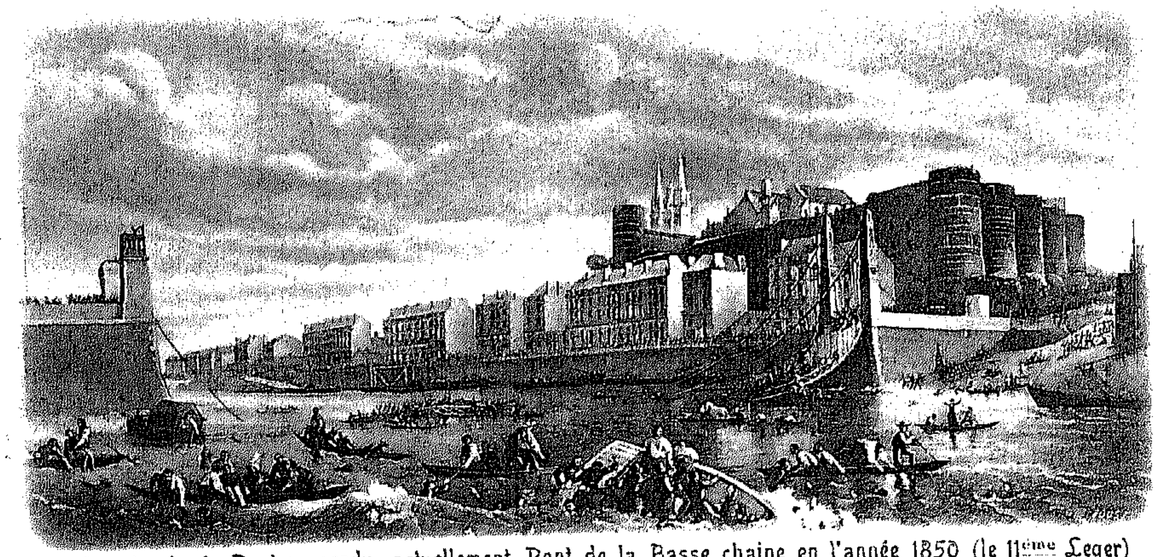
De Softwarerevolutie

Rede uitgesproken door prof. dr. P. Klint op zijn afscheidscollege als hoogleraar Software Engineering aan de Universiteit van Amsterdam op 22 januari 2014.



Catastrophe du Pont suspendu, actuellement Pont de la Basse chaine en l'année 1850 (le 11ème Leger)

Angers

2885

Samenvatting

Op 16 april 1850 marcheerde een regiment soldaten over een brug over de rivier de Maine bij Angers, Frankrijk. De brug begon te resoneren op de maat van de voetstappen en stortte in. Op 7 november 1940 stortte een nieuwe brug in Tacoma Narrows in de Amerikaanse staat Washington in door een constructiefout. Er bestaan nog veel meer voorbeelden van instortende bruggen. Is er soms sprake van een 'bruggencrisis'? Natuurlijk niet, dit zijn normale groeistuipen van technologie die zich ontwikkelt. Elke fout of ramp is betreurenswaardig maar leidt uiteindelijk tot inzicht en verbetering.

Toch bedacht in oktober 1968 een aantal onderzoekers de term 'softwarecrisis'. De term is gebruikt als publiciteitsmiddel om nieuwe onderzoekssubsidies aan te trekken maar getuigt van gebrek aan historisch inzicht in de evolutie van wetenschap en technologie, en is uiteindelijk schadelijk gebleken voor het hele vakgebied van de software engineering. Software krijgt alleen nog maar aandacht in de media als er iets fout gaat.

Een beter perspectief is dat van een revolutie: software heeft in de ruim 70 jaar van zijn bestaan een duizelingwekkende ontwikkeling ondergaan en maakt daardoor tot voor kort ondenkbare toepassingen mogelijk. Er is zeker ook sprake van 'instortende softwaresystemen' met grote materiële gevolgen en soms zelfs verlies van mensenlevens. Desondanks is software het fundament geworden van elke maatschappelijke, economische, industriële of wetenschappelijke activiteit.

Wat zijn de essentiële ideeën die deze softwarerevolutie mogelijk hebben gemaakt en hoe zullen deze zich verder ontwikkelen? Waarom moeten we blijven investeren in onderzoek op dit gebied? Terugblik én vooruitblik op het vakgebied van de software engineering.

Bruggencrisis?

Op 16 april 1850 marcheerde een regiment soldaten over een brug over de rivier de Maine bij Angers, Frankrijk. De brug begon te resoneren op de maat van de voetstappen en stortte in. Het resultaat is te zien in bovenstaande illustratie. Sindsdien lopen soldaten nooit meer in het gelid over een brug. Op 7 november 1940 stortte een nieuwe brug in Tacoma Narrows in de Amerikaanse staat Washington in door een constructiefout. Er bestaan nog veel meer voorbeelden van instortende bruggen. Is er soms sprake van een 'bruggencrisis'? De vooraanstaande bruggenontwerper Othmar Amman heeft het Tacoma Narrows incident onderzocht en concludeert:[1]

"The Tacoma Narrows bridge failure has given us invaluable information... It has shown [that] every new structure [that] projects into new fields of magnitude involves new problems for the solution of which neither theory nor practical experience furnish an adequate guide. It is then that we must rely largely on judgement and if, as a result, errors, or failures occur, we must accept them as a price for human progress."

Deze ongevallen zijn normale groeistuipen van technologie die zich ontwikkelt. Elke fout of ramp is betreurenswaardig, maar leidt uiteindelijk tot inzicht en verbetering.

Softwarecrisis? Softwarerevolutie!

Het waren mistige najaarsdagen in oktober 1968. Tegen de achtergrond van de heuvels van Garmisch-Partenkirchen spraken de deelnemers aan een door de NAVO-gesponsorde conferentie over het onderwerp 'Software Engineering', een nog nauwelijks gebruikte aanduiding van een nog erg jong vakgebied. Peter Naur, Brian Randell, Edgser Dijkstra en een dozijn andere onderzoekers maakten zich zorgen over kwaliteit en ontwikkelwijzen van programmatuur (een mooi Nederlands woord maar ik zal hierna steeds over 'software' spreken). Hun zorgen waren zo groot dat ze besloten om ze een naam te geven. Het begrip 'softwarecrisis' was geboren en is vele decennia door onderzoekers gebruikt als publiciteitsmiddel om het belang van de kwaliteit van software onder de aandacht van de overheid en het grote publiek te brengen. En natuurlijk ook om nieuwe onderzoekssubsidies aan te trekken.

Maar heeft de softwarecrisis ook werkelijk bestaan? Er storten regelmatig vliegtuigen neer, maar is er sprake van een vliegtuigcrisis? Ook bruggen bezwijken, maar is er een bruggencrisis? Zoals hierboven al aangegeven, denk ik[2] dat er in geen van deze gevallen, en ook niet bij software, sprake van een

---

1 http://en.wikipedia.org/wiki/Tacoma_Narrows_Bridge_(1940)#A_lesson_for_history

2 Ik gebruik in deze tekst 'ik' als het om een opinie of ervaring van mijzelf gaat en 'wij' als het geza-

crisis is (geweest). Software is een relatief nieuwe productiefactor die spectaculaire veranderingen heeft veroorzaakt in onze maatschappij. Software heeft alle kenmerken van een oerkracht die zich van zijn ketenen ontdaan heeft en die we maar met moeite kunnen temmen. Gaat alles goed? Zeker niet! Kan het beter? Jazeker! Softwarecrisis? Nee! Softwarerevolutie? Ja, adembenemend! Ik wil u graag meenemen in een persoonlijke tocht langs enkele hoogtepunten van de softwarerevolutie.

Een nieuwe liefde

Op mijn twaalfde wist ik het zeker: ik word natuurkundige. Geïnspireerd door de illustraties in de natuurkundeboeken die ik van mijn oom Jan gekregen had, en door boeken als *Het Draadloos Amateurstation* deed ik het ene experiment na het andere. Onze buren hebben geloof ik nooit geweten dat hun radio-ontvangst zo slecht was dankzij mijn vonkzenders. Hoe groter de vonk, hoe groter de vreugde. Maar na een succesvolle start met de studie natuurkunde sloeg de twijfel toe. Geen enkel experiment op het practicum gaf de goede uitkomst en alleen door de gemeten waarden te manipuleren kwam ik in de buurt van de te verwachten waarden. Ik vond dit zo teleurstellend dat ik al snel naar de studie wiskunde ben overgestapt. Naast standaardvakken als Lineaire Algebra en Functietheorie trokken twee vakken daar mijn aandacht: Numerieke Wiskunde en Programmeren van Rekenautomaten.

Numerieke Wiskunde werd gegeven door Prof. dr. ir. A. Van Wijngaarden. Een parmantige, altijd bijzonder netjes geklede man, die fenomenaal college kon geven in een stijl die ik heb proberen na te volgen: tijdens het college werd je als student geënthousiasmeerd voor het behandelde onderwerp en je verliet de collegezaal in de volle overtuiging dat je het helemaal snapte. Als je thuis je aantekeningen uitwerkte en aan de opgaven begon dan ontdekte je pas de complicaties waar de docent zo meesterlijk langs gesurfd was. De docent wekt belangstelling, de student ontdekt zelf de complicaties van een onderwerp. Ik vind dit nog steeds een uitstekend didactisch model. Ik heb Van Wijngaardens college driemaal gevolgd en bij elke volgende editie kwamen er meer elementen in van de programmeertaal Algol68 die hij op dat moment aan het ontwerpen was. Op sommige momenten droeg hij zelfs de regels van de Algol68 grammatica voor op een jambisch metrum. Ik hoor het hem nog zeggen: ROWSETY ROWSETY NONROW SLICE. Van Wijngaarden was ook directeur van het Mathematisch Centrum (nu: Centrum Wiskunde & Informatica) waar ik na mijn studie ben gaan werken. Ik ben er erg trots op dat Van Wijngaardens erfgenamen mij zijn toga hebben geschonken en dat ik die bij alle officiële gelegenheden en ook tijdens het uitspreken van deze rede heb kunnen dragen.

Links: Prof. dr .ir. A. Van Wijngaarden

Rechts: Prof. dr. F.E.J. Kruseman Aretz

Het college Programmeren van Rekenautomaten werd gegeven door Prof. dr. F.E.J. Kruseman Aretz en ik heb ook daarvan enkele edities gevolgd. Hij behandelde daarin onderwerpen als computerarchitectuur, vertalerbouw, programmeertalen, maar ook wat we nu software engineering noemen. Ik herinner me bijvoorbeeld de OEBRA: de Onbestaanbaar Eenvoudige Binaire RekenAutomaat, een Algol60 programma dat de werking van een eenvoudige computer simuleerde. Of een simulator van een Turingmachine waar de oneindige tape voorgesteld werd door een call-by-name parameter. Maar het hoofdbestanddeel van het college was de Algol60 compiler voor de Electrologica X8. Concepten als stapel, recursie en display werden in detail besproken. Ook de correctheid en testbaarheid van de compiler waren belangrijke onderwerpen. Ik heb aan het einde van mijn studie mijn scriptie geschreven over een geautomatiseerd bewijs van de terminatie van de Algol60 compiler. De precieze en elegante wijze waarop Kruseman Aretz deze onderwerpen behandelde, hebben mijn belangstelling voor vertalerbouw definitief gewekt. Later zou Frans als mijn promotor optreden.

Geen onbetrouwbare fysieke proefopstellingen meer: software als kortste weg tussen idee en realisatie.

De drie pijlers van software

Waar is het succes van software eigenlijk op gebaseerd? Het sleutelwoord is *automatisering* maar om de diepe implicaties daarvan te begrijpen, is enige nadere uitleg nodig.

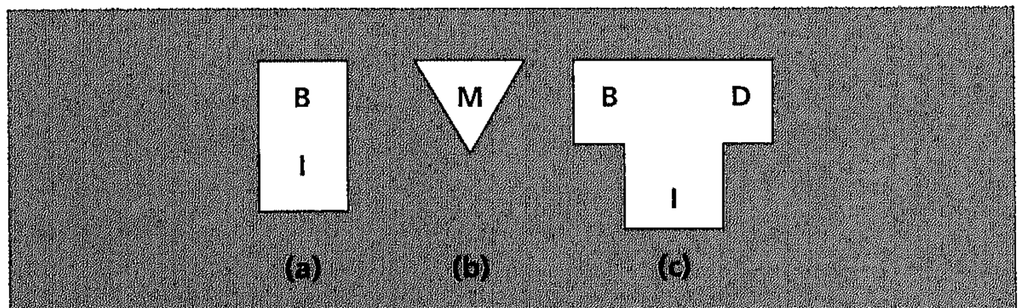
Computers zijn uitgevonden om handelingen die zich steeds herhalen te automatiseren. Een van de eerste voorbeelden daarvan is Jacquards weef-

(ponskaart) die door het weefgetouw wordt uitgelezen en omgezet in feitelijke weefstappen (kleur van de draad, bovenlangs of onderlangs, enz.). Zonder dat er nog een mensenhand aan te pas komt, is het resultaat stof met ingeweven patronen. Andere klassieke voorbeelden zijn de pianola en het draaiorgel die op ponskaarten vastgelegde muziek kunnen produceren. Al deze voorbeelden draaien om *recepten* die het te automatiseren proces beschrijven. Programma's zijn recepten die het door een computer uit te voeren proces beschrijven. Om programma's uit te drukken is een *(programmeer)taal* nodig. In bovenstaande voorbeelden zijn dit de gaten in de ponskaart, bij een computer zijn dit een stroom van éénen en nullen, die de computer opvat als instructies en uitvoert. Programmeertaal en programma's vormen de eerste pijler van software.

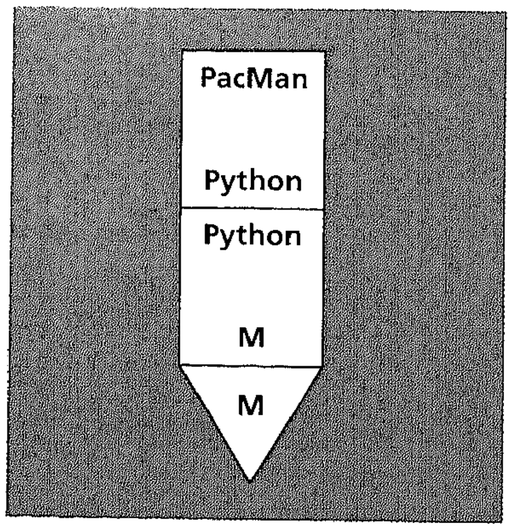
De andere twee pijlers beantwoorden de vraag hoe een programma uitgevoerd kan worden. Pijler twee is de *interpretator*: voer een gegeven program uit. Een interpretator leest een programma, voert het uit, en geeft een antwoord. In het geval van een draaiorgel is dit duidelijk: het draaiorgel interpreteert de codes op elke ponskaart en zet ze om in een toon. Voor computerprogramma's is uiteindelijk de computer de interpretator: lees het programma bestaande uit éénen en nullen en voer het uit. Probleem is echter dat het voor mensen erg moeizaam is om programma's van éénen en nullen te schrijven en daar komt de derde pijler ons te hulp, de *vertaler (compiler)*: vertaal een programma van de ene taal naar de andere. Het meest voor de hand liggende voorbeeld is vertaling van een voor programmeurs te begrijpen taal zoals C, Java of C# naar de éénen en nullen die een computer kan uitvoeren.

Het succes van software is gebaseerd op het feit dat interpretators en compilers zelf ook weer programma's zijn en als bouwstenen kunnen dienen voor steeds maar grotere bouwwerken. De bekende notatie van T-diagrammen kan ons daarbij helpen, zie Figuur 1: (a) representeert een interpretator voor brontaal B geschreven in implementatietaal I; (b) een in hardware geïmplementeerde interpretator voor machinetaal M; (c) een compiler van brontaal B naar doeltaal D geschreven in implementatietaal I. We kunnen nu enkele voorbeeldscenarios bekijken.

Figuur 1.
T-diagrammen



Figuur 2. Interpretator voor PacMan



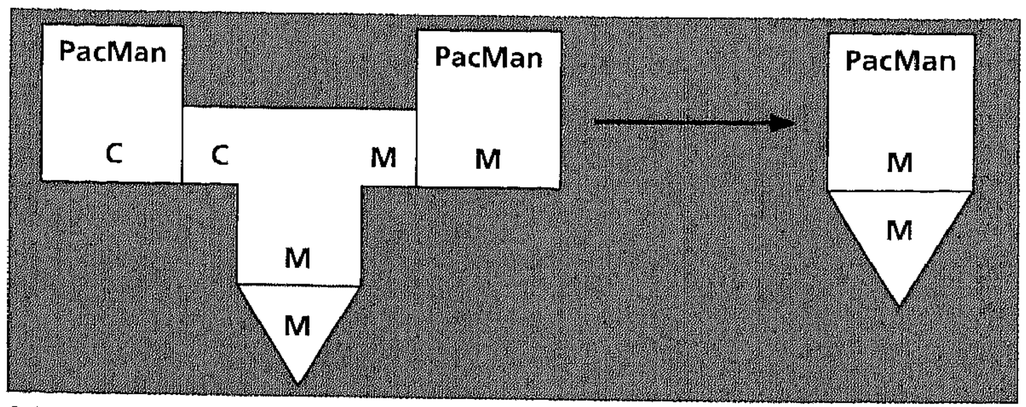
Het eerste voorbeeld is een programma voor het spel PacMan. Eerst bekijken we een versie die in Python geschreven is (Figuur 2). Het PacMan programma wordt direct uitgevoerd door de Python interpreter die zelf machine code als implementatie taal heeft en direct door de computer uitgevoerd kan worden.

Het tweede voorbeeld is een versie van PacMan geschreven in de programmeertaal C, zie Figuur 3. We beginnen met het PacMan programma en vertalen de broncode (C) naar machinecode (M). We gebruiken hiervoor een compiler die C naar machinecode vertaalt. Deze vertaler heeft als implementatietaal machinecode (M) en kan direct door de computer uitgevoerd worden. Het resultaat is een versie van het PacMan programma in machinecode die ook direct door de computer uitgevoerd kan worden.

Deze voorbeelden geven inzicht maar roepen direct ook vragen op. Bijvoorbeeld; is de C compiler zelf echt in machinecode geschreven? Het antwoord 'nee' is niet verrassend maar de details zijn dat wel degelijk. De C compiler is namelijk zelf in C geschreven! De details van deze vorm van *bootstrapping* zijn fascinerend maar zult u elders na moeten lezen[3]. Als we het hele gebied van de informatica bekijken dan worden deze technieken op grote schaal gebruikt in web-applicaties, databases, kunstmatige intelligentie, computational science, en in vele toepassingsgebieden.

De hele softwarerevolutie is te danken aan het vrijelijk stapelen en combineren van interpretators en compilers.

Figuur 3. Compiler voor PacMan



Meta-programmeren

Programma's krijgen een of meer invoerwaarden, voeren een berekening uit en produceren een antwoord. De invoerwaarden zijn meestal getallen of andere data. De interpretators en compilers die we zojuist zijn tegengekomen, zijn bijzonder: ze nemen andere programma's als invoerwaarde en produceren vaak ook weer een ander programma als resultaat. Denk aan de C compiler die een C programma als invoer heeft en een programma in machinetaal oplevert. Dergelijke programma's heten *meta-programma's*: programma's die andere programma's als invoer en/of uitvoer hebben. Gezien het belang van interpretators en compilers zijn technieken voor het maken van meta-programma's van groot belang.

In de loop der jaren heb ik samen met vele collega's, promovendi en studenten gewerkt aan het naar een hoger plan brengen van technieken voor meta-programmeren. Dit onderzoek heeft bijdragen geleverd op diverse gebieden, zoals:

- Talen en grammatica's.
- Parseren en parsergeneratie: GLR, SGLR, SGLL.
- Luie/incrementele programmageneratie.
- Pattern matching.
- Termherschrijven.
- Origin tracking.
- Backtracking.
- Coroutines.
- Coördinatietalen.
- Interactieve Ontwikkelomgevingen (IDEs).
- Continue integratie en distributie van software.
- Definiëren en extraheren van features.
- Domein-specifieke talen (DSLs).
- Program slicing.
- Analyse van software.
- Visualisatie.

Het onderzoek van fundamentele concepten moet, naar mijn smaak, altijd samengaan met het bouwen van prototypes die de werkzaamheid van de bestudeerde concepten aantonen. Nog beter is het als de prototypes zo goed zijn dat ze als open source software ook door derden gebruikt kunnen worden. Ik moet bekennen dat de eerste 'derden' daarbij vaak studenten zijn die mij dat niet altijd in dank hebben afgenomen. Daar heb ik natuurlijk begrip voor maar verwacht geen excuses; dit zijn kleine offers voor de voortgang van de wetenschap.

Bij het bestuderen van bovenstaande onderwerpen hebben we een aantal talen ontworpen en grote systemen gebouwd. De bekendste zijn:

- Het Algebraic Specification Formalism (ASF) is gebaseerd op axiomatische definities van datatypes. ASF is naar Prolog, Lisp en (op diverse manieren) naar C vertaald. De axioma's worden uitgevoerd als conditionele herschrijfregels. ASF heeft vele competities op het gebied van termherschrijven gewonnen en heeft een brug geslagen tussen de theorie van axiomatisch gespecificeerde abstracte datatypen en hun praktische toepassing.
- Het Syntax Definition Formalism (SDF) is gebaseerd op algemene contextvrije grammatica's. De implementatie bevat een aantal innovaties: gebruik van een algemene contextvrije parser en automatische constructie van syntaxbomen. Het eerste is van belang om de compositie van grammatica's mogelijk te maken, een noodzakelijke voorwaarde voor modulaire syntaxdefinities. Het tweede dient het gemak van de grammaticaschrijver. SDF is in diverse andere systemen gebruikt voor syntaxanalyse en heeft het idee van gegeneraliseerd parseren verder gepopulariseerd.
- De ASF+SDF Meta-Environment, een interactieve ontwikkelomgeving voor het formalisme ASF+SDF dat beide bovenstaande formalismen combineert. Unieke eigenschap is dat hiermee axioma's geschreven kunnen worden in een geheel door de gebruiker te bepalen notatie. De ASF+SDF Meta-Environment is gebruikt om een grote variatie aan systemen en talen te definiëren. Rond het jaar 2000 zijn er miljoenen regels COBOL mee geanalyseerd. In de loop van de tijd is de primaire implementatietaal verschoven van Lisp, via C naar Java. Het bouwen van een dergelijk systeem heeft mijn ogen geopend voor het probleem van *software evolutie*: software verandert en zo veel mogelijk ondersteuning daarbij is wenselijk, zo niet essentieel. Promovendi moeten proefschriften schrijven en worden niet afgerekend op het opleveren van onderhoudbare software. Overleven van de software van de promotor vereist daarom mechanismen om *phdware* om te zetten in onderhoudbare software en om mechanismen aan te bieden waarmee promovendi direct onderhoudbare software kunnen maken.
- De ToolBus, een op Proces Algebra[4] gebaseerd systeem om de samenwerking te coördineren tussen softwarecomponenten die in verschillende programmeertalen geschreven zijn. De ToolBus is uit nood geboren om de hierboven geschetste problemen van software-evolutie op te lossen. We zijn daarbij werkelijk in staat gebleken om systeemcomponenten bij herhaling door nieuwere technologieën te vervangen. Dankzij de

4 *J.A. Bergstra & J.W. Klop, Process Algebra for Synchronous Communication, Information and*

ToolBus is de ASF+SDF Meta-Environment bijvoorbeeld gecombineerd met gebruikersinterfaces geschreven in Tcl/Tk, Java Swing, en Eclipse SWT zonder enige wijziging in andere componenten.

- De Rascal Meta-Programming Language is ons meest recente systeem en verdient een aparte sectie.

Van ASF+SDF naar Rascal

Het is niet mijn bedoeling om hier een mini-college over de taal Rascal te geven. Daarvoor verwijs ik naar de website.[1] Wat ik wel ga doen, is beschrijven hoe deze taal logisch volgt uit eerdere ervaringen en experimenten. Als we de gehele ruimte van mogelijke ontwerpen van (meta-)programmeertalen bekijken, dan bevindt ASF+SDF zich op een uniek punt. De taal is gebaseerd op twee concepten: contextvrije grammatica's en termherschrijven. Deze concepten zijn naadloos met elkaar geïntegreerd en gebundeld in een modulair systeem waarin elke module grammaticaregels en herschrijfregels kan definiëren. Elke module kan daardoor gezien worden als een aparte subtaal die zijn eigen syntax en semantiek bijdraagt. Elke ASF+SDF specificatie bestaat altijd uit een compositie van subtalen. Voor eenvoudige talen is dit makkelijk voor te stellen: de subtaal van de Booleans, de natuurlijke getallen, enz. Wat mij altijd verrast en verheugd heeft is hoe schaalbaar deze aanpak is: van Booleans tot COBOL programma's. De wens om subtalen vrijelijk te kunnen combineren, heeft wel een prijs: hiervoor is het nodig om contextvrije grammatica's willekeurig met elkaar te kunnen combineren. Geen van de mainstream technieken voor parsergeneratie en parseren laat dit toe. Dit verklaart dat wij al snel gegeneraliseerde parseringstechnieken zijn gaan ontwikkelen die dit wel mogelijk maken. Ook deze oplossingen hebben hun prijs: ze zijn minder efficiënt en er is geen garantie meer dat zinnen zonder ambiguïteiten geparseerd kunnen worden. Daarom zijn er extra mechanismen voor disambiguering nodig zoals, bijvoorbeeld, definities van prioriteit, associativiteit, en lexicale restricties.

ASF+SDF was en is een unieke, krachtige taal, maar juist de elegantie en het kleine aantal basisconcepten vormde ook een beperking. Naarmate de toepassingen groter werden, bleken er steeds meer features te ontbreken. Bovendien had ASF+SDF geen makkelijk uit te breiden standaardbibliotheek om dit op te vangen. De doorslag gaf echter de waarneming dat ook studenten het steeds moeilijker vonden om de taal te leren. De verklaring hiervoor is dat ASF+SDF weliswaar op een klein aantal concepten is gebaseerd, maar dit zijn wel concepten die enige theoretische voorkennis vereisen en deze blijkt in de huidige curricula in onvoldoende mate onderwezen te worden. Het resultaat

was dat ASF+SDF steeds meer een alles-of-niets propositie werd: met de vereiste voorkennis was de taal goed te begrijpen en appreciëren, zonder die voorkennis zag men de taal meestal als hulpmiddel voor mystieke incantaties.

Aangezien we wisten dat de concepten waarop ASF+SDF gebaseerd was zeer geschikt zijn voor meta-programmeren besloot ik samen met Jurgen Vinju en Tijs van der Storm in 2008 een nieuwe taal te gaan ontwikkelen die het goede moest behouden en de gesignaleerde tekortkomingen moest adresseren.

Het resultaat is de taal Rascal die de volgende karakteristieken heeft:

- Een makkelijk te leren syntax die aanleunt tegen de Java syntax. Dit lost het probleem op dat beginners afgeschrikt worden door onbekende, vaak door de theorie geïnspireerde, notaties.
- Een groter aantal concepten die los van elkaar te leren en begrijpen zijn. Dit lost het probleem van de leerbaarheid op. De leerervaring voor ASF+SDF was, zoals gezegd, alles-of-niets, terwijl het leren van Rascal veel incrementeler gaat: een student hoeft een concept pas te begrijpen als hij/zij het nodig heeft.
- Een grote collectie ingebouwde datatypen met efficiënte implementaties, zoals bomen, lijsten, verzamelingen, relaties en tabellen. Daarnaast kan de gebruiker ook zelf nieuwe datatypen definiëren.
- Een aantal taalfeatures die essentieel zijn voor meta-programmeren:
  - Een subformalisme voor het definiëren van grammatica's en een bijbehorende op GLL-gebaseerde parsergenerator. Het resultaat van parsering is een syntaxboom die verder binnen Rascal verwerkt kan worden. Een Rascal programma kan hierdoor een grammatica voor een taal definiëren, files in die taal parseren en daarna verder bewerken (b.v. analyseren of vertalen).
  - Een subformalisme voor pattern matching: hiermee kunnen geavanceerde zoekoperaties op syntaxbomen, lijsten, verzamelingen en andere datatypen geprogrammeerd worden.
  - Een subformalisme voor het bezoeken van bomen en datatypen. Hiermee kunnen analyses of vertalingen geprogrammeerd worden.
- Een interface met Java waardoor het heel eenvoudig is om bestaande Java bibliotheken vanuit Rascal te gebruiken.
- Inbedding in de veel gebruikte Integrated Develoment Environment (IDE) Eclipse.

Inmiddels is er een bloeiend Rascal ecosysteem ontstaan en wordt de taal op allerlei manieren in onderwijs en onderzoek gebruikt die we bij het ontwerp

Toepassingen van meta-programmeren

Naast de al eerder beschreven toepassingen van meta-programmeren bij het maken van interpretators en compilers zijn er nog twee andere toepassingsgebieden die ik expliciet wil bespreken: software-evolutie en domein-specifieke talen.

Software-evolutie

Ik heb hierboven al aangegeven dat onze eigen, naar academische maatstaven omvangrijke, activiteiten op het gebied van software ontwikkeling mijn ogen hebben geopend voor het verschijnsel van software-evolutie. In de inleiding heb ik weliswaar stelling genomen tegen de slogan 'softwarecrisis', maar het valt niet te ontkennen dat er veel software is die voor verbetering vatbaar is. Jaren geleden heb ik eens berekend dat de totale hoeveelheid software op de wereld, afgedrukt in een 10 punts letter, een tekst oplevert waarmee de aarde 9 keer omwikkeld kan worden en dat die software per aardbewoner 5 fouten bevat. Inmiddels zullen deze getallen alleen maar hoger liggen en het is een formidabele uitdaging om deze bestaande berg software in bedrijf te houden en aan nieuwe eisen aan te passen.

Er bestaat rond dit onderwerp een enorme kloof tussen de positie van veel academische informatici en de praktijk. De academicus is geneigd te zeggen: 'ik ontwikkel prachtige nieuwe methodes die software flexibel en onderhoudbaar maken en als de praktijk mijn methodes gebruikt, dan is dit probleem daarmee opgelost.' Helaas leidt dit standpunt tot een *catch 22* van allure. Ik prefereer de positie om software en software-evolutie als een gegeven natuurverschijnsel[6] te beschouwen en als zodanig te bestuderen.

Het lag dan ook voor de hand om software-evolutie te gaan onderzoeken vanuit onze kennis en ervaring met meta-programmeren. Dit heeft geleid tot velerlei toepassingen:

- Analyse van grote softwaresystemen geschreven in o.a. COBOL, Java en PHP.
- Bepalen van metrieken voor onderhoudbaarheid en complexiteit van systemen in diverse talen.
- Transformatie en *refactoring* van software geschreven in COBOL en Java.
- Analyse van de versiehistorie van grote softwaresystemen.
- Visualisatiemethodes voor de resultaten van bovenstaande analyses.

Domein-specifieke talen

Eskimo's kennen dozijnen woorden voor sneeuw en ijs[7] en kunnen daardoor effectiever over hun omgeving communiceren. Zij gebruiken een vocabulair dat specifiek is voor hun leefomgeving. Het is met software niet anders: algemene werkpaarden als de talen C of Java zijn goed bruikbaar om heel uiteenlopende problemen op te lossen maar zijn omslachtig en woordenrijk bij het uitprogrammeren van specifieke problemen in een nauw omschreven gebied. Domein-specifieke talen[8] lossen dit probleem op door een aantal gespecialiseerde concepten aan te bieden die goed passen bij het probleemdomein. Een van onze eerste voorbeelden was Risla[9], een DSL om renteproducten mee te beschrijven. De beschrijving van een lening, hypotheek of deposito is gebaseerd op begrippen als 'hoofdsom', 'kasstroom' en 'aflossingsschema'. De taal Risla biedt deze concepten direct aan de programmeur aan en de ontwikkeltijd van de software om een renteproduct te ondersteunen gaat daarmee naar beneden van 6 maanden naar 4 weken. Na 18 jaar trouwe dienst en vele bankfusies later is Risla enkele maanden geleden door de ABN AMRO uit bedrijf genomen. Een opmerkelijke levensduur voor een systeem dat aanvankelijk als onderzoeksprototype ontstaan is.

Het gebruik van een DSL heeft diverse voordelen:

- Efficiëntie: de domein-specifieke concepten die in de DSL zijn ingebouwd kunnen steeds hergebruikt worden. Dit versnelt software ontwikkeling.
- Flexibiliteit: een DSL-programma moet door een DSL-compiler vertaald worden naar het uiteindelijke executieplatform. Dit heeft twee voordelen:
  - Alle DSL-programma's profiteren vanzelf van alle kwaliteitsaspecten van de compiler (correctheid, efficiëntie van de gegenereerde code). Verbeteringen in de compiler leiden direct tot verbetering van alle applicaties.
  - De kennis van het platform zit alleen in de compiler en een eventuele overstap naar een ander platform is relatief eenvoudig.

---

7 *T.E. Armstrong, B. Roberts & Ch. Swithinbank, Illustrated Glossary of Snow and Ice, Scott Polar Research Institute, Special Publication Number 4, 1969.*

8 *Ik gebruik verder de Engelstalige afkorting Domain Specific Language (DSL).*

9 *B.R.T. Arnold, A. van Deursen, and M. Res. An Algebraic Specification of a Language for Describing Financial Products. In M. Wirsing, editor, Proceedings of the ICSE-17 Workshop on Formal Methods Applications in Software Engineering Practice, pages 6-13, Seattle, April 1995.*

Natuurlijk heeft de DSL-benadering ook nadelen:

- Door de hoge graad van specialisatie heeft een DSL per definitie een kleine gemeenschap aan gebruikers. Dit heeft implicaties voor training en werving van personeel.
- De DSL-programma's zelf en de DSL-compiler moeten onderhouden worden.

We hebben in de loop der jaren een lange lijst aan DSLs geproduceerd:

- De al eerder genoemde formalismen ASF, SDF, ASF+SDF en Rascal zijn allemaal voorbeelden van DSLs op het gebied van meta-programmeren zelf.
- ToolBus scripts voor het coördineren van softwarecomponenten.
- In andere domeinen: Risla (rente gebaseerde producten), Derric (forensisch onderzoek), Machinations (game economieën), Pacioli (computational auditing).

De softwarerevolutie

Toenemende rekenkracht, mobiele en ingebouwde apparatuur, energiebesparing, beveiliging, sociale netwerken en steeds verdere penetratie van software in nieuwe toepassingsgebieden zullen de richting van de softwarerevolutie bepalen. Ik zie daarbij de volgende thema's voor toekomstig software-onderzoek dat de softwarerevolutie nog verder zal versnellen:

- Programmeertalen, interpretators en compilers zullen de motor blijven van de softwarerevolutie. Een fascinerende ontwikkeling is daarbij de versmelting van interpretators en compilers tot *tracing just-in-time* compilers. Executie van een programma start met een interpretator die tijdens executie meet welke lussen in het programma veel uitgevoerd worden ('hotspots'). Vervolgens worden deze lussen stap voor stap uitgerold voor specifieke gevallen en naar machinecode vertaald. De prestaties benaderen en evenaren soms de resultaten van klassieke compilatiemethodes. Deze methodes zullen ook weer een impuls geven aan het introduceren van nieuwe features in programmeertalen die tot voor kort als te inefficiënt afgedaan zijn.
- De huidige webapplicaties bestaan uit ad-hoc aan elkaar geknoopte componenten die in verschillende talen geschreven zijn en vrij arbitrair over (mobiele) client en server verdeeld zijn. Er is behoefte aan integrale programmeertalen en programmeeromgevingen die het mogelijk maken om alle aspecten van een webapplicatie op een uniforme manier te behandelen. Ik denk daarbij aan het transparant kunnen balanceren van taken tussen client en server en een *end-to-end* beveiliging.

- Met het toenemend belang van data-intensieve applicaties en van *big data* zie ik kansen om concepten uit software engineering en meta-programmeren toe te passen in talen en interactieve omgevingen voor statistiek en data-analyse (zoals R en MatLab). Ik denk daarbij aan het toepassen van geran-domiseerd testen om fouten op te sporen en het introduceren van *units of measurement* om de correcte dimensionaliteit van formules te garanderen.
- Naarmate de infrastructuur meer gedistribueerd en parallel wordt, kan de software niet achterblijven. Hoewel er decennia gewerkt is aan het begrijpen van concurrency en aan het parallelliseren van bestaande (niet-parallelle) software, liggen de grootste problemen nog voor ons. Er zijn nieuwe programmeerparadigma's en nieuwe programmeertalen nodig om juist deze aspecten te adresseren.
- Nu het energieverbruik in rekencentra en serverparken langzaam maar zeker onder controle begint te komen, verschuift de aandacht naar het energiegebruik van software applicaties. Dit geeft uitgelezen kansen voor compilers die het energieverbruik van de gegenereerde code minimaliseren.
- Hoe stel je vast of software correct is? Deze vraag heeft tot een brede kloof tussen academisch onderzoek en de softwarepraktijk geleid. Formele methoden om software correct te bewijzen hebben in het onderzoek veel aandacht gekregen maar blijken moeilijk schaalbaar en toepasbaar. In de praktijk wordt meestal alleen maar (handmatig!?) getest. Ik zie hier echter een voorzichtige convergentie. Enerzijds laat het succes van geautomati-seerd, random, testen (ook wel *property-based* testen genoemd) zien dat het specificeren en daarna testen van een zeer kleine deeleigenschap van een systeem praktisch zinvol is. Anderzijds zijn de technieken voor *model checking* en *constraint solving* inmiddels zo efficiënt dat ze op steeds grotere problemen toegepast kunnen worden. Dit leidt tot de verwachting dat voor steeds grotere deeleigenschappen van softwaresystemen specificatie en geautomatiseerd testen praktisch toepasbaar zullen worden.
- De softwareberg wordt steeds groter en het belang van het begrijpen en veranderen van bestaande software wordt daardoor alleen maar groter. Hier liggen grote uitdagingen voor onderzoek op het gebied van meta-programmeren. Steeds meer informatiebronnen zullen gebruikt moeten worden om inzicht in een softwaresysteem te krijgen, zoals (1) de bron-code om inherente eigenschappen van de code te bepalen; (2) de documen-tatie om extra informatie over de source code te verkrijgen en de consisten-tie tussen code en documentatie te bepalen; (3) het versiebeheersysteem om de ontwikkelgeschiedenis te bestuderen; (4) de *bug tracker* om alle gerapporteerde fouten te analyseren; (5) software repositories om het aantal downloads en installaties te meten; (6) gebruikersfora en emaillijsten

om het sentiment en de betrokkenheid van gebruikers te bepalen, (7) de financiële historie van de bedrijven die de (open source) software ondersteunen om hun langere termijn financiële stabiliteit in te schatten, en er zijn nog veel meer andere bronnen denkbaar. Dergelijk onderzoek is typisch multidisciplinair en omvat technieken uit meta-programmeren, analyse van natuurlijke taal, statistiek en machineleren.

- Lange tijd is gedacht dat grammatica's als onderzoeksonderwerp voltooid waren. Het gebruik van grammatica's bij meta-programmeren stelt echter volstrekt nieuwe eisen: (1) in plaats van een éénmalig te ontwikkelen artefact, wordt een grammatica ook onderwerp van evolutie; (2) grammatica's worden in steeds meer toepassingen gebruikt, denk aan het parseren van velerlei, zeer uiteenlopende, dataformaten; (3) grote grammatica's zijn complexe artefacten die voor mensen moeilijk te ontwerpen, te begrijpen of aan te passen zijn. Veel vragen uit onze onderzoeksagenda voor grammarware[10] uit 2005 zijn nog onbeantwoord. Mijn ideaal blijft een Integrated Grammar Workbench die alle aspecten van het ontwikkelen, testen, visualiseren en valideren van grammatica's ondersteunt.
- Ook heeft lange tijd het misverstand bestaan dat het onderzoek naar parsergeneratie en parseren af was. Het tegendeel blijkt waar. Daar zijn diverse redenen voor: (1) door de noodzaak tot renovatie moeten steeds meer legacy programmeertalen met exotische grammatica's geparseerd worden; (2) hetzelfde geldt voor grammatica's voor dataformaten; (3) nieuwe DSLs stellen steeds hogere eisen aan notatievrijheid die door de parser ondersteund moet worden; (4) software bestaat steeds vaker uit combinaties van talen en dit vergt de compositie van de bijbehorende grammatica's. Hernieuwde aandacht voor gegeneraliseerd topdown parseren biedt uitzicht op het maken van efficiënte, fouten corrigerende, incrementele en gemakkelijk te combineren parsers.
- De meest succesvolle programmeertalen zijn zelden talen die uit academisch onderzoek voortkomen. Het meest prominente voorbeeld is Javascript dat aanvankelijk in enkele weken door Netscape ontwikkeld is en nu de basis vormt voor elke webapplicatie. Als taalontwerp is Javascript geen hoogtepunt en dit ondersteunt mijn visie dat praktische ontwikkelingen in de informatica het beste als natuurverschijnsel benaderd kunnen worden.
- Tenslotte, bij het ontwikkelen van software spelen veel niet-technische vragen een rol zoals het voorspellen van de kwaliteit van individuele

10 *P. Klint, R. Lämmel, C. Verhoef, Towards an engineering discipline for grammarware,*

programmeurs, het samenstellen van teams om optimale samenwerking te bereiken, het vaststellen van de effectiviteit van verschillende methodes voor software ontwikkeling, het boven water halen van de wensen van een opdrachtgever, het invoeren van software in organisaties, en vele andere. Ik onderken en ondersteun het belang van deze onderwerpen nadrukkelijk, zowel in onderzoek en in onderwijs als in de praktijk. In de master Software Engineering besteden we er royaal aandacht aan. Mijn eigen belangstelling is meer technisch gericht en op mijn lijst van onderzoeksinteresses zijn deze onderwerpen altijd onderaan terecht gekomen. Zo ook nu weer.

Onderwijs en kennistransfer

Onderzoek, onderwijs en kennistransfer zijn in mijn beleving onverbrekelijk met elkaar verbonden. Aanvankelijk gebruikte ik daarbij een lineair innovatiemodel dat uitgaat van innovaties die in het onderzoek gedaan worden en daarna via onderwijs en industriële of maatschappelijke toepassing de maatschappij ten goede komen.[11] Later ben ik gaan begrijpen dat de terugkoppelingen tussen onderwijs en onderzoek en tussen toepassing en onderwijs en/of onderzoek veel interessanter zijn. Het getuigt van arrogantie te denken dat de goede vragen en vernieuwingen alleen van de academicus kunnen komen.

Het is bekend dat je geen academische carrière kunt bouwen op onderwijs. Dit komt omdat het academische beoordelingssysteem een lage prioriteit aan onderwijs geeft. Ik heb onderwijs altijd erg leuk gevonden en heb veel meer onderwijs gegeven dan overeenkwam met mijn deeltijdaanstelling. De afgelopen 10 jaar heb ik besteed aan het starten en steeds weer verbeteren van de master Software Engineering.[12] Deze is net gevisiteerd en op alle punten *goed* bevonden. Deze master probeert door het leggen van interactieve koppelingen tussen onderwijs en zowel onderzoek als bedrijfsleven mee te helpen aan het creëren van een steeds beter lopende innovatiepijplijn. De master biedt bewust een combinatie aan van vakken die gericht zijn op techniek:

- Software Testen en Specificeren,
- Software Evolutie,
- Software Constructie,

en op mens en organisatie:

- Software Architectuur,
- Software Requirements,
- Software Proces.

---

11 Helaas vormt dit verouderde model nog steeds de grondslag voor een groot deel van het innovatiebeleid.

Naast de master Software Engineering heb ik de laatste twee jaar met plezier het vak *Advanced Programming* gegeven aan het relatief nieuwe Amsterdam University College. Mij lijkt de brede bacheloropleiding die hier aangeboden wordt een belofte voor de toekomst. In het intensieve onderwijsmodel van het AUC heb ik geëxperimenteerd met *computational thinking* (door studenten te vragen via een Facebookgroep afbeeldingen en foto's te delen die de op college behandelde concepten uit de informatica illustreren) en met het naadloos integreren van hoor- en werkcolleges waardoor kennisoverdracht en kennistoepassing elkaar in korte cycli van een kwartier afwisselen.

Als laatste stap in de kennistransfer ben ik betrokken geweest bij het oprichten van drie spin-off bedrijven. De meest succesvolle daarvan, de Software Improvement Group[13], is inmiddels een bedrijf van 100 werknemers met klanten in binnen- en buitenland. De dienstverlening van de SIG richt zich op het monitoren van de kwaliteit en onderhoudbaarheid van softwaresystemen.

Personen

Onderzoek is mensenwerk en de resultaten die ik hier beschreven heb zijn het resultaat van de samenwerking met velen.

Allereerst wil ik de vele UvA-bestuurders uit de afgelopen periode bedanken voor het feit dat zij mij in de gelegenheid hebben gesteld om onderzoek en onderwijs op het gebied van de Software Engineering uit te voeren. Het zwaartepunt van mijn onderzoek lag daarbij bij mijn andere werkgever; het Centrum Wiskunde & Informatica; bij de Universiteit van Amsterdam lag de nadruk op onderwijs.

Ook wil ik diegenen bedanken waarmee ik in de loop der jaren in verschillende wetenschappelijke organisaties en verbanden heb mogen samenwerken, zowel nationaal (b.v. NWO, IPN, KNAW, IPA, Lorentz Centrum), als internationaal (b.v. INRIA, Royal Holloway, Schloss Dagstuhl, INRIA, EAPLS, ETAPS).

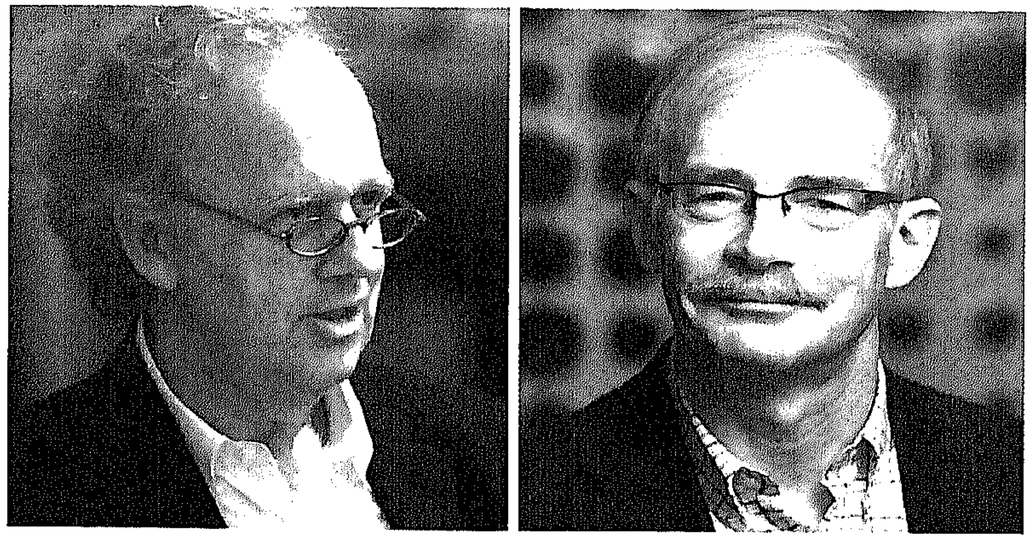
Naast mijn al eerder genoemde mentors Van Wijngaarden en Kruseman Aretz, hebben drie personen een doorslaggevende invloed gehad op mijn onderzoekscarrière. Ten eerste Jan Heering, met wie ik een lange en intensieve samenwerking heb gehad. Ik heb Jan ontmoet in een door Willem-Paul de Roever aan de Vrije Universiteit georganiseerd leesclubje om het net verschenen eerste deel van *The Art of Computer Programming* van D.E. Knuth te bestuderen.[14] Jan bestierde daar de computersystemen van het natuurkundig laboratorium en we hadden een gemeenschappelijke belangstelling voor SNOBOL, garbage collectie en vele andere thema's binnen en buiten de

---

13 Zie http://www.sig.eu/en/

Links: Jan Heering

Rechts: Jan Bergstra



informatica. We hebben lang en intensief samengewerkt aan een hele serie onderwerpen: software in het algemeen, eentalige programmeeromgevingen, algebraïsche specificaties, termherschrijven, incrementele programmageneratie en natuurlijk ASF+SDF. Jan was daarbij altijd gesprekspartner, criticaster en inspirator en door zijn brede belangstelling was elk gesprek voor mij een verrijking.

Ten tweede Jan Bergstra, die met zijn onconventionele zienswijzen altijd een interessante gesprekspartner is. Naast diepgaande discussies en samenwerking op het gebied van algebraïsche specificaties en termherschrijven, hebben we samen de ToolBus ontwikkeld, een incarnatie van Proces Algebra in een software tool. Een uitstapje naar softwarepatenten heeft ons allerlei verrijkende ervaringen gebracht buiten ons eigen vakgebied. Tenslotte wil ik niet onvermeld laten dat ik met Jan als *collega proximus* velerlei interne politieke verwikkelingen binnen de UvA heb meegemaakt. Ik zie met plezier dat Jan op dit moment met succes directeur van het Instituut voor Informatica is maar dat hij ook nog steeds eigen onderzoek verricht. Meer ondersteuning voor zijn onderzoek vanuit UvA en NWO zou mijns inziens wenselijk zijn.

Ten derde, de helaas te vroeg overleden Gilles Kahn. Begin jaren '80 van de vorige eeuw hebben we samen een tweetal ESPRIT projecten gedefinieerd en uitgevoerd die bepalend zijn geweest voor ons beider loopbaan. Gilles was een visionair en ik heb mijn herinneringen aan hem elders al beschreven[15].

Hier wil ik de oorspronkelijke doelstellingen van deze twee GIPE (Generation of Interactive Programming Environments) projecten citeren.

---

15 P. Klint, From Centaur to the Meta-Environment: a tribute to a great meta-technologist, in Y. Bertot, G. Huet, J.J. Lévi & G. Plotkin, From Semantics to Computer Science. Essays in Honour of Gilles

Gilles Kahn

U bent de meeste van deze onderwerpen al eerder in dit verhaal tegengekomen en ik heb samen met Jan Heering de GIPE-agenda dan ook steeds als leidraad voor ons onderzoek gebruikt:

The main objective of this project is to investigate the possibilities of automatically generating interactive programming environments from a language specification. An 'interactive programming environment' is here understood as a set of integrated tools for the incremental creation, manipulation, transformation and compilation of structured formalized objects such as programs in a programming language, specifications in a specification language, or formalized technical documents. Such an interactive environment will be generated from a complete syntactic and semantic characterization of the formal language to be used. In the proposed project, a prototype system will be designed and implemented that can manipulate large formally described objects (these descriptions may even use combinations of different formalisms), incrementally maintain their consistency, and compile these descriptions into executable programs.

The following steps are required to achieve this goal:

- *Construction of a shared software environment as a point of departure for experimenting with and making comparisons between language specific techniques. The necessary elements of this -- Unix-based – software environment are: efficient and mutually compatible implementations of Lisp and Prolog, a parser generator, general purpose algorithms for syntax-directed editing and prettyprinting, software packages for window management*

obtained; the main initial effort will be to integrate these components into one reliable, shared software environment.

- *A series of experiments that amount to developing sample specifications-based on different language specification formalisms, but initially based on inference rules and universal algebra--for a set of selected examples in the domain of programming languages, software engineering and man-machine interaction. The proposed formalisms have well-understood mathematical properties and can accommodate incremental and even reversible computing.*
- *Construction of a set of tools of the shared environment to carry out the above experiments. It will be necessary to create, manipulate and check (parts of) language specifications and to compile them into executable programs. The tools draw heavily upon techniques used in object-oriented programming (for manipulation of abstract syntax trees), automatic theorem proving (for inferring properties from given specifications to check their consistency and select potential compilation methods), expert systems (to organize the increasing number of facts that become known about a given specification) and Advanced Information Processing in general (man-machine interfaces, general inference techniques, maintenance and propagation of constraints, etc.).*
- *The above experiments will indicate which of the chosen formalisms is most appropriate for characterizing various aspects of programming languages and interactive programming environments. These insights will be used in constructing a prototype system for deriving programming environments from language specifications. The envisioned 'programming environment generator' consists of an integrated set of tools and an adequate man-machine interface for the incremental creation, consistency checking, manipulation and compilation of language specifications.*

Naaste deze voor mij erg belangrijke personen wil ik graag mijn voormalige promovendi bedanken, ze staan in een bijlage vermeld. Ik ben erg trots op de prestaties die zij hebben geleverd en op de diverse carrièrepaden die zij gevolgd hebben. Speciale vermelding verdienen de acht oud-promovendi die inmiddels zelf hoogleraar zijn geworden: Mark van den Brand (TU Eindhoven), Arie van Deursen (TU Delft), Anton Eliëns (TU Twente), Robert van Liere (TU Eindhoven), Frank Tip (University of Waterloo, Canada), Eelco Visser (TU Delft), Joost Visser (Radboud Universiteit), en Bruce Watson (University of Pretoria, South Africa).

Ik verheug me op de komende discussies en samenwerking met mijn huidige

Hans Dekkers bedank ik voor de sleutelrol die hij heeft gespeeld bij het tot een succes maken van de master Software Engineering.

Ook wil ik graag alle nog niet genoemde superieuren, collega's, co-auteurs, mede-ondernemers, studenten en ondersteuners bedanken waarmee ik in de loop der jaren heb mogen samenwerken.

En als allerlaatste, maar meest belangrijke, wil ik mijn vrouw Anneke en kinderen Eva en Thomas, mijn moeder Cleo, zus Astrid, en aanwezige familie en vrienden bedanken. Ik besef dat ik jullie door mijn passie voor mijn werk regelmatig tekort heb gedaan. Zonder jullie liefde en steun was dit alles niet mogelijk geweest.

Stellingen

Tot besluit leg ik u nog enkele stellingen voor:

1. Software is de kortste weg tussen idee en realisatie.
2. Software is een (te) onzichtbare innovatiemotor.
3. De kwaliteit van onderzoeksresultaten in een willekeurig wetenschapsgebied wordt mede bepaald door de kwaliteit van de software die in dat onderzoek gebruikt is.
4. De pijplijn van onderzoek via onderwijs naar industriële of maatschappelijke toepassing (en weer terug) is de sleutel tot innovatie en verdient meer aandacht van onderzoekers én beleidsmakers.
5. De ontwikkelingen in de informaticapraktijk kunnen beter als natuurverschijnsel bestudeerd worden dan als resultaat van intellectuele en/of academische activiteit.
6. Resultaten van academisch informaticaonderzoek dienen altijd vergezeld te gaan van een prototype dat de werkzaamheid van voorgestelde methodes en ideeën demonstreert.
7. De voorgaande stelling conflicteert met het vigerende dogma van *publish or perish*.
8. Een hoogleraar Software Engineering dient zelf te programmeren.
9. Promovendi worden afgerekend op het opleveren van een proefschrift en niet van onderhoudbare software. Een promotor heeft daarom beter gereedschap nodig om de *phdware* van zijn promovendi voor de langere termijn te conserveren en beschikbaar te stellen.
10. Het reproduceren en valideren van eerder gepubliceerde onderzoeksresultaten van andere onderzoekers krijgt te weinig aandacht in de informatica.
11. Het publiceren van negatieve resultaten kan collega-onderzoekers veel tijd besparen.
12. Het ontbreekt onderzoekers in de informatica vaak aan historisch besef, hierdoor worden vergelijkbare ideeën regelmatig (en vaak onder een andere naam) opnieuw uitgevonden. Dit helpt het vakgebied niet vooruit.
13. Abstract is vaak de vijand van concreet.
14. Bij het beoordelen van onderzoeksvoorstellen kan *peer review* beter vervangen worden door dobbelen.
15. De H-index wordt gebruikt om de impact van de publicaties van een onderzoeker te meten. De 'H' staat voor 'haast', de haast waarmee beoordelaars zich snel een mening willen vormen zonder zich in te spannen.
16. Onderwijskwaliteiten moeten een groter gewicht krijgen bij het beoordelen van de academische loopbaan.

17. De automatiseringsafdelingen van universiteiten zouden er goed aan doen om af en toe hun eigen experts op het gebied van software engineering te raadplegen.
18. Onderzoekers zijn niet te managen per spreadsheet.
19. Bezint eer Gij fuseert.
20. Net als stellingen bij proefschriften hebben stellingen bij afscheidsredes weinig zin.

Proefschriften met Paul Klint als eerste of tweede promotor

1. **V.J. de Jong**, *Conductor: a Multilingual Programming Environment for Statistical Software*, Universiteit van Groningen, 1988. Tweede Promotor: T.J. Wansbeek.
2. **A. Eliëns**, *DLP: A Language for Distributed Logic Programming*, Universiteit van Amsterdam, 1991. Tweede promotor: J.W. de Bakker.
3. **J.L.M. Vrancken**, *Studies in Process Algebra, Algebraic Specification and Parallellism*, Universiteit van Amsterdam, 1991. Eerste promotor: J.A. Bergstra.
4. **P.R.H. Hendriks**, *Implementation of Modular Algebraic Specifications*, Universiteit van Amsterdam, 1991.
5. **H.R. Walters**, *On Equal terms: implementing Algebraic Specifications*, Universiteit van Amsterdam, 1991.
6. **F. Wiedijk**, *Persistence in Algebraic Specification*, Universiteit van Amsterdam, 1991, Universiteit van Amsterdam. Tweede promotor: J.A. Bergstra.
7. **J. Rekers**, *Parser Generation for Interactive Environments*, Universiteit van Amsterdam, 1992.
8. **L. Helmink**, *Tools for Proofs and Programs*, Universiteit van Amsterdam, 1992, Eerste promotor: J.A. Bergstra, derde promotor: H.P. Barendregt.
9. **T.B. Dinesh**, *Object-oriented Programming: Inheritance to Adoption*, Universiteit van Iowa, 1992. Eerste promotor: A.C. Fleck.
10. **M.G.J. van den Brand**, *PREGMATIC: A Generator for Incremental Programming Environments*, Radboud Universiteit Nijmegen, 1992. Eerste promotor: C.H.A. Koster, Co-promotor: H. Meijer.
11. **E.A. van der Meulen**, *Incremental Rewriting*, Universiteit van Amsterdam, 1994.
12. **J. W. C. Koorn**, *Generating Uniform User-interfaces for Interactive Programming environments*, Universiteit van Amsterdam, 1994. Co-promotor, M. G. J. van den Brand.
13. **A. van Deursen**, *Executable Language Definitions*, Universiteit van Amsterdam, 1994.
14. **N.W.P. van Diepen**, *Modular Algebraic Specifications and Transformational Program Development*, Radboud Universiteit Nijmegen, 1994. Eerste promotor: H.A. Partch.
15. **F. Tip**, *Generation of Program Analysis Tools*, Universiteit van Amsterdam, 1995. Co-promotor: J.H. Field.
16. **B.W. Watson**, *Taxonomies and ToolKits of Regular Language*

17. **J. Kamperman**, *Compilation of Term Rewriting Systems*, Universiteit van Amsterdam, 1996. Co-promotor: H.R. Walters.
18. **E. Visser**, *Syntax Definition for Language Prototyping*, Universiteit van Amsterdam, 1997.
19. **S.M. Üsküdarli**, *Algebraic Specification of Visual Languages*, Universiteit van Amsterdam, 1997.
20. **E. Saaman**, *Another Formal Specification Language*, Universiteit van Groningen, 2000. Eerste promotor: G.R. Renardel de Lavalette.
21. **P.A. Olivier**, *A Framework for Debugging Heterogeneous Applications*, Universiteit van Amsterdam, 2000. Co-promotor: M.G.J. van den Brand.
22. **R. van Liere**, *Studies in Interactive Visualisation*, Universiteit van Amsterdam, 2001.
23. **T. Kuipers**, *Techniques for Understanding Legacy Software Systems.* Universiteit van Amsterdam, 2002. Co-promotor: A. van Deursen.
24. **L. Moonen**, *Exploring Software Systems*. Universiteit van Amsterdam, 2002. Co-promotor: A. van Deursen.
25. **J. Visser**, *Generic Traversal over Typed Source Code Representations.* Universiteit van Amsterdam, 2003. Co-promotor: R. Laemmel.
26. **M. de Jonge**, *To Reuse or To Be Reused*, Universiteit van Amsterdam, 2003. Co-promotor: A. van Deursen.
27. **J. Vinju**, *Analysis and Transformation of Source Code by Parsing and Rewriting*, Universiteit van Amsterdam, 2005. Co-promotor: M.G.J. van den Brand.
28. **H.A. de Jong**, *Flexible Heterogeneous Software Systems*, Universiteit van Amsterdam, 2007. Co-Promotor: M.G.J. van den Brand.
29. **S.R.L. Jansen**, *Customer Configuration Updating in a Software Supply Network*, Universiteit van Utrecht, 2007. Eerste Promotor: S. Brinkkemper.
30. **T. van der Storm**, *Component-based Configuration, Integration and Delivery*, Universiteit van Amsterdam, 2007. Tweede Promotor: S. Brinkkemper.
31. **M. Bruntink**, *Renovation of Idiomatic Crosscutting Concerns in Embedded Systems*, Technische Universiteit Delft, 2008. Eerste Promotor: A. van Deursen.
32. **Bas Basten**, *Ambiguity Detection for Programming Language Grammars*, Universiteit van Amsterdam, 2011. Co-promotor: J. Vinju.
33. **J. van den Bos**, *Gathering Evidence, Model-Driven Software Engineering in Automated Digital Forensics*, Universiteit van

Verantwoording illustraties

Pythagorasboom: Arze van der Ploeg
Tekening Angers: Public Domain
Portret Aad Van Wijngaarden: Centrum Wiskunde & Informatica
Portret Frans Kruseman Arerz: TU/e Technische Universiteit Eindhoven
Portret Jan Heering: Eelco Visser
Portret Jan Bergstra: Jan Willem Steenmeijer
Portret Gilles Kahn: Mark Stephane Goldberg