Testautomatisering wint aan populariteit. Doorgaans is testautomatisering niets meer en niets minder dan het automatisch runnen van testscripts. De testscripts en het daaraan voorafgaande testontwerp moeten echter door testanalisten worden geschreven én onderhouden. Wat zou het gemakkelijk zijn als zowel het testontwerp als de testscripts met één druk op de knop gegenereerd zouden kunnen worden. Dit klinkt misschien als een utopie, maar uit dit artikel zal blijken dat onder meer binnen het TT-Medal project een aantal belangrijke stappen in deze richting zijn gezet.

Testen met één druk op de knop

UML en technieken voor automatisch testen

Automatische generatie van test cases is een belangrijk onderdeel van het Europese TT-Medal project. Dit artikel is gebaseerd op de resultaten van het onderzoek binnen dit project. Meer algemene informatie over TT-Medal en TTCN-3 is te vinden in het artikel "Automatisch testen met de testspecificatietaal TTCN-3"[1].

De opzet van dit artikel is als volgt. Na een korte introductie over het gebruik van UML volgt een algemene beschrijving van testen en in het bijzonder testtechnieken. Wat zijn testtechnieken? Welke worden er gebruikt? Wanneer de basis is gelegd kunnen testtechnieken gekoppeld worden aan UML-diagrammen. Ten slotte komen de mogelijkheden van het automatisch genereren van test cases aan de hand van UML-diagrammen met behulp van testtechnieken aan de orde.

UML Zowel bij het handmatig schrijven van test cases als voor het automatisch genereren van test cases is een specificatie van het systeem nodig. De Unified Modelling Language (UML) van de Object Management Group (OMG[2]). kan in beide situaties worden gebruikt. Al zullen bij automatische generatie van test cases de eisen aan de specificatie anders zijn. Hier wordt verder in dit artikel dieper op ingegaan.

UML IN DE PRAKTIJK Theorie en praktijk gaan zelden gelijk op. Het is dan ook logisch je af te vragen hoe UML in de praktijk wordt gebruikt. Volgens Gartner hanteert 7 à 10% van alle internationale producenten van software UML voor het modelleren van hun systemen. Een ander onderzoek, uitgevoerd door Fowler[3], onderzocht niet hoe vaak, maar hoe de UML diagrammen in de praktijk worden gebruikt. Daarvoor maakte hij een onderscheid in drie modes: 'Sketch', 'Blue Print' en 'Programming Language'. De conclusie van Fowler is dat de minst formele mode, de 'Sketch mode', verreweg het populairst is. In 'Sketch mode' wordt slechts een kleine selectie van UML-diagrammen gebruikt, waarbij men vaak afwijkt van de specificatie van de Object Management Group (OMG). Uit een onderzoek binnen TT-Medal bleek dat de Use Case, Class en Sequence-diagrammen het meest worden gebruikt.

'Programming language' is vergeleken met de andere modes de meest formele mode, waarbij alle UML-diagrammen volgens de OMG-specificatie worden toegepast om het systeem in zijn geheel te specificeren. Daarom is deze mode zeer geschikt wanneer men automatisch test cases wil genereren. Fowler geeft echter aan dat 'Programming language' slechts in academische omgevingen wordt gehanteerd en hij verwacht niet dat de mode snel in het bedrijfsleven zal worden toegepast.

'Blue print' is een mode die tussen de twee genoemde uitersten valt. Er wordt een compleet ontwerp gemaakt, maar de mate van detail kan variëren. Bij het bepalen hoe UML gebruikt kan worden om test cases te genereren is het dus heel belangrijk om te kijken op welke manier de diagrammen gebruikt worden en op welke manier ze gebruikt zouden moeten worden.

TESTTECHNIEKEN Alvorens we op testtechnieken en het gebruik ervan in te gaan volgt een definitie van testen: testen is het proces bestaande uit alle life cycle

activiteiten, zowel statisch als dynamisch, die gericht zijn op planning, voorbereiding en evaluatie van software-producten en gerelateerde werkproducten om te bepalen of zij aan de gespecificeerde systeemvereisten voldoen, om te demonstreren dat de producten geschikt zijn voor het beoogde doel (fit for purpose) en om fouten te vinden. (BCS[4])

Deze definitie van testen geeft aan dat testen een volledige discipline is. Dit artikel richt zich enkel op de dynamische evaluatie van software en de voorbereiding daarop. Voorbereiding is noodzakelijk om vooraf een onderbouwde keuze te kunnen maken welke aspecten van een systeem in welke mate zullen worden getest. Mede door beperkingen qua tijd en geld is het onmogelijk alles te testen. Tijdens de voorbereiding wordt een testontwerp gemaakt wat een selectie en specificatie van test cases bevat die aan een testdoelstelling voldoen.[5]

Testtechnieken zijn nodig om op een gestructureerde manier test cases af te leiden. De kans dat situaties dubbel worden getest of juist worden vergeten, verkleint bij het gebruik van testtechnieken. Daarbij is men minder afhankelijk van een individuele tester. Bij het gebruik van een testtechniek zullen verschillende testers dezelfde test cases afleiden. Binnen diverse testtechnieken zijn variaties mogelijk die van invloed zijn op de dekkingsgraad. Een hogere dekkingsgraad houdt in dat meer situaties zijn afgedekt, maar het betekent ook meer werk. Inzicht in de dekkingsgraad is heel belangrijk om een realistische uitspraak te doen over de kwaliteit van een systeem. Een hogere dekkingsgraad van de test geeft meer vertrouwen in de uitspraak en levert daardoor minder risico op. Dit lagere risico gaat ten koste van tijd en geld.

Een testtechniek bestaat altijd uit twee stappen: eerst worden de te testen situaties geïdentificeerd (logische test cases) en vervolgens worden de concrete inputwaarden ingevuld (fysieke test cases).

Testtechnieken in de praktijk Internationaal gebruikt 56% van de software bedrijven (simpele) testtechnieken om hun software te testen.[6] De diverse testtechnieken kan men voor verschillende doeleinden gebruiken. Binnen het TT-Medal project is gekeken naar welke testtechnieken in de praktijk het meest voorkomen. Test Use Cases en Syntax Testing worden het meest gebruikt, maar ook Equivalence Partitioning (EP) en Boundary Value Analysis (BVA) zijn behoorlijk populair. Het onderzoek naar het automatisch genereren van test cases richt zich daarom vooralsnog enkel op deze technieken.

Testtechnieken gekoppeld aan UML Bij het beoordelen van de toepasbaarheid van testtechnieken op UML-diagrammen vormt UML 2.0 het uitgangspunt. Alleen de meest gebruikte testtechnieken en UML-diagrammen zijn bij het beoordelen van logische koppelingen in acht genomen. De resultaten staan in de tabel. Per testtechniek is bekeken welke informatie uit welke UML-diagrammen nodig is. Zo beschrijven Activity, Sequence en State-diagrammen conditioneel gedrag dat voor Equivalence Partitioning, Boundary Value Analysis en Syntax testen essentieel is. Class-diagrammen beschrijven de structuur van een systeem en niet het dynamische gedrag. Daarom lijken ze minder geschikt voor dynamisch testen. Toch bevatten Class-diagrammen waardevolle informatie over variabelen zoals het type. De relatie tussen State-diagrammen en State Transition Testing ligt redelijk voor de hand: beide zijn gebaseerd op een state model. Tenslotte kunnen Sequence en Activity diagrammen evenals Use Cases de interactie tussen het systeem en de gebruiker weergeven. De interactie tussen gebruiker en systeem is belangrijk voor de Syntax en Semantic testtechnieken, maar zeker ook voor de Test Use Cases. De hier beschreven koppelingen zijn een indicatie dat het genereren van test cases vanuit UML-diagrammen met behulp van testtechnieken haalbaar zou kunnen zijn. We zullen daar in het hierna volgende nader op ingaan.

Test generatie TT-Medal is niet het enige project dat gericht is op het automatisch genereren van test cases. UML 2 Test Profile (U2TP) van OMG is hier een voorbeeld van. U2TP maakt geen gebruik van testtechnieken wat het vrijwel onmogelijk maakt op voorhand het vertrouwen dat men in de test heeft in te schatten. Bij het TT-Medal onderzoek zijn testtechnieken wel in acht genomen. Uitgaande van de koppeling uit Tabel 1 is per testtechniek in detail nagegaan welke informatie waar in de UML diagrammen te vinden is, maar ook welke informatie ontbreekt. Voor Test Use Cases en Equivalence Partitioning gaan we kort op de voor- en nadelen van automatische testgeneratie in.

Test Use Cases Een Use Case beschrijft de typische acties tussen een gebruiker en het systeem. Een Use Case bestaat uit een diagram en een tekstuele beschrijving, waarbij de tekstuele beschrijving verreweg het belangrijkst is, omdat hier de daadwerkelijke acties worden beschreven[7]. Binnen een Use Case zijn drie soorten scenario's te onderscheiden: een basis scenario waarbij de meest voorkomende manier hoe er met het systeem gewerkt wordt is beschreven, een alternatief scenario en een exceptioneel scenario waarbij foutsituaties worden beschreven. Volgens Collard[8] leidt ieder scenario tot een aparte test case. Aan het genereren van deze test cases zitten toch enkele haken en ogen:

- OMG biedt geen formele specificatie voor de tekstuele beschrijving in UML,
- volgens OMG mogen interface items geen onderdeel zijn van de specificatie,
- natuurlijke taal maakt het automatiseren complex.

Omdat er geen specificatie is voor de tekstuele beschrijving in UML-diagrammen kan iedereen zijn eigen draai eraan geven. Een generator van test cases kan nooit met zo'n variërende input omgaan. Een template kan in deze situatie uitkomst bieden. Het feit dat items uit de interface niet in de specificatie genoemd mogen worden, wat overigens voor alle UML-diagrammen geldt, bemoeilijkt het generatieproces. Zonder knoppen te specificeren zou men nooit verder komen dan het genereren van logische test cases.

Het gebruik van natuurlijke taal is zeker voor klanten ideaal, maar bij het genereren van test cases zou men er nadeel van kunnen ondervinden. Zodra verschillende acties gecombineerd worden kan het voor de menselijke lezer een onleesbaar geheel worden. Bij het opstellen van de specificatie dient men dit te voorkomen. Met andere woorden: Use Cases zijn te informeel om direct Test Use Cases te kunnen genereren; extra maatregelen zijn noodzakelijk zoals het formaliseren van de specificatie en het toestaan van interface items in de specificatie.

Equivalence Partitioning Equivalence Partitioning (EP) is een formelere techniek dan Test Use Cases. Het idee is dat het inputdomein volgens de specificatie in diverse gelijkwaardige (equivalente) klassen wordt verdeeld. Alle inputattributen binnen een equivalentieklasse zullen op dezelfde manier door het systeem worden verwerkt en daarom leiden tot hetzelfde resultaat. Het testen van meerdere inputattributen binnen één equivalentieklasse zal daarom niet leiden tot het vinden van meer fouten.

Bij EP ligt de focus op ongeldige inputattributen, om te kijken of het systeem daar goed op reageert. Voorbeelden van inputattributen zijn velden op een scherm, (systeem-)parameters en interface parameters. Bij velden op het scherm loop je tegen hetzelfde euvel aan als bij Test Use Cases: interface items zijn geen onderdeel van de UML. Zodra de inputattributen zijn geïdentificeerd kunnen met behulp van de regels van Equivalence Partitioning (Kit[9]) de equivalentieklassen worden bepaald. Ieder type attribuut (Boolean, Integer, Set) kent zijn eigen regel. Zo wordt er voor een grenswaarde een test case boven de grens en een onder de grens opgesteld. Het type van diverse attributen staat in het Class-diagram. Bij sommige inputattributen zijn de condities waarin ze worden gebruikt relevant, de condities zijn terug te vinden in de Activity, Sequence en State diagrammen. Nu de relevante informatie (automatisch) is verzameld kunnen met behulp van de derivatieregels de test cases worden gegeneerd. Let wel dat ontbrekende gegevens een negatieve invloed hebben op de coverage die kan worden bereikt. Een laatste stap in EP is output partitioning, waarbij gecheckt wordt of iedere mogelijke output is afgedekt door een test case. Het achterhalen welke outputwaarden de huidige test cases afdekken is eenvoudig omdat deze test cases al bekend zijn. Het bepalen welke outputwaarden nog meer mogelijk zijn is lastiger om te automatiseren. Condities uit de Activity, Sequence en State diagrammen kunnen helpen bij het bepalen van de outputwaarden. Al met al is EP zeer geschikt voor het automatisch genereren van test cases. De meeste informatie is in de verschillende UML-diagrammen terug te vinden. Het ontbreken van items in de interface is het grootste dilemma, samen met de problematiek van het bepalen van de verwachte outputwaarden.

Conclusie In dit artikel is het automatisch genereren van test cases besproken waarbij gebruik gemaakt wordt van UML en testtechnieken. Om succesvol test cases te kunnen genereren uit UML-diagrammen zal men minimaal de 'Blue Print' mode moeten hanteren. Zelfs dan blijkt dat sommige testtechnieken informatie vereisen die niet uit UML-diagrammen is te herleiden. Met name het ontbreken van specificatie van de interface maakt dat enkel logische test cases kunnen worden afgeleid en geen fysieke. Deze problemen maken automatisch testgeneratie nog geen utopie. Met enkele aanpassingen aan het gebruik van UML is op korte termijn al heel veel mogelijk.

	Use Case Diagrams	Class Diagrams	Sequence Diagrams	State Diagrams	Activity Diagrams
Test Use Cases	++		++		+
Syntax Testing	+	+			
Equivalence Partitioning		++	+	+	+
Boundary Value Analysis		++	+	+	+
Semantic Test			+	+	+
State Transition Testing			+	++	

Relatie testtechnieken en UML-diagrammen.

Jeanne Hofmans en Rob Hendriks zijn werkzaam bij Improve Quality Services BV

1 Automatisch testen met de testspecificatietaal TTCN-3, Mark van der Zwan, Bits & Chips, 1 september 2005

2 OMG, UML 2.0 Superstructure Specification - Final adopted specification, ptc/03-08-02, 2003.

3 Fowler, M., UML Distilled - Third edition, Addison-Wesley, ISBN 0-321-19368-7, 2003.

4 British Computer Society (BCS), Working draft: glossary of terms used in software testing, version 0.5, 2004.

5 Hetzel, B., The Complete Guide to Software Testing, second edition, 1988, QED Information Sciences, Inc. ISBN 0-89435-242-3.

6 Van Uden, W., Koomen, T., Testprocesverbetering loont, Computable nummer 10, 2005

7 Fowler, M., UML Distilled - Third edition, Addison-Wesley, ISBN 0-321-19368-7, 2003.

8 Collard, R., Developing test cases from use cases, Software Testing and Quality Engineering, July/August 1999

9 Veenendaal, E. van. ,The Testing Practitioner, UTN Publishers, ISBN 90-72194-65-9, 2002