Intreerede prof.dr. Jurgen J. Vinju

# Automatische software-analyse in context

Uitgesproken op 12 februari 2016
aan de Technische Universiteit Eindhoven

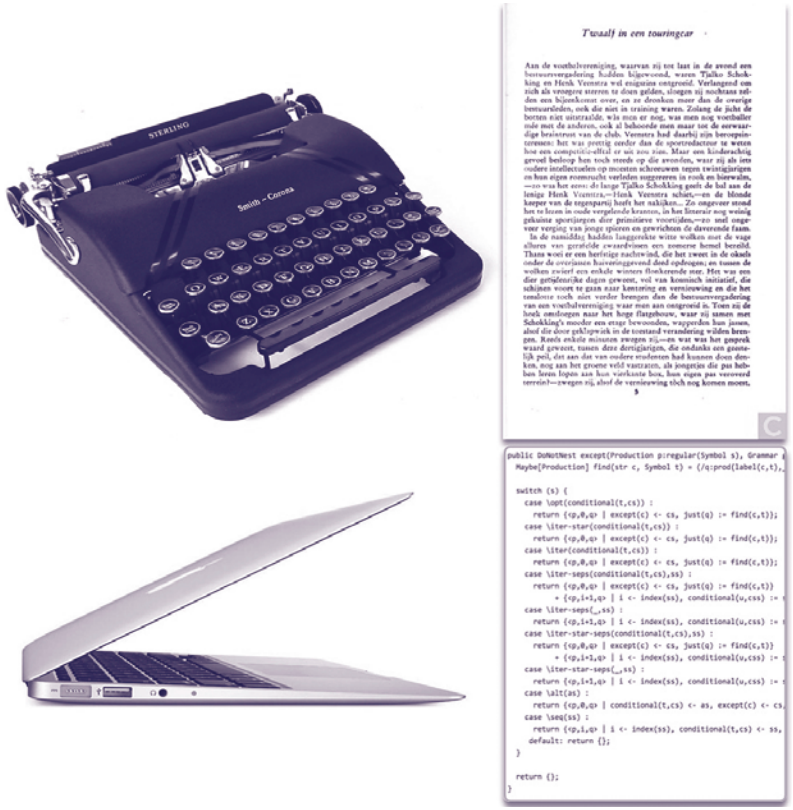
# Inleiding

**Software is een interessant concept.** Het is zowel een technologie als een economische aandrijfmotor en zelfs een maatschappelijk fenomeen. Wat is software nou eigenlijk? Waarom zouden we software automatisch analyseren? Software is een relatief jonge technologie en niet gemakkelijk vergelijkbaar met bestaande technologieën. Vergelijkingen met bijvoorbeeld bouwkunde of elektrotechniek slaan zelfs vaak de plank helemaal mis. Naast het nut en de uitdagingen van onderzoek ten behoeve van software-analyse, zal ik daarom eerst ingaan op wat bijzonder is aan software als technologie en waarom we het analyseren van software nóg beter onder de knie moeten gaan krijgen. Daarna motiveer ik gereedschappen voor automatische software-analyse en bespreek ik de technische kansen en wetenschappelijke uitdagingen van dit onderzoeksgebied.

Software wordt nog wel eens onderschat als onderzoeksgebied en nog vaker overschat als een geheimzinnige kunst. Is software slechts een representatie van bestaande wis- en natuurkundige kennis? Is software een ingewikkeld instrument, gehanteerd door een elite, die buiten onze controle onze wereld aan het veranderen is? Er bestaat ook een nuchtere kijk op software; een definitie van het concept die het lezen en schrijven van software niet verkleint tot een triviale representatie van iets anders en het ook niet vergroot tot een bovenmenselijke kunst.

**Software is een vorm van taal**. Net zoals blauwdrukken van gebouwen en muziekschrift ook een vorm van taal zijn, is software een vorm van geschreven taal. Een bijzonder ingrediënt van het concept software is dat de zenders en ontvangers van deze communicatievorm zowel mensen als machines zijn. De voor ons leesbare vorm van software heet broncode. Machines voeren software uit, de instructies op mechanische wijze interpreterend. Wij schrijven broncode om dit te bewerkstelligen. We lezen en analyseren de broncode vaker dan we het schrijven. Dit doen we om te leren wat wellicht de bedoeling is geweest of om een fout te kunnen repareren. We lezen ook broncode om de kwaliteit ervan te controleren. Meestal is het doel van het lezen van broncode om het daarna te kunnen uitbreiden met een nieuwe functionaliteit.

Software verhoudt zich tot computer- en netwerkapparatuur, zoals literatuur zich verhoudt tot de boekdrukkunst. Net zoals het gewone schrift algemeen is, is software onvoorstelbaar universeel: je kunt er muziek mee beschrijven, maar ook tekenfilms, journalistieke media, bedrijfsadministraties, weersvoorspellingen, wetten, telecommunicatieprotocollen, gezelschapsspelletjes, enzovoorts. Net zoals met het gewone schrift kunnen we – in letterlijke zin – fantastische constructies beschrijven met software. Dit is de reden van mijn persoonlijke fascinatie voor software: programmeren is een vorm van creativiteit die slechts beperkt wordt door je eigen fantasie.




Figuur 1

De eerste pagina uit ‘De kellner en de levenden’, geschreven door Simon Vestdijk op een typemachine in het Nederlands en verspreid met behulp van de boekdrukkunst en vrachtwagens. Daarnaast een gedeelte van de grammaticale analyse van programmeertalen, geschreven op een Macbook Pro in Rascal en verspreid via het Internet. Software is in eerste instantie net zoals natuurlijke taal, niet het medium maar de boodschap.

Hoewel software in de vorm van broncode erg abstract is, beleven we de effecten van deze broncode toch zeer intens. Je kunt het effect van software niet alleen ervaren door middel van randapparatuur, zoals beeldschermen, luidsprekers en printers maar ook door allerlei andere actuatoren, zoals de uitbetaling van uitkeringen, injectiespruitstukken in verbrandingsmotoren en wasmachine-programma's. De krachtige mobiele computer in je binnenzak – de smartphone – maakt deze beleving van software nog intenser.

Omdat software zo'n directe invloed op ons heeft, zou iedereen een behoorlijk goed idee moeten hebben van wat software is en wat het doet: digitale geletterdheid is noodzakelijk. Maar zelfs de experts weten geen direct antwoord te geven op simpele vragen over bestaande softwaresystemen. De neveneffecten van het gebruik van software zijn wel belangrijk, maar tegelijk ondoorzichtig en lastig uit te leggen. Om de meeste software te kunnen gebruiken, hoef je deze niet in het minst te begrijpen – wie tot tien kan tellen, kan de hele wereld bellen – terwijl je ondertussen kwetsbaar bent. Wist u dat e-mails en tekstberichten in de regel 'zonder envelop' verstuurd worden en dat in principe iedereen onderweg kan meelezen? Welke informatie wordt er over je verzameld terwijl je werkt, speelt, koopt en afspreekt? En bij wie komt die informatie terecht en hoe wordt die informatie gebruikt? Als gebruikers van software beleven we wel de effecten maar realiseren we ons bijna nooit de oorzaak: wat er geschreven is in de broncode.

Een software engineer daarentegen beleeft vooral de broncode van software en het ontwerp van de broncode. Wie software analyseert, doet dat zoals een criticus een roman analyseert: op allerlei verschillende niveaus van detail en vanuit allerlei perspectieven. Om genoeg begrip te verkrijgen, doen alle niveaus van detail er toe. Een prachtig plot van een roman kan volkomen teniet worden gedaan door brakke zinsbouw en slechte woordkeus, net zoals een briljante software-architectuur verpest kan worden door onleesbare broncode waarin een privacy-lekje verstopt zit. Het volgende niet uitputtende overzicht illustreert dat software op net zoveel manieren geanalyseerd en begrepen kan worden als bijvoorbeeld een roman:

| **Natuurlijke taal** | **Softwaretaal** |
|---|---|
| Spelling en leestekens | Lexicale syntax |
| Zinsbouw | Context-vrije syntax |
| Paragrafen | Procedures, functies |
| Semantiek | Statische en dynamische semantiek |
| Hoofdstukken | Modules, componenten |
| Verhaal, plot | Features, user-stories |
| Raamvertelling | Procedure-aanroepen |
| Karakterontwikkeling | Gebruikersinterface |
| Thematiek | Applicatiedomein |
| Intertekstualiteit | Dependency, reuse |
| Creatief proces | Ontwikkelingsproces |
| Schrijfmateriaal | Programmeeromgeving |

**Wie software kan lezen en schrijven, heeft macht**. Software is werkelijk overal aanwezig. Doordat hedendaagse computerapparatuur klein maar krachtig is, overal in verstopt wordt en volledig verbonden is via internet, heeft software als taal zo'n enorme reikwijdte in onze samenleving. Wie software schrijft, heeft daadwerkelijk macht. Wie programmeert, definieert niet alleen de toekomst in de zin van welke instructies de machines zullen gaan uitvoeren, maar bakent ook de mogelijke interacties van deze machines met hun context af. Professionele software engineers kennen deze grote verantwoordelijkheid en dragen er zorg voor dat software in nauw overleg met alle belanghebbenden ontworpen wordt.

Het is een Achilles-hiel van software dat je om nieuwe software te ontwikkelen geen volkomen begrip hoeft te hebben van het gedrag ervan in een veranderende omgeving. Dit betekent dat er nu meer willekeurige software geschreven wordt dan we waarschijnlijk ooit zullen gaan begrijpen. Wist u dat er zelfs raketgeleerden zijn die hun miljoenenprojecten hebben zien ontploffen, omdat ze niet wisten dat hun oude software het niet goed zou doen op hun nieuwe raket? Software is moeilijker dan 'rocket science'. Waarom geloven we dan klakkeloos dat de privacygevoelige gegevens over onze kinderen veilig staan opgeslagen op de administratieve websites die onze scholen gebruiken? Ik geloof het niet zomaar, want de auteurs van een softwaresysteem zijn niet de meest objectieve bron van informatie over de interne kwaliteit van de broncode, laat staan de verkopers. Dus: we moeten leren de broncode van cruciale software objectief te analyseren totdat we alle belangrijke vragen erover kunnen beantwoorden binnen een redelijk budget.

**Intermezzo**. Geachte politici, juristen, economen en collega wetenschappers, let goed op wat de invloed is van personen die onze software schrijven op onze maatschappij, op ons privéleven, op onze ondernemingen, op onze onderzoeksresultaten. Politici, kunt u wel beoordelen, laat staan controleren, of uw beleid wél of niet wordt geïmplementeerd? Kunnen de auteurs van de software dit zelf? Juristen, kunnen de experts waaraan u het oordeel over software in vertrouwen overlaat eigenlijk wel beoordelen wat u van ze vraagt? Economen, wordt de productie van en handel in software überhaupt wel volledig gemeten in economische zin? Wordt de waarde van software wel goed gerepresenteerd in onze boekhoudingen en wordt deze ook jaarlijks afgeschreven? Met behulp van welke vuistregel komt de afschrijving dan tot stand? Wordt het onvermijdelijke softwareonderhoud wel gebudgetteerd? Wetenschappers, wordt de software die u gebruikt in uw data-gestuurde onderzoeksmethoden wel met voldoende scepsis geanalyseerd? Hoe beoordeelt u dit? Wie controleert uw oordeel?

**Softwaretaal is niet hetzelfde als natuurlijke taal**. De vergelijking met natuurlijke taal is inzichtelijk, maar we moeten ook niet te ver gaan met vergelijkingen. Wie leest nu de broncode van de software die hij of zij wil gebruiken? Wie herschrijft nou een roman omdat intussen de maatschappelijke context drastisch is veranderd? Wie vindt Shakespeare's 'Romeo and Juliet' een onleesbaar prul, omdat er volgens het hedendaagse Engels allerlei spelfouten in staan? Wie schrijft samen een epos dat niemand ooit echt kan lezen omdat dat langer dan een mensenleven tijd zou kosten? Voor software engineers is dit allemaal dagelijkse realiteit.

Software is veel dynamischer dan literatuur en andere technologieën zoals bouwkunde of elektrotechniek. Het is een vergissing om te denken dat dit alleen zo is omdat software nou eenmaal makkelijker te veranderen is dan een al gebouwd treinstation, een al gedrukt boek of een al geëtste printplaat. Een minstens net zo belangrijke oorzaak is dat software vaak erg direct verbonden is met haar context. Softwaregebruikers zijn actief afhankelijk van software; daardoor zal succesvolle software voortdurend meeveranderen met de zich ontwikkelende context (panta rei). Oudere software is in de regel ook software die heel vaak veranderd is en heel vaak uitgebreid is. Dit is de aanleiding voor het vakgebied Software Evolutie dat software bestudeert vanuit een meer biologisch perspectief dan een taalkundig perspectief; in termen van groei, metamorfose en zichzelf manifesterende eigenschappen als gevolg hiervan. Het inzicht dat, in tegenstelling tot het geschreven woord, software steeds moet groeien en meeveranderen in zijn context is sterk bepalend voor het verdere onderzoek in software-analyse. Hierover straks meer.

Figuur 2

De barokke Nationale Bibliotheek van de Tsjechische Republiek in Praag. We schrijven vandaag meer software in meer verschillende talen dan we gewone boeken schrijven in natuurlijke talen. Het is de uitdaging van het vakgebied software-analyse om al deze software te kunnen analyseren, door het leren beheersen van al deze barokke variëteit.

Het inzicht dat de essentie van software taal is, past wel goed bij de realiteit dat mensen letterlijk duizenden programmeertalen tot leven hebben geroepen, dat miljoenen andere mensen deze talen hebben leren beheersen en dat zij samen al miljarden pagina's broncode hebben geschreven. Software als taal impliceert ook dat we deze broncode niet alleen moeten kunnen schrijven, maar ook moeten kunnen lezen en interpreteren. De in de praktijk geobserveerde grilligheid van de productiviteit van software engineers past ook goed bij het beeld dat software taal is: net zo grillig als het voorspellen van de creativiteit van een romanschrijver.

De enorme variëteit aan programmeertalen en programmeertechnologieën is sterk gerelateerd aan de menselijke factor: software engineers die met hun opleidingsniveau en expertisegebied zowel als schrijvers en als lezers de situatie sterk beïnvloeden. Het onderzoek naar betere gereedschappen voor software-analyse is hierdoor gevangen in een haast paradoxale cyclus van het hebben van invloed op de toekomst van software engineering en het onvermijdelijk beïnvloed worden door de status quo. Het nieuwe gereedschap moet noodzakelijk passen bij de software engineers van vandaag.

**Intermezzo**. Aan de ene kant kunnen we als onderzoekers in software-analyse blij zijn met de luxe van de vrij directe toepasbaarheid (en valorisatie) van onze resultaten. Aan de andere kant moeten we oppassen dat het fundamentele onderzoek in software-analyse zich niet laat beperken tot de nogal vergankelijke context van de tegenwoordige tijd. Ook het verleden en de toekomst zijn belangrijke bronnen van inspiratie.

# Software-analyse

**Software-analyse is noodzakelijk**. Het kunnen analyseren van bestaande software is een centrale vaardigheid die het vak van software engineer kenmerkt. Op basis van begrip van de werking van software en het begrip van de context van de software kan hij zowel kleine als grote beslissingen nemen. Kleine beslissingen gaan over het hoe en waarom van het wijzigen van bestaande code om bijvoorbeeld een bug op te lossen of een feature toe te voegen. Grote beslissingen gaan bijvoorbeeld over het opnieuw ontwerpen van grote delen van een bestaand systeem, het verwijderen van verouderde componenten of het toevoegen van afhankelijkheden op externe software.

Zowel de kwaliteit van softwareproducten en -diensten als de ontwikkelingskosten ervan, staan direct in verband met de activiteit van het analyseren van software:

1. Zonder analyse kan software niet vertrouwd worden. Software geschreven op basis van onbegrip kan alleen maar per ongeluk goed zijn. Met behulp van accurate analyses kunnen weloverwogen beslissingen worden genomen zodat software expres goed is. Onbetrouwbare software doet niet alleen niet wat je wil (“Het mag niet van de computer!”, “Hij is gecrasht!”), maar het doet vaak ook meer dan je wil (“Wat heeft hij nou gedaan?”, “Hij is nog steeds bezig!”). Voorbeelden van ongewenst gedrag zijn het opslaan of verspreiden van foutieve informatie, het lekken van geheime informatie of het verbruiken van meer tijd, meer ruimte of meer energie dan noodzakelijk.
2. Fouten in software zijn kostbaar. Niet alleen het repareren van fouten is kostbaar, maar ook de impact van softwarefouten op de samenleving is kostbaar. Sommige software is veiligheidskritisch, bepalend voor het behoud van mensenlevens en andere software is van groot economisch belang, zoals het betalingsverkeer, de belasting en de uitkeringen. Vrijwel alle bedrijven en instellingen zouden significante schade kunnen oplopen door fouten in hun software. Niet alle fouten in software zijn observeerbaar door de gebruiker, maar die kunnen toch ontzettend duur zijn. Er bestaan sluimerende fouten die alleen tot uiting komen bij een veranderd gebruik (bijvoorbeeld piekbelasting of het aanbreken van een nieuw millennium) en daarmee plotseling de eigenaar in ernstige verlegenheid kunnen brengen. Er bestaan ook ontwerpfouten in de broncode die geen acute problemen opleveren, maar wel een

alsmaar groeiende complexiteit creëren en daarmee de eigenaar op hoge kosten jagen voor de steeds moeilijker wordende analyse. Het moment waarop software engineers ineens niet meer in staat zijn om binnen budget een systeem aan te passen, komt dan toch als een verrassing.

3. Software-analyse zelf is een grote kostenpost. Om goed te analyseren, investeren professionele software engineers relatief veel tijd (meer dan de helft) in het onderzoeken van bestaande broncode en weinig tijd in het uitbreiden ervan. Met de huidige stand van zaken in software-analyse is het voorkomen van softwarefouten niet praktisch haalbaar. Hierdoor is analyseren achteraf (testen, debuggen, versimpelen, repareren) een noodzakelijke activiteit naast het analyseren vóóraf (ontwerpen, specificeren, controleren, bewijzen) van software. Moderne ontwikkelingsprocessen voor software (*agile methods*) zijn er zelfs op gericht om achteraf-analyse zo vaak en zo vroeg mogelijk tijdens de ontwikkeling van broncode toe te passen (*continuous testing and integration*) om risico's van sluimerende problemen te kunnen vermijden. Ze zijn er ook op gericht om waarschijnlijk nutteloze vooraf-analyse (die de veranderende context niet kan voorspellen) te vermijden. Meer software-analyse automatiseren tijdens het ontwikkelproces past goed in deze filosofie door de iteratiecyclus van ontwikkelen/testen/verbeteren te versnellen.
4. Software is een innovatiemotor met enorme potentie. Om nieuwe software-diensten en -producten in de markt te zetten, moet er wel tijd over zijn naast het onderhouden van de snelgroeiende berg van bestaande software. De enige manier om ruimte te creëren voor echte innovatie is om de analyse van bestaande software efficiënter aan te pakken, zodat de onderhoudskosten worden beperkt.

**Het automatiseren van software-analyse is onontbeerlijk.** Wat is er eigenlijk moeilijk aan het begrijpen van software? Gegeven iemand die de programmeertalen kan lezen en schrijven, zou diegene niet gewoon de nodige aanpassingen moeten kunnen uitvoeren? Nee, dat kan meestal niet.

De drempel is de complexiteit van softwaresystemen. Softwaresystemen zijn meestal zo groot en zo ingewikkeld dat de moeite van het analyseren niet lijkt op te wegen tegen de risico's van het ongeïnformeerd aanpassen of uitbreiden. Bovendien is software lezen veel saaier dan software schrijven. Bij de analyse van software past eigenlijk net zo'n geduld en net zo'n bescheidenheid als bij het bestuderen van de natuur: waarschijnlijk zullen we alle software die geschreven is, en zal worden, nooit helemaal begrijpen, maar we kunnen ons best doen en we zullen daarbij allerlei mogelijke hulpmiddelen willen gebruiken.

De opkomst van effectieve softwaregereedschappen, zoals moderne flexibele programmeertalen en geïntegreerde programmeeromgevingen (IDE's), herbruikbare componenten en vrij te gebruiken voorbeeld-broncode heeft de kunst van het programmeren veranderd, maar niet uitsluitend ten goede. We kunnen nu, met ondersteuning van de computer zelf en het internet, sneller meer ingewikkelde en meer flexibele software samenstellen. Het analyseprobleem is hierdoor alleen maar groter geworden.

Toch bestaan er al veel gereedschappen die zouden kunnen helpen bij het opsommen en ontwarren van software. De laatste decennia in onderzoek van Software Metrics, Program Comprehension, Software Re-engineering, Reverse Engineering, Software Refactoring, Dynamic Analysis en Static Analysis, Software Verification en Mining Software Repositories, hebben allerlei gereedschappen voor software-analyse opgeleverd, die specifieke vragen zouden kunnen beantwoorden over software. Voorbeelden van dit soort vragen zijn: Uit welke onderdelen bestaat dit enorme systeem eigenlijk? Welke onderdelen zijn van elkaar afhankelijk en hoe? Welke onderdelen zijn slecht onderhoudbaar? Welk type aanpassingen leidt typisch tot observeerbare fouten? Waar is welke functionaliteit geïmplementeerd in de broncode? Welke onderdelen dupliceren elkaars functionaliteiten? Mijn collega's van de sectie Model Driven Software Engineering (MDSE) werken ook met succes vanuit verschillende invalshoeken aan deze gereedschappen: bijvoorbeeld via het ontwikkelen en bestuderen van abstracte modellen (zoals communicatie-protocollen) en via het bestuderen van software engineering meta-data (software repositories). Mijn eigen interesse ligt bij de broncode zelf, waardoor we samen het hele vakgebied van software-analyse goed afdekken: via broncode, via modellen en via meta-data.

Geautomatiseerd analyse-gereedschap, software voor software, heeft toegevoegde waarde omdat het beter schaalt dan handmatige analyse. Dit gereedschap kan:

- grote hoeveelheden broncode (en andere gerelateerde informatie) snel doorzoeken. Dit helpt bijvoorbeeld bij het lokaliseren van broncode die gewijzigd zou moeten worden;
- grote aantallen deductie-stappen doorrekenen (voor inferentie en verificatie). Dit helpt bijvoorbeeld bij het vinden van broncode die beïnvloed zou worden bij een geplande wijziging (als dit gewijzigd wordt, dan moet ook dit veranderd worden, enzovoorts), maar het helpt ook bij het vinden van fouten (model checking);
- beschrijvende statistieken doorrekenen (om bijvoorbeeld de verdeling van complexiteit over een groot systeem inzichtelijk te maken);

- snel diagrammen produceren die een overzicht geven (om bijvoorbeeld de onderlinge verbanden tussen sub-systemen te kunnen overzien). Hierbij laten we het oordeel over aan de mens, terwijl we de computer gebruiken als visualisatiehulpmiddel.

Er is domweg te veel broncode van een te hoge complexiteit om dit soort analyses uit het blote hoofd te kunnen doen. Zonder automatische analyse kan hoge kwaliteit software dus niet bestaan. Programmeurs gebruiken dan ook vaak automatische analyses als onderdeel van hun programmeeromgeving, soms zonder het zelf te weten. Voorbeelden hiervan zijn functionaliteiten als: *content-assist/auto-complete, jump-to-definition, rename* en *browse type-hierarchy*. Het geavanceerdere gereedschap, prototypes die uit academisch onderzoek komen, is meestal niet geïntegreerd omdat het nog te traag is of nog te veel expertise vereist bij het interpreteren van de resultaten. Er is nog heel veel onderzoek te doen aan gereedschappen voor software-analyse.

**Intermezzo: is voorkomen niet altijd beter dan genezen?** Is het niet zo dat we software gewoon op een andere manier moeten gaan maken; een manier die niet leidt tot bovenmenselijke complexiteit? Kunnen we geen software schrijven die a priori makkelijk te begrijpen en in de toekomst makkelijk te veranderen is? Dit is het gebied van Software Constructie. Het past goed in het beeld dat software taal is, dat ook de vooruitgang in Software Constructie, het zo simpel en aanpasbaar opschrijven van broncode, nu vooral gedreven wordt door de ontwikkeling van nieuwe softwaretalen. Er is niet alleen onderzoek naar nieuwe programmeertalen, waarmee de computer en netwerkapparatuur van tegenwoordig beter mee kan worden aangesproken, maar ook naar modelleertalen waarmee specifieke domeinen van een hele nieuwe taal worden voorzien.

Met Model-Driven Engineering (MDE) worden domeinspecifieke talen ontworpen waarin software engineers (of gewoon direct de domeinexperts) nieuwe software op een veel simpelere manier kunnen beschrijven en ook weer aanpassen. De onderliggende vertaling van de broncode van deze nieuwe talen naar snelle code op het machineniveau, bevat juist weer allerlei interessante software-analyse. Broncode geschreven (of getekend) in modelleertalen bevat bovendien meer mogelijkheden voor nuttige analyses, zoals correctheidsbewijzen, die op het universele broncodeniveau nog niet theoretisch of praktisch haalbaar zijn. Niet voor niets bestuderen we in de MDSE-sectie deze vorm van vooruitgang.

Figuur 3

De Atheense School – Raphaël. Plato en Aristoteles in hun eeuwige en onmogelijke discussie over de essentie van waarheid: het idee versus de observatie. Aan de ene kant kan elk Platonisch idee vormgegeven worden in software, aan de andere kant wordt de broncode van elk softwaresysteem geschreven en gelezen binnen de grenzen van onze menselijke realiteit.

Welke nieuwe talen we ook bedenken, zowel broncode geschreven in modelleertalen als broncode geschreven in moderne programmeertalen, deze talen moeten geanalyseerd worden. Ook om van bestaande software naar model gestuurde software over te kunnen stappen, is weer software-analyse nodig. Software-analyse is dus zowel een ingrediënt van methoden en technieken die problemen helpt voorkomen, als een ingrediënt bij de diagnose en behandeling van softwareproblemen.

**De definiërende kwaliteit van gereedschap voor software-analyse is de nauwkeurigheid van het gevonden antwoord**. De meeste gereedschappen moeten proberen antwoorden te produceren op vragen die in principe niet eens altijd beslisbaar zijn in theorie, laat staan uitvoerbaar met beperkte middelen. Deze gereedschappen doen daarom noodzakelijk water bij de wijn en dat kan op allerlei manieren. We kunnen de antwoorden op analysevragen benaderen:

- door soms foute antwoorden (*false positives*) toe te laten, die door de mens nog moeten worden gecontroleerd. Het voordeel is dat we niet alle broncode hoeven te lezen om een lijst van potentiële antwoorden (met een hoge kans van waarschijnlijkheid) te vinden.
- door soms goede antwoorden niet te produceren (*false negatives*), zodat je als mens nooit kan weten of er nóg ergens anders een antwoord te vinden is, maar je hebt tenminste een paar antwoorden automatisch gevonden.
- door het probleem te herdefiniëren in een zwakkere vorm ('ja/nee' vervangen door 'ja/misschien' of 'misschien/nee'). Nu is het antwoord nooit meer fout in een formele zin, maar als we 'misschien' anders interpreteren, kunnen we de eerdere twee interpretaties weer toepassen.
- door te accepteren dat er in sommige gevallen helemaal geen antwoord zal worden geproduceerd (bijvoorbeeld door eeuwig door te rekenen).
- door allerlei verwarrende combinaties van bovenstaande strategieën te implementeren. Dit lijkt misschien onwaarschijnlijk, maar de status quo van omgevingen voor programmeertalen (IDE's) is dat er veel verwarrend gereedschap bestaat. De reputatie van software-analyse in het algemeen loopt hierdoor nog wel eens een knauw op.

Ook al zijn de meeste software-analyses op een arbitraire manier benaderingen, toch kunnen we hun kwaliteit op een objectieve manier meten en met elkaar vergelijken. De theoretische resultaten moeten worden geëvalueerd met behulp van empirische methoden. We gebruiken zowel 'laboratoriumexperimenten', die het effect van de verbetering isoleren of we doen 'veldonderzoek' op basis van een corpus van bestaande softwaresystemen. De eerste methode controleert de onderliggende theorie achter de nieuwe analyse-techniek door mogelijke fouten aan het licht te brengen. De tweede controleert de relevantie van de nieuwe techniek door de effecten op software 'in het wild' in kaart te brengen.

We meten de kwaliteit van een benadering door zowel de efficiëntie (tijd- en geheugengebruik) en de nauwkeurigheid (sensitiviteit en specificiteit) te vergelijken met een orakel:

- het 'met de hand' doorrekenen op een aantal voorbeeldsystemen;
- de resultaten van een vergelijkbare (vorige generatie) automatische analyse gebruiken.

Beide methoden zijn uiteraard niet 100% waterdicht. De interpretatie van de resultaten van dit soort experimenten is bovendien van nature gedetailleerd en complex. Het is niet ongebruikelijk dat gepubliceerde resultaten worden geïnvalideerd.

Aan de ene kant zijn dit de kinderziektes van een jong vakgebied, waarin de auteurs van bijna elke nieuwe publicatie een nieuwe (kwetsbare) onderzoeksmethode introduceren, toegepast op weer eens een nieuw (kwetsbaar) corpus. Aan de andere kant veranderen omgevingsfactoren snel, zoals de computer- en netwerkapparatuur waarop het onderzoek wordt uitgevoerd, software zelf waarop de experimenten worden toegepast en de programmeeromgevingen waarin het gereedschap wordt ingevoerd. *Caveat emptor* dus: een gezonde sceptische houding bij het interpreteren van gepubliceerd onderzoek over sofware-analyse is noodzakelijk.

Deze chaotische toestand zal in de nabije toekomst moeten gaan stabiliseren, want op deze manier kunnen nieuwkomers niet verder bouwen op de al bestaande kennis. Binnen het gebied Mining Software Repositories wordt hard gewerkt aan het verzamelen, opschonen en in kaart brengen van standaard corpora van softwaresystemen. Ik ga de resultaten hiervan meer gebruiken bij de evaluatie van software-analyses. Ook werkt de gemeenschap aan het reproduceerbaar maken van onderzoeksresultaten. We publiceren steeds vaker de experimentele opzet, inclusief de voorbeeldsystemen, als artefacten naast het gepubliceerde paper.

**Intermezzo.** Bij veelbelovend gereedschap wordt uiteindelijk soms ook de daadwerkelijke bruikbaarheid onderzocht door middel van A/B-testen: programmeurs worden dan geobserveerd bij het uitvoeren van taken mét en zonder het gereedschap. Dit soort onderzoek is erg kostbaar, moeilijk om goed te doen en bovendien slecht te generaliseren: er zijn nog teveel omgevingsfactoren waar we in wetenschappelijke zin geen verstand van hebben. Ik ben er dus zelf helemaal geen fan van: zonde van het subsidiegeld.

**Als de berg niet naar Mohammed komt.** Aan de ene kant is het zonneklaar dat het de uitdaging is om een zo hoog mogelijke nauwkeurigheid te halen voor een zo laag mogelijke investering in bronnen van tijd, ruimte en energie. Aan de andere kant is het definiëren van werkbare beperkingen waarin nog nét wel een effectieve analyse gedaan kan worden ook een richting van vooruitgang. We kunnen vragen aan de software engineer om redundante informatie te verschaffen die analyse mogelijk maakt (annotaties) of we kunnen de programmeertaal zodanig beperken dat bepaalde ontraceerbare interacties niet meer mogelijk zijn (typesystemen en domeinspecifieke talen). Het vakgebied van het ontwerp van nieuwe programmeer- en modelleertalen kun je zo positioneren ten opzichte van automatische software-analyse: de balans zoeken tussen vrije expressiviteit en automatische controleerbaarheid.

Figuur 4

M.C. Escher. De toeschouwer is onderdeel van het schilderij dat hij bestudeert; een situatie die de complexe relatie illustreert tussen software-analyse-onderzoek en de software die geanalyseerd wordt.

We zien dat de groeiende kracht van computer- en netwerktechnologie steeds meer software-analyses van een steeds hogere nauwkeurigheid mogelijk maakt in een steeds kortere tijd (bijvoorbeeld met behulp van snelle solid-state drives, multicore CPU's en cloud-technologie). Een andere bron van hoop is de toepassing van steeds meer contextuele informatie bij het analyseren van broncode.

**Software is contextueel**. Misschien nog wel meer dan natuurlijke taal is software-taal contextueel: alleen met begrip van de context kan het hoe en waarom van broncode echt begrepen worden. Voorbeelden van contexten zijn de zich aldoor ontwikkelende bedrijfsvoering, de hardware-technologie die de software bestuurt, het natuurfenomeen dat met behulp van de software bestudeerd wordt, de mensen die de software gebruiken, de tijd waarin de software geschreven werd en de 'best practices' van toen, de achterliggende filosofie van de gebruikte programmeerparadigma's en het ontwerp van de gebruikte programmeertalen, de versie van hergebruikte software, enzovoorts.

Daarom mag je nooit aannemen of verwachten dat een software engineer met alleen algemene softwarekennis en -vaardigheden betrouwbaar kan analyseren wat willekeurige software precies doet. Het hoe en waarom, voor een specifiek vakgebied waarvoor software werd geschreven, is essentieel ontbrekende context-informatie. Bijvoorbeeld: een fout in het onderliggende oceaanmodel, dat met kennis en kunde van oceanografie en klimaatmodellering is samengesteld, en de software die de minimale dijkhoogtes voor de volgende decennia berekent, maakt een verkeerde voorspelling. Ook al is de broncode een absolute definitie van wat een computer zal gaan doen, de kennis over de intentie van de ontwerpers en de achterliggende redenatie bij de functionaliteit is impliciet.

De contextafhankelijkheid van software is de reden dat professionele programmeurs hun broncode voorzien van commentaar om intentie van de code uit te leggen. Er bestaat ook een onderzoeksgebied Requirements Engineering dat zich ten doel stelt om software-ontwerpbeslissingen traceerbaar te maken naar het ontwerp van het pakket van eisen (*requirements*). Ook Software Specificatie en Software Testen zijn manieren om meer redundante informatie toe te voegen die de aannames, waaronder software wél goed werkt, expliciteert. Aan de ene kant zijn commentaar in broncode, traceerbare verbanden, specificaties en tests bijzonder nuttig als extra informatie, maar aan de andere kant vergroot al deze informatie alleen maar het volume en de complexiteit van de informatie waarmee een software engineer in aanraking komt. Het voordeel is dat al deze secundaire informatie over software, naast de primaire broncode, ook een weer goede gegevensbron is voor automatische analyses.

# Domeinspecifieke software-analyse

Het logische gevolg van contextafhankelijkheid van software is dat de studie van software niet vooruit kan zonder intense samenwerking met andere vakgebieden, zoals bijvoorbeeld wiskunde, elektrotechniek, bouwkunde, financiën en wetgeving. Het effectief toepassen van domeinkennis uit het eigen vakgebied (programmeertalen, frameworks en libraries), maar ook kennis en vuistregels uit de vakgebieden waar software in wordt toegepast, is noodzakelijk. Zoals modelgestuurde softwareconstructie domeinkennis a priori integreert, zo zou modelgestuurde analyse ook op grotere schaal domeinkennis moeten gaan inzetten bij het ontrafelen van bestaande softwaresystemen.

**Het toepassen van contextinformatie binnen het vakgebied software engineering is bezig aan een opmars**. Het modelleren van de semantiek van (bijvoorbeeld) software frameworks naast het modelleren van de semantiek van programmeertalen, maakt het mogelijk om veel preciezere antwoorden te geven bij statische analyses. Een voorbeeld is het koppelen van de analyseresultaten van *server-software* die in Java geschreven is met de analyseresultaten van de *cliënt software* die in Javascript geschreven is. Beide analyses kunnen wat nauwkeurigheid betreft voordeel hebben bij de toegevoegde contextinformatie. Alleen binnen het domein van het bouwen van websites heeft deze contextinformatie zin, maar zeker binnen dit domein kan het een enorme impact hebben op de analyse van complexe softwaresystemen. Neem een analyse van de veiligheid van de software van een bank. Deze is per definitie zo veilig als de zwakste schakel in de keten. Maar als de keten niet verbonden is, kan er ook geen sprake zijn van een volledige analyse. Juist met het combineren van de *cliënt* en de *server software*, kan wél een inzicht in de volledige keten worden verkregen. Ook alleen door de *cliënt* en *server software* te analyseren in de context van een mogelijk derde (malafide) communicatiepartner, kan de veiligheid van het systeem geanalyseerd worden.

Gezien de enorme variatie van talen en technologieën binnen het vakgebied is er nog heel veel potentie voor het koppelen van analyses en het integreren van informatie uit verschillende bronnen.

Figuur 5

Vik Muniz. Net zoals broncode is deze jam- en pindakaaskunst niet goed te interpreteren zonder kennis van de context: wie is die dame?

**De integratie van contextinformatie in software-analyses is nog grotendeels onontgonnen gebied**. Uitzonderingen daargelaten, is de meeste literatuur en bijbehorende technologie gericht op een zo groot mogelijke impact door zo algemeen mogelijke analyses in de markt te zetten. Ik stel dat we in de nabije toekomst aan het einde van dit lange touw zijn gekomen en er nog maar één manier voorwaarts zal zijn: contextgevoelige analyse met behulp van grootschalige integratie van allerlei externe informatiebronnen. Voorbeelden van externe bronnen zijn configuratiescripts van build- en testomgevingen, wetteksten, ontologieën, commentaar in broncode, *glue code* van dynamische programmeertalen, wiskundige modellen gespecificeerd in Matlab of Mathematica, handleidingen en ga zo maar door.

**De grootste uitdaging voor de toekomst van het automatiseren van software-analyse is het effectief betrekken van zoveel mogelijk contextinformatie bij de analyses.** Het probleem is schaalbaarheid. Hoe kunnen we analyse-gereedschap goedkoper specialiseren voor de context waarin het zal worden toegepast en gebruikmakend van de informatie waar deze context in voorziet? Het construeren van nauwkeurig gereedschap voor software-analyse is ingewikkeld, zeer gedetailleerd en lastig toetsbaar werk. Hoe kunnen we verwachten dat voor allerlei

mogelijke contexten er correct analysegereedschap geproduceerd kan worden? En wie gaat dat doen?
Dat brengt ons eindelijk aan het onderwerp van mijn eigen specialisme: metagereedschap uitvinden, evalueren en toepassen. Een metagereedschap is gereedschap om gereedschap mee te maken. Een metaprogrammeertaal is een programmeertaal om programmeertalen mee te analyseren. Om nieuw specialistisch analysegereedschap te kunnen maken, moet bestaand gereedschap makkelijk uitbreidbaar en aanpasbaar zijn. Om al deze aanpassingen te kunnen doen, moet metaprogrammeergereedschap te gebruiken zijn door een grote groep software engineers.

**Rascal[1] is een experimentele metaprogrammeertaal met het doel om snel nieuwe software gereedschappen te ontwerpen en toe te passen.** Het uiteindelijke doel is om software beter te leren begrijpen. Hiervoor zijn nieuwe gereedschappen nodig, die nauwkeurig en snel analyses kunnen uitvoeren met behulp van allerlei aanwezige contextinformatie. Om deze gereedschappen te bouwen, ga ik verder onderzoek doen in het ontwerp en de implementatie van Rascal. Tot slot kan empirisch onderzoek met deze gereedschappen tot stand komen en directe valorisatie bij het beantwoorden van software vraagstukken die relevant zijn voor regio's Eindhoven en Amsterdam (de high-techsector en de financiële/administratieve sector).

### Onderzoeksvragen over metaprogrammeren

Wanneer we een antwoord hebben op de volgende metaprogrammeervraagstukken, dan kunnen we veel efficiënter en effectiever nieuwe soorten softwareanalyses gaan bouwen:

1. Data acquisitie: Hoe kan externe informatie over broncode snel gemodelleerd en geïntegreerd worden in een grotere context van broncode-analyse? Hoe hergebruiken we effectief externe meta-definities van deze kennis?
2. Variabiliteit: Hoe kunnen we veel meer hergebruik van (complexe) analyses tussen programmeertalen en versies van programmeertalen bewerkstelligen? Welke tussenliggende algemene representaties lenen zich hiervoor? Hoe modulariseren we analyses?
3. Schaalbaarheid: Hoe kunnen we grotere hoeveelheden van gestructureerde informatie over broncode (relaties en hiërarchieën) in een keer analyseren zodat kruisverbanden tussen verschillende onderdelen kunnen worden onderzocht? Hoe implementeren we *on-demand* toegang tot deze semantisch rijke representaties van broncode als enorme database?

[1] http://www.rascal-mpl.org

### Onderzoeksvragen over software-analyse

De volgende generatie software-analyses zal sneller en meer informatie moeten gaan verwerken om nauwkeuriger antwoord te kunnen geven op vragen van zowel gebruikers als software engineers:

1. Code-to-model: Hoe extraheren we nuttige domeinkennis (modellen) op hoog abstractieniveau van de broncode op laag abstractieniveau?
2. Context: Hoe integreren we meta-informatie en configuratie-informatie over softwaresystemen in de statische analyse van broncode?
3. Kwaliteit: Hoe beoordelen we zo efficiënt mogelijk, (semi-automatisch) belangrijke kwaliteitseigenschappen zoals informatieveiligheid, beschikbaarheid en onderhoudbaarheid van softwaresystemen?
4. Renovatie: Hoe kunnen we voortdurende verandering en verbetering (*refactoring*) van softwaresystemen ondersteunen met semantisch diepe analyses van de broncode met contextinformatie?

### Empirische onderzoeksvragen over gereedschap voor software-analyse

De empirische softwarevraagstukken die ik interessant vind, concentreren zich op het evalueren van softwaregereedschap en de validiteit van evaluaties:

1. Wat is de nauwkeurigheid van nieuw en bestaand gereedschap en wat kan de (positieve of negatieve) impact zijn van het gebruik van dit gereedschap op de broncode en het ontwikkelingsproces?
2. Wat is 'normale' broncode? Wat mogen we verwachten van de impact van nieuw gereedschap op de kwaliteit van bestaande broncode en het ontwikkelingsproces ten opzichte van de ruis van andere factoren?
3. Wat is de validiteit van meta-informatie (bijvoorbeeld uit versiebeheersystemen en issue-trackers) die gebruikt wordt bij het evalueren van de effectiviteit van gereedschap voor software-analyse?

### Valorisatievraagstukken

Omdat software-analyse contextafhankelijk is, is er bij het onderzoek praktisch altijd directe samenwerking met een industriële partner of een andere domeinexpert noodzakelijk. Deze samenwerking borgt dan meestal ook directe initiële valorisatie. Maar om de resultaten van onderzoek naar gereedschap voor software-analyse ook over te brengen aan derde partijen, zijn er antwoorden nodig op de volgende vragen:

1. Welke uitdagingen met betrekking tot software-analyse hebben de hoogste prioriteit?
2. Hoe creëren we meer eigenaarschap van maatwerkgereedschap voor software-analyse? Wat zijn noodzakelijke eigenschappen van het meta-gereedschap om dit te bewerkstelligen? Welke opleidingsmiddelen zetten we hierbij in?
3. Hoe benaderen we de implementatie van nieuw softwaregereedschap in complexe bedrijfscontexten?

```
%! PS EPS
3{25 34 moveto
25 -34 lineto
17 -38.2 lineto
17 20 lineto
-17.6 0 lineto
120 rotate
}repeat stroke showpage
```

Figuur 6

Escher's onmogelijke driehoek geschreven door Kees van der Laan in de Postscript programmeertaal. Valorisatie is een noodzakelijke en gewenste uitkomst van wetenschappelijk onderzoek, maar de drie belanghebbende partijen van valorisatie – politiek, wetenschap en industrie – komen net als Escher's driehoek niet als vanzelf bij elkaar. Dat kost niet alleen persoonlijke inspanning en creativiteit, maar ook van alle kanten extra investeringen in tijd en geld.

# Conclusie

Software is interessant, niet alleen vanuit een intellectueel perspectief, maar vooral ook vanuit een maatschappelijk en economisch perspectief. Het automatiseren van software-analyse is een noodzakelijke richting van onderzoek, om in de toekomst vat te leren krijgen op het enorme volume en de complexiteit van de software van vandaag en morgen. Er is een aantal duidelijke stappen te zetten in deze richting en ik ben van plan om de nauwe samenwerking tussen de TU/e en het CWI hiervoor te gebruiken. De bestaande goede samenwerking tussen de onderzoekers en de industrie in de regio's Amsterdam en Eindhoven verder uitbouwen, is een belangrijke voorwaarde tot succes.

# Dankwoord

Het is uiteraard een persoonlijke eer om een inaugurele rede te mogen schrijven en uitspreken, maar de eer is niet geheel aan mij. Ik heb veel te danken aan mijn familie, mijn leraren, mijn vrienden en collega's.

Rebecca Vinju-Maclean en Simon en David, jullie zijn het belangrijkst, dus jullie komen eerst. Het is mijn redding dat het met jullie thuis nóg fijner is dan op mijn werk. Ik houd veel meer van jullie, dan ik van software houd en nóg veel meer dan naar de maan en terug en dan nog een rondje.

Annelies Vinju-Zuurveld, bedankt voor mijn hele leven en voor je oneindige support en onze lange leerzame avonturen samen achter het computerscherm. Fred Vinju, bedankt voor alles wat ik heb kunnen bereiken dankzij jou: onze computers thuis die ik mocht annexeren en het lef om alles zelf te gaan repareren dat ik van jou heb geleerd (het kapot maken heb ik van mezelf). Krista de Haas, bedankt voor het nogal behoorlijk hetzelfde zijn als ik, zodat ik niet steeds hoef te denken dat ik gek geworden ben. Ook al zien we elkaar veel te weinig: Bas Toeter, Winfried Holthuizen, Warner Salomons, Hugo Loomans, Mieke Loomans, Allan Jansen, Arjen Koppen, Rob Economopoulos, Tijs van der Storm en Rik Jonker: bedankt voor jullie vriendschap.

Paul Klint, bedankt voor mijn hele professionele carrière, van studiekeuze tot en met wetenschapsmanager en deeltijdhoogleraar en alles daartussenin. Het staat er zo even kort, maar het is voor een heel groot deel jouw coaching die me hier heeft gebracht. Ik ben naast je adviezen ook heel blij met al die ruimte die je hebt gelaten voor het maken van mijn keuzen en daarmee mijn fouten om van te leren. Het is ook een groot plezier om bijna dagelijks met je samen te werken aan Rascal.

Mark van den Brand, bedankt voor het begeleiden van mijn masterscriptie, PhD-onderzoek en de benoeming tot deeltijdhoogleraar bij de TU/e. Jij hebt het me voorgedaan: in teamverband onderzoek doen naar taaltechnologie door nieuwe gereedschappen te ontwerpen, te bouwen, te evalueren, toe te passen en er dan over te schrijven. Nu wil ik alleen nog maar dat. Ook jij hebt me losgelaten en dat is een reden dat ik nu weer dicht bij kon komen.

Ik ben in het bijzonder geïnspireerd door het bestuderen van de werken van en/of het voeren van gesprekken met deze collega's: Gerald Stap, Jan Heering, Tijs van der Storm, Mark Hills, Hans Dekkers, Jim Cordy, Ralf Lämmel, Alexander Serebrenik, Tudor Gîrba, Anthony Cleve, Oscar Nierstrasz, Mike Godfrey, Frank Tip, Stéphane Ducasse, Friedrich Steimann, Peter Mosses, Terence Par, Magiel Bruntink, Andreas Zeller, Yannis Smaragdakis en Robert M. Fuhrer.
Verder bedank ik op min of meer chronologische volgorde van invloed deze collega's: Chris Verhoef, Hayco de Jong, Pieter Olivier, Jeroen Scheerder, Merijn de Jonge, Leon Moonen, Eelco Visser, Arie van Deursen, Tobias Kuipers, Peter van Emde Boas, Niels Veerman, Steven Klusener, Jørgen Iversen, Slinger Jansen, Taeke Kooiker, Pierre-Étienne Moreau, Claude Kirchner, Rob Economopoulos, Jan van Eijck, Robert van Liere, Bert Lisser, Atze van der Ploeg, Paul Griffioen, Stijn de Gouw, Martin Bravenboer, Vadim Zaytsev, Ashim Shahi, Anya Helene Bagge, Elizabeth Scott, Adrian Johnstone, Dimitris Kolovos, Davide DiRuscio, Anamaria Moreira, Cleverton Heinz, Jan Friso Groote, Yanja Dajsuren, Joost Bosman senior, en Joost Bosman junior. Het met jullie eens of oneens zijn, het met jullie samenwerken, het bestuderen van jullie werk, of gewoon samen wild filosoferen over de toekomst, het is allemaal woest interessant.

Ik dank graag naast CWI als geheel, ook een aantal specifieke collega's waar ik vaker mee in contact ben: Jos Baeten, Han La Poutré, Frank de Boer, Rob van der Mei, Ronald de Wolf, Dick Broekhuis, Karin Blankers, Bikkie Aldeias, Susanne van Dam, Margriet Brouwers, Karin van Gemert, Irma van Lunenburg, Marlin van der Heijden, Léon Ouwerkerk, Niels Nes, Wouter Mettrop, Rob van Rooijen en Angelique Schilders. Bedankt voor de geweldige werkomgeving.

Ik dank Bas Basten voor het eerste proefschrift waar ik co-promotor van mocht zijn en voor de fijne samenwerking aan een geweldig onderwerp.

Tot slot, maar zeker niet in het allerminst, dank ik op alfabetische volgorde de promovendi die nu met mij werken aan hun eigen grote project: Ali Afroozeh, Anastasia Izmaylova, Davy Landman, Jouke Stoel en Michael Steindorfer. Ik leer minstens net zoveel van jullie, als jullie van mij. Ook hartelijk dank aan de andere promovendi uit de CWI SWAT groep voor jullie tomeloze energie: Gauthier van den Hove, Riemer van Rozen, en Pablo Inostroza Valdera.

# Curriculum vitae

**Prof.dr. Jurgen J. Vinju is per 1 september 2014 aangesteld als deeltijdhoogleraar Automated Software Analysis aan de Faculteit Wiskunde en Informatica van de Technische Universiteit Eindhoven.**

Jurgen Vinju (1977) promoveerde in 2005 aan de Universiteit van Amsterdam met het proefschrift 'Analysis and Transformation of Source Code by Parsing and Rewriting'. Hij heeft als gast-onderzoeker ervaring opgedaan bij Alcatel-Lucent Bell Labs (2006) en IBM TJ Watson Research (2007-2008). Hij werkte part-time bij de Master Software Engineering UvA/VU/HVA (2005-2014). Sinds 2012 is hij groepsleider van de onderzoeksgroep SoftWare Analysis and Transformation (SWAT) van het Centrum Wiskunde & Informatica.

**Colofon**

**Productie**
Communicatie Expertise
Centrum TU/e

**Fotografie cover**
Rob Stork, Eindhoven

**Ontwerp**
Grefo Prepress,
Eindhoven

**Druk**
Drukkerij Snep, Eindhoven

**ISBN 978-90-386-4031-0**
**NUR 989**

Digitale versie:
www.tue.nl/bib/