Petri netten

Jan Van den Bussche

Geavanceerde Software Engineering

Samenvatting

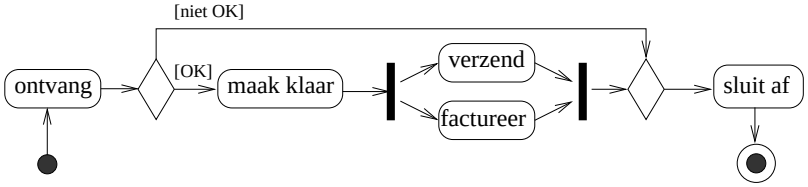
We geven een inleiding tot Petri netten: deze bieden een formele methode voor het beschrijven van concurrente systemen en liggen aan de grondslag van UML 2.0 Activity Diagrams.

1 Inleiding

Petri netten, genoemd naar hun uitvinder Carl A. Petri,[1] bieden een formele methode voor het beschrijven van *concurrente* processen. Petri was (in zijn doctoraatsthesis in 1962) inderdaad voornamelijk geïnteresseerd in wat men in het Engels *concurrency* noemt: in het Nederlands gelijktijdigheid. Een concurrent proces is dus een proces waarin verschillende activiteiten tegelijk kunnen plaatsvinden. UML Activity Diagrams laten inderdaad concurrency toe: Figuur 1 toont een lichtjes vereenvoudigde versie van het bekende Activity Diagram voor het behandelen van bestellingen. In dit voorbeeld kunnen factureren en verzenden tegelijk gebeuren.

Het is belangrijk te begrijpen dat ons begrip "tegelijk" *niet zo veel met "tijd" te maken heeft:* het gaat er niet in de eerste plaats om dat twee concurrente acties echt op een gemeenschappelijk tijdstip plaatsvinden, maar wél dat ze onafhankelijk van elkaar zijn: de ene actie heeft de andere niet nodig heeft om te kunnen starten en voltooien. In het voorbeeld kunnen factureren en verzenden onafhankelijk van elkaar gebeuren; er is geen *causaal verband* tussen factureren en verzenden (causaal: oorzakelijk). In het voorbeeld is er echter wel een causaal verband tussen klaarmaken en verzenden: het klaarmaken van de bestelling moet eerst voltooid zijn vooraleer het verzenden kan starten. Sommige acties in een concurrent proces zijn dus wel concurrent met elkaar, maar andere niet.

[1] Niet te verwarren met Julius R. Petri, de uitvinder van het Petrischaaltje.



Figuur 1: Standaard voorbeeld van een Activity Diagram.

Er is veel te zeggen voor Petri netten: ze zijn populair; ze zijn krachtig; en ze zijn formeel. Populair, omdat ze sinds hun invoering in 1962 enthousiast werden onthaald door een schare aan informatici en sindsdien duchtig zijn bestudeerd, uitgebreid, toegepast, en geïmplementeerd. Ze worden gebruikt in geavanceerde software-engineering projecten, en hun adoptie in UML 2.0 Activity Diagrams viel dan ook niet uit de lucht.

Krachtig, omdat ze nog steeds een van de krachtigste methoden zijn om concurrente systemen te beschrijven, in de zin dat je met Petri netten processen kan beschrijven die een vrij complexe en subtiele samenhang vertonen tussen de verschillende activiteiten die ervan deel uitmaken. Er zijn echter natuurlijk ook nog andere goede methoden voor het beschrijven van concurrente systemen op de markt die veel gebruikt worden door informatici, b.v., de taal LOTOS.[2]

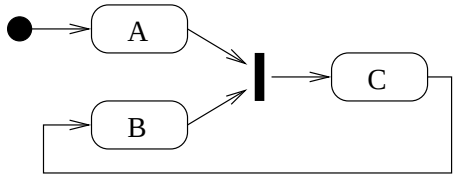
Last but not least zijn Petri netten ook formeel. Dit wil zeggen dat niet alleen hun syntax, maar ook hun semantiek volledig wiskundig exact kan gedefiniëerd worden. Waarom is formele specificatie zo belangrijk? We geven enkele verhaaltjes ter motivatie.

- Stel dat informatie-analist An het diagramma van Figuur 1 opstelt, en dat technisch analist Jan, gebaseerd op deze specificatie, een systeem implementeert dat de factuur reeds opstuurt terwijl de bestelling nog niet helemaal is klaargemaakt. An protesteert dat dit gedrag niet toegelaten is door haar specificatie, maar Jan antwoordt dat hij dat uit zo'n simpel tekeningetje niet kan opmaken. Door raadpleging van de wiskundig exacte definitie van de mogelijke gedragingen die een Petri net systeem kan vertonen, kunnen An en Jan eenduidig uit hun conflict geraken (er zal blijken dat An gelijk heeft, maar Jan had zeker ook een punt). *Alleen een systeem waarvan het gedrag formeel is gedefiniëerd kan correct geïmplementeerd worden.* Indien het systeem niet formeel is gespecificeerd, dan is het zelfs niet duidelijk wat met "correctheid" van een implementatie bedoeld wordt! Elke "bug" kan dan uitgelegd worden als een "andere interpretatie".

- Beschouw het Activity Diagram uit Figuur 2. Dit vertoont een deadlock: we beginnen met A, maar kunnen dan pas verder met C als ook B voltooid is, maar B kan op zijn beurt pas starten als C voltooid is. Om zulke fouten te vermijden willen we graag een tool die automatisch deadlocks kan opsporen. Maar vooraleer zo'n tool zelfs kan weten wat hij precies moet detecteren hebben we een wiskundig preciese definitie nodig van het gedrag van het systeem! Vooraleer je protesteert dat een deadlock toch erg simpel en direct te zien is, mag je niet vergeten dat dit in kleine voorbeeldjes wel zo is, maar Activity Diagrams in de praktijk kunnen erg groot worden, met tientallen aktiviteiten en massa's pijlen en controleknopen. Immers, visueel programmeren en specificeren in grafische notatie mag dan aantrekkelijk lijken voor kleine voorbeeldjes, wanneer de systemen groter worden moeten we oppassen voor "visual spaghetti" waar niemand nog aan uitkan. Een formele semantiek zorgt ervoor dat complexe specificaties nog steeds een eenduidige betekenis hebben en automatisch kunnen gecontroleerd en uitgevoerd worden.[3]

[2] In HCI wordt LOTOS gebruikt voor de beschrijving van taakmodellen, b.v., in de vorm van de zogenaamde ConcurTaskTree modellen.

[3] Uiteraard zijn er belangrijke hulpmiddelen om ervoor te zorgen dat grote specificaties



Figuur 2: Deadlock.

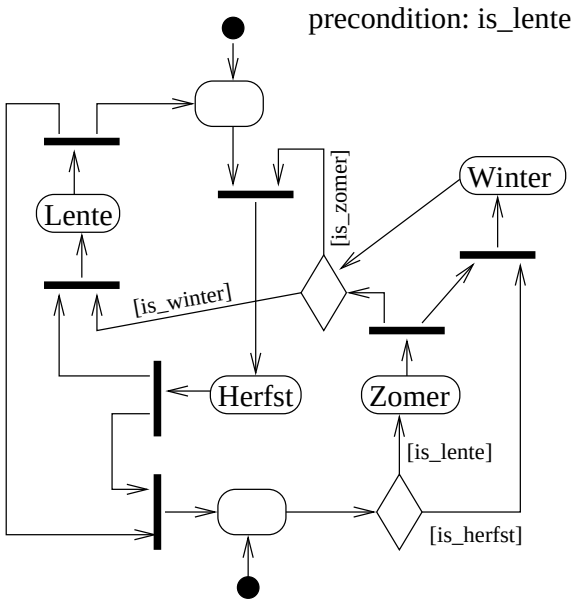
- Van visual spaghetti gesproken: beschouw het ingewikkelde Activity Diagram uit Figuur 3. Het is zeker niet op het zicht duidelijk welk systeem hier eigenlijk beschreven wordt. Als we er een beetje mee spelen merken we uiteindelijk dat dit systeem niets anders is dan de cyclische opeenvolging van de vier seizoenen, dat natuurlijk veel eenvoudiger kan beschreven worden door het Activity Diagram uit Figuur 4. In het eenvoudige diagramma merken we onmiddellijk dat er geen concurrency mogelijk is, en dus is dit evenmin mogelijk in het spaghetti-diagramma, alhoewel het wemelt van de fork nodes. We zullen zien dat met behulp van de formele semantiek van Petri netten, de twee diagramma's inderdaad *equivalent* zijn in dat ze precies hetzelfde gedrag toelaten. Het is zelfs mogelijk om equivalentie van Petri netten *automatisch* te laten verifiëren, zodat (net zoals bij de deadlocks hierboven) het mogelijk wordt dat b.v. vereenvoudiging van Petri netten door tools kan ondersteund worden. Zonder formele semantiek zouden zulke tools niet wetenschappelijk onderbouwd zijn (we zouden niet weten wat er nu juist exact moet geverifiëerd worden). Het probleem met tools zonder formele onderbouwing is dat we eigenlijk niet kunnen vertrouwen op hun resultaten.

2 Eerste kennismaking met Petri netten

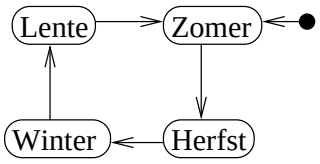
Syntactisch is een Petri net een gerichte graaf met twee soorten knopen: *places* en *transitions*. Er kunnen enkel pijlen gaan van places naar transitions, en van transitions naar places. Dus pijlen tussen places of tussen transitions zijn niet toegelaten. Grafisch stellen we places voor door cirkeltjes en transitions door rechthoekjes. Figuur 5 toont een voorbeeld van een Petri net, dat overeenkomt met het Activity Diagram uit Figuur 1.

In deze tekst beperken we ons voornamelijk tot de meest eenvoudige systemen die met Petri nets kunnen gespecificeerd worden, namelijk de zogeheten *condition/event systemen (C/E systemen)*. In een Petri net beschrijving van een C/E systeem fungeren de places als condities die al dan niet voldaan zijn, en fungeren de transitions als gebeurtenissen (Engels: *events*) die al dan niet kunnen optreden. Zo'n systeem kan zich in verschillende mogelijke *configuraties* (ook *cases* genoemd) bevinden, afhankelijk van welke condities er op het ogenblik vervuld zijn. Grafisch stellen we een configuratie voor door in de cirkels van de condities die voldaan zijn, een stip te zetten (het zogenaamde *token*).

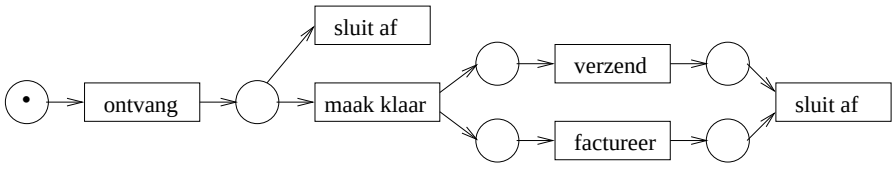
handelbaar blijven, niet in het minst de methode van gestructureerde specificatie waar een diagramma een ander diagramma kan aanroepen.



Figuur 3: Visual spaghetti.



Figuur 4: De vier seizoenen (we beginnen in de lente, d.w.z., de eerste actie is de start van de Zomer).



Figuur 5: Een Petri net voor het Activity Diagram uit Figuur 1.

We hadden dit reeds gedaan in Figuur 5, waar we de *initiële configuratie* van het systeem reeds hadden aangegeven.

We kunnen nu de betekenis uitleggen van de pijlen in een Petri net:

- Een pijl van een place c naar een event e, betekent dat c een voorwaarde is opdat e kan plaatsvinden (*preconditie*).
- Een pijl van een event e naar een place c, betekent dat het optreden van e als uitkomst heeft dat c daarna vervuld is (*postconditie*).

Wanneer dus, in een bepaalde configuratie, **alle** precondities van een bepaald event e voldaan zijn, dan kan het systeem evolueren door e te laten gebeuren. We noemen dit een *sequentiële stap*. Na deze stap komt het systeem in een nieuwe configuratie terecht, die bekomen wordt door alle tokens uit de precondities voor e weg te nemen, en tokens te plaatsen in **alle** postcondities van e. In deze nieuwe configuratie zijn dan weer nieuwe stappen mogelijk, en zo voort. Het is ook mogelijk dat in een bepaalde configuratie geen enkele stap mogelijk is.

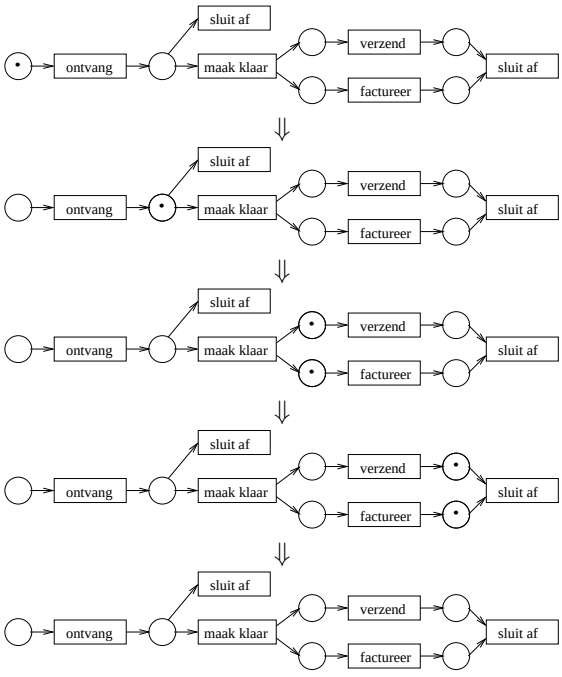
Last but not least, is het ook mogelijk dat in een bepaalde configuratie *meerdere events tegelijk mogelijk zijn (concurrency)*. Dit is zeker ook toegelaten (onder bepaalde voorwaarden die we later zullen zien); in het algemeen mag een stap dus in het algemeen ook bestaan uit een *verzameling* van events. Een sequentiële stap is dan gewoon het speciaal geval waar de stap slechts 1 element bevat.

Figuur 6 toont een opeenvolging van configuraties van ons voorbeeldnet. De eerste twee stappen alsook de vierde zijn sequentiëel; de derde is concurrent, maar dit was niet verplicht: het had evengoed toegelaten geweest om 'verzend' en 'factureer' sequentiëel uit te voeren (in eender welke volgorde). Merk op dat in de laatste configuratie geen enkele stap meer mogelijk is (er is zelfs geen enkel token meer). Een andere mogelijke run van het systeem gaat van de eerste naar de tweede en dan direct naar de vijfde configuratie; dit komt overeen met een bestelling die niet OK is (cfr. Figuur 1).

2.1 Oefeningen

Volgende oefeningen zijn overgenomen uit het boek van Reisig [1] en uit de Lectures on Petri Nets [2].

1. Stel een eenvoudig Petri net op voor de cyclische opeenvolging van de vier seizoenen, met de initiële configuratie in de lente.
2. Stel een Petri net op voor een eenvoudig producent/consument systeem dat zich gedraagt als volgt: er is 1 producent en 1 consument; en telkens de producent iets heeft geproduceerd, kan de consument het consumeren. Dit blijft zo eeuwig voortgaan. Bepaal ook zelf wat de initiële configuratie is.
3. Stel een Petri net op voor volgend systeem. Een herder wil een rivier oversteken samen met een geit, een wolf en een kool. Er is een bootje dat de herder alleen kan besturen, maar tesamen met de herder kan er slechts 1 extra object mee in de boot. Om voor de hand liggende redenen moeten de situaties vermeden worden waar de wolf en de geit, of de geit en de kool, samen alleen blijven.



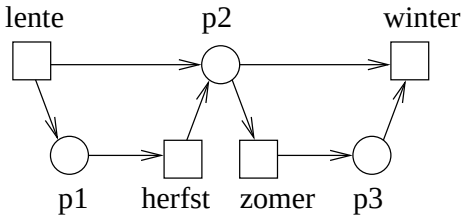
Figuur 6: Evolutie van een systeem.

4. Stel een Petri net op voor het bekende *dining philosophers* systeem. Er zijn drie filosofen die aan een ronde tafel zitten, en afwisselend eten en nadenken. Voor elke filosoof staat een bord spaghetti. Tussen elke twee filosofen ligt een lepel, maar een filosoof heeft twee lepels nodig om te kunnen eten. Elke filosoof kan volgende acties ondernemen: de lepel links, of rechts, van hem nemen (als die beschikbaar is); beginnen eten; en terug beginnen nadenken (en de beide lepels terugleggen). Initiëel zijn ze alle drie aan het nadenken.

5. Stel een Petri net op voor het bekende *mutual exclusion* systeem. Er zijn twee spelers die een shared resource bezitten. De shared resource kan echter maar door 1 speler tegelijk gebruikt worden. Elke speler kan volgende acties ondernemen: de shared resource opnemen (als ze vrij is); of de shared resource terug vrijgeven.

3 Formele definitie van Petri netten en hun gedrag

Om onze definities formeel te maken gebruiken we de taal van de verzamelingenleer. Dit is geen toeval: de verzamelingenleer is de taal bij uitstek gebruikt, al sinds het begin van de 20ste eeuw, door wiskundigen en informatici om hun ideeën precies te formuleren. Dat b.v. de formele specificatiemethode Z gebaseerd is op verzamelingenleer is dus helemaal niet zo bijzonder. Wat wel bijzonder is aan Z is dat ze een gestandardiseerde notatie invoert voor alle begrip-



Figuur 7: Voorbeeld van een Petri net overgenomen uit het boek van Reisig [1].

pen en bewerkingen op verzamelingen, en ook een formele taal (de zogenaamde schema's) invoert om de specificatie te structureren, om begrippen een naam te geven, enz. Het voordeel van zo'n formele taal is dat specificaties dan ook door computers kunnen verwerkt en gecheckt worden. Voor het gemak zullen wij echter niet de taal Z gebruiken, maar gewoon rigoureus Nederlands, doorspekt met de conventionele, basiswiskundige notatie voor het werken met verzamelingen. Het is op deze wijze ook dat de meeste wiskundigen en informatici in de praktijk werken als ze iets formeel willen definiëren.

Een Petri net bestaat uit 3 componenten: de verzameling places P; de verzameling transities T; en de verzameling pijlen R. Een pijl van x naar y kunnen we mooi formaliseren als een koppeltje (x, y).

1 Definitie. Een Petri net is een 3-tupel $N = (P, T, R)$ waarbij:

1. P een eindige verzameling is; de elementen noemen we *places*;
2. T een eindige verzameling is, disjunct van P; de elementen noemen we *transities*;
3. $R \subseteq (P \times T) \cup (T \times P)$.

2 Voorbeeld. Als we een Petri net tekenen en we willen de identifiers van de places en transities aangeven, dan schrijven we die er naast. Dat hebben we gedaan voor het Petri net in Figuur 7. Dit is overigens de Petri net versie van het Activity Diagram van Figuur 3! Als vergelijking tussen de elegantie van beide formalismen kan dit tellen. In ieder geval, formeel zien we nu dat dit net gelijk is aan (P, T, R), waarbij

1. $P = \{p_1, p_2, p_3\}$;
2. $T = \{\text{lente}, \text{zomer}, \text{herfst}, \text{winter}\}$; en
3. $$R = \{(p_1, \text{herfst}), (p_2, \text{zomer}), (p_2, \text{winter}), (p_3, \text{winter}), \\ (\text{lente}, p_1), (\text{lente}, p_2), (\text{herfst}, p_2), (\text{zomer}, p_3)\}$$

□

Een handige notatie is die van *pre-set* $\bullet x$ en van *post-set* $x\bullet$ voor een place of transitie x. We definiëren deze als volgt:

$$\bullet x = \{y \mid (y, x) \in R\}$$
$$x\bullet = \{y \mid (x, y) \in R\}$$

Dus, in de terminologie van C/E systemen, als x een event (transitie) is, dan bestaat $\bullet x$ uit alle precondities voor x, en $x\bullet$ uit alle postcondities. En als x een place is, dan bestaat $\bullet x$ uit alle events waarvoor x een postconditie is, en bestaat $x\bullet$ uit alle events waarvoor x een preconditie is.

3 Oefening. In Voorbeeld 2, wat is $\bullet p_2$? $\bullet$herfst? $\bullet$lente? lente$\bullet$? $p_3\bullet$? winter$\bullet$? □

Een fundamenteel begrip voor het C/E systeem beschreven door een Petri net is dat van *configuratie.* Een configuratie geeft aan welke condities op dit ogenblik vervuld zijn, en is dus niets anders dan een verzameling places:

4 Definitie. Zij $N = (P, T, R)$ een Petri net. Een *configuratie* voor N is een deelverzameling van P.

5 Voorbeeld. Een voorbeeld van een configuratie voor het net van Voorbeeld 2 is $\{p_1, p_2\}$. □

We zijn nu klaar om het cruciaal begrip van *stap* van een net te definiëren. Een stap is altijd relatief t.o.v. een gegeven configuratie. In die configuratie kunnen mogelijk een aantal events (transities) gebeuren, omdat hun precondities allen vervuld zijn. Er is echter nog een belangrijke **bijkomende** voorwaarde opdat een event e is toegelaten in een gegeven configuratie, waar we in het vorige hoofdstuk voor de eenvoud nog over gezwegen hebben. Deze is dat de postcondities van e nog *niet* mogen vervuld zijn in de huidige configuratie. Deze voorwaarde is misschien niet direct intuïtief. Indien alle precondities vervuld zijn, maar er is ook minstens 1 postconditie al vervuld, spreken van een *contactsituatie.* Een event in een contactsituatie is dus niet toegelaten om te gebeuren. Dit lijkt overdreven streng, maar in feite is een Petri net systeem waarin contactsituaties mogelijk zijn, niet voldoende gespecificeerd. Je kan elk Petri net altijd uitbreiden met extra condities zodat contact nooit kan voorkomen.

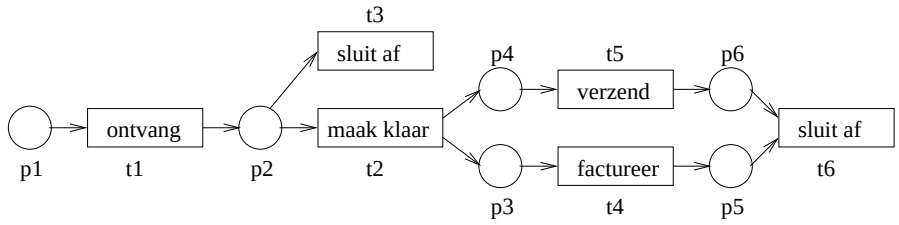
Meerdere toegelaten events kunnen nu tegelijk gebeuren (concurrency), op voorwaarde dat er geen *conflicten* zijn tussen twee events, d.w.z. dat hun presets disjunct zijn, alsook hun post-sets. Een conflictvrije verzameling van zulke events noemen we dan een stap.

Formeel kunnen we al het voorgaande samenvatten als volgt:

6 Definitie. Zij $N = (P, T, R)$ een Petri net, en zij C een configuratie voor N.

1. Voor een event $e \in T$ zeggen we dat e *toegelaten is in* C als $\bullet e \subseteq C$, en $e\bullet \cap C = \varnothing$. (In het Engels zegt men *enabled.*)
2. Voor events $e_1, e_2 \in T$ zeggen we dat e_1 en e_2 *in conflict zijn*, als $\bullet e_1 \cap \bullet e_2 \neq \varnothing$ of $e_1\bullet \cap e_2\bullet \neq \varnothing$.
3. Een *stap* vanuit C is nu een deelverzameling G van T zodat elke $e \in G$ toegelaten is in C, en zodat er geen met elkaar conflicterende events in G zitten.

We moeten tenslotte nog het resultaat van een stap definiëren. Dit komt heel intuïtief overeen met het spelletje met de tokens: we nemen het token weg uit alle precondities van alle events van de stap, en plaatsen tokens in alle postcondities. Merk op dat, doordat er in een stap geen contact en geen conflict kan voorkomen, dit nooit kan leiden tot het plaatsen van een token in een place waar al een token aanwezig was. Formeel:



Figuur 8: Analyse van het net uit Figuur 5.

7 Definitie. Zij G een stap vanuit configuratie C. De *opvolgconfiguratie van C onder G* is gelijk aan

$$(C \setminus \bigcup_{e \in G} {\bullet e}) \cup \bigcup_{e \in G} e{\bullet}$$

Als C een configuratie is, G een stap voor C, en C' is de opvolgconfiguratie, dan noteren we dit als

$$C \; [G\rangle \; C'$$

8 Voorbeeld. Voor het net van Figuur 2, vanuit de configuratie $C_1 = \{p_1, p_2\}$, is er slechts 1 stap mogelijk, namelijk {zomer} (zorg dat je begrijpt waarom!) De opvolgconfiguratie is dan $C_2 = \{p_1, p_3\}$. Vanuit C_2 is opnieuw slechts 1 stap mogelijk, namelijk {herfst}, met opvolgconfiguratie $C_3 = \{p_2, p_3\}$. Vanuit C_3 is opnieuw slechts 1 stap mogelijk, namelijk {winter}, met opvolgconfiguratie $C_4 = \varnothing$. Tenslotte, vanuit C_4 is opnieuw slechts 1 stap mogelijk, namelijk {lente}; merk op dat die transitie geen precondities heeft, maar dat ze voorheen niet toegelaten was wegens contact in p_1 of p_2. De opvolgconfiguratie is terug C_1 en we zijn rond.

9 Oefening. Beschouw terug het net in Figuur 5 uit het vorig hoofdstuk, en we geven de verschillende places en transities een identifier zoals getoond in Figuur 8. (Merk op dat transities t_3 en t_6 verschillende knopen zijn, alhoewel ze met dezelfde actie 'sluit af' gelabeld zijn.)

1. Welke stappen zijn mogelijk vanuit de configuratie $\{p_2\}$? Wat is telkens de opvolgconfiguratie?
2. Zelfde vraag voor de configuratie $\{p_3, p_4\}$.
3. Zelfde vraag voor $\{p_3, p_6\}$.

4 Petri netten met acties

Tot nu toe hebben we de transities in een Petri net beschouwd als abstracte knopen. Maar in de praktijk staan deze transities natuurlijk model voor bepaalde acties die het systeem uitvoert. We hebben dit reeds gezien in Figuur 5 waar we de transities hebben gelabeld met acties. In Activity Diagrams doen we dat immers ook.

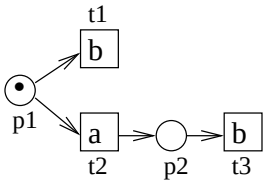
Het beschikbaar repertoire aan acties formaliseren we als een eindige verzameling $\mathcal{A}$ waarvan we de elementen, wat dacht je, *acties* noemen. Het labelen

van de transities van een net met acties wordt dan mooi geformaliseerd door een functie van T naar $\mathcal{A}$, die we meestal als λ noteren. We besluiten:

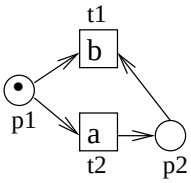
10 Definitie. Een *gelabeld Petri net* is een koppel (N, λ), waarbij $N = (P, T, R)$ een Petri net is, en λ een functie van T naar $\mathcal{A}$.

Merk op dat we toelaten dat twee verschillende transities gelabeld worden met eenzelfde actie (dit was trouwens ook reeds het geval in Figuur 5). Met andere woorden, de functie λ is niet noodzakelijk injectief. Dit komt handig van pas bij het modelleren van alternatieven.

11 Voorbeeld. Onderstel acties a en b, en beschouw de activiteit '$ab + b$', waarmee we bedoelen dat we ofwel a doen gevolgd door b, ofwel doen we alleen b. We kunnen deze activiteit modelleren door volgend gelabeld net:



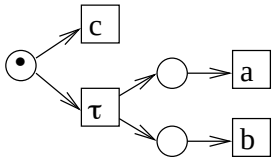
De functie λ hier is dus $\lambda(t_1) = b$; $\lambda(t_2) = a$; en $\lambda(t_3) = b$. Het gebruik van verschillende transities t_1 en t_3 voor actie b is hier echt wel nodig. Als we domweg t_1 en t_3 zouden samensmelten, als volgt:



dan zou dit een foute modellering zijn, omdat hier na uitvoering van a, de actie b niet meer mogelijk is; het token is dan weg uit p_1, en t_1 heeft tokens nodig in zowel p_1 als p_2.

De "stille" actie Een handig, soms onontbeerlijk middel om systemen juist te modelleren, is het gebruik van een speciale, *stille* actie (Engels: silent action), die meestal met τ wordt aangegeven.

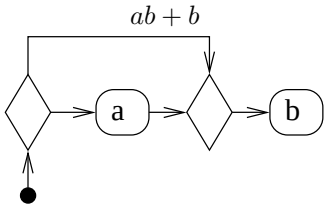
12 Voorbeeld. Een eenvoudig voorbeeld is de activiteit '$(a \mid b) + c$', waarmee we bedoelen dat we ofwel a en b concurrent doen, ofwel doen we alleen c. We kunnen dit modelleren met behulp van een stille actie als volgt:

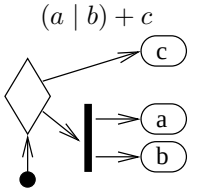


□

Zoals dit voorbeeld illustreert hebben stille transities een hulpfunctie in de synchronisatie van de acties, maar stellen ze zelf geen relevante actie voor.

Activity Diagrams De kritische student zou kunnen opmerken dat Activity Diagrams "eenvoudiger" of "krachtiger" zijn dan Petri netten, omdat het in Activity Diagrams blijkbaar nooit nodig is om verschillende action nodes te gebruiken met dezelfde actienaam. Evenmin lijkt het ooit nodig om stille acties te gebruiken in Activity Diagrams. Inderdaad, de twee vorige voorbeelden kunnen gemodelleerd worden als volgt:





Deze kritiek is echter niet helemaal terecht. Ten eerste heb je in Activity Diagrams meerdere soorten control nodes (decision, merge, fork, join), terwijl je in Petri netten alleen maar places hebt, dus je zou evengoed kunnen zeggen dat Activity Diagrams hierdoor juist ingewikkelder zijn dan Petri netten. Ten tweede is het maar de vraag of je echt elk gelabeld Petri net dat gebruik maakt van stille acties ook equivalent kan modelleren als een Activity Diagram. Voor zover we weten is dit niet bewezen (het is een interessante vraag). De omgekeerde richting kan echter wél worden bewezen, dus ook de claim dat Activity Diagrams krachtiger zouden zijn dan Petri netten is dubieus. Tenslotte hebben Activity Diagrams geen standaard formele semantiek, dus is het zelfs niet onmiddellijk duidelijk hoe je Activity Diagrams rechtstreeks zou moeten vergelijken met Petri netten! Deze discussie voert ons naadloos tot het volgende hoofdstukje.

5 Van Activity Diagrams naar Petri netten

De UML standaard geeft wel een informele beschrijving van het gedrag van Activity Diagrams, maar deze is zeker niet precies genoeg om van een formele semantiek te kunnen spreken. Door een vertaling te geven van Activity Diagrams naar Petri netten, die wel een formele semantiek hebben, kunnen we dus een formele semantiek geven aan Activity Diagrams. Het basisformalisme van gelabelde Petri netten met stille acties die we hier gezien hebben volstaat reeds om alle "control flow" aspecten van Activity Diagrams te formaliseren. Binnen het bestek van deze cursus hebben we geen ruimte om de vertaling in detail te beschrijven, maar ter illustratie kan je Figuren 1 en 5 met elkaar vergelijken, alsook de voorbeelden in het vorige hoofdstukje.

Voor de "data flow" aspecten (object nodes, data tokens) moeten we overgaan naar *high-level* Petri netten (ook *coloured* Petri netten genoemd). Eigenlijk is het jammer dat men in UML niet onmiddellijk het reeds bestaande en goed ontwikkelde formalisme van coloured Petri nets heeft geadopteerd, in plaats van er een eigen ad-hoc versie voor te verzinnen, met alle complicaties vandien. Een alternatief had geweest dat de UML standaard rechtstreeks een formele semantiek had gegeven van Activity Diagrams (dus op dezelfde wijze als wij dit hier doen voor Petri netten) maar dat heeft men jammer genoeg nagelaten te doen. In de academische wereld zijn gelukkig onderzoekers bezig deze leemte in te vullen.

6 Model Checking

Model Checking is een echt succesverhaal van de formele aanpak in software engineering. Dankzij model checking kan een model van een softwaresysteem automatisch wordt gecheckt op allerhande eigenschappen, door er graafalgoritmen op los te laten. Model Checking is een algemene methode, niet enkel toepasbaar op Petri netten; wij bekijken hier de toepassing op onze C/E systemen.

Eigenlijk hebben we nog steeds geen formele definitie gegeven van een C/E systeem, maar dat is heel eenvoudig.

13 Definitie. Een C/E systeem is een koppel (N, C_0), waarbij N een Petri net is, en C_0 een configuratie van N, die de *initiële configuratie* wordt genoemd.

Elke configuratie die bereikbaar is vanuit de initiële configuratie door een willeurig aantal stappen te maken, is een mogelijke configuratie van het systeem. Aangezien een configuratie niets anders is dan een deelverzameling van P (de verzameling places), en aangezien P een eindige verzameling is, zijn er slechts een eindig aantal mogelijke bereikbare configuraties.[4] We kunnen de verzameling *Conf* van alle bereikbare configuraties genereren met een breadth-first search vanuit de initiële configuratie. In het onderstaande algoritme gebruiken we de notatie opvolg(C, e), voor een configuratie C en een transitie $e \in T$ die is toegelaten in C, om de opvolgconfiguratie aan te duiden van C onder de sequentiële stap $\{e\}$.

Conf := $\{C_0\}$;
Nieuwe_Confs := $\{C_0\}$;
repeat
  Nieuwste_Confs := $\varnothing$;
  for each $C \in$ *Nieuwe_Confs* **do**
    for each $e \in T$ zodat e is toegelaten in C **do**
      if opvolg(C, e) nog niet in *Conf* **then**
        voeg opvolg(C, e) toe aan *Nieuwste_Confs*
  voeg alle configuraties in *Nieuwste_Confs* toe aan *Conf*
  Nieuwe_Confs := *Nieuwste_Confs*
until *Nieuwe_Confs* = $\varnothing$

6.1 De sequentialisatie-eigenschap

De wakkere student zal nu volgend potentiëel probleem opmerken: het bovenstaand algoritme exploreert de ruimte van bereikbare configuraties enkel aan de hand van *sequentiële* stappen: stappen bestaande uit 1 enkel event. We weten echter dat in het algemeen stappen *concurrent* mogen zijn, bestaande uit meerdere events. Misschien bestaan er wel configuraties die enkel bereikbaar zijn via concurrente stappen, en die je dus niet kan bereiken louter met sequentiële stappen? Als dat zo zou zijn dan zouden we het zoekalgoritme moeten aanpassen zodat het voor elke configuratie niet alleen alle mogelijke toegelaten events

[4] Meer precies, aangezien het aantal deelverzamelingen van een verzameling met n elementen gelijk is aan 2^n, zijn er dus hoogstens 2^n bereikbare configuraties, met n het aantal places.

e probeert, maar meer algemeen alle mogelijke *verzamelingen* G van events. Als er m events zijn, zijn er 2^m verzamelingen van events, en dus zou het algoritme exponentiëel veel meer werk moeten verrichten.

Gelukkig is onze wakkere student te pessimistisch: *als een configuratie bereikbaar is via concurrente stappen, is ze ook bereikbaar enkel via sequentiële stappen.* Dit volgt uit de meer algemene *sequentialisatie-eigenschap*, die ook heel interessant is op zichzelf.

Om de eigenschap handig te kunnen formuleren, herinneren we ons de notatie voor een concurrente stap $C\ [G\rangle\ C'$ uit Definitie 7. We voeren nu ook nog volgende notatie in voor een sequentie van sequentiële stappen. Stel dat we configuraties $C_1, \ldots, C_{n+1}$ hebben en events $e_1, \ldots, e_n$ zodat het volgende geldt:

$$C_1\ [\{e_1\}\rangle\ C_2\ [\{e_2\}\rangle\ C_3\ \ldots\ C_n\ [\{e_n\}\rangle\ C_{n+1}$$

Dan noteren we dit als

$$C_1\ [e_1, \ldots, e_n\rangle\ C_{n+1}$$

We hebben nu de:

Sequentialisatie-eigenschap. *Zij $N = (P, T, R)$ een Petri net, zij C en C' configuraties van N, en zij $G \subseteq T$ een verzameling events. Dan geldt $C\ [G\rangle\ C'$ als en slechts als voor elke mogelijke volgorde $G = \{e_1, \ldots, e_n\}$ van opsomming van de elementjes van G, geldt dat $C\ [e_1, \ldots, e_n\rangle\ C'$.*

Deze eigenschap bevestigt formeel een basisintuïtie van concurrency, namelijk *concurrency is hetzelfde als onafhankelijkheid van volgorde van uitvoering.* De eigenschap is niet moeilijk om te bewijzen, maar het bewijs is nogal langdradig en we laten het achterwege.

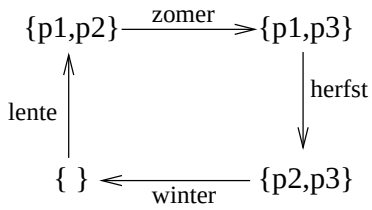
14 Oefening. Vergewis je ervan dat je begrijpt wat we eerder verklaarden, namelijk, dat de sequentialisatie-eigenschap als gevolg heeft dat een configuratie die bereikbaar is via concurrente stappen, ook bereikbaar is enkel via sequentiële stappen.

15 Oefening. Genereer, op papier, alle bereikbare configuraties voor het systeem uit Figuur 5 (gebruik de identifiers getoond in Figuur 8). Doe hetzelfde voor enkele van de systemen die je moest opstellen in Oefeningen 2.1.

6.2 De configuratiegraaf

De structuur bestaande uit alle mogelijke bereikbare configuraties van een C/E systeem, met aanduiding van de transities ertussen, kunnen we nu expliciet voorstellen in een eindige datastructuur, de *configuratiegraaf* genoemd (ook dikwijls *case graph* genoemd). Dit is een gerichte graaf, met als knopen alle bereikbare configuraties, en met pijlen als volgt: als $C\ [e\rangle\ C'$ voor een event e, dan trekken we een pijl $C \xrightarrow{e} C'$ gelabeld met e. Het mag duidelijk zijn dat het algoritme dat we hierboven hebben gezien voor het genereren van de knopen van de configuratiegraaf, meteen ook kan gebruikt wordt voor het genereren van de pijlen.

16 Voorbeeld. Beschouw het C/E systeem (N, C_0) waarbij N het net is uit Figuur 7, en waarbij $C_0 = \{p_1, p_2\}$. Volgens de analyse die we reeds maakten in Voorbeeld 8, is de configuratiegraaf van dit systeem zoals getoond in Figuur 9.



Figuur 9: Configuratiegraaf van het systeem van Figuur 7 met initiële configuratie $\{p_1, p_2\}$.

17 Oefening. Vervolg Oefening 15 door telkens de volledige configuratiegraaf te tekenen.

6.3 Model checking

Gegeven een C/E systeem, kunnen we ons heel wat vragen stellen over het gedrag. Hier zijn enkele voorbeelden van typische vragen:

1. Kan een bepaald event e ooit uitgevoerd worden? M.a.w., bestaat er een bereikbare configuratie waarin e is toegelaten? Een variant van deze vraag kan gesteld worden wanneer de events gelabeld zijn met acties, zoals gezien in Hoofdstuk 4. We kunnen dan vragen of een bepaalde actie a ooit uitgevoerd kan worden, m.a.w., is er een bereikbare configuratie waarin een event e met $\lambda(e) = a$ is toegelaten?

2. Is een bepaald event e *live?* Dit betekent dat vanuit elke bereikbare configuratie, een configuratie kan bereikt worden waarin e kan uitgevoerd worden. Opnieuw kan een analoge vraag gesteld worden voor een actie a.

3. Is het systeem *cyclisch?* Dit betekent dat vanuit elke bereikbare configuratie, we altijd terug de intiële configuratie kunnen bereiken.

4. Kan het systeem in *deadlock* komen? M.a.w., bestaat er een bereikbare configuratie die we niet beschouwen als "eindconfiguratie" maar waarin toch geen enkel event toegelaten is?

5. Kunnen acties a en b *concurrent* gebeuren? M.a.w., bestaat er een bereikbare configuratie waarin een stap $\{e_1, e_2\}$ mogelijk is met $\lambda(e_1) = a$ en $\lambda(e_2) = b$?

De lijst is eindeloos; het fantastische is dat zulke vragen duidelijk allemaal *automatisch* kunnen beantwoord worden door de configuratiegraaf te raadplegen met de gepaste graafalgoritmen. Je kan deze zelf implementeren, maar er zijn ook *generische model checking tools* beschikbaar (een bekende is SPIN) waarmee abstracte configuratiegrafen kunnen geverifiëerd worden.

Concurrency Generische model checking tools werken enkel op abstracte configuratiegrafen, en hebben dus geen kennis van het Petri net dat ten oorsprong ligt van deze graaf. Een mogelijk probleem dan is dat vragen over concurrency

niet altijd meer direct kunnen beantwoord worden. Bijvoorbeeld, om voorbeeldvraag 5 hierboven te beantwoorden zoeken we een bereikbare configuratie C waarin een stap $\{e_1, e_2\}$ mogelijk is met $\lambda(e_1) = a$ en $\lambda(e_2) = b$. Voor de generische model checker is C echter gewoon een abstracte knoop, en bij gebrek aan het oorspronkelijk Petri net kan hij dit dus niet zien aan C of de voorwaarde voldaan is. De sequentialisatie-eigenschap brengt hier echter redding! Dankzij deze eigenschap kunnen we de vraag herformuleren als volgt: bestaan er bereikbare knopen C en C', events e_1 en e_2 met $\lambda(e_1) = a$ en $\lambda(e_2) = b$, en tussenknopen C_1 en C_2, zodat de pijlen

$$C \xrightarrow{e_1} C_1 \xrightarrow{e_2} C' \qquad \text{en} \qquad C \xrightarrow{e_2} C_2 \xrightarrow{e_1} C'$$

aanwezig zijn in de configuratiegraaf? Dit kan dan gecheckt worden louter op basis van de configuratiegraaf alleen.

7 Equivalentie

Wanneer zijn twee softwaremodellen equivalent? Dit is een belangrijke vraag wanneer we modellen willen vergelijken, verfijnen of vereenvoudigen. Er is echter geen eenduidige notie van equivalentie: er zijn verschillende noties, die allemaal hun waarde hebben, en de software engineer moet de keuze maken met welke notie hij wil werken. We beschrijven hier de drie belangrijkste equivalentiebegrippen, maar er zijn er nog vele andere.

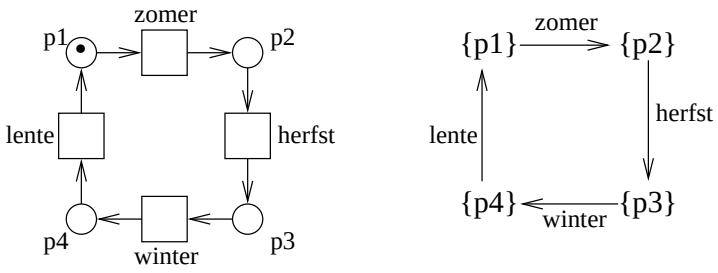
We bespreken de noties hier in termen van abstracte *labeled transition systems (LTS)*. Net zoals de configuratiegrafen van Petri netten zijn dit gerichte grafen. De knopen van een LTS worden in het algemeen *toestanden* genoemd, en er is een initiële toestand aangeduid. De pijlen $x \xrightarrow{a} y$ van het LTS zijn gelabeled met acties.

LTS'en worden algemeen gebruikt in model checking omdat bijna alle formele methoden in software engineering een semantiek hebben in termen van LTS'en. Uiteraard geldt dit ook voor onze Petri netten. Als we een ongelabeld Petri net hebben, kunnen de events rechtstreeks de rol van acties spelen: de pijlen van de configuratiegraaf zijn immers gelabeled met events en de configuratiegraaf is dan gewoon een LTS. Als we een gelabeld Petri net hebben als in Hoofdstuk 4, dan zijn de pijlen van de configuratiegraaf gelabeld met events, die dan op hun beurt gelabeled zijn met acties (door de functie λ): we maken van de configuratiegraaf dan eenvoudig een LTS door elke pijl $x \xrightarrow{e} y$ rechtstreeks te herlabelen met zijn actie, als $x \xrightarrow{\lambda(e)} y$.

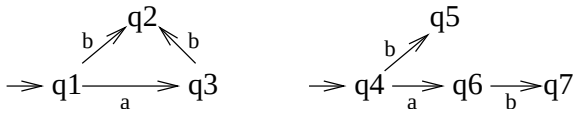
Isomorfisme De meest strenge notie van equivalentie die we kunnen eisen is dat de twee systemen *isomorf* zijn. Dit betekent dat de twee grafen er helemaal gelijk uitzien, op de keuze van de preciese identifiers van de toestanden na, omdat die keuze toch geen belang heeft.

Formeel noemen we twee LTS'en A en B *isomorf* als er een bijectie β bestaat van de verzameling toestanden van A naar de verzameling toestanden van B, die de initiële toestand van A afbeeldt op de initiële toestand van B, en zodat als er een pijl $x \xrightarrow{a} y$ is in A, dan is er een pijl $\beta(x) \xrightarrow{a} \beta(y)$, en omgekeerd.

Twee C/E systemen worden dan *isomorfie-equivalent* genoemd als hun LTS'en isomorf zijn.



Figuur 10: Een C/E systeem (links) en zijn configuratiegraaf (rechts).



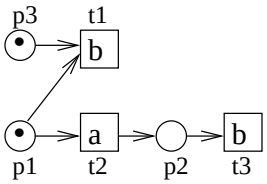
Figuur 11: Twee labeled transition systems die niet isomorf zijn, maar wel trace-equivalent. De initiële toestanden zijn aangeduid door een inkomend pijltje.

18 Voorbeeld. Van het ondertussen uitgemolken C/E systeem uit Figuur 7, laten we dit A noemen, met initiële configuratie $\{p_1, p_2\}$, hebben we het LTS (de configuratiegraaf) reeds gezien in Figuur 9. Figuur 10 toont een C/E systeem B, met bijhorende configuratiegraaf, waaruit blijkt dat A en B isomorfie-equivalent zijn. Inderdaad, als we de knopen in beide configuratiegrafen zouden vervangen door abstracte stippen zouden ze er precies hetzelfde uitzien. Formeel is volgende bijectie β een isomorfisme van de configuratiegraaf van A naar de configuratiegraaf van B:

$$\begin{aligned}\beta(\{p_1, p_2\}) &= \{p_1\}\\ \beta(\{p_1, p_3\}) &= \{p_2\}\\ \beta(\{p_2, p_3\}) &= \{p_3\}\\ \beta(\varnothing) &= \{p_4\}\end{aligned}$$

19 Oefening. Stel de LTS'en op voor de twee C/E systemen uit Voorbeeld 11. Zijn ze isomorf?

20 Oefening. Is volgend C/E systeem isomorfie-equivalent met het C/E systeem voor '$ab + b$' uit Voorbeeld 11?

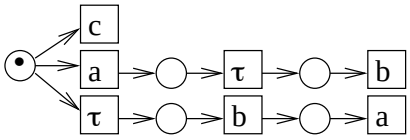


Trace equivalentie Laten we nog eens beter kijken naar de twee C/E systemen besproken in Oefening 20. Hun respectievelijke LTS'en zien er uit als

in Figuur 11. (Merk op dat we de toestanden hebben voorgesteld als abstracte q_i's. Bij het opstellen van de configuratiegraaf is de preciese identiteit van deze toestanden (configuraties) natuurlijk belangrijk, maar nadien, als we deze configuratiegraaf beschouwen als een abstract LTS, kunnen de configuraties geabstraheerd worden.) Alhoewel de twee LTS'en niet isomorf zijn (het antwoord op Oefening 20 is dus neen), laten ze wel dezelfde *traces* toe. Een *trace* is een string van acties, die je krijgt door vanuit de initiële toestand een eindig pad te volgen in de graaf en de acties in de labels van de pijlen te concateneren, in volgorde van het pad. Telkens we de stille actie τ tegenkomen, echter, noteren we die niet. We laten dus de τ's uit de trace weg, wat immers ook de bedoeling is van stille transities. Beide LTS'en uit de figuur hebben dezelfde verzameling traces, namelijk, $\{a, ab, b\}$.

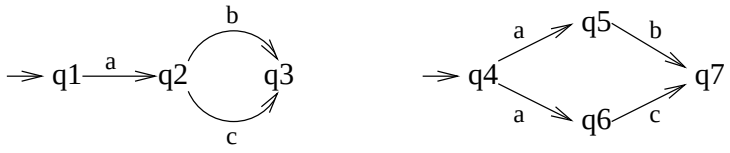
We noemen twee LTS'en *trace-equivalent* als ze dezelfde verzamelingen traces hebben, en we noemen twee C/E systemen trace-equivalent als hun LTS'en dat zijn. De intuïtie van trace-equivalentie is dat de twee systemen precies dezelfde "sequentiële runs" toelaten.

21 Oefening. Is volgend C/E systeem trace-equivalent met het C/E systeem uit Voorbeeld 12?



Merk op dat de verzameling traces van een LTS oneindig is als er cykels zijn in de graaf. Bij een acyclische LTS daarentegen is de verzameling traces altijd eindig. Het is dus, voor cyclische LTS'en toch, niet onmiddellijk duidelijk of trace-equivalentie effectief beslisbaar is (d.w.z., automatisch kan gecheckt worden). Gelukkig is dit wel zo: een LTS kan namelijk bekeken worden als een eindige automaat; de initiële toestand van het LTS fungeert als begintoestand van de automaat. Aangezien een trace een willekeurig pad kan volgen in het systeem, beschouwen we alle toestanden als eindtoestanden van de automaat. Stille transities beschouwen we als ϵ-transities. Twee LTS'en zijn dan trace-equivalent precies als ze, bekeken als eindige automaten, dezelfde reguliere taal definiëren, en we weten dat gelijkheid van reguliere talen effectief beslisbaar is.

Bisimulatie-equivalentie Trace-equivalentie is minder streng dan isomorfie-equivalentie, in de zin dat twee systemen die isomorfie-equivalent zijn, uiteraard ook trace-equivalent zijn, maar niet andersom zoals we hebben gezien in Figuur 11. Voor sommige toepassingen is trace-equivalentie echter soms te weinig streng, in de zin dat er voorbeelden zijn van trace-equivalente systemen die we eigenlijk niet als equivalent willen beschouwen. Een typisch voorbeeld van deze situatie wordt gegeven in Figuur 12. De figuur toont twee LTS'en die we aanduiden met '$a(b+c)$' en '$ab+ac$'. De twee systemen zijn duidelijk trace-equivalent: hun verzameling traces is dezelfde, namelijk $\{a, ab, ac\}$. Kwalitatief hebben de twee systemen echter een duidelijk verschilpunt: in $a(b + c)$ doe je eerst a, en daarna heb je nog de keuze om b of c te doen. In $ab + ac$ echter moet je vooraf de keuze maken. In toepassingen waar je het resultaat van actie a nodig hebt om de keuze tussen b en c te maken is dit een belangrijk verschil.



Figuur 12: Wel equivalent voor traces, maar niet voor bisimulatie.

Het begrip *bisimulatie-equivalentie* formaliseert de intuïtie dat twee systemen niet enkel precies dezelfde sequentiële runs hebben, maar ook ten allen tijde doorheen die runs dezelfde keuzes kunnen maken. Bisimulatie-equivalentie is niet hetzelfde als isomorfie-equivalentie: de twee systemen van Oefening 20 zijn niet isomorfie-equivalent, maar ze zijn wel bisimulatie-equivalent.

Er zijn veel manieren om bisimulatie formeel te definiëren; wij doen het hier aan de hand van een concreet algoritme.[5] Beschouw twee LTS'en A en B waarvan we willen weten of ze bisimulatie-equivalent zijn. (Zoals altijd zijn twee C/E systemen dan bisimulatie-equivalent als hun LTS'en dat zijn.) We geven de toestanden van A en B verschillende identifiers zodat er geen verwarring kan optreden. Het algoritme gaat nu alle toestanden samennemen en verschillen opspeuren tussen de toestanden. Als op het einde van het algoritme er nog steeds geen verschil is gevonden tussen de initiële toestand van A en die van B, dan zijn A en B bisimulatie-equivalent, en anders niet.

Initiëel zijn alle toestanden voor het algoritme "hetzelfde". We stoppen ze dus allemaal in eenzelfde zak, die we het nummer 1 geven. In latere iteraties zullen we meerdere zakken hebben: toestanden waar we een verschil hebben opgemerkt gaan in verschillende zakken terecht komen. Initiëel bepalen we ook voor elke toestand q het *pre-type*. Het pre-type van q is de eindige verzameling bestaande uit alle koppels van de vorm (a, p) die voldoen aan de volgende voorwaarde:

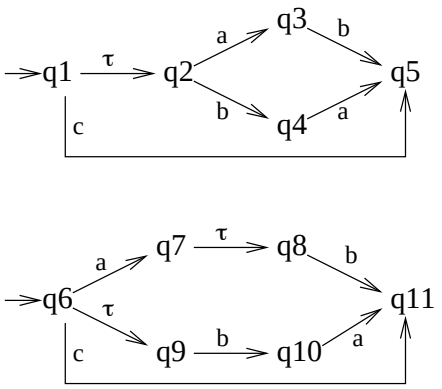
> er is een pad van q naar p bestaande uit 0 of meer stille transities; gevolgd door een a-transitie; en terug gevolgd door 0 of meer stille transities.

Hier bedoelen we met een "stille transitie" natuurlijk een pijl $\xrightarrow{\tau}$ en met een "a-transitie" een pijl $\xrightarrow{a}$.

In elke iteratie van het algoritme heeft ook elke toestand een *huidig type*. Het huidig type van een toestand q wordt bekomen uit het pre-type van q simpelweg door elke toestand p die erin voorkomt te vervangen door het nummer van de zak waarin p zit. In de eerste iteratie is dat nummer nog 1 voor elke p. Dubbels worden verwijderd. (Dubbels kunnen ontstaan in het geval dat het pre-type zowel een koppel (a, p_1) als een koppel (a, p_2) bevat met dezelfde a, en waarbij p_1 en p_2 in dezelfde zak zitten.)

Nadat we zo het huidig type hebben bepaald voor elke toestand, partitioneren we de toestanden volgens hun huidig type: toestanden die hetzelfde huidig type hebben (dus dezelfde verzameling) komen in eenzelfde zak terecht. We hernummeren de nieuwe zakken, en we doen een nieuwe iteratie.

[5] In compilers wordt hetzelfde algoritme gebruikt om de equivalentie te checken van recursieve typedefinities.



Figuur 13: Twee bisimulatie-equivalente LTS'en (boven en onder).

Tabel 1: Pre-types van de toestanden uit Figuur 13.

toestand	pre-type
q_1	$\{(a, q_3), (b, q_4), (c, q_5)\}$
q_2	$\{(a, q_3), (b, q_4)\}$
q_3	$\{(b, q_5)\}$
q_4	$\{(a, q_5)\}$
q_5	$\varnothing$
q_6	$\{(a, q_7), (a, q_8), (b, q_{10}), (c, q_{11})\}$
q_7	$\{(b, q_{11})\}$
q_8	$\{(b, q_{11})\}$
q_9	$\{(b, q_{10})\}$
q_{10}	$\{(a, q_{11})\}$
q_{11}	$\varnothing$

Wanneer we geen zakken meer moeten splitsen, dus wanneer we geen nieuwe types meer ontdekken, stopt het algoritme. Indien dan de initiële toestanden van A en B nog steeds in dezelfde zak zitten, zijn A en B bisimulatie-equivalent. Merk op dat het algoritme altijd moet eindigen, want in het slechtste geval komt elke toestand in een aparte zak te zitten en dan kan er ook niets meer gesplitst worden. Als er dus in totaal n toestanden zijn kunnen er hoogstens n iteraties gebeuren.

Als voorbeeld voeren we het bisimulatie-algoritme uit op de C/E systemen van Oefening 21. De LTS'en worden getoond in Figuur 13 (je moest die opstellen om de oefening te maken). De pre-types worden getoond in Tabel 1. De eerste twee iteraties van het algoritme worden dan getoond in Tabel 2. De derde iteratie is niet meer getoond omdat die dezelfde types oplevert als de tweede iteratie (ga dit na!) Het algoritme stopt dus na de derde iteratie. Op het einde zitten de initiële toestanden q_1 en q_6 nog altijd in dezelfde zak, en we besluiten dat de systemen bisimulatie-equivalent zijn.

22 Oefening. Ga na aan de hand van het bisimulatie-algoritme dat de systemen uit Figuur 12 inderdaad **niet** bisimulatie-equivalent zijn.

Tabel 2: Run van het bisimulatie-algoritme op Figuur 13.

iteratie	toestand	huidig type	nieuw zaknummer
1	q_1	$\{(a,1),(b,1),(c,1)\}$	1
	q_2	$\{(a,1),(b,1)\}$	2
	q_3	$\{(b,1)\}$	3
	q_4	$\{(a,1)\}$	4
	q_5	$\varnothing$	5
	q_6	$\{(a,1),(b,1),(c,1)\}$	1
	q_7	$\{(b,1)\}$	3
	q_8	$\{(b,1)\}$	3
	q_9	$\{(b,1)\}$	3
	q_{10}	$\{(a,1)\}$	4
	q_{11}	$\varnothing$	5
2	q_1	$\{(a,3),(b,4),(c,5)\}$	1
	q_2	$\{(a,3),(b,4)\}$	2
	q_3	$\{(b,5)\}$	3
	q_4	$\{(a,5)\}$	4
	q_5	$\varnothing$	5
	q_6	$\{(a,3),(b,4),(c,5)\}$	1
	q_7	$\{(b,5)\}$	3
	q_8	$\{(b,5)\}$	3
	q_9	$\{(b,4)\}$	6
	q_{10}	$\{(a,5)\}$	4
	q_{11}	$\varnothing$	5

23 Oefening. Ga na dat de twee systemen van Oefening 20 **wel** bisimulatie-equivalent zijn.

Tot slot merken we op dat bisimulatie-equivalentie strenger is dan trace-equivalentie: twee systemen die bisimulatie-equivalent zijn, zijn ook altijd trace-equivalent (zie je in waarom?) maar het omgekeerde is dus niet altijd waar. We hebben dus volgende implicaties tussen de drie equivalentiebegrippen, waarbij de implicaties strikt zijn in de zin dat we tegenvoorbeelden hebben voor de omgekeerde richtingen:

$$\text{isomorfie-equivalentie} \Rightarrow \text{bisimulatie-equivalentie} \Rightarrow \text{trace-equivalentie}$$

8 Geavanceerde Petri net modellen

We mogen deze tekst niet besluiten zonder Place/Transition nets te vermelden. Dit is een ingewikkeldere vorm van systemen, eveneens beschreven door Petri netten, dan de C/E systemen die we in deze tekst hebben behandeld. Het belangrijke verschil is dat places in P/T nets meerdere tokens kunnen bevatten. Dit maakt het systeem krachtiger (je kan dan b.v. buffers modelleren), maar de theorie en de automatische verificatie (model checking, equivalentie) worden een pak ingewikkelder. Een andere uitbreiding van Petri netten zijn de *coloured* Petri netten, die naast control flow ook dataflow toelaten. We verwijzen naar de literatuur voor meer informatie.

Bibliografie

[1] W. Reisig: *Petri Nets, An Introduction*. Springer-Verlag, 1985.

[2] W. Reisig en G. Rozenberg (editors): *Lectures on Petri Nets, volume I: Basic Models* en *volume II: Applications*, volumes 1491 en 1492 in *Lecture Notes in Computer Science*, Springer, 1998.