Centrum voor Wiskunde en Informatica
Centre for Mathematics and Computer Science

G.J. Hofman, J.C. van Vliet

Over certificatie en tekstverwerking

Afdeling Informatica Notitie CS-N8506 Juli

On certification and text processing

The Centre for Mathematics and Computer Science is a research institute of the Stichting Mathematisch Centrum, which was founded on February 11, 1946, as a nonprofit institution aiming at the promotion of mathematics, computer science, and their applications. It is sponsored by the Dutch Government through the Netherlands Organization for the Advancement of Pure Research (Z.W.O.).

Over certificatie en tekstverwerking

G.J. Hofman[1] & J.C. van Vliet
Centre for Mathematics and Computer Science
P.O. Box 4079, 1009 AB Amsterdam, The Netherlands

Hoe kunnen functionele aspecten van tekstverwerkers gecertificeerd worden? Het blijkt dat bestaande testtechnieken hiervoor veelal ongeschikt zijn. In dit artikel geven wij een schets van een alternatieve testmethode en illustreren aan de hand van een case study dat er voor certificatie van een gegeven type software zowel een forse investering in tijd als de nodige kennis van het betreffende probleemgebied nodig is.

1983 CR Categories: K.7.3, I.7.2
Noot: dit artikel is elders ter publicatie aangeboden.

Bij de aanschaf van een nieuwe wasmachine let men vaak op de aanwezigheid van een KEMA-keur, aangezien dit een waarborg levert wat betreft het voldoen aan bepaalde kwaliteitsaspecten. Men kan zich afvragen of iets dergelijks ook voor software mogelijk is. Stel er is een instantie die softwarepakketten voor financiële administratie onderzoekt op aspecten betreffende de bescherming van gegevens. Gegeven normen op dit gebied kan aan pakketten die hieraan voldoen het GEGOB-keur (GEgevens GOed Beschermd) worden verleend. Men zou dan verwachten dat bij de selectie van een dergelijk pakket de aanwezigheid van het GEGOB-keur mede een rol zal spelen.

Er valt een stijgende belangstelling te bespeuren voor diverse vormen van certificatie van software. Hiervoor zijn diverse redenen aan te wijzen, waaronder:

- software wordt steeds vaker ingezet bij toepassingen die op enigerlei wijze kritisch zijn; men wil graag dat een door software bestuurde robot geen fysiek gevaar voor de in de buurt zijnde personen oplevert (in de VS is, naar aanleiding van een dodelijk ongeluk, de eerste rechtzaak op dit gebied al gevoerd);
- steeds meer niet-professionals maken gebruik van computers en programmatuur. Deze personen zijn in het algemeen niet in staat de door hen aan te schaffen hard- en software op hun merites te beoordelen. Er ontstaat dan behoefte aan een onpartijdige instantie die dit soort kwaliteitsbeoordeling voor hen verricht.
- de onderhoudskosten van programmatuur stijgen de pan uit. In veel organisaties is dit een probleem, en men wil zich graag reeds bij aanschaf een oordeel vormen over de kwaliteit van bijvoorbeeld de technische documentatie, of de gebruikte ontwerpmethodiek. (Hier valt een algemene tendens te bespeuren: in toenemende mate wordt aan kwaliteit de voorkeur gegeven boven lage kosten.)

Er is, internationaal, reeds enige ervaring op het gebied van certificatie van software. Dit betreft voornamelijk certificatie van compilers (Ada[2], Pascal, Minimal Basic, e.d.) en communicatieprotocollen [1]. Certificatie van compilers betekent in dit verband: testen of de compiler precies die taal

1. Stagiair IHBO Eindhoven
2. Ada is een geregistreerd handelsmerk van het US Department of Defense

accepteert welke in de standaard (van bijvoorbeeld ISO) is vastgelegd, en niet een ingeperkte of uitgebreide versie hiervan. Een hieruit resulterend certificaat zegt dus niets over:
- de correctheid van de geleverde code;
- de efficiency van de geleverde code;
- de kwaliteit van door de compiler geleverde foutmeldingen;
- enz.

De ervaring leert dat aanwezigheid van een certificaat bij een compiler vaak geïnterpreteerd wordt als "dit is een goede compiler". Deze conclusie is echter misplaatst. Op dezelfde wijze zullen veel mensen meer in het KEMA-keur lezen dan gerechtvaardigd is. Een KEMA-keur zegt enkel iets over de electrische veiligheid. In theorie kan een wasmachine waarvan de trommel niet draait toch het KEMA-keur krijgen[2].

Certificatie is een zaak die niet alleen Nederland aangaat. In EEG-verband zijn dan ook de nodige activiteiten op dit gebeid ontplooid [1, 3]. [1] bevat informatie over bestaande en potentiële certificatiemogelijkheden in de verschillende EEG-landen. Men voorziet in dit rapport twee mogelijke wegen:
- certificatie wordt gestuurd door regelgeving van de EEG of nationale regeringen;
- 'vrijwillige' certificatie ten opzichte van internationale standaards, zoals die van ISO.

Laatstgenoemde mogelijkheid wordt momenteel veelal toegepast. Deze route levert minder juridische problemen op, en lijkt beter aan te sluiten bij de wensen van de betrokkenen. Het rapport spreekt dan ook een voorkeur uit voor deze aanpak.

CEN (Comité Européen de Normalisation — het overkoepelend orgaan van nationale normalisatie-organisaties in Europa) heeft prioriteit gegeven aan standaardisatie op het gebied van informatietechnologie. In het verlengde hiervan worden door CENCER (de certificatietak van CEN) de mogelijkheden onderzocht om tot certificatie op Europees niveau te komen. Huidige voorstellen gaan in de richting van:
1) Er komen een aantal keuringsbureaus, die aan nog nader te bepalen eisen moeten voldoen;
2) Certificaten worden door CENCER verleend, nadat keuring door een bureau heeft plaatsgevonden;
3) Een klant die een bepaald certificaat voor zijn product wenst heeft de vrijheid voor wat betreft de keuze van een keuringsbureau;
4) Er worden afspraken gemaakt voor wat betreft prijsstelling en keuringsprocedures, teneinde een zo groot mogelijke uniformiteit tussen de verschillende bureaus te bewerkstelligen.

Ook in Nederland worden de nodige activiteiten op dit gebied ontwikkeld. Er is een Initiatiefgroep Software-Certificatie gevormd met als doel te komen tot de uitvoering van een kwaliteitsbeoordeling voor programmatuur [4]. Hierbij worden twee wegen bewandeld: certificatie van programmatuurpakketten (pakketcertificatie) en certificatie van het kwaliteitssysteem van bedrijven die programmatuur leveren (bedrijfscertificatie). De sectie EDP Auditing van het NGI heeft in 1984 een symposium over Certificering van Software georganiseerd. Enkele bijdragen hieruit zijn eerder in Informatie verschenen [5, 6].

Er is door het CWI in samenwerking met IWIS-TNO een voorstudie verricht naar mogelijkheden om te komen tot certificatie van pakketsoftware. De opdracht hiertoe is verleend door de Directie Overheidsorganisatie en -Automatisering van het Ministerie van Binnenlandse Zaken.

Onder pakketsoftware wordt hier verstaan: algemeen verbreide software die aan nog onvoldoende gestandaardiseerde eisen moet voldoen. Voorbeelden hiervan zijn: een financieel/administratief database pakket, een statistisch of numeriek pakket, programmatuur voor realtime besturing, een tekstverwerker.

Ter ondersteuning van de eerdergenoemde voorstudie is een case study verricht naar certificatie van tekstverwerkers. Over deze case study wordt hier verslag uitgebracht. In sectie 1 wordt kort ingegaan op functionele aspecten van tekstverwerkers die voor certificatie in aanmerking komen. Onder functionele aspecten verstaan we die aspecten welke essentieel zijn voor de functie van de programmatuur. Niet-functionele aspecten, zoals de efficiency en de kwaliteit van het mens-machine interface worden,

hoe belangrijk deze ook mogen zijn, hier niet in beschouwing genomen.

Een verder uitgangspunt bij de case study is geweest dat de certificerende instantie **niet** de beschikking heeft over de feitelijke programmacode. Certificatie moet dan plaats vinden op basis van tests die gebaseerd zijn op de gebruikersdocumentatie en vergelijkbaar materiaal. Op grond van de resultaten van deze tests vormt men zich dan een oordeel over de functionaliteit van de betreffende tekstverwerker.

In sectie 2 beschouwen we bestaande testtechnieken en hun toepasbaarheid bij de door ons beschouwde vormen van certificatie. In sectie 3 wordt een nieuwe testtechniek ontwikkeld en op een voorbeeld toegepast. Tenslotte geven we enkele conclusies die uit deze case study getrokken kunnen worden.

1. Over Tekstverwerking

Het is niet onze intentie op deze plaats een uitgebreide verhandeling te geven over het automatisch verwerken van documenten. De geïnteresseerde lezer wordt hiervoor verwezen naar bij voorbeeld [7] of [8]. We zullen ons hier beperken tot het noemen van enkele aspecten die voor de verdere discussie van belang zijn.

We onderscheiden bij de electronische verwerking van documenten voor het gemak twee fasen: een tekst wordt aangemaakt met behulp van een editor, en vervolgens geformatteerd. Tijdens het aanmaken wordt de eigenlijke tekst ingevoerd, en worden tevens opmaakcommando's tussengevoegd die het formatteerproces sturen. Dit zijn commando's als:

- ga over op een bepaald font (lettertype);
- begin een nieuw hoofdstuk;
- ga over op een nieuwe pagina.

Bij een batchgeoriënteerd systeem is het onderscheid tussen deze fasen duidelijk, bij een interactief systeem zijn ze tot op zekere hoogte geïntegreerd.

Belangrijke aspecten die bij certificatie van tekstverwerkers een rol kunnen spelen, zijn:

- Vorm en type van opmaakcommando's. We kunnen een onderscheid maken in structuurcommando's en layoutcommando's. Bij structuurcommando's (ook wel logisch, abstract, high-level of declarative genoemd) wordt het document gezien als een gestructureerde hoeveelheid gegevens, en deze commando's geven de koppeling tussen de tekst en de structuur aan. Typische voorbeelden van structuurcommando's zijn: nu begint een nieuw hoofdstuk, hier begint een voetnoot.
  Layoutcommando's (ook wel fysiek, concreet, low-level of procedural genoemd) geven harde wensen met betrekking tot de layout aan. Voorbeelden zijn: ga over op een nieuwe pagina, ga over of font X, enz.
  Voor beide typen commando's is het onder andere van belang te weten of de gebruiker zelf nieuwe commando's kan definiëren, of hij deze kan parametriseren, en, zo ja, hoe.
- Het bereik van commando's. Bij het verwerken van teksten speelt een aantal attributen een rol, zoals het font waarin de tekst wordt afgedrukt, de regellengte, enz. Op elk gegeven punt in de tekst kan men de verzameling waarden van deze attributen opvatten als de omgeving waarin de tekst wordt verwerkt. Verandering van de waarde van een of meer attributen komt dan overeen met een overgang naar een nieuwe omgeving. Hierbij spelen vragen een rol als:
  - moet de nieuwe omgeving geheel gespecificeerd worden, of hoeven alleen de afwijkingen opgegeven te worden;
  - heeft de instelling van een nieuwe waarde een lokale betekenis, of is deze globaal;
  - kan aan een omgeving een naam gegeven worden, zodat deze meermalen gebruikt kan worden.
- Layoutkwesties. Hieronder vallen zaken welke samenhangen met de uiteindelijke verschijning van de tekst op papier. Men kan hier denken aan:

- Is er een algoritme voor het afbreken van woorden. Hoe kan de gebruiker hier invloed op uitoefenen (bij voorbeeld door zelf aan te geven waar een woord afgebroken mag worden, of via uitzonderingslijsten).
- Hoe wordt de tekst over regels verdeeld. Wordt dit per regel bepaald, of is er een algoritme welke een gehele alinea of pagina beschouwt. Wordt er extra moeite gedaan om nette pagina-overgangen te creëren.
- Is het mogelijk tekst op te nemen welke elders in het document geplaatst moet worden (voetnoten, referenties, het aangeven van termen die in een index opgenomen moeten worden, kopregels bovenaan een pagina).
- Is het mogelijk 'drijvende' tekstdelen op te nemen, zoals plaatjes die niet over een paginagrens mogen vallen.
- Welke mogelijkheden zijn er om de paginalayout te beïnvloeden (hoogte en breedte van pagina's, centreren van tekst, meerkoloms uitvoer, enz).
- Welke tekens kan men gebruiken, en in welke grootte (verschillende fonts als Times Roman, Baskerville, exotische fonts als Arabisch, Devanagari, Kanji, diacritische tekens, speciale fonts voor bij voorbeeld wiskundige formules).

- Hoe is de filestructuur. Kan men bij voorbeeld de tekst over meerdere files verdelen, en hoe zit het dan met de geldigheid van definities, kunnen referenties in een aparte file (databank) opgeslagen worden.
- Overdraagbaarheid. Een belangrijke vraag hier is of het systeem in staat is uitvoer voor verschillende apparaten te leveren, bij voorbeeld een eenvoudige laserprinter tijdens de proeffase, en een fotozetmachine voor de uiteindelijke versie. Hieraan gerelateerd is de vraag of de invoertekst door verschillende systemen verwerkt kan worden, waarbij bij voorbeeld een eenvoudig systeem commando's negeert die alleen in een uitgebreide versie aanwezig zijn.
- Niet-tekst componenten. Hierbij treden natuurlijk voor een deel dezelfde problemen op die ook bij platte tekst een rol spelen; ook binnen formules moeten elementen op de juiste wijze ten opzichte van elkaar gepositioneerd worden. Speciale vragen die een rol spelen, zijn:
  - Bij tekeningen: is het mogelijk rechtstreeks op het scherm te tekenen, kunnen tekeningen in woorden omschreven worden, in hoeverre kan gebruik gemaakt worden van standaardvormen als cirkels, rechthoeken, pijlen, is een eenmaal gedefinieerde figuur meermalen (bij voorbeeld als component in een andere figuur) te gebruiken.
  - Bij tabellen: is het mogelijk tekst op verschillende wijze in kolommen te plaatsen (gecentreerd, getallen met de decimale punt onder elkaar), in hoeverre kan de tabel aangekleed worden met horizontale en verticale afscheidingen, is het systeem in staat zelf tekst over verschillende regels te verdelen, afhankelijk van de breedte van de diverse kolommen.
  - Bij formules: is het systeem in staat zelf de grootte van bepaalde symbolen (zoals som- en integraalteken) te bepalen, wat is het beschikbare tekenrepertoire.

  In meer specifieke omstandigheden kan men de behoefte hebben aan de mogelijkheid om ook andere twee-dimensionale zaken af te beelden, zoals scheikundige formules of muziekschrift. Er zijn ook combinaties mogelijk, zoals tabellen met formules.
- Eigenschappen van de editor. Bij niet-interactieve systemen voor documentverwerking spelen algemene eigenschappen van editors een rol, zoals: maakt men gebruik van menu's, joystick, functietoetsen, zijn er helpfaciliteiten, is het mogelijk een gegeven commando weer ongedaan te maken, enz. Speciaal van belang hier is de vraag welk bereik commando's hebben: een karakter, woord, regel, paragraaf, enz.

  Meer specifiek voor interactieve systemen zijn vragen als: wat is de correspondentie tussen hetgeen op het scherm wordt getoond en hetgeen afgedrukt wordt, met name bij fonts met een variabele karakterbreedte en van verschillende grootte, en hoe verloopt deze correspondentie in de tijd (is het systeem echt WYSIWYG - What You See Is What You Get - of wordt deze relatie van tijd tot tijd, bij voorbeeld aan het eind van een alinea, hersteld).

Deze lijst is verre van volledig. Een door ons opgestelde lijst van relevante punten is 10 pagina's lang.

Wij zullen ons in het vervolg beperken tot die aspecten die direct gerelateerd zijn aan de mogelijkheden van de formatteerfase. Eigenschappen en mogelijkheden van het editorgedeelte worden dus niet beschouwd. Dit wil overigens niet zeggen dat dit gedeelte door ons niet als belangrijk wordt beschouwd. Integendeel zelfs, de kwaliteit van het totale systeem wordt in niet-onbelangrijke mate bepaald door de functionaliteit en gebruikersvriendelijkheid van de mens-machine interface.

2. Over Testen

Ook in deze sectie is het niet mogelijk een volledig overzicht te geven van alle gangbare methoden en technieken voor het testen van software. Voor uitvoeriger overzichten van deze materie wordt verwezen naar andere bronnen, zoals [9], [10], of [11]. Vermeldenswaard is wel dat, ook al zijn er vrij veel technieken en hulpmiddelen bekend, men zich toch in de praktijk veelal beperkt tot het gebruik van vrij primitieve methoden, zoals code inspections en reviews [12].

Een veel gehanteerde vuistregel is om circa 40% van de beschikbare tijd voor de ontwikkeling van programmatuur te besteden aan testen. Dit komt in veel gevallen neer op het testen van programmacode. De verantwoordelijkheid hiervoor ligt veelal bij de producent. Deze test de verschillende componenten (moduultest) en de integratie hiervan tot een systeem (integratietest). In een aantal gevallen wordt een afspraak met de opdrachtgever gemaakt over de te volgen procedure bij de afname van het product (acceptatietest). Deze laatste stap wordt vaak niet gevolgd bij de selectie van pakketsoftware.

Zelfs uitvoerig testen van software levert geen garantie v.w.b. de correctheid. Hiervoor is een correctheidsbewijs nodig. Hoe lastig deze materie is wordt wel geïllustreerd door het feit dat het voorkomt dat in een bepaald artikel de schrijver op zijn manier bewijst dat een programma correct is, en dat enige tijd later een andere schrijver nog een aantal fouten in datzelfde programma aantoont. Een bloemlezing van foute correctbewezen programma's is te vinden in [13]. De moeilijkheidsgraad van deze materie staat algemene invoering vooralsnog in de weg, terwijl ook de waarde van correctheidsbewijzen door sommigen in twijfel wordt getrokken [14]. De huidige stand van zaken is dat men alleen van eenvoudige programma's de correctheid kan aantonen. Een tekstverwerker hoort zeker niet in deze categorie thuis.

Testen levert in het algemeen een negatief resultaat op: *"Program testing can be used to show the presence of bugs, but never to show their absence"* ([15]). Dit laatste hoeft echter niet het geval te zijn. Indien men een programma uitputtend test moeten alle eventuele fouten ontdekt worden. Als er op deze manier geen fouten geconstateerd zijn, is daarmee ook bewezen dat het programma correct is. Het is echter niet moeilijk in te zien dat het genereren en uitproberen van alle mogelijke invoer bij de meeste programma's ondoenlijk is. Het streven is dan ook een manier te vinden waarop het aantal tests gereduceerd kan worden, zonder echter de bewijskracht van het totaal aan te tasten. Een aanzet tot het construeren van praktische en formele tests is te vinden in Goodenough et al [16]. Hierin probeert men een 'goed' criterium te vinden voor het beoordelen van testsets. Het oordeel 'goed' voor een criterium splitsen de schrijvers in de predikaten *valid* en *reliable.*

VALID(C)
: geeft aan of in de verzameling van testsets die met criterium C te construeren zijn er voor iedere fout in een programma minstens een testset is die deze fout ontdekt.

RELIABLE(C)
: geeft aan of de testsets die met dit criterium C te construeren zijn allemaal hetzelfde oordeel opleveren indien ze als invoer voor het te testen programma dienen.

De schrijvers bewijzen vervolgens dat, indien beide predikaten 'waar' opleveren voor een bepaald criterium, alle testsets die hiermee te bepalen zijn alle (eventuele) fouten moeten aantonen in een programma. Met 'fout' bedoelt men hier het geval dat bepaalde invoer niet de gespecificeerde of gewenste uitvoer oplevert. Verder onderzoek zal er op gericht moeten zijn dergelijke criteria te vinden,

de predikaten 'valid' en 'reliable' verder op te splitsen, of een eigen omschrijving van een 'goed' criterium te definiëren. HOWDEN heeft inmiddels aangetoond ([17]) dat er geen criterium bestaat dat voor élk programma een betrouwbare testset kan bepalen.

In de praktijk valt men dan ook veelal terug op heuristische technieken en wordt programmatuur niet volledig getest. Bij het pogen programmatuur te certificeren, krijgt men echter nog met andere problemen te maken: men mag er van uitgaan dat de oorspronkelijke programmacode niet voorhanden is; de fabrikant zal deze waarschijnlijk niet beschikbaar willen stellen. ([18] p. 94: *"... the source text is often a commercial secret and unavailable"*) Dit sluit een aantal testmethoden uit. Het construeren van tests waarbij door geschikte keuze van invoer alle 'paden' binnen een programma worden begaan, is niet goed mogelijk; die paden zijn immers op grond van de objectcode nauwelijks vast te stellen. Ook is het niet mogelijk een programma in delen te testen. Correctheidsbewijzen steunen ook voor een groot deel op de programmacode of een symbolisch equivalent daarvan, waardoor deze methode onbruikbaar wordt. Hetzelfde geldt voor symbolische executie.

Er zijn weinig testmethoden die goed toepasbaar zijn op complete programma's zonder dat daarbij de programmacode voorhanden is. Praktisch alle theorieën zijn opgesteld voor gebruik door de programmeur zelf, of het team dat het programma ontwikkelt.

In feite zijn alle statische test- en bewijsmethoden, en alle dynamische waarbij men uitgaat van de programmacode, onbruikbaar. Ook het bepalen van andere kwaliteitsaspecten dan correctheid kan niet uitgevoerd worden op de manier zoals die onder andere in BOEHM et al [19] beschreven staat. Dit alles reduceert de kijk die men heeft op een programma tot die op een 'black box': een zwarte doos die langs magische weg een bepaalde invoer omzet in bepaalde uitvoer. Een hiervoor bruikbare testmethode is 'functioneel testen'. Bij functioneel testen gaat men uit van een *beschrijving* van het programma. Het is ideaal als deze beschrijving gegeven is in een soort algebraïsche, of in ieder geval formele, vorm. Het zal echter nog wel enige tijd duren voordat de specificaties van tekstverwerkers in een wiskundige vorm gegeven (kunnen) worden, hoewel een tekstverwerker met SGML [20] als basis een stap in de goede richting is.

Een van de methoden van functioneel testen is gebaseerd op een indeling van de mogelijke invoer in een eindig (liefst klein) aantal equivalentieklassen, en wel zo dat ieder element uit een equivalentieklasse deze volledig representeert. Men probeert vervolgens tests te vinden die zoveel mogelijk equivalentieklassen overdekken; alle tests tezamen moeten alle klassen overdekken.

Het bepalen van deze equivalentieklassen is een heuristisch proces. Men kan zich hierbij bijvoorbeeld baseren op de diverse invoercondities welke in de beschrijving van het programma voorkomen. Elke invoerconditie levert twee of meer klassen op. Zo levert de regel

elk boek heeft een nummer, bestaande uit 6 cijfers

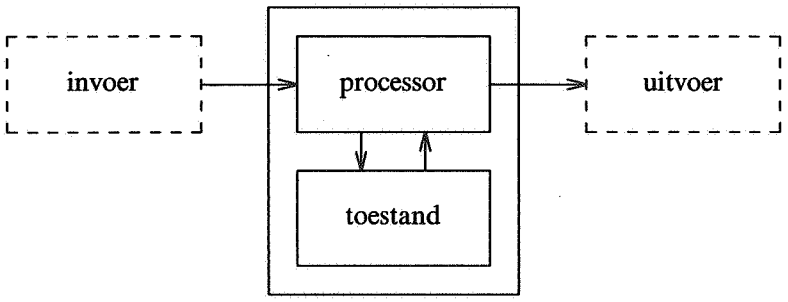
drie equivalentieklassen op: een geldige, en twee ongeldige (corresponderend met een nummer met te veel respectievelijk te weinig cijfers).

Wat betreft de hoeveelheid tests levert deze methode een hele vooruitgang ten opzichte van uitputtend testen. Bij minder triviale programma's echter is het samenstellen van dit soort tests geen sinecure. Men kan zich ook afvragen of het bij certificatie wel nodig en mogelijk is zo diep op de structuur van een programma en zijn invoer in te gaan. Men zou gebaat zijn bij een methode die tot beduidend minder testgevallen leidt.

De testmethoden die in de literatuur zijn beschreven leiden tot een schier eindeloze hoeveelheid testgevallen, zelfs bij vrij kleine programma's. Indien we daarbij nog de opmerking van DIJKSTRA in ogenschouw nemen dat kleine en grote programma's *"utterly incomparable"* zijn [15], dringt dit de conclusie op dat bestaande testmethoden niet gebruikt kunnen worden bij certificatie.

3. Aanzet tot een nieuwe Testmethode

Het is in het algemeen niet voldoende alleen de correcte verwerking van een commando of bepaalde invoer *op zich* te testen. De voorafgaande invoer bepaalt immers de 'toestand' waarin een programma verkeert, en deze toestand kan van invloed zijn op de verdere verwerking. Men kan dit vergelijken met de werking van een *Eindige Automaat*. (Een korte omschrijving van dit begrip wordt gegeven in [21].) De stap die een dergelijke automaat neemt naar aanleiding van een bepaald symbool is niet alleen afhankelijk van dat specifieke symbool, maar wordt mede bepaald door de toestand (configuratie) van de automaat. Deze toestand is bepaald door de invoer die de automaat hiervoor reeds verwerkt heeft. Het algemene model van een programma dat wij hanteren is afgeleid van een Eindige Automaat en wordt door figuur 1 weergegeven.



Figuur 1. Basis-model van een programma

Bij het onderzoeken van de verwerking van een bepaald commando is het dus niet voldoende te kijken of deze onder één toevallige toestand goed geschiedt, maar in principe moet dit commando in alle mogelijke toestanden getest worden. Bij de meeste programma's is het aantal mogelijke toestanden zeer groot. Bij een praktische benadering van testen zal men dus uit deze (zeer) grote verzameling een aantal representatieve waarden moeten kiezen. De methode die we daarvoor willen presenteren is afgeleid van de methode van 'functioneel testen' zoals deze in [10] en [22] beschreven staat.

Wij onderscheiden in een programma twee delen; een procesdeel en gegevensverzamelingen. Deze verzamelingen vormen in feite de omgeving of context van de processen. Een dergelijke verzameling kan men opgebouwd zien uit verschillende 'variabelen'; de waarden van deze variabelen vormen de toestand van de gegevensverzameling en daarmee van het proces dat deze verzameling gebruikt. (Als voorbeeld kan men bij tekstverwerking denken aan 'variabelen' als 'huidige regellengte' en 'huidige positie op de pagina'.)

Voor deze variabelen — we zullen ze *attributen* noemen — kiest men een aantal representatieve waarden. De toestanden die men gebruikt, zijn alle combinaties van deze waarden voor de verschillende attributen. Als er bijvoorbeeld twee attributen zijn waarvoor men drie waarden gekozen heeft, dan zijn hieruit negen toestanden af te leiden.

Op deze manier is reeds een forse reductie in het aantal testgevallen te bereiken. Als echter in een gegevensverzameling twintig attributen te onderscheiden zijn — wat zeker niet onaannemelijk lijkt — waarvoor men steeds drie waarden gekozen heeft, dan is het aantal te gebruiken toestanden toch nog 20^3. Dit noodzaakt een verdere reductie.

Eén van de mogelijkheden is het programma in onafhankelijke delen op te splitsen, en die ieder apart te testen. Hiervoor is echter de programmacode en kennis betreffende de werking van dit programma noodzakelijk. Onze hypothese is dat deze niet voorhanden zijn. Wij stellen daarom een soort opsplitsing 'op logisch niveau' voor, waarbij men steeds één onderdeel test, en de andere (zoveel mogelijk) buiten beschouwing laat.

Hiertoe doet men een aantal aannames betreffende de werking van een programma. Deze aannames formuleert men in de vorm van een model. In dit model laat men tot uitdrukking komen welke deelfuncties te onderscheiden zijn, en welke gegevens hierdoor gebruikt worden. Het doel hiervan mag duidelijk zijn: per deelfunctie heeft men nu een beperkt aantal attributen waarvan

combinaties bepaald moeten worden. Het mes snijdt echter aan twee kanten; per deelfunctie hoeft men ook niet alle commando's te testen, maar alleen die welke betrekking hebben op die deelfunctie.

Mogelijke deelfuncties bij een tekstverwerker zijn o.a.: de eigenlijke formatteeralgoritme, met attributen als: de huidige positie van de cursor in de uitvoer, het huidige font en de grootte van de huidge pagina, en de gehanteerde methode voor het afbreken van woorden. Deze laatste deelfunctie wordt in sectie 3.2 nader uitgewerkt.

Uiteindelijk is het effect dus dat een commando getest wordt in verschillende toestanden van de totale gegevensverzameling waarbij slechts de relevante attributen gevarieerd worden.

Een bezwaar dat men tegen deze methode zou kunnen maken is dat het model niet het feitelijke functioneren van het programma hoeft te weerspiegelen, en dat tests die hierop gebaseerd worden niet de relevante toestanden van de gegevensverzamelingen in beschouwing nemen. Het is echter de bedoeling dat in het model vooral informatie wordt gerepresenteerd die onafhankelijk is van een specifieke implementatie. Als voorbeeld bij tekstverwerking: het gegeven 'huidig font' is altijd van belang voor het programma-onderdeel dat de uiteindelijke verdeling van tekst over regels bepaalt, maar nooit voor, zeg, een ingebouwde macroprocessor. Op dit niveau is de relatie tussen gegevens en processen nog niet of nauwelijks beïnvloed door de exacte werking van een programma.

Dit duidt natuurlijk op een gevaar dat men loopt als het model te ver gespecificeerd wordt. Er is echter nog een andere reden om niet al te veel deelprocessen en gegevensverzamelingen te onderscheiden; een 'grof' model zal waarschijnlijk eerder een hele categorie programma's representeren dan een zeer gedetailleerd model, dat hoogstens de werking van één of twee programma's zal weergeven. Hoe ver men in de praktijk gaat met dit specificeren wordt echter bepaald door de vraag hoe gedetailleerd het model moet zijn eer het bruikbaar is bij het testen; het kan nodig zijn meerdere modellen te formuleren voor een bepaalde categorie software. Zo lijkt het niet mogelijk de zogenaamde 'batch' en interactieve tekstverwerkers redelijkerwijs in één model te verenigen.

Het zal voor een aantal categorieën software waarschijnlijk mogelijk zijn zo'n model over te nemen uit de literatuur. Dit verdient zeker de voorkeur, omdat het zelf formuleren hiervan toch vrij veel kennis en inzicht vergt in het functioneren van een dergelijk programma. Om een algemeen model te kunnen formuleren moet men op de hoogte zijn van de stand van zaken op het betreffende vakgebied. Dit alles noodzaakt een vrij uitgebreide studie, welke misschien verkort kan worden door andermans werk te raadplegen.

3.1. Het samenstellen van tests

Algemeen gebruikte methoden voor het vaststellen van testwaarden zijn niet zozeer gestoeld op theoretische beschouwingen, als wel empirisch bepaald met het oog op veel voorkomende fouten. Een goede beschrijving hiervan is te vinden in [10] en [22]. De regels daarin beschreven nemen wij voor een deel over in onze theorie. Dit ondanks het feit dat men in bovengenoemde artikelen uitgaat van de waarden die een interne variabele van een programma kan aannemen; nagenoeg dezelfde regels gelden namelijk voor de attributen binnen onze methode.

Het bepalen van tests voor attributen met slechts twee mogelijke waarden (zoals 'wel of niet afbreken' of 'wel of niet regels uitvullen') is eenvoudig. Bij een attribuut als de regellengte ligt dit iets moeilijker, hier moet men kiezen uit meerdere mogelijkheden. Het is de gewoonte om in een geval als dit elementen uit de volgende klassen te kiezen:

- 'normale' gevallen
- grensgevallen (nog wel correct), ook wel extreme waarden genoemd
- speciale waarden
- incorrecte gevallen (indien mogelijk)

Omdat onze methode zich niet specifiek richt op het vinden van fouten in merkwaardige situaties, lijkt het minder zinvol om de laatste klasse in beschouwing te nemen. We nemen ook wat betreft de grenswaarden niet zonder meer de theorie van functioneel testen over; bij een tekstverwerker lijkt het weinig zinvol om nul te gebruiken als regellengte of paginalengte. In zo'n geval is het nuttiger een kleine en een grote waarde te kiezen en niet specifiek de grenzen van het domein. Hier zijn natuurlijk

ook uitzonderingen op.

Bij tekstverwerking moet men in de werking van een commando twee aspecten onderscheiden. (Dit geldt niet noodzakelijk voor andere soorten software.) Deze aspecten zijn:

- uitvoering van het commando veroorzaakt een verandering van de omgeving waarbinnen dit commando werkt;
- het commando veroorzaakt uitvoer.

Een commando draagt een of beide kenmerken, en de commando's zijn dus in drie categorieën onder te brengen:

- Het commando verandert alleen de omgeving;
- Het commando veroorzaakt alleen uitvoer;
- Het commando verandert de omgeving én veroorzaakt uitvoer.

De werking van commando's is niet altijd rechtstreeks te testen omdat het effect niet noodzakelijk direct zichtbaar is voor de gebruiker. Het effect van een commando om de regellengte te veranderen geeft veelal pas een zichtbaar effect nadat een zekere hoeveelheid tekst is ingevoerd. Wanneer opdracht gegeven wordt om een bepaalde voetnoot te plaatsen moet men wachten tot de pagina vol is voor het effect zichtbaar wordt.

Of de omgeving veranderd is in de zin die men verwachtte, zal men aan de oppervlakte moeten zien te halen door commando's uit te voeren die wel uitvoer genereren en afhankelijk zijn van de omgeving van het te testen commando. Het is hierbij het meest zinvol zich te richten op die commando's die de verandering van de attributen, waarvan men verwachte dat ze zouden veranderen, het beste zichtbaar maken. Men kan het natuurlijk ook overwegen na te gaan of in de betrokken omgeving(en) nog attributen veranderd zijn waarvan men dit niet verwachtte.

Tot nu toe is alleen gesproken over het testen van commando's. Bij een tekstverwerker bestaat de invoer echter niet slechts uit opdrachten, er zit ook nog tekst bij. Het is echter niet noodzakelijk een fundamenteel onderscheid te maken tussen commando's en tekst. Een woord (uit de tekst) kan men ook zien als commando: 'druk dit woord af'.

Het belangrijkste aspect dat bij de verwerking van een woord voor de tekstverwerker van belang is, is de lengte. Dit is echter niet helemaal voldoende; bij het afbreken is de 'inhoud' wel degelijk van belang voor het vaststellen van de plaats waar het woord eventueel afgebroken kan worden (zie ook sectie 3.2).

Zoals hierboven reeds gezegd, worden de aspecten van een omgeving die men bij het testen in beschouwing moet nemen, bepaald door de commando's die er zijn om die omgeving te beïnvloeden. Een voorbeeld: in een tekstverwerker is de gegevensverzameling 'layout-specificaties' te zien als een onderdeel van de omgeving van een bepaald programma-onderdeel. Als er bij deze tekstverwerker een opdracht is om een ander font te kiezen dan zal men bij de gegevensverzameling 'layout-specificaties' het attribuut 'huidig font' in beschouwing moeten nemen.

Per functie uit het model van het programma bepaald men een serie tests. Deze tests zijn in feite alle permutaties van de attributen uit de omgeving van deze functie voor al hun waarden. Aan het begin van zo'n test worden deze attributen geïnitialiseerd. Hierna geeft men het ene commando, dat men in deze specifieke omgeving wil testen, als invoer. Eventueel volgen hierna nog enige opdrachten om het effect van die ene opdracht op de omgeving te onderzoeken. Hiermee is de test afgesloten.

3.2. Voorbeeld: Het afbreken van woorden

In het model van een tekstverwerker is het programmadeel dat afbrekingen van woorden bepaalt als aparte functie te onderscheiden. Deze functie bepaalt niet zelf de uiteindelijke plaats van afbreken maar geeft de plaatsen aan waar dat eventueel mogelijk is. Bij een specifieke tekstverwerker die men wil testen haalt men uit bijvoorbeeld de gebruikersdocumentatie de informatie dat de plaats waar een woord wordt afgebroken op drie manieren bepaald kan worden:

- door een ingebouwde 'afbreekalgoritme' die zelfstandig de plaats van afbreken kan bepalen;
- door middel van een lijst van uitzonderingen die door de gebruiker opgegeven moet worden;
- doordat de gebruiker in een woord in de tekst zogenaamde 'stille streepjes' plaatst.

Bovengenoemde mogelijkheden staan in volgorde van oplopende prioriteit; als een woord op de lijst van uitzonderingen staat dan heeft dit voorrang bij het bepalen van de plaats van afbreken boven de automatische algoritme. Indien een woord voorzien is van stille streepjes wordt de uitzonderingslijst niet geraadpleegd, en wordt de ingebouwde algoritme niet geactiveerd. Het is de taak van de afbreekfunctie om deze prioriteitsregels van toepassing te laten zijn. Er zijn drie deelfuncties te onderscheiden, een voor elk van de drie methoden van afbreken. Tezamen vormen deze de omgeving van de afbreekfunctie.

Voor het testen zal men eerst moeten weten in welke toestanden deze omgeving kan verkeren. De eerste functie (de 'automatische' manier van het bepalen van de plaats van de afbreking) is altijd in dezelfde toestand; de gebruiker kan deze niet beïnvloeden. De methode die met de woordenlijst werkt kan wel in verschillende toestanden verkeren: een woord dat afgebroken moet worden staat wel of niet op deze lijst. Wat betreft het gegeven of er stille streepjes in een woord staan kan men ook twee verschillende toestanden onderscheiden: er zijn wel of geen stille streepjes aangebracht.

Dit levert in totaal vier verschillende toestanden op van de omgeving van de afbreekfunctie betreffende een bepaald woord. In elk van deze verschillende toestanden van de omgeving zal men verschillende woorden moeten testen.

De eerste deelfunctie die we onderscheiden is die welke afbrekingen bepaalt aan de hand van de zogenaamde stille streepjes. De enige invoer voor dit deelproces is een woord met dergelijke streepjes dat afgebroken moet worden. Aan zo'n woord kunnen we twee aspecten onderscheiden:

- de lengte van het woord, en
- de plaats van de stille streepjes in dat woord.

Taalkundige aspecten van het woord spelen hier geen rol, en men zou dus 'xxxxx' als testgeval kunnen gebruiken. Wat betreft de testwaarden voor de bovengenoemde attributen van een woord nemen we voor de lengte een kort (zeg 4 karakters) en een lang (zeg 30 karakters) woord, voor het aantal streepjes daarin weinig (1 of 2) en veel (bijvoorbeeld tussen iedere twee karakters). Dit resulteert in vier testgevallen. Om na te gaan of er correct wordt afgebroken moet het woord meerdere malen op ongeveer de regelgrens in de uitvoer afgedrukt worden.

De tweede deelfunctie die men kan onderscheiden is die welke afbrekingen bepaalt aan de hand van de lijst van uitzonderingen. Hier spelen twee verschillende opdrachten een rol: een commando om een woord, met de mogelijke afbreekpunten, aan die lijst toe te voegen, en een opdracht om gegeven een woord de afbreekmogelijkheden aan te geven. De lijst met uitzonderingen is te zien als de omgeving van dit deelproces. De relevante aspecten van de toestand waarin deze lijst kan verkeren zijn:

- het aantal woorden dat al op de lijst staat, en
- de lengte van de woorden die al op de lijst staan.

Voor het aantal woorden dat op de lijst staat zal men een aantal representatieve waarden moeten kiezen; we nemen 0, weinig (zeg 4) en veel (zeg 1000) woorden, wat betreft de lengte van de woorden nemen we korte (zeg 5 karakters) en lange (zeg 30 karakters) woorden. De combinaties van deze mogelijkheden zijn de toestanden waarin men beide opdrachten moet testen.

Zowel bij het toevoegen van een woord aan de lijst met uitzonderingen als bij het zoeken in deze lijst spelen de lengte van het woord en de plaats van de stille streepjes een rol. De testgevallen kunnen dus op dezelfde wijze gekozen worden als bij de vorige deelfunctie.

Als laatste is er de automatische afbreek algoritme. Deze is te beschouwen als een 'black box', men kan alleen maar kijken hoe deze reageert op bepaalde invoer. De functie van dit onderdeel is het afbreken van woorden in een bepaalde taal. Om dit onderdeel te testen zal men dus een aantal representatieve woorden uit die taal als invoer moeten geven, en bepalen of deze correct afgebroken worden. Het bepalen van deze representatieve woorden en hun afbreking vereist wel enige kennis van die specifieke taal.

Bovengenoemde aanpak van afbreken werkt alleen voor zover woorden gesplitst kunnen worden door tussen twee opeenvolgende karakters een streepje te plaatsen. Om Nederlands echt goed af te breken is meer nodig. Zo wordt 'menuutje' afgebroken als 'menu-tje'. Met de semantiek van woorden wordt in het geheel geen rekening gehouden ('dijkramp' leidt tot 'dij-kramp' dan wel 'dijk-ramp').

4. Conclusies

We kunnen uit deze case study twee conclusies trekken:

- Voor de door ons beschreven vorm van certificatie zijn bestaande testtechnieken ongeschikt;
- Certificatie van een bepaald type software vereist een forse investering, terwijl expertise op het betrokken vakgebied onontbeerlijk is.

De geschetste methode zal zeker niet alle fouten in het betreffende programma aantonen. Dit wordt al in de hand gewerkt door het feit dat alleen combinaties van 'logisch gerelateerde' functies getest worden. De methode is ook niet ontwikkeld om zoveel mogelijk fouten in een programma op te sporen. Daarvoor is het beter methoden te gebruiken die bijvoorbeeld afgaan op kennis betreffende gevallen waar programmeurs vaak fouten maken (overschrijden van array-grenzen e.d.).

De definitie van testen die Myers [11] hanteert (*'Testing is the process of executing a program with the intent of finding errors.'*) levert voor ons doel geen geschikt uitgangspunt. Wat hij beschrijft is in feite falsificatie. Bij certificatie is dit niet het primaire doel.

Onze methode is meer bedoeld om een algemeen beeld van 'de mate van correctheid' van een programma te krijgen. Er wordt in zekere zin een 'steekproef' gedaan. De elementen uit de steekproef worden bepaald aan de hand van de bovengeschetste methode.

Er is een aanzienlijke hoeveelheid menskracht nodig voor het opstellen van een volledige lijst van criteria die een rol spelen bij een gegeven type software, terwijl er tevens een grondige kennis van het betreffende onderwerp nodig is. Voor een ander type software, zoals databasepakketten voor financiële administratie, is deze investering opnieuw nodig.

Tot slot.

Discussies met de overige bij deze voorstudie betrokken personen hebben in belangrijke mate bijgedragen tot ons begrip van de geschetste problematiek.

Referenties

1. *A Study on Methods Available to the Commission for Ensuring the Setting Up of European (or World-wide) Reference Testing Services to Informatics Standards,* Brussels (1984), Commission of the European Communities.
2. T. de Heer (1985). *Persoonlijke communicatie.*
3. *Information Processing Standardization Specification,* Brussels (1985), Commission of the European Communities.
4. Initiatiefgroep Software Certificatie (november 1985). Persbericht, *Informatie*, 26.11, 937.
5. G. C. Nielen, R. Paans, M. A. Bongers (januari 1985). Certificering van Software: Noodzakelijk of Zinloos?, *Informatie*, 27.1, 21-24.
6. R. A. C. Thomas, R. Paans (januari 1985). Certificering van Software: Utopie of Werkelijkheid?, *Informatie*, 27.1, 25-34.
7. L. G. Bouma, J. Bruijning, J. C. van Vliet (1985). *Documentverwerking*, CWI, Amsterdam, CS-N8504.
8. R. Furuta, J. S. Scofield, A. Shaw (september 1982). Document Formatting Systems: Survey, Concepts, and Issues, *Computing Surveys*, 14.3, 417-472.
9. J. C. van Vliet (1984). *Software Engineering*, Stenfert Kroese.
10. W. R. Adrion, M. A. Branstad, J. C. Cherniavsky (juni 1982). Validation, Verification and Testing of Computer Software, *Computing Surveys*, 14.2, 159-192.

11. G. J. MYERS (1979). *The Art of Software Testing*, Wiley-Interscience.
12. M. V. ZELKOWITZ, R. T. YEH, R. G. HAMLET, J. D. GANNON, V. R. BASILI (juni 1984). Software Engineering Practises in the US and Japan, *IEEE Computer*, 17.6, 57-66.
13. S. L. GERHART, L. YELOWITZ (september 1976). Observations of Fallibility in Applications of Modern Programming Methodologies, *IEEE Transactions on Software Engineering*, 5.2, 195-207.
14. R. A. DE MILLO, R. J. LIPTON, A. J. PERLIS (mei 1979). Social Processes and Proofs of Theorems and Programs, *Communications of the ACM*, 22.5, 271-280.
15. O. J. DAHL, E. W. DIJKSTRA, C. A. R. HOARE (1972). *Structured Programming*, Academic Press.
16. J. B. GOODENOUGH, S. L. GERHART (juni 1975). Toward a Theory of Test Data Selection, *IEEE Transactions on Software Engineering*, 1.3, 156-173.
17. W. E. HOWDEN (september 1976). Reliability of the Path Analysis Strategy, *IEEE Transactions on Software Engineering*, 5.2, 208-215.
18. B. A. WICHMANN, Z. J. CIECHANOWICZ (1983). *Pascal Compiler Validation*, John Wiley & Sons.
19. B. W. BOEHM, J. R. BROWN, H. KASPAR, M. LIPOW, G. J. MACLEOD, M. J. MERRIT (1978). *Characteristics of Software Quality*, TRW Series of Software Technology, North-Holland Publishing Company.
20. C. F. GOLDFARB (juni 1981). A Generalized Approach to Document Markup, *SIGPLAN Notices*, 16.6, 68-73.
21. P. M. B. VITÁNYI (1984). Theoretische Informatica, *Polyautomatiserings Zakboekje*, Koninklijke PBNA 423-443.
22. W. E. HOWDEN (juni 1982). Validation of Scientific Programs, *Computing Surveys*, 14.2, 193-227.