Consistentie tussen model en code

Intreerede
prof.dr. Mark van den Brand
3 juli 2008

/ Faculteit Wiskunde en Informatica

TU/e Technische Universiteit **Eindhoven** University of Technology

Consistentie tussen model en code

Where innovation starts

Intreerede prof.dr. Mark van den Brand

Consistentie tussen model en code

Uitgesproken op 3 juli 2008
aan de Technische Universiteit Eindhoven

De effecten van software

We kunnen ons al vrijwel geen wereld zonder computers en de daarbij behorende software voorstellen. In zestig jaar tijd is onze samenleving veranderd in een samenleving die volledig afhankelijk is geworden van de informatietechnologie. De eerste toepassingen van software waren in de numerieke wiskunde. De eerste commerciële toepassingen van software waren grote administratieve systemen, bijvoorbeeld bij banken en verzekeringsmaatschappijen. Software is meer en meer gemeengoed geworden dat zich in allerlei gedaantes manifesteert: in mobiele telefoons, navigatiesystemen, televisies, spellen, internetbankieren, belastingaangiften en dergelijke. Software bestuurt niet langer alleen de administratieve processen in een organisatie, maar ook de complexe logistieke processen en productieprocessen. Een sprekend voorbeeld is de rol van software in de gezondheidszorg. Software zorgt er momenteel voor dat resultaten van röntgenopnames, MRI-scans en dergelijke direct bij de specialist op zijn beeldscherm verschijnen, maar ook dat MRI-scans en CT-scans überhaupt mogelijk zijn. De verinformatisering van onze samenleving heeft grote sociale en culturele consequenties; positieve zoals de ontwikkeling van e-mail, het World Wide Web, chatten, maar ook negatieve zoals spam, phishing, RSI, et cetera.

In deze rede wil ik stilstaan bij de consequenties van deze verinformatisering voor het gebied van de software engineering zélf. De enorme vlucht die software heeft genomen, heeft ook consequenties voor de software zelf en zeer zeker ook voor de manier waarop deze software ontwikkeld wordt en in de toekomst ontwikkeld zal gaan worden. Hierbij moet ook gekeken worden naar de consequenties die deze ontwikkelingen hebben voor de informaticacurricula. De software-industrie heeft in de laatste zestig jaar ook een turbulente ontwikkeling doorgemaakt. Een zeer sprekende en goed leesbare beschrijving hiervan is onder meer te vinden in Campbell-Kelly (2003). Bedrijven als ASML, NXP, Philips en Océ zijn bezig met de ontwikkeling van betrouwbare (embedded) software; bedrijven als DAF Trucks en Vanderlande Industries staan aan de vooravond van ‘de software-uitdaging’. De laatst genoemde bedrijven komen voort uit de werktuigbouw. Zij zien dat steeds meer traditionele mechanische processen worden aangestuurd door software. Eerst gebruikten deze bedrijven alleen elektronica voor de besturing (hard-wired) en later (primitieve) software op PLC’s. De vergaande integratie van software in

onze samenleving stelt hoge eisen aan de betrouwbaarheid van deze software en aan de snelheid waarmee wij deze software kunnen ontwikkelen en onderhouden. Als oplossing voor de ontwikkelsnelheid en het terugdringen van de ontwikkelingskosten maken steeds meer bedrijven gebruik van offshoring, maar is dat wel de oplossing voor de lange termijn? De effecten van langdurig onderhoud aan softwaresystemen zijn beschreven in de intreerede van professor Arie van Deursen (Van Deursen, 2005).

Op 14 december 2004 heb ik als lector Softwarekwaliteit aan de Hogeschool van Amsterdam een openbare les uitgesproken met als titel *Softwarekwaliteit - Hypes versus onderzoek in de software engineering* (Van den Brand, 2004). Nog geen vier jaar later mag ik een intreerede uitspreken als hoogleraar Software Engineering and Technology aan de Technische Universiteit Eindhoven. Ook in deze rede zal softwarekwaliteit een belangrijke plaats innemen, maar de focus verschuift naar productiviteitsverhoging, zonder het kwaliteitsaspect uit het oog te verliezen. Ik wil stilstaan bij de wijze waarop betrouwbare software op een efficiënte manier geproduceerd kan worden, zonder de toevlucht te nemen tot offshoring van de softwareontwikkeling. De belangrijkste motivatie momenteel voor offshoring is goedkope arbeid om de software te produceren, echter het opstellen van de gebruikerseisen en het ontwerp zijn moeilijker te offshoren. Het gebruik van (formele) modellen en softwaregeneratoren zou een alternatief voor offshoring kunnen zijn.

Generieke Taaltechnologie

De informatica dankt haar bestaan voor een groot deel aan ontwikkelingen en onderzoek op het gebied van compilers die worden gebruikt om computerprogramma's te vertalen in executeerbare code. Door het gebruik van compilers zijn we in staat onze software op een hoger abstractieniveau te schrijven. In de jaren zestig en zeventig van de vorige eeuw is er veel onderzoek gedaan op het gebied van de vertalerbouw. Het genereren van compilers uit taalbeschrijvingen is een van de onderzoeksgebieden van de vertalerbouw. Het schrijven van een compiler is namelijk een aanzienlijke hoeveelheid werk. Compilers werden en worden vaak 'met de hand' geschreven. Daarbij zijn grote delen van een compiler in vrijwel iedere taal hetzelfde. Het definiëren van specificatietalen voor het beschrijven van (programmeer)talen en het ontwikkelen van gereedschappen om vanuit deze beschrijvingen onderdelen van een compiler te genereren is de focus van de generieke taaltechnologie.

De belangrijkste mijlpalen op dit gebied zijn Lex+Yacc[1] en attribuutgrammatica's (Alblas en Melichar, 1991). Lex+Yacc, of een van de varianten, zijn gereedschappen met bijbehorende formalismen voor het beschrijven van de lexicale en contextvrije syntax van een (programmeer)taal waaruit een scanner en een parser gegenereerd kunnen worden. Attribuutgrammatica's zijn een verzameling van formalismen voor het beschrijven van de semantiek van (programmeer)talen. Er bestaan diverse implementaties van attribuutgrammatica's waarbij de uitwisseling van de specificaties niet altijd mogelijk is. Deze diversiteit heeft ertoe geleid dat attribuutgrammatica's niet zo populair zijn geworden en geen gangbare compilertechnologie zijn geworden.

In de jaren tachtig van de vorige eeuw verschoof de toepassing van generieke taaltechnologie van compilers naar programmeeromgevingen. Uitgaande van taalbeschrijvingen werden niet alleen onderdelen van een compiler gegenereerd, maar complete interactieve programmeeromgevingen. Het ontwikkelen van programmatuur binnen deze interactieve omgeving werd ondersteund door syntaxgestuurde editors, interpreters, prettyprinters en debuggers. Al deze gereedschappen werden via een gemeenschappelijke gebruikersinterface ontsloten. Het doel van deze omgevingen was gebruikers in een vroeg stadium te attenderen op syntactische

[1] http://dinosaur.compilertools.net/

en semantische fouten in de ontwikkelde programmatuur en daarmee de kwaliteit van het eindproduct te verhogen. Diverse onderzoeksgroepen hebben gewerkt aan de ontwikkeling van programmeeromgevinggeneratoren. De meest bekende en succesvolle zijn de Synthesizer Generator (Teitelbaum en Reps, 1989) en de ASF+SDF Meta-Environment (Klint, 1993).

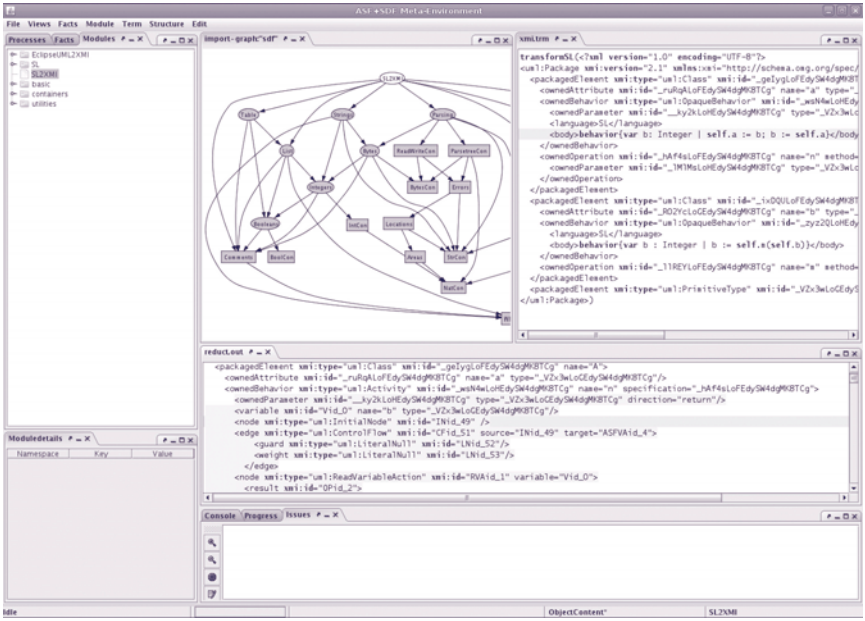
Het werk van Teitelbaum en Reps was mijn eerste kennismaking met programmeeromgevinggeneratoren. Ik was zo gefascineerd door deze technologie, dat ik meteen een onderwerp had voor mijn afstudeerproject bij professor Koster in Nijmegen. Daarna heb ik van 1987 tot 1992 onderzoek gedaan op het gebied van het genereren van programmeeromgevingen op basis van Extended Affix Grammatica's (Van den Brand, 1992). Van 1992 tot 2006 heb ik in de groep van Paul Klint (UvA/CWI) mogen werken en bijgedragen aan de verdere ontwikkeling van ASF+SDF[2] (Van Deursen et al, 1996) en de ASF+SDF Meta-Environment. ASF+SDF is een specificatietaal voor het beschrijven van de syntax en semantiek van (programmeer)talen, de ASF+SDF Meta-Environment is een interactieve programmeeromgeving die het schrijven van taalspecificaties in deze taal ondersteunt. De Meta-Environment biedt verder de mogelijkheid om gereedschappen, parsers, interpreters, prettyprinters, et cetera, te genereren op basis van deze taalbeschrijvingen. ASF+SDF is ontwikkeld om bestaande programmeertalen te beschrijven en daaruit programmeeromgevingen te genereren; de talen Pascal en Lotos zijn in ASF+SDF gespecificeerd. De focus verschoof van het beschrijven van programmeertalen naar het maken van prototypen van domeinspecifieke talen en richting reverse engineering.
De specificatie van de syntax van de domeinspecifieke taal Risla, een taal om financiële producten mee te specificeren en het specificeren van een prototype om Risla naar Cobol te vertalen waren voor mij het startschot voor de ontwikkeling van generieke prettyprintingtechnologie voor ASF+SDF (Van den Brand en Visser, 1996). Cobol is lay-outgevoelig en het was dus noodzakelijk om de gegenereerde Cobolcode te formatteren om deze geaccepteerd te krijgen door een Cobolcompiler. ASF+SDF bleek uitermate geschikt voor het ontwikkelen van prototypes voor domeinspecifieke talen.
Het toepassen van ASF+SDF voor de reverse engineering van Cobolcode werd geïnitieerd door het beroemde jaar-2000-probleem. De modulariteit van SDF stelde ons in staat om in korte tijd een vrijwel volledige grammatica voor Cobol te ontwikkelen waarbij ook stukken SQL, CICS, et cetera geanalyseerd konden worden. Deze SDF-definities werden daarnaast gebruikt om een ASF+SDF-specificatie te

[2] Algebraic Specification Formalism plus Syntax Definition Formalism

genereren waarmee transformaties op de Cobolcode konden worden uitgevoerd. De omvang van de Cobolgrammatica en de omvang van de te ontleden en te transformeren Cobolcode gaf aanleiding tot een volledige herimplementatie van de ASF-compiler (Van den Brand et al, 2002). De aanpak van het genereren van ASF-vergelijkingen voor het transformeren van Cobolcode leidde tot het ontwikkelen van de traversalfuncties in ASF+SDF (Van den Brand et al, 2003).
Bovenstaande activiteiten vormden het startschot voor een volledig herontwerp en een volledige herimplementatie van de ASF+SDF Meta-Environment (Van den Brand et al, 2001). Dit herontwerp- en implementatieproject, waaraan overigens een groot aantal mensen een bijdrage heeft geleverd, heeft geresulteerd in een aantal fundamentele softwaregereedschappen, onder andere ToolBus (Bergstra en Klint, 1998) en ATermen (Van den Brand et al, 2000). Verder heeft de ontwikkeling van de Meta-Environment bij mij geleid tot inzichten betreffende voor- en nadelen van academische softwareontwikkeling. De Meta-Environment is een bron van publicaties, welgeteld zijn er meer dan twintig proefschriften uit het onderzoek rond ASF+SDF voortgekomen en een veelvoud daarvan aan wetenschappelijke publicaties. De nadelen van een project als de Meta-Environment zijn niet zo zichtbaar. Softwareontwikkeling staat op gespannen voet met het schrijven van wetenschappelijke publicaties, al is dat misschien gezien het aantal proefschriften en publicaties rond ASF+SDF minder evident. De toename van het aantal gebruikers van ASF+SDF en de Meta-Environment vertaalde zich onmiddellijk in een toename van het aantal bug reports en feature requests. Dit heeft uiteindelijk geresulteerd in vergaande modulaire softwareontwikkeling (veel kleine softwarepakketten met specifieke functionaliteit), een strikte configuratie- en releasemanagement en het toepassen van geavanceerde softwaregeneratietechnieken, zoals ApiGen (De Jong en Olivier, 2004; Van den Brand et al, 2003). De vraag blijft of een onderzoeksgroep, ondanks de inzet van bovengenoemde technieken, het zich kan permitteren om softwaresystemen zoals de ASF+SDF Meta-Environment te ontwikkelen en te onderhouden. Mijn mening is dat dit kan, zolang het mogelijk blijft er over te publiceren.

De ontwikkeling van ApiGen is voor mij aanleiding geweest om me verder te verdiepen in softwaregeneratie. Het doel van ApiGen was om software die gebruik maakt van de ATerm-bibliotheek sneller te kunnen ontwikkelen en te zorgen dat deze software minder fouten bevat. Termen spelen een belangrijke rol in de Meta-Environment. ATermen zijn uitermate geschikt om termen efficiënt op te slaan en te manipuleren. Iedere term is uniek (maximale term sharing) en niet gebruikte termen worden automatisch opgeruimd. Het manipuleren van termen via functies uit de ATerm-bibliotheek is foutgevoelig, omdat er eigenlijk maar één soort term

bestaat. De ATerm-bibliotheek beschikt dus niet over functionaliteit om termen die verschillende types representeren van elkaar te onderscheiden. ApiGen genereert een extra laag boven op de ATerm-bibliotheek waarmee we wel in staat zijn om onderscheid tussen verschillende termtypes te maken. Hierdoor kunnen veel programmeerfouten voorkomen worden zonder de voordelen van de ATerm-bibliotheek te verliezen. De input voor ApiGen is een beschrijving van de structuur van de bomen en de typering van de knopen. Deze beschrijving is eveneens gebaseerd op SDF.



Model-Driven (Software) Engineering

Modellen nemen een centrale plaats in in de moderne softwareontwikkeling. De manier waarop software wordt ontwikkeld heeft in de afgelopen zestig jaar een groot aantal veranderingen ondergaan. De uitdaging door de jaren heen is het verhogen van het abstractieniveau waarop software ontwikkeld wordt. In de generieke taaltechnologie is dat gebeurd door het introduceren van specificatieformalismen om (programmeer)talen te beschrijven. In de beginjaren van de informatica werd software ontwikkeld in machinecode. Momenteel wordt veel software ontwikkeld op basis van domeinspecifieke talen (Van Deursen en Klint, 1998) of op basis van Unified Modeling Language (UML)[3] (Fowler, 2004). De ontwikkeling van hoog-niveau programmeertalen in de afgelopen zestig jaar heeft verschillende effecten gehad: het ontwikkelen van software werd eenvoudiger, de productiviteit van de programmeur ging omhoog en de kwaliteit van de software werd beter. De omvang en complexiteit van de software groeide echter harder dan de uitdrukkingskracht van de programmeertalen, waardoor de software toch in omvang toenam.

Naast de evolutie van programmeertalen is ook het vakgebied software engineering ontstaan. De complexiteit van het ontwikkelen van software gaf aanleiding tot het toepassen van ingenieursprincipes uit andere vakgebieden, zoals (werktuig)bouwkunde en elektrotechniek. Hierdoor heeft het vakgebied van software engineering zich in de loop van de jaren ontwikkeld: van het watervalmodel tot extreme programming. Het watervalmodel is een software life-cycle model, beschreven door Royce (Royce, 1970), waarbij de ontwikkeling de volgende sequentiële fases moet doorlopen: requirementsanalyse, ontwerp, bouw en testen (validatie) integratie en onderhoud (maintenance). Tegenwoordig is bijna iedere informaticus het wel eens met de stelling dat dit model niet meer voldoet en dat iteratieve en/of incrementele modellen betere resultaten leveren en een grotere kans tot succesvolle afronding van projecten.
Een van de belangrijkste elementen van software engineering is softwarekwaliteit. Softwarekwaliteit omvat meer dan programmacorrectheid, het is ook functionaliteit, betrouwbaarheid, bruikbaarheid, efficiëntie, onderhoudbaarheid en

[3] http://www.uml.org/

portabiliteit (Heemstra et al, 2001; Glass, 1992). Deze aspecten richten zich op het product en niet op het proces waardoor het product tot stand is gekomen.
Het jaar-2000-probleem en de introductie van de euro hebben het vakgebied reverse engineering een enorme stimulans gegeven. Een van de doelen van reverse engineering is het onderliggende model van de broncode te achterhalen. Oorspronkelijk is de broncode ontwikkeld op basis van een verzameling gebruikerseisen, een functioneel en een technisch ontwerp, vastgelegd in de zogenaamde documentatie. De ervaring leert echter dat deze documenten niet worden bijgewerkt als er onderhoud plaatsvindt, daardoor loopt de documentatie niet meer synchroon met de broncode. De mate waarin het model gereconstrueerd kan worden is afhankelijk van de programmeertaal. Voor Cobol is het bijvoorbeeld moeilijk om gedetailleerde informatie te achterhalen; vaak kan de call-graaf en modulestructuur geëxtraheerd worden. Voor moderne talen als Java, C# en C++ is het achterhalen van het klassemodel bijna triviaal, terwijl met enige inspanning zelfs het achterliggende dynamische model, bijvoorbeeld een state-machine- of sequencediagram, geëxtraheerd kan worden. De geëxtraheerde modellen kunnen vervolgens verbeterd worden en dan weer als invoer dienen om code te genereren.

Het maken van modellen voor software is tegenwoordig bijna synoniem met het maken van UML-modellen. UML bestaat uit een verzameling van diagrammen waarmee software gespecificeerd kan worden. UML is ontstaan door verschillende modelleringstalen met elkaar te verenigen, onder meer de Object Modeling Technique van (Rumbaugh et al, 1991), de Booch-methode (Booch, 1993) en use-cases van Jacobson (Jacobson et al, 1992). Later zijn daar nog andere modelleringstalen aan toegevoegd. De huidige versie van UML, UML2.0, bevat dertien standaarddiagramtypen[4]. De software industrie heeft dit UML-initiatief volledig omarmd en UML is een standaard geworden voor het modelleren van software. Maar is UML wel de meest geschikte modelleringstaal? Is UML niet veel te rijk en divers?

De ervaring leert dat de meest gebruikte diagrammen de klassendiagrammen, sequencediagrammen, use-casediagrammen, activitydiagrammen en statemachinediagrammen zijn, eventueel aangevuld met specificaties in OCL (Object Constraint Language) of Action Semantics. Is UML het eindpunt of is het juist een startpunt wat betreft de ontwikkeling van specificatietalen? In de jaren zestig van de vorige eeuw zijn de programmeertalen Fortran en Cobol ontworpen met het idee dat er niet meer talen nodig zouden zijn. Wie had toen kunnen voorspellen dat er vijftig jaar later honderden verschillende programmeertalen zouden zijn?

[4] http://www.omg.org/gettingstarted/what_is_uml.htm

De syntax van een programmeertaal wordt beschreven met behulp van een grammatica, bijvoorbeeld (E)BNF[5] of SDF. (E)BNF en SDF kunnen zelf ook weer beschreven worden in (E)BNF respectievelijk SDF. Deze mooie hiërarchie bestaat echter alleen voor de syntax en niet voor de semantiek van programmeertalen. Een vergelijkbare situatie bestaat voor modelleringstalen zoals UML. De structuur van UML-diagrammen kan beschreven worden op een meta-modelniveau en dit meta-model kan weer gedefinieerd worden op een meta-meta-modelniveau. Bézivin (Bézivin, 2006) beschrijft deze hiërarchie. Deze hiërarchie vertoont grote overeenkomsten met de hiërarchie van (contextvrije) grammatica's van (programmeer)talen (Kunert, 2006). Via meta-modellering zijn we in staat om de structuur van modelleringsformalismen te definiëren en transformaties tussen modellen te definiëren. Modeltransformaties worden eenvoudiger als voor zowel het invoermodel als het uitvoermodel een meta-model bestaat. De transformatie kan dan namelijk op dit meta-modelniveau worden beschreven. Het meta-model beschrijft echter alleen de structuur van het model en niet de semantiek, zoals (E)BNF of zoals SDF dat alleen de syntax beschrijft en niet de semantiek. Een transformatie van Cobol naar Java kan syntactisch wel, maar is semantisch niet gegarandeerd. Vergelijkbare semantische problemen kunnen ontstaan bij het vertalen van modellen.

In het vervolg van deze intreerede wil ik laten zien hoe generieke taaltechnologie een bijdrage kan leveren aan model-driven engineering en de ontwikkeling van robuuste softwaregeneratietechnologie in het bijzonder. Dit is een van de onderzoeksgebieden waar ik met mijn groep momenteel aan werk en ik hoop dit onderzoek de komende jaren verder uit te bouwen.

[5] (Extended) Backus Naur Form

Modeltransformaties en consistentie

Een voorbeeld van semantische inconsistentie hebben we ontdekt in het kader van onderzoek binnen het ESI-project Falcon, waarin we werken aan het vertalen van beschrijvingen van de diverse processen in een distributiecentrum naar executeerbare code. Op procesalgebra gebaseerde formalismen zoals Chi (van Beek et al, 2005), zijn uitermate geschikt om de processen in dit soort gedistribueerde systemen te modelleren. Naast op procesalgebra gebaseerde formalismen zijn er ook op Petri-net gebaseerde formalismen (Reisig, 1982). We hebben een meta-model in de vorm van een SDF-definitie voor de procesalgebra ACP (Bergstra en Klop, 1986) opgesteld en een transformatie van ACP naar UML state-machinediagrammen gerealiseerd. Deze state-machinediagrammen worden gebruikt voor het beschrijven van (parallel) gedrag. Een tool als Raphosody (Telelogic) kan deze diagrammen vertalen naar executeerbare code. Syntactisch is de transformatie van ACP naar UML state machines mogelijk, maar gezien de eisen betreffende structuur is er een zwaar mechanisme nodig om dit semantisch passend te krijgen.
De parallelle operator in ACP beschrijft meer dan het gedrag van twee parallelle processen. Acties in de argumenten van de parallelle operator kunnen, indien de specificatie dat toestaat, interageren en samensmelten tot één actie. Normaliter worden ACP-specificaties genormaliseerd en resulteert dit in een ACP-specificatie waarin alle parallelle operatoren zijn verdwenen en slechts het resultaat van eventuele interactie overblijft. Deze genormaliseerde specificaties beschrijven echter niet meer de oorspronkelijke structuur van het te modelleren systeem. UML state machines hebben een run to completion semantiek. Dat wil zeggen dat de uitvoering van acties niet onderbroken kan worden. Omdat acties dus na elkaar uitgevoerd worden is interactie onmogelijk. Om dit semantische gat te overbruggen en de structuur betreffende parallellisme te behouden, is het noodzakelijk een executieomgeving te introduceren. Deze omgeving reguleert het uitvoeren van acties en zorgt ervoor dat acties kunnen interageren en zo samensmelten tot één actie.

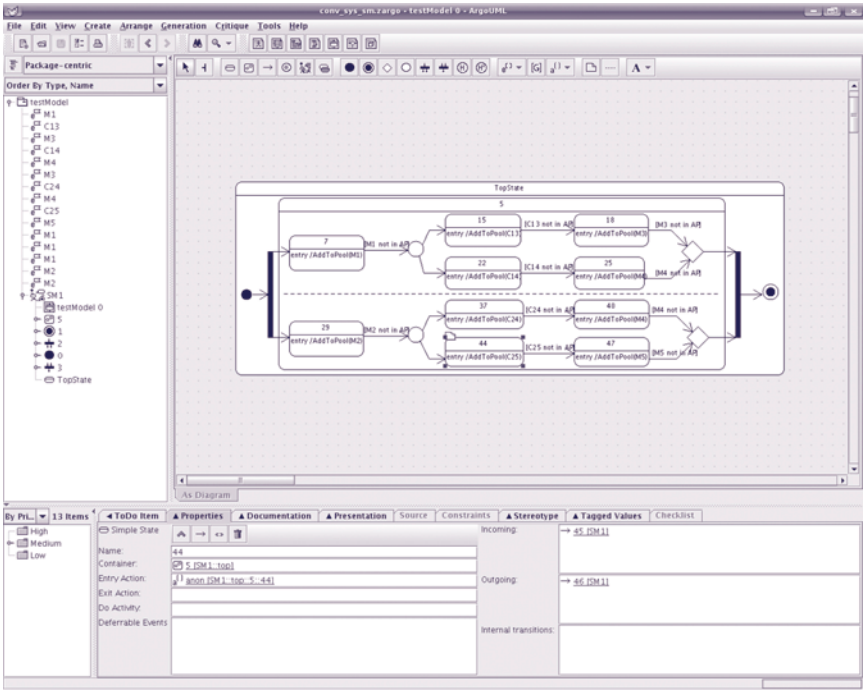
Via meta-modellen kunnen we eigen varianten van UML definiëren. Dit lijkt op het eerste gezicht een zeer veelbelovende route, een domeinspecifieke modelleringstaal gebaseerd op UML, misschien wel bedrijfsspecifiek. Domein- of bedrijfsspecifieke elementen in de specificatietaal kunnen het opstellen van specificaties aanzienlijk vereenvoudigen. De terminologie in de vorm van taalconstructies of

bibliotheken geeft al een kader dat op het probleemdomein is afgestemd. Veel bedrijven zijn deze route met groot enthousiasme aan het onderzoeken. Chi, mCRL2 (Groote et al, 2007) en POOSL (Theelen et al, 2007) zijn talen die wel regelmatig in het bedrijfsleven worden gebruikt, maar dan vaak alleen in de vorm van een proof-of-concept project waarbij de onderzoekers of ontwikkelaars van deze talen intensief betrokken zijn. Naast deze modelleringstalen worden (gekleurde) Petri-netten ook veelvuldig gebruikt voor modellering van (dynamische) processen (Van Hee et al, 2005). Medewerkers van bedrijven komen met deze modelleringstalen wel in aanraking, maar zullen ze zelf zelden gebruiken voor vervolgprojecten. UML heeft wel voet aan de grond gekregen in het bedrijfsleven. Hoewel steeds weer blijkt dat het moeizaam is om niet-onderzoekers formele methoden te laten gebruiken, zijn ze misschien wel bereid om te modelleren in een (beter gedefinieerde) aangepaste versie van UML. Vertalingen van modellen in die nieuwe versies van UML naar de bestaande formele talen bieden de mogelijkheid om gereedschappen voor analyse die nu al op de plank liggen voor een groot publiek toegankelijk te maken en hiermee de kwaliteit van de modellen te verbeteren.

Het ESI-project Ideals[6] had als doelstelling de onderhoudbaarheid van de code base van ASML te verbeteren. In het kader van Ideals hebben we onderzoek gedaan naar de mogelijkheid om UML-gedragsmodellen op te stellen die direct naar POOSL vertaald kunnen worden. POOSL werd bij ASML al gebruikt om onderdelen van de waferstepper te modelleren en op basis van die modellen prestatievoorspellingen te doen. Deze vertaling bleek niet haalbaar: enerzijds was het te omslachtig en anderzijds was er gebrek aan uitdrukkingskracht. Daarom was het vervolgonderzoek gericht op het integreren van POOSL in UML. De resulterende specificaties kunnen vervolgens in POOSL-code worden vertaald om prestatie-analyses uit te voeren. Het onderzoek richtte zich op het definiëren van eenvoudige UML-activitydiagrammen die eenvoudig naar POOSL of een andere imperatieve taal vertaald konden worden. De grafische syntax die UML populair maakt heeft ook nadelen, in het bijzonder wat betreft het modelleren van gedrag. De grafische syntax maakt het noodzakelijk vele kleine handelingen te verrichten om het gedrag te specificeren.
Een alternatief is het gebruik van surface languages, talen met een tekstuele syntax die hetzelfde meta-model als UML hebben. De leesbaarheid van modellen en het gemak waarmee ze gemaakt worden, kunnen verhoogd worden door surface languages voor delen van de gedragsmodellen te gebruiken. In het kader van Ideals is de mogelijkheid van het gebruik van surface languages om gedrag te

[6] http://www.esi.nl/ideals

beschrijven, onderzocht en is gekeken hoe dat vertaald kan worden naar POOSL of een andere imperatieve taal. Beide onderzoeksactiviteiten hebben aangetoond dat deze aanpak mogelijk is, maar dat de inspanning om dit met gereedschappen te ondersteunen bijzonder groot is. Ook hier is het mogelijk om een parallel met programmeertalen te trekken. Het is mogelijk de syntax van een domeinspecifieke taal te beschrijven, maar naast een syntaxdefinitie is het ook noodzakelijk om de semantiek te definiëren en gereedschappen te ontwikkelen. Verder is het nog maar zeer de vraag of dit soort activiteiten tot kwalitatief betere software leidt (Staron en Wohlin, 2006). De ontwikkelaars moeten zich namelijk een nieuwe taal en een nieuwe methodologie eigen maken en leren omgaan met nieuwe gereedschappen.
Modeltransformaties spelen een belangrijke rol in model-driven engineering, via modeltransformaties kunnen modellen omgezet worden in broncode en zo bijdragen aan het verhogen van de productiviteit. Hoe kunnen we de correctheid van deze transformatie garanderen of bewijzen? Hoe moeten deze correctheidsbewijzen in de gereedschappen ondersteund worden?



High-fidelity softwaregeneratie

Het op een efficiënte wijze produceren van kwalitatief hoogwaardige software kan onder meer bereikt worden door het inzetten van gereedschappen die de softwareontwikkeling vereenvoudigen en de kwaliteit van de geproduceerde software verhogen. Hierbij valt te denken aan softwareontwikkelingsomgevingen en/of programmeeromgevingen, waarbij de programmeur continu ondersteund wordt in zijn werkzaamheden, zoals het compileren, testen en herstructureren met bibliotheken van voorgedefinieerde componenten, versiemanagement en het documenteren van de code.

Softwaregeneratoren vergroten ook de efficiency van het ontwikkelen van kwalitatief hoogwaardige software. Model Driven Architecture® (MDA) (Kleppe, Warmer en Bast, 2003) van de Object Management Group (OMG)[7] is een initiatief waarbij UML als specificatietaal wordt gebruikt voor softwaregeneratie. Vanuit een hoogniveauspecificatie in UML wordt via transformaties een meer concreet model in UML afgeleid, waarbij bepaalde architectuurbeslissingen zijn ingevuld, bijvoorbeeld twee- versus drielagenarchitectuur. Deze transformaties vallen binnen hetzelfde meta-model en zijn semantiekbehoudend. Dit concretere model dient vervolgens als basis voor het genereren van de code voor een specifiek platform.

De softwaregeneratie beperkt zich vaak tot de statische aspecten gespecificeerd in het zogenaamde klassendiagram van UML. Het dynamische gedrag moet in bijvoorbeeld Java geprogrammeerd worden, waarbij natuurlijk wel gebruik gemaakt kan worden van de gegenereerde code. De MDA-ontwikkeling zou een enorme stap voorwaarts kunnen zijn, maar er moet nog behoorlijk wat onderzoek plaatsvinden voordat volledige applicaties gegenereerd kunnen worden.

Het kenmerk van programmageneratoren is dat zij stukken software kunnen genereren op basis van abstracte beschrijvingen van informatie. Het is bijvoorbeeld mogelijk om, uitgaande van de abstracte beschrijving van datatypen, bibliotheken voor het manipuleren van deze datatypen te genereren. Deze abstracte beschrijvingen kunnen bijvoorbeeld de klassendiagrammen uit UML zijn of een abstracte beschrijving in een XML-achtig[8] formalisme. Met behulp van deze automatisch gegenereerde bibliotheken kunnen vervolgens applicaties op een efficiënte en

[7] http://www.omg.org

[8] eXtensible Markup Language

eenvoudige wijze worden geprogrammeerd. Het saaie werk van het ontwikkelen van de basisdatatypen is uitbesteed aan de generator. Bij veranderingen in de datatypen, hoeft de programmeur alleen maar de abstracte beschrijving aan te passen en de bibliotheken te hergenereren.
Het ontwikkelen van een goede softwaregenerator is een behoorlijke investering en moet pas gebeuren als men zeker weet dat de generator meerdere malen gebruikt kan worden. ApiGen is een voorbeeld van een softwaregenerator die binnen de ASF+SDF- Meta-Environment diverse malen goede diensten heeft bewezen en nog steeds bewijst. ApiGen is een zogenaamde string-based generator. De te genereren code is verpakt in strings, die naar een uitvoerbestand worden geschreven. De generator en de te genereren code zijn helemaal met elkaar verweven (zie Figuur 1 voor voorbeeldcode). De syntactische correctheid van de gegeneerde code is niet gegarandeerd en wordt pas gecheckt als de gegeneerde code wordt gecompileerd. Völter (Völter, 2003) geeft een overzicht van de verschillende typen softwaregeneratoren.

```
private void genConstructor() {
 println("  public "
       + getClassName()
       + "("
       + factoryClassName
       + " abstractTypeFactory, aterm.ATermList annos, aterm.AFun fun,
aterm.ATerm[] args) {");
 println("    super(abstractTypeFactory.getPureFactory(), annos, fun,
args);");
 println("    this.abstractTypeFactory = abstractTypeFactory;");
 println("  }");
 println();
}
```

figuur 1

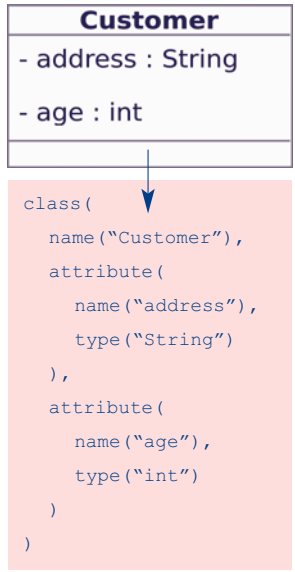
Voorbeeldcode uit ApiGen

```
public class <%class/name%>{
 <%foreach class/attribute do%>
 private <%type%> <%name%>;
 public <%type%> <%"get"||name%>(){
       return <%name%>
 }
 public void <%"set"||name%>(<%type%> <%name%>){
   this.<%name%>=<%name%>;
 }
 <%od%>
}
```

figuur 2

Java-template: Java-code met gaten

Een ander type softwaregenerator is de template-based generator. Deze generator is eveneens gebaseerd op strings, maar de generator en de te genereren code zijn losgekoppeld. Een template of sjabloon is een stuk programmacode met gaten (zie Figuur 2). De gaten zijn plaatsen waar informatie uit een model moet worden ingevuld. Uitgaande van informatie uit een invoerbestand worden deze gaten gevuld en ontstaat er 'echte' programmacode. De generator en de templates zijn entiteiten die los staan van elkaar, dit in tegenstelling tot de string-based generatoren. Het nadeel van de string-based generatoren is dat de correctheid van de genereerde code pas door compilatie vast te stellen is. Dit geldt eveneens voor template-based generatoren. Dit nadeel is echter te ondervangen door het inzetten van generieke taaltechnologie.

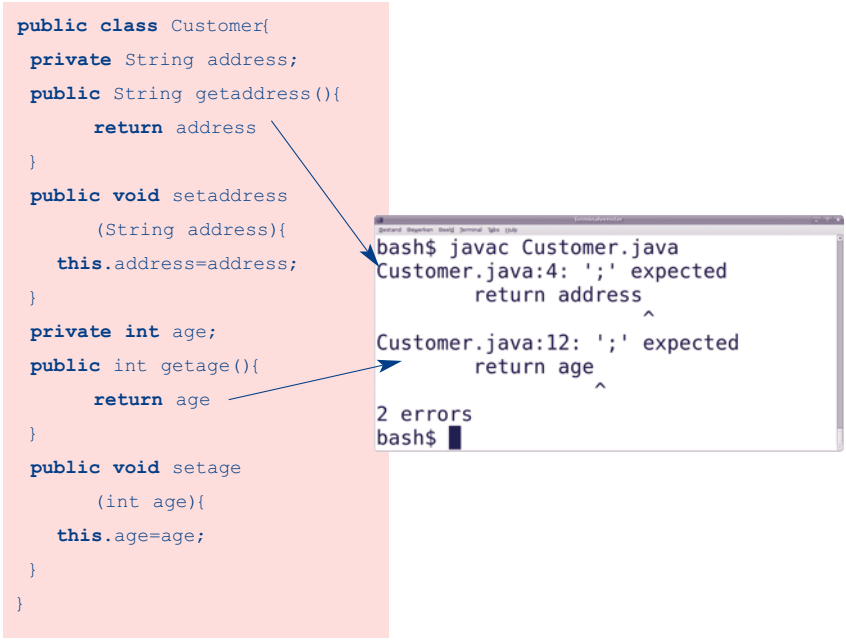


figuur 3
Invoer-data

Een template in een template-based generator is eigenlijk niets anders dan een stukje programmacode dat opgebouwd is uit twee verschillende talen: de programmeertaal voor programmacode en een meta-taal voor de gaten. Door nu de taalbeschrijving van de programmeertaal met de taalbeschrijving van de meta-taal te combineren zijn we in staat in een vroeg stadium, voordat de programmacode gegenereerd is, al de syntactische correctheid van de template te garanderen. Figuur 2 laat de Java-code met gaten zien en Figuur 3 de invoerdata. Figuur 4 laat de gegenereerde code zien en de foutmeldingen van de Java-compiler, namelijk de ontbrekende puntkomma's. Deze manier van het controleren van templates op syntactische fouten is niet alleen toepasbaar voor Java, maar voor iedere taal waar een contextvrije grammatica in SDF voor bestaat. Zo kunnen we bijvoorbeeld ook HTML-templates valideren op syntactische correctheid (zie Figuur 5). Dit betekent dat in een vroeg stadium alle syntactische fouten opgespoord kunnen worden, ook fouten ten gevolge van incorrecte invoerdata; zie Arnoldus et al (2007) voor meer details.

Een belangrijke vraag is of het mogelijk is om statisch semantische fouten in templates in een vroeg stadium op deze wijze op te sporen. Het volledig statisch semantisch controleren van een template is waarschijnlijk niet mogelijk, tenzij er een volledige typechecker voor de brontaal gebouwd wordt die kennis heeft van de geïntegreerde meta-taal. Het blijkt dat het in het geval van Java- en C-templates mogelijk is om te controleren of aanroepen naar klassen en methoden in de onder-

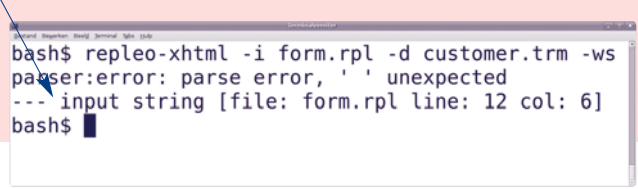
liggende bibliotheken correct zijn. Na syntactische analyse wordt de opgebouwde syntaxboom doorlopen en worden feiten zoals klassenamen, variabelen, definities en aanroepen van methoden, geëxtraheerd. Deze feiten worden in een database opgeslagen. Bovendien wordt bepaald welke softwarebibliotheken er gebruikt zijn, bijvoorbeeld de AWT-bibliotheek[9]. Uit deze bibliotheken worden ook allerlei feiten geëxtraheerd. De combinatie van beide verzamelingen feiten wordt gebruikt om te bepalen of de gebruikte typen en methoden wel beschikbaar zijn via de bibliotheken. Als dat niet het geval is, dan resulteert dat in een foutmelding. In een aantal gevallen is het echter mogelijk om simpele statisch semantische controles uit te voeren, bijvoorbeeld op de aanwezigheid van dubbel gedeclareerde variabelen. Het controleren van Java- en C-templates is mogelijk gegeven de gebruikte bibliotheken, maar een van de onderzoeksvragen is hoe generiek deze aanpak is. Is het mogelijk op een vergelijkbare wijze templates in andere talen te controleren of combinaties van talen zoals Java met SQL? Een andere onderzoeksvraag is of het semantisch controleren van de programmeertaal in combinatie met de meta-taal mogelijk is.



figuur 4
Resulterende gegenereerde code

[9] Abstract Window Toolkit

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
 "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<html>
 <head>
  <title><%class/name || " form"%></title>
 </head>
 <body>
  <form action="/commit" method="post" />
  <%foreach class/attribute do%>
   <p>
    <%name%> <br />
    < input type="text"
      name=<%"\""||name||"\""%> size="20">
   </p>
  <%od%>
   <p>
    <input type="submit" value="Submit">
    <input type="reset" value="Reset">
   </p>
   </form>
  </body>
</html>
```



figuur 5

Syntactisch incorrect HTML-template

Alternatieve benaderingen voor consistentie

Tot nu toe heb ik hoofdzakelijk stilgestaan bij het genereren van broncode uit modellen. Er zijn echter ook andere manieren om model en broncode consistent te houden. Een van die manieren is om uit het (formele) model de broncode af te leiden met behulp van correctheidsbehoudende transformaties. Deze manier van softwareontwikkeling leunt sterk op de onderliggende programmeeromgeving. De omgeving zal niet alleen de transformatiestappen moeten ondersteunen maar ook hulp moeten bieden bij het uitvoeren van correctheidsbewijzen. In het VIDE-project[10] werken diverse leden van de Software Engineering and Technology groep aan het ontwikkelen van een dergelijke omgeving.
Tenslotte is het ook mogelijk om modellen uit broncode te extraheren. Reverse engineering heeft zich ontwikkeld tot een belangrijk vakgebied en gezien de groei van de hoeveelheid software wordt de vraag naar krachtige reverse-engineering-gereedschappen steeds groter. Het inzichtelijk maken van de structuur en complexiteit van broncode wordt steeds belangrijker en een breed scala aan analyse- en visualisatiegereedschappen voor verschillende programmeertalen staat de reverse engineer ter beschikking. Het extraheren van gedragsmodellen, bijvoorbeeld sequencediagrammen of state-machinediagrammen uit broncode staat echter nog in de kinderschoenen. De geëxtraheerde modellen bevatten teveel detailinformatie om bruikbaar te zijn voor analysedoeleinden. In samenwerking met LaQuSo hebben we meegewerkt aan de ontwikkeling van gereedschappen om gedragsmodellen uit C++-code te extraheren (Korshunova et al, 2006). LaQuSo (Laboratory for Quality Software)[11] is een van de drie NIRICT-laboratoria die voortkomen uit de 3TU-samenwerking. LaQuSo richt zich primair op de verificatie en validatie van software-artefacten.

[10] Verifying Integrated Development Environment

[11] http://www.laquso.com

Om de consistentie tussen model en broncode te kunnen realiseren moeten de opgestelde modellen helder en eenduidig zijn. Heldere en eenduidige modellen spelen ook een belangrijke rol bij de offshoring van softwareontwikkeling om de correctheid van de ontwikkelde software te kunnen garanderen. Dat de modellen helder en eenduidig zijn betekent dat de modellen ook als uitgangspunt voor softwaregeneratie kunnen dienen. Dit betekent weer dat we studenten moeten onderwijzen in het opstellen van hoogwaardige en correcte modellen en bovendien in het ontwerpen en bouwen van softwaregeneratoren.

Informatica-onderwijs in Nederland

Het gaat niet goed met het informatica-onderwijs in Nederland. De studentenaantallen lopen al jaren terug en zelfs economische oplevingen leiden niet meer tot een stijging van de instroom van studenten. Dit gebeurt juist in een periode waarin steeds meer informatici nodig zijn vanwege de toenemende hoeveelheid software. Paul Klint gaf een zeer heldere en scherpe analyse van dit probleem als keynote speaker tijdens NIOC 2007, waar hij over 'ICT inside' sprak. Kinderen zijn waarschijnlijk de grootste afnemers van producten die voortkomen uit de ICT, zoals games, internet en mobiele telefoons. Zij realiseren zich echter onvoldoende dat in al deze producten en diensten software zit. Het is onduidelijk hoe we het tij kunnen keren. De TU/e geeft informaticalessen op middelbare scholen. Misschien zouden we ook informaticalessen op de basisschool moeten geven. Techniek, en dan met name ICT, moet weer hoog op de agenda komen te staan, van zowel middelbare als basisscholen. Het beroepsperspectief is zonder meer goed, informatica is een spannend vak met vele toepassingen, zowel in het domein van entertainment, als in het medische en het sociale domein. Het zijn deze aspecten die we meer en beter zullen moeten benadrukken bij onze aankomende studenten.

Een meer inhoudelijk aspect van het informatica-onderwijs is dat studenten beter onderwijs moeten krijgen in het leren omgaan met grote bestaande softwaresystemen. Hierbij maakt het niet uit of het modellen of broncode betreft. Het kunnen begrijpen van andermans werk is cruciaal en de mentaliteit van not invented here or by me moet nog verder teruggedrongen worden. We hebben in de afgelopen decennia mensen opgeleid met het idee dat je modelleerde of programmeerde voor de computer. We zullen nu mensen moeten gaan opleiden die niet alleen modelleren of programmeren voor de computer maar ook voor elkaar: modelleren/programmeren is communiceren. Het lezen van programmacode en het daadwerkelijk begrijpen wat er gebeurt, komt onvoldoende aan bod in de diverse informatica-opleidingen. Juist deze vaardigheden zijn van cruciaal belang bij de toekomstige beroepsuitoefening van de studenten. Ongeveer zeventig procent van de informatici is werkzaam op het gebied van onderhoud van software. Ook als de code in India wordt ontwikkeld is code-inspectie essentieel om de kwaliteit van het resultaat te beoordelen. Bovendien is het leren lezen van andermans code belangrijk om op effectieve wijze gebruik te maken van componenten uit

bibliotheken. Veel te vaak gebruiken studenten de trial-and-errormethode bij het programmeren met bibliotheken om te zien wat er gebeurt als een functie of methode wordt aangeroepen. Het bestuderen van de specificatie of broncode van een methode levert meer inzichten op en een beter beeld of de methode al dan niet geschikt is. Men spoort via gedegen code-inspecties of code walkthroughs meer fouten op dan met welke andere vorm van testen dan ook (McConnell, 2004). Via het doorgronden van verschillende programmeerstijlen leert de student ook veel over zijn eigen programmeerstijl.

Het programmeeronderwijs zou dan ook veel sterker gericht moeten zijn op het uitbreiden van bestaande niet-triviale software. Een voorwaarde daarbij is wel dat de studenten naast het leren van de diverse taalconstructies en de semantische aspecten van een programmeertaal ook leren omgaan met gereedschappen die het analyseren van programma's ondersteunen. Studenten zouden overigens in staat moeten zijn om dit soort analysegereedschappen zelf te ontwikkelen. Hiervoor moeten de studenten echter wel basistechnieken uit de generieke taaltechnologie onderwezen krijgen. Dit kan men onderwijzen aan de hand van programmageneratoren of domeinspecifieke talen. Aspecten zoals het maken van taalbeschrijvingen en het ontwikkelen van gereedschappen voor het scannen en ontleden van programmateksten komen hierbij van pas. In het vak Generic Language Technology dat ik in de masteropleiding Computer Science Engineering geef, komen deze technieken uitgebreid aan bod.
Informatica is een jong vakgebied met mooie toepassingen en grote uitdagingen. Dit zou aankomende studenten moeten aanspreken. Het is aan ons om de informaticavlam over te dragen aan volgende generaties. Doen we dat niet, dan zal Nederland op ICT-gebied afglijden tot een derdewereldland.

Dankwoord

Het punt waar ik nu ben in mijn carrière is ver voorbij de dagdromen (kok of bakker) die ik had op de MAVO. Het was een lange weg van de MAVO, via HAVO en VWO, naar de universiteit, gevolgd door een promotie, lectoraat en leerstoel. Ik wil mijn familie, in het bijzonder mijn moeder, en mijn vrienden bedanken voor de steun die zij mij in al die jaren hebben gegeven om te komen waar ik nu ben.
Ik heb tot nu toe in mijn academische carrière een groot aantal mensen mogen ontmoeten. Het is onmogelijk om iedereen hier te noemen, maar er is een aantal mensen dat ik in het bijzonder wil bedanken omdat zij hebben bijgedragen aan wat ik nu bereikt heb: Jos Baeten, Jan Bergstra, Marjan Freriks, Jan Friso Groote, Kees van Hee, Paul Klint, Kees Koster, Erik Meijer, Hans Meijer en Koos Rooda.
Ook wil ik alle promovendi die ik heb mogen (mee)begeleiden en momenteel begeleid, Marcel van Amstel, Jeroen Arnoldus, Loek Cleophas, Yanja Dajsuren, Luc Engelen, Joris Hillebrand, Hayco de Jong, Wilco Koorn, Christian Lange, Ronald Middelkoop, Pieter Olivier, Zvezdan Protic, Jackline Ssanyu, Ke Sun en Jurgen Vinju bedanken. Verder wil ik de leden van de Software Engineering and Technology groep bedanken voor hun vertrouwen en flexibiliteit, zij hebben in de afgelopen twee jaar niet alleen de naam van de groep van Software Construction naar Software Engineering and Technology zien veranderen maar ook de onderzoeksrichting en het ambitieniveau. Eveneens dank ik iedereen die deze intreerede heeft proefgelezen.
Tenslotte bedank ik José voor al haar steun en zorg in de 28 jaar die we samen door het leven gaan. Ik maak het haar absoluut niet eenvoudig met mijn vele korte reizen en soms hele lange buitenlandse reizen en toch blijft zij mij steunen, motiveren en inspireren om verder te gaan op het carrièrepad dat ik ingeslagen ben. Zij heeft door op zaterdagen of zondagen weg te zijn, mij in de gelegenheid gesteld deze rede te schrijven. Ik hoop dat we nog heel lang en gelukkig samen onze toekomst en carrières kunnen vormgeven.

Referenties

Alblas, H., Melichar, B. (eds): *Attribute Grammars, Applications and Systems, International Summer School SAGA*, LNCS 545, 1991.

Van Amstel, M.F., Van den Brand, M.G.J., Protic, Z., Verhoeff, T., 'Transforming Process Algebra Models into {UML} State Machines: Bridging a Semantic Gap?', To appear in: *Proceedings of the International Conference on Model Transformations*, 2008.

Arnoldus, J., Bijpost, J., Van den Brand, M.G.J., 'Repleo: a syntax-safe template engine', In *Proceedings 6th International Conference on Generative Programming and Component Engineering (GPCE2007)*, p 25-32, 2007.

Van Beek, D.A., Man, K.L., Reniers, M.A, Rooda, J.E., Schiffelers, R.R.H., 'Syntax and semantics of timed Chi'. CS-Report 05-09, Department of Computer Science, Eindhoven University of Technology, 2005.

Bergstra, J.A., Klint, P., 'The discrete time ToolBus – a software coordination architecture', In: *Science of Computer Programming* 31(2-3):205-229, 1998.

Bergstra, J.A., Klop, J.W. 'Algebra of communicating processes'. In de Bakker, J.W., Hazewinkel, M., Lenstra, J.K. (eds) *Proceedings of the CWI Symposium*. Volume 1 of CWI Monographs, Centre for Mathematics and Computer Science, North-Holland, p. 89-138, 1986.

Bézivin, J., 'Model Driven Engineering: An Emerging Technical Space'. In: Lämmel, R., Saraiva, J., Visser, J., (eds), *Generative and Transformational Techniques in Software Engineering (GTTSE 2005)*, 36-64, LNCS 4143, 2006.

Booch, G., *Object-oriented Analysis and Design with Applications*, 2nd ed., Benjamin Cummings, 1993.

Van den Brand, M.G.J., *Pregmatic; A Generator For Incremental Programming Environments*, Proefschrift, Katholieke Universiteit Nijmegen, 1992.

Van den Brand, M.G.J., *Softwarekwaliteit – Hypes versus onderzoek in de software engineering*, Openbare les, Hogeschool van Amsterdam, 2004[12].

Van den Brand, M.G.J., Van Deursen, A., Heering, J., De Jong, H.A., De Jonge, M., Kuipers, T., P. Klint, Moonen, L., Olivier, P.A., Scheerder, J., Vinju, J.J., Visser, E., Visser, J., 'The ASF+SDF Meta-Environment: a Component-Based Language Development Environment', In: R. Wilhelm (ed), *Proceedings of Compiler Construction (CC'01)*, LNCS 2027, p. 365-370, 2001.

Van den Brand, M.G.J., De Jong, H.A., Klint, P., Olivier, P.A., 'Efficient annotated terms', In: *Software : Practice and Experience*, *30*(3), 259-291, 2000.

Van den Brand, M.G.J., Heering, J., Klint, P., Olivier, P.A., 'Compiling language definitions: the ASF+SDF compiler', In: *ACM Transactions on Programming Languages and Systems*, *24*(4), 334-368, 2002.

Van den Brand, M.G.J., Klint, P. en Vinju, J.J., 'Term rewriting with traversal functions', In: *ACM Transactions on Software Engineering and Methodology*, 12(2), p. 152-190, 2003.

Van den Brand, M.G.J, Moreau, P.-E. en Vinju, J.J. *A generator of efficient strongly typed abstract syntax trees in Java*, Technical report, SEN-E0306, 2003.
Van den Brand, M.G.J. and Visser, E., 'Generation of formatters for context-free languages', In: *ACM Transactions on Software Engineering and Methodology*, 5, p. 1-41, 1996.

Campbell-Kelly, M., From Airline Reservations to Sonic the Hedgehog – A History of the Software Industry, The MIT Press, 2003.

Van Deursen, A., *The Software Evolution Paradox*, intreerede, Technische Universiteit Delft, 2005.

Van Deursen, A., Heering, J. en Klint, P., *Language Prototyping: An Algebraic Specification Approach*, World Scientific, 1996.

Van Deursen, A. en Klint, P., 'Little languages: Little maintenance?', In: *Journal of Software Maintenance*, 10, p. 75-92, 1998.

[12] http://www.hva.nl/lectoraten/documenten/olo7-041215-vandenbrand.pdf

Glass, R.L., *Building Quality Software*, Prentice-Hall, 1992.

Groote, J.F., Mathijssen, A.H.J., Reniers, M.A., Usenko, Y.S., Van Weerdenburg, M.J., 'The Formal Specification Language mCRL2', In: E. Brinksma, D. Harel, A. Mader, P. Stevens, R. Wieringa, (eds), *Methods for Modelling Software Systems (MMOSS)*, Dagstuhl Seminar Proceedings 06351, Internationales Begegnungs- und Forschungszentrum fuer Informatik (IBFI), Schloss Dagstuhl, Germany, 2007.

Van Hee, K.M., Oanea, O.I., Sidorova, N., 'Colored Petri nets to verify extended event-driven process chains', In R. Meersman, Z. Tari (eds), *On the Move to Meaningful Internet Systems 2005*, LNCS 3760, p. 183-201, Springer-Verlag, 2005.

Heemstra, F.J., Kusters, R.J. en Trienekens, J.J.M., *Sofwarekwaliteit; Op weg naar betere software*, ten Hagen & Stam, 2001.

Jacobson, I., Christerson, M., Jonsson, P., Overgaard, G., *Object-Oriented Software Engineering: A Use Case Driven Approach*, Addison-Wesley, 1992.

De Jong, H.A., en Olivier, P.A., 'Generation of abstract programming interfaces from syntax definitions' In: Journal of Logic and Algebraic Programming, 59, p. 35-61, 2004.

Kleppe, A., Warmer, J. en Bast, W., *MDA Explained; The Model Driven Architecture™: Practice and Promise*, Addison-Wesley, 2003.

Klint, P., 'A Meta-Environment for Generating Programming Environments' In: *ACM Transactions on Software Engineering and Methodology*, 2(2), p. 176-201, 1993.

Korshunova, E., Petkovic-Ilic, M., Van den Brand, M.G.J., Mousavi, M., 'CPP2XMI : reverse engineering of UML class, sequence, and activity diagrams from C++ source code (Tool Paper)', In: *Proceedings 13th Working Conference on Reverse Engineering (WCRE'06)*, IEEE, 2006.

Kunert, A., 'Semi-Automatic Generation of Metamodels and Models from Grammars and Programs', In: Proceedings of Fifth International Workshop on Graph Transformations and Visual Modeling Techniques, ETAPS, 2006.

McConnell, S., *Code Complete 2*, Microsoft Press, 2004.

Reisig, W., *Petri Nets: An Introduction*, Springer-Verlag, 1982.

Reps, T.W., Teitelbaum, T., *The Synthesizer Generator: a system for constructing language-based editors*, Springer-Verlag, 1989.

Royce, W.W., 'Managing the development of large software systems', In: *Proceeding of IEEE WESCON*, p. 1-9, 1970[13].

Rumbaugh, J.E., Blaha, M.R., Premerlani, W.J., Eddy, F., Lorensen, W.E., *Object-Oriented Modeling and Design*, Prentice-Hall, 1991.

Staron, M. en Wohlin, C., 'An Industrial Case Study on the Choice Between Language Customization Mechanisms', p. 177-191, In: Münch, J. en Vierimaa, M., Product-*Focused Software Process Improvement*, LNCS 4034, 2006.

Theelen, B.D., Florescu, O., Geilen, M.C.W., Huang, J., Van der Putten, P.H.A., Voeten, J.P.M., 'Software/Hardware Engineering with the Parallel Object-Oriented Specification Language', In: *Proceedings of the ACM-IEEE International Conference on Formal Methods and Models for Codeesign (MEMOCODE)*, p.139-148, IEEE, 2007.

Völter, M., 'A Catalog of Patterns for Program Generation', In: Eighth European Conference on Pattern Languages of Programs, 2003.

[13] http://facweb.cs.depaul.edu/jhuang/is553/Royce.pdf

Curriculum Vitae

Prof.dr. Mark van den Brand is per 1 januari 2006 benoemd tot voltijds hoogleraar op het gebied van software engineering en technology aan de faculteit Wiskunde en Informatica van de Technische Universiteit Eindhoven (TU/e).

Mark van den Brand (1962) begon in 1982, na een lange Mammoettijd (MAVO, HAVO, VWO), met zijn studie Informatica aan de Katholieke Universiteit Nijmegen (tegenwoordig Radboud Universiteit Nijmegen) waar hij in 1987 afstudeerde en in 1992 promoveerde. Vervolgens heeft hij vijf jaar gewerkt als universitair docent aan de Universiteit van Amsterdam en is hij van 1997 tot 2006 werkzaam geweest bij het CWI (Centrum voor Wiskunde en Informatica). Daarnaast heeft hij gedurende veertien maanden bij een INRIA-instituut in Nancy onderzoek gedaan. Ook heeft hij in deeltijd als universitair hoofddocent bij de Vrije Universiteit gewerkt en was hij lector Software Kwaliteit bij de Hogeschool van Amsterdam. Sinds zijn afstuderen heeft het ontwikkelen van generieke taaltechnologie zijn interesse. Van den Brand heeft over dit onderwerp diverse publicaties in vooraanstaande tijdschriften en conferentieproceedings op zijn naam staan. Hij benadert het onderzoek op het gebied van software engineering zeer pragmatisch. Om zijn ideeën te valideren gaat hij het ontwerpen en implementeren van prototypes niet uit de weg. De toepasbaarheid van de ontwikkelde software voor het oplossen van niet-triviale (industriële) problemen heeft voor hem een hoge prioriteit.

Colofon

Productie
Communicatie Expertise Centrum TU/e
Communicatiebureau Corine Legdeur

Fotografie cover
Rob Stork, Eindhoven

Ontwerp
Grefo Prepress, Sint-Oedenrode

Druk
Drukkerij van Santvoort, Eindhoven

ISBN 978-90-386-1336-9
NUR 918

Digitale versie:
www.tue.nl/bib/

Bezoekadres
Den Dolech 2
5612 AZ Eindhoven

Postadres
Postbus 513
5600 MB Eindhoven

Tel. (040) 247 91 11
www.tue.nl

Technische Universiteit
Eindhoven
University of Technology