Processen, Modellen en Spin

Deel 1: Promela programma is een executeerbaar model van gedistribueerde processen

Software, en met name control, bestaat dikwijls uit verschillende processen of threads die met elkaar moeten samenwerken. De non-deterministische afwisseling van executies in die processen kan leiden tot zeldzame en moeilijk reproduceerbare fouten. In de 'process meta language' (Promela) kunnen procesmodellen worden beschreven, die met de 'Spin model checker' kunnen worden 'doorgemeten' op door de ontwerper gespecificeerde systeemtoestanden en executiepaden. Dit eerste artikel van twee beschrijft het modeleren in Promela; het tweede behandelt de verificatie van dergelijke modellen met de Spin-tools.

HANS VAN THIEL

Een klassiek computerprogramma leest een bepaalde invoer, voert daarop berekeningen uit, schrijft het resultaat naar een opslagmedium en stopt dan. De software op een moderne PC bestaat al uit vele verschillende programma's (processen) die schijnbaar tegelijkertijd (concurrent) lopen, in werkelijkheid steeds afgewisseld op dezelfde CPU en gecoördineerd door een OS. Een netwerk, in een auto bijvoorbeeld, verbindt verschillende systemen die onafhankelijk van elkaar programma's uitvoeren, maar ook dikwijls met elkaar moeten communiceren om hun taken te vervullen. Kenmerken van dergelijke gedistribueerde programma's zijn dat zij nooit stoppen, alleen soms wachten op andere processen, en dat hun snelheden en de verschillen daarin niet van te voren vast liggen.

Omdat elke executiestap in een bepaald proces op elk moment afgewisseld kan worden door een executiestap in een van de andere processen (interleaving), wordt al gauw een gemeenschappelijke

'resource', bijvoorbeeld een globale variabele, op het verkeerde moment veranderd of afgelezen. Verder kunnen processen in een toestand komen dat de ene op de andere wacht en de andere op de ene (deadlock), of dat een bepaald proces steeds verdrongen wordt door de anderen (starvation).

Het ergste is dat sommige combinaties die tot een fout leiden zo zeldzaam zijn dat ze niet voorkomen tijdens het testen, of, als ze al voorkomen, niet gereproduceerd kunnen worden. Dergelijke onwaarschijnlijke fouten manifesteren zich weliswaar soms nooit, ook niet als het betreffende product in bedrijf is, maar soms ook wel, en dan met desastreuze gevolgen.

Modellen

Het is echter mogelijk om vereenvoudigde modellen op te stellen van de uiteindelijke software, waarin van de (meeste) berekeningen wordt geabstraheerd, maar juist de communicatie tussen en coördinatie van software processen wordt benadrukt. Dit modeleren voegt een extra stap toe aan het ontwerptraject, maar aan het begin, daar waar fouten het gemakkelijkst hersteld kunnen worden en de kosten ervan nog laag zijn.

Extra aandacht is wel nodig voor validatie van het model, de toetsing of dat inderdaad op de wezenlijke punten overeenkomt met het gedachte systeem. Ten tweede moet de uiteindelijke 'echte' software op de wezenlijke punten natuurlijk overeenstemmen met het model daarvan.

De tussenstap zelf, het model van de gedistribueerde processen, kan echter in principe volledig worden geverifieerd op alle mogelijke interacties die in het model kunnen voorkomen (1). Dit kan als het model bestaat uit een toestandsdiagram met mogelijk oneindige (door cyclussen) executiepaden, maar een eindig aantal toestanden. De moeilijkheid is dan om de complexiteit te beheersen door toepassing van de juiste zoektechnieken en -algoritmen. Dit nu wordt in hoge mate gerealiseerd door de Spin model checker (2, 3).

Spin is een open source tool dat is ontwikkeld door Gerard Holzmann van het NASA JPL 'laboratory for reliable software' (zie 4). Holzmann schreef ook het standaardwerk over Spin, maar er zijn inmiddels diverse boeken over gepubliceerd (zie 3). Spin is in de praktijk voor diverse projecten gebruikt. In Nederland is het toegepast bij het ontwerp van het 'beslis- en ondersteuningssysteem' van de stormvloedkering in de Nieuwe Waterweg.

De modellen, die met Spin kunnen worden gesimuleerd en geverifieerd, zijn geschreven in de 'process meta language'. Een Promela programma is dus een executeerbare (door simulatie) specificatie van de uiteindelijke software.

Non-determinisme

Promela lijkt veel op de C programmeertaal, zoals in Listing 1 is te zien, maar het abstraheert van de meeste berekeningen die in de uiteindelijke software zullen voorkomen. Zo heeft Promela geen functies (wel inline macro's) en geen schuivende-komma variabelen. Ook control structuren zoals 'while', 'for' en 'if... else' ontbreken. Een reeks vertakkingen met tests wordt aangegeven met if ... fi voor een eenmalige executie en do ... od voor een lus. Een test begint met twee dubbele punten en de statements na de test komen na een pijl (of een punt-komma, naar keuze van de programmeur).

Spin is een open source tool dat is ontwikkeld door Gerard Holzmann van het NASA JPL 'laboratory for reliable software'

Het belangrijkste in Promela is dat de programma-statements na een test alleen worden uitgevoerd als de test slaagt. Tot die tijd is het betreffende proces geblokkeerd. Verder is de volgorde van de tests non-deterministisch, en kunnen er meer tests 'tegelijkertijd' tot 'true' evalueren.

Statements binnen een proces zijn sequentieel en dus deterministisch binnen dat proces. Elk statement in een bepaald proces kan echter afgewisseld worden door een statement van een ander proces. Dat geldt ook voor een statement direct na een test. Als dus bijvoorbeeld een globale variabele x in het ene proces getest wordt op gelijkheid met 5, kan die variabele in een ander proces op o gezet worden. Pas dan wordt in het eerste proces de volgende expressie geevalueerd op basis van de, nu foutieve, waarde 5.

Het doel is om een softwaremodel te ontwerpen dat vrij is van zulke fouten, en de functie van de model checker is om ze te vinden als ze er toch in voorkomen.

Besturing

Listing 1 is een model van de besturingssoftware van een spoorwegovergang. Het proces 'Trein' gebruikt hiervoor een lokale variabele die aangeeft of de trein ver weg is, de overgang nadert, of op de overgang is.

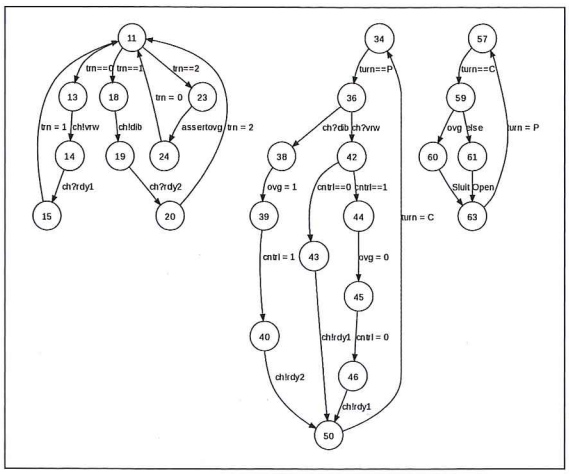
Het proces 'Controller' onderscheidt tussen de mogelijkheden dat de trein ver weg is of dichtbij, met een extra bit dat aangeeft of de trein nadert of juist vertrekt. De overgang wordt gerepresenteerd door een globale variabele die door een derde proces, 'Log' wordt afgelezen. Alle drie processen zijn non-terminerend; als het eind van de lus is bereikt begint het proces weer. (De goto statement en de do ... od lus zijn hier gelijkwaardig.)

Alle drie processen zijn gedistribueerd. Zonder coördinatie zou het kunnen dat de trein dichtbij is, de controller aangeeft dat hij ver weg is, de overgang open staat, en dat de logger niet de actuele maar een vorige toestand noteert.

Misschien is deze weergave van de gang van zaken enigszins misleidend, omdat die suggereert dat de listing de trein en de overgang simuleert. Het Promela programma simuleert echter de besturingssoftware, en slechts indirect de daadwerkelijke trein met overgang.

Figuur 1 is een door JSpin (zie noot) gegenereerde grafische representatie van alle drie processen uit listing 1. Elke toestand wordt in de figuur voorgesteld door een cirkel met het betreffende regelnummer in die listing. Elke evaluatie van een expressie, dus ook een test, is een toestandsovergang.

Processen in Promela kunnen communiceren via message channels, of via globale variabelen. Het proces 'Log' is geco-



Figuur 1: Een grafische representatie, gegenereerd door JSpin, van de drie processen uit listing 1. Cirkels representeren toestanden; de cijfers corresponderen met de regelnummers in de listing. In een Promela model is een test geen toestand, maar een toestandsovergang. Het model van het gehele systeem bestaat uit alle mogelijke combinaties van toestanden en overgangen.

ordineerd met de controller via het producer/consumer mechanisme; elk proces wacht op zijn beurt terwijl de ander iets doet, en elk proces geeft zelf aan het eind de beurt aan de ander. De trein en de controller processen communiceren via een message channel.
In de declaratie wordt aangegeven dat het betreffende kanaal ongebufferd is (door het getal 0) en dat het type van de boodschap 'mtype' is.
Een message channel kan zenden, aangegeven door een uitroepteken achter de naam, of ontvangen, aangegeven door een vraagteken. Achter het uitroep- of vraagteken staat de inhoud. Als een kanaal niet gelezen kan worden wordt het geblokkeerd tot dit wel het geval is.
Omdat kanaal chan in listing 1 ongebufferd is werkt het, met twee verschillende messages en twee responsboodschappen, als rendez-vous tussen de twee processen.
De drie processen kunnen nu met Spin en desgewenst de grafische tools XSpin en JSpin worden gesimuleerd en geverifieerd. De assert in het trein proces is analoog aan een assert statement in C; als de conditie niet geldt wordt dat als een software fout beschouwd. Anders dan in een C programma (en in Spin simulatie modus) geldt de conditie in verificatiemodus niet alleen voor de uitgevoerde executies, maar voor alle executies die maar mogelijk zijn. Deze volledige verificatie van Promela modellen is het onderwerp van het tweede artikel.

Literatuur

1] "Principles of Model Checking", Christel Baier and Joost-Pieter Katoen, MIT Press, 2007.
2] "The Spin Model Checker, Primer and Reference Manual", Gerard J. Holzmann, Addison-Wesley, 2004
3] http://spinroot.com/spin/sitemap.html
4] Elektronica + embedded systems 7/8-2008, p 30-31

Noot

Er is een grafische interface XSpin, gebaseerd op Tcl/Tk, en een andere, JSpin, die in Java is geschreven door Mordechai Ben-Ari. Spin, XSpin en JSpin zijn vrij verkrijgbaar op de web site (3). Voor de grafische tools zijn Tcl/Tk, Java en Graphviz (visualisatie diagrammen) vereist, en voor verificatie (niet simulatie) een C compiler. De huidige versie 5.2.4 van Spin is van december 2009. ■

```
mtype = {vrw, dib, rdy1,rdy2 };  /* signalen voor message channel ch */
chan ch = [0] of { mtype };      /* ongebufferd channel, message is 1 mtype veld */
bool ovg= false;                 /* globale variabele voor overgang */
mtype = {P, C };                 /* producer/consumer waarden */
mtype turn = P;                  /* globale variabele voor Log synchronisatie */

active proctype Trein() {
      byte trn;      /* default altijd 0, krijgt hier waarden 0, 1 of 2 */

 again :
     if
     :: trn == 0 ->          /* trein is ver weg */
                ch!vrw;   /* stuurt message dat trein ver weg is */
                ch?rdy1;  /* controller is klaar */
                trn = 1;  /* volgende toestand trein */

     :: trn == 1 ->          /* trein is dichtbij */
                ch!dib;   /* stuurt message dat trein dichtbij is */
                ch?rdy2;  /* controller is klaar */
                trn = 2;  /* volgende toestand trein */

     :: trn == 2 ->            /* trein is op overweg */
                assert(ovg);  /* test dat overweg dicht is */
                trn = 0;      /* volgende toestand trein */
     fi;

     goto again
}

active proctype Controller() {

     bit cntrl;   /* verre trein kan komen (0) of vertrekken (1)  */

     do
     :: turn == P ->
     if
     :: ch?dib  ->           /* ontvang message dat trein dichtbij is */
              ovg = true; /* zet spoorboom sein op dicht */
              cntrl = 1;  /* cntrl toestand wordt dat trein vertrekt */
              ch!rdy2;    /* stuur ready message 2 */

     :: ch?vrw  -> if
              :: cntrl == 0 ->  ch!rdy1;      /* stuur ready message 1 */
              :: cntrl == 1 ->  ovg = false;  /* spoorboom sein op open */
                        cntrl = 0;     /* (nieuwe) trein is ver */
                        ch!rdy1;       /* stuur ready message 1 */
              fi;
     fi;

     turn = C;
     od
}

active proctype Log() {

     do
     :: turn == C ->
     if
     :: ovg  -> printf("Sluit\n");
     :: else -> printf("Open\n");
     fi;
     turn = P;
     od
}
```

Listing 1: Een specificatie van de besturingssoftware van een trein en een spoorwegovergang in Promela. De trein communiceert met de controller via rendez-vous messages. De controller is gesynchroniseerd met een logger via een globale variabele. De 'assert' in het trein proces verifieert dat, wanneer de trein over de overgang rijdt, de overgang is gesloten.

Processen, Modellen en Spin

Met Linear Temporal Logic kunnen executiepaden in Promela worden gechecked

Veel embedded software, vooral die voor control, bestaat uit verschillende processen die met elkaar communiceren en samenwerken. De Spin Model Checker is een tool voor simulatie en verificatie van modellen van zulke gedistribueerde systemen. De software modellen zijn geschreven in een taal die op C lijkt, de Process Meta Language. Promela werd besproken in een eerder artikel; dit tweede is een inleiding tot de verificatie met Spin en de bijbehorende grafische tool XSpin.

HANS VAN THIEL

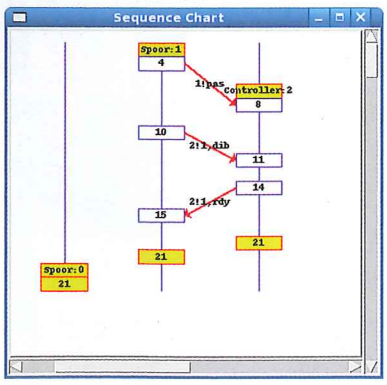
Gedistribueerde of concurrent software bestaat uit verschillende processen of threads die enerzijds gelijktijdig of parallel aan elkaar lopen, en anderzijds op bepaalde punten met elkaar gecoördineerd moeten worden. Vooral in control software, zoals in het voorbeeld in het vorige artikel van een trein en een spoorwegovergang, is dat het geval. Testen van zulke software wordt bemoeilijkt doordat bepaalde combinaties van toestanden heel sporadisch kunnen optreden, en dus moeilijk zijn te reproduceren. Anderzijds zijn de consequenties van dergelijke fouten dikwijls onaanvaardbaar.

Met de specificatietaal Promela kunnen nu modellen worden beschreven, waarin geabstraheerd wordt van de eigenlijke software zelf, en de nadruk juist ligt op de communicatie en coördinatie van processen.

De Spin model checker, die ontwikkeld is door Gerard Holzmann van het NASA JPL Laboratory for Reliable Software (2,3,4) kan dergelijke modellen niet alleen simuleren maar ook 'doormeten' op allerlei eigenschappen. Dit checken is volledig; alle relevante mogelijke toestanden van het model worden in aanmerking genomen, en het resultaat is een volledige verificatie (van het model). De huidige versie van Spin (mei 2008) is het resultaat van jarenlang onderzoek (zie ook 2) en is een toepassing van formele methoden in softwareontwikkeling. Spin is ook in Nederland gebruikt; een bekend geval is de verificatie van software voor de stormvloedkering in de Nieuwe Waterweg.

```
mtype = { dib, pas, rdy };          /* message typen voor overweg */
bool ovg= false;                    /* globale variabele voor overgang */
chan slch = [0] of { pid, mtype };  /* rendez-vous channel voor dichtbij */
chan opch = [1] of { mtype };       /* gebufferd channel voor gepasseerd */

active [2] proctype Spoor() {

   byte trn= 3;  /* treinen zijn dichtbij (1), op overweg (2), of gepasseerd (3) */

  again :
        if
        ::  trn == 1 ->                 /* trein is dichtbij */
                        slch!_pid,dib;  /* stuurt message dat trein dichtbij is */
                        slch?_pid,rdy;  /* controller is klaar */
                        trn = 2;        /* volgende toestand trein */

        ::  trn == 2 ->                 /* trein is op overweg */
                        assert(ovg);    /* test dat overweg dicht is */
                        trn = 3;        /* volgende toestand trein */

        ::  trn == 3 ->             /* trein is gepasseerd */
                        opch!pas;   /* stuurt message dat trein gepasseerd is */
                        trn = 1;    /* volgende toestand trein */
        fi;

        goto again
}

active proctype Controller() {

   byte psn = 2;    /* aantal treinen dat gepasseerd is */
   pid wlk;         /* process id van trein */

   do
   :: slch?wlk,dib ->                   /* lees message trein pid, dichtbij */
                      ovg = true;       /* sluit overgang */
                      psn--;            /* 1 gepasseerde trein minder */
                      slch!wlk,rdy;     /* stuur ready message naar trein wlk */

   :: opch?pas  ->
                  psn++;            /* 1 gepasseerde trein meer */
   :: psn == 2 ->                   /* 2 treinen gepasseerd */
                  ovg = false;      /* open overgang */
   od
}
```

Listing 2: Een Promela model van twee treinen op een dubbelspoor, met een overgang en een controller. Simulatie en verificatie met Spin laten zien dat de assert faalt; de overgang kan open staan terwijl er een trein op rijdt.



Figuur 2: Een 'guided' simulatie na een verificatie van Listing 2 kan het kortste pad tonen dat tot de fout voert. Het sequentiediagram van XSpin geeft al een aanwijzing, die door de (hier niet getoonde) simulatie output wordt bevestigd. Weliswaar sluit de overgang op correcte wijze als de trein is genaderd, maar hij kan weer geopend worden voordat de trein voorbij is. In het simpelste geval heeft proces 0 zelfs nog geen boodschap verzonden.

Treinen

Het eerste artikel behandelde een Promela-model van een trein met een spoorwegovergang, een controller en een logger. In het proces voor de trein stond een 'assert' statement, die moest controleren of de overgang altijd dicht stond als de trein zich op de overweg bevond. Zo'n 'assert' is volledig vergelijkbaar met de gelijknamige constructie in de C programmeertaal. In een Spin-simulatie worden alle doorlopen toestanden gecontroleerd, en stopt de executie als de 'assert' tot 'false' evalueert. in een Spin-verificatie worden echter alle mogelijke toestanden van het model gechecked op de 'assert'. Dat het model, met betrekking tot die claim, correct functioneert is zeker.

Uiteraard is het niet zeker dat het model ook het werkelijke systeem correct representeert, en dat de uiteindelijke code niet afwijkt van het model, maar verificatie dekt wel alle mogelijkheden in dat model.

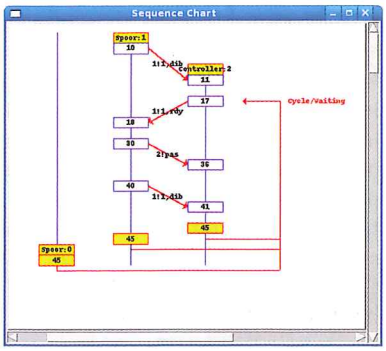
Listing 2 bouwt voort op listing 1 uit het vorige artikel. Nu zijn er twee identieke (non-deterministische) processen die treinen op twee verschillende sporen modeleren. Er is een controller, en voor de eenvoud is de logger weggelaten. Het idee is dat de overweg altijd moet sluiten als er een trein nadert (ook als hij al dicht staat). Elke trein moet wachten tot de controller bericht dat de overweg gesloten is, en dit wordt weer geïmplementeerd door ongebufferde rendez-vous messaging. De inhoud van een message bestaat uit een ready-boodschap en de procesindentificator (keyword _pid), zodat de processen zichzelf kunnen herkennen. Als het kanaal geblokkeerd is door een trein, dan moet een andere wachten. Voor het sluiten is dit ongewenst; elke trein moet gewoon kunnen doorrijden als de overweg is gepasseerd. Hiervoor

wordt een gebufferd kanaal gebruikt.
Elke trein zendt, anders dan in het eerste voorbeeld met slechts 1 spoor, een boodschap bij het naderen en een na het passeren. Weer controleert een assert of de overweg altijd dicht is als er een trein over heen gaat.
Nu faalt de assert echter. Spin biedt in zo'n geval de mogelijkheid om het kortste pad te vinden dat tot de fout leidt, en een 'guided' simulatie uit te voeren. XSpin genereert, als visueel hulpmiddel, ook 'message sequence' diagrammen.
Weliswaar lijkt de overweg op een juiste wijze te worden gesloten als een trein nadert, maar hij kan weer worden geopend voordat de trein is gepasseerd.
Het model is niet goed. Bovendien is uit de simulatieuitvoer (hier niet getoond) op te maken dat de 'assert' niet alle ongewenste situaties afdekt.

Een beter model staat in listing 3. Elke trein krijgt een extra toestand aan het begin (corresponderend met 'nog ver weg') en de overweg wordt nu alleen geopend als de teller op 0 staat. Niet het passeren van de treinen wordt geteld, maar het aantal dat is genaderd en nog niet gepasseerd, en nu slaagt de 'assert' wel.

LTL

Spin controleert in elke verificatie niet alleen eventuele 'asserts' die de gebruiker heeft geschreven, maar ook standaard of er 'deadlock' (het ene proces wacht op het andere, maar het andere wacht op het ene) kan optreden.
Behalve op deadlocks kan Spin ook controleren op 'liveness' eigenschappen (processen komen weer aan de beurt of worden verdrongen door andere). Hiertoe kunnen verschillende labels worden toegevoegd in Promela modellen, maar het is ook mogelijk om systeemeigenschappen te formuleren in 'Linear Temporal Logic'. XSpin heeft zelfs een kleine grafische editor hiervoor.
LTL is een uitbreiding van de propositielogica, waarmee niet alleen uitspraken kunnen worden gedaan over bereikbare en onbereikbare toestanden, maar ook over executiepaden in het model. In proposietielogica kan men, bijvoorbeeld, formuleren dat, als de trein op de overweg is, dan de overweg dicht staat. In Promela kan met een '#define' (net zoals in C) dan een symbool 'p' gedefinieerd worden als 'x == 2' en de booleaanse variable 'ovg' als zichzelf.
De genoemde uitspraak over de spoorwegovergang wordt dan 'p -> ovg' (p impliceert ovg) in propositielogica, ofwel: als p waar is dan is ovg waar.
In LTL komen er nu een paar extra operatoren bij, waarvan de meest gebruikte 'eventually' en 'always' zijn. In LTL-formules in XSpin wordt voor 'ooit' de operator '<>' gebruikt, en voor 'altijd' '[]'.
Nu kan een bewering als 'p -> <> ovg' worden geformuleerd, wat dan betekent dat, als variabele x gelijk is aan 2 (volgens de bovengenoemde definitie van p), dan de overweg ooit dicht gaat.
Zoals een 'assert' een uitspraak doet over een systeemtoestand, zo doet een LTL formule een uitspraak over mogelijke reeksen van executies.
Voor het model uit listing 3 kan zo de bewering '(ovg -> <> ! ovg)' worden geformuleerd (het uitroepteken staat voor negatie). Dit betekent: als de overweg dicht staat, dan gaat hij ooit weer open.
Met de LTL-module van XSpin kan deze claim heel gemakkelijk geformuleerd en gechecked worden. De verificatie slaagt, dus in het model van listing 3 zal een gesloten overgang ook ooit weer opengaan.
Maar voorzichtigheid is geboden. De geverifieerde LTL-formule betekent dat er vanuit elke gesloten overgang een reeks executies te vinden is met een open overgang als resultaat. Maar dit betekent nog niet dat in elk van die executiepaden de overgang ook weer geopend zal worden.
De LTL formule '[] (ovg -> <> !ovg))' betekent: het is altijd het geval dat, als de overweg gesloten is, die ooit weer open zal gaan.



Figuur 3: Het model uit Listing 3 lijkt veilig, maar gaat de overweg, als die gesloten is, ook ooit weer open? Verificatie met de LTL formule (ovg -> <> !ovg) slaagt, maar die met LTL formule '([] (ovg -> <> !ovg))' faalt. Er is (minstens) een patroon in de dienstregeling dat, als het zich voortdurend zou herhalen, het openen van de overweg verhindert.

De Spin-verificatie van deze bewering vindt een tegenvoorbeeld, en uit de simulatieuitvoer (niet getoond) en het sequentiediagram blijkt wat er kan gebeuren.
Op spoor 1 passeert een trein en nadert direct een nieuwe. Op het moment dat die zelf passeert nadert nu op spoor 0 een trein, voordat de controller de overweg heeft kunnen openen. Dit kan zich voortdurend blijven herhalen, en daardoor is de sterkere claim, namelijk dat de overweg in alle mogelijke executiepaden ook eens weer open zal gaan, onjuist. Er is (minimaal) een mogelijkheid waarin dat niet gebeurt. Die mogelijkheid is echter niet de enige (non-determinisme) en de zwakkere LTL-claim, dat er executies bestaan waarna de overweg weer open gaat, slaagt. ■

Referenties

1) *"Principles of Model Checking"*, Christel Baier and Joost-Pieter Katoen, MIT Press, 2007.
2) *"The Spin Model Checker, Primer and Reference Manual"*, Gerard J. Holzmann, Addison-Wesley, 2004
3) http://spinroot.com/spin/sitemap.html
4) Elektronica + embedded systems 7/8-2008, p 30-31


```
mtype = { dib, pas, rdy };          /* message typen voor overweg */
bool ovg= false;                    /* globale variabele voor overgang */
chan slch = [0] of { pid, mtype };  /* rendez-vous channel voor dichtbij */
chan opch = [1] of { mtype };       /* gebufferd channel voor gepasseerd */

active [2] proctype Spoor() {

   byte trn= 0;  /* treinen ver (0), dichtbij (1), overweg (2), gepasseerd (3) */

  again :
        if
        :: trn == 0 ->                  /* trein is nog ver */
                       trn = 1;         /* volgende toestand trein */
        :: trn == 1 ->                  /* trein is dichtbij */
                       slch!_pid,dib;   /* stuurt message dat trein dichtbij is */
                       slch?_pid,rdy;   /* controller is klaar */
                       trn = 2;         /* volgende toestand trein */

        :: trn == 2 ->                  /* trein is op overweg */
                       assert(ovg);     /* test dat overweg dicht is */
                       trn = 3;         /* volgende toestand trein */

        :: trn == 3 ->                /* trein is gepasseerd */
                       opch!pas;      /* stuurt message dat trein gepasseerd is */
                       trn = 1;       /* volgende toestand trein */
        fi;

        goto again
}

active proctype Controller() {

   byte tnum = 0;    /* aantal treinen dat nadert of op overweg is */
   pid wlk;          /* process id van trein */

   do
   :: slch?wlk,dib ->                  /* lees message trein pid, dichtbij */
                      ovg = true;      /* sluit overgang */
                      tnum++;          /* 1 trein meer */
                      slch!wlk,rdy;    /* stuur ready message naar trein wlk */

   :: opch?pas  ->
                   tnum--;         /* 1 trein minder */
   :: tnum == 0 ->                 /* geen treinen dichtbij of op overgang */
                 ovg = false;      /* open overgang */
   od
}
```

Listing 3: Dit Promela model van twee treinen en een controller gaat er vanuit dat initieel elke trein ver weg is. De controller telt nu niet het aantal gepasseerde treinen dat nog niet is genaderd, maar het aantal dat is genaderd en nog niet gepasseerd. De overgang gaat open als dat getal 0 is. Nu slaagt de Spin verificatie met 'assert(ovg)' wel.