Testning av inbyggda system

Lars Fröjdh lars.frojdh@gmail.com

2022-09-14

Innehåll

Testning av inbyggda system

Introduktion

Syftet med ett test är, i min mening, att exekvera ett stycke mjukvara för att demonstrera en observerbar egenskap. Den observerbara egenskapen kan till exempel vara ett resultat av en beräkning, logiska operationer eller kanske parsning av en token. Egenskapen behöver i sig inte vara observerbar, utan syftet med test är att göra den observerbar så att man kan resonera om egenskapen[1].

Inbyggda system är hårt knutna till plattformen de är implementerade i; man kan resonera huruvida sedvanliga test-metodologier är pragmatiska. Allt som ofta är kravbilden ställd såpass annorlunda att man tvingas att tänka om sin test-strategi.

Till exempel så är oftast test-cyklerna väldigt långa och uppdateringar kostsamma. Det kan också vara svårt att utveckla test för funktioner som är *djupt inbäddade* i ett system.

Syftet med det här dokumentet är att visa på ett antal olika pragmatiska metoder som kan användas för att säkerställa funktionen hos ett inbyggt system. Metoderna som diskuteras är:

1. Embedded differentiell testing
2. Model validation / cross-target testing
3. Propositionell testning

Det här dokumentet är ingen manual - utan snarare ett underlag för diskussion av praktiska aspekter relaterade till kostnaden av komplicerad mjukvarutestning.

Embedded differentiell testing (EDT) som metod tänker om väldigt mycket vad test är för något och kan uppfattas som kontroversiell. Men innan vi diskuterar det bör något sägas om automatiserade tester.

CI/CD pipeline

Betydelsen CI/CD (Continous integration / Continous delievery) pipelines för mjukvaruuveckling kan inte överskattas. För ett inbyggt system är det kritiskt att ha en CI/CD pipeline på plats.

När man har en sådan på plats - då är det underförstått att man kan, på ett programmatiskt sätt:

1. Extrahera data som diagnostik från ett inbyggt system
2. Genomföra uppdateringar av mjukvaran av ett inbyggt system

[1] Det omvända förhållandet gäller även alltså; kan man inte göra en egenskap observerbar så kan man inte heller testa den.

Om man inte kan göra dessa två saker blir man lätt blind för indeterministiska fel som inträffar med låg frekvens. Med en bra integration kan man lättare skatta fel frekvensen mot kostnaden av en defekt och därmed utforma en bättre underhållsplan.

Manuell testning är tidskrävande och sällan uppskattat - en felkälla i sig. Erfarenhetsmässig så upplever man oftast avsaknaden av en pipeline som ett problem än det omvända.

System testning i en CI/CD pipeline ger en god konfidens hur ett system fungerar på makro-nivå. Det går inte att ersätta ett system test; men kombinerat en väl genomtänkt test-strategi på unit-nivå kan man erhålla en god kostnadseffektiv kombination.

Hur man genomför testning på unit-nivå leder diskussionen onekligen till embedded differentiell testning.

Embedded Differentiell testing

Differentiell testning innebär att man jämför en tidigare representation av observerbart resultat med nuvarande för att bättre *tolka* vad en kodförändring har för effekt.

Embedded differentiell testing (EDT) är en metod där man har anpassat differentiell testning till inbyggda system. EDT har låg underhållskostnad och lämpar sig väl till komplex mjukvara där traditionell unittestning inte fungerar särskilt väl. Metoden fungerar också ypperligt i CI/CD-pipelines.

EDT kan förklaras med tre steg:

```
1. gcc -DDEBUG bar.c -o bar
2. ./bar > bar.txt
3. git diff
```

I det första steget har vi instrumenterat källkodsfilen bar.c till att inkludera spårutskrifter, så kallade edt-traces, i sin körning. Det enklaste sättet att generera trace-information är att lägga in debug-makron i koden. I exemplet innehåller bar.c data för att kunna köras självständigt på ett deterministiskt (idempotent) sätt.

I det andra steget exekverar vi testapplikationen och sparar trace-informationen i en text fil som vi har under revisionskontroll. Givetvis är även källkodsfilen under revisionskontroll.

I det tredje steget frågar vi revisionskontroll-systemet (git i exemplet) vilka ändringar av koden vi har och hur de påverkar det observerbara resultatet i form av differenser av edt-traces.

Om utvecklaren ändrar definitionen av funktionen double kan resultatet efter en körning se ut så här (main i bar.c innehåller ett anrop i stil med *printf(”The result of double(2) -> %i”, double(2);*):

```
$ git diff

--- a/bar.c
+++ b/bar.c
int double(int f)
{
-   return f;
+   return f * 2;
}

--- a/bar.txt
+++ b/bar.txt
- The result of double(2) -> 2
+ The result of double(2) -> 4
```

Här kan man överblicka det observerbara resultatet av testkörning i form av edt-traces tillsammans med kodändringarna. När man tittar på den totala differensen ser vi att korrigeringen är *konsistent* med vad vi förväntade oss observera.

Utvecklaren bokför kodändringen med differensen av resultatet i revisionskontrollsystemet.

Instrumentering för EDT

För att överhuvudtaget har något att jämföra med måste applikationen under test generera trace-information som man kan observera och resonera kring. Det enklaste sättet att göra det är att inkludera det direkt i källkoden, till exempel:

```
int main(void)
{
  DEBUG("The result of double(2) should be 4 -> %i", double(2))
}
```

Syftet med ett test är att exekvera ett stycke mjukvara för att få den att demonstrera en observerbar egenskap. Ett enkelt sätt att uppnå detta är att mjukvaran skriver ut en debug utskrift som demonstrerar att en *observerbar egenskap* har inträffat. Allt som ofta har man redan gjort det som ett led av den initiala utvecklingsfasen.

För att instrumenteringen skall vara meningsfull måste man givetvis kunna exekvera mjukvaran på ett idempotent (deterministiskt) sätt och kunna samla trace-informationen i filer. Det enklaste sättet att göra detta är givetvis exekvera delar eller hela mjukvaran i samma host-miljö som utvecklingen sker i.

Det innebär att åtminstone delar av mjukvaran är portabel. Numera är det inte särskilt stor ansträngning att göra det, och när man väl har genomfört det så kan man öppna upp mjukvaran för annan sort instrumentering som till exempel statisk analys. Man kan även vid det här laget göra cross target validering vilket förtjänar en diskursion i sig själv.

Det viktiga när man skapar trace-makron är att det är meningsbärande för den som implementerar mjukvaran i fråga. Det vill säga när utvecklaren ser trace-informationen hjälper det utvecklaren att *tolka om intentionen* med koden ifråga är konsistent. Till exempel har man i DEBUG-makron ovan lagt till annotationer som underlättar tolkningen av det beräknade resultatet.

Vid det här laget borde det vara ganska givet att EDT primärt är ett verktyg som skall stöjda utvecklaren i sitt arbete med att underhålla samt utforma mjukvara.

För mjukvara som inte har inbäddade spårutskrifter kan man använda källkodsinstrumentering med clang-kompilator ramverket för att modellera spårutskrifter ovanpå mjukvaran. Mer om det senare.

Exekvering av test

Givet att man har implementerat delar av mjukvaran portabelt kan man exekvera den i utvecklingsmiljön och spara resultatet i en trace-fil:

```
$ ./bar > bar.txt
```

I det här fallet innehåller binären all data så att beräkningar kan ske på ett idempotent sätt. För mer komplex mjukvara där man vill titta på en observerbar egenskap åt gången är det önskvärt att lägga test-data utanför så man kan få mer precis exekvering (alternativt skapa många test binärer):

```
$ ./bar input-1.txt > output-1.txt
```

I fallet av en parser så kanske man tittar på hur en typ av token processeras.

Tyvärr i fallet av fördefinierade debug makron så faller man lätt i en situation av all-or-nothing när man behöver trace-information. I vissa fall har man för lite eller rentutav för mycket information - källkodsinstrumentering är en metod att komma runt det problemet.

Differentiering av test resultat

När exekveringen av test är genomförd så måste man utvärdera resultatet. Då vi har valt att hålla trace-informationen och källkoden under revisions-kontroll (i git) kan man enkelt se allt resultat:

```
$ git diff
```

```
--- a/bar.c
+++ b/bar.c
int double(int f)
{
-   return f;
+   return f * 2;
}

--- a/bar.txt
+++ b/bar.txt
- The result of double(2) -> 2
+ The result of double(2) -> 4
```

Här kan utvecklaren se dels kodändringen men även hur det differerar test resultatet. Förutsatt att uvecklaren har med omsorg valt trace-information underlättar det för utvecklaren att *tolka* eller interpretera resultatet. Utvecklaren kan med lätthet se att kodändringen med differensen på testresultatet är *konsistent*.

Om utvecklaren drar slutsatsen att kodändringen och förändringen av testresultatet är konsistent skall utvecklaren bokföra det i revisionssystemet. I och med att alla ändringarna är bokförda kan man alltid gå bakåt och utvärdera (*omtolka*) slutsatsen. Man kan då utvärdera om slutsatsen man gjorde var konsistent med den observerbara egenskapen man skulle testa. Denna tolkning kan man göra direkt utan en ny test exekvering.

Arbetsflöde (workflow)

Utvecklaren som vill demonstrera en observerbar egenskap, som underförstått hen har ändrat, exekverar test applikationen och tittar på differensen av testresultatet. I den har situationen kan tre fall inträffa:

1. Kodförändringen manifesteras *inte* som differans av EDT-trace.
2. Kodförändringen manifesteras som en differans av EDT-trace.
3. Kodförändringen manifesteras som en alldeles *för stor* EDT-trace.

I fall 1 så kan man misstänka flera saker. Det kan vara så att test applikationen inte har path coverage för egenskapen vi förändrar. Det kan också vara så att förändringen vi genomför inte påverkar någon observerbar egenskap - det är det önskvärda fallet för faktoriseringar.

Fall 2 är normal fallet då kodförändringen manifesteras i differensen av ett EDT-trace. Utvecklaren kan då tolka om kodändringen är konsistent med förändringen av tracet. Om förändringen är konsistent och önskvärd bör utvecklaren bokföra ändringarna så att de kan utvärderas på nytt i framtiden. Det kan också visa sig att man ser differens som man inte förväntar sig; då utmanas utvecklaren att tolka kodändringen ånyo.

I fall 3 är differensen av EDT-tracet för stort för att man på ett säkert sätt kan tolka differensen. Här kan utvecklaren välja att faktorisera testet av den observerbara egenskapen. Det kan också vara så att kodförändringen påverkar andra observerbara egenskaper än de som man avsedde i första hand. Då kan man relativt enkelt detektera degraderingar av funktionalitet. I prestanda kritiska applikationer kan man i fall 3 relativt lätt detektera degraderingar.

De här tre fallen kan sammanfattas i ett beslutsschema:

1. Ingen differens -> tolka och eventuellt expandera.
2. Differens -> tolka
3. Stor differens -> tolka och eventuellt faktorisera.

Det kritiska är att man bokför förändringarna, dels för att man i efterhand kan gå igenom de tolkningar och beslut man har gjort; men även för att sätta sitt godkännande på att de förändringar man gjort är konsistenta. Om man senare i CI/CD pipelinen exekverar EDT-tester och ändå får en differens betyder det att antigen (1) test-applikationen inte är idempotent eller att en testutvecklare har bokfört en kodförändring utan att exekverar korresponderande testapplikation (som resulterade i en faktiskt differens).

Om man håller både kodändringarna och trace-informationen under revisions-kontroll blir det trivialt att gå bakåt och ringa vilken kodändring som påverkade representationen - detta utan att man behöver exekvera ett test ånyo. Detta kan vi göra direkt i revisions-kontroll systemet med de mekanismer den erbjuder.

Semantisk interpretation

EDT som metod är enkelt att förstå och implementera. I realiteten har man oftast debug-makron som kan tjäna som EDT-traces. Och allt som ofta är delar av mjukvaran portabel vilket är en förutsättning för att exekvera idempotent.

EDT är både en testmetod och ett sätt att synliggöra observerbara egenskaper i ett stycke mjukvara, vilket man som utvecklare är tvungen att göra.

Traditionella unittest-ramverk fungerar bäst för bibliotek; merparten av mjukvara för inbyggda system är inte bibliotek. Snarare liknar det mer kontrollflöden där tillståndsövergångar kräver avsevärd instrumentering för att extraheras programmatiskt. Till exempel så är det svårt att utvärdera om alla branches i en beräkningskedja utvärderas under test. I traditionell testning måste man titta på coverage information ; men med EDT ser man direkt på trace-information om testen och trace-makrona är tillräckligt uttömmande.

EDT synliggöra information om mjukvaran och ställer sedan frågan direkt till utvecklaren; här är en representation av exekveringen - är informationen konsistent? Utvecklaren som lägger EDT-makron har en mental bild av hur representationen av exekveringen bör se ut. Frågan om representationen överensstämmer med den mentala bilden ställs på sin spets. Detta sker per automatik då hen som skapar förändringen också skapar representationen[2].

Det är svårt att programmatiskt beskriva ett korrekt resultat av beräkningar - även för så pass enkla funktioner som sortering av heltal. Men människan har en unik förmåga att tolka information, magskänslan leder hen vidare. EDT som metod försöker maximera det semantiska interpretations maximet:

[2] Sannolikt så har utvecklaren som gör förändringen en mental bild av intentionen och därav kan skapa både förändringen men även en meningsbärande representation som kan återkoppla till den mentala bilden. Tolkningen av testexekveringen kommer att visa ifall den meningsbärande representationen av förändringen är **konsistent** med den mentala bilden.

> It is hard to describe how something is broken, but it is trivial to interpret that something is broken.

Ibland räcker det att titta på en differans för att förstå att något inte stämmer, att däremot förstå exakt vad som är felet är betydligt svårare. Det är också något som utvecklaren av ett stycke mjukvara kan göra då hen har intima kunskaper om detaljerna. Många gånger är naturen för en förändringen mer intressant än differensen i sig. Till exempel med faktoriseringar så kan man förvänta sig att differensen inte alltid skall bli särskilt stor. Om så inte är fallet kan det indikera en inkonsistens. I traditionell unittestning är det bara det programmatiska resultatet som är signifikant.

Debuggning är en interaktiv process där utvecklaren söker att minska gapet mellan vad mjukvaran faktiskt gör och vad den borde göra. Processen är riktad både inåt och utåt[3]. EDT tvingar utvecklaren att tolka vad mjukvaran enligt en meningsbärande representationen faktiskt gör. Traditionell unittestning ger bara en programmatiskt representation av ett resultat.

Det paradoxala med EDT är att det ger både mer information och kräver betydligt mindre av utvecklaren. Ingen integrering av UT-ramverk, ingen programmatiska analysering av resultat - bara att man visualiserar processering av data för tolkning.

Kompilator instrumentering

För större applikationer är kompilator instrumentering nödvändig för att EDT skall fungera effektivt. För enklare applikationer räcker det med enklare makron för att få tillräckliga trace-information:

```
int double(int f)
{
    DEBUG("double(%i) -> %i", f, f*2);
    return f * 2;
}
```

För att kunna demonstrerar många olika observerbara egenskaper i större applikationer kommer trace-informationen från debug-makrona att överlappa varandra. Vi hamnar i en all-or-nothing situation där vi får för mycket information för att effektivt kunna tolka trace-informationen.

Lösningen är att man före kompileringen av test-applikationen kan välja vilka debug makron som är aktiverade. Till exempel:

```
$ edt -a double
```

```
alternativt:
```

[3] Utvecklaren söker att både förstå vad mjukvaran gör genom konstruktion av mentala bilder; men även genom att förstå hur mjukvaran fungerar i en större kontext av dess användare och förväntningar.

```
$ edt -a double '"double(%i) -> %i", f, f*2'
```

Här har man valt att aktivera trace-informationen i funktionen double. Genom att aktivera trace-information i ett subset av alla funktioner i ett stycke mjukvara kan man erhålla rätt mängd information för effektiv tolkning.

För att kunna dynamiskt enabla trace-makron måste man kunna peka ut vars och hur man vill enabla makron. Det enklaste sättet är att peka ut en funktion - då funktioner är rigida designatorer som oftast följer med en implementation av en feature. Antigen kan man aktivera en definerad makro i en funktion eller injecta en i funktions-kroppen. I det senare fallet kan man göra en approximation från de primitiva typerna i en funktions-signatur.

Att instrumentera koden på ett säkert sätt kräver kompilator stöd. Clang kompilator infrastruktur erbjuder sådan instrumentering då man med lätthet kan skriva sådana source-transformator applikationer. Under utvecklingen måste man givetvis skapa bygg regler som sparar resultatet av trace-informationen i unika filer; detta för att testet skall kunna mergas med en huvudbranch på ett meningsfullt sätt.

Givetvis skall man även kunna disabla ett EDT-makron före kompilering:

```
$ edt -d double
```

```
alternativt:
```

```
$ edt --clean
```

Det går att åstadkomma EDT-profiler med debug-patchar för features. Tyvärr är det svårt att underhålla patchar då konflikter lätt kan uppkomma. Ett specifikt EDT-verktyg är inte lika känslig för konflikter. Det är också enklare och säkrare för testare att använda. Man kan också i en CI/CD-pipeline använda ett sånt verktyg för att ta fram logg-filer för applikationer med komplicerade tillstånd för lättare analys.

Avancerad instrumentering

EDT-makron kan integreras mer direkt till testsystemet. Till exempel så finns det inget som säger att DEBUG-makron nedan endast är ett maskerat printf/putc:

```
int double(int f)
{
    DEBUG("double(%i) -> %i", f, f*2);
    return f * 2;
}
```

En ganska enkel förbättring man kan göra är att makro skriver ut __FILE__:__LINE__ tillsammans med annotationen:

```
-bar.c:10:double(2) -> 2
+bar.c:10:double(2) -> 4
```

Annotationen som fått extra information som senare kan användas för att deducera vilka testfall som sannolikt skall exekveras. För att det skall fungera måste testsystem hålla reda på vilket testfall som generera vilka annotationer.

```
$ edt --run bar ${test_case}.txt ${test_case}.output
```

edt[4] bygger en databas vilka källkodsrader som sannolikt svarar mot vilka testfall. När väl en kodändring snappas upp av testsystemet kan den köra testfallen en prioriterad ordning som fortare hittar defekterna.

Med det här schemat kan man köra testfall i kritiska segment oftare vilket är viktigt för systemtester.

Att generera coverage information är ett ganska enkelt schema förbättrar testsystemet; det finns givetvis mer avancerad instrumentering som man kan åstdakomma. För beräkningskedjor kan man inkludera komplett beräkningsinformation i annotationerna. Då kan man mata EDT-traces till en extern modell som kan kors-verifiera de beräknade värden.

Sammanfattning

EDT är en metod som både kan användas för att testa mjukvara med komplexa kontrollflöden och underlätta för mjukvaruutvecklaren att tolka mjukvaran. Metoden är enkel att genomföra och kräver egentligen bara att man har integrerat revisions-kontroll utöver sedvanliga verktyg.

EDT flyttar fokus från hur test-ramverk fungerar/instrumenteras och fokuserar på hur utvecklaren på ett effektivt sätt skapar en meningsbärande representation för vad mjukvaran *gör* och hur det *tolkas*.

EDT är ett arbetssätt för utvecklaren så att denne på ett pragmatiskt sätt kan arbeta för att kunna *testa och tolka* mjukvaran som hen ansvarar för.

EDT som metod har hämtat inspiration från filosofen Ludwig Wittgensteins bildteori för korrelation mellan språk och verklighet som beskrivs i hans verk ”Logisch-Philosophische Abhandlung”.

Tolkningen kommer först.

[4] Man bör implementera funktionaliteten som en applikation/script - detta för att lättare kunna särskilja mellan olika fel tillstånd. Till exempel att binären kraschar, inte genererar någon utdata eller inte är idempotent vid repetition.

Modell validation

EDT fungerar utmärkt när utvecklaren, eller test-ingenjören, kan rimligen tolka om en beräkning eller tillstånd är konsistent. För mer komplexa fall är det väldigt svårt att göra en sådan tolkning. För att komma runt det problemet kan man implementera en modell som man integrerar i sitt test-system.

Modellen fungerar som ett test orakel som givet korrekt indata kan producera korrekt utdata. När man utvärderar om en beräkning är korrekt har man valet att jämföra med känd utdata eller en modell som man litar på - skillnaden är inte så stor. Modellen kan också ge ytterligare information som annars hade klottrat ner loggar - till exempel medelvärden, standard-avvikelser etc.

Modell validering för inbyggda system

När man implementerar en modell, eller referens-implementation som det ibland kallas, har man i princip två val:

1. Man implementerar modellen i samma språk som det inbyggda systemet.
2. Man implementerar modellen i ett högre nivå språk.

Fall 1 har den uppenbara fördelen att det är mindre att implementera. Vi kan använda återanvända kod för att utveckla test-oraklet. Nackdelen är att test-oraklet inte blir en fristående implementation, fel kan smyga sig in i både test-oraklet och target.

Att implementera en referens implementation i ett högre språk, som i fall 2, kräver mer av utvecklaren men har långsiktigt betydligt fler fördelar. Man kan implementera problem-domänen, till exempel en beräkningskedja eller parser, i ett språk där problemet kan uttryckas deklarativt. Modellen kan sedan användas för att resonera kring problem-domänen på ett helt annat sätt än den faktiska implementationen.

För kritiska system realiserar man sannolikt båda angreppssätten då de kan appliceras i olika steg i en CI/CD-pipeline. Till exempel så kan en fall 1 modell användas som ett regressions-test medan en fall 2 modell kan kanske användas för att utvärdera mätningar gjorda på target.

Implementera en funktionell modell

För att implementera en modell som går att lita på rekommenderar jag att man implementerar den i ett hög nivå språk som Haskell. Haskell är ett deklarativt typ-deducerande starkt typat språk som lämpar sig för att beskriva beräkningar, parsers och kompilatorer. Den starka typningen i kombination med det rika typ systemet gör det möjligt att skriva effektiva implementationer som garanteras konsistenta av kompilatorn. Ett exempel:

```
data Color = Red | Green | Blue

isGreen Green = True
isGreen _     = False
```

Den första deklarationen definerar en datatyp som heter Color (som består av tre värden - Red|Green|Blue) och en funktion som returnerar True fall input-färgen är Green (i andra fall returnerar funktionen False). Typ-signaturen för funktionen isGreen är följande:

```
>:t isGreen
isGreen :: Color -> Bool
```

Funktionen isGreen ser ut som en deklaration men är egentligen ett konditionellt uttryck, om värdet är Green så returneras True - i andra fall False. Kompilatorns uppgift här är att garantera att datatypen Color och funktionen isGreen används på ett konsistent sätt. När man skriver ett program i Haskell måste man deklarera följande saker:

1. Datatyperna som beskriver data i problem-domänen.
2. Funktioner som manipulerar data i problem-domänen.

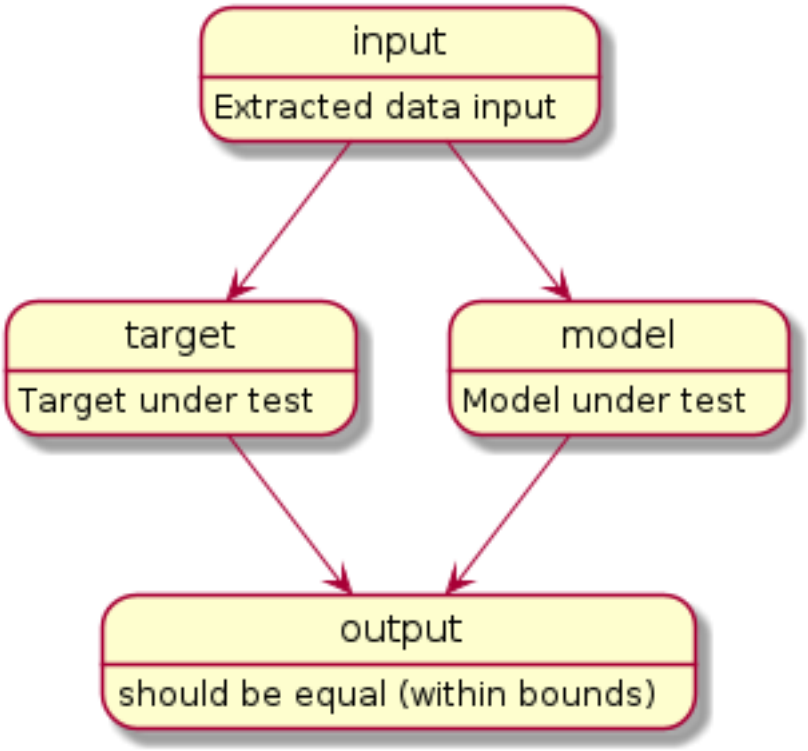
Då man beskriver relationer, inte sekvenser av beräkningar, behöver man beskriva färre saker; Kompilatorn med dess typ kontroller garanterar att relationerna är konsistenta.

Detta göra att Haskell är ett utmärkt val för att beskriva modeller. Deklarationer för ett parametriskt binärt sökträd kan se ut så här (definitioner utelämnade för läsbarhet):

```
data Tree x = Leaf | Node x (Tree x) (Tree x)

insert v tree = undefined
remove v tree = undefined
find v tree = undefined
size tree = undefined
flatten tree = undefined
```

Model validation / cross-target testning



Det bästa sättet att använda en modell i ett test-system är som en cross-target oracle. Till exempel så kan ett inbyggt system generera beräkningar, och förutsatt att man kan extrahera såväl indata som färdiga beräkningar kan man köra samma data genom modellen. Alla skillnader i beräkningar som inte kan förklaras av precisions-förluster kan indikera ett fel i antigen modellen eller systemet under test.

Modellen kan även visa annan data i beräkningar, till exempel standard avvikelser som kan behövas under utvecklingsfasen. Att ha en färdig modell som kan utföra sådana visualisera sådana beräkningar är oerhört instrumentellt speciellt för system i komplexa lab-miljöer.

För att illustrera differentiellt följer man dessa tre steg (DUT - device under test):

```
1. DUT > data.txt
2. ./model data.txt > model-data.txt
3. diff data.txt model-data.txt
```

DUT genererar en komplett beräkningskörning som innehåller både input-data samt uträknat data. Datat från den körningen matar man till modellen (eller test-applikationen) som beräknar värdena på nytt. Sedan differerar man de båda körningarna och kollar om det är någon skillnad.

Om man har en skillnad betyder det antigen att det finns ett fel i firmware eller modeller och/eller så är inte beräkningarna i target idempotenta.

Sammanfattning

Alla angreppssätt till testning skall utvärderas utifrån vilken ansträngning mot vilka fördelar det medför, okonventiell eller ej. Tyvärr faller man i fällan att implementera en stor mängd unittester där en korrekt implementerad modell hade varit en mer flexibel lösning.

Modellen kan i många fall tjäna som ett verktyg för att resonera kring problem-domänen, detta då modellen oftast är befriad från implementations detaljer som hör till target. Modellen kan också tjäna som en inkörsport för nya utvecklare i ett projekt. Man kan också använda modellen för att utvärdera nya sätt att utföra beräkningar utan behöva testa på target.

En flexibel modell kan användas för att utvärdera data från fält (förutsatt att vi har all indata för beräkningar) eller rentutav bli ett fristående verktyg som man kan ge fält-ingenjörer. I fall 2 modeller (helt fristående implementation) kan det vara möjligt att re-distribuera dem.

Man skall definitivt överväga att implementera modellen i ett starkare typat språk, detta för säkerställa att modellen är korrekt.

Det kan tyckas onödigt att implementera en modell, men då glömmer man av att det oftast är bara en liten definierad del av koden som oftast står för för merparten av defekterna. Dessutom hur skall man utvärdera ifall 0.3183098861837907 är ett korrekt värde? Om man har en deklarativ modell som producerar samma värde kan man luta sig mot att det nog är korrekt.

En bra modell kan användas för att definera rand-villkor i implementationen och hitta korrekt testdata som kan användas på den faktiska implementationen. Detta leder oundvikligen vidare diskussionen till propositionell testning.

Propositionell testning

Propositionell testning kallas även för parametariserad testning och är en teknik som kan lösa ett problem med konventionell unit-testning. I PT (propositionell testning) beskriver man en egenskap (proposition) som funktionen under test skall uppfylla och sedan försöker man genom att använda kombinator bibliotek hitta data som invaliderar propositionerna. Propositionell testning söker att stödja utvecklaren att hitta rand-villkor för kod under test.

Propositionell testning är mycket enklare än det låter; ponera att vi har implementerat en funktion som kan sortera heltal i numeriskt stigande ordning. Man kan anta att den funktionen uppfyller några propositionella egenskaper:

1. Stoppar man in n-stycken heltal i funktionen; skall den returnera n-stycken heltal.
2. Funktionen skall returnera heltal i numeriskt stigande ordning.
3. Stoppar man in ett heltal i funktionen skall funktionen också returnera en lista innehållande det heltalet.

Givet att man programmatiskt kan beskriva propositionerna kan ett PT-bibliotek hjälpa utvecklaren att hitta data som invaliderar (falsifierar) propositionerna och därmed upptäcka defekter i implementationen.

Implementera propositionell testning

Propositionell testning med hjälp av kombinatorer skapas med fördel med QuickCheck som är ett bibliotek för Haskell. Till exempel så kan propositioner för ett binärt träd se ut så här (förkortade för läsbarhet):

```
-- insert n-items in the tree ; the tree should have size n.
prop-insert items =
    size items == size (flatten | map (\item -> insert ...

-- insert n-items in the tree it should contain those n items.
prop-exists items =
    and $ map (\item -> find item tree ...) items

-- insert n-item in the tree - the result is an ordered tree.
prop-tree items =
    and $ map ..
```

Med de här tre propositionerna kan QuichCheck söka efter test-data (*items*) som invaliderar propositionerna. QuickCheck kan i många fall söka att reducera och hitta ett minimalt test-set för en falsifierad

proposition. QuickCheck printar givetvis test-datan så att man kan flytta det till ett konventionellt unit-test.

Fördelen med att implementera en modell i Haskell är att man kan skriva deklarativa propositioner som modellen skall uppfylla. Man kan på så sätt säkerställa att modellen sannolikt är korrekt. Med fördel går implementation av en funktion och dess propositioner hand i hand.

Propositionell testning är väldigt intressant för att det är en testmetodologi men den har ett helt annat syfte än konventionell testning, och den ifrågasätter det senare. Konventionell unittestning testar mjukvara med syfte att visa att den fungerar som man förväntar sig ; och i ett regressions-test att mjukvaran fungerar fortfarande som man förväntar sig. Propositionell testning utforskar i testrymden för att visa att mjukvaran inte fungerar. Man kan inte förvänta sig att ett propositionellt test inte fungerar idempotent i ett regressions-test.

Den begränsningen är inte ett riktigt problem. Korrekt utformade propositioner för korrekt mjukvara kommer med låg sannolikhet att fallera. Och även om det skulle fallera så kan man skatta felfrekvensen mot konsekvensen. Sen kan även idempotenta tester också fallera av externa faktorer (fallerande skrivningar, nätverks-problem etc).

Propositionell testning gör att man onekligen tänker mer i termer av egenskaper som ett stycke mjukvara under test har - och i förlängningen leder det till färre propositioner och bättre test. Generella egenskaper är i många fall enklare att formulera än rand-villkor.

Sammanfattning

Det är oerhört svårt att skriva unit-tester som är uttömmande för till exempel en data-struktur. Paradoxalt är det oftast enklare att beskriva propositioner. Steget från konventionell unittestning till propositionell testning är inte så långt som man tror. Sannolikt kommer merparten av använda UT-bibliotek innehålla funktionalitet som QuickCheck har om några år.

Om man väljer att skriva sin modell i en funktionellt språk (som Haskell) får man dels de garantier som kompilatorn till den miljön erbjuder, men kan även utnyttja QuickCheck för att skriva propositioner för implementationen. Man kan i det fallet erhålla följande artefakter:

1. En deklarativ modell av problem domänen.
2. Logiska propositioner som modellen skall uppfylla.
3. Datakombinatorer som kan generera data som passar modellen.

Den tredje artifakten är intressant för att det kan möjliggöra data som man kan använda i ett systemtest. Det kanske också är så att konstruktionen av testdata som är det kritiska problemet, och att bygga en säker deklarativ modell är den enda möjligheten för att kunna generera rätt data.

Att testa innebär att man jämför med något. Antigen så jämför man med känd data som man vet är korrekt ; men i fallen då man inte har sådan data måste man bygga något som kan göra jämförelsen åt en. Man kanske tvingas att bygga ett test-orakel.

Man kan då välja att bygga test-oraklet (eller modellen) i ett högre ordningens språk och välja verktyg för den miljön, till exempel propositionell testning.

Men testning är mer än att bara jämföra data. Testutveckling innebär att man måste skapa verktyg för att genomföra tolka test, ta fram testdata, skriva differentiella kontrakt och till och med sätta upp CI/CD-pipelines. Förr eller senare kommer man att behöva en modell som är solid.

Sammanfattning

En svårighet med inbyggda system är att att extrahera information ur ett system under test. Därav tvingas man att göra mycket av testningen i host miljön på ett portabelt sätt.

Den begränsningen är inte alltid negativ; den gör snabb prototyping möjlig och man kan dessutom använda allt mer sofistikerade statisk-analys verktyg.

När man pratar om testutveckling måste man skilja mellan olika scenarion:

1. Prototyping under initial utveckling
2. När en produkt underhålls
3. När en produkt valideras

Metoderna som beskrivs i detta dokument rör scenarios 1 och 2. EDT är synnerligen lämplig när man vill utforska vilka egenskaper ett stycke mjukvara har (test-space exploration). Modell validering kan användas för produkt validering men allt som oftast ställs krav på extern validering och då duger inte en egen utvecklad referens-implementation.

EDT och liknande tekniker visar, i mitt tycke, att vi har allt vi behöver för att skriva bra och tillförlitlig mjukvara. Ny utvecklade programmeringsspråk kommer främst att användas för att validera implementationen i inbyggda system[5].

Framtiden kommer att stavas kompilator instrumentering; att på ett flexibelt sätt skapa profiler för RT-information för vidare analys. Det som sker är att delar av testsystemet kommer att modelleras ovanpå byggsystemet.

Källkods instrumentering är också nödvändig för att göra testning mer tidseffektiv; merparten av all testning som utförs går igenom vilket betyder att den inte hade behövt exekveras. Genom att knyta revisionskontrollen till testsystemet (med kompilatorns stöd) kan man precisera testningen och hitta defekter tidigare.

Här kommer tekniker som propositionell testning bli betydelsefulla - istället för att slösa energi på tester som alltid går igenom kan man låta kombinator bibliotek precisera testningen för att söka invarianter.

[5] Konventionella programmeringsspråk utvecklas också för att möta förändringar.