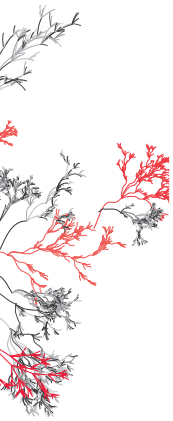
17 november 2014



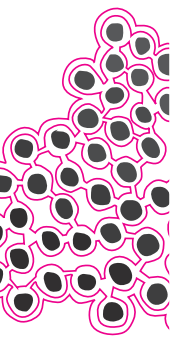
STAGEVERSLAG

TEST GUIDANCE FOR THE NEDAP TEST AUTOMATION TOOL

Jeroen Vonk

NEDAP
University of Twente

Comité:
Daan van Beek MSc (Nedap)
dr.ir. Axel Belinfante (FMT, University of Twente)
Gijs Kant MSc (FMT, University of Twente)



UNIVERSITY OF TWENTE.

Inhoudsopgave

Hoofdstuk 1

Stage bij Nedap

In het kader van mijn master Computer Science heb ik de afgelopen periode stage gelopen bij Nedap Security. Gedurende een twintigtal weken ben ik bezig geweest met het ontwerpen van een systeem om het automatisch testen van software te versnellen. Het systeem, Nedap Test Automation (NTA), is een in-house test tool ontworpen door Daan van Beek. De lezer van dit verslag wordt geacht een minimale kennis te beschikken betreffende model checken, inhoudende dat de gebruiker bekend is met termen als: (bounded) model checking, model driven tests, use cases, en (Reduced Ordered) Binary Decision Diagrams (BDD's) [1].

1.1 AEOS

Het doel van NTA is het "Model Driven" testen van de AEOS-dashboard. AEOS is het integrale beveiligingssysteem van Nedap. AEOS is een volledig softwareplatform dat met behulp van zogeheten controllers een veelvoud aan functies kan vervullen. Dit kan toegangscontrole, al dan niet met RFID, inbraakdetectie, of het beheer van persoonlijke kluisjes zijn. Om personen in het systeem in te voeren of bepaalde rechten toe te kennen, is er een dashboard. Het AEOS-dashboard is een in flash ontwikkelde applicatie. Dit dashboard communiceert vervolgens met de achterliggende back-end van AEOS. Gezien AEOS continu doorontwikkeld wordt, en sommige van deze nieuwe functies ook invloed hebben op het functioneren van het dashboard is het noodzakelijk dat het dashboard getest wordt. Het testen van het dashboard gebeurde voorheen handmatig. Echter, neemt dit handmatige testen veel tijd in beslag. Dit betreft voornamelijk het testen of reeds geïmplementeerde functionaliteiten beïnvloed worden door nieuwe functies, de zogeheten regressie-testen.

1.2 NTA

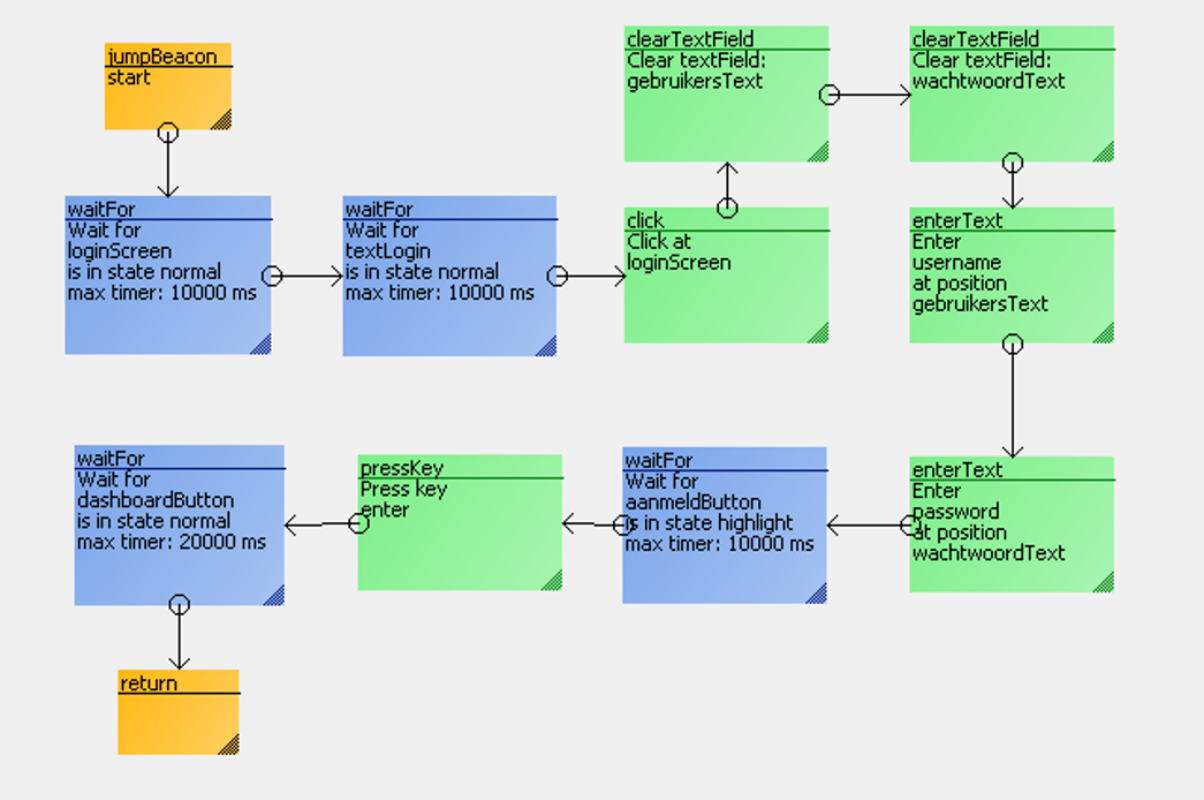
Om deze regressietesten te vergemakkelijken is er besloten om een automatisch test-systeem te ontwerpen, NTA. NTA kan grafische interfaces bekijken en beïnvloeden. Daarmee kan NTA het gedrag van een gebruiker of handmatig tester emuleren. Om dit te doen heeft NTA een intern model van het te testen systeem. Door gelijktijdig een functie uit te voeren op het model als in de werkelijkheid kan NTA discrepanties tussen het model en de werkelijkheid detecteren. Zodra een fout gevonden is kan dit betekenen dat er of: een fout in het model zit, of dat er een functionaliteit niet meer naar verwachting werkt. Mits je een goed model hebt zal het tweede het geval zijn.

Het model is non-deterministisch en bevat cykels. Het beste kan het gezien worden als een opeenvolging van verschillende use-cases, waarbij na de executie van een use-case één of meerdere andere use-cases geselecteerd kunnen worden. Dit scala aan use-cases kunnen door de cykels en het non-determinisme in het model zeer complexe testen genereren, die een handmatig tester niet zo snel zou uitvoeren.

Een voorbeeld van een simpele login is getoond in Figuur 1.2. (Voor een uitgebreidere uitleg van het model verwijs ik naar hoofdstuk 3.3)

Tevens bevat NTA de mogelijkheid om variabelen te gebruiken. Hiermee kan er bijvoorbeeld bepaald worden of een use-case uitgevoerd kan worden. Een voorbeeld is het onthouden of de huidige ingelogde gebruiker al dan niet een administrator is. Het zou immers voor te stellen zijn dat bepaalde testen enkel uitgevoerd dienen te worden wanneer er momenteel een administrator is ingelogd. Deze variabelen tezamen met de huidige positie in het model zijn te beschouwen als de huidige staat van het model in NTA.

[1] Daan van Beek, *NTA NodeTypes*


jumpBeacon
start
waitFor
Wait for
loginScreen
is in state normal
max timer: 10000 ms
waitFor
Wait for
textLogin
is in state normal
max timer: 10000 ms
click
Click at
loginScreen
clearTextField
Clear textField:
gebruikersText
clearTextField
Clear textField:
wachtwoordText
enterText
Enter
username
at position
gebruikersText
enterText
Enter
password
at position
wachtwoordText
waitFor
Wait for
aanmeldButton
is in state highlight
max timer: 10000 ms
pressKey
Press key
enter
waitFor
Wait for
dashboardButton
is in state normal
max timer: 20000 ms
return

Hoofdstuk 2

Stage-inhoud

Doordat de tool het dashboard, of "System Under Test"(SUT), via de grafische interface benaderd is er soms sprake van een pauze in de executie. Zo kan de SUT bijvoorbeeld nog een scherm moeten laden. Daaruit kunnen we twee dingen concluderen:

- NTA zal niet altijd de volledige rekenkracht van de computer gebruiken.
- Het uitvoeren van tests neemt veel tijd in beslag, dus het is zaak zoveel mogelijk diverse tests binnen een gegeven tijd uit te voeren.

Gedurende deze stage is het doel te kijken naar mogelijkheden om de executie van de hoeveelheid nuttige tests door NTA te vergroten. Wanneer het model enkel deterministisch zou zijn, dan zou deze versnelling puur in de executie van de tests gevonden moeten worden. Het versnellen van de tests is niet wenselijk. De testmethode dient namelijk de interactie en invoer van een normale gebruiker te simuleren. Wanneer de tests uitzonderlijk snel zouden worden uitgevoerd dan kan het gedrag van de SUT niet representatief zijn voor normaal gebruik.

Omdat de tests echter ook enig non-determinisme bevatten, is daar ruimte om enkele verbeteringen door te voeren. De aanwezigheid van een keuze welke test of pad gekozen dient te worden na de huidige test is aan NTA. Een random keuze garandeert uiteindelijk volledige dekking van het model (coverage), mits er een aanzienlijke hoeveelheid testen worden gedraaid.

Wellicht zou er een manier zijn om deze keuzes dusdanig te maken dat er; geen tests worden overgeslagen, maar dat een voldoende verkenning van de tests sneller gegarandeerd zou zijn. Het doel is dus, gegeven een bepaalde testtijd meer diverse testen te kunnen uitvoeren. Door deze verhoogde coverage zou de kans op het detecteren van bugs omhoog gaan. Voor het berekenen van de prioriteit van de uit te voeren tests zou de idle-tijd van NTA gebruikt kunnen worden.

Mogelijke oplossingen

Na enig overleg hebben wij besloten dat hiervoor enkele oplossingen waren.

Een optie is om de verkende paden, of tests, bij te houden. Hiermee zou een eerder genomen pad een mindere voorkeur kunnen krijgen om te verkennen of bijvoorbeeld volledig uitgesloten kunnen worden. Probleem is echter dat bepaalde paden een noodzaak zijn om te nemen, wanneer iemand de bereikbare paden na dat punt wenst te bereiken.

Een andere oplossing is om alle mogelijke toekomstige paden te verkennen. Hierbij zou bij punten met een keuze beide opties verkend worden. Gezien het legio van de beslissingen niet non-deterministisch zijn, zou op deze manier een groot deel van het model verkend kunnen worden middels bounded model checking. Op basis van deze verkenning zou vervolgens een geïnformeerde keuze kunnen worden gemaakt voor het te volgen pad. Deze keuze zou naar alle waarschijnlijkheid een aanzienlijke verbetering zijn ten opzichte van de huidige random-tactiek, bij het kiezen van een vervolgpad vanaf een non-deterministische keuze.

De laatste oplossing, met een statische analyse, bied potentieel de meeste snelheidswinst. Tevens lag deze oplossing in het verlengde van de interesses en voorkennis van zowel de stagiair als begeleider.

Hoofdstuk 3

TestAdvisor

Het doel van de stage is gesteld een zogeheten "Test Advisor"te ontwerpen voor NTA. Deze Test Advisor dient op beslissingspunten een statische analyse te doen van het onderliggende model om te bepalen wat de meest geschikte keuze is om de exploratie te vervolgen.

Nota bene, bij deze exploratie zullen wij het gedrag van het model nabootsen, om te kunnen bepalen welke nodes bereikbaar zijn. Wij zijn dus in feite een model van het model aan het exploreren. Hierbij kunnen wij mogelijk gedrag van het originele systeem als non-deterministisch beschouwen. Een voorbeeld hiervan is of een tekstveld al dan niet zichtbaar is in een GUI. Derhalve zullen we beide mogelijkheden verkennen. Wel kunnen we de staat van variabelen modelleren. Met deze variabelen kunnen we de uitkomst van zogeheten guards uitrekenen. Daardoor kunnen bepaalde executiepaden bij voorbaat uitgesloten worden aangezien deze door de guards (in combinatie met de huidige staat van de variabelen) niet bereikbaar zullen zijn. Hiermee zal de statespace drastisch verminderd worden.

3.1 Statespace

Het model bevat een groot aantal mogelijke paden. Gezien het gebruikelijk is dat het model cykels bevat, zijn er een oneindig aantal mogelijk paden. Een oneindig grote statespace is uiteraard niet volledig te verkennen. Dit is zowel qua tijd als opslagruimte een onmogelijkheid. Een manier om toch binnen afzienbare tijd tot een geschikt advies tijd te genereren voor NTA is om het model slechts deels te doorzoeken. Deze methode van enkel een deel van het model verkennen is ook bekend als bounded model-checking. Gegeven dat NTA bij een non-deterministisch punt is beland, zal de Test Advisor een bounded model-check doen vanaf dat punt. Alle mogelijke paden zullen verkend worden, de bound kan of een maximale zoekdiepte behelzen, of een maximale tijd die de Test Advisor mag besteden aan het zoeken. Na deze zoektocht zal er een verzameling van bereikbare nodes zijn. Uit deze verzameling kan vervolgens volgens een nader te bepalen metriek een doelnode gekozen worden. Gegeven dit doel kan vervolgens het te nemen pad beredeneerd worden startend op de huidige positie van NTA. De te gebruiken metriek voor het kiezen van een node kan simpel zijn; uit de bereikbare nodes willekeurig een node kiezen, maar er kan ook een complexere methode bedacht worden. Bij een complexe methode zou er bijvoorbeeld rekening gehouden kunnen worden met eerdere verkenningen van NTA (coverage) en de kans op het succesvol bereiken van deze node.

Statespace grootte

De huidige representatie van het model in NTA is een honderdtal megabytes groot. Wanneer wij bounded checking zouden willen doen zou dat inhouden dat er twee keuzes zijn. Of wij exploreren 1 pad per keer, na elke exploratie het model terug brengend in de originele staat. Dit zou weinig tot geen extra ruimte kosten, ten slotte hoeven wij enkel na elke verkenning enkel de bereikbare nodes op te slaan. Een sequentiële exploratie zou echter veel tijd kosten. De tweede optie is de verkenning van meerdere paden gelijktijdig plaats te laten vinden. Bij een dergelijke parallelle verkenning zou bij elke (non-deterministische) keuze beide paden verkend moeten worden. Om beide paden te verkennen zou dus het model gedupliceerd moeten worden bij een dergelijke keuze. De grootte van het model in acht nemende is het makkelijk om te zien dat in een dergelijk geval het geheugenlimiet van de computer aanzienlijk sneller bereikt zou worden voordat het model voldoende diep verkend is. Daarom zou in een dergelijk geval de gehele verkenning een vruchteloze operatie zijn.

Een geschikte oplossing voor deze statespace explosie is het reduceren van het formaat van het model. Dit kan door enkel de relevante informatie op te slaan in ons kopie. Deze relevante informatie zou enkel de beschrijving van de paden zijn die er in het model aanwezig zijn - en de relevante variabelen. Deze variabelen zijn nodig omdat deze mede bepalen welke paden toegankelijk zijn. Alle

andere functionaliteiten die in het model zitten hoeven wij niet in het model over te nemen. Deze overige functionaliteiten zijn bijvoorbeeld de nodige code en data in het model die nodig zijn om de SUT te besturen. Naast deze reductie kunnen we ook kiezen om de kopie op een alternatieve wijze op te slaan. Het gebruik van Binary Decision Diagrams (BDD's) is hiervoor zeer geschikt. Een BDD is een boomstructuur (graaf) waar bij elk niveau in de boom een binaire variabele representeert. Bij elk knoop in deze boom is de keuze tussen twee takken, één waarbij de gerepresenteerde variabele waar is, en één tak waarbij de variabele onwaar is. Onderaan deze boom zijn de bladeren met enkel de waarde WAAR of ONWAAR. Elk pad wat eindigt in een WAAR blad is een geldige staat voor de keuze van waardes voor de gerepresenteerde variabele in de BDD. Één enkele BDD kan dus een veelvoud aan geldige staten representeren.

3.2 BDD representatie van het model

Gezien elk decimaal getal ook in het binaire stelsel omgezet kan worden, kunnen wij in de BDD getallen opslaan. Middels het toepassen van zogeheten relaties op de BDD kunnen we dan dit getal aanpassen. Het originele model zou dan versimpeld kunnen worden als een graaf, elke vertex met zijn eigen unieke nummer. De BDD representeert deze huidige vertex in de variabele $\mathcal{V}_{\{\}}$ met $\mathcal{V}_{\{\}} \subseteq \texttt{Vertices}$. Gezien de aard van de BDD kan een enkele BDD nu een set van momenteel bereikbare vertices beschrijven. Gegeven dat wij bij vertex 3 of 1 starten met verkennen ($\mathcal{V}_{\{\}} = 1, 3$). Vervolgens passen wij een relatie $\mathcal{R}$ toe die de transitie van vertex 3 of 8 naar vertex 4 of 5 beschrijft. $\mathcal{R}_v : \mathcal{V}_{\{3,8\}} \rightarrow \mathcal{V}_{\{4,5\}}$. Na toepassing op deze relatie zal de BDD als volgt veranderen:

$$\mathcal{V}_{\{1,3\}} \xrightarrow{\mathcal{R}_v} \mathcal{V}_{\{1,4,5\}}$$

Na deze transitie zullen vertex 4 & 5 dus ook bereikbaar zijn.

Variabelen

Wij kunnen nu ook de BDD verrijken door alle variabelen uit het model in de BDD te laden. Dit kan op een vergelijkbare manier als de vertex. Gegeven een variabele v_i uit het model met de mogelijke waarden $w_{i,j} \in v_i$ die v_i kan aannemen. Nu kunnen wij een variabele $\mathcal{W}_i \in \{0..j\}$ maken waarbij $\mathcal{W}_i(j)$ de waarde van $w_{i,j}$ vertegenwoordigd. De zo verkregen BDD bevat dus $\mathcal{V}$ en $\mathcal{W}$. De BDD zal nu dus elke mogelijke variabele combinatie en vertex positie kunnen opslaan. Hiermee kunnen we met deze BDD dus één of meerdere states van het systeem weergeven.

Transities

Zoals eerder aangegeven hebben we reeds een relatie $\mathcal{R}_v$ om een transitie van de ene naar de andere vertice aan te geven. Echter deze vertex is niet de volledige beschrijving van een state. Wat er momenteel nog mist bij deze relatie is een tweetal elementen. Beide elementen hebben betrekking op de variabelen ($\mathcal{W}$) die tevens deel uitmaken van de state. Een eerste element is dat de transitie de waarde van een variabele kan veranderen.

$$\mathcal{R}_\mathcal{W} : \mathcal{W} \rightarrow \mathcal{W}'$$

Een tweede element is het feit dat een transitie alleen toegestaan kan zijn wanneer een variabele een bepaalde waarde heeft. Dit zijn zogeheten 'guards'.

$$\mathcal{R}_g : \mathcal{W}$$

Tezamen kunnen wij dus een relatie $\mathcal{R}$ definiëren die als volgt is samengesteld:

$$\mathcal{R} = \mathcal{R}_v \wedge \mathcal{R}_\mathcal{W} \wedge \mathcal{R}_g$$

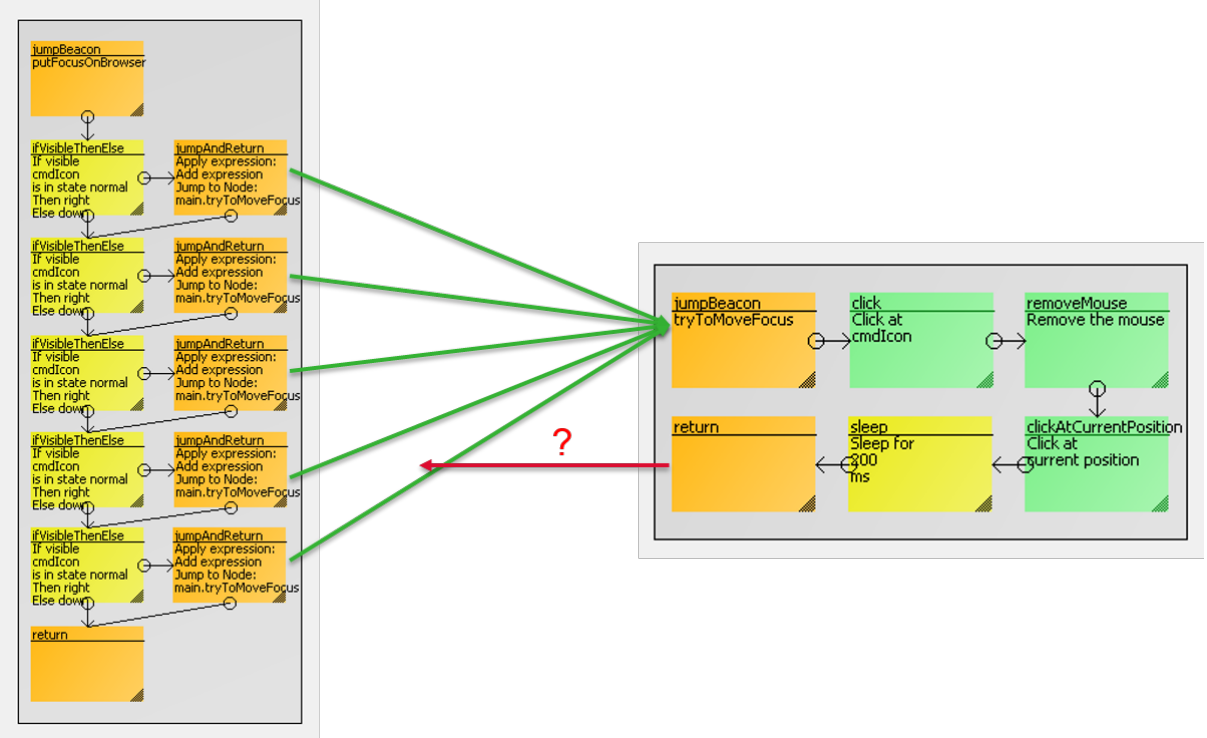
Een transitie in de BDD zal dus gecodeerd worden als:

$$\{\mathcal{V}, \mathcal{W}\} \xrightarrow{\mathcal{R}} \{\mathcal{V}', \mathcal{W}'\}$$

Functies

NTA ondersteund ook nog het gebruik van functie-aanroepen. Dit gebeurt middels de *JumpAndReturn*-node, een voorbeeld van een functie aanroep is te zien in Figuur 3.2. Functies zouden zonder problemen in de bovenstaande structuur geïmplementeerd kunnen worden. Immers, op het moment dat een functie wordt aangeroepen kan de code van de desbetreffende functie worden ingevoegd op

die positie, of ïnlining". Dit zou echter zorgen voor veel extra states. Idealiter zouden we de functionaliteit van functieaanroepen verplaatsen naar de relaties over de BDD. Gezien functies geen variabelen mee krijgen, en tevens geen waarde retourneren is dit relatief simpel. Met een simpele stack in de BDD kunnen wij bij een functieaanroep de locatie van de aanroep opslaan. Op het moment dat de functie beëindigt wordt zal de executie verder gaan vanaf het punt waar de functie is aangeroepen.



Wij definiëren in de BDD een set van variabele $\mathcal{S}_i$ met $i = 0..n$ en n de maximale diepte waarin de functies genest zullen worden. Deze n zullen we met behulp van een statische analyse kunnen achterhalen. $\mathcal{S}_i$ zal vervolgens de waarde bevatten van de vertex waar een functie aanroep vanaf is gedaan. Een relatie $\mathcal{R}$ met betrekking tot een functie-aanroep zal dus ook een element $\mathcal{R}_s$ bevatten. Deze $\mathcal{R}_s$ geeft aan hoe de stack gemanipuleerd dient te worden. Gegeven drie vertices:

- $\mathcal{V}_s$: De plek waar de functie aangeroepen wordt
- $\mathcal{V}_r$: De plek waar na de functie verder gegaan dient te worden
- $\mathcal{V}_{fs}$: Het begin van de functie
- $\mathcal{V}_{fr}$: Het einde van de functie
- $\mathcal{V}_\emptyset$: Gebruikt om een leeg element in de stack aan te geven.

Bij een functie aanroep zal deze dus in de vorm zijn van:

$$\mathcal{V} = \mathcal{V}_s \wedge \mathcal{S}_d = \mathcal{V}_\emptyset \rightarrow \mathcal{V} = \mathcal{V}_{fs} \wedge \mathcal{S}_d = \mathcal{V}_r \wedge d = d - 1$$

Hierbij is d de huidige top van de stack Bij het retourneren ziet de relatie er als volgt uit:

$$\mathcal{V} = \mathcal{V}_{fr} \wedge \mathcal{S}_d = \mathcal{V}_r \rightarrow \mathcal{V} = \mathcal{V}_r \wedge \mathcal{S}_d = \mathcal{V}_\emptyset \wedge d = d + 1$$

3.3 Statische Analyse

Wij hebben hierboven een simpele weergave gegeven hoe wij het model willen representeren in een BDD. Tevens is er beschreven welke informatie uit het model in de BDD gebruikt zal worden. Echter, wij hebben nog niet gekeken hoe de data nodig voor de BDD uit het model wordt geëxtraheerd. De benodigde informatie zal middels een statische analyse uit het model gehaald worden.

Variabelen

Voor de variabelen willen wij weten welke waardes alle variabelen kunnen aannemen. Pas als we hiervan een bovengrens hebben bepaald kunnen wij de variabele in de BDD opslaan. Indien wij een variabele vinden die maar één mogelijke waarde kan aannemen, dan is deze variabele als statisch te beschouwen. In plaats van de variabele in de BDD aan te maken kunnen we dan overal de desbetreffende variabele vervangen met zijn waarde.

Functies

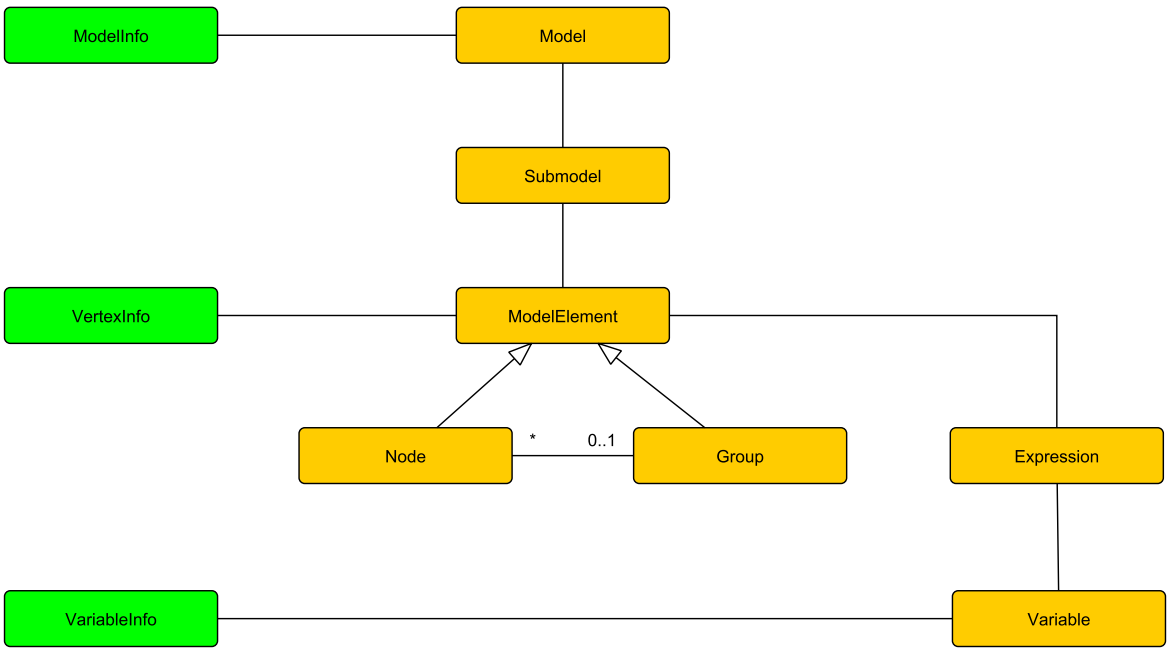
Voor functies willen we de volgorde van geneste functie-aanroepen bepalen. Zodoende kunnen we het minimale formaat van de stack bepalen voor de BDD. Ook kan op deze manier de relatie voor elke mogelijke functie-aanroep gegenereerd worden.

Model

Het originele model bestaat uit een aantal submodellen. Deze submodellen bevatten vervolgens zogenaamde ModelElements (ME's). Een ModelElement is middels lijnen verbonden aan één of meerdere andere ME's. Elke ME heeft een specifieke functie in het model [1]. Een ME kan ook een Guard bevatten, dit is een expressie die geëvalueerd wordt tot een booleaanse waarde. Alleen wanneer deze expressie WAAR is zal NTA deze ME verkennen. Als laatste kan een ME een expressie bevatten, deze expressie kan bepalend zijn voor het gedrag van het ModelElement. Het kan bepalen welke lijnen vanuit deze ME gevolgd mogen worden (deterministisch gedrag) of bijvoorbeeld de waarde aanpassen van een variabele.

Opslag gegevens

In Figuur 3.3 staat een simpele weergave van het model. Om informatie met betrekking tot het model op te slaan hebben we een aantal klassen ontworpen die informatie bevatten over het model, de ME's, en de variabelen in het model. Op deze manier kunnen we op een makkelijke manier informatie opslaan in het model, zonder het model heel erg aan te hoeven passen. ModelInfo bevat algemene informatie over het model. Ook wordt hier relevante informatie opgeslagen met betrekking tot de voortgang van de statische analyse. VertexInfo is gekoppeld aan een ModelElement, zijnde een groep die ModelElementen bevat, of een losse node. in VertexInfo wordt onder andere informatie opgeslagen met betrekking tot het type node.



[1] Daan van Beek, *NTA NodeTypes*

De laatste hulpklasse VariableInfo, bevat alle mogelijke waarde die een variabele kan aannemen. Naast deze waarde wordt ook opgeslagen welke waardes andere variabelen dienen aan te nemen om deze waarde aan te nemen. Men neme bijvoorbeeld een variabele a en een variabele b, van b is bekend dat deze de waarde 1 of 2 kan aannemen. a komt voor in drie expressies: $a = 5$, $a = b$, en $a = 3 * b$. Van a kan nu vermeld worden dat het de volgende waardes kan aannemen:

- $a = 5$
- $a = 1$ gegeven $b = 1$
- $a = 2$ gegeven $b = 2$
- $a = 3$ gegeven $b = 1$
- $a = 6$ gegeven $b = 2$

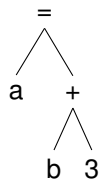
Ook wordt er in VariableInfo opgeslagen of een variabele refereert naar een andere variabele (vergelijkbaar met pointers).

Analyse

In plaats van één enkele functie om het gehele model te analyseren is er gekozen dit in stappen te doen. Een probleem bij de analyse is dat NTA ontwikkeld is naar gelang wat nodig was bij het testen. NTA heeft dus geen zeer duidelijke omschrijving van het innerlijk functioneren van de tool. Bij het uitvoeren van expressies kunnen er dus ambiguïteiten optreden. An sich is dit geen probleem, NTA functioneert tenslotte naar behoren. Maar bij het vertalen van het model naar een BDD dient het exacte gedrag van NTA geïmiteerd te worden. Mocht dit niet het geval zijn dan kan het zijn dat de NTA Test Advisor (NTA^2) bepaalde bestaande paden niet kan bereiken. Of zelfs niet bestaande paden verkend en adviseert.

Recycling van functies

Om het gedrag van NTA zo exact mogelijk na te bootsen zullen we voor zover mogelijk gebruik maken van de bestaande functies in NTA. Een eerste functie in NTA is de functie die expressies evalueert. Deze functie, `calculate()`, retourneert een string met daarin de geëvalueerde expressie. De functie wordt recursief aangeroepen op een expressie. Een expressie bestaat vervolgens uit een linker- en rechtergedeelte. Men neme bijvoorbeeld de expressie $a = b + 3$, dit evalueert tot:



Wanneer wij de calculate functie aanroepen, dan zal de waarde van b opgehaald worden. Vervolgens wordt de waarde van 3 opgehaald. Dan wordt de operator $+$ uitgevoerd op b & 3. De variabele a wordt geladen. En vervolgens wordt het resultaat van de som opgeslagen in a. Wij willen echter voor de statische analyse alle mogelijke waardes van een variabele weten. Daarom hebben wij de functie `calculate()` omgeschreven naar: `calculate(boolean calculateStatic=false)`. Wanneer de boolean `calculateStatic` ge-enabled is, dan zal calculate alle mogelijke waardes uitrekenen, gegeven de mogelijke waardes van de variabeles. Deze mogelijke waardes voor de variabele zijn opgeslagen in de klasse `VariableInfo`.

Verkennen van het model

Zoals eerder vermeld zullen wij het model in stappen verkennen. Om NTA zelf minimaal aan te hoeven passen hebben we gekozen voor een soort visitor-pattern. Elk ModelElement zal een functie `traverseModelElement(func)` krijgen die aangeroepen zal worden met een functie `visitModelElement()` als argument en vervolgens:

- `visitModelElement()` aanroept met het huidige ModelElement als argument
- `visitModelElement()` zal vervolgens de verzamelde data opslaan in het VertexInfo-object gekoppeld aan dit ModelElement
- `traverseModelElement()` aanroept voor alle ModelElementen verbonden met dit ModelElement, hetzij via een normale verbinding, hetzij via een functie-aanroep (jumpAndReturn).

Voor expressies hebben wij eenzelfde patroon gebruikt. Echter, waar de traversal-functie voor ME's een top-down benadering gebruikt, zullen expressies bottom-up geëvalueerd worden. Er zal dus eerst de traversal-functie aangeroepen worden voor de linker- en rechter-subexpressie. Vervolgens wordt de `visitExpression()` aangeroepen voor de huidige expressie. Resultaten zullen, indien van toepassing, opgeslagen worden in VariableInfo.

Travelers

Het model zal in een paar iteraties verkend worden met verscheidende *travelers*, de gebruikte travelers worden in de volgorde gebruikt waarin deze hieronder zijn beschreven. Elk van deze ModelElement-travelers zal worden aangeroepen op het initiële ModelElement van het model. Vervolgens zullen deze recursief door het model heen gaan, totdat elk element bezocht is. De Expression-travelers zullen indien nodig worden aangeroepen worden door een ME-traveler op de expressie van het betreffende ModelElement.

GraphEdges

Deze klasse zorgt ervoor dat alle verbindingen tussen de ME's opgeslagen worden in VertexInfo. Tevens slaat het op of een klasse een *JumpAndReturn* is. Indien dat het geval is, dan wordt dit tezamen met de locatie opgeslagen in VertexInfo.

GraphReturns

Nadat alle verbindingen zijn opgeslagen worden alle JumpAndReturns geanalyseerd. Zodra deze functie een JumpAndReturn tegenkomt zal hij zichzelf recursief aanroepen op de locatie waar de JumpAndReturn heen wijst. De recursie wordt beëindigd zodra de functie een *Return* tegen komt. Op deze manier kunnen we de maximale diepte van functies achterhalen. Deze diepte wordt opgeslagen - en gebruikt om het formaat van de functie-stack van de BDD te bepalen. Ook worden alle mogelijke functiecalls met hun huidige recursiediepte opgeslagen. Met deze informatie kan uiteindelijk de relatie voor de BDD worden aangemaakt die stack manipuleert. Waarmee de BDD de functiecalls emuleert zonder alle functies te hoeven inlinen.

VariableAssignments

De VariabeleAssignments-klasse, evalueert of het huidige ModelElement een variabeletoewijzing doet. Dit kan of doordat het ModelElement een zogeheten *MultipleOptionsChooser* is, of omdat de expressie van de huidige ME een *BecomesExpression* bevat. Een *MultipleOptionsChooser* (MOC) heeft als argument (expressie), een variabele en een lijst van waardes. De MOC zal uit de lijst een waarde random kiezen en toekennen aan de variabele.

Becomes De ExpressionTraveler *Becomes* kijkt of een expressie een instantie is van een becomes-Expression. Vervolgens wordt de linkerkant van de expressie geëvalueerd om te weten te komen aan welke variabele een waarde toegekend dient te worden. Laten we deze variabele "LHS-var"noemen. De rechterzijde kan of een expressie zijn, of het kan een pointer zijn naar een andere variabele.

RetrieveVars Met behulp van de expressionTraveler *RetrieveVars* wordt gekeken welke variabelen in de rechter subexpressie zitten. Vervolgens wordt in de VariableInfo van LHS-var worden opgeslagen dat LHS-var afhankelijk is van deze variabelen.

VariableBounds

Nu alle afhankelijkheden voor de variabelen zijn uitgeplozen kunnen wij proberen een upperbound te maken voor de waardes die een variabele kan aannemen. Dit wordt gedaan door de "TravExprBounds-klasse

TravExprBounds Het verkennen van de expressie met behulp van de TravExprBounds-klasse kan het best worden uitgelegd met behulp van een stuk psuedo-code. Zie hiervoor de code in Listing 3.1.

```
1  while (not VariableBounds.allBound)
     VariableBounds.allBound = true
3    forall (ME as me)
       bound = true
5      if (me.expression instanceof BecomesExpression)
       forall (me.expression.info().usedVariables as var)
7        if (not var.isBound())
           bound = false
9      if (bound)
         info.addValues(me.expression.calculate())
11     else
         VariableBounds.allBound = false
```

Listing 3.1: Pseudo code voor het verkennen van de bounds van variabelen

Alleen wanneer all variabelen waar een LHS-var van af hangt bounded zijn kan de waarde voor deze variabele uitgerekend worden. Dit proces zal dus herhaald worden tot er geen ongebonden variabelen meer zijn.

VariableReferences

Wij hebben momenteel voor elke variabele de mogelijke waardes uitgerekend. We missen echter nog een aantal waardes. Er is namelijk nog geen rekening gehouden met het gebruik van pointers. Op het moment dat a bijvoorbeeld naar b wijst, en er een andere waarde aan a wordt toegekend zal deze eigenlijk aan b worden toegekend. VariableReferences zal voor elke LHS-var recursief een functie aanroepen. Deze functie zal alle mogelijke waarden van deze LHS-var toevoegen aan alle variabelen waar LHS-var naar kan refereren. De functie wordt vervolgens weer aangeroepen op de gerefereerde variabelen. Aan het einde hebben we een bovengrens bepaald voor elke variabele, rekening houdende met pointers.

StaticVars

Nu we een bovengrens hebben bepaald kunnen we kijken of er variabelen geëlimineerd kunnen worden. Indien de bovengrens bestaat uit precies één waarde dan is de variabele statisch. StaticVars-traveler gaat alle variabelen na, en markeert deze variabelen. Bij het bouwen van de BDD kunnen deze variabelen dan genegeerd worden, en vervangen worden met de waarde van de variabele.

For loops

Na alle bovenstaande analyses is er nog een constructie in NTA waar we geen rekening mee hebben gehouden. Te weten de Groups, en specifiek de foreach-group. De foreach zal een loop-variabele voor elke waarde uit een gegeven list initialiseren - en de ModelElements in de foreach-group voor elke waarde van die variabele uitvoeren. Er is voor gekozen om deze loops uit te vouwen (loop unrolling) teneinde ze in de BDD te passen. In een eerste poging is er getracht de uitgevouwde loops in het NTA-model onder te brengen. Dit bleek echter niet mogelijk om alleen lokaal te doen gezien de aard van NTA. NTA wil namelijk elke verandering aan het model doorgeven aan de server, zodat alle andere clients eenzelfde model hebben. Hierna is besloten om zodra de bovengrens van alle variabelen is uitgerekend, er een boolean in ModelInfo wordt aangepast. Deze boolean geeft aan dat bij het verkennen van het model ook in de for-groups gekeken dient te worden. Bij de verkenning zal elke foreach dus geinitialiseerd worden met elke mogelijke waarde uit de lijst. Gezien de forloop ook weer invloed kan hebben op de waardes van variabelen worden bovenstaande ME-travelers allemaal opnieuw uitgevoerd, deze keer rekening houdende met foreach-groups. Mogelijke problemen hierbij is dat alle initialisaties van foreach-executies onderling moeten worden verbonden in de BDD. Verder zal er bij een JumpAndReturn uit een foreach-group een inlining van die code vereist zijn. Verder is het ook toegestaan om foreach-groups in for-each-groups te plaatsen. Dit alles heeft gedurende de stage een aanzienlijke hoeveelheid tijd gekost om te doorgronden en proberen te vangen in code.

BuildBDD

De BuildBDD traveler zet de informatie in de Info-klasses om in een BDD. Dit gebeurt door achtereenvolgens de variabelen en stack aan te maken. Vervolgens worden alle relaties toegevoegd. Als laatste wordt de BDD in de begin-state gezet. Gegeven deze BDD en de gegenereerde relaties kan het model verkend gaan worden.

DotBuilder

De DotBuilder is een klasse die `.dot`-bestanden kan lezen. Deze bestanden beschrijven een graaf. Het gegenereerde dot-bestand is een grafische weergave van hoe de ModelElement-travelers door het model heen gaan. Een voorbeeld van een dergelijke weergave is te zien in Figuur 4.3 & 4.3. Deze figuren zijn het best te bekijken in de PDF, daar er dan ingezoomd kan worden. Tevens is dit dus een weergave van hoe het model door de BDD geinterpreteerd zal worden. Deze klasse is puur ontworpen om in de ontwerpfase visuele feedback te hebben over hoe het model geanalyseerd wordt. Op vergelijkbare wijze kan er ook een graaf worden gecreëerd die alle variabelen en hun onderlingen verbanden toont.

Hoofdstuk 4

Conclusie

In de vorige hoofdstukken is uitgeweid over verscheidende dingen. Het probleem is aangeduid, zijnde een model gedreven testtool die naar mate het model groter wordt aanzienlijk complexer wordt. Er is gekeken naar wat een mogelijke oplossing zou zijn om dit tegen te gaan. We hebben uiteindelijk voor een van deze oplossingen gekozen om deze gedurende een twintigtal weken uit te werken. Dit idee is hierboven uitgelegd, tezamen met een uitweiding hoe deze oplossing binnen de code van NTA uitgevoerd dient te worden. Uiteraard zijn er ook nog subtiliteiten die niet in dit rapport beschreven zijn, daar deze zichzelf wijzen wanneer iemand de code zou doornemen.

4.1 Tijdsbesteding

Gedurende de eerste weken heb ik mij bekend gemaakt met mijn werkomgeving en NTA. Ik heb eerst als gebruiker met NTA kennis gemaakt om een idee te krijgen van de functionaliteiten van NTA. Hierbij was ik ook zeer geholpen door de tutorials geschreven door Daan van Beek, die zowel diende om mij bekend te maken met het systeem als documentatie, en een goed handvat om de werking van het systeem te doorgronden. Gedurende deze weken hebben we ook in overleg de exacte opdracht vorm gegeven. De contouren van de opdracht waren reeds tijdens onze eerste gesprekken voor aanvang van de stage bepaald. Om bij de exacte invulling als stagair ook meer mee te kunnen denken was een eerste kennismaking met NTA echter onmisbaar. De weken hierna heb ik literatuur gelezen met betrekking tot model driven testing, en BDD's. Verder heb ik een aantal papers over JTorx bestudeerd. JTorx is een model driven test-tool, ontwikkeld door Axel Belinfante. Met deze tool ben ik reeds bekend via een vak dat ik heb gevolgd in het kader van mijn master, en het leek mij nuttig deze tool nogmaals te bekijken in het kader van mijn huidige opdracht. Gelijktijdig ben ik begonnen met een raamwerk om met behulp van JavaBDD van een graaf een BDD te kunnen maken. Later ook met de toevoeging van variabelen, stacks, transitierelaties en een methode om een pad te bepalen gegeven een beginnode en één van de bereikbare nodes die met behulp van de transitierelaties te bereiken is. Na dit raamwerk ben ik begonnen met het ophalen van informatie uit het model. In eerste instantie met behulp van vele extra (nieuwe) functies binnen het bestaande model. Later met behulp van "Travelersëen methode die erg op het bekende visitors-pattern lijkt. Waarbij het model gedecoreerd werd met klasses waar de verkregen informatie in opgeslagen werd. Hierna is er nog een aantal weken nodig geweest om de statische analyse van variabelen correct te implementeren, en nog enige tijd om pointers te kunnen analyseren. Een veelvoud aan weken is vervolgens besteed aan het verkennen van for-loops. Eerst door deze binnen het model uit te vouwen. Later door deze op een andere manier te verkennen met behulp van de travelers. Dit zorgde echter weer voor de nodige complicaties in het creëren van de BDD. Rond dat punt was echter de tijd voor mijn stage inmiddels ten einde gelopen. Reste mij enkel nog verslaglegging en een (interne) presentatie. Bij de presentatie heb ik de inhoud van dit project en de behaalde resultaten toegelicht. Ook heb ik toen een aantal mogelijk in de toekomst te volgen ideeen besproken, waaronder de haalbaarheid en wenselijkheid voor het afronden van deze benadering.

4.2 Behaalde resultaten

Helaas is het niet gelukt gegeven de tijd die voor de stage stond de gehele opdracht af te ronden. Gelukkig hebben we wel alsnog nuttige resultaten kunnen behalen. Allereerst kunnen we met de statische analyse nuttige informatie over het model extraheren. Deze informatie kan worden gebruikt voor de verkenning van het model, zij het met behulp van BDD's of op een andere manier. Ook kunnen er door de traveller-klasses makkelijk bepaalde functionaliteiten aan NTA worden toegevoegd. Deze functionaliteiten waren voorheen al bedacht en gewenst verklaard. Het implementeren van deze functionaliteiten zonder het traveler-raamwerk kosten echter dusdanig veel tijd dat deze onderaan de featurelijst waren

beland. Een feature waar men aan kan denken is bijvoorbeeld opzoeken waar een variabele exact gebruikt wordt, en wat de impact is wanneer een gebruiker een variabele zou wijzigen. Momenteel is het verwijderen of aanpassen van een variabele een zeer precaire klus, gezien de neveneffecten niet goed zichtbaar zijn. Men zou dus of op blind vertrouwen een variabele moeten aanpassen, of een volledige kennis van het model hebben. In de praktijk betekend dit dat in een dergelijk geval vaak een nieuwe variabele geïntroduceerd wordt. Dit zal in de toekomst een hoop legacy variabelen, en dus een onnodig grote state-space kunnen opleveren. Gedurende deze stage zijn er ook al een aantal bugs en onvolledigheden in NTA gevonden en opgelost. Ook kunnen we nu een (veel) betere schatting maken van de haalbaarheid om NTA een op BDD-gebaseerde test-advisor te geven. Indien dat het geval is zou ook het raamwerk voor de BDD generatie kunnen gebruikt worden als basis hiervoor. Het is echter discutabel of een op BDD's gebaseerde advisor een wijze keuze is in retrospect.

4.3 Aanbevelingen

Indien dit gelukt zou zijn zou het een zeer efficiënte en misschien wel de beste oplossing zijn om een goed advies uit te brengen aan NTA. Het probleem is echter dat de implementatie perfect dient te zijn, dit omdat er anders geen enkele garantie kan worden gedaan over de verkenning van het model mocht de BDD een fout bevatten. De complexiteit van dit project is wellicht uit te drukken door te trachten een naam te bedenken voor wat de NTA^{2}eigenlijk is. Het is een test guidance advisor voor een model based testing tool, gebaseerd op het verkennen van een model van het model middels een BDD. Misschien wel een:

"BDD-based meta-Model Exploration based Test Guidance Tool for a Model Based Test Tool"

Ook al zou deze tool correct werken, dan nog zou NTA^{2}in sommige aspecten complexer zijn dan NTA zelf. Dit klinkt niet als een wenselijke eigenschap. Al helemaal niet wanneer in de toekomst het gedrag van NTA wijzigt, dit kan bijvoorbeeld door een uitbreiding van functionaliteiten komen, of door de introductie van een extra NodeType. Maar ook iets simpels als een bug-fix kan ervoor zorgen dat het gedrag van NTA^{2}niet meer overeenkomt met NTA. Wat zou vereisen dan iemand deze functionaliteit dus ook in NTA^{2}aanpast.

Voltooiing van NTA^2

Daargelaten dat de volledige correcte implementatie van een BDD-based Test advisor nog een aanzienlijke hoeveelheid werk zou betekenen. Mede doordat NTA nog wat ambiguïteiten bevat, verwacht ik dat NTA nog wel een aantal verassingen in petto heeft die de vlekkeloze implementatie van NTA^{2}in de weg zullen staan. Met betrekking tot de implementatie van foreach-Nodes stel ik voor deze ook in de BDD te encoderen, zoals bij JumpAndReturns ook gedaan is. Het uitrollen van for-loops lijkt een nodeloos ingewikkelde zaak te worden, ondanks dat dit op het eerste gezicht eenvoudig lijkt.

Gebruik van een bestaande modelchecker

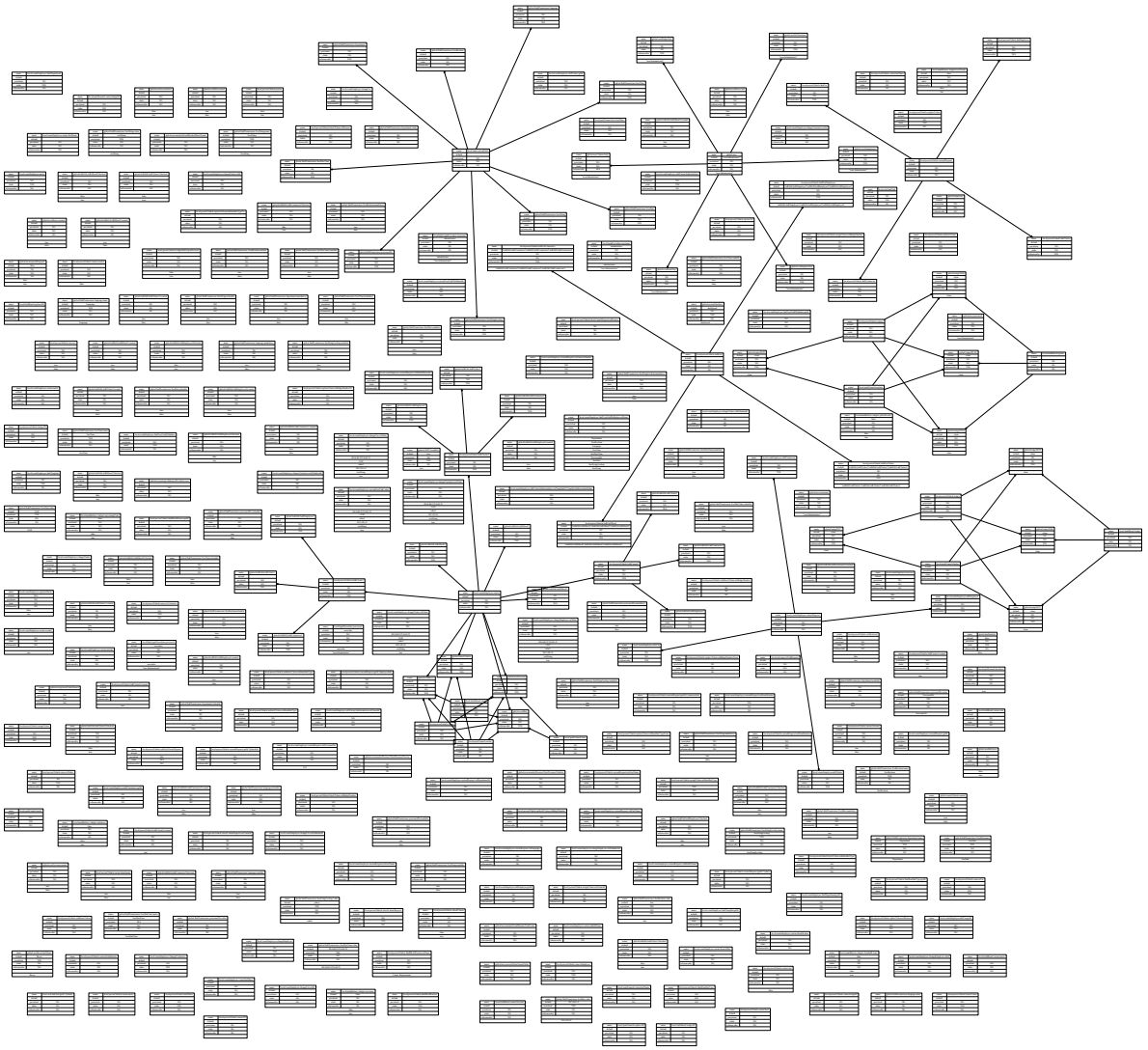
Een andere optie is om de verkregen informatie van de statische analyse in een model te stoppen dat door een traditionele modelchecker kan worden verkend. Persoonlijk lijkt mij dit geen heel goed idee. Dit omdat een volwaardige modelchecker NTA een stuk complexer maakt en dat nog steeds al het gedrag van NTA een vertaalslag dient te maken van NTA-model naar een andere modelleertaal. Daargelaten dat deze modelchecker bounded reachability checking op een manier implementeert die voor ons doel bruikbaar is.

Test Guidance op basis van eerdere verkenningen

Ondanks dat een benadering met test-guidance op basis van (bounded) verkenning van de state-space waarschijnlijk het beste advies genereert denk ik dat in de praktijk een andere keuze handiger is. Momenteel verkend NTA heel oppervlakkig door te kijken of de aangrenzende nodes al zijn bezocht, en kiest de helft van de tijd voor de minst bezochte node. Om te zorgen dat wel elke node eventueel gekozen wordt zal NTA de andere helft van de tijd een willekeurige richting kiezen. Ik denk dat wanneer in deze oppervlakkige keuze de huidige waarde van de variabelen mee wordt genomen, welke verkregen kunnen worden met de statische analyse, dat er al een iets betere keuze kan worden gemaakt. Een grotere verbetering verwacht ik wanneer de coverage van het model centraal wordt opgeslagen. Op het moment dat er getest wordt zal NTA namelijk met meerdere instanties tegelijkertijd draaien. Ik denk dat wanneer van al deze instanties centraal wordt bijgehouden waar ze zijn geweest, dat er vanuit die centrale plek een betere en snellere coverage bereikt kan worden. Deze server zou dan ook kunnen meewegen in de te kiezen richting voor NTA. In plaats van 5 instanties die allemaal op vergelijkbare

manier door het model gaat zou de server de verschillende instanties kunnen sturen om gezamelijk het model te verkennen. Met behulp van de statische analyse zou op de server van te voren al kunnen worden bepaald of een bepaalde node wel of niet bereikbaar is, dit zou onnodig verkennen kunnen minimaliseren. Ook zou deze server verdachte plekken in het model in kaart kunnen brengen. Dit zijn plekken waar NTA of nooit komt, of maar een enkele keer. Dit kan meewegen in de beslissing om deze plekken te verkennen. Wanneer op een dergelijke manier het model wordt verkent zal dit zorgen voor: een minder complex systeem voor de test-advisor dan de BDD-oplossing Een robuuste verkenner die grotendeels ongevoelig is voor de interne werking van NTA. Tenslotte hoeven we niks te weten daarvan, we sturen enkel op basis op verkregen data over het bezoeken van locaties. En hiervoor zouden reguliere algorithmen kunnen worden gebruikt voor het verkennen van grafen. Ik denk dat dit redelijk makkelijk te passen is binnen de huidige structuur van NTA. De complexiteit van de verkenningstactieken zou in eerste instantie tamelijk simpel kunnen zijn, maar zou eenvoudig uit te breiden zijn. In tegenstelling tot de BDD-oplossing, wat meer een alles-of-niets-benadering is. Tevens denk ik dat er dan ook verschillende verkenningstechnieken gebruikt kunnen worden door verschillende instanties van NTA.

Figuur 4.1: Weergave van het NTA-model zoals verkend door NTA[2]



Figuur 4.2: Weergave van de variabelen na statische analyse door NTA[2]