AFSTUDEERVERSLAG

Het testen van Shell Excel applicaties met Model-Based testing

Corné de Ruijt
Zoetermeer
29-05-2013
In opdracht van: ORTEC

VOORWOORD

Voor de Studie bedrijfswiskunde aan TIS Delft, onderdeel van de Haagse Hogeschool, heb ik van 4 februari tot en met 31 mei 2013 een afstudeeropdracht uitgevoerd. De afstudeeropdracht is uitgevoerd in opdracht van ORTEC binnen de InfoSystems groep, een ontwikkelgroep die zich bezig houdt met Excel VBA ontwikkelingen voor Shell.

Dit verslag is bedoeld voor iedereen die zich bezighoud met het ontwikkelen van software in verschillende ontwikkelomgevingen, in het bijzonder voor VBA ontwikkelaars. Model-Based testing is een breed concept dat in vele settings kan worden toegepast. Een van die settings is Excel VBA maar er zijn nog veel meer ontwikkelgebieden waar Model-Based testing mogelijk en effectief zou kunnen zijn.

Ik wil graag het InfoSystems team bedanken voor alle hulp bij deze afstudeeropdracht. En natuurlijk iedereen van ORTEC die mij heeft geholpen op wat voor manier dan ook om mij te ondersteunen tijdens deze opdracht. In het bijzonder wil ik Pascha Iljin bedanken die tijdens de afstudeerperiode mijn bedrijfsmentor was binnen ORTEC en Edwin van Noort voor de ondersteuning vanuit de opleiding Bedrijfswiskunde.

INHOUDSOPGAVE

SAMENVATTING

In dit afstudeerverslag staat een onderzoek beschreven naar de beste methode om Excel applicaties te testen. Onder testen wordt hierbij verstaan het controleren of een Excel applicatie correct reageert op gegeven input. De Excel applicaties zijn gemaakt door de InfoSystems groep, een ontwikkelgroep binnen de Shell afdeling van de ORTEC Consultancy Group. De Excel applicaties zijn gemaakt voor het bedrijf Shell. Deze Excel applicaties zijn ervoor bedoeld om verschillende processen binnen Shell te analyseren.

De InfoSystems groep wil weten of het huidige testproces efficiënter kan worden gemaakt zonder dat dit consequenties heeft voor de kwaliteit van de Shell Excel applicatie. In theorie is Model-Based testing de beste methode om het testproces te optimaliseren omdat:

- Het automatisch uitvoeren van de testscripts minder tijd kost dan het handmatig testen.
- Het maken van een model minder tijd kost dan het handmatig schrijven van testscripts.

In dit onderzoek is geanalyseerd of dit ook in de praktijk geldt bij het testen van Shell Excel applicaties. De onderzoeksvraag is daarom: ***"Kan het testproces van Shell Excel applicaties worden versneld als er gebruik gemaakt wordt van Model-Based testing onder de voorwaarde dat de kwaliteit van Shell Excel applicaties minstens gelijk blijft? "***

De huidige testmethodieken die worden gebruikt om Excel applicaties te testen zijn handmatig testen en capture and replay testing. Het nadeel van deze methodieken is dat ze veel tijd kosten om te worden uitgevoerd, er naderhand niet gezegd kan worden welke onderdelen van de Excel applicatie hoe goed getest zijn en de oorzaak van fouten lastig te achterhalen is.

Met behulp van het Model-Based testing framework NModel is een testexperiment uitgevoerd. Hierbij is zowel de efficiëntie als effectiviteit van de huidige testmethode en van Model-Based testing gemeten. Uit dit experiment kwam naar voren dat Model-Based testing op lange termijn efficiënter is dan de huidige testmethodiek. De effectiviteit is onder bepaalde voorwaarden ook beter dan de huidige.

Aan de hand van dit onderzoek kan op de volgende manier de onderzoeksvraag beantwoord worden. Het testproces van Shell Excel applicaties kan worden versneld als er gebruik gemaakt wordt van Model-Based testing onder de voorwaarde dat de kwaliteit van Shell Excel applicaties minstens gelijk blijft. Dit antwoord geldt echter alleen als aan de volgende eisen wordt voldaan:

- Er moet sprake zijn van een grote Excel applicatie die over een langere periode ontwikkeld wordt. Bij kleine korte ontwikkelingen is de huidige ontwikkelmethode beter.
- Het model moet de werking van de tool voldoende concreet beschrijven. Als een model te abstract de correcte werking van een programma beschrijft zijn de tests niet in staat om specifieke fouten in de Excel applicatie te vinden. Een model is voldoende concreet beschreven als de met het model gegenereerde testscripts de testdoelstelling bereikt kan worden.

VERKLARENDE WOORDENLIJST

Naam/woord/afkorting:	Betekenis:
ORTEC	Bedrijf dat geavanceerde oplossingen voor planning levert
Shell	Een internationale Oliemaatschappij
InfoSystems	Een afdeling binnen ORTEC die zich o.a. bezighoudt met het ontwikkelen van Shell Excel applicaties
SEA	Afkorting voor Shell Excel Applicaties
SUT	Afkorting voor System Under Test. De SUT is de applicatie die getest wordt
Model-Based testing (MBT)	Een testtechniek om software automatisch mee te testen
Testengine	Een computerprogramma dat vanuit een model testscripts kan genereren
Testscripts	Instructies in een programeer of natuurlijke taal die beschrijft wat de input van een test is en wat de correcte output moet zijn.
Regressietesten	Test die controleert of onaangepaste onderdelen van de applicatie nog steeds werken
SUT Coverage	Een indicator die aangeeft hoeveel functionaliteiten van het totaal zijn getest
Finite State Machine (FSM)	Een gerichte graaf die de correcte werking van een programma aangeeft
Keyword-driven testing	Een testtechniek die gebruik maakt van testscripts die meerdere tests automatisch kan uitvoeren
Script-Based testing	Een testtechniek die gebruik maakt van testscripts die automatisch één test kan uitvoeren
Capture/replay testen	Een testtechniek die handmatige tests kan opnemen en weer kan afspelen op de SUT
Handmatig testen	Een testtechniek waarbij de tester handmatig test uitvoert
C#	Een 3^{e} generatie programmeertaal
.Net framework	Een grote hoeveelheid voorgeprogrammeerde functionaliteiten die gebruikt kunnen worden voor ontwikkelingen in Microsoft producten.
Action	Een beschrijving van de werking van een functionaliteit in NModel
Finite State Machine (FSM)	Een graaf die de werking van een programma weergeeft

1 INLEIDING

In deze scriptie staat een onderzoek beschreven naar de effectiviteit van Model-Based testing bij het testen van bepaalde software applicaties. Deze scriptie richt zich op het praktisch gebruik van Model-Based testing waarbij het gebruik en de effectiviteit van Model-Based testing centraal staat. In deze inleiding zal de achtergrond van dit onderzoek worden toegelicht. Verder zal de opbouw van deze scriptie worden doorgenomen.

Het onderzoek naar de effectiviteit van Model-Based testing werd uitgevoerd bij het bedrijf ORTEC. ORTEC is een van de grootste aanbieders van geavanceerde optimalisatie-oplossingen voor planning. De ORTEC producten en diensten leiden tot een optimale rit- en routeplanning, belading van voertuigen en pallets, personeelsinzet, vraagvoorspelling, logistieke netwerkplanning en magazijn beheer. ORTEC biedt zowel stand-alone, alsook maatwerk en SAP®-gecertificeerde oplossingen, ondersteund door strategische partners. ORTEC heeft meer dan 1750 klanten wereldwijd, 700 werknemers en verschillende kantoren in Europa, Noord-Amerika en Zuid-Amerika.[1]

Een van de klanten van ORTEC is Shell. Shell is een van oorsprong Britse multinational behorend tot de Supermajors de zes grootste staats-onafhankelijke oliemaatschappijen. Het bedrijf telde in 2011 ongeveer 90.000 werknemers. Shell is tevens de meest winstgevende onderneming van Nederland. Met een omzet van ruim $484 miljard (2012) is Shell de grootste private onderneming van de wereld, waarvan de omzet verdeeld is over meer dan 140 landen.[2]

De InfoSystems groep is een groep binnen ORTEC die zich bezighoudt met het ontwikkelen van Excel applicaties voor Shell. Deze Shell Excel applicaties zullen in het vervolg van deze scriptie worden afgekort tot SEA's.

ORTEC maakt talloze software applicaties voor Shell zodat Shell zijn processen en systemen kan analyseren, automatiseren en optimaliseren. ORTEC wil ook in de toekomst veel voor Shell betekenen als het gaat om nieuwe software oplossingen. Omdat meerdere oplossingen in Microsoft Excel worden ontwikkeld, is het van belang van ORTEC om deze applicaties robuuster en van nog betere kwaliteit te maken.

Het doel van deze scriptie is: ***"De lezer informeren over de mogelijkheden van Model-Based testing bij het testen van Shell Excel applicaties".*** Dit doel volgt op de onderzoeksvraag die in deze scriptie wordt beantwoord***: "Kan het testproces van Shell Excel applicaties worden versneld als er gebruik gemaakt wordt van Model-Based testing onder de voorwaarde dat de kwaliteit van Shell Excel applicaties minstens gelijk blijft? "***

Het is niet mogelijk om deze hoofdvraag direct te beantwoorden. Om deze rede zijn een aantal deelvragen opgesteld. Deze deelvragen zullen in de verschillende hoofdstukken van dit afstudeerverslag worden beantwoord.

- Wat is het huidige ontwikkel- en testproces van Shell Excel applicaties?
- Wat is de gewenste situatie?
- Waarom is Model-Based testing in theorie de beste methode om Shell Excel applicaties te testen?
- Hoe kan met Model-Based testing Shell Excel applicaties getest worden?
- Is Model-Based testing in de praktijk efficiënter dan de huidige testmethodiek zonder effectiviteit te verliezen?

[1] Bron: http://www.ortec.nl/about/company_profile.aspx
[2] Bron: http://nl.wikipedia.org/wiki/Royal_Dutch_Shell

Deze scriptie heeft de volgende opbouw.

- In hoofdstuk 2 zal de huidige situatie worden geschetst.
- In hoofdstuk 3 zal de gewenste situatie worden geschetst.
- In hoofdstuk 4 worden een aantal testmethodieken geïntroduceerd en zal beargumenteerd worden waarom Model-Based testing daarvan de beste is om Shell Excel applicaties te testen.
- In hoofdstuk 5 wordt toegelicht hoe Model-Based testing kan worden geïmplementeerd om SEA's te testen.
- Hoofdstuk 6 geeft de resultaten weer van een klein onderzoek naar de effectiviteit en efficiëntie van Model-Based testing ten opzichte van de huidige testmethode.
- In hoofdstuk 7 zal aan de hand van de antwoorden op de deelvragen de hoofdvraag van dit onderzoek worden beantwoord.

2 HET HUIDIGE ONTWIKKEL- EN TESTPROCES VOOR SHELL EXCEL APPLICATIES

2.1 INLEIDING

Om te kunnen bepalen hoe het testproces van Shell Excel applicaties versneld kan worden moet er eerst gekeken worden naar de huidige situatie. Als eerste zal in dit hoofdstuk een toelichting worden gegeven over wat een Shell Excel applicatie eigenlijk inhoud. Vervolgens zal het huidige ontwikkelproces van Shell Excel applicaties worden doorlopen, gevolgd door het huidige testproces.

2.2 SHELL EXCEL APPLICATIES

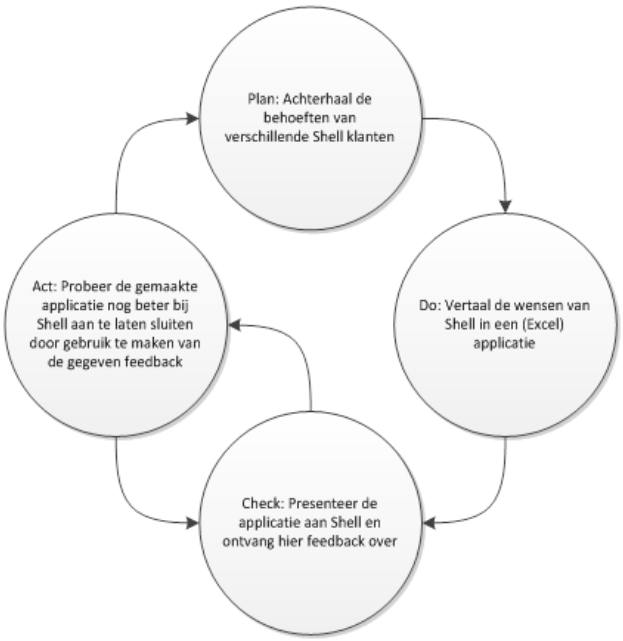
InfoSystems maakt veel cost-estimating applicaties in Excel voor Shell. Shell doet tal van projecten, van het bouwen van boorplatformen tot het vervoeren van olie via pijpleidingen. Shell wil graag een schatting kunnen maken hoeveel dit soort projecten gaan kosten. InfoSystems levert Excel applicaties waarbij een gebruiker de karakteristieken voor een project kan invullen in invoersheets. Vervolgens kan de gebruiker een berekening starten. In verschillende outputsheets wordt het resultaat van de berekening weergegeven. Dit resultaat geeft vaak een schatting van kosten en/of de tijd die een project in beslag gaat nemen.

De implementatie van Shell Excel applicaties kan grofweg worden opgedeeld in twee delen:

- De berekeningsmodulen: De berekeningsmodule zorgt ervoor dat er een accurate schatting berekend wordt over de kosten/duur van een project. De berekeningsmodulen zijn onzichtbaar voor de gebruiker.
- Input en output functionaliteiten: Deze functionaliteiten ondersteunen de gebruiker bij het geven van input voor de berekeningsmodulen en bij het analyseren van de output. Hierbij moet gedacht worden aan foutmeldingen bij foute invoer, de mogelijkheid om bepaalde inputvelden zichtbaar of onzichtbaar te maken of de mogelijkheid input te kopiëren van een andere tool.

2.3 HET ONTWIKKELPROCES VAN SHELL EXCEL APPLICATIES

De InfoSystems groep gebruikt de agile ontwikkelmethode om de SEA's te ontwikkelen. Bij agile ontwikkeling wordt een software applicatie iteratief opgebouwd volgens de Deming cyclus. Bij het ontwikkelen van SEA's ziet dit eruit als figuur 1. Meestal wordt er tijdens één ontwikkelcyclus meerdere keren met de klant overlegd of het product goed genoeg aansluit bij zijn/haar behoeften (check) waarop weer gereageerd wordt door het verbeteren van de applicatie (act). Met deze agile ontwikkelmethode probeert de InfoSystems groep producten af te leveren met een zo hoog mogelijke kwaliteit.



FIGUUR 1, DEMIN CIRKEL ONTWIKKELING SEA'S

Het voordeel van deze agile ontwikkelmethode is dat het eindproduct goed aansluit bij de behoeften van de klant. Een klant heeft vaak slechts een vaag idee wat hij/zij eigenlijk wil. Door al na een korte tijd een ontwerp of 1e versie van een applicatie te presenteren kan de klant in een vroeg stadium zich concreet oriënteren op de eindoplossing. Een klant kan aangeven bij welke behoeften deze applicatie aansluit en op welke behoeften nog niet. De eerste versie van de applicatie bevat nog niet de complexiteit van een volledige applicatie en is daarom nog gemakkelijk aan te passen. De agile ontwikkelmethode geeft de InfoSystems groep dus de mogelijkheid de klant te oriënteren op wat zijn/haar behoefte eigenlijk is, en hier vervolgens op in te spelen.

Een nadeel van de agile ontwikkelmethode is dat een applicatie sneller (te) complex wordt. Bij agile ontwikkeling wil de ontwikkelaar zo snel mogelijk feedback van de klant. Als een klant namelijk aangeeft dat hij/zij een functionaliteit waar 3 maanden aan is gewerkt toch niet interessant vindt, dan is dit een flinke tijdsverspilling. Het maken van documentatie en/of het maken van een gedetailleerd ontwerp kost op de korte termijn veel tijd. Veel ontwikkelaars kiezen er daarom voor om minder tijd te besteden aan documentatie en meer aan het ontwikkelen van functionaliteiten die de klant interessant kan vinden. Op de lange termijn heeft dit als gevolg dat een applicatie een mengelmoes wordt van functionaliteiten waarvan soms niet volledig bekend is wat ze doen en hoe ze dat doen. Het verder ontwikkelen of testen van deze applicaties is door de grote complexiteit en gebrek aan documentatie een ingewikkeld proces wat veel tijd in beslag neemt.

2.4 HOE ZIET HET TESTPROCES VAN SHELL EXCEL APPLICATIES ERUIT?

De agile ontwikkelmethode heeft gevolgen voor het testproces van Shell Excel applicaties. De requirements van een klant, die aangeven waar de applicatie aan moet voldoen, kunnen elke keer als het “check” proces wordt doorlopen wijzigen. InfoSystems wil hier snel op reageren door binnen een korte tijd een versie te ontwikkelen waar de nieuwe functionaliteiten als deels of volledig in zijn geïntegreerd. Het schrijven van een testplan of het ontwikkelen van automatische tests voor deze functionaliteiten kost veel tijd. Daarnaast zijn deze tests statisch: als de klant toch van gedachte verandert moet dit testplan en de automatische tests weer worden gewijzigd wat

opnieuw tijd kost. De agile ontwikkelmethode zorgt er dus voor dat InfoSystems de SEA's met dynamische testmethoden wilt testen.

Tijdens het testen wordt een programma dat getest wordt (in dit geval een Shell Excel applicatie) System Under Test (SUT) genoemd. De InfoSystems groep test SUT vaak op de volgende manier:

- Handmatig testen: Een ontwikkelaar van een SUT weet meestal wel hoe een SUT zou moeten werken en waar de zwakke plekken van een SUT zitten. De ontwikkelaar test een SUT door handmatig zoveel mogelijk functionaliteiten aan te roepen en te verifiëren of de functionaliteiten correct werken. Handmatig testen wordt vooral veel gedaan bij het testen van de input en output functionaliteiten.
- Capture and replay testen: Capture and replay tests wordt een enkele keer gedaan bij het testen van de berekeningsmodulen. Als men er nagenoeg zeker van is dat een berekeningsmodule correct werkt dan worden hier een aantal berekeningen op gedaan. De invoer en uitvoer van deze berekeningen worden opgeslagen en beschouwd als het correcte gedrag van de berekeningsmodule. De correctheid van de berekeningsmodule kan vervolgens worden getest door de opgenomen invoer te geven en te verifiëren of de output en de opgenomen output hetzelfde zijn.

Het voordeel van handmatig testen is dat het gemakkelijk uit te voeren is. Een ontwikkelaar weet hoe de applicatie werkt (of zou moeten werken) en kan door verschillende functionaliteiten aan te roepen al snel zeggen waar er fouten zitten in de SUT. Een nadeel van handmatig testen is dat bij programma's met veel functionaliteiten het ook veel tijd kost om alle functionaliteiten handmatig te testen.

Een voordeel van capture and replay testen is dat de test automatisch kan worden uitgevoerd. Hierdoor neemt replay testen niet veel tijd in beslag. Het nadeel van replay testen is dat het een erg statische methode is om mee te testen. Als een berekeningsmodule verandert dan zal deze eerst een hoop handmatige tests moeten doorlopen voordat gezegd kan worden dat de berekeningsmodule klopt en opgenomen kan worden. Voor SEA's waarbij de berekeningsmodule dus vaak verandert moet deze berekeningsmodule alsnog veel handmatig worden getest.

2.5 Welke problemen levert de huidige situatie?

Zowel handmatig als replay testen kost relatief veel tijd. InfoSystems schat dat ongeveer 10 tot 20% van de totale ontwikkeltijd zit in het testen van SEA's. Deze tijd kunnen ontwikkelaars goed gebruiken om de gevonden fouten tijdens het testen te repareren of om andere functionaliteiten door te ontwikkelen. De totale test tijd verminderen zonder kwaliteitsverlies van het testen kan dus tot een verbetering van de kwaliteit van de SEA's leiden.

Een ander probleem is dat bij het handmatig testen de zogenaamde SUT coverage niet wordt bijgehouden. De SUT coverage geeft aan welke functionaliteiten van de SUT zijn getest en hoe deze zijn getest. De SUT coverage bijhouden heeft voordelen. Er zal minder snel functionaliteiten dubbel worden getest en na het testen is precies bekend wat er is getest en hoe grondig dit is gedaan. Als de SUT coverage wel zou worden bijgehouden zou dit dus zowel de totale test tijd als de kwaliteit verbeteren.

Een tester houdt meestal niet exact bij welke handelingen hij/zij op de SUT heeft uitgevoerd. Soms kunnen dit een grote reeks handelingen zijn die een ontwikkelaar niet precies heeft onthouden. Als een tester dan een fout tegenkomt is het bijna onmogelijk om deze hele reeks handelingen te reproduceren. Als de oorzaak van de fout ook nog eens al een tijd geleden heeft voorgedaan en nu pas tot uiting komt is het moeilijk de oorzaak te achterhalen. Het bijhouden van de handelingen zal dus het opsporen van fouten in de SUT een stuk makkelijker maken.

2.6 CONCLUSIE

In dit hoofdstuk is duidelijk geworden dat Shell Excel applicaties worden ontwikkeld met behulp van de agile ontwikkelmethode. De agile ontwikkelmethode lijkt op de Deming cyclus met als verschil dat de processen “check” en “act” meestal meerdere keren achter elkaar worden uitgevoerd. Met de agile methode wordt geprobeerd zo goed mogelijk aan de wensen van de klant te voldoen.

Het testen van Shell Excel applicaties gebeurt door handmatig testen en soms met capture and replay testing. Met handmatig testen worden de input en output functionaliteiten getest. Capture and replay tests worden gebruikt bij het testen van de correctheid van berekeningen.

De huidige testsituatie levert ook een aantal problemen op. Ten eerste kost handmatig testen veel tijd, ten tweede wordt de SUT coverage niet bijgehouden. Als laatste kunnen fouten in de SUT met de huidige testmethoden moeilijk worden opgespoord.

3 DE GEWENSTE SITUATIE

3.1 INLEIDING

InfoSystems wil het testproces van SEA's verbeteren. In het hoofdstuk over de huidige situatie hebben we gezien dat deze situatie problemen veroorzaakt. Het oplossen van deze problemen kan voor InfoSystems een beter testproces en betere kwaliteit van de SEA's opleveren. In dit hoofdstuk zal eerst worden toegelicht waarom InfoSystems graag het testproces van SEA's wil versnellen. Daarna zal worden toegelicht hoe InfoSystems het testproces van SEA's wil versnellen.

3.2 HET VERSNELLEN VAN HET TESTPROCES

InfoSystems wil graag het testproces van SEA's versnellen zonder dat dit ten koste gaat van de kwaliteit van SEA's. Minder tijd verspelen aan testen betekent dat ontwikkelaars meer tijd kunnen besteden aan het oplossen van fouten in de code en het door ontwikkelen van functionaliteiten. Een SEA kan dan ook sneller gepresenteerd worden aan de klant. Een versneld testproces kan er dus voor zorgen dat InfoSystems beter aan de behoeften van Shell klanten kan voldoen.

Om de totale test tijd te verminderen wil InfoSystems meer gaan doen aan automatisch testen met behulp van testing profiles. Deze testing profiles beschrijven over welke functionaliteiten een SEA moet beschikken en hoe deze functionaliteiten moeten werken. Hoe kan automatisch gecontroleerd worden of een SEA werkt volgens een testingprofile? Dit kan door met een test softwareapplicatie de functionaliteiten die beschreven staan in een testingprofile aan te roepen in de SEA. Daarna kan gecontroleerd worden of de SEA correct heeft gereageerd op het aanroepen van de functionaliteit. Het liefst zijn deze testing profiles makkelijk op te stellen en te wijzigen zodat ze passen in de agile ontwikkelmethode. De testingprofile beschrijft dus de correcte werking van het programma, en een test softwareapplicatie controleert of de SEA reageert volgens het testingprofile. Als deze testing profiles makkelijk kunnen worden opgesteld/gewijzigd en volledig automatisch een SEA kunnen testen dan zal dit veel tijdwinst opleveren.

Testprofiles zijn handig om veel functionaliteiten te testen, maar er zijn ook beperkingen. De testprofiles moeten aan de ene kant gemakkelijk op te stellen/wijzigen zijn, maar aan de andere kant wel genoeg kracht hebben om een SEA te testen. Bij simpele functionaliteiten is dit goed mogelijk, bij complexere functionaliteiten neemt ook de complexiteit van de testprofiles flink toe. Dit maakt de testprofiles minder onderhoudbaar. Bij SEA zijn de berekeningsmodulen van die complexe functionaliteiten. Er zal dus onderzocht moeten worden of het mogelijk is complexe functionaliteiten met testingprofiles te testen.

Om de risico's van niet onderhoudbare testprofiles te vermijden moet een testprofile zo gemaakt kunnen worden dat deze gemakkelijk kan worden opgezet en kan worden aangepast. Testprofiles moeten hiervoor zowel overzichtelijk als volledig zijn. Overzichtelijk zodat de testprofile makkelijk kan worden opgezet, aangepast en dat eraan te zien is welke functionaliteiten hoe worden getest. Volledig zodat de testprofile wel de functionaliteiten voldoende kan beschrijven. Er zijn verschillende testmethodieken beschikbaar die testprofiles kunnen implementeren. In deze scriptie zal duidelijk worden welke methodiek dit het beste kan.

3.3 CONCLUSIE

InfoSystems wil het testproces van SEA's versnellen zonder dat dit ten koste gaat van de kwaliteit van SEA's. Om deze doelstelling te bereiken wil InfoSystems gebruik gaan maken van testprofiles. Aan de hand van deze

testprofiles kan een SEA automatisch worden getest. De uitdaging bij het maken van testprofiles is dat deze zowel overzichtelijk als volledig moeten zijn om tijdwinst op te leveren.

4 SOFTWARE TESTEN

4.1 INLEIDING

Er zijn een aantal testmethoden waarmee de gewenste situatie bereikt kan worden. In dit hoofdstuk zal eerst een afbakening worden gemaakt die aangeeft in welke ruimte en vanaf welk standpunt er naar softwaretesten is gekeken. Vervolgens zal er aandacht worden besteed aan met welke karakteristieken testmethoden vergeleken kunnen worden. Als laatste zullen de testmethodieken toegelicht en vergeleken worden, waarbij ook zal worden toegelicht waarom er voor Model-Based testing is gekozen.

4.2 WAT WORDT ER VERSTAAN ONDER SOFTWARETESTEN?

Software testen is erg breed begrip dat verschillende betekenissen kan hebben. Voordat überhaupt testmethodieken bekeken kunnen worden zal eerst het software testgebied moeten worden afgebakend. In welk gebied van softwaretesten moeten we eigenlijk kijken als we zo effectief mogelijk SEA's willen testen? Tijdens dit onderzoek is de volgende definitie van softwaretesten gebruikt: *"Software testing bestaat uit de dynamische verificatie van het gedrag van een applicatie door een eindige verzameling van testcases, handig geselecteerd uit een soms bijna oneindige groep mogelijke input, tegen verwacht gedrag."*[3] Binnen dit onderzoek is dus alleen naar software testmethodieken gekeken die binnen deze definitie vallen. In deze paragraaf zullen een aantal belangrijke afbakeningen in de zoekruimte worden besproken.

Ten eerste wordt er gesproken over ***dynamische verificatie.*** Dit houdt in dat de software wordt getest terwijl deze wordt uitgevoerd. In theorie zou software ook kunnen worden getest zonder deze uit te voeren (statisch testen). In elke applicatie wordt veel gebruik gemaakt van externe code of applicaties. De koppeling hiermee kan niet worden getest zonder daadwerkelijk de code uit te voeren. Statische tests kunnen daarom alleen in de theorie testen of een applicatie werkt. Dynamische tests kunnen in de praktijk testen of een applicatie werkt.

Softwareapplicaties hebben al snel een grote hoeveelheid mogelijke gedragingen. Bewijzen dat software altijd het goede gedrag uitvoert wordt hierdoor bemoeilijkt. Er bestaan methodieken om te bewijzen dat software altijd correct werkt. Alleen duurt het uitvoeren van deze methodieken soms langer dan het implementeren van de software zelf. We beperken ons daarom tot methodieken die succesvol in een korte tijd veel fouten in de implementatie kunnen ontdekken. Het hoofddoel is dus in een korte tijdsperiode zoveel mogelijk fouten opsporen in de implementatie en niet bewijzen dat een applicatie 100% correct werkt.

Bij het uitvoeren van de tests wordt er enkel ***blackbox*** getest. Blackbox testen houdt in dat de code achter een applicatie tijdens het testen niet wordt gebruikt. We beschouwen de software applicatie als het ware als een zwarte doos waar input in gaat en output uit komt. Pas nadat het testen is uitgevoerd wordt er in de code gekeken wat er mis ging. Er wordt dus tijdens het testen geen informatie gebruik over de implementatie.

Er wordt bij het testen alleen getest op ***functionaliteit***, ***performance*** en ***robuustheid***. Hierbij houdt functioneel testen in dat er wordt gecontroleerd of een functionaliteit van het programma het correcte gedrag vertoond. Bij performance testen wordt een applicatie getest onder zware omstandigheden, zoals een grote hoeveelheid inputdata. De robuustheid wordt getest door de applicatie niet-toegestane input te geven en te controleren of het programma hier correct op reageert. Het testen van bijvoorbeeld gebruikersvriendelijkheid wordt achterwegen gelaten.

[3] Vrij vertaald uit Morgan Kaufmann Practical Model-Based Testing A Tools Approach (2007)

Dit onderzoek focust zich vooral op ***integratie testen***. Dit houdt in dat de volledige applicatie of grote componenten van de applicatie worden getest. Tegenover integratie tests staan unit-testing waarbij slechts een kleine component van een software applicatie wordt getest. Er zal dus minder aandacht worden besteed aan het testen van kleine componenten.

Er is in dit onderzoek alleen gekeken naar testmethodieken die binnen een paar maanden geïntegreerd kunnen worden om SEA's te testen. Sommige testmethodieken zijn simpelweg te complex of kosten teveel tijd om geïmplementeerd te worden. Andere methoden zijn enkel geschikt low-level software (software die direct hardware aanstuurt) te testen. Deze testmethodieken zijn achterwegen gelaten.

Bij het testen ligt de focus alleen op de correctheid van de eigen geïmplementeerde functionaliteiten. SEA's maken zeer veel gebruik van standaard Excel functionaliteiten. Aangezien deze functionaliteiten niet kunnen worden aangepast heeft het geen zin om deze te testen. Functionaliteiten die standaard in Excel zitten worden dus niet getest.

4.3 HOE KAN DE EFFECTIVITEIT EN EFFICIËNTIE VAN EEN TESTMETHODIEK GEMETEN WORDEN?

Bij het analyseren van een testmethodiek kan zowel de effectiviteit als de efficiëntie van zo'n testmethodiek worden geanalyseerd. In deze paragraaf zullen de test karakteristieken worden doorgenomen waarop de effectiviteit en de efficiëntie van een methode kunnen worden gemeten. [4]

- ***Effectiviteit:***
    - SUT Coverage: De SUT Coverage, model-coverage of gewoonweg coverage is een maatstaf dat aangeeft wat er in een bepaalde test precies is getest. Er zijn verschillende manieren om dit te meten. Voorbeelden hiervan zijn het percentage uitgevoerde functionaliteiten van de totale functionaliteiten, of het percentage van het aantal verschillende geteste input van de totaal mogelijke inputs.
    - Regressietesten, een regressietest test of niet veranderde functionaliteiten nog steeds het correcte gedrag vertonen. Om de kwaliteit van een regressietest te meten kunnen de verschillende maatstaven voor effectiviteit en efficiëntie worden gebruikt.
    - Aantal gevonden fouten per tijdseenheid: Deze maatstaf spreekt voor zich, voor verdere analyse kunnen deze fouten ook worden opgedeeld in functioneel, robuustheid of performance fouten.
- ***Efficiëntie***
    - Test voorbereidingstijd: de meeste testmethodieken vergen voorbereidingstijd. Bij handmatig testen wordt er soms een testplan opgesteld. Hierin staat beschreven waar de tester zich op moet richten tijdens het testen. Bij automatisch testen moeten er tests geïmplementeerd worden.
    - Test uitvoeringstijd: met test uitvoeringstijd wordt bedoeld hoeveel tijd het kost om alle testcases uit te voeren en te verifiëren of de software het juiste gedrag heeft vertoond.
    - Aanpasbaarheid van de tests: is het gemakkelijk om een al eerder gemaakte test aan te passen zodat deze veranderingen in de SUT kan testen? Maatstaven hiervoor zijn de tijd die het kost om de tests aan te passen en de effectiviteit/efficiëntie van een nieuwe test

[4] De meetstaven voor testmethodieken zijn afgeleid uit (Utting & Legeard, 2007)

- Complexiteit van de methodiek: Bij automatisch testen heeft een tester al snel kennis nodig over programmeren en modelleren. Hoe complex een methode is wordt subjectief bepaald aan de hand van de mening van de tester.
- Tijd tussen het vinden van een fout en het vinden van de oorzaak daarvan: Als er een fout wordt gevonden wil een tester of programmeur zo snel mogelijk weten wat de oorzaak hiervan was. Een aantal testmethodieken zijn erg sterk in het leveren van informatie over de gevonden fout.
- Terugkoppeling naar de requirements: De requirements van een softwareprogramma zijn als het ware de bouwtekening van de applicatie. De requirements geven dan ook aan welk gedrag correct en niet correct is. In sommige gevallen kunnen requirements dubbelzinnig zijn, niet haalbaar of onjuist zijn. Deze maatstaaf geeft aan in hoeverre een testmethodiek een fout kan terugkoppelen naar een requirement. Testmethodieken die hier goed in zijn kunnen zowel de softwareapplicatie als de kwaliteit van de requirements meten.

4.4 WELKE TESTMETHODIEKEN ZIJN ER?

Binnen de scope die in paragraaf 4.2 is aangegeven zijn er een aantal testmethodieken die kunnen bijdragen aan het bereiken van de gewenste situatie. In deze paragraaf zullen deze test methodieken worden toegelicht. Ook zal er worden aangegeven wat de mogelijkheden en beperkingen van deze testmethoden zijn.

Er zijn drie testmethodieken die het idee van testingprofiles kunnen implementeren. Dit zijn script-Based testing, keyword-driven testing en Model-Based testing. Er zal kort voor elke methode worden uitgelegd wat de methode inhoud.

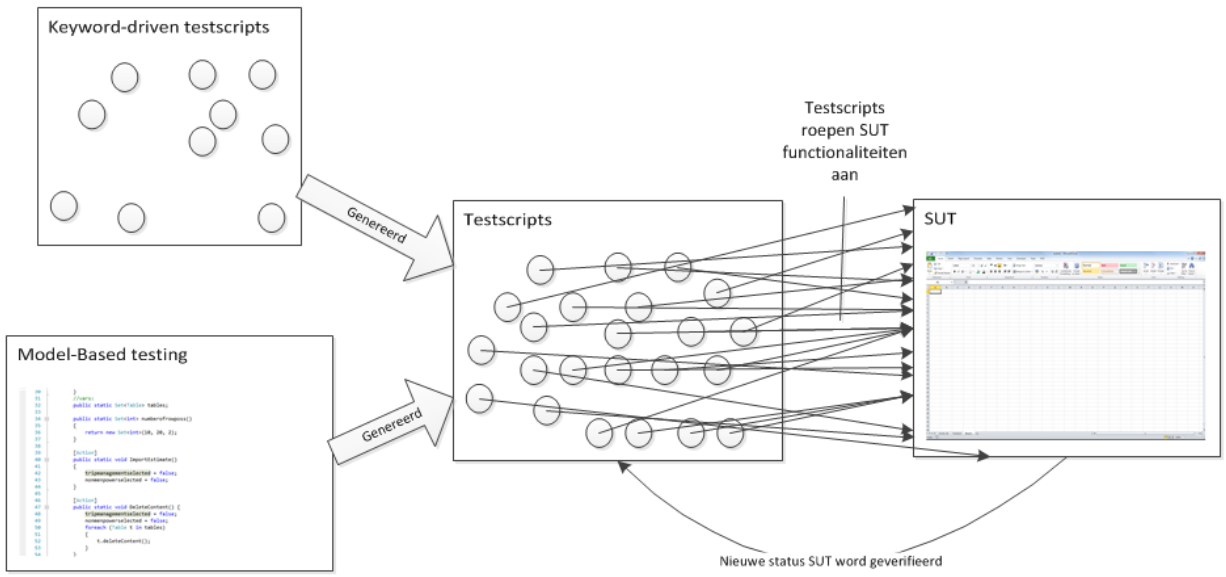
Een software applicatie kan in sommige gevallen goed vergeleken worden met een vijver. Deze vijver bevat verschillende vissen (de functionaliteiten van het programma) die allemaal een bepaalde kant op zwemmen (de werking van de functionaliteiten). De verschillende testmethodieken geven antwoord op de volgende vraag: ”Zitten de correcte vissen in de vijver en zwemmen deze de goede kant op?”. Oftewel bevat een softwareapplicatie de juiste functionaliteiten en werken deze functionaliteiten correct?

Er zijn talloze manieren om antwoord te geven op deze vraag. In de huidige situatie gebeurt dit met handmatig testen en capture and replay testing. Als InfoSystems echter wil overgaan op volledige automatisering van testen met behulp van testingprofiles dan zijn de volgende testmethodieken mogelijk:

- ***Script-Based testing***. Stel we zouden in de vijver sensoren hangen die van een klein stukje vijver constant kunnen verifiëren of de correcte vissen in de vijver zitten en of deze vissen de goede kant op zwemmen. In dat geval zouden we feitelijk doen aan script-Based testing. Bij script-Based testing wordt er voor elke functionaliteit een testscript geschreven. Dit testscript is een stukje code die automatisch kan worden uitgevoerd en kan verifiëren of de functionaliteit werkt.
  Script-Based testing maakt het mogelijk om volledig automatisch de test uit te voeren. Testprofiles zouden met deze methode kunnen worden opgezet door in zo’n testprofiel allemaal testscripts te integreren die allemaal een bepaalde functionaliteit testen.
  Beperkingen van script-Based testing liggen in het feit dat er relatief veel testscripts geschreven moeten worden voordat een applicatie redelijk effectief getest kan worden. Hierdoor kost het veel tijd om testscripts te schrijven en te onderhouden.
- ***Keyword driven testing***. Keyword-driven testing lijkt veel op unit-testing. Het verschil is dat bij Unit-testing één test één functionaliteit test. Bij keyword driven testing kan één test meerdere functionaliteiten testen. Dit komt omdat een test gemaakt met keyword-driven testing op meerdere

manieren interpreteerbaar is. Bepaalde keywords in een testscript kunnen slaan op meerdere functionaliteiten. Om de keyword-driven tests uit te voeren moet zo'n test wel eerst vertaald worden naar een testscript, dit gebeurt automatisch met een adapter.
Keyword driven testing maakt het mogelijk om op een abstractere wijze tests te beschrijven wat de tests makkelijker onderhoudbaar maakt. Beperkingen zijn dat een tester kennis nodig heeft hoe op een effectieve wijze keyword driven tests te maken. Ondanks een hoger abstractieniveau moeten er nog steeds veel tests geschreven worden voordat een applicatie goed getest kan worden.

- ***Model-Based testing***. Het ophangen van de sensoren in de vijver kost veel tijd en er is een kans dat sommige delen van de vijver geen sensor hebben. Om dit probleem te verhelpen kan er een model gemaakt worden van de vijver, met bijvoorbeeld de afmeting van de vijver, welke vissen erin zwemmen en hoe deze vissen zwemmen. Dit model wordt vervolgens in een softwareapplicatie gestopt (in Model-Based testing heet deze softwareapplicatie een testengine) die berekend waar de sensoren moeten worden opgehangen en die ze ook uiteindelijk ophangt. Bij Model-Based testing worden er dus aan de hand van een model testscripts gegenereerd en uitgevoerd.
  Model-Based testing maakt het mogelijk om op een abstract niveau te beschrijven hoe een softwareapplicatie moet werken. Door het model aan te passen en opnieuw de testengine te runnen kunnen in één klap vele tests worden gegenereerd. Een beperking van Model-Based testing is dat er veel kennis nodig is over het effectief opzetten van een model om de testmethodiek effectief te kunnen uitvoeren.



FIGUUR 2, WERKING VERSCHILLENDE TESTMETHODIEKEN

4.5 WAAROM IS MODEL-BASED TESTING IN THEORIE DE BESTE METHODE?

In de vorige paragraaf hebben we gezien dat Model-Based testing de methode is die automatisch softwareapplicaties kan testen aan de hand van een abstract model. Echter neemt de complexiteit bij deze methode flink toe waardoor andere automatische testmethoden beter zouden kunnen werken. In deze paragraaf zal worden toegelicht waarom Model-Based testing toch is gekozen als in theorie de beste methode om SEA's te testen.

In paragraaf 4.3 zijn een aantal criteria genoemd waarmee de effectiviteit en efficiëntie van softwareapplicaties gemeten kan worden. Om de gewenste situatie te bereiken is het vooral van belang om de efficiëntie van het testen te vergroten, zolang de effectiviteit minstens gelijk blijft. Er is dus gekeken naar de testmethode die op efficiëntie beter scoort dan andere testmethodieken zonder op effectiviteit te verliezen.

In Tabel 1 staat een overzicht van de prestaties van de testmethodieken ten opzichte van een aantal criteria. De tabel is gebaseerd op zowel informatie uit bronnen als (Utting & Legeard, 2007) en (Jacky, Veanes, Campbell, & Schulte, 2007)als op aanwezige praktijkervaring binnen InfoSystems.

	Criteria:							
Testmethodiek:	Voorbereidingstijd:	Test uitvoeringstijd	SUT coverage bepalen	In staat fouten te vinden	In staat om complexe systemen te testen	In staat om oorzaak van fouten te achterhalen	Complexiteit van testmethodiek	Mogelijkheid tot regressietesten:
Handmatig testen:	++	--	--	-	+-	--	++	-
Capture/replay testen:	+	+-	--	-	+-	+	+	+-
Script-Based testing	--	++	+-	+-	+	++	+-	++
Keyword-Driven testing	--	+	+-	+-	+-	+	-	++
Model-Based testing	-	+	++	++	+	+	--	++
	scoort positief op criteria: +							
	Scoort negatief op criteria: -							

TABEL 1, THEORETISCHE PRESTATIES VAN VERSCHILLENDE TESTMETHODIEKEN

Aan de hand van deze tabel is er geconcludeerd dat Model-Based testing inderdaad in theorie de beste methode is om SEA's te testen en wel om de volgende redenen:

- Met Model-Based testing kunnen tests automatisch worden uitgevoerd. Dit levert tijdwinst op ten opzichte van handmatige of deels handmatige methoden. Model-Based testing is dus efficiënter in testuitvoering dan deze methoden. Omdat dezelfde test meestal meerdere malen worden uitgevoerd loopt het verschil in test uitvoeringstijd al snel op naar dagen of weken.

Een model is overzichtelijker, abstracter, duidelijker en makkelijker aanpasbaar dan testscripts. Dit maakt dat Model-Based testing efficiënter is in testvoorbereiding en aanpasbaarheid dan andere automatische testmethodieken. Weliswaar verliest Model-Based testing tijd in de voorbereiding ten opzichte van handmatige methoden. De praktijk wijst in veel gevallen uit dat deze tijd wordt ingehaald door de winst in test uitvoeringstijd (Utting & Legeard, 2007).

- Model exploration zorgt ervoor dat de testscripts van hoge kwaliteit zijn. Bij alle andere methodieken zijn de testcases alleen afhankelijk van welke tests een tester definieert. Bij Model-Based testing wordt het gemaakte model verkend met slimme algoritmen wat de kans vergroot dat foutgevoelige stukken code worden getest.

Model-Based testing neemt echter ook risico's met zich mee:

- Het model kan te simplistisch of te complex zijn. Een te simplistisch model zorgt ervoor dat er geen goede tests kunnen worden gegenereerd. Dit komt omdat het model dan te weinig informatie geeft over de correcte werking van een programma. Een te complex model daarin tegen geeft teveel informatie. Het model wordt dan moeilijk onderhoudbaar. Daarnaast zijn sommige testgeneratie algoritmen niet goed in staat om uit een uitgebreid, complex model tests te genereren.
  Om dit risico te vermijden is het noodzakelijk dat er een modelleertaal gekozen wordt die in staat is de soms complexe functionaliteiten van een applicatie helder te beschrijven. Daarnaast moet het model een bepaald abstractie niveau hebben ten opzichte van de applicatie. Dan blijft het model onderhoudbaar en zijn de testgeneratie algoritmen in staat om goede tests te produceren.
- Er kunnen fouten in het model voorkomen. Een tester kan een functionaliteit verkeerd beschrijven in het model. Dit komt pas ten uiting tijdens het uitvoeren van de testcases. Het lijkt dan alsof er een fout zit in de implementatie terwijl deze in het model zit. Fouten in het model kunnen niet voorkomen worden. Wel kan een overzichtelijk model in een heldere modelleertaal helpen om fouten in het model op te sporen. Daarnaast zal de tester er attent op moeten zijn dat een gevonden fout zowel in de applicatie als in het model kan zitten.
- Model-Based testing is in praktijk moeilijker uit te voeren dan andere testmethodieken. Een model moet vertaald worden naar testscripts. Deze moeten op de een of andere manier worden uitgevoerd op in dit geval Excel. En uit Excel moet bepaalde informatie worden gehaald over welk gedrag er is vertoond. Tijdens dit proces moeten verschillende systemen met elkaar communiceren en dat is niet altijd mogelijk. Het is daarom noodzaak om een Model-Based testing tool te zoeken die kan communiceren met de verschillende systemen.

4.6 CONCLUSIE

In deze paragraaf zijn verschillende testmethodieken geïntroduceerd en met elkaar vergeleken. Tijdens het onderzoek naar de werking van software testmethodieken is alleen gekeken naar softwaremethodieken die binnen de scope van dit onderzoek vallen. Deze testmethodieken kunnen gemeten worden naar verschillende maatstaven om de effectiviteit en de efficiëntie ervan aan te geven. De onderzochte testmethodieken zijn: script-Based testing, keyword-driven testing en Model-Based testing. Ondanks dat Model-Based testing risico's met zich meebrengt is Model-Based testing toch in theorie de beste methode om SEA's te testen. Dit omdat Model-Based testing in theorie het beste scoort op het gebied van efficiëntie en effectiviteit.

5 MODEL-BASED TESTING MET NMODEL

5.1 INLEIDING

Om Model-Based testing te kunnen integreren in het testproces van SEA's is NModel gebruikt. In dit hoofdstuk zal worden toegelicht wat NModel is en hoe NModel is gebruikt om SEA's te testen. Dit zal gedaan worden door eerst toe te lichten wat NModel is en waarom NModel wordt gebruikt. Vervolgens zal worden toegelicht hoe NModel werkt en welk proces InfoSystems ontwikkelaars/testers moet volgen om slim gebruik te maken van NModel.

5.2 WAT IS NMODEL?

NModel is een Model-Based testing tool ontwikkeld door de Microsoft research ontwikkelgroep. Microsoft was een aantal jaar terug zelf op zoek naar methodieken om het testproces van zijn producten te versnellen en ontwikkelde daarop een eigen Model-Based testengine. Deze testengine is de afgelopen jaren doorontwikkeld en dit heeft 2 producten opgeleverd: het closed source SpecExplorer Model-Based testing tool en het open source NModel Model-Based testing tool. SpecExporer is uitgebreider dan NModel maar kan helaas niet gebruikt worden voor commerciële doeleinde en het is dus niet mogelijk om Shell Excel applicaties te testen met SpecExplorer. NModel is erop gericht om applicaties te testen die gemaakt zijn voor Microsoft besturingssystemen.

NModel bestaat uit de volgende vier componenten:

- Modelling library: De modelleertaal die NModel gebruikt is C#. C# is eigenlijk een object georiënteerde programmeertaal die veel lijkt op programmeertalen als Java of c++. In NModel kan handig gebruik worden gemaakt van de object georiënteerde opbouw van C#. Object georiënteerde modellen kunnen namelijk op een zeer duidelijke manier een applicatie beschrijven.
  Naast de normale C# syntax zijn er een aantal componenten toegevoegd om een model te beschrijven. Deze zullen worden toegelicht in paragraaf 5.4.1.
- Offline test generator (otg): De offline test generator genereert aan de hand van een gegeven model testscripts.
- Comformance tester (ct): De comformance tester voert de testscripts uit op de SUT.
- plotting tools: NModel bevat een aantal tools om een model te visualiseren. Echter kunnen deze plotting tools niet zomaar voor commerciële doeleinde worden gebruikt. Om deze reden zijn deze tools niet gebruikt bij het testen van SEA's

5.3 WAAROM IS ER VOOR NMODEL GEKOZEN?

Ondanks dat het Model-Based testing concept al langer bestaat is er de afgelopen 10 jaar pas steeds meer gebruik van gemaakt. Grotendeels door een steeds sterkere behoefte aan testmethodieken die zowel efficiënt als effectief zijn. De ontwikkelingen in Model-Based testing hebben een aantal Model-Based testing tools opgeleverd.

Niet elke Model-Based testing tool is geschikt om SEA's te testen. Welke acties er aangeroepen kunnen worden in een SEA is vaak afhankelijk van acties die daarvoor zijn aangeroepen. We kunnen bijvoorbeeld een cel waarde niet wijzigen als we net de sheet hebben verwijderd. MBT tools die deze afhankelijkheid niet kunnen beschrijven vallen daarom af. De tools die overblijven zijn:

- Finite State Machine (fsm) tools: Deze MBT tools gebruiken grafentheorie om tests te genereren.

- Gespecialiseerde MBT tools: Dit zijn uitgebreide Model-Based testing tools waarin de tester soms meerdere modelleertalen en testgeneratie algoritmen kan gebruiken. Voorbeelden hiervan zijn Conformiq Qtronic, Axini en Smart testing.

Om tot een beslissing te komen welke tool te gebruiken om SEA's te testen zijn verschillende criteria gebruikt. Ten eerste is het noodzakelijk dat een Model-Based testing tool überhaupt kan worden gebruikt om SEA's te testen. Als een tool bijvoorbeeld niet aan de volgende eisen voldoet kan deze niet worden gebruikt:

- De MBT tool moet met Excel kunnen communiceren. Anders kunnen tests niet automatisch worden uitgevoerd.
- De MBT tool moet op Shell laptops kunnen draaien. Shell laptops hebben over het algemeen meer restricties dan standaard laptops.
- De tool moet voor commerciële doeleinde gebruikt mogen worden.

De tools die aan deze eisen voldoen zijn vervolgens nog beoordeeld op onder andere de volgende criteria:

- *Wat is de prijs van de tool*? Voornamelijk de gespecialiseerde MBT tools bleken erg prijzig terwijl het niet altijd aantoonbaar is dat deze beter presteren dan FSM tools.
- *In hoeverre zijn de programmeurs/testers bekend met een modelleertaal?* Het opleiden van ontwikkelaars/testers in het maken van een model in een onbekende modelleertaal kost zeker een paar weken. Hoe dichter de modelleertaal bij de huidige kennis van de ontwikkelaars/testers staat des te sneller zullen de testers/ontwikkelaars in staat zijn om een testmodel te bouwen
- *Wat is de kwaliteit van de tool?* Om dit te bepalen moeten vragen beantwoordt worden zoals: hoe goed zijn de test die worden gegenereerd?, hoe sterk is de modeleertaal? en wat is er allemaal mogelijk met de tool? Ondanks dat deze vragen niet direct beoordeeld kunnen worden zonder de MBT tool uit te proberen is er met behulp van al eerder gedane onderzoeken(Eslamimehr, 2008)(Shafique & Labiche, 2010) hier een inzicht in te verkrijgen. Een vergelijking tussen verschillende MBT tools is te vinden in Appendix A.

Uiteindelijk is er door de volgende argumenten voor NModel gekozen:

- NModel voldoet aan alle eisen die gesteld zijn aan MBT tools.
- In het Microsoft.Net framework, een zeer grote bibliotheek aan voorgeprogrammeerde klassen waar in C# gebruikt van kan worden gemaakt, zitten een aantal voorgeprogrammeerde bibliotheken die de communicatie met Excel volledig kunnen verzorgen. Behalve SpecExplorer is er geen andere MBT tool die zo gemakkelijk kan communiceren met Excel. Dit is dan ook het belangrijkste argument waarom er voor NModel is gekozen.
- C# en de toegevoegde componenten zijn een sterke combinatie om op een abstracte manier complexe systemen te modelleren.
- C# redelijk bekend bij programmeurs/testers. Een aantal programmeurs zijn bekend met C-achtige programmeertalen waar C# vanaf stamt. Object oriënteert programmeren is breed bekend bij programmeurs.
- NModel beschikt over een aantal effectieve test generatie algoritmen. Hier zal op worden teruggekomen in paragraaf 5.4.2.
- NModel is volledig open-source. Hierdoor kunnen er makkelijk eigen gemaakte features aan NModel worden toegevoegd.

Het gebruik van NModel heeft ook een nadeel. NModel wordt niet meer ondersteund door Microsoft. Microsoft heeft zicht volledig gericht op het doorontwikkelen van SpecExplorer. Dit houdt in dat NModel vanuit Microsoft niet meer wordt verbeterd met handige features. Daarnaast is er weinig ondersteuning voor NModel vanuit Microsoft. Daartegenover staat dat NModel en SpecExplorer erg veel op elkaar lijken. Mocht een volgende versie van SpecExplorer wel voor commerciële doeleinde gebruikt kunnen worden dan is een mogelijke overstap van NModel naar SpecExplorer niet heel erg groot.

5.4 HOE WERKT NMODEL?

In paragraaf 5.2 zijn de vier componenten van NModel kort toegelicht. In deze paragraaf zal er dieper worden ingegaan in hoe deze componenten en eigen toegevoegde componenten werken.

Om NModel begrijpelijker te maken zal er in deze paragraaf gebruik worden gemaakt van een simpel voorbeeld. Stel we hebben een Excel applicatie genaamd “testExcel” met één sheet. Op deze sheet staat een knop die als volgt werkt:

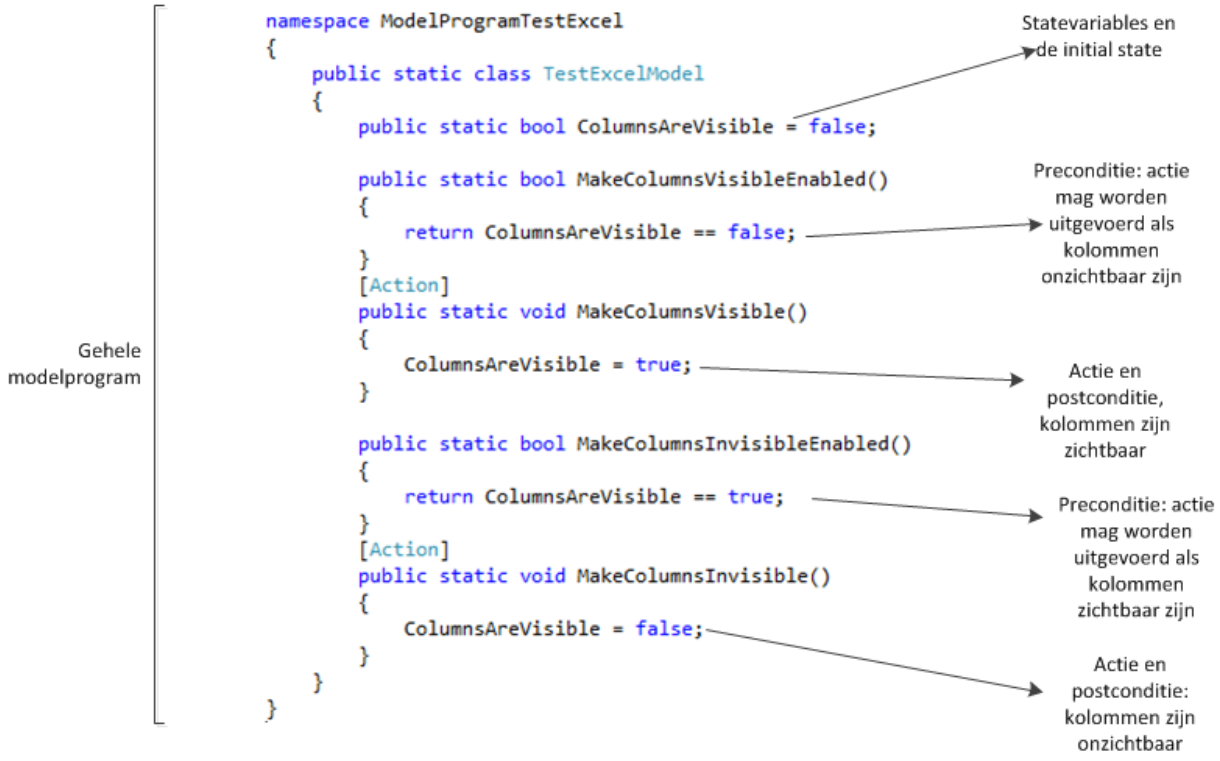
- Als de kolommen A t/m C zichtbaar zijn en er wordt op de knop gedrukt dan worden deze kolommen onzichtbaar.
- Als de kolommen A t/m C onzichtbaar zijn en er wordt op de knop gedrukt dan worden deze kolommen zichtbaar.

5.4.1 MODELLING LIBRARY

De correcte werking van een software applicatie wordt bij Model-Based testing beschreven met een model, in NModel dit model een modelprogram genoemd. Dit modelprogram heeft als basis C# en daarnaast nog een aantal componenten.

- *Actions*: In een Action beschrijft een tester een actie die uitgevoerd kan worden op de SUT. Een actie wordt beschreven met zogenaamde pre- en postcondities. De preconditie van een action beschrijft in welke status de SUT moet zijn voordat deze actie mag worden uitgevoerd op de SUT. De postconditie beschrijft de status van de SUT nadat de actie is ondernomen.
- *State variables*: De state variables van een model beschrijven in welke status een programma zich bevindt. In het "testExcel" voorbeeld heeft bijvoorbeeld 2 mogelijke statussen: Kolommen zijn zichtbaar en de kolommen zijn onzichtbaar. Elke unieke combinatie die de state variables in een model kunnen hebben levert een andere status.

  Elke software applicatie heeft een bepaalde beginstatus, vandaar dat alle statevariables ook een beginwaarde moeten hebben. De status die wordt gevormd door de unieke combinatie van beginwaarden van alle statevariables wordt de initial state genoemd.
- Overige componenten: Actions en state variables zijn de twee belangrijkste componenten om de werking van een softwareapplicatie te beschrijven. Daarnaast heeft NModel nog een grote hoeveelheid componenten om een softwareapplicatie te modelleren. Deze kunnen gevonden worden in(Campbell, Veanes, & Jacky, 2008).Omdat deze overige componenten niet veel toevoegen aan de basiswerking van NModel zijn deze componenten hier achterwege gelaten.(Jacky, Veanes, Campbell, & Schulte, 2007)

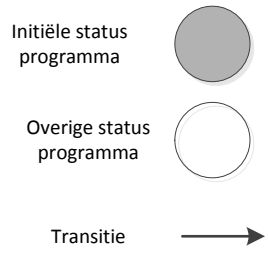


FIGUUR 3, MODELPROGRAM TESTEXCEL

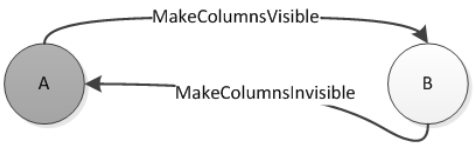
Nadat een modelprogram is opgesteld kan dit model worden omgezet in een graaf. Een graaf wordt in NModel een *Finite State Machine (FSM)*genoemd. Het maken van de graaf wordt gedaan door met een algoritme het model te verkennen(Jacky, Veanes, Campbell, & Schulte, 2007). Dit algoritme probeert elke identieke status die bereikt kan worden door het uitvoeren van een action te vinden. Elke identieke status in het model kan worden weergegeven als een knoop in een graaf. De actions brengen het model van de ene status naar een andere status en worden daarom weergegeven als zijden. Elke zijde wordt in NModel een *transitie* genoemd. Aangezien transities niet inverteerbaar zijn ontstaat er een gerichte graaf.

Het is niet noodzakelijk om het model om te zetten in een graaf. In sommige gevallen is het zelfs niet mogelijk omdat het model teveel statussen en transities heeft. Zo'n model wordt een IFSM genoemd.

In het voorbeeld van "testExcel" zou een FSM er uitzien als figuur 5

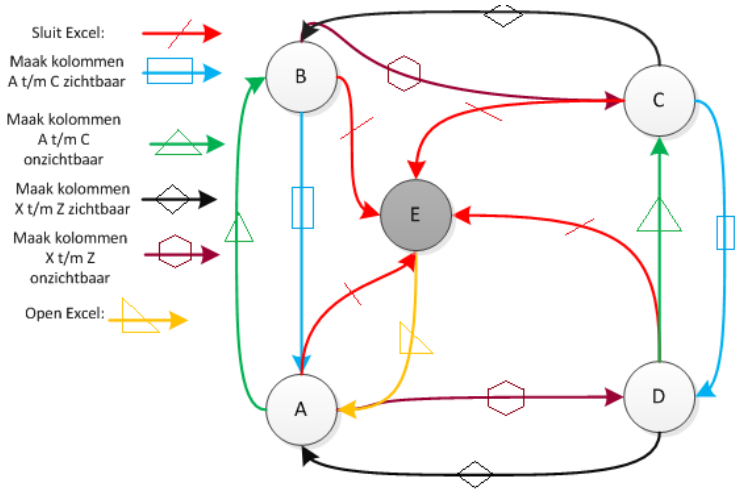


FIGUUR 4, LEGENDA GRAAF



FIGUUR 5, GRAAF TESTEXCEL MET A ALS INITIAL STATE

In figuur 5 lijkt het alsof elke actie correspondeert met één transitie. Een actie kan echter corresponderen met meerdere transities in de FSM. We breiden het "testExcel" voorbeeldprogramma uit met de mogelijkheden om Excel te openen en te sluiten. Daarnaast voegen we een knop toe die hetzelfde werkt als de al bestaande knop, alleen worden nu kolommen X, Y en Z zichtbaar of onzichtbaar.



FIGUUR 6, UITGEBREID TESTMODEL MET E ALS INITIAL STATE

Het resultaat van dit uitgebreide voorbeeld is figuur 6. In deze graaf is duidelijk te zien dat één beschreven actie in het model correspondeert met meerdere transities in de graaf.

Het moet in een modelprogram altijd mogelijk zijn om vanaf een state terug te keren naar de initial state. Door deze eis ontstaan er een grote hoeveelheden cyclussen in de graaf. Deze cyclussen komen goed van pas bij het genereren van tests. SEA's kunnen terug worden gezet naar de initial state door Excel zonder op te slaan opnieuw op te starten.

5.4.2 TEST GENERATIE MET NMODEL

NModel bevat verschillende algoritmen om vanuit een FSM testscripts te creëren. Echter is bij elk algoritme het basisidee hetzelfde. Er wordt op een bepaalde manier vanaf de initial state door de graaf gewandeld. De genomen routes worden opgeslagen en functioneren als testcases. Welk algoritme het beste kan worden gebruikt ligt aan het gemaakte modelprogram. Van de verschillende testmethodieken zal kort worden toegelicht wat ze inhouden en wanneer ze het beste kunnen worden gebruikt.

- Random walk: Er wordt volledig willekeurig door de graaf gewandeld. Als er een keuze gemaakt moet worden tussen meerdere paden dan hebben al deze paden dezelfde kans om genomen te worden.

Random walks zijn simpel en kunnen wel altijd worden gebruikt. Ook als het model een IFSM is. Bij een random walk is namelijk geen hele graaf nodig, er hoeft alleen bekend te zijn welke paden vanaf de huidige transitie genomen kunnen worden.

Een random walk stop in principe nooit, als er een status wordt bereikt waaruit geen transities vertrekken dan wordt de SUT gewoon gereset en begint een random walk opnieuw. Een tester moet daarom in het model expliciet definiëren bij welke status de random walk moet stoppen.

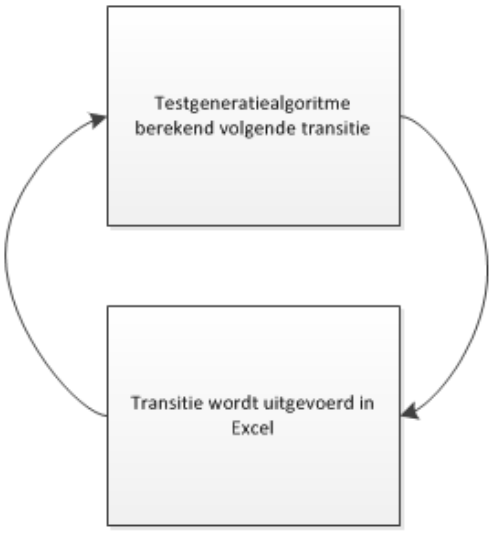
Het voordeel van random walks is vooral dat transities getest kunnen worden die een tester niet snel zou uitvoeren. Een nadeel aan random walks is dat sommige transities dubbel zullen worden getest.

- Random walk met Markov Chains: Bij het gebruik van Markov Chains wordt er opnieuw willekeurig door de graaf gewandeld. Echter heeft nu het ene pad een grotere kans om bewandeld te worden dan een ander pad. De kans dat een pad wordt bewandeld kan worden aangegeven in het gemaakte modelprogram.

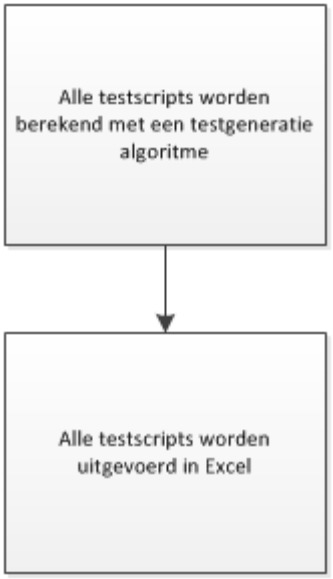
  Met Markov Chains blijft de testgeneratie random, maar toch kan een tester ervoor zorgen dat bijvoorbeeld gevoelige acties of veelgebruikte acties in de SUT een grotere kans hebben om getest te worden.
- Euler circuit: Bij een Euler circuit worden alle transities in de graaf minstens 1x doorlopen. Het vinden van een Eulercircuit in een gerichte graaf wordt ook wel het New York Street Sweeper problem genoemd. Dat alle transities minstens 1x worden doorlopen is ook gelijk het voordeel van deze methode, alle in het model gedefinieerde acties worden minstens 1x getest. Het nadeel van het gebruikt van Euler circuits is dat dit algoritme alleen toepasbaar is bij FSM omdat de gehele graaf bekend moet zijn bij het uitvoeren van het algoritme.
- Dijkstra's kortste pad algoritme: In sommige gevallen wil men alleen het pad naar één status testen. Om dit pad te bepalen kan gebruik worden gemaakt van Dijkstra's kortste pad algoritme. Ook het kortste pad algoritme kan alleen worden toegepast bij FSM omdat de hele graaf bekend moet zijn.(Robinson, 1999),(Jacky, Veanes, Campbell, & Schulte, 2007),(Utting & Legeard, 2007)

5.4.3 TEST UITVOEREN MET NMODEL

Het uitvoeren van tests kan in NModel op twee manieren gebeuren: met offline Model-Based testing of met online Model-Based testing. In deze paragraaf zal voor beide methoden worden toegelicht hoe NModel testcases uitvoert op de SUT.



FIGUUR 7, ONLINE MODEL-BASED TESTING



FIGUUR 8, OFFLINE MODEL-BASED TESTING

Bij online Model-Based testing wordt er door een testgeneratie algoritme steeds slechts één volgende uit te voeren transitie bepaald. Deze transitie wordt vervolgens gelijk uitgevoerd op de SUT. Het testgeneratie algoritme dat wordt gebruikt is random walk (wel of niet op basis van Markov Chains).

Het voordeel van Online Model-Based testing is dat het mogelijk is om zogenaamde niet deterministische applicaties te testen. Dit zijn applicaties waarbij het uitvoeren van een transitie alleen maar een bepaalde kans heeft dat de applicatie in een status A terecht komt. In andere gevallen zorgt het uitvoeren van die transitie dan het programma in status B misschien C terecht komt. Bij online Model-Based testing kan er na elke transitie aan de SUT gevraagd worden in welke status deze zich bevindt. Zo kan online Model-Based testing reageren op niet deterministische situaties. Het online Model-Based testproces ziet eruit als figuur 7.

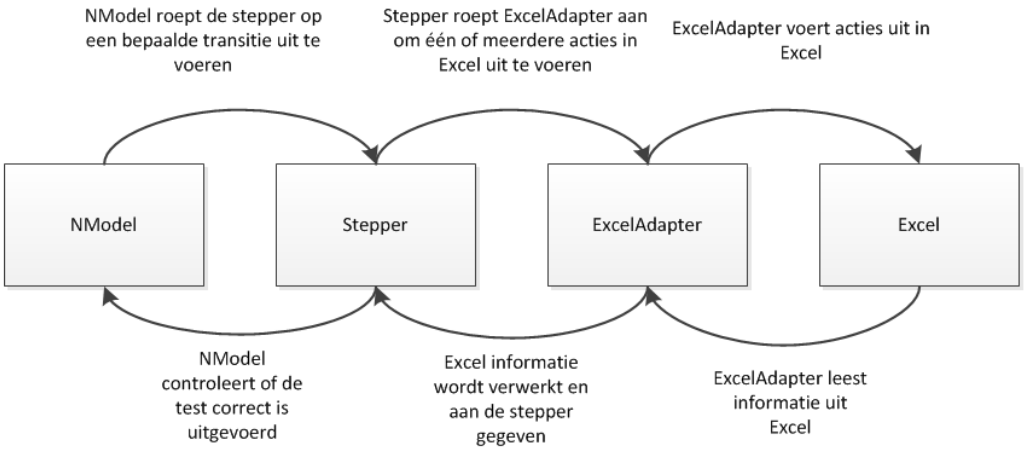
Bij offline Model-Based testing worden in één klap alle te nemen transities bepaald door een testgeneratie algoritme. De volgorde waarin deze transities moeten worden bewandeld wordt opgeslagen in een zogenaamde testsequence. Bij offline MBT wordt een algoritme voor het vinden van Eulercircuits en Dijkstra's kortste pad algoritme gebruikt. Bij offline Model-Based testing moet het model deterministisch zijn en moet er sprake zijn van een FSM. Na het bepalen van de uit te voeren transities worden deze uitgevoerd op de SUT.

Om een transitie ook daadwerkelijk uit te voeren op een SUT moet er gecommuniceerd worden met de SUT. In dit communicatieproces zitten verschillende componenten:

- NModel: NModel bepaalt welke transitie er moet worden uitgevoerd. Dit kan bepaald zijn met online MBT, waarbij NModel ook telkens de volgende transitie berekend. Bij offline MBT is het rijtje transities (de testsequence) dat moet worden uitgevoerd al bepaald en loopt NModel simpelweg dit rijtje af.
- Stepper: De Stepper zet een transitie om in opdrachten die uitvoerbaar zijn op de SUT. De Stepper is gebouwd door de tester/ontwikkelaar en is specifiek voor de communicatie met één SUT. In de Stepper wordt de transitie gelezen en geïdentificeerd welke opdracht er aan de SUT gegeven moet worden. Na het uitvoeren van de opdracht kan de Stepper ook informatie teruggeven aan NModel over de nieuwe status van de SUT.

- De Excel Adapter is een eigen gebouwde component die communicatie vergemakkelijkt tussen C# en Excel. Zo hoeft een tester/ontwikkelaar minder moeite te doen om in de Stepper acties uit te voeren in Excel.
- In Excel (de SUT) worden alle transities uitgevoerd. Een voorbeeld hiervan is de macro runnen die achter de button zit in het “testExcel” voorbeeld.

Dit communicatieproces ziet eruit als figuur 9.



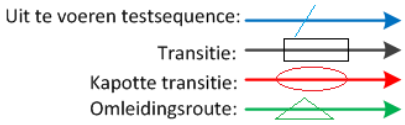
FIGUUR 9, COMMUNICATIEPROCES UITVOERING TESTCASES OP EXCEL

Het proces dat beschreven staat in figuur 9 gaat goed zolang een test niet faalt. Als NModel bezig is met online testing wordt teruggegaan naar de initial state en wordt Excel opnieuw opgestart. De kapotte transitie wordt verwijderd uit de graaf zodat deze niet meer kan worden uitgevoerd. Vervolgens wordt er weer random door de graaf gewandeld.

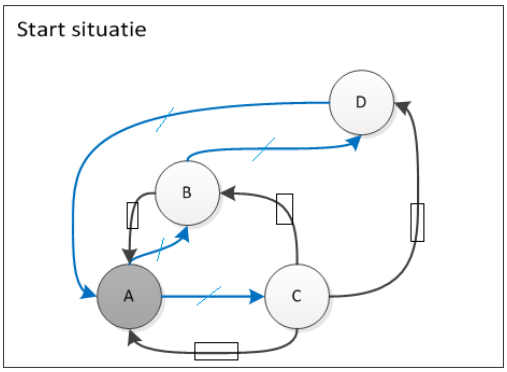
Bij offline testing is een kapotte transitie een groter probleem. Excel kan niet simpelweg worden gereset om de volgende test in de testsequence uit te voeren. Dit komt omdat de volgende test misschien in de initiële status helemaal niet kan worden uitgevoerd. Om ervoor te zorgen dat de testcase wel kan worden vervolgd worden de volgende stappen ondernomen:

1. De kapotte transitie wordt toegevoegd aan een lijst met kapotte transities.
2. Er wordt een omleidingsroute berekend.
3. Excel wordt herstart en alle eerder genomen transities tot de kapotte transitie worden uitgevoerd. Zo komt de SUT in de status voordat de fout werd gevonden.
4. De omleidingsroute wordt uitgevoerd.
5. Het normale testscript wordt vervolgd.

Het bepalen van een omleidingsroute is niet eenvoudig. De graaf verandert door het verwijderen van de kapotte transitie waardoor sommige paden niet meer kunnen worden genomen. Om dit te illustreren gebruiken we de graaf uit figuur 11.In dit voorbeeld bestaat de testsequence uit het pad: A->B->D->A->C. We willen in deze graaf de transitie A->B nemen maar deze transitie faalt. Er zijn voor dit probleem vier verschillende oplossingen:

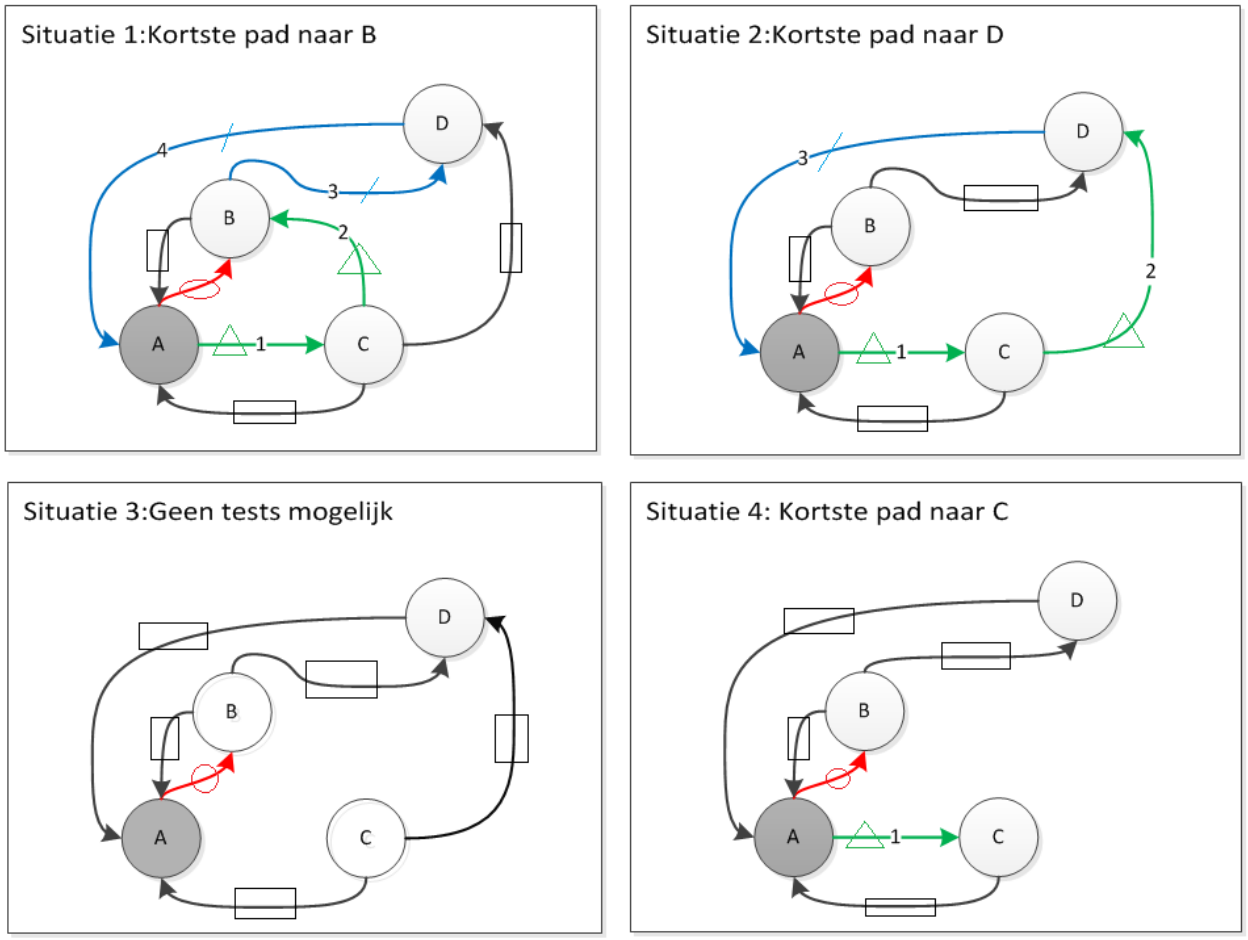


FIGUUR 10, LEGENDA OMLEIDINGSROUTE GRAFEN



FIGUUR 11, BEGINSITUATIE VOORBEELD KAPOTTE TRANSITIE. A IS DE INITIAL STATE.

1. Er is een ander pad naar knoop B: In dit geval wordt met Dijkstra's kortste pad algoritme het kortste pad naar B berekend. Dit pad is dan de omleidingsroute.
2. Er is geen pad naar knoop B. Als knoop B niet meer kan worden bereikt wordt er gekeken naar welke transitie er na knoop B in de testsequence staat. Staat er na status B geen transitie meer in de testsequence dan wordt de testsequence gestopt. Als er wel een transitie staat in de testsequence dan wordt er een kortste pad gezocht naar het eindpunt van deze transitie (in het voorbeeld knoop D).
3. Er zijn geen paden vanaf de begin state van de kapotte transitie (in dit geval A). In dit geval zijn er geen tests mogelijk en wordt de testuitvoering gestopt. Een voorbeeld van deze situatie is als Excel niet wil opstarten.
4. Stel dat er geen pad loopt van A naar knoop B en D. In dat geval wordt er gezocht naar de eerste knoop die de testsequence aandoet die wel vanaf A kan worden bereikt. In dit voorbeeld is knoop C de eerste knoop die in de testsequence staat na de kapotte transitie A->B en bereikt kan worden vanaf de beginstatus van de kapotte transitie, namelijk A. De test die wordt uitgevoerd is daarom A->C.



FIGUUR 12, FOUTAFHANDELING IN VERSCHILLENDE SITUATIES

5.4.4 HOE ZIET HET TESTRESULTAAT ERUIT?

Het testresultaat geeft een tester of ontwikkelaar niet alleen informatie over of een test geslaagd is of niet. Het testresultaat kan ook informatie geven over wat er fout is gegaan en hoe de fout gereproduceerd kan worden. Dit wordt op de volgende manier gedaan:

- Alle ondernomen transities worden opgeslagen in een logfile. Als er een test faalt kan de tester dus zien welke combinatie van transities leiden tot een bepaalde fout.
- De stepper leest informatie op vanaf de Excel Adapter. Als de Excel Adapter een fout geeft dan wordt deze geprint in de logfile. Zo wordt er informatie gegeven over welke fout zich heeft voorgedaan.

Op deze wijze wordt geprobeerd de tester zoveel mogelijk informatie te geven over een gevonden fout in de implementatie.

5.5 HOE KAN NMODEL WORDEN GEBRUIKT OM SEA'S TE TESTEN?

Het NModel framework biedt genoeg functionaliteiten om met Model-Based testing Shell Excel applicaties te testen. Deze functionaliteiten moeten wel op een goede manier gebruikt worden om de effectiviteit en efficiëntie van Model-Based testing te optimaliseren. In deze paragraaf zal worden uitgelegd hoe NModel zo effectief mogelijk gebruikt kan worden om Shell Excel applicaties te testen.

Een SEA's testen met Model-Based testing wordt gedaan in vier stappen. Deze stappen zullen kort worden doorgelicht.

Stap 1 is het afbakenen van de testruimte.

Voordat een tester begint met het maken van een modelprogram en de Stepper moet eerst duidelijk zijn waarom de applicatie wordt getest, wat er wordt getest en hoe dit wordt getest. Met het afbakenen van de testruimte wordt geprobeerd een test de juiste betekenis te geven. Zo wordt er voorkomen dat achteraf blijkt dat eigenlijk hele andere functionaliteiten getest hadden moeten worden. Een tester moet drie vragen beantwoorden om de testruimte af te bakenen:

1. Wat is het doel van het testen?
   (Met andere worden: Wat wil de tester bereiken door het testen van de applicatie?)
2. Welke features worden er getest?
3. Welk niveau van abstractie wordt er gekozen?
   Het niveau van abstractie is een belangrijke vraag die een tester zich moet stellen voordat hij/zij begint met testen. Het abstractieniveau hang nauw samen met de vraag wanneer een test geslaagd is of niet. In veel gevallen kunnen niet alle statevariables van een SUT gecontroleerd worden of deze in de correcte status zijn. Dat zou te veel tijd kosten. Een tester moet dus een keuze maken bij welke waarde van welke statevariables de SUT in de correcte status is.

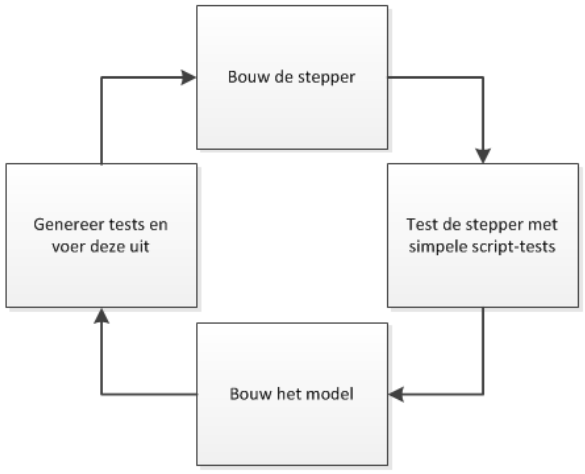
Stap 2 is de modelontwerpfase.

Tijdens deze fase worden de volgende stappen doorlopen:

- Selecteren van de te testen acties
  Elke feature bestaat uit verschillende acties. Een tester beslist welke acties er worden getest en welke niet.
- In kaart brengen van de verbanden tussen de verschillende acties.
  Een actie mag vaak pas worden uitgevoerd als een andere actie is uitgevoerd. Deze afhankelijkheden in kaart brengen helpt bij de latere implementatie van een modelprogram.
- Identificeren van de state variables.
  De afhankelijkheden tussen de acties moeten worden aangegeven met state variables. Door de state variables te baseren op de afhankelijkheden van de acties wordt voorkomen dat er onnodig veel state variables in het model worden gebruikt (en de graaf daardoor onnodig groot wordt).

Stap 3: Implementatie van het model.

Als de acties en statevariables bekend zijn dan worden deze per feature geprogrammeerd. Voor elke actie wordt eerst de communicatie geregeld met Excel in de Stepper. Als deze communicatie namelijk niet werkt dan kan de actie überhaupt niet worden uitgevoerd. Als de communicatie werkt worden de pre- en postcondities van de acties en de statevariables gemodelleerd in het modelprogram. Om te controleren of het modelprogram klopt kunnen worden er alleen voor deze feature tests gegenereerd en uitgevoerd. Gevonden fouten in de SUT hebben dan namelijk nog een grote kans veroorzaakt te zijn door fouten in het modelprogram. Dit implementatieproces ziet eruit als figuur 13.



FIGUUR 13, PROCES IMPLEMENTATIE VAN 1 FEATURE

Stap 4: Run de test voor de gehele applicatie:

Als alle features zijn gemodelleerd en getest kan voor de gehele applicatie tests worden gerund. Als het model slechts een paar features bevat kan deze het beste worden getest met offline testing. Bij grotere modellen (die een IFSM creëren) is offline MBT geen optie en zal voor online MBT gekozen moeten worden.

5.6 CONCLUSIE

NModel is een open-source Model-Based testing tool die gebruikt kan worden om applicaties die draaien op Microsoft besturingssystemen te testen. De hoofdreden waarom voor NModel is gekozen is dat er vanaf NModel een goede koppeling bestaat naar Excel en NModel een krachtige modelleertaal en testgeneratie algoritmen bezit.

In NModel wordt een SUT gemodelleerd in een modelprogram. Dit modelprogram werkt op basis van pre- post condities. Als dit model kan worden omgezet in een graaf dan wordt dit een FSM genoemd. Op een FSM kunnen algoritmen voor het vinden van een Eulercircuit en Dijkstra's kortste pad algoritme worden toegepast om tests te genereren. Als het model geen graaf kan creëren dan is het model een Infinite State Machine (IFSM) en kan wel of niet op basis van Markov Chains een random walk worden uitgevoerd op het model.

De communicatie tussen NModel en Excel wordt grotendeels verzorgd door de Stepper. Deze component vertaald een transitie in uitvoerbare opdrachten die doorgegeven kunnen worden aan Excel.

Om NModel zo optimaal mogelijk te gebruiken om SEA's te testen moeten de volgende 4 stappen worden ondernomen:

1. Afbakenen van de testruimte
2. Ontwerpen van het model
3. Implementeren van het model
4. Uitvoeren van de tests

6 NMODEL IN DE PRAKTIJK

6.1 INLEIDING

Tot dusver is Model-Based testing van Shell Excel applicaties alleen vanuit theoretisch oogpunt bekeken. In dit hoofdstuk zal een toelichting worden gegeven op de effectiviteit en efficiëntie van Model-Based testing ten opzichte van de huidige methode in de praktijk. Als eerste zal worden toegelicht hoe Model-Based testing en de huidige testmethode in de praktijk zijn vergeleken. Vervolgens zullen de resultaten hiervan geanalyseerd worden om er conclusies over te kunnen trekken.

6.2 OPZET VAN HET TESTEXPERIMENT

In deze paragraaf zal worden toegelicht hoe het testexperiment is uitgevoerd. In de voorgaande hoofdstukken is toegelicht waarom Model-Based testing in theorie de beste methode is om Shell Excel applicaties te testen. Echter hoeft dit nog niet te zeggen dat dit in de praktijk ook zo is. Het zou zo kunnen zijn dat een goed model modelleren in NModel te complex is. In dat geval kunnen er geen goede testscripts worden gegenereerd. Daarnaast willen we de theorie kunnen kwantificeren. Hoeveel sneller/effectiever is Model-Based testing wel of niet ten opzichte van handmatig testen? Uiteindelijk willen we de volgende hypothesen toetsen.

H_0: *Model Based testing is niet efficienter dan handmatig testen van SEA's zonder effectiviteit te verliezen.*

H_1: *Model Based testing is efficienter dan handmatig testen van SEA's zonder effectiviteit te verliezen.*

Om deze nulhypothese te toetsen is veel data nodig over zowel de Model-Based testing als handmatig testen. Deze data moet informatie bevatten over de verschillende maatstaven die in paragraaf 4.3 zijn geïntroduceerd. Daarnaast moeten de maatstaven gemeten zijn over een lange tijdsperiode van minstens een half jaar. Omdat deze informatie niet beschikbaar is, is het op dit moment niet mogelijk om deze hypothesen te toetsen.

Om toch iets te kunnen zeggen over de efficiëntie en effectiviteit van Model-Based testing ten opzichte van de huidige testmethode is een kleinschalig experiment opgezet. Hierbij is één Shell Excel applicatie eenmalig getest met Model-Based testing en handmatig testen. Het doel van dit experiment is niet zo zeer het toetsen of Model-Based testing een betere testmethodiek is. Belangrijker is het kunnen kwantificeren van de effectiviteit en efficiëntie van beide testmethodieken. Aan de hand van deze data kan een inschatting gemaakt worden van wat de verwachte effectiviteit en efficiëntie van beide methodieken zou kunnen zijn. Dit experiment kan dus een indicatie geven van de effectiviteit en efficiëntie van Model-Based testing ten opzichte van handmatig testen in de praktijk.

Het experiment kent de volgende opbouw:

1. In een stabiele[5] Shell Excel applicatie worden een aantal foutenopzettelijk geïmplementeerd. Van deze fouten wordt geregistreerd:
    - Bij welke acties de fout tot uiting komt.
    - In welke staat het programma verkeerd na de fout.

[5]Met stabiel wordt bedoeld een SEA die van zichzelf weinig fouten bevat. Er kan niet bewezen worden dat er helemaal geen fouten in een SEA zitten.

    - Wat voor soort fout het is (functioneel, robuust of performance)
2. De kapotte SEA wordt eerst handmatig getest, hierbij wordt bijgehouden:
    - De totale test tijd.
    - Welke acties de tester uitvoert op de SUT.
    - Welke fouten de tester tegenkomt.
    - Wat de tester denkt dat de oorzaak is van een fout.
    - Wat de tijd is tussen het vinden van een fout en het weten van de oorzaak daarvan.
3. De kapotte SEA wordt met Model-Based testing getest. Dit wordt gedaan zoals in paragraaf 5.5 staat beschreven. De volgende gegevens worden bijgehouden:
    - Totale voorbereidingstijd.
    - Totale test tijd.
    - De stappen die worden genomen door de testengine.
    - De fouten die de testengine tegenkomt.

De Shell Excel applicatie die is getest is een cost-estimating applicatie. De applicatie maakt een schatting van project kosten aan de hand van gegeven data. De tool was op het moment van dit testexperiment in ontwikkeling en zal binnenkort daadwerkelijk gebruikt worden door klanten vanuit Shell. De applicatie bestaat typisch uit een berekeningsmodule en input/output functionaliteiten. Dit testexperiment is alleen op de input/output functionaliteiten gericht. De functionaliteiten van deze SEA kunnen worden opgedeeld in de volgende features:

- ***Checkbox functionaliteiten:*** Om de gebruiker te ondersteunen in het invoeren van karakteristieken van een project kan de gebruiker checkboxes aan of uit zetten. Als een checkbox wordt aangezet dan verschijnen er bijvoorbeeld extra sheets of kolommen in Excel. Als een checkbox wordt uitgezet verdwijnen deze weer.
- ***Dropdownbox functionaliteiten:*** in sommige invoervelden is een gebruiker gelimiteerd in welke mogelijke input hij/zij in dit veld kan geven. De mogelijkheden worden dan opgesomd in een dropdownbox. Het is van belang dat deze dropdownboxes de juiste waarden bevatten.
- ***Estimate flow:*** Om een schatting te kunnen geven van project kosten wordt een gebruiker ondersteund door verschillende functionaliteiten. Voorbeelden hiervan zijn het importeren van gegevens uit een andere Excel applicatie, het valideren van de inputdata of het exporteren van gegevens.
- ***Admin rates:*** Een belangrijke input voor het schatten van projectkosten is het opgeven van de benodigde mankracht. Deze kosten voor mankracht kunnen flink verschillen in wat voor mankracht nodig is en in welke omgeving. In sommige landen zijn bijvoorbeeld personeelskosten een stuk lager dan in andere landen. Om een goede schatting te kunnen geven kan een gebruiker daarom met verschillende rates aangeven in welke omgeving de mankracht nodig is. De admin rates feature bestaat uit verschillende functionaliteiten die het gebruik van rates voor personeelskosten ondersteund.
- ***Tabel functionaliteiten:*** De gebruiker voert de meeste input in tabellen i.o.m. de gebruiker hierin te ondersteunen kunnen tabellen worden gesorteerd, het aantal rijen worden aangepast of alle data in de tabellen worden verwijderd.

  In paragraaf 4.3 zijn een aantal maatstaven geïntroduceerd die tijdens dit experiment gebruikt kunnen worden. Er zal per maatstaaf worden aangeven of deze gebruikt is. Ook zal worden toegelicht waarom meestaven niet zijn meegenomen. Daarnaast zal voor elke wel gemeten maatstaaf de eenheid waarin de maatstaaf is gemeten worden toegelicht.

Effectiviteit:

- SUT Coverage. De SUT coverage wordt per feature beoordeeld. Er wordt per feature beoordeeld of deze: niet is getest, er een aantal functionaliteiten van het totaal zijn getest, alle functionaliteiten minstens 1x zijn getest of alle mogelijke combinaties van de functionaliteiten zijn getest.
- Regressietests. Om de kwaliteit van regressietests te meten moet er meerdere malen getest worden. Het was tijdens dit testexperiment echter alleen de mogelijk om 1x te testen. Vandaar dat deze maatstaaf niet is meegenomen in de analyse.
- Aantal gevonden fouten per tijdseenheid. Omdat bij deze enkele test de totale testtijd van Model-Based testing vele malen hoger ligt dan handmatig testen door het opzetten van een model is er slechts naar het aantal gevonden fouten gekeken en niet naar de gevonden fouten per tijdseenheid. De fouten zijn opgedeeld in functionele, robuuste en performance fouten.

Efficiëntie

- Test voorbereidingstijd. Het aantal uren voorbereidingstijd is voor de onderzochte testmethodieken bijgehouden.
- Test uitvoeringstijd Het aantal uren uitvoeringstijd (ook wel runtijd genoemd) is voor de onderzochte testmethodieken bijgehouden.
- Aanpasbaarheid van de tests. Om de aanpasbaarheid van de tests te meten moet een applicatie meerdere malen worden getest. Dat is in dit testexperiment niet gedaan. Deze meetstaaf wordt daarom ook niet meegenomen in de analyse.
- Complexiteit van de methodiek. Om iets zinnigs te kunnen zeggen over de complexiteit van een testmethodiek in de praktijk zijn veel meningen van testers nodig. In dit testexperiment zijn slechts twee testers betrokken geweest. Deze meetstaaf wordt daarom niet meegenomen in de analyse.
- Tijd tussen het vinden van een fout en het vinden van de oorzaak daarvan. Tijdens dit experiment is de tijd tussen het vinden van een fout en het vinden van de oorzaak daarvan bijgehouden. Echter bleek het voor de gevonden geïmplementeerde fouten dusdanig gemakkelijk om de oorzaak van een fout te achterhalen dat deze tijd te verwaarlozen is.
- Terugkoppeling naar de requirements. InfoSystems heeft geen documentatie over de requirements. Daarom is het niet mogelijk om terug te koppelen naar de requirements.

Is het in deze paragraaf toegelichte testexperiment valide om uitspraken te doen over het testen van SEA's in het algemeen? Er zijn een aantal maatregelen getroffen om ervoor te zorgen dat de werkelijke testsituatie zoveel mogelijk is nagebootst tijdens dit experiment. Dit is onder andere gedaan door:

- Een tool te testen die kenmerken bevat die veel terugkomen in andere tools. De genoemde features zijn features die niet alleen in deze tool terugkomen, maar in heel veel SEA's. Daarom kunnen testresultaten voor deze tool gegeneraliseerd worden voor alle SEA's.
- Bij het handmatig testen van de SEA's is de gebruikelijke handmatige testmethode gebruikt. Dit houdt in dat een tester zonder testplan probeert om zoveel mogelijk functionaliteiten van de tool handmatig te testen. Verder was de tester op de hoogte van slechts een deel van de requirements. In de praktijk komt het namelijk vaak voor dat testers slechts globaal de correcte werking van het programma kennen, maar niet exact.
- De geïmplementeerde fouten zijn gebaseerd op fouten die in de praktijk zijn gevonden.

- Model-Based testing is uitgevoerd zoals in paragraaf 5.5 staat beschreven. Dit is het Model-Based testproces zoals het in praktijk geïntegreerd zou moeten worden. Daarom is deze methode valide.
- De testers hadden uiteraard voor het testen geen voorkennis over welke fouten waar in de SEA geïmplementeerd waren.

Door deze maatregelen is het mogelijk om aan de hand van dit testproces iets te zeggen over de effectiviteit en efficiëntie van de testmethodieken bij het testen van SEA's.

6.3 WAT ZIJN DE ONDERZOEKSRESULTATEN?

In deze paragraaf zal toelichting gegeven worden op de gevonden resultaten van het testexperiment. Als eerste zal er toelichting gegeven worden op de efficiëntie van beide methodieken, daarna zal de effectiviteit van beide methodieken worden toegelicht.

6.3.1 EFFICIENTIE VAN DE TESTMETHODIEKEN

	Handmatig: (in uren)	**Model-Based testing: (in uren)**
Testplan maken:	0	8
checkbox functionaliteiten modelleren	0	4
dropdown functionaliteiten modelleren	0	4
estimate flow modelleren	0	4
Totale voorbereidingstijd	0	20
runtijd:	4	0.5
Tijdkosten voor tester om tests uit te voeren:	4	0

TABEL 2, VERGELIJKING EFFICIENTIE

Tabel 2 geeft de voorbereiding en testtijd aan van beide methodieken tijdens het testexperiment. De totale voorbereidingstijd van Model-Based testing was 2,5 dag waarbij 1 dag is besteed aan het opzetten van het testplan (zie stap 1 & 2 van paragraaf 5.5). Daarna zijn 3 van de 5 features gemodelleerd in NModel. Niet alle features zijn gemodelleerd door tijdsgebrek. Er is ervoor gekozen om een beperkt aantal features concreet te beschrijven in plaats van alle features abstract. Anders bestaat het risico dat er fouten zitten in functionaliteiten die wel zijn getest. We zien dat Model-Based testing veel langer duurt dan handmatig testen. Voor één enkele test is handmatig testen dus efficiënter dan Model-Based testing.

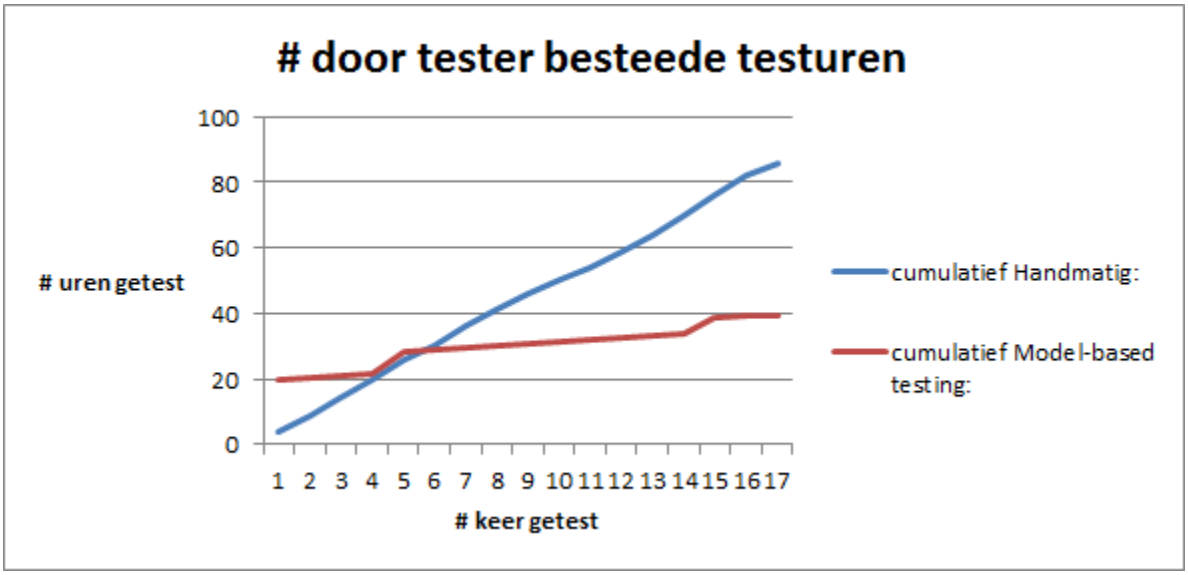
Daartegenover staat de lage runtijd van Model-Based testing opzichte van handmatig testen. Handmatig testen kostte de tester ongeveer een halve dag totdat er geen fouten meer werden gevonden. NModel heeft ongeveer een half uur gerund totdat er geen fouten meer werden gevonden. Tijdens het runnen van NModel hoeft de tester geen overige handelingen uit te voeren en kan hij/zij prima andere werkzaamheden uitvoeren mits dit geen werkzaamheden zijn op de computer die de tests uitvoert. Als tests dus vaak worden uitgevoerd zal Model-Based testing waarschijnlijk efficiënter zijn dan handmatig testen.

Om een beeld te krijgen van de totale testtijd van Model-Based testing en handmatig testen op lange termijn is aan de hand van de gevonden gegevens een simulatie uitgevoerd. Hiervoor zijn de volgende aannamen gebruikt

- De totale handmatige testtijd is uniform verdeeld tussen 3,5 en 6,5 uur per test. In de praktijk wordt een applicatie ongeveer maximaal 1 dag handmatig getest (afhankelijk van de grootte van de applicatie).

Hierbij wordt ook gelijk een poging gedaan om gevonden fouten op te lossen. Als we deze tijd eraf halen is de effectieve testtijd tussen de 3,5 en 6,5 uur.

- Het maken van een eerste model met Model-Based testing kost 20 uur. In het testexperiment kostte het 20 uur om het model te bouwen voor 3 features. De geteste tool is echter relatief groot ten opzichte van andere tools. Vandaar dat 20 uur een redelijke aanname is.
- Bij een normale test met behulp van Model-Based testing kost het opstarten van de tests een half uur. De tester moet voor het uitvoeren van de test kijken of de koppelingen naar Excel goed werken. Als een test voor het eerst wordt uitgevoerd kost dit, als er geen grote fouten zitten in de koppeling met Excel, maximaal een half uur.
- Na elke 5 tests moet het testmodel worden aangepast aan mogelijk nieuwe eisen van de gebruiker. De tijd die het kost om deze test aan te passen is uniform verdeeld tussen 1 en 8 uur. Bij grote releases wordt ongeveer de gehele applicatie 5x handmatig getest. Deze 5 tests worden uitgevoerd op dezelfde applicatie. Bij Model-Based testing zou voor deze tests dus 5x hetzelfde model kunnen worden gebruikt. De aanpastijd van het model is afhankelijk van de grootte van de wijziging. Deze aanpastijd kan een paar minuten zijn tot een hele dag.



FIGUUR 14, VERWACHTE EFFICIENTIE MODEL-BASED TESTING EN HANDMATIG TESTEN

In figuur 15 is de cumulatieve handmatige testtijd te zien ten opzichte de cumulatieve testtijd van Model-Based testing. Dit figuur is ontstaan door het uitvoeren van de simulatie. Het is duidelijk te zien dat naar mate een applicatie meer getest is Model-Based testing efficiënter is dan handmatig testen. Het snijpunt ligt rond de 5 uitgevoerde tests.

De testuitvoeringstijd en testvoorbereidingstijd voor zowel Model-Based testing als handmatig testen is afhankelijk van de complexiteit van de applicatie. Hoe meer features een applicatie heeft, des te complexer de applicatie wordt. Wat voor effect zeer complexe of niet complexe applicaties hebben op figuur 15 is aan de hand van dit testexperiment niet te zeggen. Uiteraard zal handmatig testen meer tijd per test in beslag nemen bij complexere applicaties, maar hoe dit zich verhoudt ten opzichte van Model-Based testing is niet te zeggen. De meeste SEA's

zitten echter qua complexiteit niet heel erg ver uit elkaar, hierdoor kunnen we stellen dat de situatie die in figuur 15 is geschetst voor elke SEA kan kloppen.

6.3.2 EFFECTIVITEIT VAN DE TESTMETHODIEKEN

Model-Based testing is dus efficiënter dan handmatig testen bij meer testruns. Maar is het ook effectiever? Tabel 3 geeft de gevonden fouten aan

	Handmatig:	**Model-Based testing:**
Gevonden bugs:	6	4
Waarvan gemodelleerd:	2	4
Waarvan functioneel:	4	3
Waarvan robuust:	2	1
Waarvan performance:	0	0
Fouten in checkbox functionaliteiten:	0	1
Fouten in Dropdownbox functionaliteiten:	0	1
Fouten in Estimate flow:	3	2
Fouten in admin rates	1	0
Fouten in Tabel functionaliteiten	2	0
Totaal aantal geïmplementeerde fouten:	10	10

TABEL 3, EFFECTIVITEIT TESTMETHODEN

1. Met handmatig testen zijn meer bugs gevonden dan bij Model-Based testing. Deze vergelijking is echter onterecht omdat bij handmatig testen delen zijn getest die niet waren gemodelleerd.
2. Als we de gevonden fouten reduceren tot de features die gemodelleerd zijn met Model-Based testing dan vindt Model-Based testing wel meer fouten.
3. Met Model-Based testing worden er meer fouten gevonden in features die een grote hoeveelheid aan mogelijke statussen kunnen aannemen. Zo zijn er voor 6 checkboxes $2^6 = 64$ mogelijke combinaties. Als we daar de volgorde waarin deze checkboxes worden ingedrukt ook in meenemen wordt de hoeveelheid mogelijke combinaties nog veel groter. Handmatig alle combinaties afgaan is onmogelijk. Automatisch ligt de grens van het maximaal aantal mogelijk uitvoerbare combinaties veel hoger.
4. De twee methodieken vinden andere fouten. Dit komt waarschijnlijk doordat het model een beperkt aantal features grondig heeft getest en andere features niet. In de features die niet getest zijn werd met handmatig testen voornamelijk veel fouten gevonden.

Niet getest:

Een aantal functionaliteiten getest:

Alle functionaliteiten getest:

Alle configuraties van alle functionaliteiten getest:

FIGUUR 15, LEGENDA MODEL-COVERAGE

	Coverage				
	Feature:				
Methode:	Werking checkboxes:	Dropdownboxes:	Estimate flow:	Admin rates:	tabel functionaliteiten
Handmatig:					
Model-Based Testing:					

TABEL 4, MODEL COVERAGE PER METHODE PER FEATURE

De model-coverage van beide methodieken tijdens het testexperiment ziet eruit als tabel 4. We zien dat bij handmatig testen meestal wordt geprobeerd alle functionaliteiten 1x aan te roepen. Bij Model-Based testing piekt de model-coverage bij de features waar de werking van alle functionaliteiten is beschreven. Als slechts een deel van de functionaliteiten zijn beschreven is dit terug te zien in de Model-Coverage. De model-coverage is van Model-Based testing is dus alleen beter dan bij handmatig testen als het model beschrijvend genoeg is.

Wanneer is het model beschrijvend genoeg? Het gewenste abstractieniveau is niet voor elke SEA hetzelfde. Simpelweg kan gesteld worden dat een model beschrijvend genoeg is als hiermee de doelstelling van het testen mee bereikt kan worden. Als het model niet in staat is het doel van de test te bereiken dan is dit model niet voldoende om mee te testen. Het gewenste abstractieniveau is dus afhankelijk van de doelstelling van de test.

In deze paragraaf is gekeken naar de effectiviteit van de verschillende testmethodieken. Kan deze data ook van toepassing zijn op andere SEA's? Dit is wel aannemelijk. Andere SEA's bevatten veelal dezelfde soort features als de geteste tool. Vandaar dat deze data representatief is voor het testen van SEA's in het algemeen.

6.4 MODEL-BASED TESTING, THEORIE VERSUS PRAKTIJK

In hoofdstuk 4 is een overzicht gemaakt van de theoretische effectiviteit en efficiëntie van verschillende testmethodieken. In deze paragraaf zal per eigenschap die in die tabel staat aangegeven toegelicht worden of de theorie verschild met de praktijk. Ook zal de rede van dit verschil worden aangegeven. Op die manier kan een totaaloverzicht gemaakt worden van de effectiviteit en efficiëntie van Model-Based testing en handmatig testen.

Als we de theorie vergelijken met de praktijk dan zijn er de volgende overeenkomsten:

- De voorbereidingstijd, deze is inderdaad bij Model-Based testing hoger dan bij handmatig testen.

- Test uitvoeringstijd, de theorie wijst erop dat het handmatig uitvoeren van een test veel tijd kost. Het uitgevoerde testexperiment wijst dit ook uit.
- In staat fouten te vinden, zowel de theorie als de praktijk wijst uit dat met Model-Based testing meer fouten gevonden kunnen worden in de SUT.
- De complexiteit van de testmethodieken, Model-Based testing is veel complexere methodiek dan handmatig testen. Als gebruik gemaakt wordt van NModel is er kennis van pre- en post conditie modellen nodig en van C#. Bij handmatig testen is alleen kennis nodig over de correcte werking van de applicatie.
- Mogelijkheid tot regressietesten, bij Model-Based testing kunnen dezelfde tests meerdere malen worden uitgevoerd. Op deze manier kan heel effectief gekeken worden of niet gewijzigde functionaliteiten nog steeds correct werken. Bij handmatig testen kan niet exact dezelfde test worden herhaald omdat niet wordt bijgehouden welke stappen de tester ondernomen heeft. De praktijk wijst dus net als de theorie uit dat regressietesten met Model-Based testing beter gaat dan met handmatig testen.
- SUT coverage, Model-Based testing is in staat veel meer combinaties uit te proberen dan handmatig testen. In de praktijk bleek dit ook tot een hogere model-coverage te leiden dan handmatig testen.

Er zijn ook verschillen:

- In staat om complexe systemen te testen, het is gebleken dat met pre- en postcondities het lastig is om complexe systemen te testen. Het is mogelijk dat met andere modellen dit wel kan, maar daar kan in dit onderzoek niets over gezegd worden. De hogere score voor het testen van complexe systemen in de theorie is dus in de praktijk niet altijd terecht.
- In staat om de oorzaak van fouten te achterhalen, in dit testexperiment was het voor zowel handmatig als Model-Based testing vrij gemakkelijk om de oorzaak van een fout te achterhalen. Dit komt doordat elke geïmplementeerde fout werd veroorzaakt door een duidelijke oorzaak. Uit dit onderzoek is daarom niet goed vast te stellen of beide testmethodieken ook de oorzaak van fouten kunnen achterhalen als deze oorzaak niet voordehand liggend is.

6.5 Conclusie

In dit hoofdstuk is toegelicht hoe Model-Based testing en handmatig testen zijn vergeleken in de praktijk. Met behulp van een testexperiment zijn gegevens over de efficiëntie en effectiviteit van beide methodieken bijgehouden en vergeleken.

Uit de vergelijking van de twee testmethodieken komt naar voren dat Model-Based testing bij meerdere testruns efficiënter is dan handmatig testen. Het break-even punt ligt ongeveer rond de vijf uitgevoerde tests. De effectiviteit van Model-Based testing erg afhankelijk van het model dat is gebouwd. Als een model de werking van een applicatie voldoende beschrijft dan is de effectiviteit beter. Is dit niet het geval dan is handmatig testen effectiever.

De theorie over de effectiviteit en efficiëntie van Model-Based testing en handmatig testen komt grotendeels overeen met de praktijk. De verschillen liggen in hoeverre beide methodieken in staat zijn om complexe systemen te testen en in hoeverre de methodieken in staat zijn om de oorzaak van fouten te achterhalen.

7 CONCLUSIE

In de inleiding is de onderzoeksvraag van dit onderzoek geïntroduceerd: ***"Kan het testproces van Shell Excel applicaties worden versneld als er gebruik gemaakt wordt van Model-Based testing onder de voorwaarde dat de kwaliteit van Shell Excel applicaties minstens gelijk blijft? "***. Deze hoofdvraag bleek te complex om in één keer te beantwoorden en is daarom opgedeeld in deelvragen. In deze conclusie zal elke deelvraag beantwoord worden om zo de hoofdvraag van dit onderzoek te beantwoorden.

De eerste deelvraag luidt: ***"Wat is het huidige ontwikkel- en testproces van Shell Excel Applicaties?"*** Het huidige ontwikkelproces gebeurd met de agile ontwikkelmethode. Deze methode heeft veel weg van de Deming cyclus waarbij de processen "check" en "act" in één cyclus meerdere malen kunnen voorkomen. Het testen van Shell Excel applicaties gebeurd met handmatig testen en capture and replay testing.

De tweede deelvraag is: ***"Wat is de gewenste situatie?"*** Om het testproces van Shell Excel applicaties te versnellen zonder dat dit ten koste gaan van de kwaliteit van de Shell Excel applicaties wil InfoSystems gebruik gaan maken van testprofiles. Deze testprofiles geven in een programmeertaal, natuurlijke taal of model aan wat er getest moet worden. Een softwareapplicatie kan aan de hand van deze testprofiles de tests automatisch uitvoeren.

Deelvraag drie is: ***"Waarom is Model-Based testing in theorie de beste methode om Shell Excel applicaties te testen?"*** Om dit te bepalen is de theoretische efficiëntie en effectiviteit van verschillende testmethodieken onderzocht. Model-Based testing bleek van deze methodieken de beste omdat:

- Model-Based testing een automatische testmethodiek is. Daardoor heeft Model-Based testing veel tijdswinst in testuitvoering ten opzichte van niet-automatische methodieken.
- Model-Based testing modelleert een programma in plaats van het schrijven van grote hoeveelheden testscripts. Dit modelleren is efficiënter dan het schrijven van testscripts.
- Model-Based testing is in staat om veel mogelijke inputcombinaties te testen. Daarom is Model-Based testing in theorie effectiever dan andere testmethodieken.

Deelvraag vier luidt: ***"Hoe kan met Model-Based testing Shell Excel applicaties getest worden?"*** Dit kan met het Model-Based testing framework NModel. Met NModel kan een Shell Excel applicatie gemodelleerd worden met pre- en post condities in de programmeertaal C#. Vandaaruit is NModel in staat om testscripts te genereren en deze uit te voeren op in Shell Excel applicatie. Na het uitvoeren van een actie op een Shell Excel applicatie kan gekeken worden of de Shell Excel applicatie het juiste gedrag heeft vertoond.

De laatste deelvraag is: ***"Is Model-Based testing in de praktijk efficiënter dan de huidige testmethodiek zonder effectiviteit te verliezen?"*** Om deze vraag te beantwoorden is een testexperiment uitgevoerd waarbij de efficiëntie en effectiviteit van zowel handmatig testen als Model-Based testing zijn vergeleken. Uit dit testexperiment is het volgende gebleken:

- Model-Based testing is pas nadat ongeveer 5 tests zijn uitgevoerd efficiënter dan het handmatig testen van Shell Excel applicaties. Dit komt doordat het veel tijd kost om het model op te zetten.
- Model-Based testing is effectiever dan handmatig testen onder de voorwaarde dat de Model-Based testscripts de doelstelling van het testen kunnen bereiken.

Uit de antwoorden van de deelvragen kan de volgende conclusie worden getrokken over de hoofdvraag: ”Kan het testproces van Shell Excel applicaties worden versneld als er gebruik gemaakt wordt van Model-Based testing onder de voorwaarde dat de kwaliteit van Shell Excel applicaties minstens gelijk blijft? ”

Uit de resultaten van dit onderzoek kan geconcludeerd worden dat dit inderdaad het geval is mits er aan de volgende eisen wordt voldaan:

- Er moet sprake zijn van een Shell Excel applicatie die minstens 5x handmatig zal worden getest.
- Het model moet de werking van de Shell Excel applicatie voldoende concreet beschrijven. Als een model te abstract de correcte werking van een programma beschrijft zijn de tests niet in staat om specifieke fouten in de Excel applicatie te vinden. Een model beschrijft een Shell Excel applicatie voldoende als de met het model gegenereerde testscripts de doelstelling van het testen kunnen bereiken.

8 AANBEVELINGEN

Aan de hand van dit onderzoek kunnen ook een aantal aanbevelingen worden gedaan omtrent het testen van Shell Excel applicaties.

- Onderzoek hoe Model-Based testing het beste geïntegreerd kan worden in het huidige testproces. Om Model-Based testing goed te integreren in het huidige testproces zullen verschillende processen in gang worden gezet. Met name kennisoverdracht is hierbij belangrijk. De manier waarop Model-Based testing wordt geïmplementeerd in het huidige testproces zal bepalend zijn voor het succes met Model-Based testing.
- Veel modellen en communicatie met Excel is grotendeels hetzelfde. Door slim gebruik te maken van deze overlap kan het modeleren van een Excel applicatie worden versneld. Onderzoek hoe optimaal gebruik gemaakt kan worden van eerder gemaakte modellen.
- Maak altijd een testplan onafhankelijk van welke testmethodiek wordt gebruikt. Een van de nadelen van het huidige testproces is dat er behalve de gevonden fouten niets wordt geregistreerd over een uitgevoerde test. Hierdoor is het moeilijk om achteraf te bepalen welke onderdelen van de applicatie op welke manier zijn getest. Door te definiëren welke features tijdens een test op welk manier zijn getest kan de kwaliteit van de tests worden gemeten zonder dat dit veel tijd hoeft te kosten. Om een testplan te maken kunnen stap 1 en 2 uit paragraaf 5.5 worden gebruikt. Door een testplan te maken kan er garandeert worden dat er aan een bepaalde kwaliteitseis wordt voldaan.
- De effectiviteit van Model-Based testing is, als gebruikt gemaakt wordt van NModel, sterk afhankelijk van de mogelijkheid om de werking van een applicatie in pre- en postcondities te beschrijven. Test deze moeilijk te testen functionaliteiten met capture and replay testing. Bepaal hierbij verschillende testinputs waarbij een functionaliteit zowel op functie, op robuustheid en op performance kan worden getest. Op die manier kunnen ook complexe functionaliteiten automatisch worden getest.
- Model-Based testing is de afgelopen jaren flink in ontwikkeling. Hierdoor zullen er ongetwijfeld nog meer nuttige manieren beschikbaar komen om efficiëntere en/of effectiever Shell Excel applicaties te testen. Blijf daarom investeren in onderzoek naar Model-Based testing en software testmethodieken in het algemeen.

9 APPENDIX A, OVERZICHT MODEL-BASED TESTING TOOLS

In figuur 16 is een overzicht van Model-Based testing tools te zien (zowel commercieel als niet commercieel). Lang niet alle tools zijn volledig onderzocht omdat deze tools niet geschikt zijn om Shell Excel applicaties te testen.

Testengines:	Model:	Website:	Offline testing mogeljik	Hoe goed is de tracebility?	Welke documentatie wordt geschreven?	Hoeveel tijd gaat er in implementatie zitten?	Kan het op Shell machines runnen?	Beschikbaarheid v/d tool, kan het binnen de tijd doen wat we willen?	Kosten	opmerking kosten	(verwachte) kwaliteit/robuust heid van (de resultaten van) het testen	Het gemak voor het opstellen/aanpas sen test profiles/test cases	Het gemak bij het gebuik van testengine	Tot hoever zijn ORTEC/Shell bekend met de code v/d testscripts
Smart testing	B-notatie/UML 2.0	www.smartesting.com	ja	++	+	+	++	+	€ 3.300,00	per license	+	+	+	+-
MaTeLo	Markov Chain	www.all4tec.net												
Conformiq Qtronic	UML	www.conformiq.com	ja	+	++	+	++	++	€ 28.000,00	per license	++	+	++	+-
Reactis	Mathlab simulink stateflow	www.reactive-systems.com												
Spec Explorer	Spec#	http://visualstudiogallery.msdn.microsoft.com/271d0904-f178-4ce9-956b-d9bfa4902745/												
Case Maker	data domain	www.casemaker.com												
Rave	Tabular notation	www.t-vec.com												
Rhapsody ATD TUA tester	TTCN-3	www.telelogic.com/products/tau/tester/index.cfm												
test Cover	data domain	www.testcover.com												
ModelJUnit	FSM	http://www.cs.waikato.ac.nz/~marku/mbt/modeljunit/	ja	-	--	-	?	-	€ 0,00	open source	-+	-	-	-
Sogeti Cover	activity diagrams	http://www.sogeti.nl/onze-diensten/testen-en-kwaliteitszorg/model_based_services/	?	?			?							
DTM (DT)	Activity diagrams	http://www.dtmtool.com/#!/	ja						€ 1.100,00	single license				
Torx	??													
Axini	Ruby	http://www.axini.com/	++	++	++	-	++	-	€ 14.400,00	Vanaf per jaar	++	++	++	++
ATOS Temppo	FSM	http://atos.net/en-us/solutions/business_integration_solutions/test-and-acceptance-	ja	+	+	+-	++	+-	€ 9.500,00	single license	+-	+-	+	-
Testoptimal	FSM	http://testoptimal.com/TestOptimalSupport.html	ja	+	+	+-	+-	+	€ 1.995,00	single license	+	+	+	+-
Nmodel	FSM	http://nmodel.codeplex.com/	ja	-	--	-	+	+-	€ 0,00	open source	+	-	-	++

FIGUUR 16, OVERZICHT MODEL-BASED TESTING TOOLS (ESLAMIMEHR, 2008) EN (SHAFIQUE & LABICHE, 2010)

10 APPENDIX B, GEBRUIKTE LITERATUUR

Campbell, C., Veanes, M., & Jacky, J. (2008). *NModel reference.*

Eslamimehr, M. M. (2008). The survey of model based testing and industrial tools.

Jacky, J., Veanes, M., Campbell, C., & Schulte, W. (2007). *Model-Based software testing and analysis with C#.*

Robinson, H. (1999). Graph theory techniques in Model-based testing.

Shafique, M., & Labiche, Y. (2010). *A Systematic Review of Model Based Testing Tool Support.* Ottawa: Carleton University.

Utting, M., & Legeard, B. (2007). *Practical model-based testing: A tools approach.* Morgan Kaufmann.