Methodology

Het ontwikkelen van software gaat niet zonder fouten. Wie denkt ooit foutloze software te ontwikkelen, komt er snel achter dat dit een utopie is. Software systemen van enige omvang laten zien dat het gedrag dusdanig complexe vormen aanneemt, dat die door mensen niet overzien kan worden. Om toch enige grip uit te oefenen op de correctheid, worden vaak testen en simulaties uitgevoerd om zoveel mogelijk fouten op te sporen.

Grip op ontwikkelen van correcte software

Formele methoden zorgen voor overzicht

Jan Friso Groote
(j.f.groote@tue.nl)

Frank Stappers
(f.p.m.stappers@tue.nl)

Michel Reniers
(m.a.reniers@tue.nl)

Na een reis van 286 dagen in 1998, start de Mars Climate Orbiter zijn motoren om in een baan rond Mars te komen. Maar zodra de motoren zijn gestart, raast het ruimtevaartuig door de atmosfeer van de planeet, waarna het neerstort. De oorzaak is een fout in het metrieke stelsel van de software, waardoor een project van $125.000.000 ten einde komt. In 2000, vinden acht mensen de dood en raken 20 personen in een kritieke toestand, als gevolg van blootstelling aan een te hoge straling tijdens radiotherapie. De oorzaak hier was een dubbele dosis straling, als gevolg van de volgorde waarop data was ingevoerd, terwijl dit door software voorkomen had moeten worden. Zondagavond, 31 mei 2009, stort een Airbus 330-200 neer in de Atlantische Oceaan vlak bij de Braziliaans kust tijdens zwaar weer. Volgens deskundigen zou een onjuiste indicatie van de snelheidsmeters, in wisselwerking met de besturingssoftware, geleid hebben tot 228 doden.

Dit zijn slechts enkele voorbeelden, waarin door software miljoenen dollars, en zelfs mensenlevens verloren zijn gegaan. Een terechte vraag die we onszelf mogen stellen is: Is het mogelijk systemen foutvrij te ontwikkelen? Software systemen van enige omvang laten zien dat het gedrag dusdanig complexe vormen aanneemt, dat die door mensen niet overzien kan worden. Om toch enige grip uit te oefenen op de correctheid, worden vaak testen en simulaties uitgevoerd om zoveel mogelijk fouten op te sporen. Als we een analogie proberen te trekken met andere vakgebieden, bijvoorbeeld bouwkunde, dan zien we dat het testen van software overeenkomt met de inspectie van de bouw bij oplevering. Wat er in de bouw ook gebeurt, is dat een architect een model maakt en een constructeur dat model doorrekent op constructiefouten.

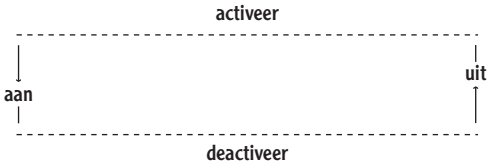
Bij de constructie van software is die eerste fase slecht ontwikkeld, met name het doorrekenen op constructiefouten. Daar betalen we gezamenlijk een hoge prijs voor, want het missen van fouten in de ontwerpfase is een kostbare aangelegenheid. Door herstel van laat ontdekte softwarefouten worden budgetten overschreden en is software onderhevig aan patches om “onvoorzien” gedrag te maskeren. Dit laatste tast de onderhoudbaarheid aan. We kennen al jaren methoden om software modellen te verkrijgen en te noteren. Bijvoorbeeld Yourdon’s “structured design method”, Hatley-Pirbhai modelleren en in toenemende mate UML. Het helpt om de software te structureren, maar helaas zijn deze methoden nog onvoldoende precies waardoor de interpretatiefouten blijven. Belangrijker nog is dat het niet mogelijk is om systematisch het ontwerp door te rekenen op correctheid. `Constructiefouten’ worden op deze manier nog steeds niet uitgesloten.

Er zijn verschillende academische onderzoeksgroepen in de wereld die werken aan doorrekenbare specificatiemethoden voor software. Het vakgebied waarin zij werkzaam zijn wordt aangeduid als de `formele methoden’. Een aantal van deze onderzoeksgroepen hebben hun methoden omgezet in talen en tools die meestal vrij beschikbaar zijn. Tools waar nu nog actief aan gewerkt wordt zijn

FDR, mCRL2, Uppaal, Caesar/aldebaran, muSMV en SPIN. Langzamerhand vinden de methoden ook hun weg naar commerciële aanbieders. Recente producten van Nederlandse bodem, zijn ASD van Verum en Trustware ontwikkeld door Imtech.

De theorie

Formele methoden zijn gebaseerd op een eenvoudig onderliggende gedachte. Het gedrag van een systeem is te zien als een automaat met toestanden en overgangen. Extreem versimpeld is een computer een apparaat met twee toestanden: `AAN' en `UIT'. Er zijn twee acties `activeer' en `deactiveer'. Het toestandsdiagram ziet er uit als:

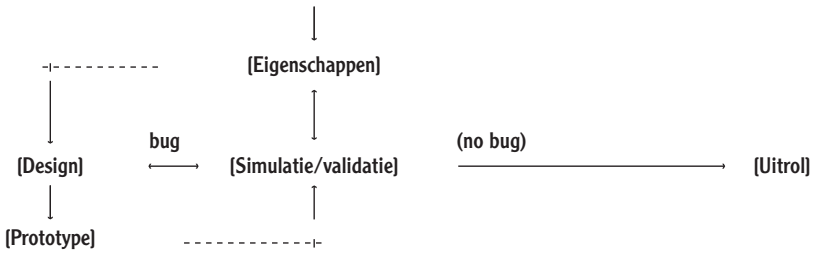


Een simpel apparaat met twee toestanden.

De werkelijkheid is veel complexer en bestaat uit heel veel van dergelijke gedragsautomaten die parallel staan, en elkaar beïnvloeden. Op een moderne computer kan de aan/uit schakelaar al beschouwd worden als een automaat die als die ingedrukt wordt nagaat of de computer feitelijk aan of uitgeschakeld is, en afhankelijk daarvan verschillende acties onderneemt. Een samenstel van dergelijke automaten is weer een gedragsautomaat. Het probleem is dat deze automaten extreem groot kunnen worden. Voor eenvoudige systemen is 10^1000 (10 tot de macht 1000) niet absurd groot.

Een moderne computer heeft veel meer dan 10^(10^10) toestanden. Dit zijn getallen die niet op te schrijven zijn. We spreken hier over informaticagetallen, om ze te contrasteren met de astronomische getallen die in verhouding verwaarloosbaar klein zijn. Het aantal atomen in het heelal is kleiner dan 10^100 (10 tot de macht 100). Lachwekkend klein gezien vanuit het perspectief van gedragsautomaten. Er zijn in essentie twee manieren om dergelijke automaten te analyseren op correctheid. De eerste is door het verifiëren dat het systeem aan zekere eigenschappen voldoet. Typische eigenschappen waar aan te denken valt zijn dat een systeem geen deadlock bevat, of dat een computer onder alle omstandigheden uit te zetten is. Als deze eisen complex worden, worden ze vaak geformuleerd in een zogenoemde modale logica. Populair zijn de logica's CTL, LTL en de modale mu-calculus. Het gaat hier te ver deze logica's in detail uit te leggen, maar belangrijk is te weten dat ze inmiddels zo krachtig zijn dat vrijwel iedere verifieerbare eigenschap kan worden uitgedrukt. Het verifiëren van een modale formule op de gedragsautomaat heet model checking. De tweede methode bestaat uit het versimpelen van de automaat door het toepassen van een gedragsreductiemethode. Een populaire methode is bijvoorbeeld branching bisimulatie reductie, maar er zijn nog vele anderen. Door eerst acties van een automaat te verbergen en die daarna te reduceren kan een automaat teruggebracht worden tot een zo'n klein aantal toestanden dat het resultaat met de hand kan worden geïnspecteerd. Op deze wijze wordt snel duidelijk of er ongewenst gedrag aanwezig is. Als eenmaal bewezen is dat de gespecificeerde gedragsautomaat alle gewenste eigenschappen heeft, kan die als uitgangspunt dienen voor implementatie van het systeem. Commerciële aanbieders hebben vrijwel altijd code generatoren waarmee automatisch code kan worden gegenereerd uit de formele specificatie. De formele specificatie kan ook worden gebruikt om automatisch testcases te genereren om te controleren of de implementatie wel voldoet aan de specificatie.

Formele methoden gaan uit van eenvoudige gedachtes.



De systeemontwikkeling, zoals deze er tegenwoordig meestal uitziet.

Het softwareontwikkelproces

Als we een schets maken van de huidige systeemontwikkeling, dan kan deze worden gekarakteriseerd als in bovenstaande afbeelding.:
Voordat men begint aan de ontwikkeling van een systeem, maken de verschillende stakeholders hun requirements bekend. Aan de hand van deze requirements worden er verschillende designs gemaakt. Deze designs bestaan vaak uit tekstuele beschrijvingen, systeem diagrammen en soms een eenvoudig programma wat het gedrag van het systeem simuleert. Om de kwaliteit van de documenten te waarborgen worden Peer-2-Peer reviews gehouden, wat vaak neerkomt op een oppervlakkige analyse van de structuur van het systeem. Om het gedrag beter te begrijpen worden deze designs vertaald naar eenvoudige opzichzelfstaande simulaties die het gedrag van het toekomstige systeem beogen te representeren. Deze vertaling komt vaak tot stand op basis van eerder opgedane ervaring en interpretatie van de ontwerper. Wanneer het design het systeem representeert, worden de simulaties uitgebreid met verschillende soorten software testen (unit tests, characterization tests, regressie tests, etc..). Parallel daaraan wordt vaak de ontwikkeling van hardware

Formele methoden worden al succesvol toegepast.

gestart. Is de hardware volwassen genoeg dan wordt deze geïntegreerd met de software, waardoor geleidelijk een eerste prototype ontstaat. Door middel van verschillende systeemtesten (integratie, stress, smoke, error, performance) worden diverse aspecten van het systeem geanalyseerd. Naar aanleiding van de simulatieresultaten komen vaak foute ontwerpen of conflicterende eigenschappen aan het licht. Door de software en hardware ontwikkelingen te itereren, worden deze vaak in volgende slagen verholpen. Wanneer het systeem voldoende foutvrij blijkt gaat deze in productie. Kiest men voor de integratie van formele methoden in het ontwikkelproces, dan moet men meer tijd spenderen aan de design fase. De noodzaak om fysieke tests uit te voeren zal daarbij afnemen, doordat men in de beginfase gedwongen wordt om beter na te denken over het design.

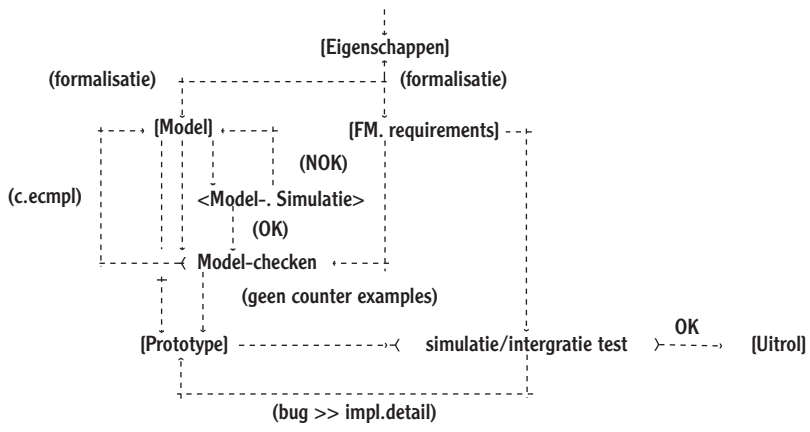
Formaliseren van requirements

De eerste stap is het formaliseren van requirements. Deze vormen de uitgangsbasis voor het te beschrijven model. Voor het beschrijven van functionele eigenschappen moet gedacht worden aan het opstellen van modale formules. Het opstellen van deze formules is niet eenvoudig, waardoor de designer gedwongen wordt om hier goed over na te denken. Het is niet ongebruikelijk dat bij verificatie van de requirements op het model, de requirements niet correct blijken.

Na het opstellen van de requirements kan een model van het gedrag worden gemaakt. Dit gaat overigens vaak tegelijkertijd met het opstellen van de requirements. Modellen kunnen buitengewoon omvangrijk worden. Tientallen pagina's is niet buitensporig.

Als het model beschikbaar is, kan het worden gebruikt om te toetsen of het model voldoet aan de verwachting van een designer en klant. Doordat formele modellen een eenduidig gedrag beschrijven, zijn de simulatoren vaak generiek en de modellen specifiek. In de praktijk blijkt dat de koppeling met grafische simulatoren in de richting van klanten uitstekend blijkt te werken, terwijl gedragsdiagrammen als communicatiemedium nog niet erg serieus worden genomen.

Bij de integratie van formele methoden in het ontwikkelproces, moet men meer tijd spenderen aan de design fase.



Zodra men er van overtuigd is dat het model een correcte abstractie van het beoogde systeem beschrijft, kan men het model verifiëren. In deze fase wordt getracht alle geformaliseerde requirements correct te bewijzen (door al het mogelijke gedrag te beschouwen). Indien een requirement niet geldt voor een model, wordt deze ontkracht door middel van een tegenvoorbeeld, wat betekent dat de designers terug moeten naar de tekentafel. Wordt er geen tegenvoorbeeld gevonden, dan is het beoogde systeem correct in termen van de opgestelde requirements.

Correct gedrag

Als aaneensluitend de implementatie van het prototype conform het gedrag is van het model, dan is daarmee bereikt dat het gedrag van het prototype ook correct is in termen van de opgestelde requirements. Om hierbij mankracht en bugs te minimaliseren kan het voordelen bieden om delen van de code te generen. Een uitrol kan nu plaatsvinden, maar het is natuurlijk wel verstandig om altijd tests uit te voeren. Het is namelijk mogelijk dat er discrepanties zijn opgetreden tussen het model en de implementatie. Echter de fouten die in deze fase optreden zijn, zijn vaak terug te voeren op fouten in de implementatie of niet beschreven gedrag in de requirements.

Als we de traditionele procesgang vergelijken met die van een formele systeemontwikkeling, dan biedt de laatste de volgende voordelen. Door validatie te verplaatsen van de prototype fase naar de design fase, worden de ontwikkelaars gedwongen de requirements eerder concreet te maken. Relatief ingewikkelde en moeilijk reproduceerbare bugs, als gevolg van racecondities tussen parallelle processen, worden in de design fase ontdekt. Dit leidt in de eindfase tot minder gedetecteerde problemen. Daardoor wordt er minder "gesleuteld" aan de software in de eindfase wat het process voorspelbaarder maakt en de software onderhoudbaarder. Helaas kent de formele systeemontwikkeling ook uitdagingen die essentieel zijn om het gehele proces te doen slagen.

Het formaliseren van modellen uit stakeholder requirements is een specialistische taak, waarbij kennis van het systeem, de in te zetten technieken, en de graad van abstractie essentieel zijn. In modellen treedt het probleem van zeer grote statespaces op. Een modelleur moet de juiste keuze weten te maken tussen een voldoende abstracte systeembeschrijving (met weinig toestanden) en

een beschrijving die voldoende met de werkelijkheid overeenkomt (maar te veel states heeft om alle eigenschappen te kunnen verifiëren). Het is ook een uitdaging om te werken aan (semi-)automatische technieken om een groot model terug te brengen tot een kleiner model dat nog voldoende informatie bevat om correctheid vast te stellen. Veel van het onderzoek in de formele methoden gaat over het ontwerpen van methoden om grote toestandsruimtes door te kunnen rekenen. Het inzetten van dergelijke methoden vergt een gedegen training.

De praktijk

Er zijn inmiddels vele voorbeelden waar formele gedragspecificaties succesvol zijn toegepast. Een mooi voorbeeld is de verificatie van de automatische parkeergarage, waarin wagens door middel van shuttles, liften en schachten worden gemanoeuvreerd naar hun parkeerpositie. Heeft men de auto weer nodig, dan wordt deze automatisch terug gemanoeuvreerd naar de uitgang.
Ondanks dat het systemen getest was, bleek er sprake te zijn van autoschade. Na een grondige (niet formele) analyse werd het veroorzaken van schade in verband gebracht met de safety afhandeling in de control software. Om verdere schade te voorkomen, werd ervoor gekozen om het design van de safety controller met behulp van formele verificatietechnieken te herontwikkelen. Hierbij werd de bestaande software als leidraad genomen. Om de analyse te ondersteunen werd daarbij een op maat gesneden visualisatie tool ontwikkeld. Door beide technieken te combineren, bleek men in staat een correct bewezen aansturingsysteem voor de parkeergarage te ontwerpen.

Een recentere toepassing is te vinden in het grootste sterrenobservatorium van astronomische organisaties binnen Europa, Noord-Amerika en Japan, genaamd Atacama Large Millimeter Array (ALMA).

Een voorbeeld van succesvol toegepaste formele gedragsspecificaties is de verificatie van een automatische parkeergarage, waarin wagens door middel van shuttles, liften en schachten worden gemanoeuvreerd naar hun parkeerpositie.

Formele methoden bewijzen vooral hun nut bij constructies, die niet mogen falen.

Bij oplevering zal dit systeem bestaan uit meer dan 50 telescopen, die opererend in millimeter marges. In deze applicatie is er gekeken naar de op CORBA gebaseerde software architectuur. Dit raamwerk biedt het ALMA software systeem een collectie van componenten en services waarop delen van het systeem vertrouwen. Met behulp van de specificatie taal en de bijbehorende modelchecker van mCRL2 is er ingezoomd op de controller, welke verantwoordelijk is voor het opstarten van componenten en geassocieerde containers. Door te praten met verschillende ontwikkelaars, het doorlezen van documentatie en code analyse is er een model gemaakt, waarna de analyse zicht richtte op potentiële deadlocks. Tijdens de analyse is gebleken dat time-outs van essentieel belang zijn voor een deadlock vrij systeem. Verder bleek dat het geanalyseerde systeem correct werkte.

Een andere noemenswaardige casus is de verificatie van een prototype printer voor het fabriceren van printed circuit boards. Met de huidige fabricage van PCBs wordt veelal gebruikt gemaakt van masks, die door middel van een lithografisch proces op het substraat worden afgebeeld. Met behulp van een PCB-printer kan de dure productie van het opstellen van de mask worden overgeslagen en kan ieder gewenst patroon direct op een substraat worden geprint. Om ervoor te zorgen dat het systeem geen onvoorspelbaar gedrag vertoond, werd het gedrag van de controller formeel geverifieerd. De controller bestond uit 245 multi-threaded taken geïmplementeerd in C#. In totaal besloeg de controller (zonder het onderliggende concurrent framework) 170.000 regels code. Door te abstraheren van interne variabelen, verkregen we een model dat alleen de communicatie tussen componenten behield. Doordat de requirements in termen van deze communicaties waren opgesteld, waren we in staat om de safety eigenschappen van het systeem te verifiëren. «

Het grootste sterrenobservatorium in Europa, Japan en Noord-Amerika is een voorbeeld van een recente toepassing.