Uppaal voor Beginners

(concept, commentaar welkom)*

Frits Vaandrager
ICIS, Radboud Universiteit Nijmegen

9 november 2010

Samenvatting

Uppaal is een zogenaamde *model checker*, een programma waarmee je modellen kunt maken van systemen met toestanden en overgangen tussen die toestanden, en waarmee je aan deze modellen kunt rekenen. Dit artikel is een handleiding in het gebruik van Uppaal, bedoeld voor absolute beginners. Gebruikmakend van een voorbeeld van een *werkplaats*, wordt stapsgewijs uitgelegd hoe je een eenvoudig Uppaal model kunt maken, simuleren en analyseren.

1 Inleiding

Uppaal is een programma waarmee je het gedrag van systemen kunt modelleren. Dat kunnen allerlei soorten systemen zijn: een netwerk van computers of een copieermachine, een groep roddelende vriendinnen of een Sudoku puzzel, een mierencolonie of een probleemgezin, een robot of een trein, enz. Eigenlijk kun je het zo gek niet verzinnen of het kan met Uppaal gemodelleerd worden. Eis is alleen dat er sprake is van *toestanden* en van *transities (overgangen)* tussen toestanden. Wanneer je eenmaal een model hebt gemaakt met Uppaal kun je er twee dingen mee doen. Allereerst kun je het model *simuleren*, dat wil zeggen stapsgewijs van de ene naar de andere toestand springen. Daarnaast kun je ook eigenschappen van het model *verifiëren ("model checken")*. Je kunt Uppaal bijvoorbeeld vragen of er berichten verloren kunnen raken in een computernetwerk, of een Sudoku

*Deze handleiding is geschreven met steun van het Sprint Up project. De Uppaal modellen van de werkplaats die in deze handleiding beschreven worden zijn beschikbaar via `http://www.mbsd.cs.ru.nl/publications/papers/fvaan/uppaalhandleiding/`.

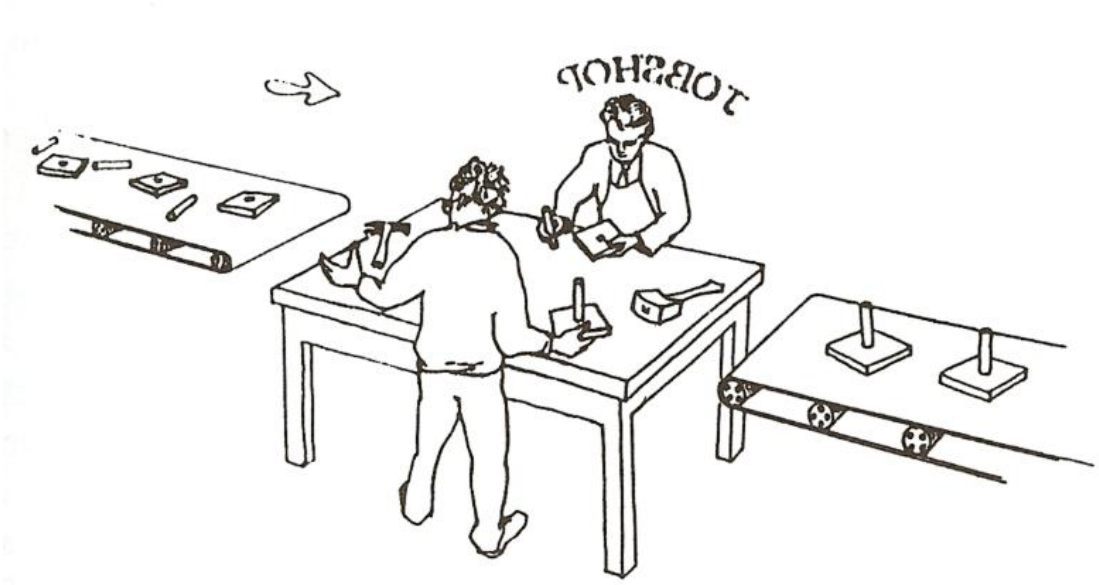
puzzel een oplossing heeft, hoe lang het duurt voordat alle vriendinnen op de hoogte zijn van een roddel, enz. Tijdens het verifiëren doorzoekt Uppaal alle toestanden in een model.

Uppaal kun je gratis downloaden via `www.uppaal.com`. Het programma draait onder Windows, Linux en Mac OS X. Het installeren, ook op je computer thuis, is zeer eenvoudig. Zorg er wel voor dat Java versie 6 (bijvoorbeeld de J2SE Java Runtime Environment) of een recentere versie op je computer draait.

Uppaal is ontwikkeld door onderzoekers uit **Upp**sala (Zweden) en **Aal**borg (Denemarken), met hulp van andere onderzoeksgroepen, waaronder mijn eigen groep in Nijmegen. Uppaal is een zeer uitgebreid tool met een schat aan mogelijkheden om systemen te modelleren en analyseren. In deze handleiding beperk ik me tot de absolute basis. Meer details en documentatie kun je vinden op de Uppaal webpagina en in [1] (dit artikel is voor niet-specialisten wellicht wat moeilijk). Daarnaast geeft het Helpmenu binnen Uppaal zelf een goede uitleg van de diverse mogelijkheden van het programma. Op het Internet zijn nog tientallen andere vergelijkbare programma's (zogenaamde *"model checkers"*) beschikbaar: Spin, Blast, μCRL, Java Pathfinder, enz. Voordeel van Uppaal is dat het een goede grafische user interface heeft en je zonder veel training snel met het programma aan de slag kunt.

2 Een Werkplaats

Ik ga nu stap voor stap uitleggen hoe je een eenvoudig systeem kunt modelleren in Uppaal. Ik doe dit aan de hand van een voorbeeld van een *werkplaats (productielijn)* dat is ontleend aan [2]. Hierbij zijn er twee mensen die samen gebruikmaken van een tweetal hamers — een ijzeren en een houten hamer – om *objecten* te maken. Deze objecten worden gemaakt door een pin in een blok te timmeren. Een paar van een pin en een blok noemen we een *klus*. Klussen komen een voor een binnen via een lopende band, en wanneer ze klaar zijn worden ze afgevoerd via een andere lopende band. Figuur 1 illustreert de situatie. Afhankelijk van de vorm van de pin en het gat in het blok zijn er drie soorten klussen: simpele klussen, moeilijke klussen en klussen met een gemiddelde moeilijheidsgraad. Werkers assembleren de eenvoudige klussen met hun handen, moelijke klussen met de metalen hamer, en de overige klussen met de houten hamer of de metalen hamer, afhankelijk van wat beschikbaar is. Om te beginnen gaan we een model maken van het gedrag van een werker.



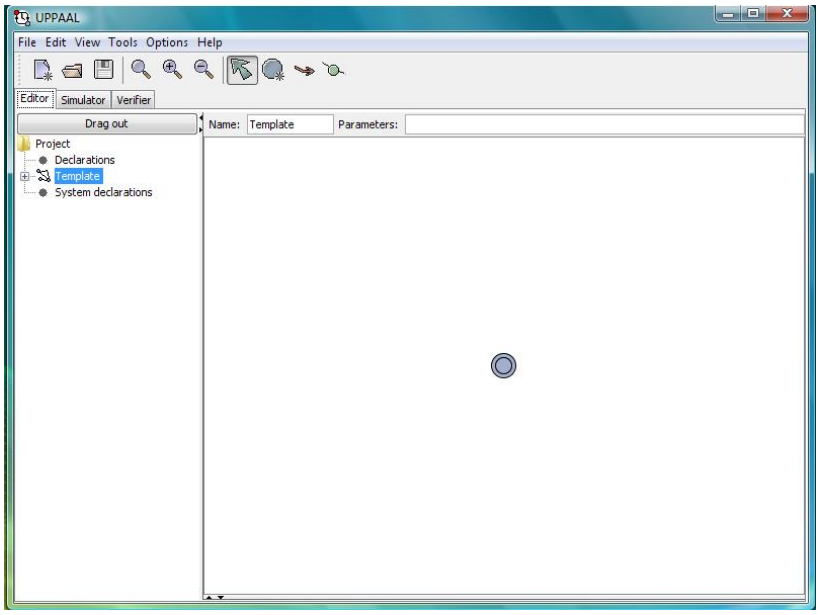
Figuur 1: Een werkplaats (afbeelding ontleend aan [2]).

3 De Editor

Wanneer je Uppaal start krijg je het scherm dat staat afgebeeld in Figuur 2. Er zijn drie tabbladen: de *Editor*, waarin je modellen kunt maken, de *Simulator*, waarin je het gedrag van modellen kunt simuleren, en de *Verifier*, waarin je modellen kunt analyseren. Na het opstarten komt Uppaal altijd eerst in de (grafische) editor. Deze editor heeft 4 "tools" voor het tekenen van modellen (zie Figuur 3): *Select*, *Location*, *Edge* en *Nail*.

- Het *Select tool* kun je gebruiken om elementen te selecteren, verplaatsen en weg te laten. Je kunt een element selecteren door er op te klikken of door een "rubberen band" om een of meer elementen te slepen. Als je een element dubbelklikt komt er een window waarin je dit element kunt editen. Als je met je rechtermuisknop klikt op een element komt er een pop-up menu waarmee je eigenschappen van het element kunt aanpassen of het element kunt weglaten.

- Het *Location tool* kun je gebruiken om nieuwe locaties (toestanden) te maken. Je klikt gewoon op de plek waar je een toestand wilt hebben. Pas op: als je een paar keer achterelkaar klikt worden de toestanden



Figuur 2: Uppaal scherm direct na het opstarten.

Figuur 3: De vier edit tools *Select*, *Location*, *Edge* en *Nail*.

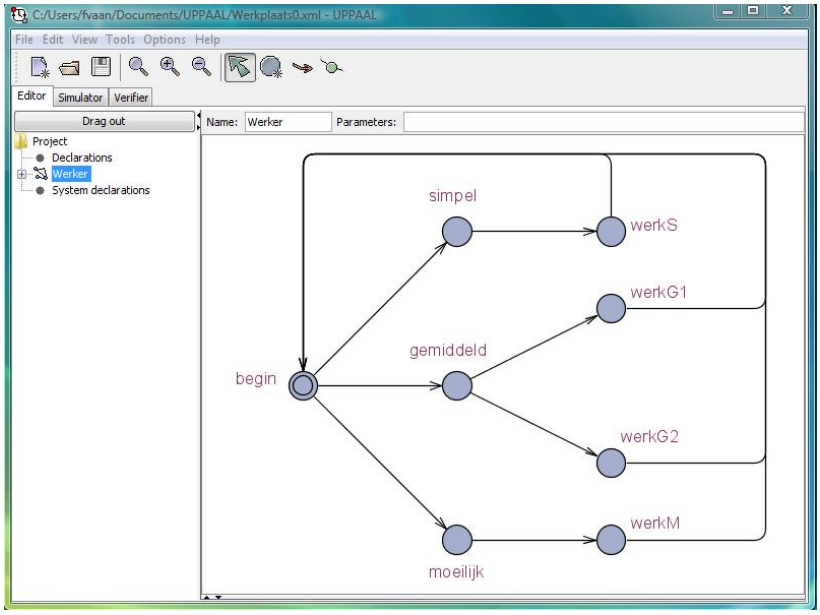
op elkaar gestapeld. Wanneer je een toestand die je net hebt gemaakt wilt verplaatsen of editen moet je daarvoor eerst weer terug naar het select tool.

- Het *Edge tool* stelt je in staat om transities te maken tussen locaties. Klik eerst op de toestand waarin de transitie begint, klik vervolgens op plekken waar je een bocht in de transitie wilt maken (met behulp van zogenaamde *nails*), en klik tot slot op de eindlocatie van de transitie. Je kunt de operatie afbreken door op de middelste of rechter muisknop te drukken.

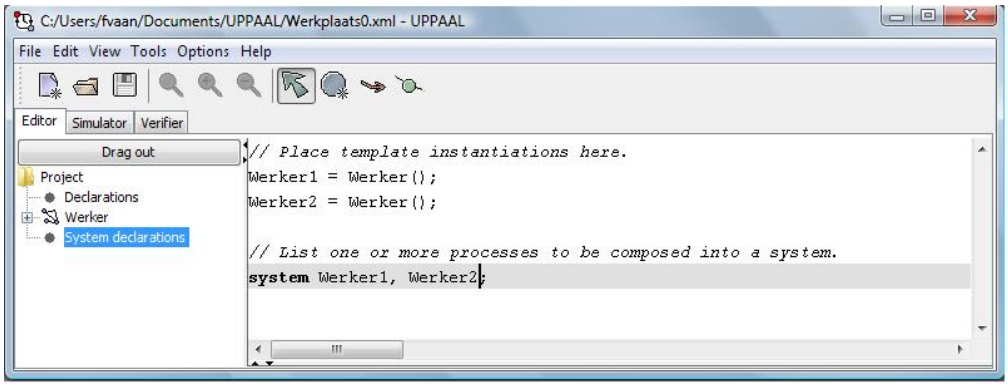
- Met behulp van het *Nail tool* kun je nieuwe nails ("spijkers") op een edge plaatsen. In het select tool kun je vervolgens die nails verplaatsen. Op deze manier kun je bochten in transities maken.

We gaan nu ons eerste Uppaal model maken. In het veld *Name* boven het tekenvenster voeren we eerst de naam in van de eerste component ("template"): `Werker`. Vervolgens klikken we (met het select tool) op de locatie die Uppaal alvast in het tekenvenster heeft neergezet. We kunnen deze locatie dan een naam geven, bijvoorbeeld `begin`. In ieder template is er maximaal één beginlocatie: in deze locatie begint het systeem wanneer je simuleert of verifieert. Door in het menu van een locatie "Initial" te selecteren kun je aangeven of het de beginlocatie is. In het menu staan nog een paar andere velden en opties ("Invariant", "Urgent" en "Committed"), maar daar doen we voorlopig niets mee.

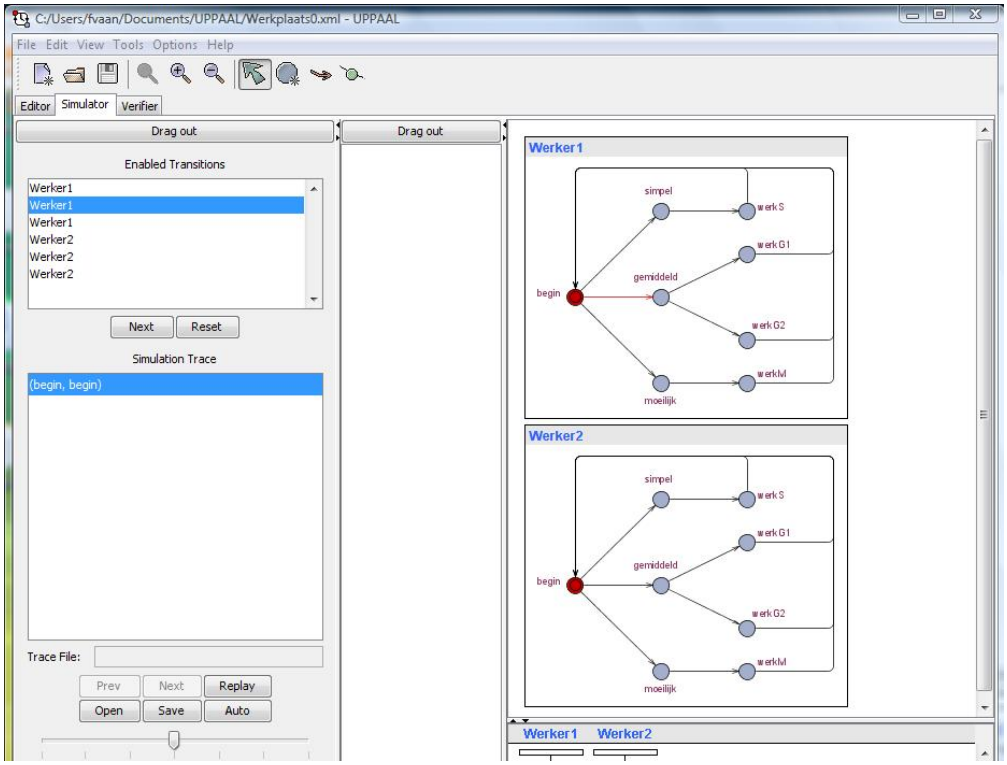
Probeer het model te construeren dat staat afgebeeld in Figuur 4. De eerste transitie vanuit de beginlocatie correspondeert met het moment dat een werker een nieuwe klus van de band pakt. Er zijn drie transities, die corresponderen met de drie soorten klussen: simpel, gemiddeld en moeilijk. De volgende transitie correspondeert met het moment waarop de werker gereedschap pakt (indien nodig) en begint te werken aan de klus. Bij een klus van gemiddelde moeilijkheidsgraad zijn er twee mogelijke transities, afhankelijk van welke hamer de klusser pakt. Wanneer het werk klaar is keert de klusser terug in de begintoestand en start alles overnieuw. Als het je gelukt is om het model van Figuur 4 na te bouwen ben je bijna klaar. Klik op "System declarations" in het linker window. Je ziet nu een scherm waarin je kunt beschrijven uit welke componenten je model bestaat. Zoals Figuur 5 laat zien, geven we aan dat ons eerste systeem bestaat uit twee instanties van het model ("template") van de werker, genaamd `Werker1` en `Werker2`. Deze instanties werken tegelijk en in parallel. Door in de menu bar op "Tools" te klikken en daarna op "Check Syntax" kun je controleren



Figuur 4: Een eerste versie van het model.



Figuur 5: Declaratie van de systeemcomponenten.



Figuur 6: Screenshot van de simulator.

of je model syntactisch correct is. Indien je fouten hebt gemaakt dan worden die rood onderlijnd/gemarkeerd. Een meer gedetailleerde beschrijving van de fouten staat in een window (nauwelijks zichtbaar maar je kunt het groter maken met je muis) helemaal onderin de editor.

4 De Simulator

Zodra een model syntactisch correct is kun je het simuleren door op het tabblad "Simulator" te klikken. Het scherm dat je dan krijgt staat afgebeeld in Figuur 6. Je ziet dat Uppaal twee copieën heeft gemaakt van het template `Werker`. Met rode stippen wordt de huidige locatie van iedere component aangegeven. In het besturingswindow links staat aangegeven dat er zes transities mogelijk zijn (drie voor iedere werker). Wanneer je een transitie selecteert wordt de corresponderende pijl in het diagram voor `Werker1` of

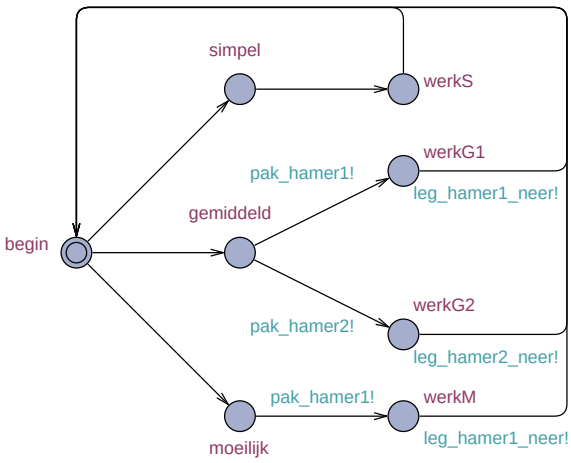
Werker2 rood gekleurd. Met de knop "Next" kun je steeds de geselecteerde transitie uitvoeren. Wanneer je op de knop "Random" klikt, dan voert Uppaal zelf willekeurig gekozen transities uit. De snelheid waarmee dat gebeurt kun je instellen door het pijltje tussen "Slow" en "Fast" te verschuiven.

In de simulator kun je goed zien hoe je in Uppaal grote modellen kunt bouwen door kleine modellen samen te stellen. Werker1 is een klein maar volwaardig model met een toestandsruimte die bestaat uit 8 toestanden (locaties) en 11 toestandsovergangen (transities). Werker2 is ook een volwaardige model, namelijk een identieke copie van Werker2. Door nu te declareren dat het totale systeem bestaat uit Werker1 én Werker2 geven we Uppaal de opdracht om een nieuw model te construeren met $8 \times 8 = 64$ toestanden en $11 \times 11 = 121$ toestandsovergangen. Toestanden in het samengestelde model zijn paren van een toestand van Werker1 en Werker2. Een voorbeeld is het paar (Werker1.begin, Werker2.werkM): dit beschrijft de samengestelde toestand waarbij Werker1 in locatie begin is en Werker2 in locatie werkM. Toestandsovergangen in het samengestelde model corresponderen met toestandsovergangen van de deelmodellen. Voorbeelden van globale toestandsovergangen zijn

(Werker1.begin, Werker2.werkM) $\rightarrow$ (Werker1.simpel, Werker2.werkM) en
(Werker1.begin, Werker2.werkM) $\rightarrow$ (Werker1.begin, Werker2.begin).

5 Kanalen

De simulator is nuttig om inzicht te krijgen in het gedrag van een systeem en om fouten in een model op te sporen. Door met het model te experimenteren in de simulator kun je er bijvoorbeeld snel achterkomen dat er iets nog niet klopt: beide werkers kunnen *tegelijk* in locatie WerkM zijn. Dit is de toestand waarin ze de metalen hamer gebruiken om een moeilijke klus te klaren. Maar dat kan in het echt niet voorkomen, want we hadden aangenomen dat er maar één metalen hamer is (en één houten hamer) en die moeten de werkers delen. We gaan het model aanpassen en dit nauwkeuriger beschrijven. Het idee is om ook templates te maken voor de beide hamers. Iedere hamer kan in twee toestanden zijn: vrij of bezet. Een hamer gaat van toestand vrij naar toestand bezet wanneer hij gepakt wordt door één van de werkers. Om dit "contact" tussen hamer en werker te modelleren gebruiken we in Uppaal synchronisatiekanalen ("channels"). Stel dat a een kanaal is, dan kunnen transities gelabeld worden met a! en a?. Dit doe je door de transitie te dubbelklikken met het select tool, en dan in het veld "Sync" a! of a? te schrijven. Wanneer twee componenten synchroniseren op een kanaal a, dan

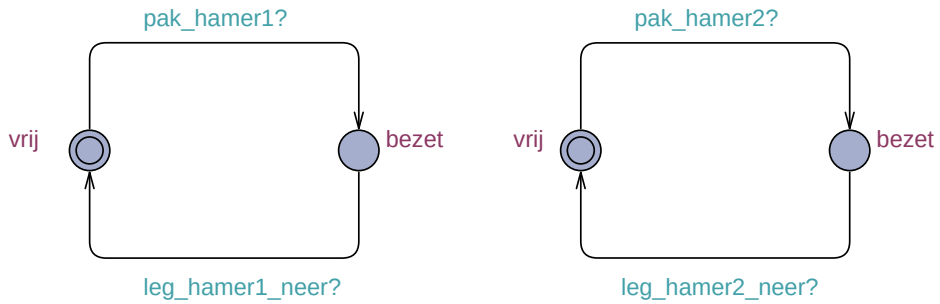


Figuur 7: Model van werker, uitgebreid met synchronisatie-labels.

betekent dit dat `a!`-transities van de ene component tegelijk plaatsvinden met `a?`-transities van een andere component. Een `a!`- of `a?`-transitie kan nooit alleen plaatsvinden: een `a!` moet altijd synchroniseren met een `a?`, en omgekeerd. Strikt genomen maakt het niet uit welke transitie een ? en welke transitie een ! krijgt. Vaak plaatsen we de ! bij de component die het "initiatief" neemt tot een synchronisatie. In het geval van de werkplaats plaatsen we de !'s bij de werkers en de ?'s bij de hamers: de werkers nemen het initiatief om een hamer te pakken, terwijl de hamers eigenlijk niets doen en alleen maar opgepakt worden. Figuur 7 laat een aangepast model zien van de werker, waarin synchronisatiekanalen zijn toegevoegd. Kanalen moeten altijd gedeclareerd worden. Dit kun je doen door in het window links te klikken op "Declarations" en dan de volgende tekst toe te voegen:

```
// Place global declarations here.
chan pak_hamer1, pak_hamer2, leg_hamer1_neer, leg_hamer2_neer;
```

Wanneer je het aldus aangepaste model simuleert dan zul je zien dat er "deadlocks" mogelijk zijn, toestanden waarin geen enkele transitie meer mogelijk is. Wanneer bijvoorbeeld beide werkers in de locatie `moeilijk` zijn dan willen ze beiden een `pak_hamer1!`-transitie doen, maar omdat er geen componenten is die een corresponderende `pak_hamer1?`-transitie kan doen loopt het systeem vast. Om deze deadlocks te voorkomen voegen we twee nieuwe templates `Hamer1` en `Hamer2` toe (dit kan door in het Edit-menu de optie "Insert Template" te selecteren). Figuur 8 toont de definities van



Figuur 8: Modellen van metalen en houten hamer (links resp. rechts).

deze templates.[1] Wanneer je nu de hamers ook toevoegt aan de System declarations via de regel

```
system Werker1, Werker2, Hamer1, Hamer2;
```

dan heb je als het goed is je eerste volwaardige Uppaal model geschreven! Je kunt dit model opslaan door in het File-menu de optie "Save System As..." te selecteren. Uppaal modellen worden bewaard als `.xml` bestanden.

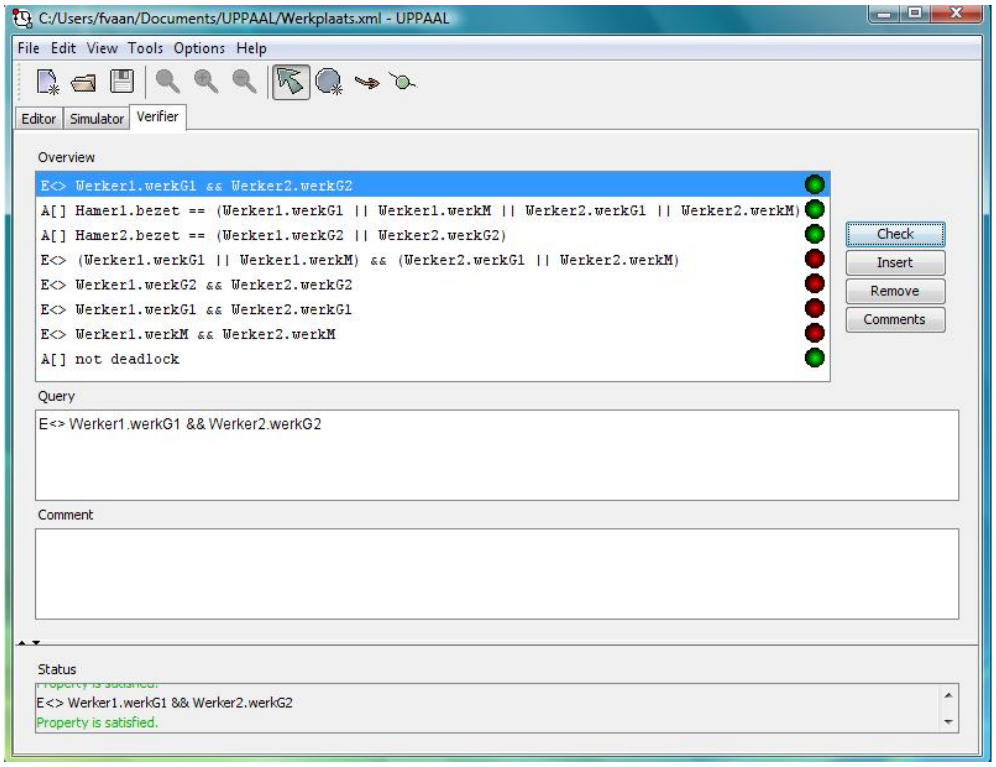
6 De Verifier

Je kunt eigenschappen van een model bewijzen met behulp van de *Verifier*. In de verifier kun je zogenaamde *queries* opschrijven, eigenschappen waarvan Uppaal kan checken of ze al dan niet gelden in een model. Figuur 9 laat een screenshot zien. Veel queries beginnen met de symbolen "`A[]`". Dit is Uppaal jargon voor "Voor alle bereikbare toestanden in het model geldt". Wanneer je bijvoorbeeld bij "Query" invult

```
A[] not deadlock
```

en vervolgens op de knop "Check" drukt, dan controleert Uppaal alle toestanden die bereikbaar zijn vanaf de begintoestand en kijkt of er in die toestanden een deadlock is (geen mogelijkheid om een transitie te doen).

[1] Eigenlijk zou het mooier zijn om maar één template `Hamer` te hebben, met twee instanties `Hamer1` en `Hamer2`, net zoals er maar één template `Werker` is met twee instanties `Werker1` en `Werker2`. Het is mogelijk om dit zo in Uppaal te definiëren, maar hiervoor moet je dan namen van kanalen als parameter meegeven aan een template. Omdat dit net wat te moelijk is om hier uit te leggen, geef ik er de voorkeur aan om twee templates `Werker1` en `Werker2` in te voeren.



Figuur 9: Screenshot van de verifier.

Voor ons model levert het checken van bovenstaande query als resultaat "Property is satisfied". Dit betekent dat het model geen deadlock bevat: in alle bereikbare toestanden kan `Werker1` of `Werker2` een stap doen. Een andere query die je kunt voorleggen aan de verifier (druk eerst op "Insert", voer onderstaande tekst in by "Query", en druk op "Check") is

```
E<> Werker1.werkM && Werker2.werkM
```

De symbolen "`E<>`" zijn Uppaal jargon voor "Er bestaat een bereikbare toestand waarin geldt". De bovenstaande query vraagt of er een toestand bestaat waarin `Werker1` en `Werker2` zich allebei in locatie `werkM` bevinden (bezig zijn met een moeilijk klus). Twee "`&`" symbolen achter elkaar is Uppaal notatie voor "en". Voor ons model levert het checken van deze query als resultaat "Property not satisfied". Dit is logisch: als een werker bezig is met een moeilijk klus dan heeft hij daarbij de ijzeren hamer nodig, en daar is er maar één van.

Wanneer we Uppaal vragen of er een bereikbare toestand is waarin de ene werker in toestand `werkG1` is en de andere in toestand `werkG2`

```
E<> Werker1.werkG1 && Werker2.werkG2
```

dan luidt het antwoord bevestigend "Property is satisfied". In dit geval kan Uppaal ook met een voorbeeld laten laten zien waarom de eigenschap geldt. Hiertoe moet je bij "Options" onder "Diagnostic Trace"de optie "Shortest" aanvinken. Wanneer je hierna bovenstaande eigenschap nog een keer checkt genereert Uppaal een trace en vraagt het programma of het deze trace mag opslaan in de simulator. Wanneer je dit toestaat kun je de trace naspelen in de simulator. In dit geval bestaat de trace uit slechts 4 transities, maar bij echte toepassingen vindt Uppaal regelmatig traces van duizenden transities. De mogelijkheid om voorbeelden te genereren is in de praktijk vaak buitengewoon nuttig. Je kunt queries opslaan door in het File-menu de optie "Save Queries As..." te selecteren. De queries worden dan bewaard als een tekstbestand met extensie `.q`. Wanneer je een Uppaal model `model.xml` opent dan wordt automatisch ook de query file `model.q` geopend (indien die bestaat).

Een aardige vraag is hoeveel globale toestanden het model van de werkplaats nu precies heeft. In eerste instantie denk je misschien $256 = 8 \times 8 \times 2 \times 2$. Immers, het model van iedere werker heeft 8 toestanden en het model van iedere hamer heeft er 2. Maar veel van deze toestanden kunnen niet voorkomen. Zo weet je in welke toestand een hamer is zodra je weet in welke toestand de beide werkers zich bevinden:

```
A[] Hamer2.bezet == (Werker1.werkG2 || Werker2.werkG2)

A[] Hamer1.bezet == (Werker1.werkG1 || Werker1.werkM
                       || Werker2.werkG1 || Werker2.werkM)
```

De eerste query zegt dat `Hamer2` bezet is precies dan wanneer hij gebruikt wordt door `Werker1` of ("`||`" in Uppaal notatie) `Werker2`. Een werker gebruikt de tweede (houten) hamer alleen als hij in locatie `werkG2` is. De tweede query zegt dat ook `Hamer1` bezet is precies dan wanneer hij gebruikt wordt door `Werker1` of `Werker2`. Een werker gebruikt de eerste (ijzeren) hamer alleen als hij in locatie `werkG1` of in locatie `werkM` is.

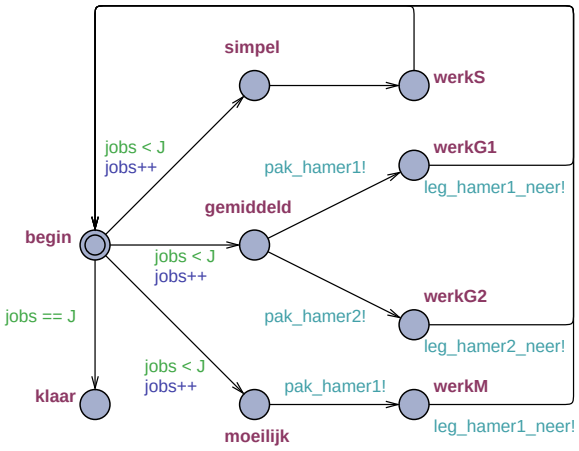
De resterende 8×8 toestanden kunnen ook niet allemaal bereikt worden: als de ene werker een hamer gebruikt kan de andere hem niet gebruiken. De toestanden `(G1,G1)`, `(G1,M)`, `(M,G1)`, `(M,M)` en `(G2, G2)` zijn derhalve onbereikbaar en dus is het totaal aantal bereikbare toestanden van de werkplaats $64 - 5 = 59$. Voor Uppaal is 59 toestanden helemaal niets: het programma rekent makkelijk met honderduizenden of zelfs miljoenen toestanden. Het is echter eenvoudig om modellen te maken die zo groot zijn dat Uppaal er niet doorheen kan rekenen omdat je computer uit zijn geheugen loopt. Wanneer je bijvoorbeeld het werkplaats model aanpast en in plaats van 2 werkers er 20 neemt, dan kan Uppaal het model beslist niet meer aan.

7 Variabelen

Aan Uppaal modellen kun je ook integer variabelen en constantes toevoegen. In transities kun je de waarde van variabelen vervolgens testen en aanpassen. Dit is vaak handig bij het modelleren van systemen en geeft Uppaal een expressieve kracht die vergelijkbaar is met programmeertalen. Ter illustratie laat ik zien hoe je het model van de werkplaats kunt aanpassen zodat werkers 10 klussen doen en dan stoppen. Voeg om te beginnen in de sectie "Declarations" de volgende regels toe:

```
const int J = 10;
int[0,J] jobs;
```

In de eerste regel declareren we een integer constante `J` met waarde 10. De waarde van een constante kan per definitie nooit veranderen. In de tweede regel wordt een integer variabele `jobs` gedeclareerd met als minimale waarde 0 en als maximale waarde J. De waarde van een variabele kan veranderen tijdens transities. In de begintoestand is de waarde van een variabele 0.



Figuur 10: Model van werker waarin precies `J` klussen worden uitgevoerd.

Integers in Uppaal zijn altijd begrensd. Wanneer je geen grenzen noemt en kortweg

```
int jobs;
```

declareert, dan is impliciet de minimale waarde van `jobs` -32768 en de maximale waarde 32768. Wanneer bij het doorlopen van de toestandsruimte een variabele een waarde krijgt toegekend buiten zijn domein, dan noemen we dit een "run time error" en wordt een foutmelding gegenereerd.

In de locatie `begin` accepteert een werker alleen een nieuwe klus als `jobs` $<$ `J`. De waarde van `jobs` wordt in dat geval met 1 verhoogd. Je kunt dit modelleren in Uppaal door de transitie van `begin` naar `simpel` te dubbelklikken en dan in het menu bij het veld "Guard" `jobs` $<$ `J` toe te voegen. Hiermee geef je aan dat de transitie van `begin` naar `simpel` alleen mag plaatsvinden indien `jobs` $<$ `J`. In het veld "Update" kun je aangeven dat de waarde van `jobs` met 1 verhoogd wordt door het nemen van deze transitie door te schrijven `jobs++`. Deze syntax is ontleend aan de programmeertaal C. Je had ook kunnen schrijven `jobs` $=$ `jobs+1` of `jobs` $:=$ `jobs` $+$ `1`, dat betekent allemaal hetzelfde. Figuur 10 laat een aangepast model zien van de werker, waarin guards en updates zijn toegevoegd aan alle uitgaande transities van locatie `begin`. Tevens is er een extra locatie `klaar`, waar de werker naar toe springt wanneer `jobs` de waarde `J` heeft en alle klussen af zijn.

In de simulator is er een apart window waarin, voor iedere toestand, de waarden van alle variabelen worden aangegeven. Met de verifier kun je vasstellen dat het model van Figuur 10 aan precies dezelfde eigenschappen voldoet als het model van Figuur 7, alleen geldt de eigenschap

```
A[] not deadlock
```

niet meer. Immers, wanneer de werkers `J` klussen hebben uitgevoerd en naar locatie `klaar` zijn gesprongen, dan zijn er geen transities meer mogelijk en is er dus een deadlock.

Merk op dat in het model de beide werkers samen `J` klussen uitvoeren. Dat komt omdat `jobs` een *globale* variabele is die door beide werkers kan worden getest en aangepast. In Uppaal kun je ook *locale* variabelen invoeren, die alleen gebruikt kunnen worden door één proces. Wanneer je in de Editor klikt op het "+" symbool links van template `Werker` dan verschijnt er een regel "Declarations" onder `Werker`. Wanneer je daar op klikt kun je de locale declaraties invoeren voor het template. Door nu bijvoorbeeld de regel

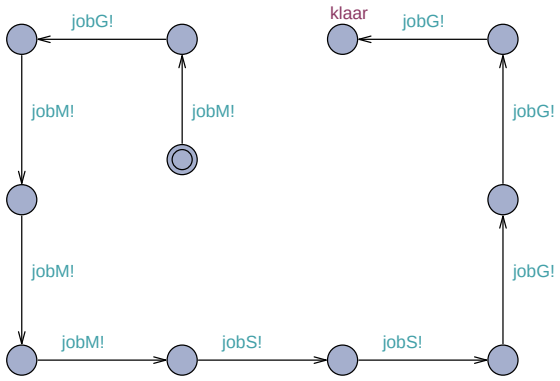
```
int[0,J] jobs;
```

te verplaatsen van de globale declarations sectie naar de locale declarations sectie van template `Werker`, geef je `Werker1` en `Werker2` ieder een eigen, locale copie van variabele `jobs`. Het gevolg is dat `Werker1` en `Werker2` nu ieder afzonderlijk `J` klussen uitvoeren, in plaats van `J` klussen samen.

Uppaal heeft een vrij uitgebreide syntax voor de expressies in guards en updates. Het is zelfs mogelijk om nieuwe datatypes in te voeren zoals arrays en Booleans, en om willekeurige functies te definiëren. De syntax komt grotendeels overeen met de syntax van de programmeertalen C, C++ en Java. Een overzicht van de syntax kun je vinden in het Help menu van Uppaal, door te klikken op "Language Reference" en dan op "Expressions".

8 Tijd en Klokken

Wanneer je modellen maakt van systemen dan speelt tijd daarbij dikwijls een rol. Je wilt bijvoorbeeld niet alleen weten dát een bepaalde toestand bereikbaar is, maar ook binnen hoeveel tijd. Zo zou de volgende vraag zich kunnen aandienen bij de werkplaats. Stel dat een werker 5sec nodig heeft voor een simpele klus, 10sec voor een gemiddelde klus met gebruik van de metalen hamer, 15sec voor een gemiddelde klus met gebruik van de houten hamer, en 20sec voor een moeilijke klus. Hoeveel tijd hebben de

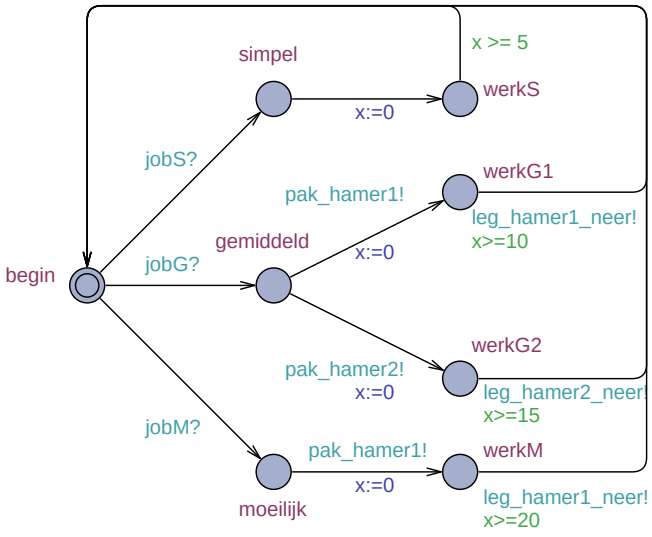


Figuur 11: Model van band die 10 klussen aflevert.

beide werkers samen minimaal nodig voor het afwerken van 10 klussen met moeilijkheidsgraad respectievelijk M, G, M, M, M, S, S, G, G en G?

Uppaal is min of meer ontworpen om dit soort vragen te beantwoorden. Om de binnenkomende klussen te modelleren declareren we drie nieuwe kanalen `jobS`, `jobG` en `jobM`. De template (en tevens automaat) `Band` die staat afgebeeld in Figuur 11 beschrijft het gedrag van een lopende band die de 10 klussen in de genoemde volgorde aanlevert. Door `Band` toe te voegen aan het model bij "System declarations" en de drie uitgaande transities van locatie `begin` in de automaat van Figuur 7 op voor de hand liggende wijze te labelen met `jobS`?, `jobG`? en `jobM`?, hebben we precies gemodelleerd hoe de werkers achtereenvolgens 10 klussen op hun bord krijgen.

In de modellen die we tot nu toe gezien hebben is tijd niet expliciet gemodelleerd. We nemen wel altijd aan dat transities instantaan plaatsvinden en geen tijd kosten. Tijd verstrijkt alleen wanneer een automaat wacht in een locatie. Wanneer we onder- of bovengrenzen willen aangeven voor de tijd dat een automaat in een locatie verblijft, dan kunnen we dit doen met behulp van zogenaamde *klokken*. Een klok is een speciaal soort variabele waarvan de waarde een reëel getal is. In de begintoestand hebben alle klokken waarde 0. Wanneer een automaat wacht in een locatie en de tijd verstrijkt dan neemt de waarde van klokken toe. Meer precies: wanneer er t tijdseenheden verstrijken dan neemt de waarde van alle klokken met t toe. In de werkelijkheid lopen klokken natuurlijk nooit helemaal precies, maar in de wereld van modellen maken we dankbaar gebruik van perfecte klokken om daarmee aan te geven hoeveel tijd er precies kan verstrijken tussen transities. Figuur 12 laat een model van de werker zien dat is uitgebreid met



Figuur 12: Model van werker uitgebreid met een klok.

de nieuwe synchronisaties `jobS`?, `jobG`? en `jobM`?, maar tevens met een klok `x`. Deze klok is gedeclareerd door bij de "Declarations" sectie van template `Werker` de volgende regel toe te voegen:

```
// Place local declarations here.
clock x;
```

Wanneer een werker van locatie `simpel` naar locatie `werkS` springt, en dus begint te werken aan een simpele job, dan wordt met een update `x := 0` klok `x` op 0 gezet. Vervolgens test de guard van de uitgaande transitie van `werkS` naar `begin` dat `x >= 5`. Hiermee dwingen we af dat deze transitie pas kan plaatsvinden als de automaat minimaal 5 tijdseenheden in locatie `werkS` is geweest. Immers, een werker heeft 5 tijdseenheden nodig voor het afhandelen van een simpele job. Op vergelijkbare manier dwingen we af dat de automaat minimaal 10, 15 en 20 tijdseenheden verblijft in locaties `werkG1`, `werkG2` en `werkM`.

In het aldus verkregen model hebben `Werker1` en `Werker2` allebei een locale klok `x` die registreert hoe lang ze al bezig zijn met een bepaalde job. Steeds wanneer een werker begint met een nieuwe job dan wordt zijn klok op 0 gezet. Om bij te houden hoeveel tijd er in het totaal verstreken is voeren we ook een globale klok `tijd` in. De globale declaraties zien er dan alsvolgt uit:

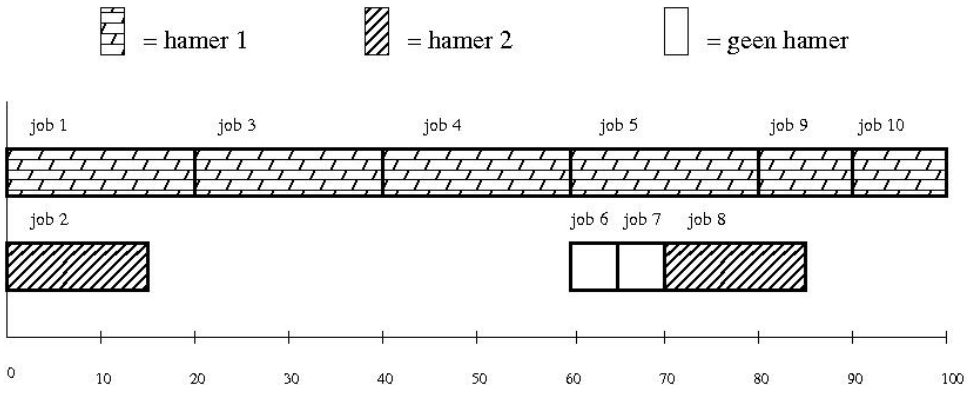
```
// Place global declarations here.
chan jobS, jobG, jobM, pak_hamer1, pak_hamer2,
    leg_hamer1_neer, leg_hamer2_neer;
clock tijd;
```

We kunnen Uppaal nu vragen of er een toestand is waarin al het werk gedaan is, de band leeg is en de beide werkers terug zijn in hun begintoestand:

```
E<> Band.klaar && Werker1.begin && Werker2.begin
```

Wanneer Uppaal een diagnostische trace levert die laat zien dat deze eigenschap geldt, dan is dit in het algemeen niet de kortste trace. Immers, we hebben in ons model alleen maar ondergrenzen gegeven aan de timing en geen bovengrenzen. Zo kan de band er willekeurig lang over doen voordat hij een job aflevert, een werker kan willekeurig lang treuzelen voordat hij er mee begint, en vervolgens kan hij er ook nog eens willekeurig lang over doen om de job af te maken. Echter, door bij "Options" onder "Diagnostic Trace"de optie "Fastest" aan te vinken zoekt Uppaal de snelste executie. In de simulator kun je vervolgens zien dat in de eindtoestand van deze executie `tijd >= 100`. Dit betekent dat de beide werkers in 100 tijdseenheden de 10 jobs kunnen afhandelen. De makkelijkste manier om het door Uppaal gevonden schema te begrijpen is met behulp van de zogenaamde "message sequence chart" in het window rechts onder in de simulator. Figuur 13 visualiseert het snelste schema dat Uppaal gevonden heeft nog wat compacter. In dit schema is één werker permanent in de weer met de ijzeren hamer (hamer 1) terwijl de andere werker aan het fliereFluiten is en af en toe een klusje doet waarbij hij soms de houten hamer (hamer 2) gebruikt. Kun je een schema bedenken waarbij de totale tijd gelijk blijft en de werklast eerlijk is verdeeld?

Zoals we hebben gezien kunnen *ondergrenzen* aan timing worden beschreven met behulp van guards. Zo geeft een guard `x >= 5` bij de transitie van `werkS` naar `begin` aan dat deze transitie pas genomen mag worden als de werker minimaal 5 tijdseenheden in locatie `werkS` is. Met Uppaal kunnen we ook *bovengrenzen* geven aan timing: dit doen we met behulp van zogenaamde *invarianten*. Wanneer je de locatie `werkS` dubbelklikt verschijnt er een venster met een veld "Invariant". Wanneer je daar nu `x <= 7` invult dan geef je daarmee aan dat in deze locatie de waarde van `x` altijd kleiner of gelijk 7 zal zijn. Met andere woorden, de werker heeft maximaal 7 tijdseenheden nodig voor een simpele klus! Wanneer er meer dan 7 tijdseenheden verstreken zijn dan heeft hij locatie `werkS` verlaten. In Figuur 14 heb ik



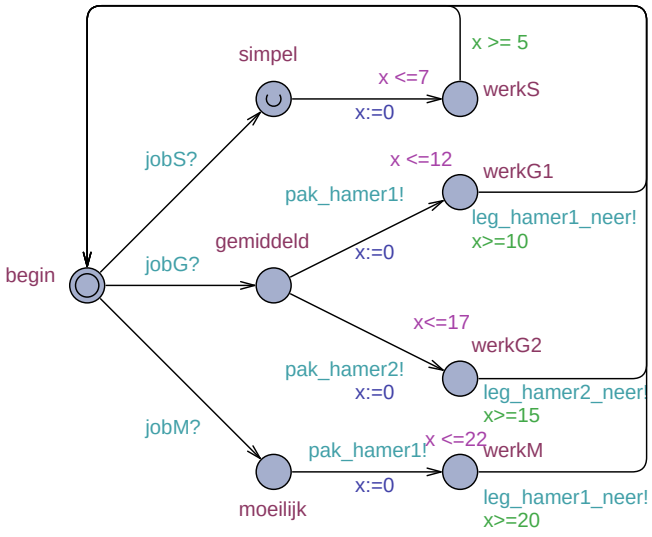
Figuur 13: Het snelste schema voor het uitvoeren van de 10 klussen.

bovengrenzen van 7, 12, 17 en 22 toegevoegd voor locaties `werkS`, `werkG1`, `werkG2` en `werkM`. Tevens heb ik een bovengrens van 0 toegevoegd voor locatie `simpel`: als een werker een simpele klus heeft gepakt moet hij er direct mee aan de slag. Ik had deze bovengrens kunnen opleggen door klok `x` op nul te zetten bij aankomst in `simpel` en dan met een invariant `x <= 0` af te dwingen dat de werker deze locatie direct weer verlaat. Maar het kan eenvoudiger, namelijk door de locatie `simpel` te dubbelklikken en het veld "urgent" aan te vinken. Wanneer een locatie *urgent* is dan betekent dat dat tijd daar niet kan verstrijken.

Na al deze aanpassingen van het model is er nog steeds geen bovengrens aan de tijd die nodig is om alle klussen af te krijgen. Immers, de werkers kunnen willekeurig lang wachten met het van de band pakken van een klus, en ze kunnen willekeurig lang wachten met het pakken van gereedschap. Maar hier kunnen we wel wat aan doen. We kunnen de synchronisaties voor het pakken van een klus of een stuk gereedschap *urgent* maken. Dit betekent dat zodra zo'n synchronisatie mogelijk is er direct een transitie moet plaatsvinden en de processen niet langer mogen wachten. In Uppaal kunnen we dit beschrijven door de declaraties van de kanalen alsvolgt aan te passen:

```
urgent chan jobS, jobG, jobM, pak_hamer1, pak_hamer2;
chan   leg_hamer1_neer, leg_hamer2_neer;
```

Uppaal heeft geen aparte optie om direct de langzaamste executie naar een toestand uit te rekenen. Maar met een paar opeenvolgende queries kun je die executie toch berekenen. De volgende eigenschap geldt volgens Uppaal:



Figuur 14: Model van werker uitgebreid met bovengrenzen aan timing.

```
A[] tijd >= 200 imply (Band.klaar && Werker1.begin && Werker2.begin)
```

Dit betekent dat we er zeker van kunnen zijn dat na 200sec alle klussen af zijn. Er geldt zelfs

```
A[] tijd >= 150 imply (Band.klaar && Werker1.begin && Werker2.begin)
```

maar niet

```
A[] tijd >= 110 imply (Band.klaar && Werker1.begin && Werker2.begin)
```

De werkers hebben dus maximaal tussen de 110sec en 150sec nodig voor het afronden van alle klussen. Door het interval steeds verder te verkleinen kun je er achter komen dat de volgende eigenschap geldt:

```
A[] tijd >= 127 imply (Band.klaar && Werker1.begin && Werker2.begin)
```

maar deze eigenschap niet:

```
A[] tijd >= 126 imply (Band.klaar && Werker1.begin && Werker2.begin)
```

Het tegenvoorbeeld voor de laatste eigenschap geeft het traagst mogelijke schema dat precies 126 tijdseenheden duurt. In dit schema ontbreken alleen de laatste transities van de werkers terug naar de begintoestand, maar die kosten geen tijd.

Referenties

[1] G. Behrmann, A. David, and K.G. Larsen. A tutorial on Uppaal. In M. Bernardo and F. Corradini, editors, *Formal Methods for the Design of Real-Time Systems, International School on Formal Methods for the Design of Computer, Communication and Software Systems, SFM-RT 2004, Bertinoro, Italy, September 13-18, 2004, Revised Lectures*, volume 3185 of *Lecture Notes in Computer Science*, pages 200–236. Springer, 2004.

[2] R. Milner. *Communication and Concurrency*. Prentice-Hall International, Englewood Cliffs, 1989.