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

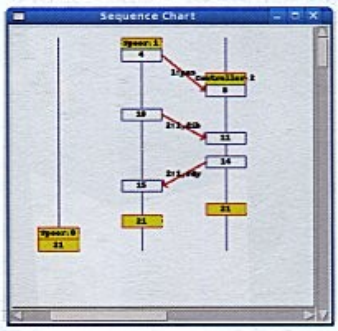
Gedistribueerde of concurrent software bestaat uit verschillende processen of threads die enerzijds gelijktijdig of parallel aan elkaar lopen, en anderzijds op bepaalde punten met elkaar gecoördineerd moeten worden. Vooral in control software, zoals in het voorbeeld in het vorige artikel van een trein 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.

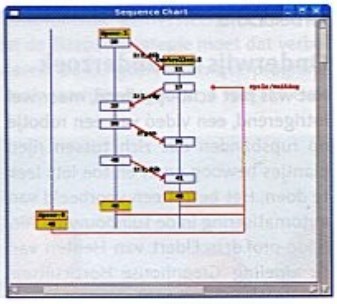
Listing 2 bouwt voort op listing 1 uit het vorige artikel. Nu zijn er twee identieke (non-deterministische) processen die treinen op twee verschillende sporen modeleren. Er is een controller, en voor de eenvoud is de logger weggelaten. Het idee is dat de overweg altijd moet sluiten als er een trein nadert (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.