Trollkarlen blir Ingenjör

—

Formella Metoder: en Översikt

Joachim Parrow
Professor i Distribuerade System
Kungl. Tekniska Högskolan, Stockholm

Maj 1997

Den fjärde juni 1996 exploderade den första Ariane 5 raketen, 39 sekunder efter start, på grund av ett programmeringsfel i styrsystemet. För erfarna systemutvecklare är haveriet knappast förvånande. Man hade tagit delar av programmen från föregångaren, Ariane 4. Huvudregeln i programmering, att inte ändra i det som fungerar bra, gjorde att de nya förutsättningarna i Ariane 5 inte beaktades.[1] Med en fyrverkerikostnad om 8 miljarder dollar är detta ett spektakulärt symptom på ett djupt och länge känt problem: för komplicerade datorsystem är det svårt att fullständigt förutsäga beteendet.

Att samma förhållande gäller för hårdvara illustreras av den pinsamma incidenten med Intels tidigaste pentiumprocessorer: i oktober 1994 upptäcktes att de ibland räknar fel när de dividerar två tal. Kostnaden att ersätta miljontals felaktiga chips uppgick till 475 miljoner dollar. Och att det gäller för telefonväxlar vet det amerikanska telefonbolaget AT&T: den 15 januari 1990 gick halva deras telefonnät ner under nio timmar på grund av ett programfel. Liknande fadäser är legio, om än inte lika synliga, inom alla datorintensiva tillämpningar.

Flera så kallade "formella metoder" utvecklas för att komma tillrätta med sådana problem. Idén är att beteenden ges noggranna beskrivningar som kan analyseras logiskt för att avslöja fel. De senaste tio åren har forskningsinsatserna ökat kraftigt och de senaste åren har några av metoderna även börjat tillämpas systematiskt industriellt. I denna översikt beskriver jag några vanliga ansatser och svårigheter.

[1] Se tex `http://www.esrin.esa.it/htdocs/tidc/Press/Press96/press33.html`.

Inledning

Tillräckligt utvecklad teknologi kan likna magi. Trollkarlens hatt, mjölnarens kvarn och sekreterarens dator är alla lika häpnadsväckande för den oinvigde. Men det finns en viktig skillnad: vi litar på att kvarnens konstruktörer har begripit hur hjulen ska snurra. Den som använder magi däremot riskerar oväntade och oönskade resultat då en trollkarl inte alltid är herre över sin konst.

På detta sätt kan man med fog klassa datorer som trolltyg. Hackern är vår tids häxmästare som frambesvärjer fantastiska och förfärande ting utan att förstå dem. Datorsystem styr nu allt från telefonväxlar till flygplan och kärnkraftverk trots att ingen, inte ens deras skapare, kan redogöra för varför och under vilka förutsättningar de fungerar.

Under de senaste tjugo åren har därför ett nytt forskningsområde vuxit fram inom datalogin: *Formella Metoder.* Man har förväntat sig att det ska frälsa oss från ”går det så går det” mentaliteten och tillhandahålla ingenjörsmässiga sätt att bygga datorsystem. Till viss del har förhoppningarna varit överdrivna. Även om vetenskapliga framsteg skett så arbetar informationsingenjörerna i stort på samma sätt som förut.

Internationellt är området gigantiskt med en mängd vetenskapliga böcker, tidskrifter och konferenser, men också en hel del dokumenterade erfarenheter från stora användningsområden inom industriella projekt. Tillgången på tekniska översikter av olika slag är utomordentligt god.[2] Här har jag därför valt en presentation tänkt för läsare utan högre vetenskaplig och teknisk skolning i ämnet.

Beteenden

Ingenjörskonsten är högt utvecklad inom många områden och dess adelsmärke är att konstruktion inleds med omfattande planering. När man ska bygga en kvarn börjar bara amatören med att spika vid stranden. Yrkesmannen gör först ritningar och modeller och kan på så vis räkna ut viktiga egenskaper hos det som ännu inte är byggt. Om ett kugghjul blir för klent upptäcks detta redan i modellen och man kan lätt ändra konstruktionen.

Vi informationsingenjörer däremot arbetar i bästa fall med ofullständiga och tvetydiga ritningar. Oftast skruvar vi snabbt ihop ett system och vill kompensera bristen på planering med en grundlig utprovning. Många skyller detta på att informationsteknologi är en ung vetenskap som inte ännu hunnit resultera i en utvecklad ingenjörskonst. Detta kan dock knappast vara hela sanningen. Datorn som teknisk uppfinning är ungefär jämnårig med både atombomben och jetplanet och äldre än rymdraketen.

[2] Den som är intresserad kan med fördel besöka hemsidan `http://www.comlab.ox.ac.uk/archive/formal-methods.html` för att få mer information, den innehåller referenser till flera utmärkta introduktioner.

En rimligare förklaring är att informationsteknologi behandlar immateriella konstruktioner som inte låter sig beskrivas med välkända mekaniska egenskaper som vridmoment och friktion. När datorerna arbetar är deras fysiska attribut oviktiga, det intressanta är deras beteende. Men exakt vad menas med "beteende" och i vilka termer kan det göras begripligt? Man vill gärna tänka på det materiellt: knappar trycks in, lampor tänds, signaler åker genom ledningar. Eller man kan se det antropomorfiskt: datorn tror, maskinen vill, systemet surar. Men de materiella liknelserna är för begränsade och antropomorfin för vag för att vara ritningsunderlag.

Lämpliga begrepp för beteenden är därför starkt beroende av tillämpningsområdet. En allmän idé är att identifiera viktiga *händelser*, till exempel "förbindelsen kopplas upp", "värdet beräknas", "data tas emot". En annan är att klassifiera *tillstånd*, till exempel "klar att ta emot data", "förbindelsen är uppkopplad" etc. Exakt vad som utgör händelser och tillstånd måste bero av vad det är som beskrivs och på vilken detaljnivå beskrivningen ska gälla. Så kan till exempel "förbindelsen kopplas upp" vara en händelse, medan en mer detaljerad beskrivning av samma sak använder flera händelser "Luren lyfts", "Signal går fram", ... Ju rikare detaljnivå ju större precision, men desto längre och komplicerad blir också hela beskrivningen — ett vanligt dilemma för ritningar i allmänhet.

Formella språk

Men även i de fall där man bestämt detaljnivån återstår problemet att beskriva ett beteende på ett entydigt sätt. Att detta inte alltid är lätt illustreras av följande enkla exempel. Antag att vi har valt två viktiga händelsetyper i ett datanät, "förbindelsen kopplas upp" (U) och "förbindelsen kopplas ner" (N), och vill beskriva att först måste U ske, därefter N, därefter möjligen U, därefter igen N och så vidare. Hur kan detta formuleras så att missförstånd utesluts?

Beskrivning	Möjligt missförstånd
"U och N alternerar"	Först kan N ske, och sedan U

Vi hade glömt att ange det självklara att förbindelsen måste kopplas upp innan den kan kopplas ner. Men motsatsen är rent språkligt en möjlig tolkning. Så låt oss precisera utsagan:

Beskrivning	Möjligt missförstånd
”U och N alternerar med början på U”	Först kan U ske, sedan N, sedan U, sedan N, och sedan behöver inget mer ske!

I själva verket hade vi ju tänkt att uppkoppling skulle kunna ske hur många gånger som helst, inte bara två gånger. Återigen får vi precisera för att undvika missförståndet:

Beskrivning	Möjligt missförstånd
”U och N alternerar godtyckligt länge med början på U”	Först kan flera U ske i rad, och sedan kan lika många N ske i rad.

Nu är det uppenbarligen ordet ”alternerar” som plötsligt inte är helt klart. För att göra beskrivningen mer bestämd måste vi exakt definiera de ord som förekommer där. Men hur gör vi det utan att skapa nya missförstånd, och när kan vi vara säkra på att det bara finns en möjlig tolkning?

Våra beskrivningar ovan är formulerade på vanlig svenska. Sådana kan inte fungera som ritningar i ingenjörens bemärkelse eftersom de är tvetydiga. Av den anledningen används istället speciella notationer, så kallade *formella språk*. Dessa är inte språk i ordets vanliga mening eftersom det som kan uttryckas i dem är starkt begränsat. Till exempel går det inte att säga ”var snäll och köp lite mjölk på hemvägen” eller ”det var väldigt vad han såg vissen ut”. Däremot går det bra med ”U och N alternerar”. I formella språk är betydelsen av utsagor definierade med matematisk noggrannhet och man har reducerat uttrycksfullheten för att vinna i precision. Samma förhållande gäller för ingenjörens ritningar inom andra områden.

Tyvärr finns idag ingen allmänt vedertagen standard för hur ett formellt språk ska se ut för att bäst kunna beskriva datorsystem. Det finns tex ”grafiska” språk där de betydelsebärande elementen är olika former av diagram. Dessa kan vara lättare att förstå för nybörjare, men vana ingenjörer finner diagrammen klumpiga och utrymmeskrävande jämfört med textbaserade beskrivningar. I ”objektorienterade” notationer sönderdelas en beskrivning i en samling växelverkande objekt. Det finns ”deklarativa” språk som används för att beskriva önskade egenskaper hos datorsystemet utan att gå in på detaljer, och ”operationella” som beskriver noggrant, steg för steg, vad som händer under körning. De senare liknar ofta moderna programmeringsspråk och det gör dem lättare att lära.

I den praktiska tillämpningen av formella språk upplevs ofta — paradoxalt nog — matematisk precision inte enbart som en tillgång. Anledningen är att i tidiga

Några olika notationer

Att beskriva ”U och N alternerar” på ett entydigt sätt är enkelt i en formell notation. Vi kan rita ett litet tillståndsdiagram, där innebörden är att de möjliga sekvenserna av händelser fås genom att följa pilarna med början vid tillstånd 1:

$$\begin{tikzcd} 1 \arrow[r, shift left, "\mathrm{U}"] & 2 \arrow[l, shift left, "\mathrm{N}"] \end{tikzcd}$$

I en textbaserad notation kan vi använda ett *konnektiv*, det vill säga ett sätt att bilda sammansatta beteenden, som betyder ”i följd”. Om B_1 och B_2 är beteenden, låt ”B_1 ; B_2” betyda att först utförs B_1 och sedan B_2. Upprepad alternering av U och N kan då ges av

$$\mathrm{ALT} = \mathrm{U}\ ;\ \mathrm{N}\ ;\ \mathrm{ALT}$$

vilket betyder att beteendet ALT är att först göra U, sedan N, och sedan återigen fortsätta som ALT.

Olika språk ritar symboler och skriver konnektiv på olika sätt, vilket kan vara förvirrande. Dock är skillnaderna oftast ytliga. Antalet principiellt olika konnektiv och diagramtyper är inte stort trots att det finns hundratals språk och dialekter.

Språket SDL (Specification and Description Language) utvecklas sedan mitten av 1980-talet av intressenter inom datakommunikationsområdet, bland annat i Sverige. SDL är ”operationellt”, det vill säga de möjliga händelseföljderna beskrivs i detalj. Det finns både en grafisk och en textbaserad variant, och en stor mängd hjälpmedel för att skriva och bearbeta formella modeller.

Språket Z är av annan art. Det tillhör familjen ”deklarativa” språk där önskvärda egenskaper kan uttryckas med logiska formler utan att nödvändigtvis använda händelser. Så kan exempelvis utsagan att listan L är sorterad uttryckas

$$\forall i, j : \operatorname{dom} L \ \mid \ i \leq j \ \bullet \ (L\, i) \leq (L\, j)$$

Parafraserat, ”För alla platser i och j i listan där i är före j så måste elementet på plats i vara mindre än det på plats j.” Z utvecklas bl.a. vid universitetet i Oxford och har en god internationell spridning. Hjälpmedlen är inte lika välutvecklade som för SDL. Bra hjälpmedel är svårare att göra för deklarativa språk eftersom de tillåter mer generella utsagor.

Mer information om SDL finns på hemsidan `http://www.sdl-forum.org/`.
Mer information om Z finns på `http://www.comlab.ox.ac.uk/archive/z.html`.

faser av ett konstruktionsarbete är förutsättningarna inte tillräckligt utredda för att möjliggöra exakta modeller. Så kan till exempel en konstruktör ibland faktiskt vilja uttrycka ”U och N alternerar” i dess vaga form för att först i ett senare skede precisera hur detta ska tolkas, eller ”datorn har minst 16 megabyte arbetsminne” utan att ange en övre gräns. Därför måste formella språk också tillåta begrepp som inte ges en fullständig eller entydig tolkning. De har då fortfarande den viktiga funktionen att att ange vad som är definierat och vad som återstår att utröna. I hårt tidspressade projekt låter man lätt annars detta vara osagt.

Analysproblemet

När en ingenjör har konstruerat en modell eller ritning är nästa steg att utsätta den för beräkningar och tester för att se om den håller måttet. Exakt hur detta går till beror av modellens art och här har ingenjörsvetenskapen samlat en stor mängd tumregler och procedurer för olika fall. Kärnan i discipliner som mekanik, hållfasthetslära och flödesdynamik består av sätt att bestämma egenskaperna hos modeller och förutsättningarna under vilka de fungerar som avsett.

På samma sätt vill vi kunna analysera en modell av ett beteende för att säkerställa att det är acceptabelt. Om vi till exempel har gjort en modell av en telefonväxel vill vi kunna fastställa att den kopplar samtal korrekt, att debiteringen fungerar riktigt etc. Exakt vad ”uppkoppling av samtal” och ”korrekt debitering” innebär kan uttryckas i en formell notation. Allra bäst vore det om det också funnes en *metod*, det vill säga en uppsättning räkneregler och anvisningar om hur de ska tillämpas, så att vi garanterat kan räkna ut huruvida modellen är riktig.

Här stöter vi dock på patrull: för det mesta saknas metoder som garanterat ger ett korrekt svar. Ingen har hittills funnit några sådana. Och vad värre är: det finns logiskt oantastliga bevis för att sådana metoder inte kan finnas. Detta är en följd av Turings resultat (1936) om oavgörbarhet av utsagor om beräkningar. Exempel på frågeställningar där vi inte kan hoppas på allmänna metoder är:

- Uppfyller systemet ett visst krav?
- Kommer systemet någonsin att slutföra beräkningen?
- Kan två samverkande program att hamna i en situation där båda väntar på det andra (ett så kallat ”deadlock”)?
- Har två system samma beteende?
- Är två krav motstridiga?

Listan kan göras lång, och det är lika illa för utsagor om beteenden i formella notationer som för program. Praktiskt taget alla sådana utsagor är *formellt oavgörbara* i den meningen att allmän lösningsmetod saknas. På nästa sida kan den matematiskt intresserade läsaren se principen för ett sådant oavgörbarhetsresultat för den enkla frågeställningen ”blir resultatet av programmets beräkning 0”. Samma typ av resonemang gäller för andra frågor. Idén liknar Gödels berömda ofullständighetsbevis (1931) som säger att inget logiskt slutledningssystem kan bevisa alla sanna aritmetiska utsagor. Detta resultat, liksom Turings, har blivit föremål för allehanda filosofiska spekulationer om den artificiella intelligensens begränsningar. Just inom vårt område, logisk analys av beteenden, är dock den rent tekniska konsekvensen betydelsefull: Att leta efter fullständiga lösningsmetoder för de problem vi vill behandla är lönlöst. Detta gäller vare sig vi vill kunna programmera metoderna så att en dator utför dem automatiskt, eller bara formulera dem så att ingenjörer kan följa dem. Det finns helt enkelt ingen entydig uppsättning räkneregler som garanterat alltid ger resultat. Vi måste hålla tillgodo med metoder som bara fungerar i vissa speciella fall.

Avgörbara formalismer

Tre vägar har hittills prövats för att komma tillrätta med oavgörbarheten. Den första innebär att man minskar språkets uttrycksmöjligheter. Om dessa beskärs tillräckligt kraftigt blir frågeställningarna avgörbara. Till exempel kan man inskränka tillståndsdiagram till att bara innehålla ett ändligt antal olika tillstånd, eller logiska formalismer så att universella utsagor (”för alla x gäller att...”) bara gäller ändliga mängder.

Nu kan i förståne sådana inskränkningar tyckas harmlösa. Oändliga tillståndsdiagram kan man ju ändå inte rita, och oändliga mängder är det ont om i naturen. Men trots det har de faktiskt stor betydelse i många sammanhang. Ta till exempel en allmän beräkningsprocedur för division av heltal. För en sådan vill man avgöra att den är riktig för *alla* heltal, dvs. för en oändlig mängd olika indata. I och för sig har varje dator en övre gräns för hur stora tal som kan behandlas, men eftersom dessa gränser är olika för olika datorer är det olyckligt om procedurens riktighet beror av en sådan inskränkning.

En annan svårighet är att även om frågeställningarna blir teoretiskt avgörbara är det tyvärr inte säkert att lösningsmetoderna är praktiskt lämpliga. En dator med 16 megabyte minne har mer än 10^{10^7} olika tillstånd; detta är ett ändligt antal men en metod som måste utgå från att söka igenom alla dessa är dödsdömd ur praktisk synpunkt, även om den i princip är avgörbar. Forskningen arbetar därför inom två områden. Det ena söker sätt att göra beskrivningar där tillståndsrummen och datamängderna inte bara är ändliga utan också ganska små, genom att utesluta onödig information och genom att dela upp beskrivningarna i hanterbara delar. Goda resultat finns här inom vissa tillämpningsområden, till exempel konstruktion av små integrerade kretsar och

Idén i ett oavgörbarhetsbevis

Antag att vi har en metod **M** som kan användas för att avgöra huruvida ett godtyckligt valt program P (med något visst indata) producerar resultatet noll eller inte. Vi ska visa att det skulle medföra en logisk motsägelse. Alltså kan ingen sådan metod finnas!

Säg för enkelhetens skull att alla programmen P som **M** ska kunna hantera har ett heltal som indata. Låt $P_1, P_2, P_3, \ldots$ vara en uppräkning av alla sådana program i programmeringsspråket. Uppräkningen är oändlig eftersom det inte finns någon övre gräns på hur långt ett program kan vara. Men poängen är att en sådan uppräkning finns — man kan till exempel ta programtexterna i längdordning och lika långa texter i bokstavsordning — så att varje program är med.
Nu kan vi göra ett program Q som utför följande tre steg:

1. Q läser in ett heltal k, och konstruerar programmet P_k.
2. Sedan beräknar Q effekten av att köra P_k med indata k.
3. Slutligen ger Q det motsatta resultatet mot P_k, dvs om P_k med indata k ger noll, så fortsätter Q med att producera resultatet ett. Om P_k med indata k inte ger noll, så forstätter Q med att ge resultatet noll.

Steg 1 är möjligt, om än omständligt, tex genom att räkna upp alla program och sluta vid k. I steg 2 behöver Q bara arbeta i enlighet med metoden **M** som vi antog finns. Poängen är att Q med indata k beter sig på precis motsatt sätt gentemot P_k med indata k, dvs det ena ger resultatet noll precis om det andra inte gör det. Detta gäller för alla tal k. Därav följer att Q inte kan vara samma program som P_k för något k (de beräknar ju olika resultat om de får k som indata), så Q självt kan inte ingå i uppräkningen $P_1, P_2, P_3, \ldots$

Men Q är ju faktiskt onekligen ett program med ett heltal (k) som indata. Därför måste Q ingå i vår uppräkning $P_1, P_2, P_3, \ldots$ som innehåller alla sådana program. Q måste alltså både ingå och inte ingå i uppräkningen. Detta är en logisk motsägelse. Något i vårt bevis eller i våra antaganden är alltså fel. En noggrann analys (som är lång och teknisk) visar att allting är oantastligt utom möjligen antagandet att metoden **M** existerar. Alltså kan inte detta antagende stämma: det finns ingen sådan metod.

av kommunikationsprotokoll. Lyckade ansatser kräver ingående kunskap om tillämpningen för att avgöra vilken information som ska ingå och hur uppdelning ska ske.

Det andra området är att uteckla metoder som effektivt kan hantera allt större tillståndsrum och större mängder. I dagsläget är det till exempel möjligt att, med datorns hjälp, uttömmande analysera tillståndsrum med uppemot en miljon tillstånd. Vissa metoder kan på speciella problem nå betydligt bättre resultat. För tio år sedan var gränsen snarare tusen tillstånd. Det är svårt att sia om utvecklingen — den beror både av tekniska landvinningar inom datortekniken som gör datorerna snabbare och av förbättringar i själva metoderna — men med samma takt kan man om tio år uttömmande analysera någon miljard tillstånd. Detta är dock så långt ifrån det verkliga antalet tillstånd i datorsystem att man fortfarande kommer att behöva dela upp beskrivningar i mindre delar och skära bort information i beskrivningarna. Även om analyssteget är helautomatiskt är det därför sannolikt att speciell träning kommer att behövas för att använda metoderna.

Avgörbara frågeställningar

En annan väg att undvika formellt oavgörbara problem är att ställa frågor som inte direkt rör "beteenden" som sådana. Istället kan man kontrollera om beskrivningen till sin *struktur* är riktigt uppbyggd. Även om detta i och för sig inte garanterar riktiga beteenden visar erfarenheten att ett stort antal fel avslöjas vid sådana kontroller. Så kan man systematiskt kontrollera att växelverkan mellan systemets olika delar följer etablerade mönster, att alla ingående begrepp ges ordentliga definitioner, att inga uppenbart onödiga delar finns etc.

Ett bra exempel på detta är den så kallade typkontroll som finns för många moderna notationer. Beskrivningarna använder ju ofta komplicerade datastrukturer; typkontrollen säkerställer att de används på rimliga sätt. Om det till exempel någonstans står att X ska ökas med ett är det rimligt att anta att X måste representera ett numeriskt värde. Om det å andra sidan står att man ska ta första bokstaven i X bör X vara en text. På detta sätt kan man klassifiera data i olika "typer" utifrån sättet de används, och kontrollera att samma datavärde inte förutsätts ha motstridiga typer på olika ställen av beskrivningen. Haveriet med Ariane 5 skulle ha undvikits om en ordentlig typkontroll hade gjorts av styrprogrammet.

Många metoder för typkontroll och andra strukturella egenskaper finns färdiga att använda idag. De låter sig lätt automatiseras, och har visat sig framgångsrika i moderna programmeringsspråk. Ändå finns ett psykologiskt motstånd eftersom programmerarna ibland upplever dem som begränsande och onödigt omständliga. Forskningen inom området arbetar exmpelvis med typsystem för avancerade objektorienterade formalismer, och metoder som tillåter att värden kan ha olika

Automatisk verifiering

Om en formell modell ska kunna utsättas för helt automatisk analys får den inte vara för stor, eftersom beräkningarna annars tar för lång tid. En vanlig automatisk analysmetod är att gå igenom alla ”tillstånd” som modellen kan anta. Tillstånden bestäms, grovt uttryckt, av de möjliga värdena på de ingående variablerna. Betrakta följande lilla program som beräknar kvoten av två heltal:

```
input X;
input Y;
output X/Y;
```

I en modell av detta finns ett tillstånd för varje kombination av värden på `X` och `Y`. Om inget är sagt om hur stora talen får vara är alltså tillståndsmängden oändlig. Om vi däremot begränsar storleken, till exempel så att talen ligger i intervallet 0–255, finns precis $256 \times 256 = 65536$ olika tillstånd. Tillståndsmängden ökar med antalet olika värden på variablerna, och också med antalet variabler i modellen.

System som unviker omfattande numeriska beräkningar kan ofta modelleras utan att tillståndsmängderna blir ohanterligt stora. Flera datorprogram klarar idag av automatiska analyser av miljontals tillstånd. Exempelvis kan nämnas SPIN, utveckat vid AT&T, som framgångsrikt använts bland annat vid utvecklingen av telefonväxlar. SPIN har ett grafiskt användargränssnitt och är väl dokumenterat. Vid Carnegie Mellon universitetet har man nyligen med ett annat program automatiskt verifierat en krets för 64-bits division. Bägge analysmetoderna bygger på listiga sätt att representera stora tillståndsrum.

Mer information om SPIN finns på hemsidan
`http://netlib.bell-labs.com/netlib/spin/whatispin.html`.
Mer information om forskningen vid Carnegie Mellon finns på:
`http://www.cs.cmu.edu/~modelcheck/`.

På bägge ställena kan analysprogram hämtas gratis.

typer i lägen där detta kan ges en rimlig innebörd.

Bevismetoder

Den tredje vägen består helt enkelt i att acceptera att en analysmetod är ofullständig och därmed inte är garanterad att nå ett svar. Det finns nämligen metoder som är riktiga i den meningen att alla svar som kommer fram är korrekta. Det är inte säkert att metoden ger något svar alls, men *om* den gör det är svaret korrekt.

Dessa metoder är baserade på bevissystem i formell logik och principen är att hitta bevis i speciella logiska notationer. Frågeställningen måste då först formuleras inom en sådan notation. Om man hittar ett bevis är frågan avgjord. Om inte, kan detta antingen bero på att inget bevis finns, eller på att ett bevis finns men att man har misslyckats att hitta det. Det är omöjligt att veta vilket av alternativen som är fallet eftersom det inte, på grund av oavgörbarheten, kan finnas någon allmän strategi för att finna bevis.

En fullständig automatisering av sådana metoder är inte att rekommendera med dagens teknologi. När det gäller att leta bevis behövs listighet, förmåga att se mönster, och allmän erfarenhet. Detta kan man träna ingenjörer till men det är svårt att programmera datorer att prestera bra. Dessutom är resultatet av en bevisföring ibland mer än det summariska ”utsagan är sann”; ofta når man också en intuitiv insikt om *varför* utsagan är sann. Denna kan vara värdefull i många sammanhang men är svår att representera i en dator.

Fullständiga bevisföringar blir till sin längd jämförbara med beskrivningarna som analyseras, eller ofta till och med ännu längre. Om sannolikheten att ett misstag begås är proportionell mot längden så är alltså egentligen mycket lite vunnet med en bevisföring! Vinsten ligger i att ett bevis kan kontrolleras på ett mekaniskt sätt. Så även om datorer inte själva kan utföra bevisen kan de i efterhand kontrollera att inget misstag begåtts. Det finns idag flera program för detta och till och med program som hjälper till lite under själva bevisföringen.

Formella logiska notationer och bevismetoder har studerats åtminstone sedan Freges arbeten i slutet av 1800-talet. Forskningen har nu den stora utmaningen att överföra bevissystem till notationer som ingenjörer är mer vana att behandla, och att utveckla praktiska strategier som med viss automatisk hjälp hittar bevis. Svårigheterna ökar med den formella beskrivningens längd, därför är uppdelning av stora beskrivningar viktigt även här.

Syntesmetoder

Slutligen vill ingenjören kunna använda sina ritningar och modeller som underlag för konstruktion, och då ha förtroende för att de föremål som produceras faktiskt

Logiska korrekthetsbevis

Trots att bevis i allmänhet inte kan utföras automatiskt finns flera exempel på att datorer kan användas för att underlätta logisk analys genom att kontrollera att alla steg i beviset är riktiga, och föreslå strategier under det att beviset konstrueras.

Vid Stanford Research Institute utvecklar man till exempel sedan 15 år PVS, Prototype Verification System. Liksom alla liknande system kräver det att användaren behärskar en speciell notation:

```
remainder_bound_equiv_legitimate:  LEMMA
 (FORALL (p_new, p, q, d):
     recurrence?(p_new, p, q, d) IMPLIES
       (remainder_bound?(d, p_ new)
        IFF legitimate?(q, d, p)))
```

Detta är från en del av ett korrekthetsbevis för den del av en processor som utför division. Hade Intel genomfört ett sådant (vilket är fullt möjligt med dagens teknologi) skulle felaktigheten i de första pentiumprocessorerna förmodligen avslöjats!

Nackdelarna med en sådan metod är att notationen är svårgenomtränglig och bevisstegen många. Detta gäller även mer industriellt anpassade system, som exemplevis *B-metoden*, som har en välutvecklad flora av olika datorhjälpmedel för storskalig konstruktion, många med avancerade grafiska gränssnitt. B-metoden blev berömd när järnvägssignalerna för pendeltågen i Paris programmerades om för att öka tågtätheten i slutet av 1980-talet. Nio tusen rader kod analyserades logiskt för att bevisa riktigheten av kritiska delar. Den ansvariga myndigheten i Paris krävde en sådan analys innan systemet fick tas i drift.

Mer information om PVS finns på hemsidan
`http://www.csl.sri.com/sri-csl-pvs.html`.
Mer information om B-metoden finns på:
`http://www.comlab.ox.ac.uk/archive/formal-methods/b.html`.

PVS är gratis medan de mer omfattande programmen runt B säljs kommersiellt.

stämmer överrens med ritningen. På samma sätt önskar vi *syntesmetoder* som utifrån en formell beskrivning producerar delarna i datorsystem, till exempel i form av program som kan köras på datorer eller kretsdiagram för integrerade kretsar. Hur lätt detta är beror av beskrivningens art.

Utifrån operationella beskrivningar är det ofta enkelt att automatiskt generera programkod. Operationella beskrivningar anger ju exakt vad som ska hända vid en körning, samma sak som i princip anges av ett program. Problemet är här snarast att översätta mellan två notationer, från ett formellt språk till ett programmeringsspråk; sådana översättningar är inte svårare än de som rutinmässigt utförs av programmeringsspråkens kompilatorer som genererar körbar kod utifrån ett program. Kompilatorer har funnits i runt 40 år och erfarenheten hur man bygger effektiva sådana är stor.

Ett problem i detta sammanhang är dock att de flesta programmeringsspråk saknar en matematiskt exakt semantik, det vill säga en definition av hur programmen i språken motsvarar beteenden. Detta bestäms oftast pragmatiskt, genom att ge exempel på program och deras beteenden. För oklara fall hänvisas till "rikslikare" (eller snarare "världslikare"), ungefär som begreppet "en meter" tidigare definierades som längden av en viss stång i Paris. Programmeringsspråkens rikslikare är utvalda kompilatorer. Detta hjälper föga för att upprätthålla en matematisk stringens i alla led av en konstruktion. Det försvårar också möjligheten att använda analysmetoder direkt på programmen.

Forskningen utvecklar här metoder för att ge semantik till programmeringsspråk. Området har funnits i 35 år men har hittills haft liten effekt i industriell tillämpning eftersom definitionerna anses stora och klumpiga. Det finns syntesmetoder som används industriellt idag. Trots att de knappast kan anses matematiskt korrekta är de framgångsrika för att snabbt konstruera prototyper.

Syntes utifrån deklarativa beskrivningar är ett intressantare område eftersom de lättare formulerar krav på vad som ska utföras utan att i detalj ange hur. Sådan syntes är betydligt svårare eftersom strukturen på en konstruktion inte nödvändigtvis följer det sätt som kraven är formulerade, och det saknas allmängiltliga resultat även om vissa specialfall har behandlats med framgång. Mest känt är att språket PROLOG och dess efterföljare kan användas att för att generera körbar kod motsvarande en del deklarativa utsagor.

Sammanfattning

Forskningen i området har utvecklats mycket starkt de senaste tio åren och utgör idag en stor del av datalogisk forskning överhuvud taget. I Sverige finns den i olika former representerad vid tex. KTH, Chalmers, Uppsala universitet och SICS. Man studerar matematiska principer bakom formella notationer, möjligheter att beskriva fler aspekter av beteenden, bättre automatiska analysmetoder och mer anpassade bevissystem. Ett betydande allmänt problem inom den tillämpade forskningen är uppdelning av komplicerade beteenden så att delarna kan analyseras separat.

Den allra största delen av industriell produktion sker fortfarande utan användning av formella beskrivningar. Dock har de senaste fem åren vissa betydande insteg skett. Framför allt används då formella metoder för att upprätthålla entydiga beskrivningar av konstruktioner. Analysmetoder används relativt lite och då mest i form av helt automatiserade procedurer. Syntesmetoder från operationella beskrivningar är inte ovanliga även om dessa sällan gör anspråk på matematisk korrekthet. Intressant nog har en del statliga myndigheter börjat kräva att formella metoder ska användas vid säkerhetskritiska områden som styrning av kärnkraftverk och militära system. I Sverige har dock industrin med några få undantag förhållit sig ganska kallsinnig.

Det finns flera rapporter som beskriver industriella erfarenheter av formella metoder. Så visar det sig till exempel att de mycket väl kan ha god effekt även utan automatiska analysmetoder. Kvaliteten på produkterna bedöms öka liksom kontrollen över produktionsprocessen, däremot minskar inte kostnaden eller utvecklingstiden. Detta kan bero på att utbildningsinsatser är nödvändiga. För att citera ur en av de mest omfattande (300 sidor) och vederhäftigaste översikterna av hur formella metoder påverkat industriell produktion:

> A major conclusion of the study is that formal methods, while still immature in certain important respects, are beginning to be used seriously and successfully by industry to design and develop computer systems.[3]

Olika meningar råder om vad som är viktigast för att driva området framåt. En del hävdar att framtiden ligger i att utveckla helautomatiska analys- och syntesmetoder till den grad att de kan tas i bruk utan nämnvärd träning. Mot detta kan invändas att även om analyssteget är automatiskt är modelleringssteget det inte, och modeller för helautomatiska analyser kräver extra omsorg för att minska komplexitetsgraden. Andra menar att en stor utbildningsinsats är nödvändig för att lära ingenjörer att genomföra logiska bevis; på samma sätt som stora företag har teknikspecialister inom viktiga områden skulle de kunna ha ”bevisspecialister” i utvecklingsavdelningarna. Slutligen några ser att ”formal methods light” är den framkomligaste vägen, där man framför allt använder beskrivningarna för att uppnå entydighet, utan att göra anspråk på helt fullständiga beskrivningar, och gör analyser bara där det finns metoder som kan användas utan ansträngning.

För att området ska få större effekt industriellt tror jag, framför allt, att man måste hitta sätt att föra in metoderna i de konstruktionsprocesser som

[3] Dan Craigen, Susan Gerhard och Ted Ralston: *An international survey of industrial applications of formal methods*, US Department of Commerce, NISTGCR 93/626 1993, sid ii.

redan används idag. Formella metoder kan aldrig ensamma garantera att datorsystem beter sig korrekt. Därtill är felkällorna för många. Även en formell beskrivning kan vara felaktig i den meningen att den inte representerar vad konstruktören avsett. Analysproblem kan formuleras felaktigt. Analysmetoder, även automatiserade sådana, kan innehålla fel och analysresultat kan tolkas felaktigt.

Vinsten med formella metoder är, liksom all ingenjörskonsts ritningar och modeller, att de minskar sannolikheten för vissa typer av fel, att de dokumenterar konstruktioner och förhållandet till de krav som ställs på dem, och att de ökar precisionen i samtal mellan konstruktörer. Delvis finns sådana metoder redan färdiga och den vetenskapliga utvecklingen inom området är stark. Hur de ska tas i bruk mer allmänt är dock fortfarande en öppen fråga. Kanske är denna tröghet oundviklig:

> Engineers, the infantry of every industrial revolution, do not spontaneously generate. They are trained out of the bad habits developed by the craftsmen that preceded them. Until the lessons of computer science inculcate a desire not merely to build better things but also to build things better, the best we can expect is that software developers will undergo a slow, and probably painful, industrial revolution.[4]

Två år efter dessa ord exploderade Ariane 5, säkerligen inte det sista av IT-revolutionens olycksbarn.

[4] W. Wayt Gibbs, Software's chronic crisis, *Scientific American* september 1994, sid. 81.