Masterscriptie Software Engineering

Extractie van dode code in een heterogeen systeem

Statische analyse in combinatie met dynamische analyse

J.J.H. van Willegen

Amsterdam, 15-08-2006

Eenjarige Master Software Engineering
Afstudeerdocent: Prof. Dr. P. Klint
Opdrachtgever: Centrum voor Wiskunde en Informatica

Publicatiestatus: openbaar

Universiteit van Amsterdam,
Hogeschool van Amsterdam,
Vrije Universiteit

Samenvatting

Voor het vaststellen van dode code binnen een applicatie is het benodigd om inzicht te krijgen in methodes die worden aangeroepen en methodes die niet worden aangeroepen. Op het moment dat binnen de programmeertaal waarin de te analyseren applicatie is geschreven sprake is van mechanismen zoals polymorfisme treden hier moeilijkheden op. Als gevolg van polymorfisme ontstaat er onzekerheid over de runtime uitgevoerde methode wanneer statisch wordt getracht te bepalen welke methodes wel en niet worden uitgevoerd. Om dit probleem te beperken kan gebruik worden gemaakt van geavanceerde type-analyse systemen zoals in het geval van points-to analysis[18]. Probleem hierbij is dat deze algoritmes vaak complex zijn en de taal-afhankelijkheid van de analyse doen toenemen.

Binnen dit onderzoek wordt een dode code analyse gepresenteerd welke bestaat uit een statische analyse gecombineerd met een dynamische analyse. Hierbij vindt binnen de statische analyse slechts een beperkte type analyse plaats die volledig gericht is op het bepalen van de compile-time types van variabelen en de methode-aanroepen die hierop plaatsvinden. Er wordt niet geredeneerd over de runtime types waarnaar de variabelen zouden kunnen verwijzen maar er wordt actief met deze onzekerheid omgesprongen.

De statische analyse maakt een opdeling in een dode code set, werkende code set en een grijze set. In de dode code set komen alle methodes waarvan vooraf met zekerheid kan worden vastgesteld dat deze niet worden aangeroepen. In de werkende code set komen alle methodes waarvan vooraf met zekerheid kan worden aangegeven dat deze mogelijk direct worden aangeroepen (zoals bij constructors of in het geval van casting). De grijze set bevat als gevolg hiervan alle methodes waarvan door toedoen van polymorfisme niet met zekerheid kan worden bepaald of deze wel of niet worden aangeroepen. Van deze grijze set kan vervolgens met de dynamische analyse worden bepaald welke methodes runtime worden aangeroepen. Hiervoor moeten scenario's voor het gebruik van de applicatie worden opgesteld die verband houden met de methodes in de grijze set. Na het uitvoeren van de dynamische analyse kunnen de geconstateerde aangeroepen methodes weer als entry-points naar de applicatie worden beschouwd en weer in de statische analyse worden gebruikt om met een grotere zekerheid te redeneren. Door het steeds verder wegnemen van de onzekerheid als gevolg van mechanismen als polymorfisme wordt de kans groter dat de methodes die in de grijze set overblijven ook dood zijn.

Groot voordeel van bovenstaande aanpak is dat er geen ingewikkeld type-analyse systeem nodig is en dus de bijbehorende administratie volledig kan vervallen. Door het op deze manier lichtgewicht maken van de statische analyse kan de taal-afhankelijkheid worden beïnvloed wat van belang is bij heterogene systemen. In het onderzoek zal naar voren komen dat door de combinatie van statische analyse en dynamische analyse de taal-afhankelijkheid valt te sturen door de dynamische analyse een groter component te laten uitmaken van de dode code analyse. Zo wordt namelijk de statische analyse steeds minder taal-afhankelijk. Hiervoor is een bepaald optimum dat afhangt van verschillende eigenschappen welke in het onderzoek de revue zullen passeren.

Binnen het onderzoek is een praktische implementatie van de dode code analyse nagestreefd. De implementatie is gedeeltelijk gerealiseerd en het nastreven ervan heeft een groot voordeel gehad doordat verschillende praktische problemen zijn belicht. De dode code analyse is vergeleken met de ongebruikte methodes die de tool RevJava[W18] detecteerde bij een geconstrueerd testvoorbeeld. Hieruit bleek dat de analyse zoals deze binnen het onderzoek is nagestreefd meer ongebruikte methodes kon detecteren dan deze tool.

Voorwoord

Overzicht scriptie
In dit afstudeeronderzoek wordt ingegaan op de ontwikkeling van een dode code analyse voor een heterogeen systeem. Van groot belang is hierbij dus de afweging met betrekking tot taal-afhankelijkheid van de analyse. Om deze taal-afhankelijkheid te beperken en te kunnen sturen is gekozen voor een aanpak waarbij een statische en dynamische analyse worden gecombineerd. In het eerste hoofdstuk zal de onderzoeksvraag worden beschreven die binnen dit onderzoek wordt beantwoord. Het tweede hoofdstuk gaat in op bestaande literatuur die is gebruikt voor het vormen van een beeld van de problematiek die een rol speelt bij dode code analyse. In het derde hoofdstuk wordt kort ingegaan op de het plan van aanpak en de hypothese. Het vierde hoofdstuk gaat dieper in op de onderzoeksvraag en geeft onder andere antwoord op de vraag wat de voordelen zijn van dode code analyse. In het vijfde hoofdstuk wordt ingegaan op de statische analyse en de realisatie hiervan en in hoofdstuk zes wordt vervolgens de dynamische analyse besproken. Het zevende hoofdstuk is geheel gewijd aan de taal-afhankelijkheid en de factoren die daarbij een rol spelen. Dit hoofdstuk speelt een grote rol bij de beantwoording van de onderzoeksvraag . In hoofdstuk acht worden vervolgens de resultaten besproken van het onderzoek. Dit bestaat uit een overzicht van voor- en nadelen van de nieuwe aanpak, het verkregen inzicht door de beantwoording van onderzoeksvraag, toepassing van de dode code analyse op een geconstrueerd voorbeeldprogramma en enkele opmerkingen over mogelijk toekomstig werk. In het laatste hoofdstuk wordt nog even kort teruggeblikt op het algehele project.

Leeswijzer
De belangrijke conclusies zijn gemarkeerd met een uitroepteken en omkaderd in een tabel. Hiermee kan snel inzicht worden verschaft in belangrijke conclusies binnen een hoofdstuk of paragraaf. De bijlages zijn bedoeld voor diegene die meer in detail willen weten hoe de dode code analysetechniek werkt en hoe bijvoorbeeld is omgegaan met specifieke eigenschappen van de programmeertaal Java.

Dankwoord
Allereerst wil ik Paul Klint bedanken voor het bieden van de mogelijkheid aan mij om af te studeren bij het Centrum voor Wiskunde en Informatica (CWI) en het mij laten uitvoeren van een onderzoek op een uitdagend gebied. Daarnaast wil ik Jurgen Vinju en Mark van den Brand bedanken voor hun medewerking en inspiratie. Ook wil ik de andere medewerkers van de afdeling Software Engineering (SEN) bedanken voor de prettige werksfeer. Bij deze wil ik ook nogmaals mijn reviewers, Corina van Willegen, Arnold Lankamp en Paul Klint, bedanken voor de tijd die zij hebben genomen om mijn scriptie door te lezen en te voorzien van commentaar.

Inhoudsopgave

1. Onderzoeksvraag

Tijdens de ontwikkeling van een softwareproduct worden er verschillende processen doorlopen. Er wordt allereerst begonnen met het ontwerpen van de software. Dit zal leiden tot een eerste versie van de applicatie. Een proces wat hieruit voortvloeit, is dat de applicatie moet worden verrijkt met nieuwe functionaliteit om aan bijvoorbeeld de groeiende vraag van de markt te blijven voldoen. Een proces dat te allen tijde een grote rol speelt is het onderhoud van de applicatie.

Grote softwareprojecten kenmerken zich door de meestal lange looptijd en het feit dat er meerdere programmeurs aan werken. Als gevolg van de lange looptijd zal er bijvoorbeeld verloop zijn in personeel. Een ander kenmerk is dat verschillende mensen aan dezelfde componenten van een applicatie werken. Een risico hiervan is dat het voor de verschillende partijen die aan een component werken onduidelijk wordt wat de daadwerkelijk gebruikte functionaliteit van een component is ten opzichte van de geprogrammeerde functionaliteit. Als gevolg hiervan ontstaat het fenomeen dat wordt aangeduid met dode code.

Met dode code worden die delen van de broncode van een applicatie bedoeld die nooit zullen worden gebruikt tijdens de executie van een programma. De delen waarin zich dode code bevindt zijn dus wel opgenomen in de applicatie maar dienen geen doel voor de functionaliteit van de applicatie ten opzichte van hetgeen aan de eindgebruiker wordt aangeboden. Dode code kan men omschrijven als de delen van de broncode van de applicatie die men kan weglaten zonder dat daarmee de semantische eigenschappen van de applicatie veranderen.

Eén van de voordelen van het detecteren en verwijderen van dode code is een verbetering van de onderhoudbaarheid van de applicatie. Aangezien de meeste tijd die in een applicatie wordt gestopt betrekking heeft op onderhoudbaarheid is het interessant om de dode code te verwijderen.

Het idee voor het verwijderen van dode code is niet nieuw en uit de literatuurstudie blijkt ook dat er al verschillende pogingen zijn gedaan om hiervoor een goed algoritme te ontwerpen. Het grootste probleem bij de verschillende statische aanpakken blijft de type-analyse voor het achterhalen van de specifieke methode die als gevolg van een aanroep wordt uitgevoerd. Het probleem is de onzekerheid die ontstaat over de aangeroepen methode door het gebruik van mechanismen zoals pointers en polymorfisme. Er zijn verschillende oplossingsrichtingen, zoals met behulp van slicing [9][12] en points-to analysis[18], om het type-analyseprobleem te verminderen. Een bijkomend effect van dit soort aanpakken is dat de complexiteit van de analyse toeneemt en de daadwerkelijke realisatie ervan moeilijk wordt. De moeilijkheid van sommige algoritmes bleek ook uit de hoeveelheid artikelen die slechts een theoretisch aanpak beschrijven maar hiervan geen praktische implementatie geven.

Het hoofdonderwerp van deze scriptie richt zich op een nieuwe oplossingsstrategie voor het lokaliseren van dode code door middel van een combinatie tussen statische en dynamische analyse. Het doel hiervan is om op deze manier de voordelen van zowel statische als dynamische analyse te combineren en de nadelen van beide aanpakken te compenseren. Hieruit zou de mogelijkheid moeten vloeien om op een relatief eenvoudige wijze met de onzekerheid op het gebied van type-analyse om te gaan. Door te redeneren met de onzekerheid als gevolg van het type-analyseprobleem en niet te proberen de statische analyse perfect te krijgen, kan de statische analyse beperkt worden in zijn complexiteit. Hierdoor zou de analyse ook realiseerbaar moeten zijn voor applicaties in verschillende programmeertalen.

De onderzoeksvraag die uit bovenstaande bespreking van het onderwerp volgt is:

> ***Op welke manier kunnen statische en dynamische analyse elkaar aanvullen ten behoeve van dode code analyse in een heterogeen systeem.***

Als context waarbinnen de nieuw te ontwikkelen strategie zou kunnen worden toegepast is gekozen voor een groot softwareproject genaamd de ASF+SDF Meta-environment. De Meta-omgeving is een verzameling tools die de gebruiker de mogelijkheid biedt om de syntax en semantiek van een (programmeer)taal te beschrijven en zo bijvoorbeeld transformaties mogelijk te maken naar andere talen[5]. De Meta-omgeving is een component-gebaseerd systeem waarbij de componenten (tools) in verschillende programmeertalen zijn geschreven. Hieruit volgt ook de voorkeur om de dode code analyse zo taal-onafhankelijk mogelijk te maken.

Uit de onderzoeksvraag en de bespreking komen enkele deelvragen naar voren:

(1) *Op welke wijze kan binnen de Meta-omgeving dode code worden geconstateerd*
    - *Hoofdstuk 4.4..*
(2) *In hoeverre is het mogelijk om het analyse gedeelte taal-onafhankelijk te houden.*
    - *Hoofdstuk 5.2, 6, 7.*
(3) *Op welke manier kan de programmeur op de hoogte worden gesteld van de delen dode code.*
    - *Hoofdstuk 4.5*
(4) *Hoe kan worden gevalideerd of de gevonden dode code ook daadwerkelijk niet gebruikt wordt.*
    - *Hoofdstuk 2.4*
(5) *Wat zijn specifieke eigenschappen van een programmeertaal waarmee rekening moet worden gehouden in de analyse*
    - *Hoofdstuk 5.2, 6.1 en bijlage D*

Gezien de beperkte tijd die voor het afstudeeronderzoek beschikbaar is moeten er prioriteiten worden gesteld met betrekking tot de beantwoording van de onderzoeksvraag en de deelvragen. De doelstellingen van dit afstudeeronderzoek op basis van de priorisering zijn:

(1) *Creëer een algemeen beeld over hoe dode code binnen een component-gebaseerd systeem kan worden bepaald op basis van specifieke eigenschappen van de Meta-omgeving*
    - *Hoofdstuk 4.4*
(2) *Maak een gedetailleerde uitwerking van de manier waarop een statische analyse in combinatie met een dynamische analyse kan worden gebruikt en wat hiervan de voor- en nadelen zijn.*
    - *Hoofdstuk 4.2, 4.3, 5, 6 en 8*
(3) *Richt de gedetailleerde uitwerking van de dode code analyse op eigenschappen van Java maar verlies de taal C niet uit het oog en houdt dus rekening met de taal-onafhankelijkheid*
    - *Hoofdstuk 5, 6, 7 en bijlage B,C,D en G*
(4) *Probeer zo ver mogelijk te komen met de daadwerkelijk praktische implementatie van de analyse*
    - *Hoofdstuk 5.2, 6.2 en bijlage B,C,D, F en G*

Nu de onderzoeksvraag, deelvragen en de daaruit volgende doelstellingen bekend zijn zal in het volgende hoofdstuk een introductie worden gegeven van de verschillende facetten die een rol spelen bij het onderzoek. Dit gebeurt door middel van een bespreking van gevonden literatuur en de kennis die als invoer is gebruikt voor het behalen van de doelstellingen.

2. Achtergrond en context

In dit hoofdstuk zal door middel van een globale bespreking van de onderdelen die een rol spelen in het afstudeeronderzoek een link worden gelegd met de gevonden literatuur. Deze literatuur is gebruikt om inzicht te krijgen in verschillende onderdelen die tijdens het onderzoek aan bod komen en om op deze wijze een afweging te kunnen maken in welke richting het onderzoek zal gaan. De informatie uit deze artikelen heeft de basis gelegd voor de oplossingsstrategie die in dit onderzoek wordt nagestreefd.

In de eerste paragraaf zal worden ingegaan op de opvatting van dode code en de door mij aangehouden definitie hiervan. In de tweede paragraaf zal kort worden ingegaan op het heterogene systeem dat met de resultaten volgend uit dit onderzoek kan worden geanalyseerd. In de derde paragraaf zullen enkele vormen van dode code extractie worden besproken zoals deze in de literatuur naar voren kwamen. Er wordt stilgestaan bij methodes voor zowel statische als dynamische analyse aangezien hiertussen fundamentele verschillen zitten en deze verschillen van belang zijn voor het beantwoorden van de hoofdvraag. De laatste paragraaf zal ingaan op mogelijke vormen van evaluatie van de gevonden dode code.

2.1 Definitie van dode code

Voor het onderzoek is het allereerst van belang om inzicht te krijgen in de verschillende opvattingen die over dode code bestaan. In [11] wordt onderscheid gemaakt tussen "*Dead code*" en "*Partially dead code*". Dit is niet het enige artikel dat dode code definieert, maar in dit artikel wordt een uitvoerig onderbouwde definitie gegeven. De belangrijkste opmerking is "*dead code may be removed from a program without changing the program's result*". Hieruit volgt de volgende definitie van dode code:

> ***Onder dode code verstaan we alle delen van een programma die kunnen worden verwijderd zonder dat daarbij semantische eigenschappen van het programma veranderen.***

Bovenstaande definitie zal worden gehanteerd op het moment dat er sprake is van dode code. Onder deze definitie vallen vele vormen van dode code. Het afstudeeronderzoek beperkt zich tot het vinden van dode code op het niveau van methodes. Eén van de voordelen van het extraheren van dode code is [11] "*performance enhancement*". Deze prestatieverbetering heeft positieve effecten op zowel het compileerproces als op de runtime uitvoering van een applicatie. Daarnaast is er een positief effect op de onderhoudbaarheid van de applicatie.

2.2 Te onderzoeken heterogeen systeem

De applicatie welke de basis vormde voor de onderzoeksvraag en waarvan vooraf de intentie was om ervan de dode code te achterhalen is de Meta-omgeving. Deze applicatie is ontwikkeld bij het CWI. In [5] wordt de Meta-omgeving gekarakteriseerd als "*an interactive development environment for the automatic generation of interactive systems for constructing language definitions and generating tools for them.*". De Meta-omgeving kan dus worden gebruikt voor het interpreteren en transformeren van invoer waarvan men een syntactische beschrijving heeft. Een voorbeeld van een eenvoudige toepassing is het bepalen van de normaalvorm van een booleaanse expressie.
De Meta-omgeving bestaat uit een verzameling componenten (tools) die via de ToolBus met elkaar zijn verbonden [2] "*The goal of the ToolBus is to integrate tools written in different languages running on different machines*". Het feit dat er sprake kan zijn van componenten geschreven in verschillende programmeertalen is een belangrijk punt om rekening mee te houden voor de uiteindelijk toe te passen techniek voor dode code extractie.
De Meta-omgeving kan naast bron van onderzoek ook gebruikt worden om de extractie van dode code te ondersteunen. Met de ASF+SDF specificatie [6] kunnen herschrijfregels worden uitgedrukt waarmee bijvoorbeeld de broncode van componenten kunnen worden aangepast en feiten kunnen worden geëxtraheerd. De ToolBus scripts beschrijven de interface die de componenten aanbieden aan de ToolBus. Wanneer dit in een ASF+SDF specificatie wordt uitgedrukt zou men bijvoorbeeld kunnen redeneren over de mogelijke entry-points naar een component. De entry-points zijn van belang voor het vinden van dode code.

2.3 Analysetechnieken voor dode code extractie

Voor het extraheren van dode code is het nodig om een analyse uit te voeren op de broncode van het desbetreffende programma. Een analyse kan zowel statisch als dynamisch zijn [10].
In het geval van een statische analyse geeft [10] "*Static analysis examines the program code and reasons over all possible behaviors that might arise at run time*". Het belangrijkste hierbij is het gebruik van het woord

"*might*". Er wordt dus geredeneerd over het gedrag dat tijdens de executie van het programma zou kunnen optreden maar het programma wordt niet daadwerkelijk uitgevoerd.
In het geval van een dynamische analyse geeft [10] "*Dynamic analysis operates by executing a program and observing the executions.*". In dit geval wordt dus daadwerkelijk naar de runtime executie van het programma gekeken. Er wordt dus niet geredeneerd op basis van de broncode. Het is duidelijk dat de conclusies die kunnen worden getrokken uit statische en dynamische analyse verschillend zijn.
In [10] worden enkele voor- en nadelen van beide analysemethoden gegeven. De belangrijkste conclusie is dat het gaat om [10] "*notably sound versus precision*". Hiermee wordt bedoeld dat conclusies, volgend uit een statische analyse, altijd correct[1] zijn en er dus mee kan worden geredeneerd. Echter, in het geval van een dynamische analyse kunnen incorrecte conclusies worden getrokken. Dit heeft te maken met het feit dat een dynamische analyse afhankelijk is van de runtime eigenschappen. Met dynamische analyse is het vervolgens wel mogelijk nauwkeurigere antwoorden te geven dan in het geval van een statische analyse. Dit is logisch als men bedenkt dat er bijvoorbeeld gebruik kan worden gemaakt van pointers in programma's of er sprake kan zijn van polymorfisme. Hetgeen waarnaar de pointers refereren is lang niet altijd statisch te bepalen, maar deze informatie is runtime wel beschikbaar. Overigens geeft [9] een techniek waarmee het pointer-probleem statisch kan worden beperkt door "*slicing techniques aim at extracting a minimal program that captures the behaviour of a source-code program with respect to a specified variable"*. In het artikel wordt er niet specifiek over gesproken maar naar mijn mening kan een slicing techniek worden gebruikt om te bepalen op welke punten in een programma de waarde van een pointer wordt beïnvloed. Hierdoor kan men een "boundary set" bepalen zodat men weet in welke range de pointer zich bevindt. In [18] wordt wel specifiek ingegaan op een techniek (point-to analysis) voor het bepalen waar variabelen binnen een Java programma naar kunnen verwijzen. Hiervoor wordt gebruik gemaakt van een graaf waarin onder andere staat welke constructors op welke variabelen zijn toegepast. Hiermee wordt op een flow-onafhankelijke manier voor een specifieke variabele bepaald waar deze naar kan verwijzen.
In [12] worden verschillende slicingtechnieken besproken en wordt ook kort aangegeven dat static slicing kan worden gebruikt voor het bepalen van dode code. In dit artikel ziet men dode code als "*statements which cannot affect any output of the program*".

Het feit dat statische en dynamische analyse veel tegengestelde voor- en nadelen hebben maakt het zeer interessant om beide te gebruiken tijdens de extractie van dode code en de toepasbaarheid ervan te onderzoeken. In theorie zou een dynamische analyse antwoorden moeten kunnen geven die met een statische analyse niet kunnen worden achterhaald en omgekeerd.

2.3.1 Statische analyse

Eén van de theoretische artikelen welke zich toespitst op statische analyse is [11] waarin de revival transformation wordt besproken. Deze methode wordt in [11] omschreven als "*the revival transformation detaches a partially dead definition d from the point(s) at which it is attached in program P and reattaches it at a point(s) at which it is maximally live*". Het gaat in dit geval om de optimalisatie van een applicatie door bijvoorbeeld de definitie van een variabele die slechts in een beperkt aantal conditionele paden wordt gebruikt te verplaatsen naar de paden waar de definitie daadwerkelijk wordt gebruikt. Het kan bijvoorbeeld zo zijn dat een definitie van een variabele slechts in het geval van het ingaan van een else-tak wordt gebruikt en niet in de then-tak. Deze definitie kan dan, als optimalisatie, naar de else-tak worden verhuisd. Het gaat hier dus om "*partially dead code*".

Om te kunnen achterhalen of het geoptimaliseerde programma (waarvan de dode code is verwijderd) nog dezelfde semantische eigenschappen heeft als het originele programma moet een verificatie plaatsvinden. In [4] wordt uitgelegd hoe dit theoretisch zou kunnen worden aangepakt "*we formally verify dead code elimination (DCE) within the theorem prover Isabelle/HOL*". Er wordt ook aangegeven dat een algoritme voor optimalisatie aan een voorwaarde moet voldoen [4] "*not change the semantics of programs*". Dit is ook wat er in de eerder gegeven definitie van dode code naar voren komt.

Naast theoretische artikelen over de extractie van dode code zijn er ook artikelen over praktische toepassingen te vinden. In [8] wordt dode code extractie gezien als "*comparing the closure set against a complete set of program entities*". Hier wordt dus gesteld dat voor de extractie van dode code het verschil wordt genomen tussen de aangeroepen methodes (vanuit een bepaald entry-point in de applicatie) en de volledige verzameling van gedeclareerde methodes binnen de applicatie. Dit is in eerste instantie een zeer logische definitie maar [8] "*The lesson we learned immediately was that not all entities reported by our tools should be deleted*". Dit heeft te maken met het feit dat de applicaties op verschillende manieren (stand-alone, shared) konden worden gebruikt en

[1] Gegeven een correct algoritme voor extractie van de feiten en aannames waaronder deze extractie plaatsvindt.

de entry-points dus in lang niet alle gevallen duidelijk waren. Dit is precies het probleem dat bij de Meta-omgeving ook zal spelen aangezien [5] "*components of the Meta-environment are used as stand-alone tools in a variety of applications"*. De componenten behorende bij de Meta-omgeving worden dus in combinatie met de ToolBus en stand-alone gebruikt. Het gevolg is dat er false positives kunnen ontstaan in de set van dode code.

Voor de verwerking van de geëxtraheerde feiten uit de broncode van applicaties [7] "*query performance can be improved … by simply using a database*.". Dit geeft ook een mogelijkheid om de feiten van de programma's vanuit de verschillende programmeertalen naar één vorm te converteren wat belangrijk is voor een heterogeen systeem. Zo kunnen de query's op de geëxtraheerde feiten worden gedaan zonder dat men weet in welke programmeertaal een component is geschreven.

Voor de eclipse omgeving biedt [3] "*Our plug-in provides an easy interface to the programmer to define and use verification tasks*" een eclipse plug-in om onder andere een reachability analyse te doen. Op het moment dat een reachability analyse mogelijk is, is het uiteraard ook mogelijk om dode code te vinden.

Uiteindelijk is de belangrijkste conclusie die kan worden getrokken uit de drie theoretische artikelen [4], [9], [11] en de artikelen over de praktische toepassing [3], [7], [8] dat een call-graph in alle gevallen van statische analyse een belangrijke rol speelt en onmisbaar is. De statische oplossing die zal worden nagestreefd in het afstudeeronderzoek is hierop ook gebaseerd en zal in paragraaf 4.3.2 worden besproken.

2.3.2 Dynamische analyse

Voor dynamische analyse van een programma wordt vaak gebruik gemaakt van een profiler. Het doel van program profiling is volgens [1] "*captures which statements execute, but not the order in which they run.*". Een voordeel van profiling is dat [1] *"a path profile helps a compiler statically predict a program's expected behaviour, which enables more precise program analysis and facilitates optimization decisions.*". In [17] wordt een combinatie gemaakt tussen statische en dynamische analyse waarbij de dynamische analyse wordt gebruikt om het resultaat van de statische analyse te evalueren "*a dynamic call chain analysis takes as input a set of call chains computed by some static call chain analysis*". Onder static call chain analyses wordt verstaan dat [17] "*static analysis of call chains computes a set of chains that is a conservative estimate of the actual chains that may be observed runtime*". Het verschil met dynamic analysis is dat [17] "*Dynamic analysis of call chains constructs a set of chains observed during a particular execution*".

Veel optimalisatietechnieken komen voort uit de compilerbouw. Dynamische analyse speelt hier ook een belangrijke rol [19] "*We utilitze dynamic profile data to determine intra-method code regions that are rarely or never executed, and compile and optimize the code without those regions*". Vanuit de compilerbouw wordt hiermee bereikt dat er een afweging wordt gemaakt tussen de tijdsduur van het compileerproces en hoe vaak een bepaald deel van de code wordt uitgevoerd. Door de delen van de broncode die weinig of niet worden gebruikt tijdens de runtime executie van het programma weg te laten uit het compileerproces en pas te compileren op het moment dat het nodig is, kan tijdswinst worden behaald. Dit is dus het principe van een soort just-in-time compiler. Voor de extractie van dode code kan dit principe ook worden gebruikt waarbij [16] "*We gather the code coverage information by adding instrumentation code to basic blocks*". Op deze manier kan een profiel worden opgebouwd welke informatie geeft over hoe vaak een procedure is uitgevoerd. Er zou gebruik kunnen worden gemaakt van AOP om deze dynamische analyse te bereiken. AOP maakt het mogelijk om zogenaamde "crosscutting concerns" te specificeren. Hiermee wordt bedoeld dat zaken die over het gehele programma invloed hebben met behulp van AOP op eenvoudige wijze kunnen worden beschreven en in de applicatie kan worden opgenomen. Op deze manier zou informatie over de dynamische executie kunnen worden verzameld.

2.4 Evaluatie

Nadat er door middel van analysemethoden een conclusie getrokken is over waar zich dode code bevindt is het vervolgens nodig om deze conclusies te evalueren.

De evaluatie is in feite een losstaand probleem. Het valideren van de correctheid van geëxtraheerde feiten is beschreven in bijvoorbeeld [15]. Hierin wordt reeds aangegeven dat "*Validating that an extractor correctly produces facts at a given level of completeness is in general very challenging.*". De manier waarop de evaluatie plaatsvindt is "*It works by recovering a version of the source program from the extracted factbase and compiling that version*.". Vervolgens wordt de semantische equivalentie bepaald door "*checking that the generated assembly language is identical for the original and recovered programs.*". Deze aanpak is in het geval van de extractie van dode code niet haalbaar. Het assembly-programma dat zou ontstaat na het verwijderen van de dode code uit de broncode is (in veel gevallen) niet hetzelfde als het assembly-programma dat ontstaat uit de originele

broncode. Daarnaast is het zo dat er voor de analyse van dode code op het niveau van methodes lang niet alle informatie uit de broncode nodig is en dus met de geëxtraheerde feiten het originele programma niet kan worden hersteld. Er wordt in [15] ook gesteld dat "*the equality of semantics is undecidable*". Het is dus nooit met zekerheid te stellen dat het verwijderen van dode code niet leidt tot een semantische verandering van het programma. Hieraan ligt dezelfde redenering ten grondslag als bij het Halting Problem. Het aantal toestanden waarin een applicatie zich kan bevinden is in veel gevallen onnoemelijk groot als gevolg van de mogelijke invoer naar een applicatie. Hiermee is ook meteen de vierde deelvraag van het afstudeeronderzoek beantwoord.

Toch zijn er wel inschattingen te doen over de legitimiteit van conclusies over dode code. Er zou bijvoorbeeld gebruik kunnen worden gemaakt van metrieken over de broncode om de plekken waar zich dode code kan bevinden te lokaliseren. Probleem hierbij is alleen dat men eerst inzicht moet hebben in de correlaties tussen de plekken waar dode code zich bevindt en de metrieken die op de broncode kunnen worden uitgevoerd. Voor het bepalen van de plekken waar zich dode code kan bevinden is een interview met de actieve programmeurs werkend aan een onderzochte applicatie waarschijnlijk het meest waardevol.

Een gerelateerd evaluatieprobleem is om te bepalen in hoeverre de technieken voor dode code extractie werken en hoeveel dode code wordt verwijderd. Aangezien het afstudeeronderzoek tot doel heeft dode code te herkennen is het op het moment niet duidelijk waar de dode code zich in een te analyseren applicatie bevindt. Hierdoor is niet te bepalen hoeveel van de dode code ook daadwerkelijk wordt herkend. Over dit onderwerp was weinig literatuur te vinden. Er worden in enkele artikelen wel uitspraken gedaan over de verhoudingen van gevonden dode code ten opzichte van de orginele code van enkele opensource pakketten. Met deze resultaten zou het te ontwikkelen algoritme kunnen worden vergeleken om een indruk van de werking te krijgen. Toch is deze vorm van evaluatie niet zuiver aangezien de resultaten uit artikelen niet gegarandeerd vrij zijn van false positives. In [8] wordt hier specifiek op gewezen. Een algemeen probleem bij het vergelijken van deze resultaten is het ontbreken van aannames waaronder de analyse plaats vindt.

Een belangrijke vorm van evaluatie is het onderbouwen van de gepresenteerde dode code analyse en het aangeven van de manieren waarop false positives worden tegengegaan en de manier waarop false positives zouden kunnen ontstaan als gevolg van de dode code analyse.

Een manier die in ieder geval kan worden toegepast om inzicht te krijgen in de effectiviteit van de analysemethode is om deze uit te voeren op een vooraf geconstrueerd voorbeeldprogramma. Door het voorbeeldprogramma beperkt te houden wat betreft grootte kan de dode code met de hand worden bepaald en kan de effectiviteit worden gemeten. Een risico welke hieraan is gekoppeld is dat er een bias kan ontstaan door de constructies die binnen het voorbeeldprogramma worden gebruikt en de constructie van het algoritme.

Met dit onderwerp wordt de bespreking van de achtergrond en context besloten. In het volgende hoofdstuk zal worden ingegaan op het plan van aanpak.

3 Plan van Aanpak

Voor aanvang van het afstudeeronderzoek is er tijd beschikbaar gesteld voor het uitvoeren van een literatuurstudie. Hiermee is een eerste indruk verkregen over de omvang en complexiteit van het probleem van dode code analyse. Vanuit de bevindingen van de literatuurstudie is een plan van aanpak opgesteld om het onderzoek te kunnen uitvoeren. Aan het begin van het onderzoek bleek reeds snel dat de gestelde doelstellingen in het plan van aanpak niet binnen de termijn van het afstudeeronderzoek haalbaar waren. Op basis van die constatering is ook een nieuw plan opgesteld en is de planning veranderd. Het nieuwe plan zal in dit hoofdstuk worden besproken.

Er is gekozen voor het beperken van de dode code analyse tot Java broncode. Al snel in het project bleek dat het uitzoeken van de syntax en semantiek van deze taal op de punten die van belang zijn voor de analyse een aanzienlijke tijd in beslag zouden nemen. Java heeft voldoende uitdagingen in de vorm van bijvoorbeeld polymorfisme waardoor de onderzoeksvraag ondanks deze beperking kan worden beantwoord.

De eerste stap binnen het project is het vormen van een algeheel beeld van de problematiek en het bepalen van de te beantwoorden vragen. Er is voor gekozen om niet alleen de nagestreefde dode code analyse te beschrijven maar om ook een algemener beeld te geven van hoe deze zou kunnen worden toegepast binnen de Meta-omgeving. De reden hiervan is om een duidelijke toepassing te laten zien en daarnaast om bijkomende problemen (zoals het bepalen van entry-points) te kunnen bespreken.

De tweede stap zal zijn om te achterhalen wat de eigenschappen van Java zijn waarmee rekening moet worden gehouden en de invloed van deze eigenschappen op de te realiseren dode code analyse. Zo zal bijvoorbeeld duidelijk moeten worden op welke manier false positives en false negatives kunnen ontstaan.

Vanuit de ervaringen opgedaan bij de Master Software Engineering is ervoor gekozen om na de bovenstaande twee stappen het onderzoek in iteratieve stappen te verdelen. Zo kan gericht naar een eindresultaat worden gewerkt en zal er na iedere iteratie een werkend product zijn. De te volgen strategie bij de praktische implementatie van de oplossing voor dode code analyse zal zijn om zo klein mogelijk te beginnen en om vervolgens zo snel mogelijk een resultaat te hebben. Volgens dit zogenaamde “grote stappen snel thuis” principe is het mogelijk om snel te kunnen achterhalen wat de mogelijke punten van verbetering zijn en is het mogelijk om te constateren op welke punten de theorie mogelijk tekort schiet.

In de eerste iteratie zal worden gekeken welke tekortkomingen er zijn aan het analysegereedschap dat als uitgangspunt wordt genomen om de dode code analyse te realiseren. Afhankelijk hiervan volgen meerdere iteraties om uiteindelijke de dode code analyse voor de Java broncode te kunnen realiseren. Hiervoor zal het nodig zijn om een representatieformaat op te stellen voor de feiten die nodig zijn vanuit de broncode om de dode code analyse te kunnen uitvoeren. In alle iteraties zal er gebruik worden gemaakt van een opgestelde voorbeeldapplicatie waarvan de dode code vooraf bekend was.

	De volgorde waarin de iteraties worden uitgevoerd staat niet geheel vast. Dit heeft te maken met het feit dat het naar mijn mening verstandig is om vanuit het gewenste resultaat naar het begin toe te werken. Het voordeel hiervan is dat er meteen duidelijkheid is over de informatie die vanuit de broncode nodig is om de uiteindelijke dode code analyse te realiseren. Wanneer andersom wordt gewerkt kan men er halverwege achterkomen dat er informatie is weggevallen bij een eerdere stap die toch nodig blijkt te zijn om de dode code te kunnen detecteren. Het meest rampzalige gevolg hiervan kan zijn dat verschillende iteraties opnieuw moeten worden uitgevoerd.

In de hoofdstukken 5 en 6 zal worden ingegaan op de praktische implementatie van de statische en dynamische analyse en op de onderdelen waaruit deze bestaan.

De uiteindelijke hypothese waarnaar ik probeer toe te werken is:

> ***Dynamische analyse kan worden gebruikt in combinatie met statische analyse om de complexiteit rondom type-bepaling bij statische dode code analyse te beperken. Hiermee wordt de taal-onafhankelijkheid van de analyse verbeterd.***

Hiermee wordt bedoeld dat de dynamische analyse ervoor kan zorgen dat de statische analyse minder zwaar hoeft te zijn wat betreft complexiteit. Het beperken van de complexiteit van de statische analyse is met name interessant voor applicaties die geschreven zijn in verschillende programmeertalen (heterogene systemen) zoals

het geval is bij de Meta-omgeving. Het beperken van de complexiteit zal invloed hebben op de taal-onafhankelijkheid van de analyse en dus ook effect hebben op de realiseerbaarheid van de dode code analyse voor een verscheidenheid aan programmeertalen. Een groot voordeel van een vorm van dode code analyse die toepasbaar is voor verschillende programmeertalen is dat de onderliggende gedachte onafhankelijk van de te analyseren taal hetzelfde is. De aannames waaronder de analyse plaatsvindt zijn in dat geval steeds hetzelfde. Deze luxe vervalt op het moment dat er verschillende tools worden gebruikt voor de dode code analyse.

4 Bespreking hoofdvraag

In dit hoofdstuk zal de algemene oplossingsrichting worden besproken voor het beantwoorden van de hoofdonderzoeksvraag. Dit is een beschrijving van de onderdelen die komen kijken bij het extraheren van dode code. Door de beperkte tijd van het onderzoek is uiteindelijk maar een beperkt deel van de gehele aanpak in detail uitgewerkt en praktisch geïmplementeerd. Hier zal vanaf hoofdstuk 5 op worden ingegaan.

In de eerste paragraaf zal worden stilgestaan bij de nadelige effecten van dode code. De mogelijk vormen van analyse worden vervolgens besproken in de tweede paragraaf waarna in de derde paragraaf wordt ingegaan op de mogelijk combinaties hiervan. In hoofdstuk 4.3.2 wordt stilgestaan bij de analyse zoals deze binnen het onderzoek wordt nagestreefd. De vierde paragraaf wordt gebruikt om aan te geven hoe binnen de Meta-omgeving de dode code analyse vorm kan worden gegeven.

Als laatste wordt kort ingegaan op een manier waarop de resultaten van de dode code analyse kunnen worden gepresenteerd aan een programmeur zodat deze uiteindelijk de beslissing kan nemen of er daadwerkelijk sprake is van dode code. Dit houdt verband met het feit het bepalen of een stuk code dood is in veel gevallen onbeslisbaar is (zie hoofdstuk 2.4)

4.1 Nadelige effecten van dode code

Om te kunnen bepalen waarom een onderzoek naar dode code op het niveau van methodes interessant is en waarom dit praktisch nut heeft, zal in deze paragraaf worden besproken wat de nadelige effecten zijn van het hebben van dode code in een applicatie. In [20] heb ik hier reeds bij stilgestaan en dit hoofdstuk is ook grotendeels gebaseerd op dit paper en de informatie die voor het schrijven van het paper was bestudeerd.

Dode code heeft de volgende vier nadelen:

1. Dode code maakt de begrijpbaarheid van de broncode onnodig moeilijker.
   - Wanneer voor bijvoorbeeld onderhoudswerkzaamheden de code door een programmeur moet worden geanalyseerd zal dode code leiden tot extra werk om de code goed te begrijpen. Dit komt doordat de functie van de code moet worden begrepen en daarnaast de plaats van de code in de algehele applicatie. Het gevolg hiervan is dat er onnodig tijd verloren kan gaan en dat er mogelijk verkeerde conclusies over de werking worden getrokken wanneer slechts de structuur wordt geanalyseerd.

2. Dode code kan vertragend werken op de applicatie door het laden van onnodige functionaliteit
   - Als er bijvoorbeeld objecten worden gecreëerd waarin methoden aanwezig zijn die niet worden gebruikt kan dit tijdsvertraging opleveren en onnodig geheugengebruik.
   - Afhankelijk van de interne representatie van het programma is het mogelijk dat het verwijderen van dode code leidt tot een betere cache strategie. Door het verwijderen van dode code kan de spreiding van de adressen die worden gebruikt voor het uniek identificeren van methodes worden verkleind.

3. Dode code levert onnodige vertraging van het compileerproces
   - Ondanks de vele vorderingen op het gebied van compilerbouw is van een groot gedeelte van de code vooraf niet duidelijk of deze wel of niet zal worden gebruikt. Dit argument wordt wel enigszins afgezwakt door de vooruitgangen die worden geboekt op het gebied van “Just in Time” compilation. Het is bij het gebruik van een JIT compiler alleen wel zo dat deze techniek alleen beschikbaar is voor talen waarbij een bytecode representatie wordt gebruikt [W13] en is dus niet toepasbaar bij talen zoals C. Bij deze talen wordt gewoonweg alles vooraf gecompileerd en onderling gekoppeld om als executable danwel library te functioneren.

4. Dode code kan gevaarlijk zijn doordat deze mogelijk niet structureel wordt onderhouden
   - Een stuk code welke niet meer wordt gebruikt binnen een applicatie en welke niet wordt verwijderd kan wat betreft onderhoud achter gaan lopen doordat niemand hiermee meer in aanraking komt. Op het moment dat deze code nog wel een functionaliteit aanbiedt zou het kunnen dat na lange tijd de functionaliteit weer in gebruik wordt genomen. Het is de vraag of de code op dat moment nog voldoet aan de eisen van het ontwikkelende bedrijf en of dit degene opvalt die de methode gaat gebruiken wanneer deze uitgaat van de interface specificatie als bron van informatie.

Zoals uit bovenstaande opsomming naar voren komt zijn er veel redenen te bedenken om dode code uit een applicatie te verwijderen. In de volgende paragraaf zal worden ingegaan op de manier waarop deze voordelen kunnen worden behaald door het bespreken van mogelijke vormen van dode code analyse.

4.2 Verschil in analyse-mogelijkheden van dode code

De extractie van dode code kan vanuit drie perspectieven worden benaderd. De eerste mogelijkheid is het gebruik van een analyse van het build-proces (build-time). De tweede mogelijkheid is om een statische analyse-techniek te gebruiken (compile-time) en de derde mogelijkheid is om de analyse te benaderen vanuit een dynamisch oogpunt (runtime).

Bij build-time analyse wordt het compileren van de bronbestanden en het combineren van de uitvoer van compilatie bekeken en gebruikt om feiten te extraheren. Het bleek al snel dat deze vorm niet de benodigde informatie zou opleveren voor de detectie van dode code. Er is een losse bespreking van de analyse van het build proces opgenomen in bijlage E.

In het geval van statische analyse wordt de broncode als invoer gebruikt en zal men hieruit feiten extraheren. Bij dynamische analyse wordt het programma daadwerkelijk geëxecuteerd en gaat men over de runtime eigenschappen redeneren. In deze paragraaf zal stil worden gestaan bij de voor- en nadelen van deze beide aanpakken. Dit is een belangrijke stap voor het uiteindelijk beantwoorden van de onderzoeksvraag en de onderbouwing van waarom een combinatie van deze vormen een goede strategie is voor het beperken van de complexiteit en het ondersteunen van de taal-onafhankelijkheid.

Voordelen statische analyse
In het geval van statische analyse is er geen afhankelijkheid van runtime eigenschappen [10]. Dit houdt in dat de feiten die men statisch extraheert ook daadwerkelijk waar zijn en er sprake is van soundness[W14]. Dit komt doordat er geen afhankelijkheid is van een bepaald scenario of toestand waarin de applicatie zich bevindt. In [10] wordt het effect van soundness omschreven als *"Soundness guarantees that analysis results are an accurate description of the program's behavior, no matter on what inputs or in what environment the program is run*". Het gevolg is wel dat de conclusies die men kan trekken beperkt zijn door het ontbreken van informatie over runtime gedrag. Type-analyse is als gevolg hiervan statisch niet nauwkeurig uit te voeren aangezien runtime pas daadwerkelijk wordt bepaald naar welk type een variabele verwijst. Het bepalen van het type van een variabele is van groot belang voor dode code analyse om te kunnen achterhalen van welk type een methode wordt uitgevoerd. Het verschil zit hem hier in het gedeclareerde type (compile-time) van een variabele en het type waarnaar de variabele uiteindelijk verwijst (runtime).

Een minder belangrijk voordeel van statische analyse is dat de invoer naar de analyse volledig syntactisch is vastgelegd. De opbouw van de broncode waarvanuit de feiten worden geëxtraheerd is volledig bekend waardoor betrouwbare en herhaalbare analyses mogelijk zijn. Dit voordeel is eigenlijk ook meteen een nadeel aangezien een statische analyse dus in veel gevallen sterk gebonden is aan een specifieke programmeertaal. Hierdoor wordt het statisch analyseren van een applicatie welke uit meerdere programmeertalen is opgebouwd, zoals de Meta-omgeving, bemoeilijkt.

Nadelen statische analyse
In veel programmeertalen is het mogelijk om gebruik te maken van pointers of constructies met eenzelfde soort karakter, zoals polymorfisme bij Java en andere objectgeoriënteerde talen. Deze constructies maken het mogelijk om te verwijzen naar bijvoorbeeld een methode zonder dat direct uit de broncode valt te extraheren welke gedeclareerde methode wordt aangeroepen. De referenties voor de pointers bij talen als C en de specifiek aan te roepen methodes bij het gebruik van methode-overschrijving bij Java, worden runtime bepaald waardoor deze in veel gevallen niet statisch zijn te bepalen [10]. Het gevolg is dat de statische analyse beperkt wordt in besluitvorming met betrekking tot de type-analyse door de onzekerheid die optreedt volgend uit bovenstaand nadeel.

Een ander nadeel, wat met bovenstaande samenhangt, is de mogelijkheid tot het dynamisch laden van code en het compileren en executeren van code. Een voorbeeld is het gebruik van het dynamisch laden van klassen binnen Java met behulp van reflectie. Java biedt hiermee de mogelijkheid om tijdens het executeren van een applicatie functionaliteit uit classes te laden. Dit geeft een zelfde soort problematiek als bij het gebruik van pointers. Dit komt door het feit dat het kan lijken dat een losstaand klasse niet wordt gebruikt terwijl deze mogelijk dynamisch wordt geladen. Hierdoor is het, net als in het geval van het gebruik van pointers, niet met zekerheid te bepalen of er nou daadwerkelijk sprake is van dode code.

Voordelen dynamische analyse
In het geval van dynamische analyse is het wel duidelijk waar pointers naar verwijzen en welke klassen er uiteindelijk binnen de applicatie worden geladen. Met dynamische analyse is het namelijk mogelijk om preciezer te zijn bij het extraheren van feiten[10]. Met preciezer wordt in dit geval bedoeld dat men over daadwerkelijke runtime informatie beschikt. Het is belangrijk om op te merken dat precisie in dit geval niet gerelateerd is aan correctheid aangezien het trekken van conclusies aan onzekerheid gebonden is doordat er geen sprake is van soundness bij dynamische analyse.
Een ander voordeel is dat dynamische analyse in veel gevallen sneller is dan statische analyse [10]. Dit komt doordat bij statische analyse in veel gevallen meer redeneerwerk (interpreteren van de broncode) komt kijken dan bij dynamische analyse aangezien men bij deze laatste soort eenvoudigweg de executie kan waarnemen.

Nadelen dynamische analyse
Het belangrijkste nadeel van dynamische analyse is dat het runtime gedrag van de te analyseren applicatie afhankelijk is van de manier waarop de applicatie wordt gebruikt. Het is bijvoorbeeld zeer goed mogelijk dat een bepaalde methode niet wordt aangeroepen tijdens een dynamische analyse maar dat deze toch geen dode code is. Dit komt doordat in een ander scenario deze methode wel een rol zou kunnen spelen als gevolg van conditionele selectie-mechanismen binnen de applicatie. Als gevolg hiervan kunnen er dus geen harde conclusies worden getrokken zoals dat bij statische analyse wel mogelijk is [10]. Het moge duidelijk zijn dat dynamische analyse nooit sound kan zijn zoals dat wel bij statische analyse geldt.

Een ander nadeel van dynamische analyse is dat het lastig kan zijn om de dynamische executie te koppelen aan specifieke plekken in de broncode. Een oorzaak hiervan is de compiler die optimalisaties kan toepassen op de code waardoor de gelijkenis met de broncode kan wegvallen. Hierdoor is het moeilijk om de feitelijke dode code te verwijderen. Een voorbeeld is de omzetting van C++ code naar assembler waarbij de gelijkenis ver te zoeken is. Maar in het geval van Java byte code is de gelijkenis met de oorspronkelijke broncode weer groot. Daarnaast is de ernst van dit nadeel zeer afhankelijk van de gedetailleerdheid waarmee dode code moet worden gezocht. In het geval van dode code op het gebied van methodes valt dit nadeel te voorkomen door gewoonweg iedere methode een uniek bericht te laten afdrukken wanneer de methode wordt aangeroepen.

Samenvattend komt het erop neer dat in het geval van een dynamische aanpak de kans groot is dat er stukken code als dood worden geïdentificeerd terwijl ze dit eigenlijk niet zijn doordat de scenario-afhankelijkheid een rol speelt. Bij statische analyse kunnen, als gevolg van de nadelen met betrekking tot het gebruik van structuren zoals pointers, polymorfisme en het dynamisch laden van klassen, maar beperkt uitspraken worden gedaan over waar zich dode code bevindt. Er moet hier rekening worden gehouden met de conservatieve instelling die men moet aanhouden om false positives (delen die eigenlijk niet dood zijn) in de dode code set te voorkomen.

4.3 Mogelijke combinaties van statische en dynamische analyse

Uit de literatuurstudie bleek dat statische analyse en dynamische analyse in geen van de bestudeerde gevallen werd gecombineerd voor de analyse van dode code. Zoals aangegeven in hoofdstuk 2 is er wel een artikel [17] waarin de combinatie van statische en dynamische analyse wordt besproken bij het opstellen van een call chain. Afhankelijk van de onderliggende gedachte en aannames waarmee deze is geconstrueerd zou deze aanpak kunnen worden gebruikt voor dode code analyse. Uit de literatuurstudie bleek dat toch in veel gevallen gewerkt wordt met statische analyse. De nagestreefde oplossing voor het beantwoorden van de onderzoeksvraag zal zich juist richten op de combinatie tussen statische en dynamische analyse om zo de voordelen van beide vormen te kunnen combineren en dode code te kunnen detecteren.

Om te kunnen komen tot een strategie voor het verwijderen van dode code door middel van een combinatie van statische en dynamische analyse is het allereerst nodig om te bepalen op welke manieren deze twee aanpakken kunnen worden gecombineerd. Hierbij zijn de onderstaande drie vormen te onderscheiden:

- *Filtering van broncode door middel van dynamische analyse voordat deze aan de statische analyse wordt onderworpen.*
- *Filtering van de resultaten die voortkomen uit de statische analyse met behulp van dynamische analyse.*
- *Zelfstandige toepassing van dynamische analyse en statische analyse waarbij achteraf de vereniging wordt genomen van de resultaten van beide analyses.*

In de volgende paragrafen zal worden ingegaan op enkele mogelijke invullingen van elk van deze combinaties. In paragraaf 4.3.2 zal in detail worden ingegaan op de tweede combinatie aangezien deze naar mijn mening de

enige is die een raamwerk biedt waarmee op structurele wijze met de onzekerheden op het gebied van type-analyse kan worden omgegaan. Daarnaast is bij deze combinatie de belangrijkste rol weggelegd voor de statische analyse. Dit strookt met de vele artikelen over statische analyse en de gereedschappen die op basis hiervan zijn gebouwd.

4.3.1 Dynamische analyse voor statische analyse

Een mogelijke invulling van deze combinatievorm is het gebruik van de dynamische analyse om een indruk te krijgen van de entry-points naar de applicatie. Afhankelijk van de manier waarop de statische analyse is vormgegeven is het mogelijk om iedere methode als entry-point te beschouwen of slechts de daadwerkelijke beginpunten van de applicatie (hierover meer in bijlage D). Op het moment dat iedere methode als entry-point kan worden beschouwd geeft het vooraf uitvoeren van de dynamische analyse een goede aanvulling op de entry-points die benodigd zijn voor de statische analyse. Als gevolg van de scenario-afhankelijkheid van dynamische analyse blijft er onzekerheid aanwezig over het feit of men daadwerkelijk alle entry-points heeft verzameld.

Het vooraf uitvoeren van dynamische analyse kan eigenlijk de code slechts opdelen in twee sets. De eerste set zijn de methodes die daadwerkelijk zijn uitgevoerd tijdens de executie van de applicatie en de tweede set zijn de methodes waarover geen zekerheid is of deze wel of niet worden gebruikt. Het gevolg hiervan is dat het vooraf uitvoeren van dynamische analyse alleen de mogelijkheid biedt om de eerder genoemde entry-points aan te vullen en voor de rest geen substantiële bijdrage kan leveren. In de volgende paragraaf zal blijken dat het uitvoeren van dynamische analyse na de statische analyse meer inzicht zal geven in de code.

4.3.2 Statische analyse voor dynamische analyse

Zoals besproken in paragraaf 4.2 is een voordeel van statische analyse dat deze aanpak sound is. Dit komt doordat bij statische analyse de geëxtraheerde feiten van een applicatie niet afhankelijk zijn van scenario's over hoe de applicatie wordt gebruikt. Een voordeel van dynamische analyse is juist weer dat er geen onzekerheid is over waar een referentie binnen een programma naar verwijst op het moment dat deze wordt gebruikt tijdens een bepaald scenario.

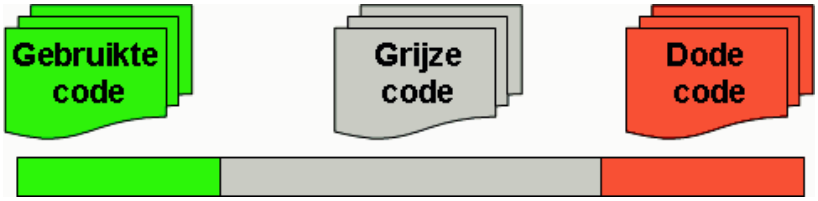
Het type-analyseprobleem dat ontstaat binnen statische analyse door het niet kunnen bepalen van referenties is reeds veelvuldig onderzocht. Uit de literatuurstudie bleek dat er reeds meerdere oplossingsrichtingen [18](points-to analysis) voor type-analyse zijn waarbij de meeste voornamelijk theoretisch zijn.

Een mogelijke invulling voor statische dode code extractie is het gebruik van slicing. Slicing is een geavanceerde techniek waarbij wordt gekeken op welke plekken een variabele wordt beïnvloed om zo te kunnen achterhalen wat de waarde ervan is[9][12]. Als gevolg hiervan kan naar mijn mening ook worden geredeneerd over het type waarnaar een variabele kan verwijzen. In feite kan er dus voor iedere variabele een set worden gecreëerd waarin de elementen staan die de variabele beïnvloeden. Op basis hiervan zou het mogelijk moeten zijn om het type waarnaar de variabele kan verwijzen te bepalen. In het gunstigste geval bevat deze set met mogelijke types waarnaar wordt gerefereerd slechts één element maar op het moment dat er gebruik wordt gemaakt van principes zoals methode-overschrijving bevat deze set meerdere elementen. Dit kan bijvoorbeeld komen door het gebruik van een abstract factory waardoor er onzekerheid ontstaat over waarnaar de referentie voortkomende uit de factory verwijst. Een voordeel van slicing is dat men ook gebruik kan maken van het feit dat men informatie heeft over argumenten die aan de abstract factory worden meegegeven. Door vervolgens de selectielogica binnen de factory te sturen op basis van deze argumenten kan men een nauwere set krijgen van mogelijke verwijzingen. Statisch kan men dus op deze manier de mogelijke referenties waarnaar een variabele verwijst minimaliseren. Een zeer groot probleem dat voortvloeit uit deze toepassing is dat de complexiteit van de statische analyse als gevolg van bijvoorbeeld het kunnen interpreteren van selectielogica toeneemt en de taal-afhankelijkheid steeds groter wordt op het moment dat er met meer zekerheid moet worden geredeneerd.

Vanuit de eerder besproken problematiek volgend uit het type-analyse probleem en vanuit het feit dat de analyse voor meerdere programmeertalen toepasbaar moet zijn volgt mijn opzet. In deze benadering wordt de onzekerheid vastgelegd via statische analyse en wordt vervolgens met behulp van dynamische analyse deze onzekerheid weer ingeperkt. Op deze manier worden de voordelen van beide vormen van analyse gecombineerd en de nadelen beperkt. Het feit dat de nadelen worden beperkt komt doordat de scenario-afhankelijkheid bij dynamische analyse gedeeltelijk wordt gecompenseerd door het vooraf uitvoeren van de statische analyse. Daarnaast kan de onzekerheid over het type waarop een methode wordt aangeroepen binnen statische analyse worden beperkt door het uitvoeren van een gerichte dynamische analyse op de gedeeltes waarvan men niet zeker is.

Om te kunnen beantwoorden op welke manier dynamische analyse een bijdrage kan leveren aan de analyse van dode code in combinatie met statische analyse is het allereerst benodigd om te achterhalen hoe met de onzekerheid binnen statische analyse kan worden omgesprongen. Voor het achterhalen van de onzekerheid binnen de statische analyse is het nodig om de statische analyse twee vragen te laten beantwoorden; “Wat is dode code binnen de applicatie?”, “Wat is werkende code binnen de applicatie?”. Het is hierbij van belang dat de vraag “Wat is dode code?” op een conservatieve manier wordt beantwoord om false positives tegen te gaan. Met conservatief wordt bedoeld dat als er onduidelijkheid is over het feit of een methode wordt aangeroepen dat deze dan als aangeroepen wordt beschouwd en dus niet als dood wordt geregistreerd. De beantwoording van de vraag “Wat is werkende code?” moet op een niet conservatieve manier worden beantwoord. Dit betekent dat op het moment dat er onduidelijkheid is over welke methode wordt aangeroepen omdat er onzekerheid is over het type dat deze niet als werkend wordt beschouwd.

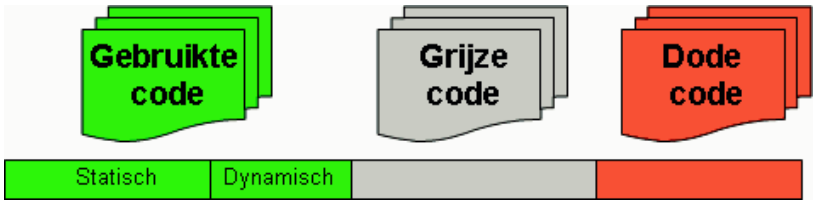
In feite kan men de antwoorden op de bovenstaande vragen zien als sets met code waarvan men de status weet. Met behulp van figuur 1 zal dit principe worden verduidelijkt.



Figuur 1 Statische opdeling van code

De meest rechter set (rood) bevat de methodes die het antwoord zijn op de vraag “Wat is dode code?” en de meest linker set (groen) bevat het antwoord op de vraag “Wat is werkende code?”. Het middelste gebied (grijs) bevat de methodes waarvan men niet zeker is of deze werkend zijn of als dode code kunnen worden beschouwd. Aangezien, zoals eerder genoemd, deze onzekerheid voortkomt uit het feit dat het statisch niet altijd mogelijk is om te bepalen waarnaar een referentie wijst (pointers, polymorfisme, enz.) kan dynamische analyse hier een oplossing bieden.

Om dynamische analyse te kunnen toepassen op de grijze set is het nodig om scenario’s op te stellen die specifiek gericht zijn op delen van de applicatie waarin de methodes worden gebruikt die in de grijze set zitten. Door vervolgens op basis van deze scenario’s de applicatie runtime te analyseren kan worden vastgesteld of er methodes uit de grijze set zijn die worden aangeroepen. Deze kunnen dan worden opgenomen in de gebruikte code set. De dynamische analyse zorgt er dus voor dat het aantal methodes in de grijze set zal afnemen. Als gevolg hiervan zal er meer zekerheid ontstaan over de status (dood/werkend) van de methodes die in de applicatie aanwezig zijn.



Figuur 2 Dynamische opdeling van code

In figuur 2 is de situatie geschetst waarbij de omvang van de grijze set afneemt en deze elementen zich naar de gebruikte code set verplaatsen. In de set met gebruikte code is het mogelijk om een onderscheid te maken tussen de code die statisch als gebruikt is bestempeld en die dynamisch als gebruikt is bestempeld. Het voordeel hiervan is dat op het moment dat men juist niet geïnteresseerd is in de dode code maar juist in de werkende code (reachability analysis) het dynamisch geëvalueerde gedeelte van de gebruikte code gegarandeerd geen false positives bevat. De code die statisch als gebruikt is bestempeld kan namelijk wel false positives bevatten vanuit het reachability oogpunt doordat er bij de statische analyse vanuit wordt gegaan dat alle takken van selectielogica worden doorlopen. Een methode welke alleen wordt gebruikt in een tak welke nooit wordt doorlopen zal bij de statische analyse toch in de gebruikte code set komen. Dit is een voorzorgsmaatregel voor het optreden van de onbeslisbaarheid bij het bepalen van de daadwerkelijk uitgevoerde takken en dus om te voorkomen dat gebruikte code als dood wordt bestempeld.

Een belangrijke constatering is dat het mogelijk is dat zich in het statisch geanalyseerde gedeelte van de set met gebruikte code nog dode code kan bevinden. De reden hiervan is dat er vanuit wordt gegaan bij de in dit onderzoek toe te passen statische analyse dat alle mogelijke paden worden doorlopen. Er wordt dus geen rekening gehouden met het feit dat een bepaalde tak nooit bereikt kan worden. Hiervoor is gekozen om false positives in de dode code set te voorkomen. De takken die worden gevolgd binnen een programma kunnen afhankelijk zijn van scenario's en er kan dus onterecht worden geconcludeerd dat een pad niet wordt gevolgd. Daarnaast is voor het statisch evalueren van selectielogica van een te onderzoeken applicatie veel extra functionaliteit benodigd in de analyse en er treedt taal-afhankelijkheid op.

Op het moment dat er geen zinvolle nieuwe scenario's voor de dynamische analyse zijn te bedenken kan men de methodes die vanuit de grijze set naar de werkende set zijn verhuisd beschouwen als entry-points naar de applicatie (zie bijlage D). Het gevolg hiervan is dat het mogelijk is dat vanuit zo'n nieuw toegevoegd entry-point er door middel van de statische analyse kan worden bepaald dat een methode in gebruik is die zich na de dynamische analyse nog steeds in de grijze set bevond. De reden hiervan is dat het kan voorkomen dat het gebruikte scenario voor de dynamische analyse een bepaalde selectietak vanuit de nieuw gevonden entry-point heeft overgeslagen terwijl dit met de statische analyse wel kan worden geconstateerd. Dat deze conclusie niet kon worden getrokken voor de dynamische analyse kan ontstaan als gevolg van polymorfisme. Het kan zijn dat ten tijde van de statische analyse nog geen zekerheid was of de later met dynamische analyse gevonden entry-point werd aangeroepen. Het onderstaande voorbeeld zal worden gebruikt om dit te verhelderen.

```
public class A {
    public int getInt() {
      ...
    }
    ...
}

public class B extends A {
    public int getInt() {
      ...
       if(Math.random() == 0.5) {
          int a = super.getInt()
          ...
       }
    }
    ...
}

public class Analysis {
    public void static main(String [] args) {
        A test;
        ...
        test.getInt();
    }
}
```

Voorbeeld Statische conclusie op dynamische entry-point

In bovenstaand voorbeeld is te zien dat er twee klassen aanwezig zijn die beide een methode `getInt()` hebben en waarbij klasse `B` overerft van klasse `A`. In de main methode van klasse `Analysis` wordt een variabele van het type `A` aangemaakt met de naam test. Deze wordt vervolgens in de loop van het programma geïnitialiseerd door bijvoorbeeld gebruik te maken van een Abstract Factory en uiteindelijk wordt er de methode `getInt()` op aangeroepen. Bij deze aanroep treedt de onzekerheid op dat het niet duidelijk is van welke van de klassen `A` en `B` de methode wordt aangeroepen omdat de variabele `test` naar beide types kan wijzen. Deze twee methodes worden als gevolg hiervan opgenomen in de grijze set om vervolgens dynamisch te worden geanalyseerd. Stel dat tijdens de dynamische analyse wordt geconstateerd dat de `getInt()` methode van klasse `B` wordt aangeroepen en de dynamische methode geen uitsluitsel geeft over de aanroep naar de `getInt()` van klasse `A` omdat `Math.random()` tijdens de dynamische analyse toevalligerwijs geen 0.5 oplevert. De dynamische analyse levert dus een nieuw entry-point op welke de `getInt()` methode van klasse `B` is. Wanneer nu opnieuw de statische analyse wordt uitgevoerd is er binnen de statische analyse zekerheid over het feit dat `getInt()` van klasse `B` wordt aangeroepen aangezien deze als nieuw entry-point is toegevoegd. Met behulp van de statische analyse kan vervolgens worden geconcludeerd dat vanuit het nieuwe entry-point de aanroep

`super.getInt()` wordt aangeroepen. Gegeven de semantische interpretatie van `super` binnen Java kan dus alsnog statisch worden bepaald dat de `getInt()` van klasse `A` ook wordt aangeroepen. Deze methode zal dus na de tweede iteratie van de statische analyse niet meer in de grijze set zijn opgenomen maar zich in de set werkende code bevinden.

Het toevoegen van een entry-point kan nooit nieuwe elementen voor de dode code set opleveren aangezien er als gevolg van het toevoegen van een entry-point en het rekening houden met de onzekerheid alleen maar meer gebruikte methodes kunnen ontstaan. Het toevoegen van een entry-point als gevolg van de dynamische analyse geeft dus alleen meer zekerheid over gebruikte methodes en geeft meer inzicht in de grijze set vanuit de statische analyse. Hier bevindt zich dus de koppeling.

Op het moment dat iteratief meerdere keren de statische en dynamische analyse zijn uitgevoerd neemt de grijze set in omvang af als gevolg van bovenstaande constatering. De grootste invloed op het verkleinen van de grijze set wordt uiteindelijk bepaald door de kwaliteit van de scenario's die tijdens de dynamische analyse worden gebruikt. De statische analyse werkt immers volgens een vaste strategie en is afhankelijk van de nieuw gevonden entry-points.

De kans dat de methodes die in de grijze set achterblijven dode code is wordt als gevolg van de iteratieve toepassing van statische en dynamische analyse steeds groter naarmate de grijze set kleiner wordt. Het blijft een probleem om te bewijzen dat deze overgebleven methode-aanroepen in de grijze set ook daadwerkelijk niet gebruikt worden. Zoals in [15] is aangegeven kan men nooit met volledige zekerheid garanderen dat het verwijderen van dode code de applicatie semantisch onveranderd laat. Het bepalen of een methode niet wordt gebruikt blijft onbeslisbaar. Uiteindelijk zal toch de programmeur moeten beslissen of er sprake is van dode code. Om dit mogelijk te maken is het benodigd om een geschikte representatie van de dode code te hebben welke op eenvoudige wijze aan de programmeur de benodigde informatie kan overbrengen over de geconstateerde dode code. Hierop zal in paragraaf 4.5 worden ingegaan.

De dode code analyse zoals in deze paragraaf is besproken is de vorm van analyse die binnen dit onderzoek als uitgangspunt is genomen om de onderzoeksvraag te beantwoorden. Deze combinatie van statische en dynamische analyse zal in de verdere hoofdstukken van deze scriptie uitgebreid worden behandeld.

4.3.3 Onafhankelijke toepassing statische- en dynamische analyse

De derde manier waarop statische en dynamische analyse kan worden gecombineerd is door ze onafhankelijk van elkaar toe te passen. Bij de onafhankelijke toepassing van statische en dynamische analyse heeft men in feite weinig winst. De voor- en nadelen die bij beide vormen van analyse een rol spelen zullen elkaar niet compenseren doordat de resultaten van de analyses geen invloed op elkaar hebben. De false positives die ontstaan bij de dynamische analyse en de false negatives die ontstaan bij de statische analyse worden in feite bij elkaar opgeteld. In dit geval zou het dus beter zijn om de resultaten van slechts één van deze analysevormen te gebruiken om het aantal incorrecte resultaten niet te vergroten. Uiteraard is het wel zo dat een slimmere vorm van combinatie van de resultaten in de vorm van bijvoorbeeld intersectie een beter resultaat zal opleveren. Toch blijft het zo dat het sturen van de dynamische analyse met het resultaat van de statische analyse meer mogelijkheden biedt en tenminste hetzelfde kan bereiken als een onafhankelijke toepassing. Dit is ook de reden waarom de onafhankelijke toepassing niet verder is onderzocht en gekozen is voor de aanpak besproken in paragraaf 4.3.2.

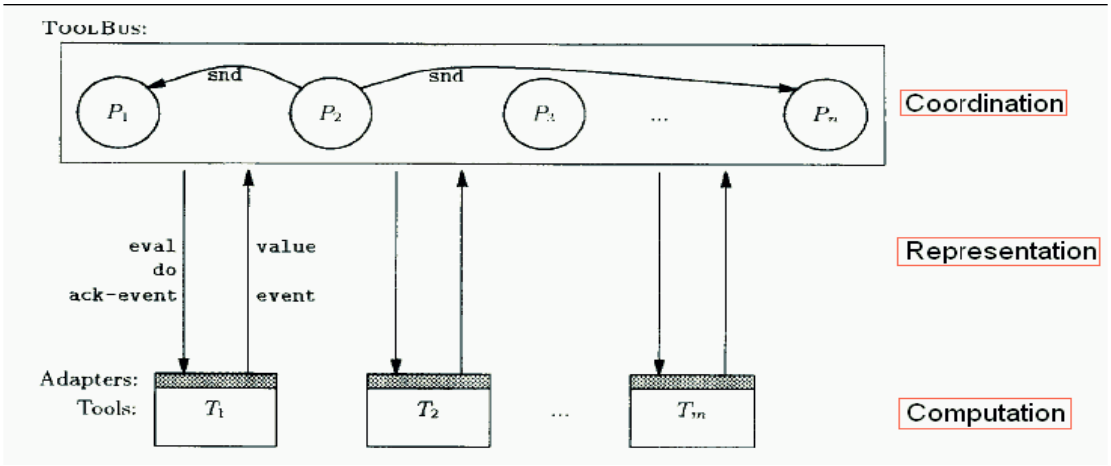
4.4 Bepaling van dode code in Meta-omgeving

In de vorige paragrafen is ingegaan op de analysemethodes die mogelijk zijn en welke zal worden nagestreefd voor beantwoording van de onderzoeksvraag. In de komende twee paragrafen zal de manier worden besproken waarmee in de Meta-omgeving kan worden gezocht naar dode code, gebruikmakende van de analyse-methode zoals besproken in 4.3.2. Deze bespreking is een antwoord op de eerste deelvraag en zal leiden tot het vervullen van de eerste doelstelling.

4.4.1 Bepaling van entry-points naar componenten

De bepaling waar zich dode code bevindt start bij het achterhalen van de entry-points naar de te onderzoeken applicatie. Hiermee worden methoden bedoeld die als startpunt dienen voor de applicatie. Een voorbeeld hiervan is een `main` methode. Vanuit de entry-points kan vervolgens worden geredeneerd over de methodes die vanuit deze startpunten wel of juist niet worden aangeroepen. In deze paragraaf zal worden besproken op welke manier de entry-points van de Meta-omgeving kunnen worden geëxtraheerd.

In hoofdstuk 2 is reeds aan bod gekomen dat alle componenten van de Meta-omgeving met elkaar verbonden zijn via de ToolBus. Aangezien de ToolBus hierdoor kennis heeft over alle aangesloten componenten is dit een belangrijk uitgangspunt bij de bepaling van de entry-points. Om het geheel te verduidelijken zal kort worden ingegaan op de architectuur van de ToolBus om zo te kunnen aangeven op basis waarvan de entry-points kunnen worden bepaald.



Figuur 3 ToolBus Architectuur

Zoals in figuur 3 [2] is te zien kan de ToolBus worden opgedeeld in drie niveaus. Het computationele gedeelte wordt vertegenwoordigd door de componenten die met de ToolBus zijn verbonden. Om het mogelijk te maken voor de tools om te kunnen communiceren met de ToolBus is er een representatielaag aanwezig. De coördinatielaag is de belangrijkste laag en zorgt ervoor dat door middel van ToolBus scripts een aanvraag van een component voor informatie terecht komt bij het component die deze aanvraag kan afhandelen.

Het coördinatiemechanisme in de ToolBus is het gedeelte dat onafhankelijk is van de implementatietaal van een aangesloten component. Op deze manier wordt ervoor gezorgd dat componenten geschreven in verschillende programmeertalen met elkaar kunnen samenwerken. De bibliotheek die benodigd is voor de representatielaag is wel afhankelijk van de taal waarin een component is geschreven. Dit is ook het punt waarop van taal-onafhankelijk naar taal-afhankelijk wordt overgestapt.

⚠	Aangezien het coördinatiemechanisme tot doel heeft de communicatie tussen de componenten mogelijk te maken heeft deze laag informatie nodig over de aangeboden functionaliteit van ieder component. Dit houdt dus in dat de functies van een component die kunnen worden aangeroepen vanuit ToolBus scripts gespecificeerd moeten zijn. Deze specificatie van de functionaliteit van een component gebeurt ook door middel van een ToolBus script. De coördinatielaag, met bijbehorende ToolBus scripts, is dus de plek op basis waarvan de entry-points naar ieder afzonderlijk component kunnen worden vastgesteld.

Er kunnen impliciet twee soorten ToolBus scripts worden onderscheiden. Dit zijn de zogenaamde Idef scripts en de Tb scripts. Ieder van deze scripts bevat informatie welke kan worden gebruikt om de entry-points naar de componenten te bepalen. In de volgende twee tabellen zijn de mogelijkheden binnen deze scripts beschreven die van belang zijn voor het achterhalen van de entry-points.

Soort	**IDEF**		
Omschrijving	Een ”idef” script beschrijft de interface die een component aan de ToolBus aanbiedt. Op basis hiervan kunnen berichten heen en weer worden verstuurd om functionaliteit van het aangesloten component te gebruiken.		
Communicatie	Communicatie vanuit ToolBus naar Component	Commando	Omschrijving
		snd-eval	Aanspreken van een functie die vanuit het component wordt aangeboden en waarop een reactie vanuit het component richting de ToolBus wordt verwacht.

		snd-do	Aanspreken van een functie die vanuit het component wordt aangeboden en waarop géén reactie nodig is vanuit het component
	Communicatie vanuit het Component naar de ToolBus	rec-event	Geeft aan welke gebeurtenissen die vanuit het component worden geïnitieerd door de ToolBus worden geregistreerd.

Tabel 1 Beschrijving Idef processen

Soort	**TB**		
Omschrijving	De "tb" scripts definiëren de processen binnen de ToolBus. Hieronder valt bijvoorbeeld het aanvragen en versturen van informatie naar de ".idef" proces instanties voor de communicatie met een component.		
Communicatie	Communicatie vanuit ToolBus proces naar ToolBus proces	Commando	Omschrijving
		snd-msg	Persoonlijk bericht welke wordt afgehandeld door een specifiek proces die dit bericht verwacht
		rec-msg	Geeft aan welk persoonlijk bericht het proces waarin dit statement staat afhandelt.
		snd-note	Algemeen bericht welke wordt verstuurd naar alle processen die op dit type bericht zijn geabonneerd.
		rec-note	Geeft aan welk algemeen bericht waarop het proces is geabonneerd wordt afgehandeld.
		subscribe	Hiermee kan worden aangegeven welke algemene berichten het proces wil ontvangen.

Tabel 2 Beschrijving Tb processen

De commando's[2] die in bovenstaande tabellen zijn opgenomen zijn de commando's die van belang zijn voor het achterhalen van de entry-points. De meest eenvoudige manier om de entry-points naar een component te bepalen is om vanuit de idef scripts de aanroepen van en naar het component te extraheren op basis van de argumenten die aan de commando's worden meegegeven. Nadeel van deze "kort door de bocht" aanpak is dat er geen rekening wordt gehouden met het feit dat er binnen de ToolBus scripts ook reeds dode code aanwezig kan zijn. Er zijn in feite twee vormen van dode code binnen de ToolBus scripts te onderscheiden.

De eerste vorm van dode code gezien vanuit de ToolBus is in de richting van de functionaliteit die het verbonden component aanbiedt aan de ToolBus. Het kan namelijk zo zijn dat een idef script een bericht kan ontvangen (rec-msg) welke vervolgens een aanroep naar een component doet (snd-eval,snd-do) maar dat er geen enkel proces is die het betreffende bericht verstuurt(snd-msg). In feite is er dan wel een entry-point gedefinieerd in het idef script naar het component maar deze zal nooit als zodanig worden gebruikt en is dus in feite dode code.

De tweede vorm van dode code gezien vanuit de ToolBus is in de richting van het verbonden component naar de ToolBus. Een voorbeeld hiervan is dat een idef-script een gebeurtenis vanuit het component afhandelt (rec-event) en vervolgens een algemeen bericht verstuurt (snd-note) maar dat niemand daarop is geabonneerd (subscribe) of dat de abonnee niets met het bericht doet (rec-note). In dat geval is er ook sprake van dode code binnen het component als gevolg van de dode code binnen het ToolBus script en hoeft de gebeurtenis vanuit het component nooit te worden verstuurd.

Bij beide vormen van analyse is uiteraard de voorwaarde dat men over alle Toolbus Scripts beschikt die hierbij een rol spelen.

Buiten de entry-points die kunnen worden achterhaald op basis van de ToolBus scripts moet er ook rekening worden gehouden met het feit of een component naast de connectie met de ToolBus ook nog standalone danwel als bibliotheek kan worden gebruikt. In het geval er ook sprake is van standalone gebruik zal de "`main`" methode ook als entry-point moeten worden beschouwd. Wanneer een component ook wordt gebruikt als bibliotheek treedt er een fundamenteel probleem op bij de vaststelling van de entry-points. In dat geval is het namelijk nodig om te achterhalen welke delen van de bibliotheek allemaal worden gebruikt. Dit is een

onmogelijke opgave bij de bibliotheken die behoren tot de Meta-omgeving aangezien deze in verschillende andere projecten worden gebruikt. Als gevolg hiervan kan men bij een bibliotheek slechts beperkt dode code vinden aangezien de entry-points in dat geval alle mogelijke toegangen tot de bibliotheek moet bevatten. Hierdoor zal de meeste code als werkend worden beschouwd en kunnen eigenlijk alleen uitspraken worden gedaan over code die gewoonweg nooit bereikbaar is binnen een bibliotheek.

Nu duidelijk is op welke wijze de entry-points naar de verschillende componenten van de Meta-omgeving kunnen worden bepaald zal in de volgende paragraaf kort worden stilgestaan bij de dode code analyse op basis van deze entry-points.

4.4.2 Bepaling van onbereikbare methodes vanuit gevonden entry-points

De combinatie van statische en dynamische analyse zoals deze besproken is in paragraaf 4.3.2 kan worden toegepast bij de Meta-omgeving om vanuit de entry-points dode code te detecteren. Het is hierbij van belang dat de analyse zowel in C als in Java kan plaatsvinden aangezien de componenten behorende bij de Meta-omgeving hierin zijn geschreven. Veel van de broncode is echter gegenereerd vanuit een ASF+SDF specificatie[2]. Het feit dat een groot gedeelte van de code gegenereerd wordt maakt het vinden van de dode code enigszins ingewikkelder.

Wanneer een groot gedeelte van een applicatie wordt gegenereerd door middel van een code-generator is het achterhalen van de plek waar de dode code ontstaat een stuk onduidelijker. Dit komt doordat er moet worden achterhaald of de dode code voortkomt uit het model dat is gebruikt om de code uit te genereren of dat de dode code wordt veroorzaakt door de manier van code genereren door de gebruikte generator.

Er is voor gekozen om geen rekening te houden met de problemen die komen kijken bij het achterhalen van de daadwerkelijke bron die de dode code veroorzaakt op het moment dat er sprake is van gegenereerde code. Daarnaast zal niet verder worden ingegaan op de manier waarop de entry-points naar de onderdelen van de Meta-omgeving kunnen worden bepaald.

Vanaf hoofdstuk 5 zal worden ingegaan op de invulling van de dode code analyse. Er zal worden stilgestaan bij de invulling hiervan bij het analyseren van Java broncode. Hiervoor is gekozen omdat veel van de C code wordt gegenereerd en ook omdat de Java code een steeds groter deel gaat uitmaken van de Meta-omgeving.

4.5 Representatie van gevonden dode code

De toepasbaarheid van dode code analyse is niet alleen afhankelijk van de onderliggende strategie waarmee de analyse wordt uitgevoerd. Om daadwerkelijk te kunnen worden gebruikt is het belangrijk om erbij stil te staan op welke wijze de resultaten van de analyse aan de programmeur kunnen worden overgebracht. Op basis hiervan is ook de derde deelvraag van het onderzoek opgesteld.

Zoals besproken in hoofdstuk 2 is het nooit aan te tonen dat door het verwijderen van dode code de applicatie semantisch onaangetast blijft. Uit veiligheidsoverwegingen zal het dus altijd een programmeur moeten zijn die de uiteindelijke beslissing neemt of de geconstateerde dode code ook daadwerkelijk niet gebruikt wordt of gebruikt gaat worden. Om dit mogelijk te maken zal de informatie dus moeten worden overgebracht aan de programmeur.

De meest eenvoudige manier van representeren is het geven van een lijst van methodes welke niet worden gebruikt en een verwijzing naar het bestand waarin deze methodes zijn opgenomen. Dit is relatief eenvoudig te realiseren maar is onhoudbaar richting de programmeur wanneer er erg veel dode code wordt gevonden en de mogelijkheid op false positives aanwezig is. Met een dergelijke vorm van feedback naar de programmeur toe is de kans dat de resultaten van de dode code analyse worden gebruikt erg klein.

Om de programmeur snel inzicht te kunnen geven in de resultaten van de analyse is het naar mijn mening verstandig om te kiezen voor een visuele representatie waarbij de informatie die moet worden overgebracht in het getoonde figuur zit opgeslagen. Wanneer als uitgangspunt wordt gekeken naar Java dan zou een representatie waarin men de package structuur laat zien en de hoogte van de package de hoeveelheid dode code aanduidt een zeer informatief karakter hebben. De programmeur kan in één oogopslag zien waar de aandacht op moet worden

[2] Een ASF+SDF specificatie maakt het mogelijk om een grammatica en transformatieregels te beschrijven om zo bijvoorbeeld relatief eenvoudig een parser en transformatietool te bouwen.

gericht en waar zich dus de meeste dode code bevindt. Vervolgens zou de programmeur moeten kunnen inzoomen op een package om vervolgens de onderliggende klassenstructuur te zien. Hierbij zou de hoogte van iedere klasse weer de relatieve hoeveelheid dode code ten opzichte van een andere klasse moeten aangeven. De laatste stap zou dan zijn dat de programmeur op een specifieke klasse inzoomt om de broncode te zien en om hierin aanpassingen te kunnen maken. Door het systeem te laten bijhouden welke dode methodes de programmeur heeft verwijderd of als false positive heeft bestempeld kan het systeem de hoogtes van de verschillende klassen en packages updaten.

Deze vorm van grafische representatie geeft een snel beeld van de plekken in de broncode waar aandacht aan moet worden besteed en stelt de programmeur in staat om via deze grafische wijze de broncode in te zien en de methode die mogelijk dood is te bekijken en zelfstandig te analyseren. Dit verhoogt de toepasbaarheid van de applicatie van de analyse enorm.

5 Statische analyse van broncode

Nu in de voorgaande hoofdstukken de verschillende onderdelen zijn besproken die komen kijken bij dode code analyse zal vanaf dit hoofdstuk specifiek worden ingegaan op het gedeelte met betrekking tot de statische en dynamische analyse en de praktische realisatie. Gezien de termijn waarbinnen de afstudeeropdracht moest worden voltooid is ervoor gekozen om de analyse te richten op de taal Java en om het analyseren van entry-points buiten beschouwing te laten.

In dit hoofdstuk zal worden ingegaan op de statische analyse. In de eerste paragraaf zal het gebruik van relaties voor het representeren van feiten vanuit de broncode worden besproken en zullen kort enkele tools worden besproken die als uitgangspunt kunnen dienen voor de praktische implementatie. In de tweede paragraaf wordt ingegaan op het relationele model dat is gebruikt voor het representeren van de informatie die benodigd is om dode code analyse te kunnen uitvoeren. Verder wordt er in deze paragraaf ingegaan op de verschillende fases waaruit de statische analyse bestaat en wordt een overzicht gegeven van de praktisch gerealiseerde functionaliteit.

5.1 Relaties als representatie van broncode

Om dode code analyse mogelijk te maken is het nodig om de informatie vanuit de broncode te extraheren die van belang is voor dode code analyse. Een voorbeeld hiervan is de informatie over de gedeclareerde methodes binnen een applicatie.

Een natuurlijke manier om informatie op te slaan is in de vorm van relaties. Dit volgt uit het feit dat voor het begrijpen van een applicatie en de manier waarop een stuk broncode hierin een rol speelt er inzicht zal moeten zijn in de structuur van de broncode. De structuur van de broncode wordt binnen Java onder andere bepaald door de hiërarchie van de klassen, wat direct kan worden gezien als relaties tussen klassen. Daarnaast zijn bijvoorbeeld methode-aanroepen gerelateerd aan een aanroepende partij, ofwel de representatie van een call-graph. Door het leggen van verbanden tussen relaties is het vervolgens mogelijk om conclusies te trekken zoals alle methodes die niet worden aangeroepen vanuit een bepaald entry-point. Een praktijksituatie waarin een relationeel model tot zijn recht komt wordt besproken in [7]. In dit artikel wordt er ingegaan op het gebruik van een relationeel model voor de representatie van C programma's . Dit model wordt vervolgens gebruikt om een grafisch overzicht van de broncode te kunnen geven maar ook om een vorm van dode code analyse te kunnen doen.

Het gebruik van een relationeel model voor de representatie van gegevens brengt verschillende voordelen met zich mee. Een voordeel is het feit dat het een relatief eenvoudige notatiewijze is en daardoor een goede toegankelijkheid biedt. Daarnaast is het leggen van verbanden tussen relaties en het uitvoeren van operaties op relaties formeel uit te drukken binnen een taal als Rscript.

Een ander groot voordeel van het gebruik van relaties als representatie van de broncode is dat men kan abstraheren van syntactische eigenschappen van een taal en de uiteindelijke analyse op deze manier onafhankelijker kan maken van de onderliggende programmeertaal. In het geval van heterogene applicaties zoals de Meta-omgeving is dit essentieel. Uiteraard zal er een specifieke omzetting moeten zijn vanuit de programmeertaal naar de betreffende relaties maar vervolgens kan de analyse een relatie gebruiken zonder dat daarbij kennis nodig is van de onderliggende taal.

Helaas gaat het in veel gevallen niet of nauwelijks op dat de analyse geheel onafhankelijk is van de onderliggende programmeertaal. Dit komt doordat er veel semantische verschillen zijn in de verschillende talen en hiermee afhankelijk van de analyse wel of geen rekening mee moet worden gehouden. Tussen de verschillende talen zullen, voor het construeren van een relationeel model, overeenkomsten en duidelijke verschillen moeten worden gezocht. Een voorbeeld hiervan is het feit dat in Java gebruik wordt gemaakt van interfaces en in C gebruik wordt gemaakt van header files en men dit in één relatie zou kunnen vangen. Vanuit het oogpunt van dode code analyse hebben beide namelijk de semantische eigenschap dat ze uitdrukken welke functies er aanwezig zijn in de bronbestanden waarin ze worden geïmplementeerd[3] danwel ge-include.

Door het gebruik van een relationeel model ontstaat ook nog het voordeel dat de informatie die aan dit relationele model voldoet ook kan worden gebruikt voor andere analyses dan alleen die van dode code. Wanneer rechtstreeks zou worden geprobeerd de dode code te extraheren zonder een dergelijk tussenformaat voor

[3] Het enige verschil is dat het bij C niet zo hoeft te zijn dat iedere functie uit de header file in hetzelfde bronbestand is geïmplementeerd wat in het geval van Java alleen kan optreden als de interface door een abstracte klasse wordt geïmplementeerd.

representatie van de broncode zou dit niet mogelijk zijn. Na later zal blijken zal het opgestelde relationele model ook inzicht kunnen geven in bijvoorbeeld de genestheid van Java-klassen in andere klassen of het aantal methodes dat vanuit een bepaalde methode wordt aangeroepen. Hiermee is het mogelijk allerlei andere conclusies over een applicatie te trekken zonder dat de informatie opnieuw moet worden geëxtraheerd vanuit de originele broncode. Dit verbetert de herbruikbaarheid.

Voor de extractie van feiten uit de broncode en het analyseren van de broncode zijn verschillende pakketten beschikbaar. Voor de dode code-analyse zijn enkele afwegingen gemaakt over welke van deze zou worden gebruikt als uitgangspunt voor de statische analyse. In de volgende twee paragrafen zal kort stil worden gestaan bij enkele pakketten en de uiteindelijke keuze die is gemaakt.

5.1.1 Beschikbare relatie extractie voor Java

Voor de afweging welke tool als startpunt voor het implementeren van de dode code analyse zal moeten worden gebruikt zijn vier mogelijkheden bekeken:

- Tacle
- RevJava 0.8.5
- JAD
- Relational Calculus 3.0

Tacle
Tijdens de literatuurstudie kwam naar voren dat er een Eclipse plug-in[W4] is waarmee “reachability” analyses kunnen worden gedaan. Wanneer kan worden bepaald welke methodes bereikbaar zijn vanuit een entry-point moet het ook mogelijk zijn om dode code analyse uit te voeren. Voor de analyse maakt Tacle gebruik van Eclipse bibliotheken. Het gevolg hiervan is dat de plug-in zeer afhankelijk is van de manier waarop de bibliotheken zijn geïmplementeerd. Dit is ook het voornaamste probleem dat werd ondervonden bij het aanpassen van de plug-in. Voorzover kon worden nagegaan was het niet mogelijk om de plug-in zo aan te passen dat zowel de vraag “Wat is dode code?” als de vraag “Wat is werkende code?” kon worden bepaald zoals bedoeld in paragraaf 4.3.2. Dit had te maken met het feit dat niet kon worden achterhaald wanneer er onzekerheid optrad als gevolg van polymorfisme bij de analyse. Daarnaast was het ook niet duidelijk onder welke aannames de analyse plaatsvond en waar zich dus eventuele false positives en false negatives konden bevinden.

RevJava
Met RevJava is het mogelijk om een breed scala aan analyses op Java bytecode uit te voeren. Vanaf het niveau van packages tot op het niveau van velden worden verschillende metrieken aangeboden. Onder andere biedt RevJava de mogelijkheid om ongebruikte methodes te bepalen. De reden dat ondanks deze mogelijkheid deze tool niet gekozen is als uitgangspunt voor de dode code analyse, is het feit dat de aannames waaronder de analyse werd gedaan niet duidelijk waren. Hierdoor is het net zoals bij Tacle moeilijk te bepalen in hoeverre er bijvoorbeeld false positives zouden kunnen optreden en in hoeverre de resultaten compleet zijn. Daarnaast ontstaan door het uitvoeren van de analyse op de bytecode ook beperkingen.

JAD
De applicatie JAD [W17] is een decompiler van Java bestanden en maakt het dus mogelijk om gecompileerde bestanden om te zetten naar broncode. De reden dat dit gebruikt zou kunnen worden voor feitenextractie is omdat er zich in de gecompileerde bestanden veel meer informatie bevindt dan in de originele broncode. De Java compiler voegt namelijk extra informatie toe zoals de fully qualified names van de types van gebruikte variabelen. Het voordeel hiervan is dat er geen algoritmes hoeven te worden geschreven om deze impliciete informatie die zich in de broncode bevindt expliciet te maken. Een nadeel van een decompiler is dat de overeenkomst met de originele broncode niet gelijk hoeft te zijn doordat er bijvoorbeeld veranderingen in de volgorde van declaraties kunnen zijn. Hierdoor zal het moeilijker zijn om de programmeur naar een specifiek onderdeel binnen de broncode te leiden door gebrek aan correcte locatie-informatie. Daarnaast is de informatie die door de decompiler wordt teruggeven te beperkt aangezien bijvoorbeeld de fully qualified names van constructor-aanroepen niet beschikbaar zijn. Het gevolg hiervan is dat er toch een algoritme zal moeten worden geschreven om deze aanroepen fully qualified te maken. Dit algoritme kan dan meteen ook worden gebruikt voor het extraheren van de informatie die de decompiler wel oplevert. Er is dus in feite geen winst door het gebruik van de decompiler.

Relational Calculus 3.0
Het relational calculus pakket biedt de mogelijkheid om informatie te extraheren vanuit Java broncode in de vorm van relaties. Het pakket is geschreven in ASF+SDF waardoor de beperkingen en mogelijkheden van de

feitenextractie te achterhalen zijn op basis van de syntactische en semantische definities waaruit de ASF+SDF specificatie bestaat. Naast de feitenextractie wordt ook de mogelijkheid geboden om door middel van de taal Rscript, relationele operaties te beschrijven om deze vervolgens op de geëxtraheerde feiten toe te passen. Op basis van het inzicht in de beperkingen (bijlage A) die de huidige feitenextractie bood en de reeds opgedane ervaring met de ASF+SDF taal tijdens de opleiding is besloten om dit pakket als uitgangspunt te nemen voor het implementeren van de statische analyse.

5.1.2 Beschikbare relatie extractie voor C

Ondanks dat de dode code analyse voor C niet zal worden uitgewerkt binnen dit afstudeeronderzoek zal er toch stil worden gestaan bij enkele tools die kunnen worden gebruikt voor de extractie van relaties uit C code.

In de literatuurstudie zijn reeds twee systemen[7][8] naar voren gekomen die relationele informatie kunnen extraheren uit C programma's. Er wordt in de betreffende artikelen ook een voorbeeld gegeven van detectie van dode code op basis van deze relationele informatie. Het is helaas ook bij deze tool onduidelijk in hoeverre er sprake is van false positives en in hoeverre de analyse compleet is. Daarnaast is deze tool op het moment van schrijven niet meer beschikbaar op de bijbehorende website.

Een andere tool welke zou kunnen worden gebruikt is CFlow[W15]. Deze gnu tool maakt het mogelijk om vanuit een entry-point te bepalen welke methodes worden aangeroepen. Daarnaast geeft de tool inzicht in de signatuur van alle methodes die binnen een bronbestand zijn gedeclareerd. Dit geeft genoeg informatie om een vorm van dode code analyse te kunnen doen. Net zoals bij de andere twee tools voor C is ook hier onduidelijkheid over de compleetheid en de mate van false positives. Het voordeel van CFlow is dat de broncode beschikbaar is en op deze wijze de volledigheid van de analyse zou kunnen worden bepaald.

5.2 Benodigde relaties en operaties voor dode code extractie

Voordat het mogelijk is om dode code door middel van relationele calculus te achterhalen is de eerste stap het opzetten van een relationeel model waarop de calculus kan worden toegepast. Het doel van het relationele model is om de onderdelen uit de broncode te kunnen representeren die van belang zijn voor de dode code analyse.

Zoals besproken in paragraaf 5.1.1 is als uitgangspunt gekozen voor de relationele calculus implementatie zoals deze voor de Meta-omgeving beschikbaar is. In bijlage A is het relationele model opgenomen zoals deze in de standaard implementatie van het relationele calculus pakket aanwezig is. Met behulp van enkele Java voorbeelden[4] zijn verschillende problemen met dit relationele model naar voren gekomen op het gebied van compleetheid van de geëxtraheerde informatie. Deze informatie is ook opgenomen in bijlage A. Naast de compleetheid van de feiten waren er ook enkele syntactische beperkingen die volgden uit de gebruikte Java-grammatica voor het interpreteren van de broncode. Deze moest onder andere worden aangepast zodat interfacedeclaraties binnen interfaces konden worden geparseerd en er moesten verschillende ambiguïteiten worden opgelost.

In het plan van aanpak is aangegeven dat gekozen is voor een iteratieve ontwikkelstrategie. Om dit proces te ondersteunen en om de toegankelijkheid van de dode code analyse te verbeteren zijn er drie fases in de dode code analyse te onderscheiden:

- Extractie van feiten uit broncode
- Verrijking van de informatie in de relaties
- Daadwerkelijke dode code analyse

Het voordeel van deze opdeling in drie fases is dat er duidelijk kan worden toegewerkt naar een resultaat aangezien de eindtermen waaraan iedere fase moet voldoen vooraf zullen worden opgesteld.

In de volgende paragrafen zal worden ingegaan op deze indeling in drie fases. Om dit mogelijk te maken zal in de eerste paragraaf kort worden ingegaan op het gebruikte relationele model voor het representeren van de feiten uit de broncode. In de flow-diagrammen in bijlage I kan worden gekeken voor het bepalen van de plek van iedere fase in de gehele dode code analyse.

[4] Onder andere op basis van http://www.javaworld.com/javaworld/jw-02-2002/jw-0201-java101-p2.html

5.2.1 Onderliggend relationeel model

In paragraaf 5.1 is reeds aangegeven dat ervoor gekozen is om de geëxtraheerde feiten uit de broncode te representeren in de vorm van relaties. Om dit mogelijk te maken zal er een model moeten worden geconstrueerd waarin de benodigde informatie voor dode code analyse kan worden opgenomen. In dit model zal rekening moeten worden gehouden met de beperkte relationele operaties zoals deze binnen Rscript te gebruiken zijn. Daarnaast zal er ook rekening moeten worden gehouden met taalspecifieke eigenschappen van een programmeertaal wanneer deze van invloed zijn op de dode code analyse. Zo is het belangrijk om zowel een objectgeoriënteerde taal te kunnen uitdrukken alsmede een niet-objectgeoriënteerde taal.

In bijlage B zijn de type-definities opgenomen die in de relaties worden gebruikt. Deze type-definities worden gebruikt om in de relaties op een eenduidige manier informatie vanuit de broncode te kunnen representeren. Een voorbeeld is de type definitie voor argumenten. Deze maakt het mogelijk om zowel de volgorde als de types van de argumenten op een vaste manier te representeren. Deze type-definitie wordt vervolgens gebruikt in de representatie van methode-declaraties en in de representatie van methode-aanroepen. De vorm van de type-definitie van de argumenten maakt het op deze manier mede mogelijk om de signatuur van een methode te kunnen vaststellen, wat van belang is bij een taal waarin polymorfisme een rol speelt.

De belangrijkste informatie over het relationele model is opgenomen in bijlage C. Hierin zijn alle relaties opgenomen die worden gebruikt om de broncode te representeren zodanig dat er een dode code analyse op mogelijk is. Zoals in de bijlage valt te zien is van elke relatie de definitie opgenomen volgens het formaat dat binnen Rscript wordt gebruikt, is aangegeven wat het doel van de relatie binnen het model is en zijn de voorwaarden voor het gebruik van de relatie beschreven. De voorwaarden zijn voornamelijk belangrijk voor de dode code analyse aangezien deze de beperkingen aangeven in de informatie die in de relatie wordt opgenomen. Zo is het bijvoorbeeld van belang om te realiseren dat bij de relatie die de declaratie van variabelen representeert, het type van de variabele die in de relatie staat het type is dat compile-time bekend is. Dit komt voort uit het eerder besproken type-analyse probleem.

De vormgeving van het relationele model zoal deze in bijlage C is opgenomen is gedaan met als uitgangspunt de beschikbare relaties vanuit het Relational Calculus 3.0 pakket. De grootste invloed op het model heeft uiteraard de informatie die vanuit de broncode nodig is voor het kunnen detecteren van dode code.

De eerste gedachte is dat er slechts twee relaties nodig zijn om dode methodes te kunnen detecteren;alle methode-declaraties en alle aanroepen die plaatsvinden vanuit de entry-points. Deze stelling gaat niet op aangezien er twee problemen ontstaan bij deze visie.
Ten eerste is het zo dat voor het opstellen van deze twee relaties veel informatie benodigd is uit de broncode, zoals de types die binnen hetzelfde package worden gedefinieerd. Deze informatie wordt gebruikt voor het vaststellen van unieke namen van de types waartoe de gedeclareerde methodes behoren. Zonder deze informatie is het mogelijk dat er ambiguïteit ontstaat als gevolg van types met dezelfde naam die in verschillende packages zijn gedeclareerd.
Het tweede probleem is dat de relatie die de methode-aanroepen representeert onderhevig is aan het type-analyse probleem (polymorfisme) en er daardoor informatie nodig is over de hiërarchie van de types om te bepalen welke methode declaraties mogelijk zijn aangesproken als gevolg van een aanroep. Deze en andere afhankelijkheden zijn opgenomen in de definities van de relaties in de bijlage.

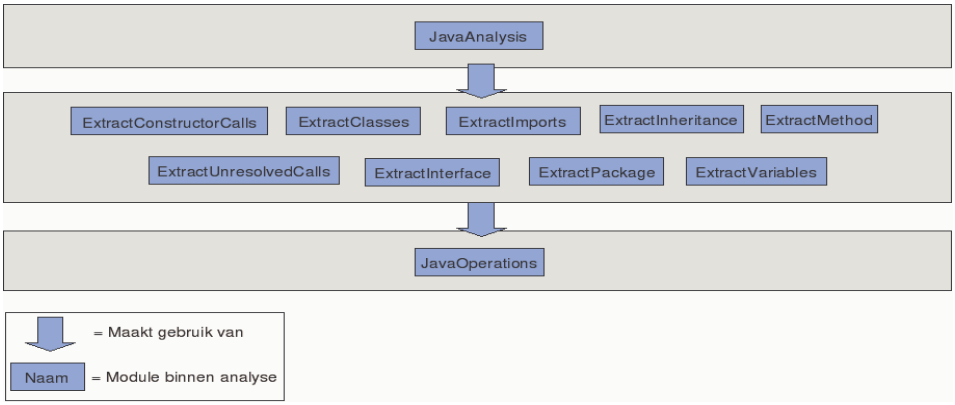
Bij het bestuderen van het relationele model in bijlage C zal opvallen dat er afstemming heeft plaatsgevonden op de objectgeoriënteerdheid van Java. Dit is het gevolg van de afstemming tussen de nauwkeurigheid van de statische analyse enerzijds en de taal-onafhankelijkheid van het onderliggende relationele model anderzijds. In hoofdstuk 7 zal op de oorzaken en afwegingen met betrekking tot de taal-afhankelijkheid worden ingegaan.

Na het opstellen van het relationele model is het nodig om deze relaties vanuit de broncode te extraheren. De volgende paragraaf zal dit onderwerp behandelen.

5.2.2 Extractie van feiten uit broncode

Voor het extraheren van de feiten en de omzetting naar de relaties wordt gebruik gemaakt van ASF+SDF specificaties en dus van de Meta-omgeving. Er zijn aanzienlijke veranderingen[5] in het orginele relational calculus pakket doorgevoerd om deze te laten voldoen aan het relationele model zoals opgenomen in bijlage C. In figuur 4 is de structuur van de modules opgenomen.

[5] Er is een beperkte changelog beschikbaar van de ASF+SDF specificatie



Figuur 4 Importrelaties van feitenextractie

In onderstaande tabel 3 zal kort op iedere module worden ingegaan.

Modulenaam	Functie	Opmerkingen
JavaAnalysis	Deze module biedt de functionaliteit aan voor het extraheren van de verschillende relaties van een enkel Java bronbestand. Hiervoor wordt gebruik gemaakt van de onderliggende modules waarin specifieke extracties zijn gedefinieerd.	Slechts beperkte wijzigingen aangebracht ten opzichte van originele pakket.
ExtractMethod	Met behulp van deze module kan de METHOD relatie worden geëxtraheerd. De informatie over het returntype en het type van de argumenten is niet fully qualified.	Grote wijzigingen om het mogelijk te maken de signatuur te bepalen.
ExtractUnresolvedCalls	Met behulp van deze module kan de UNRESOLVED_CALL relatie worden geëxtraheerd. Deze bevat informatie die benodigd is voor het achterhalen van de methode-aanroepen en het opbouwen van de relatie METHOD_CALL.	Nieuwe module
ExtractConstructorCalls	Deze module bouwt de CONSTRUCTOR_CALL relatie op. Deze is benodigd voor het opbouwen van de METHOD_CALL relatie.	Nieuwe module
ExtractClasses	Deze module biedt de mogelijkheid om alle klassen en bijbehorende structuur van een bronbestand te extraheren in de vorm van de relatie TYPE.	Wijzigingen om de structuur te kunnen bijhouden tijdens het parseren en syntactische wijzigingen in verband met compatibiliteit met Java 1.4
ExtractInterface	Met behulp van deze module kan de INTERFACE relatie worden opgebouwd en de structuur van de interfaces in een bronbestand worden bepaald.	Wijzigingen om de structuur te kunnen bijhouden tijdens het parseren en syntactische wijzigingen in verband met compatibiliteit met Java 1.4
ExtractVariables	Deze module biedt de functionaliteit om zowel de VARIABLE_DECL relatie als de FIELD_DECL relatie te extraheren. De informatie over de returntypes, de types van de argumenten en het type van de variabele of fielddeclaratie zijn niet fully qualified.	Grote wijzigingen om de extractie aan het relationele model te laten voldoen aangezien veel informatie in de bestaande relatie ontbrak.
ExtractImport	Deze module extraheert de IMPORT relatie	Nieuwe module
ExtractPackage	Deze module extraheert de PACKAGE relatie en biedt functionaliteit om de packagenaam te bepalen ten behoeve van de bepaling van fully qualified names.	Beperkte wijzigingen om deze te laten voldoen aan het relationele model. Wel nieuwe functionaliteit ten behoeven

Modulenaam	Functie	Opmerkingen
		van de fully qualified name.
ExtractInheritance	Door middel van deze module kunnen de afhankelijkheden tussen types worden bepaald. Hiermee worden de relaties EXTENDS_CLASS, EXTENDS_INTERFACE en IMPLEMENTS geëxtraheerd uit het bronbestand. In het geval van de EXTENDS_CLASS en EXTENDS_INTERFACE relatie zijn de namen van de subtypes niet fully qualified. In het geval van de IMPLEMENTS relatie is de naam van de geimplementeerde interface niet fully qualified.	Wijzigingen om de structuur te kunnen bijhouden tijdens het parseren.
JavaOperations	Dit is de onderliggende module die wordt gebruikt voor het transformeren van bijvoorbeeld een lijst met argumenten in de vorm van Java code naar de relatie representatie hiervan.	Verscheidene toevoegingen voor het laten voldoen van de extractie aan het relationele model.

Tabel 3 Uitleg modules feitenextractie

In deze eerste fase van de feitenextractie zal ieder bronbestand afzonderlijk worden bekeken. Als gevolg hiervan zijn er beperkingen aan de informatie die kan worden geëxtraheerd.

De beperkingen aan de eerste fase van de relatie-extractie zijn onder te verdelen in twee categorieën:

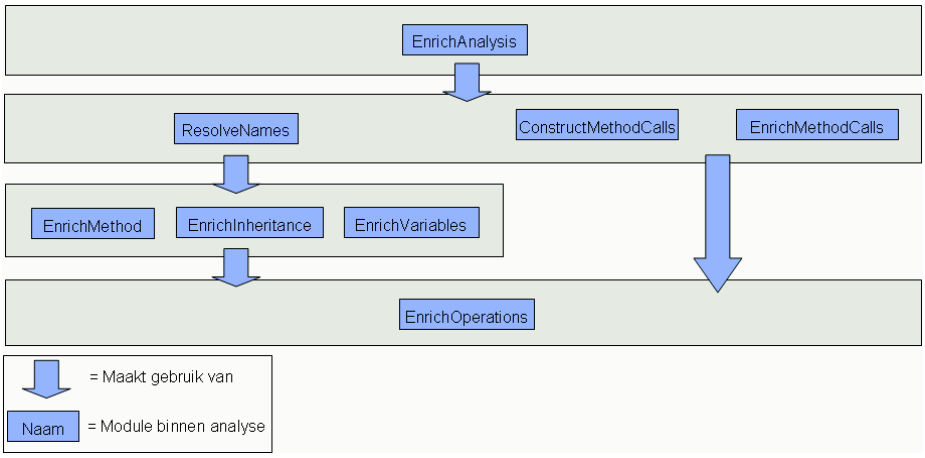
- Het niet kunnen vaststellen van de fully qualified name van de types die binnen een bronbestand worden gebruikt.
  - Dit kan ontstaan als een type binnen een ander bronbestand wordt gedeclareerd dan degene waarvan op dat moment de feiten worden geëxtraheerd.
- Het niet kunnen vaststellen van het type waarop een methode wordt uitgevoerd.
  - Dit ontstaat wanneer er gebruik wordt gemaakt van een variabele of een statische klasse of variabele waarop de methode wordt uitgevoerd.

Op basis van bovenstaande opmerkingen is er voor het volledig laten voldoen van de relaties aan het relationele model een tweede fase nodig. Deze zal in de volgende paragraaf worden besproken.

5.2.3 Verrijking van informatie in de relaties

Nadat de informatie is geëxtraheerd uit ieder bronbestand en in de vorm van relaties is omgezet moeten aanvullende operaties worden uitgevoerd om de relaties volledig te laten voldoen aan het relationele model uit bijlage C. De beperkingen die als gevolg van het per bronbestand extraheren van de relaties ontstaan worden in deze fase weer rechtgetrokken. Hiervoor zijn ASF+SDF definities gemaakt die de relaties vanuit de eerste fase kunnen interpreteren en transformeren naar het gewenste eindformaat.

In figuur 5 is de structuur van de modules van de tweede fase zichtbaar gemaakt.



Figuur 5 Importrelaties van verrijkingsfase

Zoals te zien is lijkt de structuur van de tweede fase enigszins op de structuur van de eerste fase. In plaats van extraction spreken we hier over enrichment. Dit is gedaan om duidelijk te maken hoe deze twee fases met elkaar zijn verbonden. In onderstaande tabel is de functie van iedere module opgenomen.

Modulenaam	Functie
EnrichAnalysis	Deze module zorgt ervoor dat aan de geëxtraheerde relaties uit de eerste fase de benodigde extra informatie wordt toegevoegd. Dit gebeurt door middel van de onderliggende modules.
ResolveNames	Deze module kan worden gebruikt om de typebenaming in de verschillende relaties volledig te maken. Hiermee wordt de mogelijkheid voor ambiguïteit tegen gegaan.
EnrichInheritance	Met behulp van deze module worden de relaties EXTENDS_CLASS, EXTENDS_INTERFACE en IMPLEMENTS verrijkt met fully qualified names.
EnrichVariables	Met behulp van deze module worden de relaties VARIABLE_DECL en FIELD_DECL verrijkt met fully qualified names.
EnrichMethod	Met behulp van deze module wordt de relatie METHOD verrijkt met fully qualified names.
ConstructMethodCalls	Op basis van de variabelen of klasse waarop een aanroep plaats vindt wordt het type waarop de methode-aanroep wordt gedaan bepaald.
EnrichMethodCalls	De types van de methodcall relatie worden hiermee volledig gemaakt.
EnrichOperations	Deze module wordt gebruikt door de andere modules om gegeven een mogelijk niet volledig type de bijbehorende klasse te vinden op basis van informatie uit de CLASS, INTERFACE, IMPORT en PACKAGE relatie.

De moeilijkheid bij bovenstaande modules zit voornamelijk in ConstructMethodCalls. In deze module vindt namelijk de redenering plaats om te bepalen wat het type is waarop een aanroep plaatsvindt. Hierbij moet dus rekening worden gehouden met shadowing. Deze module is niet volledig geïmplementeerd (zie bijlage F voor een overzicht van de beperkingen). Zoals later in hoofdstuk 8.4 (validatie) zal worden besproken wordt van gemiddeld ruim 70% van alle methode-aanroepen het gedeclareerde aanroepende-type herleidt. Aanroepen waarin gebruik wordt gemaakt van een field-reference (bijvoorbeeld `field.toString()`) of een aanroep naar een statisch methode (bijvoorbeeld `Class.method()`) worden nog niet omgezet. Daarnaast wanneer er gebruik wordt gemaakt van een aanroep naar een functie uit een bibliotheek waarvan de broncode niet bij de feiten-extractie beschikbaar was, wordt deze ook niet herleid.

5.2.4 Daadwerkelijke dode code analyse

Op het moment dat alle informatie vanuit de broncode in de vorm van relaties is gerepresenteerd kan de daadwerkelijke dode code analyse plaatsvinden. Hiervoor wordt gebruik gemaakt van Rscript waarmee het mogelijk is om operaties uit te voeren op de relaties.

In het Rscript wordt de opdeling gemaakt in de drie sets zoals besproken in paragraaf 4.3.2. Dit gebeurt op basis van de eigenschappen met betrekking tot polymorfisme, de entry-points en informatie over impliciete aanroepen. In bijlage I zijn de flow-diagrammen opgenomen die aangeven hoe het Rscript omgaat met polymorfisme. In bijlage D zijn de eigenschappen van Java opgenomen die een rol spelen bij de dode code analyse binnen het Rscript. In deze paragraaf zal kort stil worden gestaan bij twee van deze eigenschappen:

- Methode-overschrijving
- Methode-overloading

Methode-overschrijving

Het overschrijven van een methode binnen Java maakt het mogelijk voor een subklasse om zijn eigen invulling te geven aan een methode die reeds in zijn directe of indirecte superklasse is gedefinieerd. Als gevolg van het feit dat het mogelijk is om een variabele die het type heeft van een superklasse te laten verwijzen naar iedere directe of indirecte subklasse van deze superklasse, ontstaat er onzekerheid over welke methode daadwerkelijk wordt aangeroepen wanneer een aanroep op de variabele wordt gedaan (bijv. `a.getIt()`). Binnen het Rscript wordt dit opgelost door alle methodes te verzamelen die mogelijk worden aangeroepen. Vervolgens wordt op basis van deze informatie bepaald in welke set (zie paragraaf 4.3.2) de aanroepen worden opgenomen. Wanneer

er meerdere mogelijke aanroepen zijn wordt geen van deze aanroepen opgenomen in de werkende code set. Voor het opbouwen van de dode code set worden alle mogelijke aanroepen verzameld inclusief degene waarbij onzekerheid ontstaat en wordt vervolgens het verschil genomen met alle gedeclareerde methodes. Hieruit volgt dat de grijze set alle methodes zal bevatten die geen directe aanroep hebben ofwel waarbij onzekerheid is of deze daadwerkelijk wordt aangeroepen.

Methode-overloading
Met methode-overloading is het mogelijk om een methode te definiëren die dezelfde naam heeft als een andere methode maar waarbij de signatuur van de methode verschilt in het aantal argumenten en/of het type van de argumenten. Wanneer een methode-aanroep wordt gedaan naar een methode die overloaded is, zal dus moeten worden achterhaald welke methode daadwerkelijk is aangeroepen. Aangezien het hier gaat om een compile-time[6] beslissing zal hierbij geen onzekerheid ontstaan en is de aanroep volledig statisch te bepalen. Het is namelijk zo dat voor het matchen van de types van de argumenten die worden meegegeven aan een aanroep niet wordt gekeken naar het type waarnaar runtime mogelijk wordt gerefereerd (zoals bij methode-overschrijving) maar naar het daadwerkelijk gedeclareerde type (compile-time). Zo zal bij de aanroep `get(a)` worden gekeken naar het gedeclareerde type van argument `a` en niet naar het type van de instantie waarnaar `a` mogelijk verwijst.
Als gevolg van het feit dat voor het kunnen uitvoeren van een correcte analyse voor overloading de types van de argumenten bekend moeten zijn is dit vooralsnog niet in het Rscript opgenomen. Dit komt doordat in de tweede fase van de feitenextractie de types van de argumenten die met een aanroep worden meegegeven nog niet worden bepaald. Het probleem dat hierbij namelijk ontstaat is dat van iedere methode-aanroep het returntype bekend moet zijn. Dit houdt in dat ook van de methodes die vanuit de standaard library binnen Java kunnen worden gebruikt deze informatie benodigd is. Hiermee is nog geen rekening gehouden binnen de praktische implementatie van de analyse.

	Om geen fouten binnen de statische analyse te krijgen als gevolg van de beperkte ondersteuning van overloading binnen de analyse moet een aanname worden gemaakt. De aanname is dat wanneer er sprake is van overloading op basis van de types van de argumenten dat het returntype van deze overloaded methodes hetzelfde moet zijn. Als hieraan niet wordt voldaan kan niet met zekerheid worden bepaald wat het returntype van de methode-aanroep is en ontstaat er onzekerheid wanneer bijvoorbeeld de aanroep `Class.get(arg).toInt()` wordt gedaan. Wanneer de klasse `Class` twee `get` methodes heeft met beide 1 argument kan dan namelijk niet worden bepaald welke van de twee wordt aangeroepen. Wanneer beide methodes een ander returntype hebben ontstaat er vervolgens onzekerheid over van welk type `toInt()` wordt aangeroepen.

In bijlage D is een uitvoerige beschrijving van bovenstaande punten opgenomen en van de andere punten waarmee rekening is gehouden. In bijlage I zijn de flow-diagrammen opgenomen van de manier waarop met bovenstaande punten wordt omgegaan.

[6] De binding bij Java vindt uiteindelijk pas runtime plaats. Dit is het gevolg van methode-overschrijving en niet het gevolg van methode-overloading. Vandaar dat hier wordt gesteld dat het om een compile-time beslissing gaat op het moment dat er over methode-overloading wordt gesproken.

6 Dynamische analyse van broncode

Na het uitvoeren van de statische analyse van de broncode zal met behulp van de dynamische analyse meer inzicht moeten worden verkregen in de set met grijze code. Dit komt doordat de statische analyse geen uitsluitsel kan geven over het feit of de methodes in de grijze set worden aangeroepen(zie paragraaf 4.3.2).

6.1 Afweging mogelijkheden

In hoofdstuk 2 zijn al enige mogelijkheden geopperd om de dynamische analyse te realiseren. Er zou gebruik kunnen worden gemaakt van een profiler[1], een onderliggende virtual machine[19], AOP, of er zou gebruik kunnen worden gemaakt van ASF+SDF.

Op het moment dat de keuze zou vallen op het gebruik van een profiler ontstaat er afhankelijkheid van de uitvoervorm van een specifieke profiler. Wanneer wordt gekozen voor het gebruik van een onderliggende virtual machine om de betreffende informatie te verzamelen is er sprake van volledige taal-afhankelijkheid aangezien deze informatie dan niet zou kunnen worden hergebruikt voor een applicatie die geschreven is in een programmeertaal die niet op de virtual machine kan executeren.

Het voordeel van AOP ten opzichte van de andere mogelijkheden is dat het gebruik hiervan relatief eenvoudig is. AOP biedt onder andere de mogelijkheid om aan iedere methode een stuk code toe te voegen die een uniek bericht afdrukt op het moment dat de betreffende methode is aangeroepen. Een ander voordeel is dat het denken in de vorm van Aspects taal-onafhankelijk is. Dit is ook de reden dat AOP voor verschillende programmeertalen beschikbaar is. Het taal-afhankelijke van AOP is de benodigde compiler voor het weven, de beschrijving van de weefpunten en de functionaliteit die moet worden ingeweefd. Wanneer AOP wordt gebruikt voor de dynamische analyse dan is de taal-afhankelijkheid van deze analyse gelijk aan de taal-afhankelijkheid van AOP.

Uiteraard zijn er ook nadelen aan het gebruik van AOP voor de dynamische analyse. Het eerste nadeel is dat er functionaliteit zal moeten worden gemaakt voor de omzetting van het relatieformaat in de grijze set naar het formaat voor het beschrijven van join-points. Een zeer belangrijk ander nadeel van het gebruik van AOP is dat er afhankelijkheden ontstaan met betrekking tot de naamgeving van innerklassen. De compiler zorgt ervoor dat de namen van de innerklassen uniek worden door hier een identificatie in de vorm van een getal aan toe te kennen. Dit getal representeert de positie van de innerklassen binnen het bronbestand. Op het moment dat er twee innerklassen zijn die dezelfde naam hebben worden deze dus onderscheiden op basis van het getal en niet op basis van hun naam. Het naar mijn mening gevaarlijke hieraan is dat wanneer er als gevolg van weving met AOP of als gevolg van een optimalisatie, een verschuiving van innerklassen plaatsvindt er onzekerheid ontstaat over de daadwerkelijke innerklasse die is aangeroepen. De identificatie kan dan namelijk anders zijn dan de volgorde waarin de innerklassen zijn opgenomen in het orginele bronbestand. Het gevolg hiervan is dat de klassenaam zoals deze binnen de dynamische analyse wordt gebruikt niet hetzelfde hoeft te zijn als binnen de statische analyse. Het elimineren van methode-aanroepen uit de grijze set van dode code zal dus op deze manier met AOP niet betrouwbaar zijn wanneer er sprake is van een klasse met daarin innerklassen met dezelfde naam.

Wanneer gebruik zou worden gemaakt van ASF+SDF voor de dynamische analyse is het voordeel dat er volledige controle is over de inhoud van het bericht dat wordt afgedrukt bij een methode-aanroep. Hiermee is het mogelijk om innerklassen met dezelfde naamgeving uniek te identificeren op basis van de lokatie-informatie zoals deze ook binnen het relationele model is toegepast. Dit komt doordat ASF+SDF kan worden gebruikt om de originele bronbestanden te transformeren naar bronbestanden waarin de functionaliteit is opgenomen om de berichten af te drukken. Vervolgens kunnen deze bestanden worden geoptimaliseerd of met behulp van AOP worden gemanipuleerd zonder dat dit invloed heeft op de inhoud van het bericht dat door een methode bij de aanroep wordt afgedrukt. De onzekerheid die ontstaat bij het gebruik van AOP als gevolg van de afhankelijkheid van de volgorde van declaratie is hiermee weg. Een ander voordeel van het gebruik van ASF+SDF is dat de statische analyse ook volledig in ASF+SDF is geschreven waardoor er geen conversies hoeven te worden gedaan zoals bij AOP benodigd is in verband met het beschrijven van de joinpoints. Een nadeel van het gebruik van ASD+SDF is dat de taal-afhankelijkheid aanzienlijk groter is dan in het geval van AOP. Bij het gebruik van ASF+SDF is namelijk voor iedere programmeertaal waarop transformaties worden gedaan een grammaticale beschrijving nodig terwijl voor AOP gewoonweg de correcte compiler kan worden gedownload en aan het correcte join point formaat moet worden voldaan. Maar aangezien voor de statische analyse deze grammatica ook reeds beschikbaar moet zijn is dit niet echt een groot nadeel.

6.2 Keuze dynamische analyse

Voor de praktische implementatie van de dynamische analyse is gekozen voor het toepassen van AOP. Het gevolg is dat er onzekerheid ontstaat over de aanroep van methodes binnen innerklassen, maar een groot voordeel is dat het relatief eenvoudig te realiseren is. Hierdoor kan meer tijd worden besteed aan de praktische implementatie van de statische analyse hetgeen een stuk complexer is.

Het gevolg is dat het nodig is om de relationele informatie die vanuit de statische analyse wordt gegenereerd om te zetten naar join points die binnen AOP kunnen worden gebruikt. De ingeweefde methode die het bericht afdrukt maakt gebruik van context-informatie om de klassenaam, methodenaam en de argumenten te bepalen. Deze informatie wordt in de methode omgezet naar het entry-point formaat zoals deze binnen de statische analyse kan worden gebruikt. Hiermee is de iteratieve aanroep van de statische en dynamische analyse mogelijk te maken(zie paragraaf 4.3.2).

In de praktische implementatie van de dode code analyse is de omzetting van de methodes in de grijze code set naar de join points in AOP formaat niet gerealiseerd. Er is wel uitgezocht hoe AOP kan worden gebruikt en er is een template beschikbaar waarin ook de Java code is opgenomen waarin handmatig de benodigde join points moeten worden opgenomen. Dit is verder niet uitgezocht aangezien het toch de voorkeur heeft om uiteindelijk de dynamische analyse mogelijk te maken met behulp van ASF+SDF.

Met deze korte bespreking over de realisatie van de dynamische analyse wordt het praktische gedeelte besloten. In het volgende hoofdstuk zal worden stilgestaan bij de gevolgen van de besluiten die zijn genomen in de praktische implementatie en de gevolgen hiervan op de taal-onafhankelijkheid van de analyse.

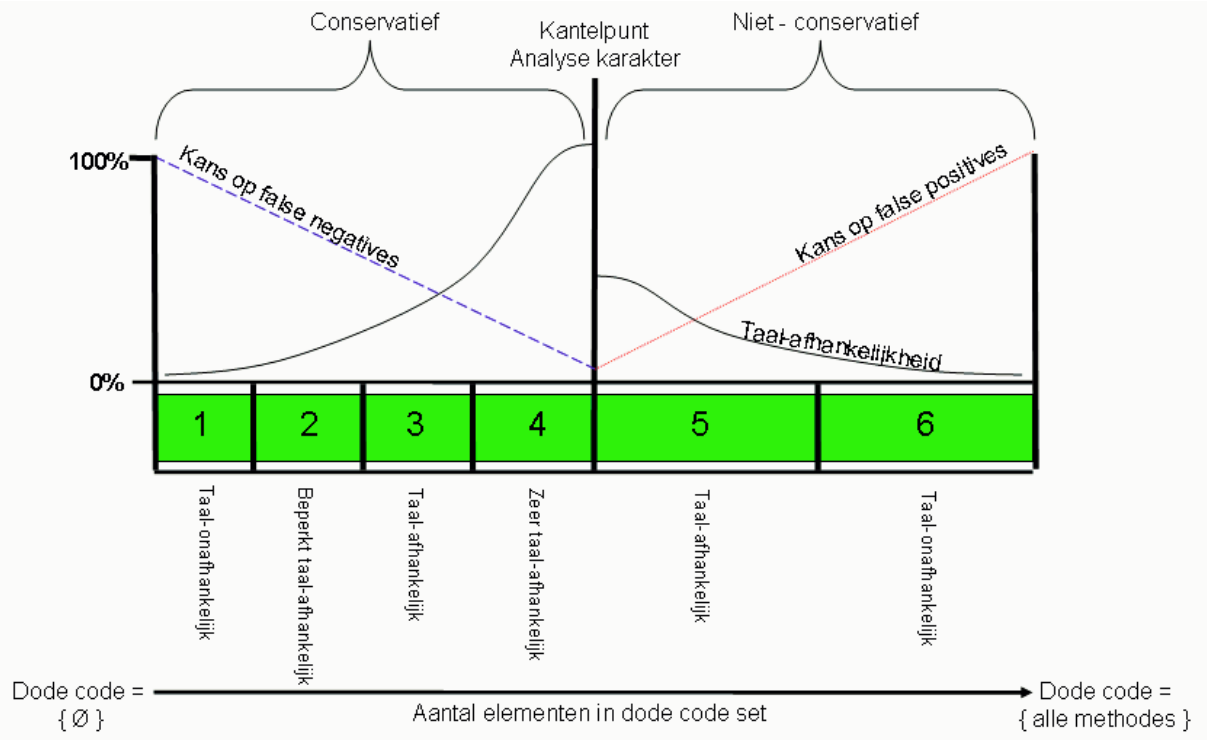
7 Taal-onafhankelijkheid van de dode code analyse

In hoofdstuk 5 is stil gestaan bij het gebruikte relationele model, de daadwerkelijke statische dode code analyse en enkele invloeden van specifieke eigenschappen van Java (zie bijlage D). Het gevolg van de complexiteit van Java is dat er taal-afhankelijkheid ontstaat in de analyse. Het is belangrijk om inzicht te krijgen in de mate van taal-afhankelijkheid ten opzichte van andere vormen van analyse die mogelijk zijn. Zo kan namelijk de geschiktheid van de dode code analyse worden beoordeeld voor heterogene applicaties. In dit hoofdstuk zal de deelvraag "*In hoeverre is het mogelijk om het analyse gedeelte taal-onafhankelijk te houden.*" worden beantwoord aan de hand van figuur 6 en sluit daarmee de beantwoording van de onderzoeksvraag.

In de eerste paragraaf zal worden stilgestaan bij de invloeden op de taal-afhankelijkheid waarna in de tweede paragraaf de voordelen worden besproken van het gebruik van dynamische analyse om de taal-afhankelijkheid te sturen.

7.1 Bepaling van invloeden op taal-afhankelijkheid

In figuur 6 zijn de dimensies[7] uitgebeeld die een rol spelen bij de taal-afhankelijkheid van de dode code analyse. In dit figuur zijn mogelijke analysevormen ingedeeld in blokken aangegeven met de nummers 1 t/m 6.



Figuur 6 Factoren voor afweging analysevorm

Het eerste belangrijke onderdeel in figuur 6 is het "Kantelpunt Analyse karakter". Aan de linkerkant van deze lijn bevinden zich de conservatieve analyses en aan de rechterkant de niet-conservatieve analyses. Aangezien door het karakter van de analyse het soort foutieve resultaten wordt bepaald is ook de kans op false positives[8] en false negatives[9] in het figuur opgenomen,

Bij conservatieve analysevormen is er, zoals in de afbeelding is te zien, wel sprake van een kans op false negatives maar niet op false positives[10]. De reden hiervan is dat bij een conservatieve analyse op safe wordt gespeeld. Het gevolg van het conservatieve karakter voor de dode code analyse is dat wanneer van een methode-aanroep niet exact duidelijk is van welke methode-declaratie deze runtime gebruik maakt, alle mogelijke

[7] De breedte van de onderverdeelde elementen tussen de zwarte lijnen heeft geen betekenis

[8] Methodes die niet dood zijn en dus worden gebruikt maar toch in de dode code set staan

[9] Methodes die dood zijn maar die niet in de dode code set zijn opgenomen.

[10] Onder false positives vallen in dit geval niet de false positives die ontstaan als gevolg van beperkingen in de vaststelling van de entry-points naar de applicatie. Dit zijn reflectie (zie bijlage D), impliciete aanroepen (zie bijlage D) en het gebruik van een applicatie als bibliotheek (zie paragraaf 4.4.1).

methode-declaraties die in aanmerking komen als gebruikt worden bestempeld. De methode-declaraties die niet in deze lijst zijn opgenomen kunnen dan ook nooit zijn aangeroepen[11] en kunnen als dode code worden beschouwd. Hierin zullen zich dan geen false positives bevinden. Door het conservatieve karakter van de analyse kan het vervolgens wel zo zijn dat er methode-declaraties zijn die niet worden gebruikt maar door de onzekerheid bij de analyse wel in de lijst met gebruikte methodes zijn opgenomen. Hier veroorzaakt het "op safe spelen" dus false negatives. Zoals in de afbeelding valt te zien levert de ultieme conservatieve instelling een lege dode code set op. Dit verklaart meteen ook het feit waarom er in dat geval sprake is van een kans op false negatives die de honderd procent benadert. Daarentegen is het aantal false positives bij een lege dode code set uiteraard gegarandeerd nul.

In het geval van de niet-conservatieve analysevormen is er juist sprake van de kans op false positives in plaats van false negatives. Dit komt doordat men (in tegenstelling tot conservatief) hierbij alleen methode-declaraties als gebruikt bestempelt op het moment dat er zekerheid[12] is over de aanroep ervan. Op deze manier wordt voorkomen dat methode-declaraties die niet worden gebruikt (en dus dood zijn) als werkend worden beschouwd. Hiermee worden false negatives in de dode code set voorkomen. Het gevolg hiervan is wel dat wanneer de methode-declaraties die niet in de aangeroepen set zijn opgenomen als dood worden beschouwd, dat er false positives kunnen optreden. Deze set kan dan immers methode-declaraties bevatten die in feite wel worden gebruikt maar waarbij dit door het niet-conservatieve karakter niet is geregistreerd.

De volgende belangrijke categorisering in figuur 6 is op taalspecificiteit. De eerste categorie bevat de analysevormen die taal-onafhankelijk zijn. De beperking die hieruit voortvloeit is dat er maar beperkt dode code kan worden gevonden en er dus sprake is van een zeer grote kans of false negatives. In de derde en tweede categorie neemt de taal-afhankelijkheid toe en neemt ook gestaag de kans op false negatives af. Met de vierde categorie worden de zeer taal-afhankelijke analysevormen bedoeld waarin de syntax en vooral de semantiek van een programmeertaal zeer duidelijke ingebed is in de analyse. Het moge duidelijk zijn dat dit ook de enige vorm van analyse is waarmee de kans op het detecteren van alle dode code het grootst is en dus zowel de kans op false positives als de kans op false negatives het nul-niveau naderen. Aangezien het detecteren van dode code nog altijd een onbeslisbaar probleem blijft zal de kans op beide vormen van foutieve resultaten nooit nul procent worden.

De vijfde categorie bevindt zich aan de rechterkant van het kantelpunt welke valt onder de niet-conservatieve analysevormen en bevat beperkt taal-afhankelijke analysevormen. Ook hierbij is het weer zo dat hoe taal-onafhankelijker men wordt hoe meer kans op incorrecte resultaten. In de zesde categorie bevinden zich tenslotte de taal-onafhankelijke niet-conservatieve vormen van analyse. Hier is er een zeer grote kans op false positives en is de kans op false negatives nihil. In een uiterst niet-conservatieve analyse zal het zelfs zo zijn dat het aantal false negatives nul is aangezien dan alle gedeclareerde methodes als dood worden bestempeld. Dit is ook meteen een taal-onafhankelijke oplossing aangezien alleen maar hoeft te worden geconcludeerd dat gewoonweg alle methodes dood zijn.

De reden dat zich aan de kant van de niet conservatieve analyses geen zeer taal-afhankelijke analysevormen bevinden heeft te maken met het feit dat false positives[13] in een taalspecifieke analysevorm naar mijn mening alleen zouden kunnen ontstaan als men de semantiek van de specifieke programmeertaal verkeerd in de analyse heeft geïmplementeerd. De ultieme taalspecifieke oplossing zou immers, in het geval van Java, het programma compleet moeten kunnen doorrekenen op de semantische manier waarop de virtual machine het programma executeert en dat sluit dus gezien het conservatieve karakter false positives uit. De kans op false negatives blijft wel aanwezig en vandaar dat de categorie van zeer taalafhankelijke analyses zich niet aan de linkerkant van het kantelpunt bevindt. Een relatief bizar voorbeeld van het ontstaan van een false negative bij een taalspecifieke oplossing is dat wanneer een methode alleen wordt uitgevoerd als `Math.random()` gelijk is aan 0,23031981 het niet is te bewijzen dat deze methode in de levensduur van de applicatie wordt uitgevoerd. Een ander voorbeeld is het uitvoeren van een methode op een specifieke datum of gewoonweg excepties. Er kan dan dus ondanks een zeer taalspecifieke oplossing nog steeds sprake zijn van dode code in de conservatief vastgestelde lijst met werkende code en dus van een false negative met betrekking tot de set met dode code. De reden dat gebruikersinteractie overigens niet als voorbeeld is opgenomen als veroorzaker van false negatives heeft te maken met het feit dat dit sterk gerelateerd is aan problemen bij het vaststellen van entry-points en dus niet direct

[11] Gegeven een correcte set van entry-points

[12] Gegeven een correcte set van entry-points

[13] Onder false positives vallen in dit geval niet de false positives die ontstaan als gevolg van beperkingen in de vaststelling van de entry-points naar de applicatie. Dit zijn reflectie (zie bijlage D), impliciete aanroepen (zie bijlage D) en het gebruik van een applicatie als bibliotheek (zie paragraaf 4.4.1).

door de analyse hoeft te worden veroorzaakt. Dit is dezelfde reden waarom de problemen met reflectie, impliciete aanroepen en het gebruik van een applicatie als bibliotheek niet is opgenomen in het plaatje als veroorzakers van false positives in de conservatieve analyse. Deze komen voort uit de onzekerheid als gevolg van de programmeertaal maar niet direct als gevolg van de analysevorm.

Met behulp van figuur 6 is het nu mogelijk om de dode code analyse, zoals deze binnen dit afstudeeronderzoek wordt nagestreefd, te plaatsen tussen andere analysevormen.

Analysevorm 1: taal-onafhankelijk
De analyse van het buildproces (zie bijlage E). Hiermee is het mogelijk om op een programmeertaal-onafhankelijke manier te controleren welke bestanden er tijdens het compileren van een applicatie niet worden gebruikt. Zo kan worden achterhaald welke bronbestanden van een applicatie niet worden gebruikt. Op deze manier kan dus op bestandsniveau worden bepaald welke methodes er binnen de applicatie niet worden gebruikt en als dood kunnen worden bestempeld. Helaas is deze analyse zeer beperkt toepasbaar omdat vaak gewoonweg alles wordt gecompileerd.

Analysevorm 2: beperkt taal-afhankelijk
Hieronder valt bijvoorbeeld een analyse waarbij slechts naar de methode-declaraties wordt gekeken en naar alle aanroepen die worden gedaan binnen het bronbestand. Hierbij wordt dan niet geredeneerd vanuit een entry-point maar worden alle aanroepen binnen de applicatie beschouwd. De methodes die wel gedeclareerd zijn in de bronbestanden maar op geen enkele wijze worden aangeroepen worden vervolgens als dode code beschouwd. Deze analysevorm is meer taal-afhankelijk dan de methode gegeven bij punt 1 aangezien het nodig is om zowel methode-declaraties als methode-aanroepen te kunnen detecteren.

Analysevorm 3: taal-afhankelijk
Hieronder valt de analysevorm die wordt nagestreefd binnen dit onderzoek (zie paragraaf 4.3.2). Binnen deze analysevorm wordt specifiek rekening gehouden met objectgeoriënteerde talen en de semantische gevolgen hiervan in de vorm van bijvoorbeeld polymorfisme. In bijlage D zijn verschillende taalspecifieke eisen besproken waarmee in de dode code analyse rekening moet worden gehouden. Er is dus sprake van een aanzienlijk taal-afhankelijkere analyse dan de vorm waarvan sprake is in categorie twee. Doordat er geen redenering wordt gedaan over bijvoorbeeld de takken die wel en niet worden gevolgd in selectielogica (control-flow) valt deze analysevorm niet in categorie 4.

Analysevorm 4: zeer taal-afhankelijk
In deze vorm van analyse treedt een zeer hoge taal-afhankelijkheid op als gevolg van het interpreteren van bijvoorbeeld functionaliteit uit standaard bibliotheken of het interpreteren van selectielogica met bijbehorende expressies. Zo zijn er verschillende programmeertalen waarin er sprake is van “short circuit” bij het evalueren van booleaanse expressies. Dit houdt in dat de elementen in de expressie niet verder worden geëvalueerd op het moment dat de waarheidswaarde al niet meer kan veranderen en dus ‘true’ of ‘false’ is. Het gevolg is dat wanneer er zich methode-aanroepen in de expressie bevinden dat er kan worden geredeneerd over het feit of deze wel of niet worden aangeroepen[14]. Door ook op het niveau van dit soort taalspecifieke constructies te gaan analyseren, en dus in feite de semantiek van de onderliggende virtual machine te modelleren, kan men meer dode code constateren en toch het conservatieve karakter bewaren. In feite wordt gestreefd naar een zo klein mogelijke kans op false negatives zonder dat dit ten kost gaat van het introduceren van false positives.

Analysevorm 5: taal-afhankelijk
Door gebruik te maken van een techniek zoals AOP kan op een relatief taal-onafhankelijke manier een dynamische analyse (runtime) worden uitgevoerd ten behoeve van dode code extractie. Het gevolg van deze dynamische analyse met AOP is wel dat de kans op false positives groter wordt als gevolg van de scenario-afhankelijkheid. Het runtimegedrag geeft alleen inzicht in de methodes die gegarandeerd worden aangeroepen onder het betreffende scenario. Wanneer op basis van deze informatie wordt geconcludeerd dat de dode code set alle methodes bevat die niet zijn aangeroepen ontstaat er een grote kans op false positives.

Analysevorm 6: taal-onafhankelijk
Een analyse welke valt onder deze laatste categorie is het achterhalen van de entry-points naar de applicatie en vervolgens stellen dat alle niet entry-points dode code zijn. Dit is uiteraard een vrij onzinnige dode code analyse. Het enige voordeel is dat de kans op false negatives gegarandeerd nul is. Een zeer groot nadeel is alleen dat de

[14] Als er geen sprake is van invloeden van externe factoren, zoals gebruikers interactie, op het resultaat van de expressie

kans op false positives in veel gevallen honderd procent is. Dit komt uiteraard doordat alle gedeclareerde methodes, behalve de entry-points, zich als gevolg van deze analysevorm in de set met dode code zou bevinden.

In feite kan men stellen dat bij de analysevormen 1, 2, 3 en 4 er daadwerkelijk sprake is van dode code analyse en aan de kant van de analysevormen 5 en 6 er eigenlijk sprake is van een vorm van reachability analyse waarbij men de methodes vaststelt die met zekerheid runtime worden aangeroepen. Dit verklaart ook het feit dat bij deze analysevormen de kans op false positives toeneemt en dat in het ongunstigste geval de set met dode code alle methodes bevat die binnen de bronbestanden worden gedeclareerd.

De situatie zoals deze in figuur 6 is geschetst is duidelijk een ideaal beeld. Het valt niet vol te houden dat een conservatieve analyse geen false positives tot gevolg heeft. De reden hiervan ligt, zoals eerder benoemd, in de mogelijkheid tot reflectie, het optreden van impliciete aanroepen en het gebruik van de applicatie als bibliotheek (zie bijlage D).

In feite worden de false positives in de dode code analyse veroorzaakt door een beperking in de vaststelling van de entry-points naar de applicatie:

- Reflectie maakt het mogelijk om functionaliteit te laden in een applicatie en bepaalt hiermee de entry-points naar het gedeelte van de applicatie dat via reflectie wordt geladen.
- Impliciete aanroepen zijn het gevolg van de interne werking van de virtual machine en de gebruikte libraries en veroorzaken daarmee in feite entry-points vanuit de virtual machine naar de applicatie.
- Op het moment dat een te analyseren component als bibliotheek wordt gebruikt bepalen alle andere applicaties die dit component gebruiken de entry-points naar de applicatie. (zie paragraaf 4.4.1)

Het is belangrijk om te vermelden dat bovenstaande veroorzakers van false positives niet voortkomen uit de manier waarop binnen dit onderzoek de dode code analyse is vormgegeven. Het zijn problemen die bij iedere vorm van dode code analyse en reachability analyse een rol zullen spelen.

7.2 Voordelen van dynamische analyse op taal-onafhankelijkheid

Zoals in de bespreking van analysevorm 5 naar voren kwam kan een dynamische analyse met behulp van bijvoorbeeld AOP worden gebruikt om inzicht te krijgen in de methodes die runtime worden aangeroepen. Als gevolg van de scenario-afhankelijkheid van de analyse (zie paragraaf 4.2) is het wel zo dat niet kan worden vastgesteld of alle mogelijke aanroepen zijn geregistreerd. Wat wel valt te concluderen is dat de methodes die aangeroepen zijn ook daadwerkelijk binnen de applicatie worden gebruikt[15]. Deze informatie kan op twee manieren vervolgens door de statische analyse weer worden gebruikt:

- Ten eerste is het zo dat met de dynamische analyse inzicht kan worden verkregen in de grijze set die vanuit de statische analyse is ontstaan als gevolg van optredende onzekerheid door polymorfisme bij objectgeoriënteerde-talen. Door gericht te zoeken naar scenario's voor de dynamische analyse die ervoor zorgen dat de methodes uit de grijze set runtime worden gebruikt kan de onzekerheid worden bestreden. Zoals in paragraaf 4.3.2 is aangegeven kan er vervolgens een iteratieve toepassing plaatsvinden om de grijze set te filteren en zo het aantal false positives te verminderen die zouden ontstaan als men de methodes uit de grijze set als dood zou bestempelen. Hiermee wordt dus de zekerheid verhoogd over de conclusie die men kan trekken over de methodes in de grijze set. Hiermee verlaagt deze combinatie ook direct de onzekerheid binnen de statische analyse die optreedt als gevolg van polymorfisme. Dit gebeurt zonder geavanceerd type-analyse systeem[18]. Hiermee wordt dus de taal-afhankelijkheid van de gehele dode-code analyse positief beïnvloed.
- Doordat er geen sprake is van het gebruik van flowsensitive-afhankelijk informatie zoals over de types waarnaar een variabele mogelijk zou kunnen wijzen, kan iedere methode als entry-point worden beschouwd(zie bijlage D). Een groot voordeel hiervan is dat de dynamische analyse dus ook kan worden gebruikt om eventuele misstanden in de dode code set als gevolg van reflectie te verbeteren. Als tijdens de runtime-uitvoering een methode wordt aangeroepen via reflectie kan deze eenvoudigweg worden gedetecteerd met de afdruk van een bericht en vervolgens worden opgenomen in de lijst met geconstateerde entry-points. Er ontstaan hierdoor binnen de dode code analyse in dit onderzoek geen problemen met de redenering over de types waarnaar variabelen zouden kunnen verwijzen terwijl dit wel zou ontstaan als men een flowsensitive type-analyse systeem zou gebruiken.

[15] Waarbij uitsluitende dat een scenario wordt toegepast dat nooit tijdens het echte gebruik van de applicatie zal optreden.

	Een conclusie die kan worden getrokken uit bovenstaande bespreking is dat de nagestreefde dode code analyse in de vorm van een combinatie tussen statische en dynamische analyse een manier is om zowel false positives als false negatives in gedetecteerde dode code tegen te gaan. Door het conservatieve karakter van de statische analyse worden false positives tegengegaan. Daarentegen gaat de dynamische analyse juist false negatives tegen door het niet-conservatieve karakter van deze vorm van analyse en de toepassing ervan op de grijze set. Beide vormen van incorrecte resultaten kunnen worden bestreden omdat de statische analyse de onzekerheid uitdrukt in de opdeling in de drie sets en de dynamische analyse de onzekerheid vervolgens kan bestrijden door de zekerheid over de grijze set te verhogen.

Samenvattend kan de deelvraag "*In hoeverre is het mogelijk om het analyse gedeelte taal-onafhankelijk te houden.*" worden beantwoord door te stellen dat de taal-afhankelijkheid wordt bepaald door de hoeveelheid toegelaten false positives en de hoeveelheid toegelaten false negatives en de manier waarop deze worden bestreden. Er moet dus een evenwicht worden gevonden tussen de hoeveelheid werk die nodig is om de false positives en false negatives tegen te gaan en de hoeveelheid incorrecte resultaten die worden toegestaan. In figuur 6 is dit ter illustratie verwoord met een lineare lijn voor zowel de false positives als false negatives en een niet-lineaire functie voor de taal-afhankelijkheid. Dit drukt uit dat het in het begin loont om taal-afhankelijker te worden, maar naarmate de taal-afhankelijkheid toeneemt er een moment is waarop de afname van incorrecte resultaten sterk terugloopt. Dit is het punt dat moet worden gevonden bij het opstellen van een dode code analyse en wat in feite dus ook de taal-onafhankelijkheid van de analyse bepaalt.

	Het voordeel van de in dit onderzoek gegeven statische analyse is dat de onzekerheid als gevolg van polymorfisme geconcentreerd zit in de grijze set. Op het moment dat de statische analyse dus taal-afhankelijker wordt gemaakt ontstaat er meer zekerheid en zal de grijze set relatief in omvang afnemen. Het voordeel van de combinatie met de dynamische analyse is vervolgens dat hiermee ook de grijze set kan worden bestreden maar dan wel scenario-afhankelijk. Doordat de dynamische analyse net zo taal-afhankelijk is als dat AOP taal-afhankelijk is van een programmeertaal (zie paragraaf 6.1) kan de taal-afhankelijkheid van de statische analyse worden verkleind door meer aan de dynamische analyse over te laten. De dynamische analyse zal niet in taal-afhankelijkheid veranderen en blijft wat dat betreft altijd hetzelfde ongeacht de onzekerheid in de grijze set. De dynamische analyse is niet gerelateerd aan onzekerheid als gevolg van polymorfisme. Dit wordt allemaal in de statische analyse afgehandeld.

8 Resultaten

Dit hoofdstuk zal worden gebruikt om stil te staan bij de conclusies die kunnen worden getrokken naar aanleiding van de bespreking in de voorgaande hoofdstukken. In de eerste paragraaf zullen de voordelen worden besproken van het uitvoeren van de dode code analyse als combinatie tussen statische en dynamische analyse. In de tweede paragraaf zullen de nadelen van deze aanpak worden belicht. In de derde paragraaf wordt nog even kort stilgestaan bij de inzichten die naar aanleiding van het onderzoek zijn opgedaan. Vervolgens wordt in de vierde paragraaf ingegaan op de toepassing van de dode code analyse op het testvoorbeeld en zal een vergelijking worden gemaakt met de analyse-tool RevJava[W18]. De laatste paragraaf is gericht op future work.

8.1 Voordelen van gekozen aanpak

In deze paragraaf zal puntsgewijs worden stilgestaan bij de voordelen van de vorm van dode code analyse zoals deze binnen dit onderzoek in de voorgaande hoofdstukken is gepresenteerd.

- Beperkte type-analyse
  - Door te redeneren met de onzekerheid die optreedt als gevolg van polymorfisme is het niet nodig om een ingewikkeld type-analyse (points-to analysis[18]) systeem te gebruiken. De statische analyse wordt hierdoor eenvoudiger en taal-onafhankelijker waardoor het ontwikkelen van de dode code analyse voor een heterogeen systeem beter haalbaar is (zie hoofdstuk 5,6,7).
- Uitbreidbaarheid
  - Het is mogelijk om de analyse krachtiger te maken door meer taalspecifieke eigenschappen te betrekken bij de analyse. Dit kan op incrementele basis waarbij men in de loop der tijd de analyse verbetert. Er kan bijvoorbeeld gebruik worden gemaakt van het feit dat aan een final variabele slechts eenmalig een toekenning wordt gedaan en het type waarnaar de variabele refereert op dat moment vaststaat. Met behulp van de “referenceOption” in de relatie die de methode-aanroepen bevat (METHOD_CALL) kan worden aangegeven wat de zekerheid is over het type waarop een aanroep plaatsvindt (zie bijlage C). Hiermee wordt vervolgens in de bepaling van de dode code set, werkende code set en grijze code set automatisch rekening gehouden. Het veranderen van de zekerheid die wordt uitgedrukt met “referenceOption” is in feite het belangrijkste instrument dat nodig is om de analyse krachtiger te maken zonder dat er wijzigingen in het Rscript moeten worden gemaakt die de uiteindelijk opdeling in de drie sets maakt. Het gebruik van de “referenceOption” voor het krachtiger maken van de analyse zal leiden tot een verkleining van de grijze set waardoor er minder scenario-afhankelijkheid als gevolg van de dynamische analyse is. Dit geheel leidt dus tot meer zekerheid over het feit dat de grijze code ook als dode code kan worden bestempeld. Uiteraard komt hierbij wel weer de afweging kijken zoals deze in hoofdstuk 7 is besproken met betrekking tot de baten en kosten van het taal-afhankelijker maken van de analyse.
- Afweging taal-afhankelijkheid
  - De opbouw van de statische analyse en de combinatie met de dynamische analyse maakt het mogelijk om een afweging te maken in de taal-afhankelijkheid van de analyse (zie hoofdstuk 7). Op het moment dat de analyse nauwkeuriger wordt door middel van taal-afhankelijke constructies zullen er minder methodes in de grijze set worden opgenomen. Door de statische analyse minder taal-afhankelijk te maken ontstaat juist een grotere grijze set welke vervolgens met de dynamische analyse weer kan worden verkleind. Een dynamische analyse zal in veel gevallen eenvoudiger te construeren zijn dan een statische analyse maar heeft het nadeel van scenario-afhankelijkheid. De afweging met betrekking tot de taal-afhankelijkheid is belangrijk op het moment dat er sprake is van een heterogeen systeem. (zie hoofdstuk 7)
- Geen specifiek entry-point
  - Doordat er in de statische analyse niet hoeft te worden geredeneerd over de types waarnaar variabelen kunnen verwijzen (zoals bij flow-senstive points-to analysis) kan iedere methode als entry-point naar de applicatie worden beschouwd (zie hoofdstuk 7 en bijlage D). Het voordeel van de aanpak waarbij iedere methode als entry-point kan worden beschouwd is dat de methodes die gevonden zijn door de dynamische analyse hiermee op eenvoudige wijze kunnen worden gebruikt om de statische analyse te verbeteren (zie voorbeeld in paragraaf 4.3.2).
- Geen evaluatie van selectie-logica
  - Binnen de statische analyse wordt geen rekening gehouden met selectie-logica. Dit voorkomt een hoop complexiteit met betrekking tot het evalueren van expressies (zie paragraaf 4.3.2) Het nadeel is wel dat er mogelijk false negatives worden geïntroduceerd.

8.2 Nadelen van gekozen aanpak

Deze paragraaf bevat de nadelen die voortvloeien uit het gebruik van de de dode code analyse zoals deze binnen dit onderzoek is nagestreefd.

- Scenario-afhankelijkheid
  - Door gebruik te maken van dynamische analyse wordt er scenario-afhankelijkheid geïntroduceerd in de resultaten van de dode code analyse. De inkrimping van de grijze set is afhankelijk van hoe goed het scenario is die tijdens de dynamische analyse (runtime) wordt toegepast op de applicatie.
- Grootte van grijze set
  - Er is een situatie mogelijk waarbij de grijze set na het uitvoeren van de statische analyse alle methodes bevat behalve de entry-points, de dode code set leeg is en de werkende code set alleen de entry-points bevat. Deze situatie ontstaat wanneer er volledige onzekerheid is over de methode-aanroepen op variabelen. Volledige onzekerheid kan ontstaan op het moment dat de gedeclareerde types van alle variabelen van één type zijn (bijvoorbeeld het type Object), er geen casting plaatsvindt en dus alleen methodes worden gebruikt die op dat type kunnen worden aangeroepen. Wanneer vervolgens alle andere klassen binnen de applicatie dit type overerven en de methodes ervan overschrijven is bij geen enkele aanroep op een variabele meer met zekerheid te bepalen of de aanroep plaatsvindt op het gedeclareerde type van de variabele of één van zijn subklassen. In dit geval zal de analyse bijna volledig berusten op het gebruik van de dynamische analyse om inzicht te krijgen in de grijze set. Aangezien dynamische analyse onderhevig is aan scenario-afhankelijkheid is dit geen gewenste situatie. De kans dat deze specifieke situatie optreedt is naar mijn mening niet groot aangezien het gebruik van één specifiek type voor alle variabelen het construeren van een applicatie onnodig moeilijk maakt.
- False negatives
  - Door de manier waarop wordt omgegaan met impliciete aanroepen en selectielogica is het aannemelijk dat er in de werkende code set methode-aanroepen zijn opgenomen die runtime nooit zullen worden aangeroepen (false negatives). Een voorbeeld is een methode-aanroep in een else tak die nooit wordt bereikt. Zoals eerder aangegeven is het niet ingaan van een selectie-tak onbeslisbaar omdat dan de waarheidswaarde moet worden bepaald. Deze is in veel gevallen afhankelijk van externe factoren(zoals de gebruiker). Om als gevolg hiervan geen false positives te laten ontstaan in de dode code set is ervoor gekozen om te veronderstellen dat alle selectietakken worden doorlopen. Het gevolg hiervan is het mogelijk optreden van false negatives. De dynamische analyse kan hier ook geen uitsluitsel bieden aangezien deze alleen kan aangeven welke methodes wel worden aangeroepen[16].
- False positives
  - Als gevolg van de mogelijkheid van reflectie binnen Java (zie bijlage D) ontstaan er mogelijk false positives in de dode code set. Een mogelijke manier om dit tegen te gaan is om ook de methodes uit de dode code set te evalueren met de dynamische analyse (zie hoofdstuk 7). Uiteraard ontstaat ook hier weer het probleem dat als gevolg van de scenario-afhankelijkheid er niet met zekerheid is te zeggen dat alle false positives verwijderd zijn.
- Invloed van entry-points
  - In hoofdstuk 4.4.1 is stilgestaan bij het extraheren van de entry-points naar een applicatie. Het bepalen van de startpunten van de applicatie heeft veel invloed op het resultaat van de analyse. Op het moment dat een entry-point wordt vergeten kan dit leiden to false positives in de dode code set. Het effect van impliciete aanroepen op de effectiviteit van de analyse is dus ook zeer groot aangezien deze in feite ook entry-points naar de applicatie zijn en dus moeten worden achterhaald.

8.3 Verkregen inzicht

Het construeren van een dode code analyse voor een heterogeen systeem is een constante afweging tussen false positives en false negatives aan de ene kant en taal-afhankelijkheid aan de andere kant. In hoofdstuk 7 is in detail ingegaan op de overwegingen bij de constructie van een analyse en de voordelen van de aanpak zoals binnen dit onderzoek heeft plaatsgevonden.

Door het nastreven van een praktische implementatie van de dode code analyse is zeer duidelijk geworden dat voor het verkrijgen van relevante resultaten het ten eerste benodigd is om een grote syntactische afhankelijkheid te hebben bij de extractie van de feitelijke informatie uit de broncode. Dit houdt in dat er kennis moet zijn over de manier waarop bijvoorbeeld een methode-declaratie is vormgegeven binnen een programmeertaal. Ten

[16] Gegeven een bepaald scenario van het gebruik van de applicatie

tweede ontstaat er een semantische afhankelijkheid als gevolg van bijvoorbeeld polymorfisme of object-georiënteerdheid. Zoals reeds benoemd in hoofdstuk 2 bleek tijdens de literatuurstudie dat veel artikelen gericht zijn op een theoretische aanpak. Hierdoor wordt de problematiek van taal-afhankelijke eigenschappen niet duidelijk aangezien men voornamelijk stilstaat bij het effect van reflectie.

Ondanks het feit dat de praktische implementatie niet volledig af is kan vanuit hoofdstuk 7 worden geconcludeerd dat een combinatie van statische en dynamische analyse ervoor kan zorgen dat zowel false positives als false negatives worden tegengegaan. Daarmee is ook de onderzoeksvraag beantwoord.

Een andere belangrijke conclusie is dat als gevolg van de opdeling van de methodes binnen een applicatie in drie sets de mogelijkheid ontstaat om de taal-afhankelijkheid van de statische analyse te sturen. Dit komt doordat wanneer men minder taal-afhankelijk wordt de onzekerheid zal toenemen en als gevolg hiervan de grootte van de grijze set ook zal toenemen. Het op deze manier lichtgewicht maken van de statische analyse en het daarmee onafhankelijker maken van een programmeertaal heeft weliswaar enkele nadelen met betrekking tot de zekerheid waarmee beslissingen kunnen worden genomen maar deze onzekerheid kan via de gepresenteerde oplossing actief met dynamische analyse worden bestreden. Aangezien het uitvoeren van een dynamische analyse die per methode-aanroep een uniek bericht afdrukt hoogstens syntactische afhankelijkheden zal hebben met de onderliggende programmeertaal en geen directe semantische afhankelijkheden valt hiermee de taal-afhankelijkheid van de gehele analyse te beïnvloeden.

Op het moment dat de statische analyse minder taal-afhankelijk wordt gemaakt en de grijze set dus mogelijk groter wordt zal de dynamische analyse moeten worden gebruikt om dit te compenseren. Het gevolg is dat er meer scenario-afhankelijkheid kan ontstaan, maar het is in ieder geval niet zo dat de dynamische analyse taal-afhankelijker wordt. De dynamische analyse zal altijd dezelfde functie hebben onafhankelijk van de grootte van de grijze set.

Het is wel zo dat de scenario-afhankelijkheid van de dynamische analyse direct invloed heeft op de resultaten van de dode code analyse. Wanneer de invloed van de dynamische analyse op het resultaat van de analyse groter wordt gemaakt zal het ook steeds van groter belang worden dat er goede scenario's worden bedacht. Het voordeel van de opdeling in de drie sets is hier wel dat de effectiviteit van de scenario's kan worden gemeten door te kijken hoe de grijze set zich ontwikkelt. Wanneer er moeite is met het verzinnen van scenario's en de grijze set relatief onvoldoende in omvang afneemt ten opzichte van de grootte van de twee andere sets dan wordt de betrouwbaarheid van de conclusie die men kan trekken over de methodes die zich in de grijze set bevinden laag. In zo'n geval zou men een lagere afhankelijkheid van de dynamische analyse willen en dus een taal-afhankelijkere statische analyse. Dit blijft altijd een wisselwerking.

8.4 Validatie

De voornaamste validatie van mijn aanpak is reeds gegeven in de verschillende hoofdstukken waarbij hoofdstuk 7 in feite het belangrijkste is. Het gaat in dit geval alleen wel om een theoretische bespreking van de analyse. Om toch te kunnen nagaan wat het effect is van het gebruik van de grijze set in combinatie met de dynamische analyse in volwaardige applicaties is het nodig om de analyse praktisch toe te passen. Het is namelijk erg belangrijk om praktisch inzicht te krijgen in de verhoudingen tussen de drie sets die volgen uit de statische analyse en het effect dat kan worden bereikt met het uitvoeren van de dynamische analyse om de grijze set te verkleinen. Dit bepaald immers de zekerheid waarmee kan worden geconcludeerd of de methodes in de grijze set ook dood zijn. In hoofdstuk 8.2 is reeds stilgestaan bij een situatie waarin de toepassing van de statische analyse leidt tot een zeer grote grijze set en het is de vraag hoe vaak dit praktisch optreedt en in hoeverre de dynamische analyse hier kan helpen.

Voor de validatie is gebruikgemaakt van het opzetten van een praktische implementatie van de dode code analyse zoals besproken in hoofdstuk 5 en 6. Er zijn nog verschillende beperkingen aan deze analyse waarmee rekening moet worden gehouden bij het uitvoeren van een analyse. Deze beperkingen zijn opgenomen in bijlage F en daarnaast zijn de voorwaarden voor het gebruik van de relaties opgenomen in bijlage D. Hiermee wordt de huidige reikwijdte van de praktische toepassing bepaald.

Toepassing op testvoorbeeld
Er is een relatief eenvoudig testvoorbeeld (5 klassen, 28 methodes) gemaakt waarop de dode code analyse is toegepast. In bijlage G is aangegeven hoe dit voorbeeld wordt behandeld door de dode code analyse. Het is nog niet volledig mogelijk om de analyse automatisch toe te passen (zie bijlage F). Op het moment dat de geëxtraheerde relaties door het Rscript worden geanalyseerd ontstaat wel het verwachte resultaat. De dode code set bevat inderdaad vijf methodes die niet worden gebruikt binnen de applicatie en de grijze set bevat na de

dynamische analyse nog vier methodes die vervolgens ook als niet gebruikt kunnen worden beschouwd. Dit levert in totaal negen methodes op die inderdaad binnen de applicatie niet worden gebruikt. Gezien het relatief eenvoudige karakter (geen reflectie of problemen door impliciete aanroepen) van de voorbeeld -applicatie treden er geen false positives op.

Het is wel zo dat niet wordt gedetecteerd dat de interface TemporaryEmployee ongebruikt is. Dit heeft te maken met het feit dat eerder is besloten om de methode-declaraties van interfaces niet te extraheren. Naar aanleiding hiervan is besloten om dit te veranderen in de implementatie en de methode-declaraties binnen interfaces als abstracte methodes te beschouwen. Daarnaast zal in het Rscript de mogelijkheid moeten worden opgenomen om te detecteren of binnen een bronbestand alle methodes als ongebruikt zijn bestempeld. In een dergelijk geval is namelijk het bronbestand als geheel ongebruikt. Dit laatste is nog niet gerealiseerd (zie beperking DC-5 in bijlage F).

Voor de vergelijking met een andere dode code analyse is gebruik gemaakt van RevJava[W18] versie 0.8.5. Zoals besproken in paragraaf 5.1.1 kunnen met deze tool van het SERC onder andere ongebruikte methodes worden gedetecteerd. Van de negen dode methodes in het testvoorbeeld worden er door de tool slechts drie gevonden. Eén van de ontbrekende dode methodes is een private methode “isGoodRobot” die door een public methode “calculateSalary” wordt aangeroepen terwijl deze public methode zelf nooit wordt gebruikt. Het vreemde is dat RevJava de methode “calculateSalary” wel als dode code ziet maar de “isGoodRobot” niet als dood herkent. Daarnaast wordt een ongebruikte “toString” niet als zodanig herkend. Dit zal te maken hebben met het feit dat de methode “toString” net zoals de “finalize” methode als standaard entry-point worden beschouwd binnen RevJava. RevJava retourneert wel de ongebruikte interface. Daarentegen retourneert RevJava ook de abstracte methode salary als ongebruikt terwijl deze in andere klassen worden overschreven en gebruikt. Dit laatste gebeurd als gevolg van beperking DC-5 in de praktische implementatie van de dode code analyse ook nog.

In hoofdstuk 2 zijn nog verschillende andere validatie mogelijkheden geopperd voor de dode code analyse. Voordat deze ook echt toepasbaar zijn moet de praktische implementatie verder zijn ontwikkeld.

Voor het controleren van de effecten van wijzigingen in de praktische implementatie van de feiten-extractiefase en de verrijkingsfase is gebruik gemaakt van vooraf geconstrueerde voorbeelden op basis van constructies die binnen de Meta -omgeving zijn aangetroffen. Voor de Rscript fase (waarin de met de hand aangevulde relaties kunnen worden gebruikt om de opdeling in de drie sets te maken) is gekozen voor een automatische test. Door middel van assert statements wordt gecontroleerd of de functionaliteit binnen het Rscript klopt en dus de opdeling volgens de afspraken in paragraaf 4.3.2 verloopt. Hiervoor wordt gebruik gemaakt van het testvoorbeeld als invoer.

Metrieken volgend uit de praktische implementatie
Ondanks dat de dode code detectie als gevolg van de praktische beperkingen niet direct kan worden uitgevoerd op applicaties zoals de Meta-omgeving kan toch enig inzicht worden verschaft in hoeverre de dode code analyse is ontwikkeld en hoe ver men afzit van een volledige implementatie. Hiervoor is een Rscript gemaakt die de grootte van enkele relaties meet. Deze informatie geeft ten eerste inzicht in de onderzochte applicatie en geeft daarnaast ook inzicht in hoeverre de implementatie af is.

De grootste “bottleneck” die ervoor zorgt dat de dode code detectie nog niet toepasbaar is op grotere applicaties is de beperking die is gelegd op overloading en de beperkingen bij het achterhalen van de types waarop een methode aanroep plaats vindt. Het achterhalen en daarmee construeren van de METHOD_CALL relatie gebeurd in de tweede fase. Zoals in bijlage F valt te lezen worden aanroepen waarbij gebruik wordt gemaakt van field-references of static methodes van klassen niet omgezet naar een METHOD_CALL maar blijven in de UNRESOLVED_CALL. Als gevolg hiervan kunnen belangrijke aanroepen worden gemist en code als dood worden beschouwd terwijl dit in feite niet zo is.

Wat kan worden gemeten zijn de standaarmetrieken zoals het aantal klassen, interfaces en aantal methode-declaraties. Ook kan worden achterhaald hoeveel methode-aanroepen er zijn in de applicatie en hoeveel daarvan door de tweede fase (feiten-verrijking) worden omgezet naar een daadwerkelijke METHOD_CALL Deze metrieken geven inzicht in hoeverre de praktische implementatie verwijderd is van een volledige implementatie door de verhoudingen te meten tussen het totale aantal methode-aanroepen die in de eerste fase zijn gedetecteerd en het aantal methode-aanroepen die hiervan kunnen worden herleid naar het gedeclareerde type waarop ze worden uitgevoerd.

Voor het verzamelen van bovenstaande metrieken is als uitgangspunt ten eerste de voorbeeldapplicatie gebruikt, daarnaast de Meta-omgeving en het ApiGen[17] pakket. Met behulp van het programma LOCC[18] zijn allereerst per pakket verschillende metingen gedaan om een indruk van de grootte van de pakketten te krijgen. Hieronder zijn deze resultaten opgenomen die zijn geëxtraheerd met behulp van LOCC.

Tabel 4 Metrieken van voorbeeldapplicatie geëxtraheerd met LOCC

	Lines Of Code	*Classes*	*Interfaces*	*Methods*
Totaal	142	5	1	28

Tabel 5 Metrieken van Meta-studio geëxtraheerd met LOCC

	Lines Of Code	*Classes*	*Interfaces*	*Methods*
Totaal	10755	166	3	1535

Tabel 6 Metrieken van ApiGen geextraheerd met LOCC

	Lines Of Code	*Classes*	*Interfaces*	*Methods*
Totaal	8557	68	3	985

Na het bepalen van de metrieken met LOCC kunnen met behulp van het Rscript onderstaande metrieken uit de eerste feiten-extractie worden bepaald en worden vergeleken met de resultaten uit LOCC.

Tabel 7 Metrieken van voorbeeldapplicatie volgende uit feiten-extractie fase

	Unresolved Calls	*Constructor Calls*	*Local variables*	*Field declarations*	*Classes*	*Interfaces*	*Methods*
Totaal	22	8	6	7	5	1	28

Tabel 8 Metrieken van Meta-studio volgende uit feiten-extractie fase

	Unresolved Calls	*Constructor Calls*	*Local variables*	*Field declarations*	*Classes*	*Interfaces*	*Methods*
Totaal	3432	692	854	339	187	3	1563

Tabel 9 Metrieken van ApiGen volgende uit feiten-extractie fase

	Unresolved Calls	*Constructor Calls*	*Local variables*	*Field declarations*	*Classes*	*Interfaces*	*Methods*
Totaal	4013	237	854	162	69	3	986

Wat meteen opvalt na het uitvoeren van de eerste fase is dat de metrieken welke zijn achterhaald met LOCC met betrekking tot het aantal methodes en klassen niet hetzelfde is als gevonden met de praktische implementatie van de feiten-extractie die binnen dit onderzoek is gemaakt. Het verschil in het aantal klassen is te verklaren door het feit dat LOCC geen innerklassen meeneemt in de berekening. In ApiGen bevindt zich één innerklasse in de klasse apigen.adt.Type welke niet door LOCC wordt meegenomen in de berekening. In de Meta-omgeving bevinden zich verschillende innerklasses in de broncode (voornamelijk door event-handling) zoals onder andere in metastudio.MetaStudio. Deze wordt ook niet herkend door LOCC. Binnen de praktische implementatie van de feiten-extractie binnen dit onderzoek worden de innerklassen uiteraard wel meegenomen. Dit effect werkt door op het aantal gevonden methode-declaraties binnen de klassen die verschilt doordat de methode-declaraties in de innerklassen door LOCC niet worden meegerekend en in de praktische implementatie wel worden achterhaald.

Als gevolg van beperking FE-6 (zie bijlage F) worden er twee aanroepen binnen het ApiGen pakket niet automatisch omgezet. Deze moeten achteraf met de hand worden verbeterd. Bij de twee andere applicaties treedt deze beperking niet op. Deze beperking en de andere beperkingen uit bijlage F zijn uiteraard puur het gevolg van

[17] http://www.cwi.nl/htbin/sen1/twiki/bin/view/SEN1/ApiGen
[18] http://csdl.ics.hawaii.edu/Tools/LOCC/

de beperkte praktische implementatie en zijn geen tekortkoming in de theorie met betrekking tot de dode code analyse.

Door de vergelijking tussen de metrieken van LOCC en de metrieken volgende uit de feiten-extractie is dus gevalideerd of de praktische implementatie van de feiten-extractie de klassen, interfaces en de methode-declaraties correct[19] extraheerd. Afgezien van het verschil als gevolg van de innerklasses die niet in LOCC worden meegerekend komen de waarden overeen. De andere metrieken, zoals het aantal methode-aanroepen, zijn alleen met de hand gevalideerd voor de voorbeeldapplicatie aangezien hiervoor niet een tool is gevonden voor validatie.

Een zeer interessant gegeven voor de dode code analyse is het aantal methode-aanroepen waarvan het gedeclareerde type waarop deze wordt uitgevoerd kan worden bepaald. Bijvoorbeeld wanneer de aanroep `a.getIt().getThat()` wordt uitgevoerd moet ten eerste het type van `a` worden bepaald en vervolgens het returntype van `getIt` om het gedeclareerde type te bepalen waarop `getThat` wordt aangeroepen. Als gevolg van het ontbreken van informatie over de methode-aanroepen is de dode code analyse nog niet volledig uit te voeren omdat er false positives kunnen ontstaan doordat er methode-aanroepen ontbreken als gevolg van de beperking in de omzetting (zie bijlage F).

In onderstaande tabellen zijn per pakket de metrieken aangegeven die ontstaan na het verrijken van de relaties met behulp van de feitenverrijking iteratie 11.

Tabel 10 Metrieken van voorbeeldapplicatie volgende uit verrijkings-fase

	Resolved Calls	*Amount of Calls*	*Resolved Calls / Amount of Calls*	*Percentage resolved*
Totaal	23	30	23 / 30	76.67

Tabel 11Metrieken van Meta-studio volgende uit verrijkings-fase

	Resolved Calls	*Amount of Calls*	*Resolved Calls / Amount of Calls*	*Percentage resolved*
Totaal	2765	4124	2765 / 4124	67.05

Tabel 12 Metrieken van ApiGen volgende uit verrijkings-fase

	Resolved Calls	*Amount of Calls*	*Resolved Calls / Amount of Calls*	*Percentage resolved*
Totaal	3240	4250	3240 / 4250	76.24

Het totale aantal methode-aanroepen is berekend door het aantal unresolved calls en het aantal constructor calls bij elkaar op te tellen. Deze vertegenwoordigen alle aanroepen die binnen het programma worden gedaan en kunnen dus worden gebruikt om de verhouding tussen het aantal herleide methode-aanroepen en het totale aantal methode-aanroepen te bepalen.

Uit de bovenstaande tabellen is op te maken dat in de drie gebruikte pakketten gemiddeld ruim zeventig procent van alle methode-aanroepen kunnen worden herleid naar het gedeclareerde type waarop deze worden uitgevoerd. Een groot gedeelte van het aantal methode-aanroepen dat niet kan worden herleid zal worden veroorzaakt door methode-aanroepen die een onderdeel zijn van bibliotheken zoals de standaard operaties die mogelijk zijn op een String. Het andere gedeelte wordt veroorzaakt door de beperkingen als gevolg van de praktische implementatie. Aangezien in de praktische implementatie helaas geen onderscheid is gemaakt in deze twee vormen van unresolved calls kunnen deze niet worden bepaald. Het is wel zo dat bij de voorbeeld-applicatie alle unresolved-calls, aanroepen zijn naar de println functie die gedefinieerd is binnen de Java bibliotheek.

Binnen de twee andere pakketten, ApiGen en de Meta-studio, is het duidelijk dat er ook unresolved calls zijn als gevolg van de beperkingen van de implementatie. Het gevolg hiervan is dat toepassing van het Rscript voor de bepaling van de grijze code set, werkende code set en dode code set geen betrouwbare resultaten zal geven aangezien er methode-aanroepen niet zijn herleid die wel van belang zijn en waardoor false positives ontstaan. Bij toepassing van het Rscript op de verrijkte feiten van de ApiGen bleek ook dat dit het geval was. De toepassing van het Rscript op de voorbeeldapplicatie verloopt wel volgens verwachtingen en levert het gewenste resultaat.

[19]Correct wil hier zeggen dat ze overeenkomen met de informatie die een andere tool uit de code extraheert.

Samenvattend komt het er op neer dat als gevolg van de beperkingen van de praktische implementatie de dode code analyse nog niet op een groot pakket kan worden uitgevoerd. Er wordt reeds een groot gedeelte van de methode-aanroepen correct herleid naar het gedeclareerde type waarop deze wordt uitgevoerd maar dit percentage moet nog worden verhoogd om de analyse correct te laten verlopen. De dode code analyse werkt wel volgens verwachting op de voorbeeld-applicatie aangezien hier geen problemen volgen uit de beperkingen. Dit geeft samen met de bovenstaande metrieken de indruk dat met het wegwerken van de beperkingen in de praktische implementatie de dode code analyse ook toepasbaar moet zijn op de grotere applicaties zoals ApiGen en de Meta-omgeving.

8.5 Future work

In deze paragraaf zal kort worden stilgestaan bij enkele zaken die in een mogelijk vervolgonderzoek zouden kunnen worden nagestreefd.

Opnemen van beperkte flow-insensitive points-to analysis[18]
Gegeven de informatie die zich reeds in het relationele model bevindt kan op relatief eenvoudige wijze een vorm van points-to analysis worden gebruikt bij het bepalen van de types waarnaar een variabele kan verwijzen. In [18] wordt per variabele achterhaald welke Constructors er direct of indirect aan zijn toegekend. Dit kan niet met het huidige relationele model aangezien er dan ook een ASSIGNMENT relatie nodig is. Wat wel kan is gewoonweg achterhalen van welke klassen één of meerdere Constructors worden aangeroepen. Deze informatie kan worden gebruikt om te bepalen naar welke subklassen en superklassen een variabele in ieder geval niet kan wijzen omdat er nooit een Constructor-aanroep voor dat type is geweest. Toch is ook dit weer niet zo eenvoudig aangezien hierbij ook moet worden gekeken naar Casting en impliciete Constructor aanroepen door de virtual machine (zoals bij het gebruik van types bij exceptie-afhandeling). Het is de vraag of het creëren van deze extra onzekerheid en taal-afhankelijkheid opweegt tegen het voordeel van een mogelijk kleinere grijze set.

Bepaling entry-points
Voor het kunnen uitvoeren van een dode code analyse is reeds eerder aangegeven dat het bepalen van de entry-points zeer belangrijk is. Het ontbreken van een entry-point leidt tot de mogelijkheid van false-positives binnen de dode code set. Aangezien er vanuit dit onderzoek geen praktische implementatie is gerealiseerd voor de extractie hiervan vanuit Toolbus Scripts zou dit kunnen worden nagestreefd. Hierbij moet ook worden gekeken in hoeverre de bestaande mogelijkheid tot het genereren van een Java interface vanuit de Toolbus Scripts hiervoor kan worden gebruikt.

Java bibliotheek parseren
Om het mogelijk te maken overloading volledig te implementeren is het nodig om de return types te weten van methodes binnen de door Java aangeboden functionaliteit. Door de broncode van deze standaard bibliotheek vooraf te parseren zou het mogelijk zijn om hiermee rekening te houden. Daarnaast zal het parseren van de bibliotheek ervoor zorgen dat er minder problemen zijn met betrekking tot impliciete aanroepen. De meeste impliciete aanroepen zijn namelijk binnen de bibliotheek een expliciete aanroep (het gebruik van native code gooit hier roet in het eten). Het principe van het vooraf parseren van de Java bibliotheek wordt toegepast bij Tacle[W4].

9 Evaluatie

Dit hoofdstuk vormt het sluitstuk van deze scriptie. Hierin zal allereerst worden ingegaan op de in hoofdstuk 3 gestelde hypothese waarna in de tweede paragraaf kort zal worden teruggeblikt op het verloop van het onderzoek.

9.1 Aanname van hypothese

Zoals uit het hoofdstuk over de resultaten blijkt en in de voorgaande hoofdstukken is besproken kan het type-analyse probleem en de daarbij behorende complexiteit zoals bij points-to analysis[18] worden beperkt door het gebruik van de opdeling in drie sets en de combinatie van statische met dynamische analyse. In hoofdstuk 7 is aangegeven dat door het combineren van een statische analyse met een dynamische analyse zowel de false positives als de false negatives worden tegengegaan binnen de dode code analyse. Dit is in principe hetzelfde wat men probeert te bereiken op het moment dat de statische analyse taal-afhankelijker wordt gemaakt met een geavanceerder type-analyse systeem aangezien men dan nastreeft om meer zekerheid te hebben bij beslissingen waar polymorfisme een rol speelt. De hypothese die in hoofdstuk 3 is gesteld kan op basis van het onderzoek en zijn resultaten ook worden aangenomen maar wel met de duidelijke kanttekening dat het lichtgewicht maken van de statische analyse en het daarmee versterken van de invloed van de dynamische analyse invloed heeft op de zekerheid van de resultaten uit de analyse. Wanneer namelijk tijdens de dynamische analyse de grijze set niet voldoende in omvang afneemt wordt het trekken van conclusies over de grijze set en het wel of niet dood zijn van deze methodes bemoeilijkt.

Evengoed zal het met een geavanceerde flow (in)sensitive points-to analysis[18] toch ook altijd zo blijven dat er onzekerheid blijft. In [18] wordt reeds aangegeven dat reflectie altijd een probleem zal blijven bij het bepalen van het type waarnaar een variabele wijst. Daarnaast wordt in het artikel even voor het gemak vergeten dat er ook nog invloeden zijn als gevolg van de onderliggende library's op de vaststelling van de types van variabelen. Het lijkt een kenmerk van artikelen die geavanceerde type-analyse systemen of dode code analyse beschrijven dat de onderliggende problematiek in veel gevallen niet geheel wordt onderkend vanwege de complexiteit van de technieken. Door het nastreven van een praktische implementatie is getracht dit effect binnen dit onderzoek niet te laten ontstaan en vandaar ook de bespreking van de invloeden van specifiek Java constructies op de resultaten van de analyse.

9.2 Terugblik

Gezien de taal-afhankelijkheid die uiteindelijk toch is ontstaan is de vraag of één dode code analyse voor een heterogeen systeem moet worden nagestreefd of dat er niet beter van bestaande tooling gebruik kan worden gemaakt die op een specifieke programmeertaal is gericht. Wanneer dit inzicht eerder in het project naar voren was gekomen dan had de focus minder gelegen op een eigen implementatie. Toch blijft zoals eerder benoemd het voordeel van een eigen implementatie dat de problematiek hierdoor veel duidelijker wordt. Dit was zeer van belang bij het beoordelen van de mogelijkheid tot taal-onafhankelijkheid en dus de beantwoording van de onderzoeksvraag.

De keuze voor het gebruik van Rscript voor de praktische implementatie van de dode code analyse zou achteraf misschien minder te rechtvaardigen zijn dan vooraf. Er is dan wel het voordeel dat de set van operaties op de geëxtraheerde informatie beperkt is maar dit is ook meteen een duidelijk nadeel. Bij het implementeren van de analyse bleek op meerdere punten dat door het ontbreken van een selectiemechanisme (if/then/else) er veel tijd moest worden gestoken in bijvoorbeeld het implementeren van een functie die de eerstvolgende superklasse vindt (zie broncode). Ook het niet ondersteunen van recursieve type-definities of recursieve-functies was een beperking waardoor lang niet alle functionaliteit makkelijk was te implementeren.

Een groot voordeel van mijn iteratieve aanpak bij de realisatie van de praktische implementatie was dat ik vanuit het resultaat (welke uit de analyse met Rscript moest komen) heb teruggewerkt naar de benodigde feitenextractie. Hierdoor kon rekening worden gehouden met de beperkte functionaliteit binnen Rscript, wat van invloed was op de vorming van het relationele model (bijlage C en D).

Geraadpleegde literatuur

[1] Ball, Thomas en James R. Larus (2000) Using Paths to Measure, Explain and Enhance Program Behavior, *Computer, Volume 33*, July 2000, pagina 57-65, http://doi.ieeecomputersociety.org/10.1109/2.869371

[2] Bergstra, J.A. en P. Klint (1998) The discrete time ToolBus – A software coordination architecture, *Science of computer programming*, Volume 31, July 1998, pagina 205-229, http://dx.doi.org/10.1016/S0167-6423(97)00021-X

[3] Beyer, Dirk e.a. (2004) An Eclipse Plug-in for Model Checking, *12th IEEE International Workshop on Program Comprehension*, pagina 251, http://doi.ieeecomputersociety.org/10.1109/WPC.2004.1311069

[4] Blech, Jan Olaf e.a (2005) Formal Verification of Dead Code Elimination in Isabelle/HOL, *Conference on Software Engineering and Formal Methods*, September 2005, http://www.info.uni-karlsruhe.de/~glesner/publications.html

[5] Brand, M.G.J. van den e.a. (2001) The ASF+SDF Meta-Environment: a Component-Based language development environment, *Proceedings of the 10th International Conference on Compiler Construction*, 2001, pagina 365-370, http://homepages.cwi.nl/~paulk/publications/all.html

[6] Brand, M.G.J. van den e.a. (2002) Compiling Language Definitions: The ASF+SDF Compiler, *ACM Transactions on Programming Languages and Systems (TOPLAS)*, Volume 24, No.4, 2002, pagina 334-368, http://doi.acm.org/10.1145/567097.567099

[7] Chen, Yih-Farn e.a. (1990) The C Information Abstraction System, *IEEE transactions on software engineering*, volume 16, no. 3, Maart 1990, pagina 325-334, http://dx.doi.org/10.1109/32.48940

[8] Chen, Yih-Farn e.a. (1998) A C++ data model supporting reachability analysis and dead code detection, *IEEE transactions on software engineering*, volume 24, No. 9, September 1998, http://doi.ieeecomputersociety.org/10.1109/32.713323

[9] Cifuentes, Cristina en Antoine Fraboulet (1997) Intraprocedural Static Slicing of Binary Executables, *Proceedings of the International Conference on Software Maintenance*, pagina 188, http://portal.acm.org/citation.cfm?coll=GUIDE&dl=GUIDE&id=656038

[10] Ernst, Michael D. (2003) Static and dynamic analysis: synergy and duality, *WODA 2003: ICSE Workshop on Dynamic Analysis, 2003*, pagina 24-27, http://pag.csail.mit.edu/pubs/staticdynamic-woda2003-abstract.html

[11] Feigen, Lawrence e.a. (1994) The revival transformation, *Proceedings of the 21st ACM SIGPLAN-SIGACT symposium on Principles of programming language*, 1994, pagina 421-434, http://portal.acm.org/citation.cfm?doid=174675.178043

[12] Frank Tip (1995) A survey of program slicing techniques, *Journal of Programming Languages*, 1995, pagina 121-189, http://citeseer.ist.psu.edu/tip95survey.html

[13] Gosling, James, Bill Joy, Guy Steele en Gilad Bracha (2000) *The Java Language Specification (2nd edition)*, ISBN 0-201-31008-2, Addison-Wesley, http://java.sun.com/docs/books/jls/download/langspec-2.0.pdf

[14] Jong, H. de en P. Klint (2003) ToolBus: the Next Generation, *First International Symposium, FMCO 2002*, Leiden, The Netherlands, November 2002, pagina 220-241, http://homepages.cwi.nl/~paulk/publications/all.html

[15] Lin, Yaun e.a. (2003) Completeness of a Fact Extractor, *Proceedings of the 10th Working Conference on Reverse Engineering*, 2003, pagina 196, http://doi.ieeecomputersociety.org/10.1109/WCRE.2003.1287250

[16] Lindhom, Tim en Frank Yelling, *The Java Virtual Machine Specification (2nd edition)*, ISBN 0-201-43294-3, Addison-Wesley, http://java.sun.com/docs/books/vmspec/2nd-edition/html/VMSpecTOC.doc.html

[17] Rountev, Atanas (2004) Static and dynamic analysis of call chains in java, *Proceedings of the 2004 ACM SIGSOFT international symposium on Software testing and analysis*, 2004, pagina 1-11, http://doi.acm.org/10.1145/1007512.1007514

[18] Sridharan, Manu e.a. (2005) Demand-driven Points-to Analysis for Java, *OOPSLA '05*, 2005, pagina 59-76, http://portal.acm.org/citation.cfm?doid=1094811.1094817

[19] Whaley, John (2001) Partial Method Compilation using Dynamic Profile Information, *Proceedings of the 16th ACM SIGPLAN conference on Object oriented programming, systems, languages, and applications, 2001*, pagina 166-179, http://doi.acm.org/10.1145/504282.504295

[20] Willegen, J.J.H. van (2006) Code tuning bij een component gebaseerd systeem, *College Software Construction van de Master Software Engineering*, 2006, beschikbaar via blackboard.

Geraadpleegde websites

[W1] Unreachable code - Wikipedia, the free encyclopedia
http://en.wikipedia.org/wiki/Unreachable_code

[W2] MetaEnvironment
http://www.cwi.nl/htbin/sen1/twiki/bin/view/SEN1/MetaEnvironment

[W3] Emma – Code coverage tool
http://emma.sourceforge.net/index.html

[W4] Call-graph analysis
http://presto.cse.ohio-state.edu/tacle/TACLE_Download/tacle.html

[W5] Virtual machine met analyse mogelijkheden
http://joeq.sourceforge.net/

[W6] Jax
http://www.research.ibm.com/jax/

[W7] CPPX – C++ fact extractor
http://swag.uwaterloo.ca/~cppx/

[W8] C & C++ Analysis Eclipse plugin
http://embedded.eecs.berkeley.edu/blast/

[W9] FoCuS – Een code coverage tool voor Java, C en C++
http://www.alphaworks.ibm.com/tech/focus

[W10] Informatie over overloading en daaraan gerelateerde polymorfisme
http://www.developer.com/java/other/article.php/966001

[W11] Informatie over verschillende aspecten met betrekking tot Java
http://mindprod.com/jgloss/jgloss.html

[W12] Informatie over overloading
http://www.particle.kth.se/~lindsey/JavaCourse/Book/Part1/Java/Chapter03/overloading.html

[W13] JIT compiler informatie
http://en.wikipedia.org/wiki/Just-in-time_compilation

[W14] Soundess
http://en.wikipedia.org/wiki/Soundness

[W15] Cflow
http://www.gnu.org/software/cflow/

[W16] Byte code engineering lab
http://jakarta.apache.org/bcel/index.html

[W17] JAD decompiler
http://www.kpdus.com/jad.html

[W18] RevJava
http://www.serc.nl/people/florijn/work/designchecking/RevJava.htm

A – Beschikbare relationele informatie

In deze bijlage zullen de relaties, zoals deze met de vooraf beschikbare Java extractie (Relational Calculus 3.0) konden worden geëxtraheerd, worden besproken. De opmerkingen geven aan in hoeverre de relatie voldoet aan zijn definitie.

* loc = locatie van item in bronbestand inclusief de volledige bestandsnaam

<FILE, rel[Filename, loc]>
- Filename = Naam van het bronbestand inclusief pad vanaf waar de feitenextractie plaatsvindt

<INTERFACE, rel[Interfacename, loc]>
- Interfacename = de naam van een interface die in het bronbestand is gedeclareerd
  - Detecteert geen interne interfaces in interfaces
  - Geeft geen fully qualified name aan de interface waardoor ambiguïteit ontstaat

<PACKAGE, rel[PackageName, loc]>
- PackageName = de volledige naam van de package waartoe het bronbestand en de daarin gedeclareerde klassen en interfaces behoren

<CLASS, rel[ClassName, loc]>
- ClassName = de naam van een klasse gedefinieerd in het bronbestand
  - Geeft geen fully qualified name aan een klasse waardoor ambiguïteit ontstaat

<METHOD, rel[MethodName, loc]>
- MethodName = de naam van een methode welke is gedeclareerd in het bronbestand
  - Bij een innerclass in een methode worden de methoden van de innerclass niet opgenomen
  - In een anonymous klasse worden de gedeclareerde methoden niet opgenomen
  - Methode-declaraties van een interface worden niet opgenomen.
  - Geen fully qualified name door het ontbreken van informatie over argumenten en de klasse waarin de methode is opgenomen

<CLASS_USE, rel[ClassUseName, loc]>
- ClassUseName = de naam van een klasse die wordt gebruikt binnen het bronbestand doordat er een instantie van wordt gecreëerd door middel van de new operatie.
  - Geen fully qualified naam waardoor ambiguïteit ontstaat.

<VARIABLE_DECL, rel[VarName, loc, TypeName]>
- VarName = De naam van de variabele zoals deze is gedeclareerd in het bronbestand
- TypeName = De naam van het type van de variabele zoals deze voor de variabele staat
  - Geen fully qualified name waardoor ambiguïteit ontstaat.

<FIELD_DECL, rel[FieldName, loc, TypeName]>
- FieldName = De naam van het veld dat wordt gedeclareerd in het bronbestand
- TypeName = De naam van het type van het veld
  - Geen fully qualified name waardoor ambiguïteit ontstaat.

<EXTENDS_CLASS, rel[SubclassName, SuperclassName]>
- SubclassName = De naam van de klasse die een klasse overerft
- SuperclassName = De naam van de klasse die als ouder dient van SubclassName
  - Geen fully qualified names waardoor ambiguïteit ontstaat

<IMPLEMENTS, rel[ClassName, InterfaceName]>
- ClassName = De naam van de klasse die de interface implementeert
- InterfaceName = De naam van de interface die wordt geïmplementeerd in ClassName
  - Geen fully qualified name waardoor ambiguïteit ontstaat

<EXTENDS_INTERFACE, rel[SubInterfaceName, SuperInterfaceName]>
- SubInterfaceName = De naam van de interface die een interface overerft
- SuperInterfaceName = De naam van de interface die als ouder dient
  - Geen fully qualified name waardoor ambiguïteit ontstaat

<METHOD_CALL, rel [MethodName, loc]>

- MethodName = De naam van de methode welke wordt aangeroepen
  - Aanroepen naar methodes van de superklasse door middel van `super.method()` worden niet als method call beschouwd.
  - Een method call welke als argument wordt gebruikt voor een andere methode-aanroep wordt niet als methode-aanroep gezien. Bijvoorbeeld bij System.out.println(e.nextElement()) moet nextElement als methode-aanroep worden opgenomen maar dit wordt niet gedetecteerd.
  - Aanroepen naar de constructor worden niet opgenomen als methode-aanroep.
  - Geen fully qualified name waardoor ambiguïteit ontstaat.

B – Type definities voor relationeel model

In deze bijlage zijn de definities opgenomen van relaties die als type zijn gebruikt binnen het relationele model voor de dode code analyse.

Locatie informatie

Naam	`loc`
Definitie	`area-in-file(FileName, area(begin-line, begin-column, end-line, end-column, offset, length ) )`
Onderdelen	• FileName = de locatie van het bronbestand in de vorm van het volledige pad inclusief bestandsnaam • begin-line = Verwijzing naar regelnummer in het bronbestand vanaf waar de gedefinieerde locatie loopt • begin-column = Verwijzing naar de kolom vanaf waar de gedefinieerde locatie start • end-line = Verwijzing naar regelnummer in het bronbestand tot aan waar de gedefinieerde locatie loopt • end-column = Verwijzing tot aan waar de kolom van de gedefinieerde locatie loopt
Voorbeeld	area-in-file ("./employee/Boss.java" , area (15 , 1 , 17 , 2 , 267 , 66))
Doel	Met behulp van deze informatie is het mogelijk om een vierkant gebied binnen een bronbestand te definiëren. Op deze manier kan dus bijvoorbeeld de locatie van een methode worden aangegeven binnen een bronbestand. Dit is nodig om op een eenvoudige manier te kunnen verwijzen naar de originele broncode wanneer er dode code is gedetecteerd.
Voorwaarden	Met behulp van `AddPosInfo` moet de locatie informatie aan de parseerboom worden toegevoegd voordat de relatie-informatie wordt geëxtraheerd. Dit is een standaard operatie welke beschikbaar is binnen de ASF+SDF omgeving.

Modifier set

Naam	`MODIFIERS`
Definitie	`{Modifier, ...}`
Onderdelen	• Modifier = Een optie welke kan worden opgenomen in de declaratie van een type, methode of een variabele. ○ Modifier is een element uit {“public”, “protected”, “private”, “static”, “abstract”, “final”, “native”, “synchronized”, “transient”, “volatile”}
Voorbeeld	{“public”, “static”}
Doel	De modifiers dienen verschillende doelen. De access modifiers hebben tot doel om te kunnen achterhalen of een methode wel aangeroepen kan worden. Zo kan een private methode die in de klasse zelf niet wordt aangeroepen nooit worden gebruikt en dus worden geregistreerd als dood. Het keyword “abstract” is van belang om ervoor te zorgen dat de abstracte definitie niet als een daadwerkelijke methode implementatie wordt gezien. Dit zou problemen kunnen opleveren bij het bepalen van de methodes waarnaar een methode-aanroep verwijst.
Voorwaarden	In het geval van een anonieme type declaratie zal de set het element “anonymous” bevatten.

Type relatie

Naam	`TYPE_NAME`
Definitie	`rel[Name, Index]`
Onderdelen	• Name = een specifiek onderdeel van de gehele type beschrijving • Index = de positie van het onderdeel in de volledige type beschrijving

Naam	`TYPE_NAME`
Voorbeeld	{ <”metastudio”,1>, <”data”,2>, <”Module”,3> }
Doel	Deze relatie maakt het mogelijk om typenamen die uit subdelen bestaan op te delen. Dit houdt in dat bijvoorbeeld de type aanduiding “`metastudio.data.Module`” wordt opgedeeld zoals aangegeven in het voorbeeld. De reden dat deze opdeling nodig is komt voort uit de beperking binnen Rscript met betrekking tot de String manipulatie. Er zijn binnen Rscript zeer beperkte operaties voor het manipuleren van Strings.
Voorwaarden	De type-aanduiding is alleen volledig wanneer het bronbestand waarin het betreffende type wordt gedefinieerd ook beschikbaar is en de informatie vanuit dit bronbestand in de CLASS, PACKAGE en IMPORT relatie aanwezig is. Het gevolg hiervan is dat type-aanduidingen vanuit bibliotheken niet kunnen worden bepaald en hierop geen dode code analyse kan worden gedaan. Wanneer er sprake is van een type-aanduiding in de vorm van een array dan wordt de array-informatie achterwege gelaten. Dat wil zeggen dat de informatie `String[]` wordt omgezet naar {<”String”,1>}. Dit is in Java gerechtvaardigd aangezien men bij dode code analyse alleen in het type waarnaar verwezen wordt is geïnteresseerd. Wanneer er sprake is van een anoniem type wordt het laatste element gelijk aan het regelnummer waar de declaratie van het anonieme type plaatsvond.

Argument relatie

Naam	`ARGUMENTS`
Definitie	`rel[ArgumentType, Name, Index]`
Onderdelen	● ArgumentType = de volledige type aanduiding van het argument ○ `ArgumentType` is van het type `TYPE_NAME` ● Name = de naam van het argument ● Index = de positie van het argument in de lijst met argumenten
Voorbeeld	{ < {<”String”, 1>}, “name”, 1>, <{<”aterm”,1>, <”AtermFactory”,2>},”factory”, 2>}
Doel	Deze relatie maakt het mogelijk om de argumenten van een methode te beschrijven. Het biedt de mogelijkheid om het type van de argumenten, de volgorde van de argumenten en het aantal argumenten te beschrijven. Dit is nodig voor de unieke aanduiding van een methode. In het voorbeeld is sprake van twee argumenten waarbij het eerste van het type `String` is en het tweede van het type `aterm.AtermFactory`. De naam van de argumenten is nodig in verband met het bepalen van methode-aanroepen op de argumenten binnen de gedeclareerde methode.
Voorwaarde	(geen)

Aanroep argument relatie

Naam	`CALL_ARGUMENTS`
Definitie	`rel[ArgumentType, Index]`
Onderdelen	● ArgumentType = de volledige type aanduiding van het argument ○ `ArgumentType` is van het type `TYPE_NAME` ● Index = de positie van het argument in de lijst met argumenten
Voorbeeld	{ < {<”Vector”, 1>} , 1>}
Doel	Deze relatie maakt het mogelijk om de argumenten te beschrijven die via een methode-aanroep worden doorgegeven. Deze argumenten hebben geen naam.
Voorwaarde	(geen)

Relatie voor niet herleidbare argument types

Naam	`UNRESOLVED_ARGUMENTS`
Definitie	`rel[ArgumentType, Index]`
Onderdelen	• ArgumentType = aanduiding van het argument ○ `ArgumentType` is van het type `TYPE_NAME` • Index = de positie van het argument in de lijst met argumenten
Voorbeeld	{ < {<"long", 1>} , 1>}
Doel	Met deze relatie is het mogelijk om de argumenten te beschrijven die aan een methode-aanroep worden meegegeven. Bij de eerste feitenextractie fase kunnen de argumenten niet allemaal direct worden geëvalueerd omdat ze afhankelijk kunnen zijn van returntypes van andere methode-aanroepen of types van variabelen. Deze informatie is benodigd om de uiteindelijke METHOD_CALL relatie te vormen.
Voorwaarde	Op het moment dat het argument een literal (long, int,enz.) is zal deze als zodanig als ArgumentType worden opgenomen. Wanneer het om een expressie gaat zoals `a.get()` dan bevat ArgumentType {<"unknown",1>, <area-in-file(....),2>}. Waarbij area-in-file de locatie van `a.get()` in het Java bestand is.

Relatie voor opslag van aanroep informatie

Naam	`CALL_DESCRIPTION`
Definitie	`rel[Name, Kind, TypeName, Index]`

Onderdelen	• Name = identificatie van de aanroep
	• Kind = geeft het soort element aan
	  ○ De mogelijke opties zijn:
	  ▪ "VAR" = Name bevat de naam van een variabele, klasse of interface
	  ▪ "TYPE" = TypeName bevat een type aanduiding (voor casting)
	  ▪ "CALL" = Name bevat de naam van de aanroep en TypeName bevat de argumenten-informatie
	  ▪ "REFR" = Name bevat "this" of "super"
	• TypeName = aanduiding van het type dat een rol speelt bij de aanroep
	  ○ `ArgumentType` is van het type `TYPE_NAME`
	• Index = de positie van het element in de volledige aanroep

Voorbeeld	{ < "", "TYPE", { <"Employee",1>} , 1>,<"toString", "CALL", {}, 2>}
Doel	Door middel van deze relatie kunnen methode-aanroepen zoals `((Employee)a).toString()` naar een formaat worden omgezet welke kan worden gebruikt om het type waarop de aanroep heeft plaatsgevonden te bepalen. Dit wordt gebruikt om de METHOD_CALL relatie te vormen.
Voorwaarde	(geen)

C – Relationeel model

In deze bijlage zijn alle relaties opgenomen die nodig zijn voor het achterhalen van dode code op het niveau van methodes. De informatie die in deze relaties aanwezig is kan worden benaderd met behulp van Rscript om de drie sets, “dode code”, “werkende code” en “grijze code” te bepalen.

Methode-aanroepen

Naam	`METHOD_CALL`
Definitie	`rel[<Method, FullTypeName, Arguments, loc>, <CalledMethodName, FullCalledTypeName, CalledArguments, ReferenceOption, loc>]`

Onderdelen	• Method = De naam van de methode die de aanroep doet naar de methode met de naam CalledMethodName.
	• FullTypeName = De volledige naam van het type waartoe Method behoort.
	  ○ `FullTypeName` is van het type `TYPE_NAME`
	• Arguments = De argumenten van de aanroepende methode `Method`
	  ○ `Arguments` is van het type `CALL_ARGUMENTS`
	• CalledMethodName = De naam van de methode die wordt aangeroepen vanuit `Method`
	• FullCalledTypeName = De volledige naam van het type waarop een methode-aanroep plaatsvindt
	  ○ `FullCalledTypeName` is van het type `TYPE_NAME`
	• CalledArguments = De argumenten van de methode `CalledMethod`
	  ○ `CalledArguments` is van het type `ARGUMENTS`
	• ReferenceOption = De aanduiding welke aangeeft of er met zekerheid kan worden geredeneerd over de methode-aanroep.
	  ○ Kan drie opties bevatten:
	  ▪ “normal” = geen zekerheid
	  ▪ “local” = de methode van FullCalledTypeName of het eerstvolgende supertype wordt benaderd voor een methode-aanroep
	  ▪ “super” = de methode van het eerstvolgende supertype van FullCalledTypeName wordt benaderd voor een methode-aanroep

Voorbeeld	< < "main" , { <"employee",1>, <"EmployeeExample",2> } , {<{ {<"String",1>}, "name",1> } , 1>} , area-in-file ("./employee/EmployeeExample.java" , area (34 , 1 , 69 , 2 , 720 , 1263)) > , < "departement" , { <"employee",1>, <"Employee",2> } , { } , "normal" , area-in-file ("./employee/EmployeeExample.java" , area (65 , 28 , 65 , 47 , 0 , 19)) > >
Doel	Deze relatie biedt de mogelijkheid om vanuit een entry-point te bepalen welke methodes worden aangeroepen. In het voorbeeld wordt vanuit de methode `main` van het type `employee.EmployeeExample` een aanroep gedaan naar de methode `departement` van het type `employee.Employee`. Het is hierbij van belang om te realiseren dat het type van de methode die is aangeroepen niet expliciet aanduidt van welk sub- of supertype de methode daadwerkelijk is aangeroepen. Dit komt voort uit de onzekerheid als gevolg van methode-overschrijving (zie bijlage D).
Voorwaarde	De `FullCalledTypeName` is de aanduiding van het type waarop de methode wordt uitgevoerd. Dit hoeft dus niet hetzelfde te zijn als het type waarnaar wordt gerefereerd wanneer de methode runtime wordt uitgevoerd. Dit kan namelijk, afhankelijk van de `ReferenceOption`, een super- of subtype zijn van `FullCalledTypeName`. Wanneer er een aanroep wordt gedaan vanuit een static initializer wordt Method gelijk gesteld aan “<static-init>”. Op het moment dat er een methode-aanroep wordt gedaan vanuit een constructor wordt Method gelijk gesteld aan “<init>”

Methode declaratie

Naam	`METHOD`
Definitie	`rel[MethodName, FullTypeName, Modifiers, ReturnType,`

Naam	METHOD
	Arguments, loc]

Onderdelen	• MethodName = De naam van de methode die wordt gedeclareerd.
	• FullTypeName = Het type waarbinnen de methode is gedeclareerd
	  ○ FullTypeName is van het type TYPE_NAME
	• Modifiers = Geeft de opties aan die onderdeel zijn van de declaratie van de methode.
	  ○ Modifiers is van het type MODIFIERS
	• ReturnType = Het type dat wordt teruggeven als gevolg van een aanroep naar de methode
	  ○ ReturnType is van het type TYPE_NAME
	• Arguments = De argument-informatie behorende bij de methode-declaratie
	  ○ Arguments is van het type ARGUMENTS

Voorbeeld	< "calculateSalary" , { <"employee",1>, <"Robot",2> } , {"private"}, { <"int",1> }, { } , area-in-file ("./employee/Robot.java" , area (23 , 1 , 29 , 2 , 462 , 129)) >
Doel	De methode-declaraties zijn nodig om op basis van de gevonden dode code en werkende code te kunnen bepalen welke methodes zich in het grijze gebied bevinden ofwel van welke methodes men geen zekerheid heeft over het feit of deze dood is. De methode-declaraties zijn ook nodig in verband met overloading en methode-overschrijving om te kunnen achterhalen welke methodes mogelijk worden aangeroepen wanneer er onzekerheid optreedt als gevolg van polymorfisme.
Voorwaarde	In het geval er sprake is van een constructor declaratie zal de MethodName gelijk zijn aan “<init>”. Dit heeft te maken met de compatibiliteit met AOP en de entry-points die vanuit de dynamische analyse worden gegenereerd. De declaratie van methodes in een Interface worden opgenomen in de METHOD relatie waarbij de Modifiers altijd “abstract” zal bevatten.

Klasse overerving

Naam	EXTENDS_CLASS
Definitie	rel[FullSubclassName, FullSuperclassName]

Onderdelen	• FullSubclassName = De volledige naam van de klasse welke een subklasse is van de klasse aangeduid met het tweede argument
	  ○ FullSubclassName is van het type TYPE_NAME
	• FullSuperclassName = De volledige naam van de klasse welke optreedt als ouder van de subklasse
	  ○ FullSuperclassName is van het type TYPE_NAME

Voorbeeld	< { <"employee",1>, <"Boss",2> } , { <"employee",1>, <"Employee",2> } >
Doel	Met deze relatie is het mogelijk om van een klasse zijn subklassen en superklasse te bepalen. Dit is nodig voor het correct kunnen afhandelen van de onzekerheid met betrekking tot polymorfisme (zie bijlage D) en het beantwoorden van de vragen “Wat is dode code?” en “Wat is werkende code?”. De informatie uit deze EXTENDS relatie wordt dus gebruikt bij het achterhalen van de methode die zou kunnen worden aangeroepen gegeven het type in de METHOD_CALL relatie. In het voorbeeld is employee.Boss een subklasse van employee.Employee.
Voorwaarde	(geen)

Interface implementatie

Naam	IMPLEMENTS
Definitie	rel[FullClassName, FullInterfaceName]
Onderdelen	• FullClassName = De volledige naam van de klasse welke de interface implementeert ○ FullClassName is van het type TYPE_NAME

Naam	`IMPLEMENTS`
	• FullInterfaceName = De volledige naam van de interface welke door de klasse wordt geïmplementeerd ○ `FullInterfaceName` is van het type `TYPE_NAME`
Voorbeeld	< { <"employee",1>, <"Boss",2> } , { <"employee",1>, <"TemporaryEmployee",2> } >
Doel	Op het moment dat het type waarop een aanroep plaatsvindt een interface is wordt deze relatie gebruikt om te bepalen welke types deze interface allemaal implementeren en dus welke methode zouden kunnen worden aangeroepen. Deze relatie is dus nodig voor het correct afhandelen van polymorfisme en de daaraan gerelateerde onzekerheid.
Voorwaarde	(geen)

Interface overerving

Naam	`EXTENDS_INTERFACE`
Definitie	`rel[FullSubInterfaceName, FullSuperInterfaceName]`

Onderdelen	• FullSubInterfaceName = De volledige naam van de interface welke een subinterface is van het tweede argument in de relatie
	  ○ `FullSubInterfaceName` is van het type `TYPE_NAME`
	• FullSuperInterfaceName = De volledige naam van de interface welke optreedt als ouder van de subinterface
	  ○ `FullSuperInterfaceName` is van het type `TYPE_NAME`

Voorbeeld	< { <"util",1>, <"Action",2> } , { <"util",1>, <"ActionListener",2> } >
Doel	Door middel van deze relatie is het mogelijk om te achterhalen welke interfaces aan elkaar gerelateerd zijn. Deze relatie wordt in samenwerking met IMPLEMENTS gebruikt om polymorfisme correct af te handelen.
Voorwaarde	(geen)

Package structuur

Naam	`PACKAGE`
Definitie	`rel[FullPackageName, loc]`
Onderdelen	• FullPackageName = Beschrijving van de volledige package naam ○ `FullPackageName` is van het type `TYPE_NAME`
Voorbeeld	{ <”metastudio”,1>, <”graph”,2>, area-in-file ("./Dummy.java" , area (1 , 1 , 2 , 2 , 462 , 129)) }
Doel	De package informatie maakt onderdeel uit van de volledige klassenaam van een klasse die binnen die specifieke package wordt gedeclareerd. Deze informatie is dus nodig om de fully qualified name te kunnen construeren.
Voorwaarde	Op het moment dat er geen package informatie in de broncode staat wordt automatisch de default package gebruikt. Voor deze bronbestanden wordt de relatie {<”<default-package>”,1>} opgenomen. Deze wordt vervolgens ook in de namen van de types opgenomen die worden geëxtraheerd uit dit bronbestand.

Import

Naam	`IMPORT`
Definitie	`rel[ImportName, loc]`
Onderdelen	• ImportName = Beschrijving van het volledige importstatement ○ `ImportName` is van het type `TYPE_NAME`
Voorbeeld	{ <”java”,1>, <”util”,2>, <”*”,3>, area-in-file ("./Dummy.java" , area (23 , 1 , 29 , 2 , 462 , 129)) }
Doel	Op basis van deze relatie en de gebruikte types in het programma is het mogelijk om de

Naam	`IMPORT`
	volledige typenaam te construeren. Deze relatie is nodig om onderscheid te kunnen maken tussen types met dezelfde naam die anders een ambiguïteit zouden veroorzaken.
Voorwaarde	Bij het achterhalen van de volledige typenaam zal in het geval er een “*” in de import relatie aanwezig is, in de CLASS en INTERFACE relatie moeten worden gekeken welke klassen met deze “*” worden bedoeld. Op het moment dat er sprake is van een import van types waarvan geen broncode beschikbaar is en er gebruik wordt gemaakt van “*” kan als gevolg hiervan niet worden bepaald welke types er worden geïmporteerd. Hierdoor kan geen uitspraak worden gedaan over de volledige typenaam van types waarvan de broncode niet aanwezig is.

Type declaraties

Naam	`TYPE`
Definitie	`rel[FullTypeName, Modifier, loc]`
Onderdelen	● FullTypeName = de volledige naam van het type (Java class) ○ `FullTypeName` is van het type TYPE_NAME ● Modifier = De set van opties die onderdeel zijn van de declaratie ○ `Modifier` is van het type `MODIFIERS`
Voorbeeld	< { <"employee", 1> , <"Boss",2> } , {"public"} , area-in-file ("./employee/Boss.java" , area (3 , 0 , 47 , 1 , 19 , 1114)) >
Doel	Met behulp van de type-declaraties is het mogelijk om alle niet volledige type-aanduidingen binnen de METHOD, METHOD_CALL, VARIABLE_DECL en FIELD_DECL op te lossen. Dit gebeurt in samenspraak met de IMPORT relatie om ambiguïteiten met betrekking tot namen te kunnen oplossen.
Voorwaarde	In het geval er sprake is van een anonieme klasse zal in de FullTypeName de structuur van de klasse tot aan de anonieme klasse plus het type van de anonieme klasse worden opgenomen. Onder types worden geen interfaces of header files verstaan. Deze zijn onderdeel van de INTERFACE relatie.

Interface declaraties

Naam	`INTERFACE`
Definitie	`rel[FullInterfacename, Modifier, loc]`
Onderdelen	● FullInterfaceName = de volledige naam van de interface ○ `FullInterfaceName` is van het type TYPE_NAME ● Modifier = De set van opties die onderdeel zijn van de declaratie ○ `Modifier` is van het type `MODIFIERS`
Voorbeeld	< { <"employee",1>, <"TemporaryEmployee",2> }, {"public"}, area-in-file ("./employee/TemporaryEmployee.java" , area (3 , 0 , 10 , 1 , 19 , 203)) >
Doel	Met behulp van de type declaraties is het mogelijk om alle niet volledige type aanduidingen binnen de METHOD, METHOD_CALL, VARIABLE_DECL en FIELD_DECL op te lossen. Dit gebeurt in samenspraak met de IMPORT relatie om ambiguïteiten met betrekking tot namen te kunnen oplossen.
Voorwaarde	(geen)

Variabele declaraties

Naam	`VARIABLE_DECL`
Definitie	`rel[FullClassName, MethodName, Arguments, VarName, loc, FullTypeName]>`
Onderdelen	● FullClassName = De naam van de klasse waarin de variabele wordt gedeclareerd ○ `FullClassName` is van het type `TYPE_NAME`

Naam	VARIABLE_DECL

	• MethodName = De naam van de methode waarbinnen de variabele wordt gedeclareerd.
	• Arguments = De argumenten van de methode waarbinnen de variabele wordt gedeclareerd
	  ○ `Arguments` is van het type `ARGUMENTS`
	• VarName = De naam van de variabele
	• FullTypeName = De volledige typenaam van de variabele die wordt gedeclareerd
	  ○ `FullTypeName` is van het type `TYPE_NAME`

Voorbeeld	< { <"employee",1>, <"CEO",2> } , "salary" , { } , "i" , area-in-file ("./employee/CEO.java" , area (36 , 10 , 36 , 11 , 750 , 1)) , { <"int",1> } > } >
Doel	Om te kunnen achterhalen op welke type de methode-aanroepen plaats vinden is het nodig om de types van de variabele declaraties te weten en de methode waarbinnen deze zijn gedeclareerd. De types zijn nodig om in het geval er een methode-aanroep wordt gedaan op een variabele, het type van de aanroep te kunnen bepalen en dus de bijbehorende implementatie. De plek van de declaratie is nodig uit het oogpunt van shadowing. Dit speelt een rol op het moment dat er twee of meer variabelen zijn met dezelfde naam en het dus niet direct duidelijk is welk type van belang is bij de methode-aanroep. Deze relatie bevat samen met de relatie FIELD_DECL alle informatie die beschikbaar is over variabelen waarop een methode-aanroep kan worden gedaan.
Voorwaarde	Er wordt op basis van de aanname over polymorfisme geen rekening gehouden met het feit dat het type van een variabele kan verschillen van het type waarnaar het field verwijst. Dit laatste wordt namelijk runtime bepaald. Wanneer er een variabele wordt gedeclareerd in de static initializer van een klasse dan wordt de MethodName gelijk gesteld aan <static-init> en is de argumentenlijst leeg. Op het moment dat er een variabele in een constructor wordt gedeclareerd is de MethodName gelijk aan <init>.

Field declaraties

Naam	FIELD_DECL
Definitie	`rel[ FullClassName, FieldName, Modifier, loc, FullTypeName]`

Onderdelen	• FullClassName = De naam van de klasse waarin het field wordt gedeclareerd
	  ○ `FullClassName` is van het type `TYPE_NAME`
	• FieldName = De naam van het veld
	• Modifier = De opties die onderdeel zijn van de fielddeclaratie
	  ○ `Modifier` is van het type `Modifiers`
	• FullTypeName = De volledige typenaam van de variabele die wordt gedeclareerd
	  ○ `FullTypeName` is van het type `TYPE_NAME`

Voorbeeld	< { <"employee", 1>, <"EmployeeExample",2> } , "ROBOT" , {"public","final","static"} , area-in-file ("./employee/EmployeeExample.java" , area (11 , 25 , 11 , 30 , 205 , 5)) , { <"int",1> } >
Doel	Om te kunnen achterhalen op welk type de methode-aanroepen plaatsvinden is het onder andere nodig om de types van de field declaraties te weten en de methode waarbinnen deze zijn gedeclareerd. Het gaat hier dus om de declaraties die niet binnen een methode plaatsvinden maar onderdeel zijn van de attributen van de klasse. De fielddeclaraties kunnen door de declaraties in VARIABLE_DECL worden geshadowed.
Voorwaarde	Er wordt op basis van de aanname over polymorfisme geen rekening gehouden met het feit dat het type van een field kan verschillen van het type waarnaar het field verwijst. Dit laatste wordt namelijk runtime bepaald.

Constructor aanroepen

Naam	`CONSTRUCTOR_CALL`
Definitie	`rel[Method, FullTypeName, Arguments, loc>, <”<init>”, FullCalledTypeName, CalledArguments, ReferenceOption, loc>]`

Onderdelen	• FullTypeName = De volledige naam van het type waartoe Method behoort.
	  ○ `FullTypeName` is van het type `TYPE_NAME`
	• Arguments = De argumenten van de aanroepende methode `Method`
	  ○ `Arguments` is van het type `CALL_ARGUMENTS`
	• FullCalledTypeName = De volledige naam van het type waarvan de constructor wordt aangeroepen
	  ○ `FullCalledTypeName` is van het type `TYPE_NAME`
	• CalledArguments = De argumenten van de methode `CalledMethod`
	  ○ `CalledArguments` is van het type `UNRESOLVED_ARGUMENTS`
	• ReferenceOption = De aanduiding welke aangeeft of er met zekerheid kan worden geredeneerd over de methode-aanroep.
	  ○ Kan twee opties bevatten:
	  ▪ “local” = de constructor van FullCalledTypeName of het eerstvolgende supertype wordt benaderd voor een aanroep
	  ▪ “super” = de constructor van het eerstvolgende supertype van FullCalledTypeName wordt benaderd voor een aanroep

Voorbeeld	< < "main" , { <"employee",1>, <"EmployeeExample",2> } , {<{ {<"String",1>}, "name",1> } , 1>} , area-in-file ("./employee/EmployeeExample.java" , area (34 , 1 , 69 , 2 , 720 , 1263)) > , < "<init>" , { <"employee",1>, <"Employee",2> } , { } , "local" , area-in-file ("./employee/EmployeeExample.java" , area (65 , 28 , 65 , 47 , 0 , 19)) > >
Doel	Deze relatie wordt gebruikt om het mogelijk te maken de METHOD_CALL relatie op te bouwen. De informatie in de CONSTRUCTOR_CALL bevat alle informatie behalve het type van de argumenten op het moment dat deze geen literal betreffen. De CONSTRUCTOR_CALL relatie bevat alleen aanroepen naar constructors.
Voorwaarde	Wanneer er een aanroep wordt gedaan vanuit een static initializer wordt de MethodName gelijk gesteld aan “<static-init>”. Op het moment dat er een methode-aanroep wordt gedaan vanuit een constructor wordt de MethodName gelijk gesteld aan “<init>”

Methode-aanroepen met ontbrekende type informatie

Naam	`UNRESOLVED_CALL`
Definitie	`rel[<Method, FullTypeName, Arguments, loc>, <CalledMethodName, loc>]`

Onderdelen	• Method = De naam van de methode die de aanroep doet naar de methode met de naam CalledMethodName.
	• FullTypeName = De volledige naam van het type waartoe Method behoort.
	  ○ `FullTypeName` is van het type `TYPE_NAME`
	• Arguments = De argumenten van de aanroepende methode `Method`
	  ○ `Arguments` is van het type `CALL_ARGUMENTS`
	• CalledMethodName = De naam van de methode die wordt aangeroepen vanuit `Method`
	  ○ `CalledMethodName` is van het type `CALL_DESCRIPTION`

Voorbeeld	< < "main" , { <"employee",1>, <"EmployeeExample",2> } , {<{ {<"String",1>}, "name",1> } , 1>} , area-in-file ("./employee/EmployeeExample.java" , area (34 , 1 , 69 , 2 , 720 , 1263)) > , < {<”a”, “VAR”, {}, 1>, <”getName”,”CALL”, {}, 2>}, area-in-file ("./employee/EmployeeExample.java" , area (65 , 28 , 65 , 47 , 0 , 19)) > >
Doel	In deze relatie staat de informatie die nodig is om een methode-aanroep op te bouwen die

Naam	UNRESOLVED_CALL
	in de relatie METHOD_CALL kan worden opgenomen. Hiervoor wordt gebruik gemaakt van de FIELD_DECL, VARIABLE_DECL en METHOD_DECL relatie. Hiermee is het mogelijk om het type te bepalen van de variabele waarop de aanroep heeft plaats gevonden.
Voorwaarde	Wanneer er een aanroep wordt gedaan vanuit een static initializer wordt de MethodName gelijkgesteld aan “<static-init>”. Op het moment dat er een methode-aanroep wordt gedaan vanuit een constructor wordt de MethodName gelijk gesteld aan “<init>”

D – Aannames en eigenschappen van Java

Het beginpunt voor statische analyse is het extraheren van de feiten vanuit de broncode in de vorm van relationele informatie. Hiervoor is het nodig om enige aannames te bespreken welke in acht zijn genomen bij deze extractie en enkele specifieke eigenschappen van Java die een rol spelen bij de dode code analyse. Deze aannames en eigenschappen bepalen in feite ook de reikwijdte van de analyse en de taal-afhankelijkheid ervan.

Ingevoerde broncode

Een belangrijke voorwaarde voor een correcte feitenextractie is dat de broncode compileerbaar is. Hiermee wordt voorkomen dat er onnodig complexe logica moet worden opgenomen om controles uit te voeren over de legitimiteit van methode-aanroepen. Daarnaast wordt door het feit dat de code compileert gegarandeerd dat de broncode voldoet aan de syntactische specificatie van Java bestanden. Als extra eis wordt er vanuit gegaan dat alle te analyseren Java broncode voldoet aan de specificatie[13] volgens Java versie 1.4 of lager. Dit betekent dus dat er geen ondersteuning wordt geboden voor bijvoorbeeld generics[20].

Native code

Het is mogelijk om in Java door middel van de native interface code te laden die in C(++) of andere talen is geschreven die uiteindelijk assembly code opleveren. Dit geeft de mogelijkheid om bepaalde stukken functionaliteit in een andere taal uit te drukken op het moment dat deze op het gebied van bijvoorbeeld performance beter kunnen worden geïmplementeerd in een andere taal of omdat bepaalde functionaliteit (zoals direct memory access) niet binnen Java mogelijk is. Hieronder staat een voorbeeld van het gebruik van een native interface.

```
public class Glue {
      static
      {
            // load Glue.dll containing Glue.nativeWidthInBits
            System.loadLibrary( "Glue" );
      }
      public static native int widthInBits( int n );
 }
```

CV 1 – Native code gebruik

In het geval van native code wordt er dus een klasse gecreëerd met de declaraties van de methodes die in de native implementatie aanwezig zijn[13]. Met behulp van de loadLibrary wordt de native code verbonden met de methodedeclaraties.

De structuur van de klasse drukt dus de vanuit de native code aangeboden functionaliteit uit. Op basis van deze informatie is het dus wel mogelijk om binnen de applicatie aanroepen naar de native code te achterhalen aangezien er gewoon aanroepen kunnen worden gedaan naar de methodes die de klasse aanbiedt.

Door het gebruik van native code is er wel een beperking met betrekking tot de analyse van de body van de methode. In het geval van niet native Java klassen kan in de methodbody worden gekeken welke aanroepen plaatsvinden. In het geval van een native implementatie is dat niet mogelijk. Het gevolg is dat wanneer er sprake is van dode code in de gebruikte native implementatie deze niet kan worden gedetecteerd vanuit de analyse van de Java code.
De analyse zal niet in de native code kunnen gaan zoeken naar dode code aangezien het ten eerste lang niet altijd zo is dat hiervan de broncode aanwezig is en het ten tweede zou kunnen zijn dat deze in een programmeertaal is geschreven die niet door de analyse wordt ondersteund. De entry-points naar deze native code vanuit de Java kunnen wel worden bepaald.

Het gevolg van de beperking met betrekking tot de native code is dat wanneer men een native applicatie heeft waaromheen een Java-schil is geschreven, er maar een zeer beperkte dode code analyse kan worden uitgevoerd. Deze beperking kan vervallen op het moment dat de dode code analyser geschikt is voor meerdere programmeertalen en de broncode van de gebruikte native code beschikbaar is.

[20] Ondersteuning van generics als toevoeging op de virtual machine van Sun zijn uiteraard ook uitgesloten.

Functionaliteit uit bibliotheken

Veel van de klassen en methodes die worden gebruikt binnen een applicatie komen rechtstreeks uit de bibliotheken van Java of uit een externe in Java geïmplementeerde bibliotheek. Het gevolg hiervan is dat alle klassen die binnen een applicatie worden gebruikt vanuit een bibliotheek dezelfde analysebeperkingen hebben als native code. Het uitvoeren van een dode code analyse van een bibliotheek is ook niet haalbaar omdat de entry-points niet duidelijk gedefinieerd zijn doordat deze bepaald worden door alle applicaties die van de bibliotheek gebruik maken.

Impliciete aanroepen naar methodes

In een Java applicatie hoeven niet alle methode-aanroepen expliciet in de broncode te zijn opgenomen. Dit komt doordat de onderliggende virtual machine deze methodes benadert (bijvoorbeeld een aanroep naar de finalizer) of dat er aanroepen ontstaan als gevolg van de functionaliteit die met de Java bibliotheken wordt aangeboden (zoals het event-mechanisme)

Voor de dode code analyse is het van belang dat er rekening wordt gehouden met de mogelijkheid dat er impliciete aanroepen worden gedaan naar methodes die in de onderzochte applicatie zijn gedeclareerd. Het gevolg is namelijk dat de mogelijkheid tot het introduceren van false positives optreedt doordat niet alle methode-aanroepen zouden kunnen worden gevonden.

Een van de impliciete aanroepen waarmee rekening moet worden gehouden is dat er sprake kan zijn van impliciete aanroepen naar de super constructor van een afgeleide klasse. Op het moment dat een afgeleide klasse een constructor bevat waarin niet een expliciete aanroep wordt gedaan naar de constructor van de superklasse wordt deze toch aangeroepen. Deze impliciete aanroep zal altijd de defaultconstructor zijn van de afgeleide klasse. Er zouden onterecht constructors als dode code kunnen worden bestempeld als gevolg van het ontbreken van een expliciete aanroep in de broncode.

Een tweede vorm van impliciete aanroepen komt voort uit het feit dat alle klassen binnen Java direct of indirect zijn afgeleid van Object[13]. Dit hoeft niet expliciet in de extend relatie te zijn opgenomen van een klasse. Iedere klasse die niet expliciet van een andere klasse is afgeleid door middel van extends, is van Object afgeleid. Sommige aanroepen naar methodes van Object gebeuren als gevolg hiervan impliciet. Zo zal de aanroep naar een finalize methode niet worden gevonden aangezien deze automatisch wordt aangeroepen voordat een instantie van een klasse wordt opgeruimd[13]. Deze methode is specifiek voor de garbagecollector bedoeld en normaal gesproken wordt er geen expliciete call naar gedaan in de broncode van een applicatie. Een andere vorm van impliciete aanroepen die belangrijk zijn heeft betrekking op de toString methode van Objecten. Het is niet nodig om deze methode expliciet aan te roepen wanneer deze wordt gebruikt in een println aanroep of bijvoorbeeld bij concatenatie met de “+” operator.

De veiligste manier om false-positives als gevolg van impliciete aanroepen in de dode code set te voorkomen is om alle methodes van Object als aangeroepen te beschouwen. Een andere wijze waarop deze false positives zouden kunnen worden voorkomen is door het expliciet gaan herkennen van constructies die impliciete aanroepen tot gevolg hebben. Een nadeel is wel dat er dan een veel taal-afhankelijkere analyse ontstaat.

Een derde mogelijkheid om false positives tegen te gaan is om de broncode van alle standaard Java bibliotheken op te nemen in de analyse. In dat geval is het namelijk mogelijk om ook de aanroepen binnen een methode uit de Java bibliotheek te bekijken en zal de impliciete toString aanroep die volgt uit een aanroep naar println een expliciete aanroep worden. Dit komt doordat in dat geval de broncode van de klasse beschikbaar is waarin de println methode wordt gedeclareerd. Het opnemen van de broncode van de Java bibliotheken zou een impact hebben op zowel de snelheid van de analyse als het geheugengebruik. Daarnaast heeft men dan nog niet gegarandeerd alle impliciete aanroepen aangezien bijvoorbeeld de aanroepen van de finalizer door de virtual machine worden geregeld.

De uiteindelijke keuze die gemaakt is om het probleem van impliciete aanroepen tegen te gaan is om in de analyse de mogelijkheid op te nemen standaard methode-aanroepen te specificeren. Het grootste voordeel ten opzichte van de andere opties is dat deze lijst vooraf aan de analyse kan worden geconstrueerd en dat er bij het constateren van een false positive als gevolg van een impliciete aanroep deze eenvoudig kan worden ingevoegd in deze lijst. In feite kan de lijst van impliciete aanroepen incrementeel worden opgebouwd, wat voordelen heeft aangezien het moeilijk zal zijn vooraf een lijst te maken van alle impliciete aanroepen. De impliciete aanroepen spelen namelijk niet alleen maar een rol bij Object maar ook bijvoorbeeld bij Threads waarbij run() automatisch wordt aangeroepen wanneer een Thread via start() wordt geactiveerd.

Het gevolg van bovenstaande keuze is wel dat de kans op false negatives toeneemt doordat zonder analyse wordt gesteld dat bepaalde aanroepen plaatsvinden terwijl dit niet zo hoeft te zijn. Dit houdt dus in dat als gevolg van het opnemen van een lijst van impliciete aanroepen het mogelijk is dat dode code niet als zodanig wordt herkend. Aangezien het optreden van false positives veel zwaarder wordt bestraft dan het optreden van false negatives is naar mijn mening de gemaakt keuze gerechtvaardigd.

Methode-overschrijving

De mogelijkheid tot methode-overschrijving is één van de twee mogelijke vormen van polymorfisme. Polymorfisme bij methode-overschrijving heeft tot gevolg dat dynamisch wordt bepaald van welke klasse een bepaalde overschreven methode moet worden aangeroepen[W11]. Daarnaast speelt polymorfisme ook een rol bij overloading.

Met behulp van onderstaand voorbeeld zal het principe van methode-overschrijving en de bijkomende moeilijkheden als gevolg van polymorfisme worden verduidelijkt.

```
//Class A
public class A {
    ......
    public int getNumber() { ... }
}

//Class B
public class B extends A {
    ......
    public int getNumber() { ... }
}

// Beginpunt applicatie
public static void main(String[] Args) {
    ....
    A a = new B();
    a.getNumber();
    ....
}
```

CV 2 – Voorbeeld methode-overschrijving

In bovenstaand voorbeeld zijn twee klassen zichtbaar, `A` en `B`, waarbij klasse `B` van `A` is afgeleid. In het beginpunt van de applicatie wordt een variabele gedeclareerd van het type `A` waaraan de instantie van het type `B` wordt toegekend. Op het moment dat nu de aanroep wordt gedaan naar de methode `getNumber` wordt op basis van afspraken met betrekking tot polymorfisme bepaald dat feitelijk de `getNumber` van klasse `B` wordt aangeroepen, ook al is de variabele waarop de aanroep plaatsvindt gedeclareerd als zijnde van het type `A`.

Uit bovenstaand voorbeeld volgt dat het gedeclareerde type van de variabele waarop een methode-aanroep wordt uitgevoerd in veel gevallen geen inzicht geeft in de daadwerkelijke methode die wordt aangeroepen. De onzekerheid over welke methode daadwerkelijk wordt aangeroepen treedt op wanneer het gedeclareerde type van de variabele waarop de methode-aanroep plaatsvindt, als superklasse wordt gebruikt door klassen die de methode overschrijven. Het enige geval waarin er wel volledige zekerheid is over de methode-aanroep is wanneer het type van de variabele waarop de methode-aanroep wordt gedaan geen subklassen bevat die deze methode overschrijven.

Wat men eigenlijk zou willen weten is het type waarnaar de variabele refereert, ofwel het type van de instantie die aan de variabele is verbonden[13]. In het voorbeeld CV2 lijkt dit nog zeer eenvoudig om te bepalen aangezien duidelijk zichtbaar is welke constructor er wordt aangeroepen en dus welke type aan de variabele is toegekend.

In meer realistische situaties is het onmogelijk om het type statisch te bepalen aangezien het type van de toegekende instantie afhangt van de wegen die worden gevolgd in bijvoorbeeld selectieprocedures van een abstract factory. Statisch, dus zonder executie van de code, kan men niet bepalen op welke manier de Factory wordt gebruikt doordat men de invoer naar de Abstract Factory in veel gevallen niet kan berekenen en de logica niet kan evalueren als gevolg van de invloed van externe factoren. Een dynamische analyse zal hier ook geen oplossing bieden in verband met de scenario-afhankelijkheid[10]

Als gevolg van de problematiek rondom methode-overschrijving ontstaat er onzekerheid binnen de statische analyse over het feit van welke klasse nou daadwerkelijk de methode wordt aangeroepen. Het feit dat er onzekerheid ontstaat bij statische analyse is het fundamentele probleem waarop de oplossing is gebaseerd die in dit onderzoek wordt nagestreefd. Zoals in paragraaf 4.3.2 valt te lezen wordt gebruik gemaakt van drie sets; dode code, werkende code en grijze code. Om de opdeling in deze drie sets mogelijk te maken moeten er met betrekking tot methode-overschrijving op twee manieren met de ontstane onzekerheid worden omgegaan.

Bij de opstelling van de dode code set en dus de beantwoording van de vraag *"Wat is dode code?"* zal er een conservatieve instelling moeten worden ingenomen. Dit houdt in dat op het moment dat een methode wordt aangeroepen van een bepaald type dat dan de methodes in alle subklassen die deze methode overschrijven ook als "gebruikt" worden bestempeld. In het geval van het eerder gegeven voorbeeld in CV 2 betekent dit dat als gevolg van de aanroep `a.getNumber()` en het feit dat de variabele `a` van het type `A` is dat de methode `getNumber()` in zowel klasse `A` als zijn subklasse `B` als gebruikt wordt bestempeld.

Op het moment dat geen van de subklassen de methode overschrijft is er geen onzekerheid meer over de methode die zou worden aangeroepen. Dit geldt omdat aan een variabele van het type `A` nooit een instantie kan worden toegekend van het type welke direct of indirect een superklasse is van `A` (zoals het type `Object`).

Wanneer op deze wijze de vraag wordt beantwoord is men conservatief over de methode die als gevolg van polymorfisme zou kunnen worden aangeroepen. Het gevolg hiervan is dat op deze manier wordt voorkomen dat een methode als dood zou worden bestempeld terwijl deze eigenlijk wordt gebruikt als gevolg van polymorfisme. Dit komt doordat men alle methodes die mogelijk zouden kunnen worden aangeroepen als "gebruikt" bestempelt. Hiermee worden dus false positives als gevolg van de onzekerheid volgende uit methode-overschrijving voorkomen.

De tweede aanname is voor het opbouwen van de gebruikte code set en beantwoordt de vraag "*Wat is werkende code?*". Dit principe kan worden bereikt door te stellen dat op het moment dat er sprake is van polymorfisme, en er sprake is van onzekerheid over van welke klasse de methode daadwerkelijk wordt gebruikt, dat de methode-aanroep word genegeerd. Dit leidt ertoe dat als gevolg hiervan de methodes die als gebruikt worden bestempeld ook gegarandeerd binnen de applicatie een methode-aanroep hebben.

De onzekerheid die bij polymorfisme op het gebied van methode-overschrijving een rol speelt kan door middel van bovenstaande eigenschappen van Java en de daaruit volgende aannames op een correcte wijze worden behandeld en meegenomen in de dode code analyse. In de flow diagrammen (zie bijlage I) is het bovenstaande principe grafisch uitgebeeld.

Methode-overloading

Overloading biedt de mogelijkheid om meerdere methodes met eenzelfde naam in een klasse te definiëren waarbij het onderscheid wordt gemaakt tussen het aantal argumenten, het type van de argumenten en de volgorde van de argumenten [13]. Een voorbeeld van method overloading staat in CV 3:

```
// Class convertor
// Nonsense class
public class Convertor {
      public static Integer get(Object a) { ... }
      public static String  get(String a) { ... }
      public static String  get(String a, Integer b) { ... }
}

// Beginpunt applicatie
public void main(String[] Args) {
    ....
    Object square = new String("length(5)");
    Integer area = Convertor.get(square);
    ....
}
```

CV 3 – Voorbeeld overloading

In voorbeeld CV 3 zijn drie methodes gedeclareerd die exact dezelfde naam hebben maar die verschillen in het type en de hoeveelheid van de argumenten. Afhankelijk van de argumenten die worden meegegeven bij de

aanroep wordt één van de drie methodes uitgevoerd. Om te kunnen bepalen welke van de drie wordt aangeroepen is het nodig om het type te weten van het argument dat wordt gebruikt.

Bij overloading is het type van het argument waarnaar wordt gekeken gelijk aan het type dat is gebruikt bij de declaratie. Er ontstaat dus geen onzekerheid zoals dat wel in het geval van polymorfisme ontstaat, aangezien de signatuur van de methode-aanroep compile-time wordt bepaald. Op het moment dat dus het type van de argumenten bekend is kan ook met zekerheid de signatuur van de aangeroepen methode worden vastgesteld. In bovenstaand voorbeeld betekent dit dat de aanroep van de `get` methode en de toekenning aan de variabele van het type `Integer` mogelijk is, doordat de eerste methode wordt aangeroepen van de klasse `Convertor`.

	In het geval van overloading van methodes spreekt men wel over het geval van compile-time polymorfisme[W10] en in het geval van het overschrijven van methodes van runtime polymorfisme. Dit komt doordat bij runtime polymorfisme de type-binding pas wordt bepaald op het moment dat het programma wordt uitgevoerd. In het geval van overloading is de binding met de methode in feite al bekend nadat de compiler het programma heeft omgezet naar bytecode. Door de mogelijkheid van de combinatie van overloading met overschrijving zal de binding echter pas runtime worden bepaald. Het is met statische analyse alleen mogelijk om over de informatie die compile-time bekend is met zekerheid conclusies te trekken. Op het moment dat de benodigde informatie pas runtime bekend wordt zal altijd de onzekerheid als gevolg van de scenario-afhankelijkheid een rol gaan spelen.

In onderstaand voorbeeld wordt nog een speciale constructie van overloading behandeld waarin de voorkeur van afhandeling een rol gaat spelen. In [13] zijn in paragraaf 15.12.3 nog enkele andere voorbeelden gegeven van speciale situaties.

```
// Class A
public class A implements Ainterface { ... }

// Class B
public class B extends A { ... }

// Methodes
public static void get(A a)          { ... }
public static void get(Ainterface a) { ... }

// Beginpunt applicatie
public void main(String[] Args) {
    ....
    //1
    A a1 = new A();
    get(a1);
    ....
    //2
    Ainterface a2 = new A();
    get(a2);
    ....
    //3
    B b = new B();
    get(b);
    ....
}
```

CV 3 – Voorbeeld special case van overloading

In bovenstaand voorbeeld zijn twee klassen `A` en `B` waarbij `B` een subklasse is van `A` en de klasse `A` de interface `Ainterface` implementeert. Daarnaast zijn er twee methodes die kunnen worden gebruikt waarbij de eerste methode een argument verwacht van het type `A` en de tweede methode een argument verwacht van het type `Ainterface`.

In het eerste voorbeeld, aangegeven met tag //1, wordt zoals verwacht de eerste `get` methode aangeroepen waarvan het argument van het type `A` is. In het tweede voorbeeld, aangegeven met tag //2, wordt de tweede `get` methode aangeroepen welke een argument van het type `Ainterface` verwacht.

Bij het derde testvoorbeeld wordt gebruik gemaakt van het type `B` welke een subklasse is van `A` en via `A` ook `Ainterface` implementeert. In feite zouden beide get methodes in aanmerking komen om te worden aangeroepen. Aangezien dit ambiguïteit zou opleveren bij de runtime-executie geeft Java de voorkeur aan de `get` methode waarbij het argument van het type A is. Als gevolg hiervan zal dus runtime de eerste `get` methode worden aangeroepen.

Op het moment dat de argumenten van een methode primitieve typen bevat zal er rekening moeten worden gehouden met de manier waarop deze types naar elkaar kunnen worden geconverteerd. In [13] is in paragraaf 5.1.2 de mogelijk conversie opgenomen. Zo is het mogelijk om van een `int` een `long` te maken, maar andersom niet zonder expliciete casting. Om dit praktisch goed te laten gaan kunnen deze regels in feite worden beschouwd als een hiërarchie zoals deze ook tussen klassen bestaat. Door te stellen dat het type `int` overerft van het type `long` zal bij overloading waarbij primitieve types als argumenten worden gebruikt de redenatie voor het bepalen van de aangeroepen methode correct verlopen. De redenering is immers dezelfde als in het geval er gebruik wordt gemaakt van klassen of interfaces.

In de praktische implementatie van de dode code analyse is overloading slechts gedaan door het selecteren op het aantal argumenten en wordt er dus niet gekeken naar de types of volgorde van de argumenten. Dit is het gevolg van het feit dat, gezien de beperkte tijd, het niet mogelijk bleek om in de praktische oplossing het type van alle argumenten bij methode-aanroepen te bepalen. Een nadeel en potentieel probleem welke voortkomt uit deze beperkte overloading strategie is dat wanneer er twee methodes zijn met dezelfde naam en hetzelfde aantal argumenten maar een verschillend returntype, het niet duidelijk is welke van de twee types zal worden geretourneerd. Het gevolg is dat wanneer er een dubbele aanroep wordt gedaan, zoals `a.b().c()` en het returntype van de aanroep naar `b` niet duidelijk is als gevolg van de beperkte overloading analyse, dat er onzekerheid optreedt over de aanroep naar `c`. Het is dan niet meer duidelijk op welk type de methode `c` wordt aangeroepen.

De aanname voor de praktische implementatie die hieruit volgt is dat wanneer er sprake is van overloading van een methode met eenzelfde aantal argumenten en eenzelfde naam dat deze ook hetzelfde returntype hebben.

Reflectie

Reflectie maakt het binnen Java mogelijk om van een klasse informatie te achterhalen over aangeboden methodes, instance variabelen en constructors[W11]. Op basis hiervan wordt via reflectie de mogelijkheid geboden om dynamisch functionaliteit te laden door bytecode te laden en via reflectie naar deze geladen delen een aanroep uit te voeren.

Hieronder staat ter verduidelijking een voorbeeld van het dynamische gebruik van onderdelen van een klasse.

```
Class classDefinition = Class.forName(packageName + operationName);
Object classInstance  = classDefinition.newInstance();
Method someMethod     = classDefinition.getMethod(methodName, params);
someMethod.invoke(classInstance, arguments);
```

CV 4 – Voorbeeld reflectie

Er ontstaat een probleem voor de dode code analyse doordat de `invoke` methode ervoor zorgt dat op basis van de informatie opgehaald met `getMethod` een bepaalde methode van de klasse wordt uitgevoerd. De methode-aanroep zit dus verkapt in de `invoke` aanroep en is niet direct zichtbaar.

Aangezien het gaat om een programmeerconstructie die altijd op een vaste manier wordt gebruikt is de eerste gedachte dat het mogelijk is om in de feitenextractie van de Java code een uitzondering te maken om deze programmeerconstructie te detecteren. Op basis van de argumenten naar de verschillende methode zou dan de aanroep kunnen worden bepaald. Fundamenteel probleem hierbij is alleen dat dit in veel gevallen niet mogelijk is doordat er gebruik wordt gemaakt van variabelen en het bijvoorbeeld mogelijk is dat de `methodName` afhankelijk is van de invoer van de gebruiker of een ander proces dat alleen dynamisch te bepalen is. Hierdoor is het niet mogelijk om dode code statisch te analyseren in klassen die met behulp van reflectie worden geladen ook al beschikt men over de broncode.

Een veel belangrijker nadeel is dat reflectie kan leiden tot het ontstaan van false positives. Dit komt doordat het kan gebeuren dat een te analyseren klasse geen expliciete aanroepen heeft naar methodes die binnen de klasse zijn gedeclareerd maar dat deze alleen worden gebruikt via reflectie. Het gebruik als gevolg van reflectie kan

niet worden gedetecteerd en zal dus leiden tot het feit dat de conclusie wordt getrokken dat er geen aanroepen naar deze methodes plaatsvinden. Hierdoor zullen deze methodes in de dode code set worden opgenomen terwijl de methodes worden gebruikt via reflectie en het verwijderden van deze methodes dus tot een semantische verandering van het programma zal leiden.

In feite zouden de methode-aanroepen naar via reflectie geladen klassen als entry-points moeten worden beschouwd. Het nadeel hiervan is dat het probleem hiermee slechts wordt verschoven. Voordeel is wel dat het ook de uitvoerder van de dode code analyse dwingt om na te denken over de gevolgen van reflectie en deze dus stilstaat bij de mogelijke incorrecte resultaten.

Voor de praktische implementatie van de dode code analyse wordt de aanname gedaan dat de aanroepen die worden gedaan op klassen geladen via reflectie en uitgevoerd via i`nvoke()` niet als methode-aanroepen worden beschouwd.

Shadowing

Van shadowing binnen Java is sprake [13] op het moment dat een lokale variabele wordt gedeclareerd die dezelfde signatuur heeft als een fielddeclaratie op het niveau van de klasse of bijvoorbeeld op het moment dat er als gevolg van import-statements en klasse declaraties binnen het bronbestand, meerdere types zijn met dezelfde simple-name. Hieronder staat een voorbeeld:

```
class Pair {
    // Field declaratie
    Object first, second;
    public Pair(Object first) {
        // Lokaal gebruik van first argument die shadowed met field
        this.first = first;
        // Lokale declaratie van variabele die shadowed met field
        Integer second = new Integer(42);
        // Aanroep naar lokale variabele
        second.intValue()
        // Aanroep naar globale variabele
        this.second.toString()
    }
}
```

CV 5 – Shadowing

In bovenstaand voorbeeld bepaalt de lokale variabele `second` bij de aanroep naar `intValue()` het type waarop de methode wordt uitgevoerd. In het tweede geval wordt een aanroep gedaan naar `toString()` waarbij het type weer wordt bepaald door de fielddeclaratie van de variabele `second`. Het is dus van belang dat deze informatie wordt behouden bij het extraheren van de feiten.
In bijlage I (flow-diagrammen) is de ondersteuning voor shadowing zichtbaar in bijvoorbeeld “Bepaling type van VAR”. De volgorde van evaluatie en de mogelijkheid tot het hebben van een “uitgangstype” zorgt hier voor een correcte afhandeling van shadowing. Dit “uitgangstype”wordt geinitialiseerd op het moment dat er gebruik wordt gemaakt van “this” danwel “super”.
In [13] wordt in paragraaf 6.3.1 stilgestaan bij de effecten van shadowing en de voorkeuren die hier uit volgen voor het bepalen van de volledige naam van een type.

Exceptie afhandeling

Wanneer binnen een Java-applicatie gebruik wordt gemaakt van exceptie-afhandeling dan wordt zowel het niet optreden van de exceptie als het wel optreden van de exceptie beschouwd. Dit houdt in dat alle methode-aanroepen in de try-, catch- en finally-blokken worden meegenomen en beschouwd als aangeroepen. Het is namelijk niet mogelijk om statische te bepalen of een exceptie zal optreden en dus zullen alle methode-aanroepen die kunnen volgen uit het wel of niet optreden van een exceptie moeten worden beschouwd.
In [13] wordt op pagina 323 een voorbeeld geven van de manier waarop excepties ervoor kunnen zorgen dat je nooit met zekerheid het type waarnaar een variabele refereert kan bepalen. Het kan namelijk zo zijn dat door het optreden van de exceptie een bepaalde toekenning aan een variabele niet plaatsvindt en het betreffende type dus nooit zal worden toegekend.

Methode aanroepen

Door de keuze van het uitvoeren van een beperkte type analyse en juist het leren leven met de optredende onzekerheid treden verschillende voordelen op. Eén van deze voordelen is dat iedere methode binnen een applicatie nu als entry-point kan worden beschouwd. Dit komt door het feit dat het niet nodig is van een argument of variabele te weten naar welk type deze daadwerkelijk refereert. Het is alleen nodig om te weten wat het gedeclareerde type is. Op basis hiervan is het bijvoorbeeld mogelijk om vooraf de impliciete aanroepen te definiëren wat voordelen heeft voor het beperken van false positives en het beperken van de taal-afhankelijkheid van de analyse.

Samenvattend is een groot voordeel van het redeneren met de onzekerheid in de voorgestelde dode code analyse dat er geen afhankelijkheid is van context informatie en het er dus niet toe doet of een gegeven entry-point ook daadwerkelijk het beginpunt van de applicatie is waaruit de context informatie valt te construeren.

E – Build-time dode code analyse

In hoofdstuk 4.2 is aangegeven dat het mogelijk is om een dode code analyse uit te voeren op basis van het build-proces. Het build-proces kan namelijk worden gebruikt om te achterhalen welke broncode daadwerkelijk wordt gecompileerd. Op het moment dat broncode niet wordt gecompileerd zal deze ook in de applicatie niet worden gebruikt. Hierbij moet meteen de kanttekening worden gemaakt dat het in een programmeertaal zoals Java mogelijk is om tijdens de executie van het programma een bronbestand te compileren en vervolgens te laden. Het gevolg daarvan zou zijn dat een bestand dat tijdens het build-proces niet wordt gebruikt uiteindelijk wel door de applicatie wordt gecompileerd en gebruikt. Dit zou dus een false positive opleveren met betrekking tot de set van dode code.

Het build-proces zal niet de gewenste informatie opleveren voor het elimineren van dode code. Ten eerste is het zo dat het kan zijn dat gewoonweg alle broncode wordt gecompileerd. Op basis van het build-proces kan niet worden geconcludeerd dat de gecompileerde broncode vervolgens in het programma wordt gebruikt. Ten tweede is het zo dat het bij veel programmeertalen mogelijk is om runtime code te compileren en uit te voeren. Het gevolg hiervan is dat er false positives kunnen ontstaan als gevolg van de analyse van het build-proces. Ten derde is het zo dat broncode die in het geheel niet wordt gebruikt ook kan worden bepaald met behulp van de combinatie tussen de statische en dynamische analyse. Op het moment dat namelijk geen enkele van de methodes in een bronbestand bereikbaar zijn en de types die worden gedefinieerd in het bronbestand op geen enkele wijze worden gebruikt binnen de applicatie kan ook de conclusie worden getrokken dat het bronbestand niet wordt gebruikt.

	Een bronbestand dat niet wordt gecompileerd en waarvan men dus tijdens het build-proces kan bepalen dat deze niet wordt gebruikt zal ook tijdens de statische analyse (hoofdstuk 5) worden gedetecteerd. Op het moment dat het bronbestand niet wordt gecompileerd is het namelijk niet mogelijk dat ernaar wordt verwezen binnen de applicatie. De applicatie zou dan immers niet volledig compileren. Onzekerheid als gevolg van polymorfisme treedt hier ook niet op aangezien de types in het niet gecompileerde bestand nooit kunnen worden gebruikt. De eerder besproken statische analyse is dus sterker dan de analyse van het build-proces. Als gevolg hiervan is de build-analyse overbodig.

Het build-proces zou wel kunnen worden gebruikt om te analyseren in welke gebieden een bepaalde verandering binnen een bronbestand effect heeft. Een goed build-proces zorgt ervoor dat op basis van de informatie over afhankelijkheden alleen de delen van de applicatie opnieuw worden gecompileerd waarop de wijziging in de broncode effect heeft. Hieruit kan men afhankelijkheden achterhalen zodat kan worden bepaald wat er in ieder geval moet worden getest wanneer code wordt gewijzigd. Aangezien bij het verwijderen van dode code het probleem optreedt dat niet kan worden aangetoond dat de applicatie semantisch onveranderd blijft, kan door middel van het build-proces een indruk worden verkregen waar de verwijdering van de dode code effect heeft. Op deze manier zou gericht kunnen worden gecontroleerd of de semantiek van het programma niet is veranderd. Uiteraard geldt dit weer niet op het moment dat er sprake is van reflectie.

F – Beperkingen en metrieken van de implementatie

In deze bijlage zal worden ingegaan op de beperkingen van de huidige praktische implementatie en zullen enige metrieken van de verschillende fases worden gegeven. De implementatie bestaat uit vier fases:

- Feitenextractie (extraction/JavaAnalysis.sdf)
- Feiten verrijking (enrich/EnrichAnalysis.sdf)
- Dode code analyse met Rscript (analyses/deadcode/DeadCodeDetection.rscript)
- Dynamische analyse met AOP (analyses/dynamic/JoinPointTraceAspect.java)

Uiteraard is het zo dat beperkingen in een eerdere fase binnen de dode code analyse invloed heeft op de onderliggende laag. Zo leggen de beperkingen in de feitenextractie beperkingen op aan de onderliggende drie fases. De beperkingen van de relaties en de voorwaarden waarmee rekening moet worden gehouden bij het gebruik zijn in bijlage B en C opgenomen.

Beperkingen feitenextractie

[FE-2] Er wordt geen rekening gehouden met innerinterfaces en innerclasses die binnen een anonieme klasse worden gedeclareerd.

[FE-3] Op het moment dat er aanroepen worden gedaan zoals `a.new Class()` voor het creëren van een instantie van een innerclass die binnen het type van de variabele `a` is gedeclareerd wordt niet gekeken naar het type van `a` om het type van de Constructor te bepalen voor de CONSTRUCTOR_CALL relatie. In gevallen waarbij er binnen een bronbestand klassen zijn met dezelfde naam zou dit tot ambiguïteit kunnen leiden.

[FE-4] Wanneer bij het creëren van een anonieme klasse de constructie `new A.B(){..}` wordt gebruikt dan wordt de anonieme klasse niet geregistreerd. Dit heeft te maken met het feit dat bij het creëren van de identifier ervanuit wordt gegaan dat er een single identifier is ofwel `new B(){..}` In alle andere gevallen waarbij deze constructie wordt gebruikt (zoals bij variabele declaraties) wordt dit wel ondersteund.

[FE-5] Wanneer een klasse alleen maar wordt gebruikt binnen een exceptie-afhandeling en de Constructor erbuiten nooit expliciet wordt aangeroepen dan is er sprake van een impliciete aanroep naar de Constructor. Een voorbeeld is `try{...} catch(MyException e) {..}` waarbij `MyException` wel een aanroep krijgt naar zijn Constructor maar dit intern gebeurt en dus niet expliciet in de broncode staat. Hier ontstaat hetzelfde probleem als bij impliciete aanroepen naar methodes van `Object` (toString, equals, enz.) of aanroepen naar run van een Thread (zie bijlage D). Dit leidt dus tot false positives in de dode code set.

[FE-6] Op het moment dat er sprake is van een aanroep die volgt op een expressie dan wordt deze niet automatisch omgezet. In de UNRESOLVED_CALL relatie wordt dan een aanroep opgenomen waarvan de naam gelijk is aan “*parse-error*”. Een voorbeeld van een dergelijke aanroep is `("a" + this.getName()).toString()` waarbij van het gedeelte voor de `toString()` het type niet wordt bepaald.

Tabel 13 Metrieken van feiten-extractie

Onderdeel	*Regels broncode*	*Regels commentaar*	*Lege regels*	*Totaal*
ASF	851	80	184	1115
SDF	699	4	173	876

Beperkingen feiten verrijking

[FV-1] De argumenten die worden meegegeven aan methode-aanroepen worden niet geëvalueerd. Alleen wanneer er sprake is van een literal (boolean/char/int/long/float/double/String) wordt het type wel geregistreerd. In alle andere gevallen wordt er als type <”unknown”,1> opgenomen in de relatie. Probleem bij het bepalen van het type van argumenten is de mogelijkheid tot het gebruik van variabele of methode-aanroepen die zijn opgenomen in de Java bibliotheek.

[FV-2] De volledige typenaam wordt bepaald op basis van de informatie die in de feitenextractie naar voren is gekomen. Waarmee geen rekening wordt gehouden is het optreden van twee of meer innerclasses binnen dezelfde klasse. Hier wordt nog geen onderscheid gemaakt omdat de lokatie informatie nog niet is opgenomen in de volledige type-aanduiding. De lokatie-informatie is wel aanwezig vanuit de eerste fase van de feitenextractie. Ook is de afhandeling

van shadowing met betrekking tot innerclasses nog niet volledig geïmplementeerd volgens de specificatie in [13].

[FV-4] De aanroepen naar default-constructors die niet expliciet in de broncode zijn opgenomen worden niet als `METHOD_CALL` geregistreerd. Hiervoor moet per Constructor worden gekeken of deze een andere Constructor aanroept en zo niet moet de aanroep naar `super` worden toegevoegd.

[FV-5] Wanneer er sprake is van een anonieme klasse zou er een toevoeging moeten worden gedaan aan de `EXTENDS_CLASS` relatie om aan te geven wat de superklasse is van de anonieme klasse. Dit is nodig om methode-aanroepen naar de superklasse die via de anonieme klasse worden gedaan te registreren. Op dit moment wordt dit nog niet gedaan waardoor false positives zouden kunnen ontstaan als gevolg van het missen van een aanroep.

[FV-6] Wanneer een aanroep wordt gedaan naar een methode via een field-reference of er sprake is van de uitvoering van een statische methode op een klasse dan wordt het aanroepende type nog niet automatische bepaald. Deze aanroepen zullen dus in de UNRESOLVED_CALL relatie blijven staan.

Tabel 14 Metrieken van feiten-verrijking

Onderdeel	*Regels broncode*	*Regels commentaar*	*Lege regels*	*Totaal*
ASF	1110	294	270	1674
SDF	261	0	93	354

Beperkingen dode code analyse met Rscript

[DC-3] Als gevolg van het feit dat de types van de argumenten bij de methode-aanroepen niet bekend zijn, is het ook niet mogelijk om overloading volledig te ondersteunen. Op het moment dat er sprake is van twee methodes met dezelfde naam binnen eenzelfde type dan wordt gekeken naar het aantal argumenten.

[DC-4] Er wordt nog geen rekening gehouden met het feit dat er een klasse kan zijn met een import-statement die volledig is (zoals java.util.Vector) en een klasse waarbij deze involledig is (zoals java.util.*) dat het gebruik van Vector in feite in beide gevallen dezelfde volledige klassenaam moet opleveren. Hiervoor moet een aanpassing komen in de feiten-verrijking of in de vergelijking van de klassenamen binnen het Rscript.

[DC-5] In de dode code set zijn nog methodes opgenomen die abstract zijn en dus nooit rechtreeks worden aangeroepen. Deze moeten nog worden gefiltered door te controleren of er minstens één implementatie van de methode niet als dood is geregistreerd. Dan valt de abstracte definitie ook buiten de dode code set.

Tabel 15 Metrieken van Rscript dode-code analyse

Onderdeel	*Regels broncode*	*Regels commentaar*	*Lege regels*	*Totaal*
Analyse	204	262	77	543
Regressietest	300	22	58	380

Beperkingen dynamische analyse met AOP

[DY-1] Door het gebruik van AOP voor de dynamische analyse is het niet mogelijk om de innerklassen te relateren aan een entry-point. Dit komt door de manier waarop de volgorde van declaratie binnen een bronbestand invloed heeft op de naamgeving (zie paragraaf 6.1) en het feit dat in de verrijkingsfase de lokatie-informatie niet wordt gebruikt voor het onderscheiden van innerklassen met dezelfde naam binnen klassen. Deze ambiguïteit moet (bij gebrek aan een ASF+SDF implementatie voor dynamische analyse) met de hand worden opgelost. Dit betekent dat de berichten die door de dynamische analyse worden gecreëerd in het geval van innerklassen met de hand moeten worden aangepast.

[DY-2] De omzetting vanuit de methode-aanroepen beschreven in de grijze set, naar de join-points in het AOP formaat moet met de hand worden uitgevoerd.

G – Uitvoering dode code analyse op voorbeeld

In deze bijlage zal de manier worden besproken waarop het testvoorbeeld door de dode code analyse wordt behandeld. In hoofdstuk 8.4 kan worden gekeken voor een vergelijking met RevJava[W18].

Het eerste gedeelte bevat een bespreking over de opdeling in de drie sets en het effect van de dynamische analyse op de grijze set. Het laatste gedeelte bevat de broncode van het voorbeeld.

Het doel van de voorbeeldapplicatie is om expliciet bij enkele constructies die in een applicatie kunnen optreden stil te staan. Dit zijn namelijk:

- Dode code detectie wanneer er sprake is van een hiërarchische structuur
- Dode code detectie wanneer er sprake is van polymorfisme
- Dode code detectie wanneer er sprake is van shadowing
- Dode code detectie wanneer er sprake is van speciale constructies zoals selectie-logica
- Dode code detectie van ongebruikte interfaces
- Dode code detectie wanneer er sprake is van impliciete aanroepen
- Dode code detectie wanneer er gebruik wordt gemaakt van een abstract factory
- Dode code detectie wanneer er gebruik wordt gemaakt van aanroepen naar super
- Dode code detectie wanneer in argumenten methode-aanroepen plaatsvinden

Bepaling entry-point
employee.EmployeeExample#main(String [])

Beantwoording van de vraag "Wat is dode code?"
Per regel zal nu worden bepaald welke aanroepen er allemaal mogelijk zijn. Deze informatie wordt vervolgens gebruikt om dode code set te bepalen. De aanroepen naar methodes uit de Java bibliotheken worden buiten beschouwing gelaten.
Onder het kopje "mogelijke aanroepen" bevinden zich steeds de aanroepen die als gevolg van polymorfisme en de daarbij behorende onzekerheid in aanmerking komen om runtime te worden uitgevoerd. De indentatie geeft aan vanuit welke methode welke andere methode wordt aangeroepen.

1. `Boss chief = new Boss("Chief")`
   - Mogelijke aanroepen:
     1. employee.Boss#<init>(String)
        - employee.Employee#setName(String)
2. `Employee robot = employeeFactory(ROBOT)`
   - Mogelijke aanroepen:
     1. employee.EmployeeExample#employeeFactory(int)
        - employee.CEO#<init>(String)
          - employee.Boss#<init>(String)
            - employee.Employee#setName(String)
        - employee.Boss#<init>(String)
          - employee.Employee#setName(String)
        - employee.Robot#<init>(String)
          - employee.Employee#setName(String)
3. `CEO ceo = new CEO();`
   - Mogelijke aanroepen:
     1. employee.CEO#<init>()
        - employee.Boss#<init>(String)
          - employee.Employee#setName(String)
4. `chief`
   - Impliciete aanroep
5. `robot.toString()`
   - Mogelijke aanroepen:
     1. employee.Robot#toString()
        - employee.Employee#toString()
        - employee.Robot#salary()
     2. employee.CEO#toString() (onzekerheid, wordt namelijk runtime niet aangeroepen door dit statement)
     3. employee.Employee#toString() (onzekerheid, wordt namelijk runtime niet aangeroepen door dit statement)

6. `ceo.getName()`
   - Mogelijke aanroepen:
     1. employee.CEO#getName()
7. `ceo.salary()`
   - Mogelijke aanroepen:
     1. employee.CEO#salary()
        - employee.Boss#salary()
          - employee.Boss#calculateSalary(int)
        - employee.CEO#salaryZeros()
        - employee.CEO#getsBonus()
8. `robot.departement()`
   - Mogelijke aanroepen:
     1. employee.Employee#departement() (onzekerheid, wordt namelijk runtime niet aangeroepen door dit statement)
     2. employee.Boss#departement() (onzekerheid, wordt namelijk runtime niet aangeroepen door dit statement)
     3. employee.CEO#departement() (onzekerheid, wordt namelijk runtime niet aangeroepen door dit statement)
     4. employee.Robot#departement()

Op basis van bovenstaande methode-aanroepen kan vervolgens het verschil worden genomen met alle methode-declaraties die bekend zijn. Daarnaast wordt er rekening gehouden met mogelijk impliciete aanroepen naar alle `finalize` methodes. Er is een keuze gemaakt om de `toString` in dit voorbeeld niet als impliciete aanroep te zien aangezien er ook een expliciete aanroep naar is. Dit levert de dode code set op:

Dode code set
- employee.Employee#getName()
- employee.Boss#getsBonus()
- employee.Robot#<init>()
- employee.Robot#calculateSalary(int)
- employee.Robot#isGoodRobot()

Nu de dode code set is berekend zal vervolgens de werkende code set moeten worden bepaald om zo te kunnen komen tot de vaststelling van de grijze set.

Beantwoording van de vraag “Wat is werkende code?”

1. `Boss chief = new Boss(“Chief”)`
   - Mogelijke aanroepen:
     1. employee.Boss#<init>(String)
        - employee.Employee#setName(String)
2. `Employee robot = employeeFactory(ROBOT)`
   - Mogelijke aanroepen:
     1. employee.EmployeeExample#employeeFactory(int)
        - employee.CEO#<init>(String)
          - employee.Boss#<init>(String)
            - employee.Employee#setName(String)
        - employee.Boss#<init>(String)
          - employee.Employee#setName(String)
        - employee.Robot#<init>(String)
          - employee.Employee#setName(String)
3. `CEO ceo = new CEO();`
   - Mogelijke aanroepen:
     1. employee.CEO#<init>()
        - employee.Boss#<init>(String)
          - employee.Employee#setName(String)
4. `chief`
   - Impliciete aanroep
5. `robot.toString()`
   - Door de aanname over polymorfisme en de optredende onzekerheid wordt geen enkele van de mogelijke aanroepen hier als “werkend” beschouwd.
6. `ceo.getName()`

- Mogelijke aanroepen:
  1. employee.CEO#getName()

7. `ceo.salary()`
   - Mogelijke aanroepen:
     1. employee.CEO#salary()
        - employee.Boss#salary()
          - employee.Boss#calculateSalary(int)
        - employee.CEO#salaryZeros()
        - employee.CEO#getsBonus()
8. `robot.departement()`
   - Door de aanname over polymorfisme en de optredende onzekerheid wordt geen enkele van de mogelijke aanroepen hier als “werkend” beschouwd.

Werkende code set
- employee.EmployeeExample#main(String[])
- employee.EmployeeExample#employeeFactory(int)
- employee.Employee#setName(String)
- employee.Boss#<init>(String)
- employee.Boss#salary()
- employee.Boss#calculateSalary(int)
- employee.CEO#<init>()
- employee.CEO#<init>(String)
- employee.CEO#getName()
- employee.CEO#salary()
- employee.CEO#salaryZeros()
- employee.CEO#getsBonus()
- employee.Robot#<init>(String)
- Impliciete aanroep die uit voorzorg als werkend wordt gezien om false positives tegen te gaan:
  - employee.Boss#finalize()

Op basis van bovenstaande twee sets kan de grijze set worden bepaald. Merk op dat zich in de werkende code set een false negative bevindt. De <init> van employee.CEO met als argument een String wordt nooit aangeroepen vanuit de main en is dus in feite dode code. De reden dat deze toch in de werkende code set staat komt doordat de selectielogica niet kan worden geëvalueerd.

Grijze set
- employee.Employee#toString()
- employee.Employee#departement()
- employee.Robot#salary()
- employee.Robot#departement()
- employee.Robot#toString()
- employee.Boss#departement()
- employee.CEO#toString()
- employee.CEO#departement()

Nu de drie sets bekend zijn kan de dynamische analyse worden toegepast. Aangezien het om een eenvoudig voorbeeld gaat waarbij maar één scenario mogelijk is (het starten van main) is hiermee de grijze set eenmalig te verkleinen.

Entry-points volgende uit dynamische analyse
- employee.Employee#toString()
- employee.Robot#toString()
- employee.Robot#Salary()
- employee.Robot#departement()

Als gevolg van de entry-points die gevonden zijn met de dynamische analyse zal de grijze set in omvang afnemen door de afname in onzekerheid.

Grijze set na herhalen statische analyse met gevonden entry-points uit dynamische analyse
- employee.Employee#departement()

- employee.Boss#departement()
- employee.CEO#toString()
- employee.CEO#departement()

Doordat er in het testvoorbeeld slechts sprake is van één scenario zal het toepassen van dynamische analyse na het opnieuw bepalen van de grijze set geen nieuwe entry-points meer opleveren. De dode code analyse is dus klaar en de grijze set kan als dode code worden beschouwd.

In het geval er sprake was geweest van een echte applicatie waarbij een willekeur aan scenario's mogelijk zou zijn dan zou men door moeten gaan tot dat de grijze set niet meer substantieel verandert om vervolgens te bepalen of men de grijze set als dode code ziet of niet.(zie paragraaf 4.3.2 en hoofdstuk 7)

Broncode
Hieronder zijn de bronbestanden opgenomen.

Klasse employee.EmployeeExample

```
package employee;

import java.util.Vector;

import javax.swing.JOptionPane;

public class EmployeeExample {

	public final static int CEO = 1;
	public final static int BOSS = 2;
	public final static int ROBOT = 3;

	// Confuser: ceo is a field declaration but is overriden
	// in the main method by a local variable declaration
	public static Employee ceo;

	public static Vector employees = new Vector();

	// USED Factory for creating an employee object
	public static Employee employeeFactory(int option) {
		switch(option) {
		case CEO :
			return new CEO("The president");
		case BOSS :
			return new Boss("Boss");
		case ROBOT :
			return new Robot("Robot");
		default :
			return null;
		}
	}

	// ENTRY-POINT
	public static void main(String [] args) {

		// Assignment of Boss object to variable with type Boss
		Boss chief = new Boss("Chief");

		// Assignment of Robot object to variable with type Employee
		// Here "polymorphism" plays a role explicitly because the type
		// of the assignment isn't the same as the variable type.
		Employee robot = employeeFactory(ROBOT);

		// Assignment of CEO object to variable with type CEO
		// No classes extend CEO so polymorfism is less difficult here.
```

```
            CEO ceo = new CEO();

            // Implicit call to toString of variable chief.
            System.out.println(chief);
            // Explicit call to toString of variable robot
            System.out.println(robot.toString());
            // Explicit call to the getName() of class CEO
            String ceoName = ceo.getName();

            // Explicit call to the salary method of class CEO
            // Just a conditional check.
            System.out.println(ceoName);
            if(ceoName.equals("The boss") && ceo.salary() > 5000) {
                  System.out.println("Ok");
            } else {
                  System.out.println("Not Ok");
            }

            // Explicit call to toString of variable robot
            System.out.println(robot.departement());

            // Call to a library method
            JOptionPane.showMessageDialog(null,"Example done");
      }
}
```

Klasse employee.Employee

```
package employee;

public abstract class Employee {

      protected String name;

      // DEAD-CODE
      // There is a call to the getName() of CEO but this method
      // is overriden in the class CEO so this one isn't used.
      public String getName() {
            return this.name;
      }

      // USED
      // This method is used by constructors for initializing the name
      public void setName(String name) {
            this.name = name;
      }

      // USED (not really a method declaration)
      // This is an abstract method but it is overriden in other classes
      // and used from the entry-point
      public abstract int salary();

      // DEAD-CODE
      public String departement() {
            return "unknown";
      }

      // USED
      // This method is used by the robot.toString in a call to super.
      // Because robot.toString is called this method is used as well.
      public String toString() {
            return "Employee: " + this.name;
```

```
	}
}
```

Klasse employee.Boss

```
package employee;

public class Boss extends Employee {

	private final int SALARY = 100;

	// USED
	// This method is called fromt the salary() method.
	public Boss(String name) {
		setName(name);
	}

	// USED
	// This method is called fromt the salary() method.
	private int calculateSalary(int salary) {
		return salary * 10;
	}

	// USED
	// This method is explicitly used from within the CEO class. Because
	// the method in the CEO class is called this method is called also
	// but in an indirect way.
	public int salary() {
		return calculateSalary(SALARY);
	}

	// DEAD-CODE
	// This method is never called by this method or any subclasses
	protected boolean getsBonus() {
		return false;
	}

	// DEAD-CODE
	public String departement() {
		return "MainDepartement";
	}

	// USED
	// This method is implicitly used by the garbage collector. The
	// method doesn't call super.finalize() to make sure that no explicit
	// call can be found to any finalize method.
	protected void finalize() throws Throwable{
		// Get rid of the name
		this.name = "";
	}

}
```

Klasse employee.CEO

```
package employee;

public class CEO extends Boss {

	// USED
	// This constructor is used in the example main
	public CEO () {
		super("The CEO");
```

```
	}

	// USED
	// This constructor is used in the example main factory
	public CEO (String name) {
		super(name);
	}

	// USED
	// This method overrides the getName method that is standard in
	// the employee. Because of this the getName() of employee is never
	// used.
	public String getName() {
		return "The boss";
	}

	// DEAD-CODE
	public String departement() {
		return "HeadDepartement";
	}

	// USED
	// This method is explicitly used from the main entry-point. It uses
	// the salary method of the superclass.
	public int salary() {
		int calculatedSalary = super.salary();

		for(int i=0; i < salaryZeros(); i++) {
			calculatedSalary *= 10;
		}

		if(this.getsBonus()) {
			calculatedSalary *= 1.1;
		}

		return calculatedSalary;
	}

	// USED by salary method.
	// Overrides the protected method from superclass
	protected boolean getsBonus() {
		return true;
	}

	// USED by salary method.
	private int salaryZeros() {
		return 5;
	}

	// DEAD-CODE
	// This method is never called from the entry-point.
	public String toString() {
		return "You'r fired!";
	}
}
```

Klasse employee.Robot

```
package employee;

public class Robot extends Employee {
```

```
	// DEAD-CODE
	// This constructor is not used.
	public Robot() {name = "unknown";}

	// USED
	// This constructor is used in the example main
	public Robot(String name) {
		setName(name);
	}

	// USED
	public String departement() {
		return "Factory";
	}

	// DEAD-CODE
	// Used for testing the dead code. This is a private method which
	// isn't called within the class. This code is dead.
	private int calculateSalary() {
		if(isGoodRobot()) {
			return Integer.MAX_VALUE;
		} else {
			return Integer.MIN_VALUE;
		}
	}

	// DEAD-CODE
	// Used as a method for calculatin the salary in the private method.
	// This method isn't called directly or indirectly
	private boolean isGoodRobot() {
		return true;
	}

	// USED
	// This method isn called from the toString
	public int salary() {
		return 0;
	}

	// USED
	// Does a call to super so the toString of the Employee class is used
	// when this toString is called.
	public String toString() {
		return super.toString() + " : " + salary();
	}
}
```

Interface employee.TemporaryEmployee

```
package employee;

public interface TemporaryEmployee {
	// DEAD-CODE (not really a method declaration)
	public int contractNumber();
	// DEAD-CODE (not really a method declaration)
	public String employmentAgency();
}
```

H – Gebruik van de dode code analyse

In deze bijlage zal het gebruik van de dode code analyse worden besproken. Er zal worden ingegaan op de manier waarop de verschillende fases moeten worden gebruikt om de analyse te kunnen uitvoeren. Deze analyse is onderhevig aan de beperkingen zoals besproken in bijlage F.

Voorbereidingen statische analyse:
Aangezien de eerste twee fases van de dode code analyse gebaseerd zijn op ASF+SDF specificaties zal de Meta-omgeving moeten worden geïnstalleerd wanneer men wijzigingen wil maken in de verschillende fases. Voor meer informatie over de installatie van de Meta-omgeving kan worden gekeken op http://www.meta-environment.org. Voor de ontwikkeling van de dode code analyse is gebruik gemaakt van de 1.5.3 stable release.

Na het installeren van de ASF+SDF omgeving moet de laatste versie van het Relational Calculus pakket uit Subversion worden gehaald. Deze zal vervolgens net zoals bij de Meta-omgeving met behulp van onderstaande commando's moeten worden geïnstalleerd:

```
./configure –-prefix=<directory-to-install>
make install
```

Na het installeren van Relational Calculus kan het analyse pakket worden geïnstalleerd. Ook hiervan moet de laatste versie uit SubVersion worden gehaald zodat het pakket, java-source-analysis, kan worden geïnstalleerd met onderstaande commando's:

```
./configure –-prefix=<directory-to-install>
make install
```

Na bovenstaande twee stappen zijn er twee pakketen geïnstalleerd in de directory die als prefix is meegegeven met het `configure` commando. In beide directories bevinden zich in /bin de binaries en shell-scripts die nodig zijn voor de uitvoering van de analyse. Om eenvoudig gebruik hiervan te kunnen maken moeten deze aan het PATH van de gebruikte shell worden toegevoegd.

De broncode van de belangrijkste onderdelen bevindt zich in het eerder opgehaalde java-source-analysis pakket uit Subversion. Hieronder worden deze even kort aangestipt.

Directory	***Omschrijving***
src/grammar	Hierin bevindt zich de Java-grammatica die wordt gebruikt voor het parseren van de Java bronbestanden. Deze syntax definitie wordt gebruikt om de feiten te kunnen extraheren in de eerste fase.
src/extraction	Hierin bevinden zich de verschillende SDF en ASF definities voor het extraheren van de relaties uit de broncode. Het gaat hier om de eerste fase van de feiten-extractie zoals besproken in paragraaf 5.2.2. De hoofdmodule die moet worden geopend wanneer de feitenextractie in de Meta-omgeving moet worden geladen is `JavaAnalysis`.
src/enrich	In deze directory staan de SDF en ASF definities voor de tweede fase van de analyse ofwel de verrijking van de relaties. De hoofdmodule die moet worden geopend wanneer de feitenverrijking in de Meta-omgeving moet worden geladen is `EnrichAnalysis`
src/analyses src/analyses/deadcode src/analyses/metrics	Hierin bevinden zich verschillende rscripts welke als voorbeeld dienen voor de manier waarop Rscript kan worden gebruikt om analyses uit te voeren. Hierbij moet worden opgemerkt dat deze niet zijn aangepast aan het nieuwe relationele model zoals deze voor dit onderzoek is opgesteld. Naast de voorbeelden bevinden zich in deze directory ook het Rscript `DeadCodeDetection.rscript` en `measure.rscript`. Dit eerste script kan worden gebruikt om (gegeven de beperkingen in bijlage F) de opdeling in de grijze set, dode code set en werkende code set te maken. Het tweede script kan worden gebruikt om een indruk te krijgen van de omvang van een Rstore waarin de geëxtraheerde relationele informatie is opgeslagen.

Uitvoering statische analyse:
Na het uitvoeren van bovenstaande voorbereidingen is het mogelijk om de statische analyse te gaan uitvoeren. De allereerste stap is om de feitenextractie (1e fase) op een directory met Java bronbestanden te gaan uitvoeren die men wil analyseren. Dit kan met behulp van onderstaand commando:

```
extract-java -i <directory> -o result.rstore.pt
```

Met `<directory>` wordt het pad naar de directory bedoeld waarin zich de Java bronbestanden bevinden. Deze directory wordt recursief doorzocht voor Java bronbestanden. Wanneer een nieuwe versie van de feiten-extractie fase moet worden opgenomen dan zal men de equations moeten exporteren vanuit de Meta-omgeving en deze compileren met asfc en de optie -a om de annotaties goed te kunnen verwerken. Wanneer dit rechtstreeks wordt gedaan vanuit de Meta-omgeving wordt de lokatie-informatie verkeerd geëvalueerd.

Met behulp van onderstaand command kan vervolgens de feitenverrijking worden uitgevoerd over de eerder gegenereerde `result.rstore.pt` uit de eerste fase.

```
./enrichAnalysis -f enrichRelations -r "RSTORE" -i result.rstore.pt -o
                        result_enriched.rstore.pt
```

Na het uitvoeren van bovenstaand commando ontstaat het bestand `result_enriched.rstore.pt`. Deze voldoet (onder de beperkingen uit bijlage F) aan het relationele model zoals deze in bijlage C is besproken. Deze kan vervolgens worden gebruikt voor het uitvoeren van de dode code detectie met het Rscript. Hiervoor moet onderstaand commando worden uitgevoerd vanuit de directory waarin zich zowel de `result_enriched.rstore.pt` bevindt als de `DeadCodeDetection.rscript`.

```
rscript -i DeadCodeDetection.rscript -s result_enriched.rstore.pt
```

Afhankelijk van de informatie die men wil achterhalen is het benodigd om enkele `yield <var>` regels uit te commentaren of juist vanuit het commentaar actief te maken. Op deze manier kan men aangegeven welke informatie wel en welke niet moet worden afgedrukt, Standaard wordt het aantal elementen in de grijze set, dode code set en werkende code set afgedrukt en de elementen in de grijze code set. Deze grijze code set kan vervolgens worden gebruikt om de dynamische analyse uit te voeren.

Voorbereiding dynamische analyse:
Voor het uitvoeren van de dynamische analyse is het benodigd om het AspectJ pakket te installeren. Deze kan worden gedownload van http://www.eclipse.org/aspectj/ . Binnen het afstudeeronderzoek is gebruik gemaakt van versie 1.5.1. Na het downloaden van AspectJ kan deze met de bijgeleverde installer worden geïnstalleerd. Hierna kan de AspectJ compiler worden gebruikt.

Uitvoering dynamische analyse:
Voordat het mogelijk is om de dynamische analyse uittevoeren moeten de methodes die zijn opgenomen in de grijze set worden omgezet naar het join point formaat dat binnen AOP wordt gebruikt. Een alternatief is om alle methode-aanroepen als Join Point te beschouwen. Zie voor voorbeelden de Cd-rom die bij dit afstudeeronderzoek behoord.

Na het opstellen van de join points is het nodig om het geheel te compileren met behulp van de AspectJ compiler. Hiervoor kan onderstaand commando worden gebruikt.

```
ajc *.java
```

Vervolgens kan het Java-programma worden gestart door de Main van het programma te draaien. Vervolgens zullen de entry-points worden afgedrukt in de error-stream van het programma. Deze kunnen vervolgens weer als input worden gebruikt binnen de statische analyse om de entry-points te specificeren. Op deze manier kan de statische analyse weer opnieuw worden uitgevoerd met de informatie vanuit de dynamische analyse.

Hieronder is de uitvoer gegeven van de uitvoering van het DeadCodeDetection rscript op de rstore die is ontstaan na het uitvoeren van de feitenverrijking op de geëxtraheerde feiten in de eerste fase van het voorbeeldprogramma.

Commando	rscript -i DeadCodeDetection.rscript -s ./rstores/automatic_extracted _employee.rstore.pt
Uitvoer	rstore (< number-of-grey-methods , int , 8 > , < grey-code-set , set [method-d ecl] , { < "departement" , { < "employee" , 1 > , < "CEO" , 2 > } , { "public" } , { < "String" , 1 > } , { } , area-in-file ("./CEO.java" , area (26 , 1 , 28 , 2 , 489 , 60)) > , < "toString" , { < "employee" , 1 > , < "CEO" , 2 > } , { "public" } , { < "String" , 1 > } , { } , area-in-file ("./CEO.java" , are a (60 , 1 , 62 , 2 , 1179 , 54)) > , < "departement" , { < "employee" , 1 > , < "Robot" , 2 > } , { "public" } , { < "String" , 1 > } , { } , area-in-file ("./Robot.java" , area (16 , 1 , 18 , 2 , 270 , 52)) > , < "salary" , { < "em ployee" , 1 > , < "Robot" , 2 > } , { "public" } , { < "int" , 1 > } , { } , ar ea-in-file ("./Robot.java" , area (40 , 1 , 42 , 2 , 837 , 36)) > , < "toStr ing" , { < "employee" , 1 > , < "Robot" , 2 > } , { "public" } , { < "String" , 1 > } , { } , area-in-file ("./Robot.java" , area (47 , 1 , 49 , 2 , 1003 , 7 5)) > , < "departement" , { < "employee" , 1 > , < "Boss" , 2 > } , { "public" } , { < "String" , 1 > } , { } , area-in-file ("./Boss.java" , area (34 , 1 , 36 , 2 , 740 , 60)) > , < "departement" , { < "employee" , 1 > , < "Employee " , 2 > } , { "public" } , { < "String" , 1 > } , { } , area-in-file ("./Emplo yee.java" , area (26 , 1 , 28 , 2 , 617 , 52)) > , < "toString" , { < "employ ee" , 1 > , < "Employee" , 2 > } , { "public" } , { < "String" , 1 > } , { } , area-in-file ("./Employee.java" , area (33 , 1 , 35 , 2 , 814 , 64)) > } > , < number-of-working-methods , int , 14 > , < number-of-dead-methods , int , 8 >)

Vervolgens zijn de bovenstaande grijze methodes als join points opgenomen. Hieronder is de representatie in de vorm van een pointcut weergegeven.

Pointcut	pointcut tracePoints() : execution(public String employee.Employee.toString()) ‖ execution(public String employee.Employee.departement()) ‖ execution(public int employee.Robot.salary()) ‖ execution(public String employee.Robot.departement()) ‖ execution(public String employee.Robot.toString()) ‖ execution(public String employee.Boss.departement()) ‖ execution(public String employee.CEO.toString()) ‖ execution(public String employee.CEO.departement());

Na het specificeren van de pointcut kan zoals eerdere aangegeven het geheel gecompileerd worden met de AspectJ compiler. Hieronder staat het resultaat van de uitvoering van deze gecompileerde versie van het voorbeeldprogramma en de geconstateerde entry-points als gevolg van de dynamische analyse.

Commando	java employee.EmployeeExample 2> entrypoints.txt
Inhoud entrypoints.txt bestand	< "toString", { < "employee" , 1 >, < "Employee" , 2 > }, { } > < "toString", { < "employee" , 1 >, < "Robot" , 2 > }, { } > < "salary", { < "employee" , 1 >, < "Robot" , 2 > }, { } > < "departement", { < "employee" , 1 >, < "Robot" , 2 > }, { } >

De entry-points vanuit de dynamische analyse kunnen vervolgens weer worden ingevoerd in het DeadCodeDetection.rscript door naast de Main methode ook bovenstaande methode-beschrijvingen als entry-point te beschouwen. Na deze aanpassing en de uitvoering van het Rscript ontstaat onderstaande uitvoer.

Command	rscript -i DeadCodeDetectionAfterDynamic.rscript -s ./rstores/automatic_extracted_employee.rstore.pt
Uitvoer	rstore (< number-of-grey-methods , int , 4 > , < grey-code-set , set [method-decl] , { < "departement" , { < "employee" , 1 > , < "CEO" , 2 > } , { "public" } , { < "String" , 1 > } , { } , area-in-file ("./CEO.java" , area (26 , 1 , 28 , 2 , 489 , 60)) > , < "toString" , { < "employee" , 1 > , < "CEO" , 2 > } , { "public" } , { < "String" , 1 > } , { } , area-in-file ("./CEO.java" , area (60 , 1 , 62 , 2 , 1179 , 54)) > , < "departement" , { < "employee" , 1 > , < "Boss" , 2 > } , { "public" } , { < "String" , 1 > } , { } , area-in-file ("./Boss.java" , area (34 , 1 , 36 , 2 , 740 , 60)) > , < "departement" , { < "employee" , 1 > , < "Employee" , 2 > } , { "public" } , { < "String" , 1 > } , { } , area-in-file ("./Employee.java" , area (26 , 1 , 28 , 2 , 617 , 52)) > } > , < number-of-working-methods , int , 18 > , < number-of-dead-methods , int , 8 >)

Het aantal gevonden dode methode staat in dit geval (zoals in bovenstaande uitvoer is te zien) op acht. Dit is niet in overeenstemming met het aantal gevonden dode methode zoals beschreven in bijlage G. Dit komt doordat in de uitvoer nog niet de abstracte implementaties zijn gefilterd ofwel het is een puur praktische beperking. De

`salary` methode van employee.Employee wordt als dode code gezien terwijl dit een abstracte implementatie is en dus uit het resultaat dient te worden gefiltered aangezien er implementaties van deze abstractie definitie zijn die niet als dode code worden beschouwd (zie beperking DC-5 in bijlage F). Daarnaast worden de definitie van de methode uit de interface als dood beschouwd en deze worden ook daadwerkelijk niet gebruikt omdat het interface volledig ongebruikt is.

Hieronder staat de gevonden dode code set als gevolg van de uitvoering van het Rscript na het uitcommentaren van de yield van de grijze set en het opnemen van een yield voor de dode code set.

Command	rscript -i DeadCodeDetectionAfterDynamic.rscript -s ./rstores/automatic_extracted_employee.rstore.pt
Uitvoer	rstore (< number-of-grey-methods , int , 4 > , < number-of-working-methods , int , 18 > , < number-of-dead-methods , int , 8 > , < dead-code-set , set [method-decl] , { < "employmentAgency" , { < "employee" , 1 > , < "TemporaryEmployee" , 2 > } , { "public" , "abstract" } , { < "String" , 1 > } , { } , area-in-file ("./TemporayEmployee.java" , area (9 , 1 , 9 , 34 , 188 , 33)) > , < "contractNumber" , { < "employee" , 1 > , < "TemporaryEmployee" , 2 > } , { "public" , "abstract" } , { < "int" , 1 > } , { } , area-in-file ("./TemporaryEmployee.java" , area (6 , 1 , 6 , 29 , 108 , 28)) > , < "salary" , { < "employee" , 1 > , < "Employee" , 2 > } , { "public" , "abstract" } , { < "int" , 1 > } , { } , area-in-file ("./Employee.java" , area (23 , 1 , 23 , 30 , 570 , 29)) > , < "getName" , { < "employee" , 1 > , < "Employee" , 2 > } , { "public" } , { < "String" , 1 > } , { } , area-in-file ("./Employee.java" , area (10 , 1 , 12 , 2 , 237 , 48)) > , < "getsBonus" , { < "employee" , 1 > , < "Boss" , 2 > } , { "protected" } , { < "boolean" , 1 > } , { } , area-in-file ("./Boss.java" , area (29 , 1 , 31 , 2 , 672 , 50)) > , < "isGoodRobot" , { < "employee" , 1 > , < "Robot" , 2 > } , { "private" } , { < "boolean" , 1 > } , { } , area-in-file ("./Robot.java" , area (34 , 1 , 36 , 2 , 731 , 49)) > , < "calculateSalary" , { < "employee" , 1 > , < "Robot" , 2 > } , { "private" } , { < "int" , 1 > } , { } , area-in-file ("./Robot.java" , area (23 , 1 , 29 , 2 , 462 , 129)) > , < "<init>" , { < "employee" , 1 > , < "Robot" , 2 > } , { "public" } , { } , { } , area-in-file ("./Robot.java" , area (7 , 1 , 7 , 35 , 108 , 34)) > } >)

Zoals aangegeven in bijlage G kunnen de overgebleven methodes in de grijze code set ook als dode code worden beschouwd aangezien er geen scenario's zijn te bedenken waarin ze wel zouden worden gebruikt. Dit komt door het relatief eenvoudige karakter van de voorbeeldapplicatie.

I – Flow Diagrammen