Report CW 423, Department of Computer Science, Katholieke Universiteit Leuven, 2005.
- Bart Jacobs and Frank Piessens. A pi-calculus semantics of Java: the full definition. Technical Report CW 355, Department of Computer Science, Katholieke Universiteit Leuven, 2003.

Biography

Bart Jacobs

- Born on December 9, 1978 in Diest, Belgium.
- Received the degree of *Licentiaat in de Informatica* (a four-year computer science curriculum) at the Katholieke Universiteit Leuven, Belgium on July 6, 2002.
- Licentiaatsthesis: *Elements of Formal Specification and Verification of Java Modules as an Application of the Computer-Aided Development of Logical Theories.*
- Started his PhD at the Department of Computer Science, Katholieke Universiteit Leuven, Belgium, on September 1, 2002. Supervisor: Prof. Frank Piessens.
- Doctoraatsbursaal K.U.Leuven, September 1, 2002 – September 30, 2003.
- Research Assistant of the Flemish Fund for Scientific Research (F.W.O.-Vlaanderen) (Belgium), October 1, 2003 – September 30, 2007.
- Internship at Microsoft Research, Redmond, WA, USA, July 5 – September 24, 2004. Host: Dr. Wolfram Schulte.
- Internship at Microsoft Research, Redmond, WA, USA, March 15 – September 2, 2005. Host: Dr. Wolfram Schulte.