Modelgebaseerde verificatie en validatie loont

De meeste ingenieurs houden mathematische methodes liefst op grote afstand omdat die hun waarde nog niet voldoende hebben bewezen. Het afgelopen decennium is er echter spectaculaire vooruitgang geboekt op het gebied van formele technieken. In dit artikel geeft Frits Vaandrager van de Radboud Universiteit Nijmegen een overzicht van de belangrijke ontwikkelingen en gaat hij in op de vraag welke methodes geschikt zijn voor welke problemen.

Frits Vaandrager

Gérard Berry, *chief scientist* van het Franse bedrijf Esterel Technologies, heeft voorgesteld de term 'formele verificatie' te vervangen door 'automatische bugdetectie'. Hij wil hiermee benadrukken dat het (vooralsnog) niet haalbaar is om realistische IT-applicaties volledig correct te bewijzen met behulp van wiskunde. In theorie kan het wel, maar het kost gewoon te veel tijd en geld. Formele methodes moeten daarom tot doel hebben om ontwerpers te helpen zoveel mogelijk fouten zo vroeg mogelijk te vinden. Het blijkt dat formele technieken hier buitengewoon goed in zijn.

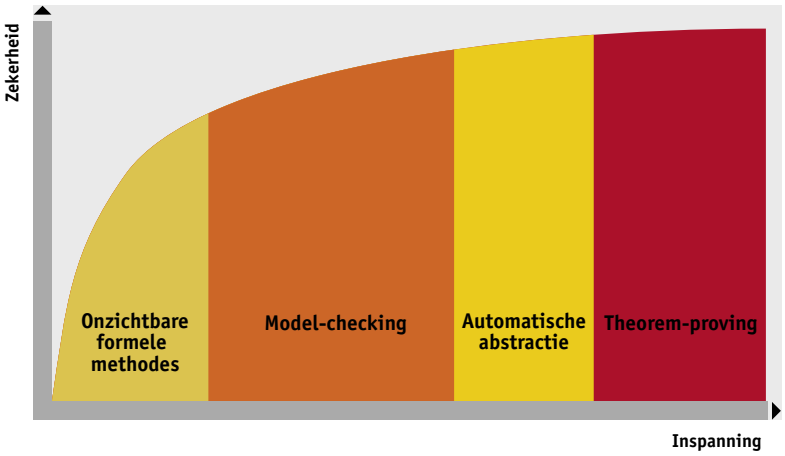
Waar ingenieurs traditioneel vooral gebruikmaken van continue wiskunde (analyse, differentiaalvergelijkingen), vergt het ontwerpen van betrouwbare computersystemen een meer discrete stijl van mathematisch redeneren. Het gedrag van computers modelleren we typisch in termen van eindige automaten met toestanden en discrete overgangen daartussen. Voor het beschrijven van en redeneren over automaten gebruiken we vaak (mathematische, formele) logica.

Het centrale probleem waar formele methodes een oplossing voor moeten bieden, is dat het aantal toestanden van een realistisch systeem al snel uit de hand loopt. Wanneer een programma (of hardwarecomponent) gebruikmaakt van honderd integervariabelen die ieder een waarde van 0 tot en met 9 mogen aannemen, dan is het aantal toestanden in potentie 10^{100}. Dit getal, in de literatuur aangeduid als 'googol' (Google heeft hier zijn naam aan ontleend), is ordes groter dan het aantal atomen in het universum. De meeste IT-applicaties hebben veel meer dan googol toestanden.

Veel verificatieproblemen laten zich vertalen naar de vraag of specifieke toestanden van een systeem bereikbaar zijn. Zijn er situaties waarin meerdere processen toegang hebben tot een kritieke sectie? Is bufferoverflow mogelijk? Kan het lezen van een register een waarde opleveren die nooit is geschreven? Het mag duidelijk zijn dat het voor het verifiëren van dergelijke eigenschappen geen optie is om alle bereikbare toestanden een voor een af te lopen.

Binnen de formele methodes gebruiken we daarom zogenaamde 'symbolische berekeningen'. Daarbij reken je over een heleboel toestanden tegelijk, net zoals je met de berekening $x^2-y^2 = (x-y)\cdot(x+y)$ in één keer laat zien dat $36-16 = 2\cdot 10$, $49-4 = 5\cdot 9$, enzovoorts. Door symbolisch te rekenen en toestandsverzamelingen compact te representeren met logische formules of slimme datastructuren, kunnen we immense toestandsruimtes efficiënt doorzoeken. De symbolische rekenregels die we gebruiken in formele methodes zijn het resultaat van decennialang onderzoek naar geautomatiseerd redeneren. Pas vrij recent zijn de barrières richting praktische toepasbaarheid echt geslecht.

In mijn overzicht van formele methodes ga ik uit van een diagram van John Rushby van het Stanford Research Institute (Figuur 1). Daarin staat de inspanning die vereist is om een techniek toe te passen, uitgezet tegen het extra vertrouwen in correctheid die het gebruik ervan oplevert. Het diagram laat zien dat onzichtbare formele methodes voor relatief weinig inspanning al veel resultaat opleveren. Met *model-checking*, automatische abstractie en *theorem-proving* is het vertrouwen in de correctheid van een systeem nog verder te vergroten, maar daarvoor moeten we wel veel investeren.



Figuur 1: John Rushby's spectrum van formele methodes

Interactieve stellingbewijzers

Een wiskundig bewijs is uiteindelijk niets anders dan een rijtje formules, opgeschreven in een welgedefinieerde logische taal, waarbij iedere formule ofwel een axioma is ofwel volgt uit voorgaande formules

op basis van een beperkt aantal bewijsregels. Voor een computer is het een koud kunstje om te checken of een rijtje formules een (correct) bewijs is. De Nederlander Dick de Bruijn was de eerste die een programma ontwikkelde waarmee wiskundige theorieën zijn te beschrijven en bewijzen mechanisch zijn te checken (Automath, *automath.webhop.net*). Sindsdien zijn er een groot aantal andere theorem-provers, of stellingbewijzers, gebouwd, waaronder ACL2, HOL, Isabelle, Nuprl en PVS.

Wanneer je als gebruiker alle stappen in een niet triviaal wiskundig bewijs moet intikken, dan is dit - gelet op het aantal stappen - buitensporig tijdrovend. Bovendien verlies je door alle details het overzicht. Moderne stellingbewijzers zijn daarom toegerust met een groot aantal slimme algoritmes en heuristieken waarmee ze, uitgaande van een bewijsschets, de details zelf invullen. Voor sommige deelproblemen (bijvoorbeeld propositielogica) kunnen ze zelfs volledig automatisch bewijzen construeren. Het gebruik van interactieve stellingbewijzers vereist veel ervaring, inzicht en tijd, maar je kunt er wel heel moeilijke problemen mee oplossen.

De meest prominente commerciële toepassing van interactieve stellingbewijzers is zonder twijfel die door Intel op het gebied van hardwareverificatie. In 1994 verloor de chipgigant 475 miljoen dollar door een fout met de drijvendekommadeling in de Pentium-processor. Sindsdien is het gebruik van formele methodes uitgegroeid tot een standaard praktijk in de hardware-industrie. Zo is bijvoorbeeld 20 procent van het Pentium IV-ontwerp formeel gecontroleerd. John Harrison en zijn Intel-collega's hebben alle drijvendekommaoperaties van de Itanium-processor geverifieerd met behulp van de stellingbewijzer HOL Light. Zoals verwacht, vonden ze diverse fouten. De verificatie leidde tot beter begrip van het probleem en uiteindelijk tot een beter ontwerp.

Nederland loopt internationaal voorop in het gebruik van theorem-provers. Zo is onder meer de PVS-stellingbewijzer gebruikt om de Splice-middleware van Thales (sinds kort Prismtech) te analyseren. Ook dit bracht een fout aan het licht. Vervolgens is er een oplossing aangedragen en is het uiteindelijke ontwerp verbeterd.

Ondanks deze succesverhalen is het gebruik van interactieve theorem-provers op dit moment nog niet lonend voor Nederlandse bedrijven, behalve in specifieke niches zoals de beschrijving en analyse van middleware en netwerken op chip. Toch gaat het hier wetenschappelijk gezien om een buitengewoon spannende ontwikkeling. Op termijn zullen stellingbewijzers een standaard gereedschap worden voor wiskundigen. In 2004 bereikten George Gonthier van Microsoft Research en Benjamin Wernier van Inria een doorbraak door het bewijs van de fameuze vierkleurenstelling te mechaniseren met behulp van de theorem-prover Coq.

Model-checking

Model-checking is een praktisch gereedschap voor het automatisch debuggen van complexe systemen, zoals netwerkprotocollen en regelaars. Hierbij modelleren we een systeem als een automaat en specificeren we het gewenste gedrag in termen van een (temporele) logica. Met een druk op de knop vertelt de model-checker vervolgens of de eigenschap al dan niet geldt. Zelfs extreem grote toestandsruimtes zijn vaak binnen enkele minuten te doorzoeken. Aanvankelijk verschenen er vooral model-checkers voor eindige automaten (MuCRL, SMV, Spin), maar recentelijk komen er ook krachtige varianten beschikbaar voor uitbreidingen met kansen (Prism) en realtime (Uppaal).

De twee grote voordelen van model-checking ten opzichte van theorem-proving zijn dat verificatie volledig automatisch is nadat je een model hebt gebouwd en de gewenste eigenschap hebt gespecificeerd, en dat het mogelijk is om tegenvoorbeelden te genereren, wat nuttig is bij het debuggen. De nadelen van model-checkers hebben vooral te maken met de beperkte expressiviteit van de modelleer- en specificatietalen die nodig zijn voor automatische verificatie. Verder moet je als gebruiker over de nodige handigheid beschikken om een model zo abstract te maken dat er geen toestandsexplosies optreden. Veel model-checkers zijn zeer gebruikersvriendelijk. Zo slagen 4vwo'ers er na een training van een half uur in om met Uppaal te werken en ze vinden dit nog 'cool' ook.

Er zijn talloze succesvolle toepassingen van model-checkers op industriële problemen. In de meeste gevallen gaat het daarbij om hardware, maar ook op het gebied van communicatie- en netwerkprotocollen zijn model-checkers zeer effectief. Nederlandse universiteiten hebben veel expertise op het gebied van model-checking. Die kennis hebben ze bijvoorbeeld aangewend om diverse protocollen van Philips te verifiëren (Havi, IEEE 1394, UPNP, Zeroconf) en de besturing van een liftsysteem te controleren dat Add-Controls heeft ontwikkeld voor het optillen van trucks.

Automatische abstractie

Een belangrijk probleem voor model-checking blijft schaalbaarheid. Wanneer we handmatig modellen maken, dan lukt het met enige moeite wel om ze zo abstract te krijgen dat een model-checker er doorheen komt. Het construeren van dergelijke modellen levert veel inzicht op (vaak vind je er meer fouten mee dan

met het model-checken zelf), maar is ook tijdrovend. Systeemontwikkelaars zouden model-checking het liefst direct loslaten op hun software of UML-modellen. Om dit mogelijk te maken, is er de afgelopen jaren veel onderzoek gedaan naar het automatisch construeren van abstracties. De computer kan bijvoorbeeld al uit zichzelf detecteren dat de precieze waarde van een integervariabele er niet toe doet en het voldoende is om te weten of die positief, negatief dan wel nul is.

Recentelijk zijn er op dit terrein een aantal doorbraken gerealiseerd. Er zijn nu gereedschappen die abstractie en model-checking combineren om broncode te analyseren (met name C en Java). Softwareverificatietools zoals Blast en Slam zijn succesvol toegepast om devicedrivers van Microsoft te debuggen. Dat zijn programma's met meer dan honderdduizend regels C-code. Zelfs Bill Gates was erg onder de indruk van het resultaat. In Nederland hebben we nog relatief weinig ervaring met softwarematige model-checking, maar in potentie is dit een buitengewoon veelbelovende techniek.

Onzichtbare formele methodes

Systeemontwikkelaars willen het liefst *push-button* verificatie. Ze moeten de technologie direct kunnen toepassen op hun software en UML-modellen. Ze vinden het best wanneer deze gebruikmaakt van allerlei moeilijke wiskunde, maar daar willen ze dan liefst zo min mogelijk van zien.

De notie van typering en de ontwikkeling van algoritmes om te bepalen of een programma typecorrect is, is een van de grote successen van het onderzoek naar formele methodes. De algoritmes en de achterliggende wiskunde zijn volledig onzichtbaar voor de programmeur, maar toch levert typering een grote bijdrage aan de verbetering van de kwaliteit van programma's.

Met de opkomst van modelgebaseerde ontwikkelmethodes ontstaan er allerlei mogelijkheden om analyses uit te voeren die verder gaan dan typecorrectheid. Rational Rose (Realtime), Rhapsody, Visualstate en andere commerciële tools ondersteunen de verificatie van functionele correctheidseigenschappen zoals de afwezigheid van deadlocks. Geheel verborgen voor de softwareontwikkelaar gebruiken ze hierbij soms zeer geavanceerde formele methodes.

Voor onzichtbare formele technieken is gebruikersgemak belangrijker dan algemeenheid. Ze vinden niet alle fouten in een ontwerp, maar wel de meeste. Bovendoen doen ze dat snel en geheel automatisch.

Een leuke voorbeeldtoepassing is Visualstate van IAR Systems, een grafisch gereedschap voor het ontwikkelen van embedded systemen. Deze tool gebruikt een buitengewoon geavanceerd verificatiealgoritme ('compositionele achterwaartse bereikbaarheidsanalyse') waarmee een standaard pc de toestandsruimte van grote industriële applicaties met meer dan duizend componenten in een paar minuten kan doorrekenen. Dit stelt ontwerpers bijvoorbeeld in staat om te verifiëren dat iedere toestand van iedere component bereikbaar is, alle constanten, variabelen en parameters worden gebruikt en er geen globale deadlock is.

Een ander voorbeeld is de Extended Static Checker voor Java (ESC/Java). Deze tool probeert met behulp van statische analyse runtime fouten te vinden in Java-programma's waarvan het gewenste gedrag is gespecificeerd met behulp van JML-annotaties. Omdat ESC/Java abstraheert van een groot aantal aspecten in Java, zal het gereedschap zeker niet alle fouten vinden. Het werkt echter volledig automatisch, waardoor het toch zeer effectief is. In de Nijmeegse groep van Bart Jacobs en Erik Poll is veel expertise voorhanden met betrekking tot de analyse van Java-programma's, onder meer met ESC/Java.

Modelgebaseerd testen

Modelgebaseerd testen (MBT) heeft als doel om automatisch testcases op te stellen, te executeren en te evalueren. Uitgangspunt is een model (meestal een automaat) van de te controleren programmatuur. In de meeste softwareprojecten verloopt het testen nog geheel handmatig. Er zijn gereedschappen die een deel van dat proces automatiseren, maar MBT heeft de ambitie om de testers al het werk uit handen te nemen.

Een van de voordelen die aanhangers noemen, is dat MBT meer en betere tests kan uitvoeren dan handmatig mogelijk is. In principe kan het de gehele functionaliteit van een systeem afdekken. Zo kan de methode testrijtjes genereren die een implementatie alle toestanden en transities van het model laten doorlopen. Andere pluspunten zijn dat testers sneller aan de slag kunnen, omdat alles is geautomatiseerd, en dat het goedkoper is, omdat ze grondiger kunnen testen met minder mensen en in minder tijd.

MBT is een uiterst belangrijke en interessante techniek, die uiteindelijk zijn weg zal vinden naar alle Model Driven Development-omgevingen. Op dit moment is echter nog niet overtuigend aangetoond dat de methode kosteneffectief is. Onderzoekers van Microsoft Research hebben een MBT-tool genaamd Specexplorer ontwikkeld (gratis beschikbaar via *research.microsoft.com/specexplorer*), die verschillende Redmondse productdivisies dagelijks gebruiken. In één specifiek geval leidde de toepassing ervan tot de ontdekking van tienmaal zoveel fouten als de tot dan toe gebruikte methode eruit haalde, waaronder veel 'diepe' fouten (die pas optreden na een heleboel stappen). Dit is veelbelovend, maar er zal nog veel onderzoek nodig zijn om vergelijkbare resultaten te bereiken voor embedded systemen.

Conclusies

Formele verificatie en validatie is een van de gereedschappen die we kunnen (en soms moeten) gebruiken bij de ontwikkeling van embedded systemen. In veel gevallen is toepassing ervan nog niet kosteneffectief, maar de situatie verandert in hoog tempo en steeds vaker loont het absoluut de moeite om kritieke componenten van een systeem formeel te verifiëren en te valideren. Nog steeds realiseren de meeste bedrijven zich niet hoe belangrijk het is voor hun eigen succes om expert te zijn op dat terrein.

Het toenemend gebruik van MDA, UML en specificatietalen als JML en OCL vormt potentieel een enorme stimulans (en in ieder geval een uitdaging) voor de toepassing van formele methodes. Zonder een model en/of een beschrijving van de gewenste eigenschappen valt er weinig te verifiëren en te valideren. Nederlandse universiteiten beschikken over uitmuntende expertise op het terrein van formele methodes en zijn altijd bereid om bedrijven te helpen bij het toepassen ervan, zowel direct als via een intermediair zoals het Laboratory for Quality Software (Laquso, *www.laquso.com*).

Frits Vaandrager is hoogleraar aan de Radboud Universiteit Nijmegen, waar hij leiding geeft aan de afdeling Informatica voor Technische Toepassingen. Tijdens de Bits&Chips Testdag op 4 oktober aanstaande geeft hij een lezing over model-checking.

Dit artikel is een verkorte versie van het whitepaper dat Vaandrager presenteerde tijdens de Progress-minisymposia in mei en juni. De volledige verhalen zijn gebundeld in de uitgave 'Progress white papers 2006'. Geïnteresseerden kunnen het boekje gratis aanvragen door een e-mailtje met afleveradres te sturen naar Dorien van der Maat (d.vandermaat@stw.nl).

Redactie Nieke Roos