Afstudeerproject (MSc): FSM-based Conformance Testing

Begeleider: Joshua Moerman
joshua.moerman@ou.nl

Samenvatting: *FSM-based conformance testing* is een manier van model-based testen om *finite state machines* op fouten te controleren. Hierbij wordt verondersteld dat er een specificatie van het te-testen systeem is gemaakt, en deze specificatie wordt gebruikt om automatisch tests te genereren. Als de implementatie voldoet aan de tests, heb je een bepaalde garantie van correctheid.

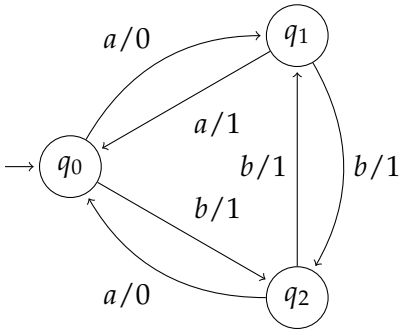
Deze techniek is ontstaan in de jaren '90, en sindsdien goed begrepen voor simpele types van finite state machines. Zo zijn er efficiënte algoritmes voor eindige automaten, (deterministische) Mealy machines, of Moore machines. Maar voor uitgebreidere modellen, die gebruik maken van symbolische inputs, of extra geheugen in de finite state machine, hebben we zulke algoritmes nog niet.

In dit project gaan we proberen de test-generatie-algoritmes uit te breiden naar zulke modellen, en zo mogelijk ook implementeren. Zie tabel 1 voor een overzicht.

Keywords: *Finite State Machines, Automated Testing, Graph Algorithms*

Tabel 1: Verschillende methoden, en voor welk type model het werkt.

Methode	FSM	Register ~	Symbolische ~
W [Cho78; Vas73]	✓	-	-
Wp [Fuj+91]	✓	-	-
HSI [LPB95]	✓	-	-
ADS [LY94]	✓	-	-
UIOv [CVI89]	✓	-	-



Figuur 1: Een finite state machine met 3 toestanden.

Achtergrond

Finite State Machines

Ik leg hier kort uit hoe een typische test-generatie methode werkt in het simpele geval van *finite state machines (FSMs)*. Een FSM bestaat uit:

- een eindige verzameling toestanden Q,
- met begintoestand $q_0 \in Q$,
- eindige verzamelingen voor input-acties X en output-labels Y,
- een transitiefunctie $\delta\colon Q \times \Sigma \to Q$,
- een output-functie $\lambda\colon Q \times X \to Y$.

In figuur 1 is een FSM afgebeeld, waarbij de rondjes de toestanden zijn en de pijlen de transities. Elke transitie heeft een input en een output (gescheiden door '/'). Wiskundig definiëren we deze FSM alsvolgt: neem $Q = \{q_0, q_1, q_2\}$, $X = \{a, b\}$ en $Y = \{0, 1\}$. De transitie van state q_0 naar q_2 wordt dan bijvoorbeeld gedefinieerd door $\delta(q_0, b) = q_2$ en $\lambda(q_0, b) = 1$.

Het onderscheiden van toestanden

Verschillende toestanden van een FSM hebben doorgaans ook verschillend gedrag. Bij het testen is het cruciaal die verschillen in gedragd ook te testen, dus we moeten bepaalde invoer vinden die de toestanden onderscheiden. De toestanden van figuur 1 kunnen allemaal onderscheiden:

- q_0 vs. q_1: De invoer a geeft in de ene toestand 0 en in de andere 1.
- q_0 vs. q_2: De invoer ba geeft in de ene toestand 10 en in de andere 11.
- q_1 vs. q_2: De invoer a geeft in de ene toestand 1 en in de andere 1.

Het bereiken van toestanden

Voor het testen moeten we ook alle toestanden kunnen bereiken. De toestanden van figuur 1 kunnen alsvolgt bereiken:

- q_0 is de begintoestand, en wordt bereikt met het lege woord ϵ.
- q_1 is bereikt met a.
- q_2 is bereikt met b.

Een *test suite*

Voor het testen van de gehele FSM gebruiken we de woorden om toestanden te bereiken, en daarna ook de woorden om toestanden te onderscheiden. We nemen eerst $P = \{\epsilon, a, b\}$ om alle states te bereieken, vervolgens nemen we $W = \{a, ba\}$ om toestanden te onderscheiden. Dit geeft de volgende verzameling:

$$T_0 = PW = \{a, aa, ba, aba, bba\}$$

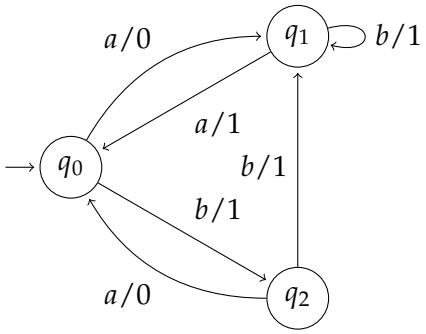
Deze verzameling is in staat het directe gedrag van een toestand te testen. Maar dit is nog geen *volledige* test suite, er zijn nog niet genoeg tests om bijvoorbeeld figuur 2 als foutief te bestempelen. Naast het testen van toestanden, moeten we ook de transities testen, dat doen we met:

$$T_1 = PXW = \{aa, ba, aba, bba, aaa, aaba, abba, baa, bba, baba, bbba\}$$

De verzameling $T = T_0 \cup T_1$ is wel volledig, op de volgende manier:

Definitie 1. Gegeven is een specificatie FSM M en een test suite $T \subseteq X^*$. De test suite T is *volledig* als voor elke implementatie I met evenveel toestanden als M geldt: als M en I dezelfde uitvoer geven voor elke test $t \in T$, dan zijn M en I equivalent (voor alle mogelijke woorden in X^*).

De bovenstaande methode heet de *W-methode* [Cho78; Vas73]. De andere methoden die genoemd zijn in tabel 1 zijn verfijningen hiervan die de verzameling T nog kleiner (en dus efficiënter) maken.



Figuur 2: Een variant van de FSM uit figuur 1. Je kunt dit zien als een *foutieve* implementatie van de specificatie.

Uitbreidingen naar andere typen systemen

Kunnen we de W-methode (en varianten) uitbreiden naar de volgende klassen van automaten?

Symbolic automata [DV14] Dit zijn automaten over een oneindig (of heel groot) alfabet X. In plaats van een transitie functie die per toestand q en symbool $x \in X$ aangeeft wat de output en volgende toestand is, wordt de transitie bepaald door een logische formule.

Denk bijvoorbeeld aan $X = \{0,1\}^8$ (de verzameling van bytes), een logische formule zou kunnen uitdrukken dat x een hoofdletter is (in de ASCII encodering), de formule is dan $\phi(x) = \texttt{0x41} \leq x \leq \texttt{0x5A}$. Door de transities "symbolisch" op te schrijven, is de automaat veel kleiner.

Register automata [KF94] Ook bij dit type automaat komt de invoer uit een groter alfabet, bijvoorbeeld $\mathbb{N}$. Bij dit type automaat, kunnen waardes uit het alfabet opgeslagen worden in de toestand. In latere transities kunnen die waardes worden vergeleken.

Een voorbeeld hiervan is in netwerk protocollen, waarbij pakketten een nummer hebben en de acknowledgement hetzelfde nummer moet hebben. De finite state machine moet dit nummer dus kunnen onthouden. Dit stukje geheugen noemen we een register.

Niet-determinisme In een niet-deterministisch systeem kan je vanuit 1 toestand met 1 invoer naar meerdere volgende toestanden gaan. De implementatie kiest dan (bijvoorbeeld willekeurig) hoe dat gebeurt. Dit maakt het testen veel ingewikkelder. Hier is al veel theorie over [PY14; Sac24], maar ik weet niet hoe toegankelijk dit is.

Overig Er zijn nog meer variaties van automaten, bijvoorbeeld *tree automata* of *weighted automata*. Het zou ook interessant zijn om de theorie van testen uit te breiden naar die modellen, maar de toepassing van die modellen is gering.

Toepassingen

Uiteindelijk worden deze test-technieken toegepast in *automata learning*. Dat is een manier om juist modellen te genereren uit een bestaand systeem (dus zonder specificatie). Het testen wordt dan gebruikt als een subroutine.

In dit project zullen we geen "real-world" toepassingen doen. Wel is het mogelijk benchmarks te doen, met bestaande modellen.

Referenties

Hieronder staan alle genoemde referenties, je hoeft ze uiteraard niet allemaal te lezen. Mocht je dit onderwerp interessant vinden, raad ik je wel aan om hoofdstuk 2 van mijn proefschrift te lezen [Moe19] of een recent paper van me [MW22].

[Cho78] Tsun S. Chow. "Testing Software Design Modeled by Finite-State Machines". In: *IEEE Trans. Software Eng.* 4.3 (1978), p. 178–187.

[CVI89] W. Y. L. Chan, S. T. Vuong en M. R. Ito. "An Improved Protocol Test Generation Procedure Based on UIOS". In: *SIGCOMM*. ACM, 1989, p. 283–294.

[DV14] Loris D'Antoni en Margus Veanes. "Minimization of symbolic automata". In: *POPL*. ACM, 2014, p. 541–554.

[Fuj+91] Susumu Fujiwara e.a. "Test selection based on finite state models". In: *Software Engineering, IEEE Transactions on* 17.6 (1991), p. 591–603.

[KF94] Michael Kaminski en Nissim Francez. "Finite-Memory Automata". In: *Theor. Comput. Sci.* 134.2 (1994), p. 329–363.

[LPB95] Gang Luo, Alexandre Petrenko en Gregor V Bochmann. "Selecting test sequences for partially-specified nondeterministic finite state machines". In: (1995), p. 95–110.

[LY94] David Lee en Mihalis Yannakakis. "Testing finite-state machines: State identification and verification". In: *Computers, IEEE Transactions on* 43.3 (1994), p. 306–320.

[Moe19] Joshua Moerman. "Nominal Techniques and Black Box Testing for Automata Learning". Proefschrift. Radboud Universiteit, 2019. URL: https://joshuamoerman.nl/thesis/.

[MW22] Joshua Moerman en Thorsten Wißmann. "State Identification and Verification with Satisfaction". In: *A Journey from Process Algebra via Timed Automata to Model Learning*. Deel 13560. Lecture Notes in Computer Science. Springer, 2022, p. 450–466.

[PY14] Alexandre Petrenko en Nina Yevtushenko. "Adaptive Testing of Nondeterministic Systems with FSM". In: *HASE*. IEEE Computer Society, 2014, p. 224–228.

[Sac24] Robert Sachtleben. "Unifying frameworks for complete test strategies". In: *Sci. Comput. Program.* 237 (2024), p. 103135.

[Vas73] MP Vasilevskii. "Failure diagnosis of automata". In: *Cybernetics and Systems Analysis* 9.4 (1973), p. 653–665.