The Game of Nim in Z and STATECRUNCHER

Graham G. Thomason

Appendix to the Thesis "The Design and Construction of a State Machine System that Handles Nondeterminism"

Department of Computing
School of Electronics and Physical Sciences
University of Surrey
Guildford, Surrey GU2 7XH, UK

July 2004

Summary

In this paper we show how a system taken from the Z literature can be modelled in STATECRUNCHER. An understanding of STATECRUNCHER is assumed, but for the purposes of this paper, most of STATECRUNCHER 's functionality will not seem strange to anyone familiar with UML dynamic modelling [UML], since that is the basis of the language.

We take a fairly easy example that nevertheless illustrates the essence of Z: the *Game of Nim* as described by McMorran and Powell [McMorran p.224]. This example covers a relation (a total function) and ó and ô operations on schemas.

We are not concerned with a strategy for winning, though a simple one exists[1]. We are concerned with specifying how the game is played and when a player has won.

STATECRUNCHER has been built for the purposes of providing an oracle to state-based tests. It has sufficient expressive power to capture the game of Nim in its entirety in a fairly intuitive way.

[1] When there is one pile left, the only winning position (i.e. after the winning player's turn) is when there is just one stick in the pile. If there are two or three piles left, winning positions are determined as follows. Express the number of sticks in each pile in binary. Add these binary numbers in column-by-column modulo-2 arithmetic (so there is no carry from one column to another). If the result is zero, the position is a winning one. For the starting position (5, 6, and 7 sticks), the modulo-2-sum is 101+110+111 = 100. So by taking 4 sticks from any pile, a winning position is obtained.

Table of Contents

1. Nim in Z

The description of the exercise given in [McMorran, p.118] is:

> The game of Nim is played by two people. The game starts with three piles containing five, six and seven sticks respectively. Each player plays alternately. On each turn, the player removes some sticks from *one* pile. The loser is the player who removes the last stick (the other player is the winner).

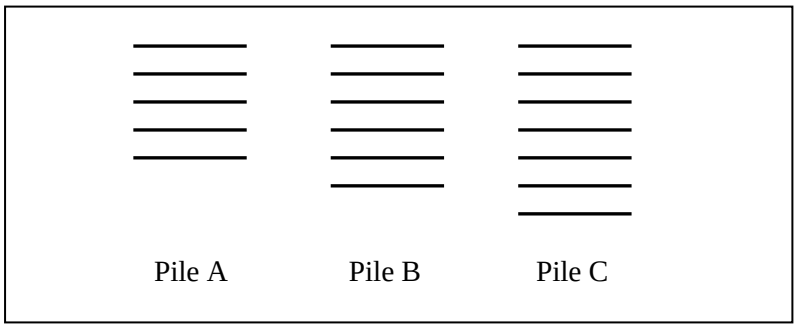

Figure 1. Nim - the starting position

We are asked to write a formal specification of the game state and a Play schema. We must distinguish between

- Game Ended
- Game Continues
- Illegal Play

A Nim specification in Z along the lines of [McMorran] follows, but we add the notion of players necessarily taking turns. The players are John and Mary. The player is not supplied as a parameter, but any move is attributed to the player whose turn it is when making the move. The additions in the specification below with respect to [McMorran] are in marked by a double line in the margin. The maker of the Z font used is indicated in reference [Z font].

Reminder of some less common terminology used:

- The *range* of a relation `R` is denoted by `ran R`.
- The *range restriction* relation `RÃS` is the subset of `R` where the range is restricted to `S`.

We will call the piles `A`, `B` and `C`.

```
pileid::= A | B | C
```

We will call the players `John` and `Mary`.

```
player::=John | Mary
```

The `Game` state is a mapping from `pileid` to the number of sticks in that pile, and a mapping from the `player` to a truth value of whether it is their turn or not. It is only one player's turn at any time.

```
ùýGameýýýýýýýýýýýýýýýýýýýýýýýýý
ú pile : pileidª£
ú turn : playerª{true,false}
ûýýýýýýýýýýýýýýýýýýýýýýýýýýýýý
ú #(turnÃ{true}) = 1
üýýýýýýýýýýýýýýýýýýýýýýýýýýýýý
```

A player may make a valid or invalid move (requesting too many sticks). A valid move will complete the game or leave sticks still available.

```
code::=ok | error | fin
```

The input values for `Play` are a pile identity, `p?`, and the number of sticks the player wishes to take, `take?`. A return code, `rc!`, shows the result.

```
ùýParametersýýýýýýýýýýýýýýýýýýýýý
ú p? : pileid
ú take? : £
ú rc! : code
üýýýýýýýýýýýýýýýýýýýýýýýýýýýýýý
```

The play is permitted if there are enough sticks.

```
ùýPlayOKýýýýýýýýýýýýýýýýýýýýýýýýýýýýýýýýýýýýý
ú óGame
ú Parameters
ûýýýýýýýýýýýýýýýýý
ú pile p? ä take?
ú pile' = pile É {p? Ş (pile p? - take?)}
ú Õp:player × turn' p = Ø(turn p)
ú rc! = ok
üýýýýýýýýýýýýýýýýýýýýýýýýýýýýýýýýýýýýýýýýýýý
```

The set of piles is updated by decrementing the count for `p?` by `take?`.

The turn mapping is updated by negating the truth value associated with each player. For the new `turn` mapping, we could have negated each player's turn explicitly

```
   turn' =   turn É {John Ş Ø(turn John)} É {Mary Ş Ø(turn Mary)}
```

The play is prohibited if there are too few sticks:

```
ùýPlayErrýýýýýýýýýýýýýýýýýýýýýýýýý
```



```
ú ôGame
ú Parameters
ûýýýýýýýýýýýýýýý
ú pile p? < take?
ú rc! = error
üýýýýýýýýýýýýýýýýýýýýýýýýýýýýýý
```

The game is complete when all piles are empty:
```
Ended á [Game | ran pile = {0}]
```

Any intermediate state, (that is, where there are sticks on the table) we will call `Open`:
```
Open á [Game | ran pile µ {0}]
```

A play that leaves the game in an `Open` state can be described thus:
```
PlayMore á [PlayOK | Open' Ù rc! =ok]
```

A play that ends the game can be described thus
```
PlayLast á [PlayOK | Ended' Ù rc! =fin]
```

We can now describe a play
```
Play á PlayMore Ú PlayLast Ú PlayErr
```

2. Nim in STATECRUNCHER

Figure 2 shows how Nim can be modelled in STATECRUNCHER.

Following the figure, a description of the model is given, then a session running the model is reproduced.

The following appendix is recommended reading prior to studying the output produced by the STATECRUNCHER models:

- *A Quick Reference of STATECRUNCHER's Output Format* [StCrOutput]

The source code of the model given at the end of this paper. It corresponds to the figure in almost every detail.

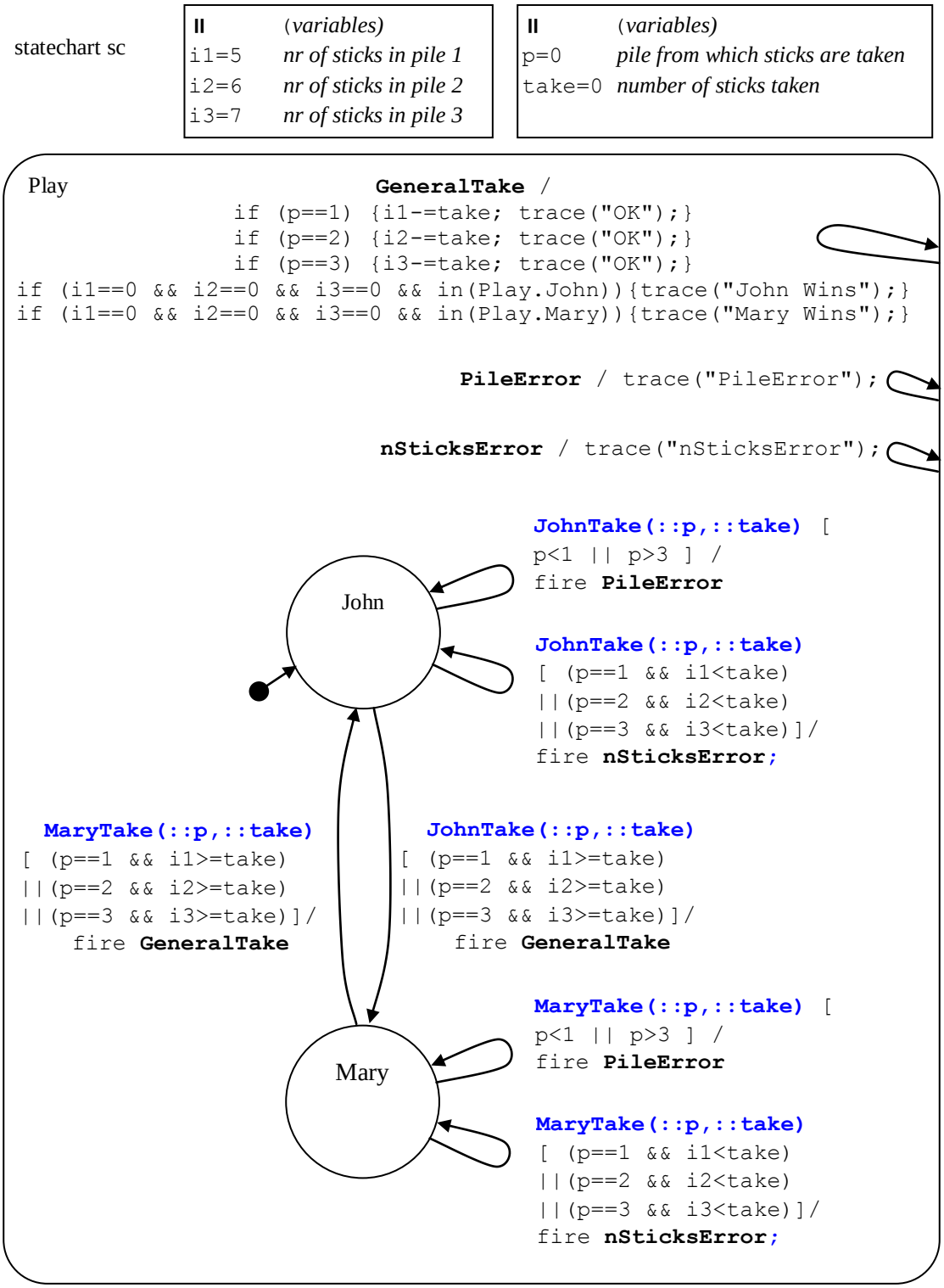

Figure 2. Nim [model t4320]

A description of the STATECRUNCHER model, with the relationship to the Z specification

The piles are held in variables `i1`, `i2`, `i3` respectively (cf. the Z `pileid::= A | B | C`).

The player whose turn it is, is held by a leafstate `John` or `Mary` being *occupied* (with the other one being *vacant*). The cluster (OR state) construction ensures that only one state is occupied. In Z this was `#{pl : player | turn player}=1`.

The `ó` operations in `PlayOK` correspond to transitions *between* states `John` and `Mary`. These always involve a legal number of sticks being taken. The events triggering the transitions are `JohnTake` and `MaryTake`, with parameters that are stored in `p` and `take`, corresponding to `p?` and `take?` in the Z specification.

The `ô` operations in `PlayErr` correspond to self transitions *on* states `John` and `Mary`. These are error moves which result in a STATECRUNCHER trace to this effect, with no transitions between states. The error code of the Z specification

```
code::=ok | error | fin
```

is reflected in the most recent STATECRUNCHER TRACE which can be:

`OK`, `nSticksError`, `PileError`, `JohnWins` or `MaryWins`.

The self-transitions on the internal events **`GeneralTake`**, **`PileError`** and **`nSticksError`** are the equivalent of a subroutine of imperative languages such as C. They execute the mechanics of a move that could come from two places, with either John or Mary initiating them.

The game ends when the STATECRUNCHER TRACE indicates this by giving the winner - no more events should be given - the model is only valid up to this point. (Any more events are traced as being in error if attempted. We could have disabled such events by introducing a new state `Ended` and transitioning to it as an additional action to tracing the winner). If the trace does not indicate a winner, the game continues. We can also see from the pile values i1, i2 and i3 whether the game has ended. Comparing with the Z specification, we have

In the Z specification, the game is complete when all piles are empty:

```
Ended á [Game | ran pile = {0}]
```

In the STATECRUNCHER model, the game is complete when

```
i1==0 && i2==0 && i3==0
```

or when the most recent TRACE is `JohnWins` or `MaryWins`.

In the Z specification, any intermediate state is called `Open`:

Open á [Game | ran pile µ {0}]

In the STATECRUNCHER model, an `Open` state is seen by

`i1!=0 || i2!=0 || i3!=0`

or when the most recent TRACE is not `JohnWins` and not `MaryWins`.

In the Z specification, a play that leaves the game in an `Open` state is:

PlayMore á [PlayOK | Open' Ù rc! =ok]

In the STATECRUNCHER model, this corresponds to, for example

event `MaryTake` or `JohnTake` *(according to which is transitionable)*

after which

`i1!=0 || i2!=0 || i3!=0`

and (last `TRACE`) = `OK` *(but even disallowed moves leave the game open)*

In the Z specification, a play that ends the game can be described thus

PlayLast á [PlayOK | Ended' Ù rc!=fin]

In the STATECRUNCHER model, this corresponds to, for example

event `MaryTake` or `JohnTake` *(according to which is transitionable)*

after which

`i1==0 && i2==0 && i3==0`

and (last `TRACE`) = `MaryWins` or `JohnWins`

In the Z specification, we describe a play as

`PlayMore á PlayMore Ú PlayLast Ú PlayErr`

In the STATECRUNCHER model, this is just

event `MaryTake` or `JohnTake` *(according to which is transitionable)*

provided the game has not ended, which is as far as the model is valid.

Session with Nim `[model t4320]`

Only essential explanations of STATECRUNCHER's output are given here. For more detail, refer to [StCrOutput].

Transitionable events are given by `TREV` lines. The only events that can be supplied from an external perspective are those at PCO `[external,sc]]`. This will give just one event from the set {`JohnTake, MaryTake`} at any stage of playing the game. An event is supplied for processing in this model by a command

`pe` *event* `p=[`*param1*`,`*param2*`]`

where *event* is `JohnTake` or `MaryTake` and *param1* is the pile (the `p?` of Z) and param2 is the number of sticks to take (the `take?` of Z).

The player whose turn it is, is evident from the transitionable event offered, but it can also be seen from the occupied leafstate, in the `leafstate John` and `leafstate Mary` lines (`OCC`=occupied, `VAC`=vacant).

The number of sticks per pile is seen in the `VAR INTEGER i1/i2/i3` lines.

The move status is indicated by the `TRACE` lines. The TRACE is read from right to left.

Further notes on the output, but which are not essential to understanding the game play are:

- terms in many lines such as `[sc]` and `[Play,sc]` give the scope (i.e. position in the statechart hierarchy) of an item. Events can be supplied without scope - in that case statechart scope is assumed.
- the TREV lines contain event names (with scope), then the number of parameters, then the ranges of parameters then the PCO of the event.

The session shows moves being made, including disallowed moves involving a disallowed pile of a disallowed number of sticks. The events supplied, and error codes just produced in the TRACE, are shown in **bold font**.

```
SC:|: run t4320
...
SC:|: gc
2    statechart sc
2       cluster Play [sc] = OCC []  **
2          leafstate John [Play,sc] = OCC []  **
2          leafstate Mary [Play,sc] = VAC []
2    VAR  INTEGER i1 [sc] =5
2    VAR  INTEGER i2 [sc] =6
2    VAR  INTEGER i3 [sc] =7
2    VAR  INTEGER p [sc] =0
2    VAR  INTEGER take [sc] =0
2    TRACE =[]
2    TREV [[JohnTake,[sc]],2,[[r,0,3],[r,0,7]],[external,[sc]]]
2    TREV [[PileError,[sc]],0,[],[internal,[sc]]]
2    TREV [[nSticksError,[sc]],0,[],[internal,[sc]]]
2    TREV [[GeneralTake,[sc]],0,[],[internal,[sc]]]

outworlds=[2]
number of outworlds=1
```



```
SC:|: pe JohnTake p=[2,4]
SC:|: gc
6    statechart sc
6       cluster Play [sc] = OCC []  **
6          leafstate John [Play,sc] = VAC []
6          leafstate Mary [Play,sc] = OCC []  **
6    VAR  INTEGER i1 [sc] =5
6    VAR  INTEGER i2 [sc] =2
6    VAR  INTEGER i3 [sc] =7
6    VAR  INTEGER p [sc] =2
6    VAR  INTEGER take [sc] =4
6    TRACE =[OK]
6    TREV [[MaryTake,[sc]],2,[[r,0,3],[r,0,7]],[external,[sc]]]
6    TREV [[PileError,[sc]],0,[],[internal,[sc]]]
6    TREV [[nSticksError,[sc]],0,[],[internal,[sc]]]
6    TREV [[GeneralTake,[sc]],0,[],[internal,[sc]]]

outworlds=[6]
number of outworlds=1
SC:|: pe MaryTake p=[2,3]
SC:|: gc
9    statechart sc
9       cluster Play [sc] = OCC []  **
9          leafstate John [Play,sc] = VAC []
9          leafstate Mary [Play,sc] = OCC []  **
9    VAR  INTEGER i1 [sc] =5
9    VAR  INTEGER i2 [sc] =2
9    VAR  INTEGER i3 [sc] =7
9    VAR  INTEGER p [sc] =2
9    VAR  INTEGER take [sc] =3
9    TRACE =[nSticksError,OK]
9    TREV [[MaryTake,[sc]],2,[[r,0,3],[r,0,7]],[external,[sc]]]
9    TREV [[PileError,[sc]],0,[],[internal,[sc]]]
9    TREV [[nSticksError,[sc]],0,[],[internal,[sc]]]
9    TREV [[GeneralTake,[sc]],0,[],[internal,[sc]]]

outworlds=[9]
number of outworlds=1
SC:|: pe MaryTake p=[2,2]
SC:|: gc
13   statechart sc
13      cluster Play [sc] = OCC []  **
13         leafstate John [Play,sc] = OCC []  **
13         leafstate Mary [Play,sc] = VAC []
13   VAR  INTEGER i1 [sc] =5
13   VAR  INTEGER i2 [sc] =0
13   VAR  INTEGER i3 [sc] =7
13   VAR  INTEGER p [sc] =2
13   VAR  INTEGER take [sc] =2
13   TRACE =[OK,nSticksError,OK]
13   TREV [[JohnTake,[sc]],2,[[r,0,3],[r,0,7]],[external,[sc]]]
13   TREV [[PileError,[sc]],0,[],[internal,[sc]]]
13   TREV [[nSticksError,[sc]],0,[],[internal,[sc]]]
13   TREV [[GeneralTake,[sc]],0,[],[internal,[sc]]]

outworlds=[13]
number of outworlds=1
SC:|: pe JohnTake p=[3,7]
SC:|: gc
17   statechart sc
17      cluster Play [sc] = OCC []  **
17         leafstate John [Play,sc] = VAC []
17         leafstate Mary [Play,sc] = OCC []  **
17   VAR  INTEGER i1 [sc] =5
17   VAR  INTEGER i2 [sc] =0
17   VAR  INTEGER i3 [sc] =0
17   VAR  INTEGER p [sc] =3
17   VAR  INTEGER take [sc] =7
```

```
17   TRACE =[OK,OK,nSticksError,OK]
17   TREV [[MaryTake,[sc]],2,[[r,0,3],[r,0,7]],[external,[sc]]]
17   TREV [[PileError,[sc]],0,[],[internal,[sc]]]
17   TREV [[nSticksError,[sc]],0,[],[internal,[sc]]]
17   TREV [[GeneralTake,[sc]],0,[],[internal,[sc]]]

outworlds=[17]
number of outworlds=1
SC:|: pe MaryTake p=[4,1]
SC:|: gc
20   statechart sc
20      cluster Play [sc] = OCC []  **
20         leafstate John [Play,sc] = VAC []
20         leafstate Mary [Play,sc] = OCC []  **
20   VAR  INTEGER i1 [sc] =5
20   VAR  INTEGER i2 [sc] =0
20   VAR  INTEGER i3 [sc] =0
20   VAR  INTEGER p [sc] =3
20   VAR  INTEGER take [sc] =7
20   TRACE =[nSticksError,OK,OK,nSticksError,OK]
20   TREV [[MaryTake,[sc]],2,[[r,0,3],[r,0,7]],[external,[sc]]]
20   TREV [[PileError,[sc]],0,[],[internal,[sc]]]
20   TREV [[nSticksError,[sc]],0,[],[internal,[sc]]]
20   TREV [[GeneralTake,[sc]],0,[],[internal,[sc]]]

outworlds=[20]
number of outworlds=1
SC:|: pe MaryTake p=[1,4]
SC:|: gc
24   statechart sc
24      cluster Play [sc] = OCC []  **
24         leafstate John [Play,sc] = OCC []  **
24         leafstate Mary [Play,sc] = VAC []
24   VAR  INTEGER i1 [sc] =1
24   VAR  INTEGER i2 [sc] =0
24   VAR  INTEGER i3 [sc] =0
24   VAR  INTEGER p [sc] =1
24   VAR  INTEGER take [sc] =4
24   TRACE =[OK,nSticksError,OK,OK,nSticksError,OK]
24   TREV [[JohnTake,[sc]],2,[[r,0,3],[r,0,7]],[external,[sc]]]
24   TREV [[PileError,[sc]],0,[],[internal,[sc]]]
24   TREV [[nSticksError,[sc]],0,[],[internal,[sc]]]
24   TREV [[GeneralTake,[sc]],0,[],[internal,[sc]]]

outworlds=[24]
number of outworlds=1
SC:|: pe JohnTake p=[1,1]
SC:|: gc
29   statechart sc
29      cluster Play [sc] = OCC []  **
29         leafstate John [Play,sc] = VAC []
29         leafstate Mary [Play,sc] = OCC []  **
29   VAR  INTEGER i1 [sc] =0
29   VAR  INTEGER i2 [sc] =0
29   VAR  INTEGER i3 [sc] =0
29   VAR  INTEGER p [sc] =1
29   VAR  INTEGER take [sc] =1
29   TRACE =[Mary Wins,OK,OK,nSticksError,OK,OK,nSticksError,OK]
29   TREV [[MaryTake,[sc]],2,[[r,0,3],[r,0,7]],[external,[sc]]]
29   TREV [[PileError,[sc]],0,[],[internal,[sc]]]
29   TREV [[nSticksError,[sc]],0,[],[internal,[sc]]]
29   TREV [[GeneralTake,[sc]],0,[],[internal,[sc]]]

outworlds=[29]
number of outworlds=1
SC:|:
```


3. Source listing of the STATECRUNCHER model

```
//-------------------------------------------------------------------------------
// Module:    Nim.scs.txt
// Author:    Graham Thomason, Philips Digital Systems Laboratories, Redhill
// Date:      11 July, 2003
// Purpose:   StateCruncher model: The Game of Nim (McMorran & Powell "Z.." p118,224)
//
// Copyright (C) 2003 Philips Electronics N.V.
//
// Revision History:
//
//-------1---------2---------3---------4---------5---------6---------7---------8-----

statechart sc(Play)

PCO internal,external;

event JohnTake,MaryTake@external;
event GeneralTake,PileError,nSticksError@internal;

enum int3 {0,..,3};
enum int7 {0,..,7};

int7 i1=5, i2=6, i3=7;     // Sticks remaining on each pile
int3 p=0;                  // Pile from which sticks are taken
int7 take=0;               // Number of sticks taken from pile

  cluster Play(John,Mary) {                                                    \
       /* If Mary took the last stick, we are now in John, and John wins */    \
       PileError    { trace("PileError");    };                               \
       nSticksError { trace("nSticksError"); };                               \
       GeneralTake                                                            \
        { if (p==1)  {i1-=take; trace("OK"); }                                \
          if (p==2)  {i2-=take; trace("OK"); }                                \
          if (p==3)  {i3-=take; trace("OK"); }                                \
          if (i1==0 && i2==0 && i3==0 && in(Play.John)) {trace("John Wins");}  \
          if (i1==0 && i2==0 && i3==0 && in(Play.Mary)) {trace("Mary Wins");}  \
      };}

     // The occupied cluster state indicates whose turn it is

     state John {JohnTake(::p,::take)                                          \
                 [(p==1 && i1>=take)||(p==2 && i2>=take)||(p==3 && i3>=take)]   \
                 -> Mary                                                       \
                 {fire GeneralTake;   };                                       \
                                                                               \
                JohnTake(::p,::take)         /*internal transition */         \
                  [ p<1 || p>3 ]             {fire PileError;}  ;              \
                                                                               \
                JohnTake(::p,::take)         /*internal transition */         \
                  [(p==1 && i1<take)||(p==2 && i2<take)||(p==3 && i3<take) ]    \
                     {fire nSticksError;}  ;                                   }
```

```
     state Mary {MaryTake(::p,::take)                                                \
                 [(p==1 && i1>=take)||(p==2 && i2>=take)||(p==3 && i3>=take)]     \
                 -> John                                                         \
                 {fire GeneralTake;   };                                         \
                                                                                 \
                MaryTake(::p,::take)         /*internal transition */           \
                  [ p<1 || p>3 ]             {fire PileError;}  ;                \
                                                                                 \
                MaryTake(::p,::take)         /*internal transition */           \
                  [(p==1 && i1<take)||(p==2 && i2<take)||(p==3 && i3<take)]      \
                     {fire nSticksError;}  ;                                     }

//------------------------[end of module]-------------------------------------------
```


4. References

STATECRUNCHER documentation and papers by the present author

Main Thesis	[StCrMain]	The Design and Construction of a State Machine System that Handles Nondeterminism

Appendices

Appendix 1	[StCrContext]	Software Testing in Context
Appendix 2	[StCrSemComp]	A Semantic Comparison of STATECRUNCHER and Process Algebras
Appendix 3	[StCrOutput]	A Quick Reference of STATECRUNCHER's Output Format
Appendix 4	[StCrDistArb]	Distributed Arbiter Modelling in CCS and STATECRUNCHER - A Comparison
Appendix 5	[StCrNim]	The Game of Nim in Z and STATECRUNCHER
Appendix 6	[StCrBiblRef]	Bibliography and References

Related reports

Related report 1	[StCrPrimer]	STATECRUNCHER-to-Primer Protocol
Related report 2	[StCrManual]	STATECRUNCHER User Manual
Related report 3	[StCrGP4]	GP4 - The Generic Prolog Parsing and Prototyping Package *(underlies the STATECRUNCHER compiler)*
Related report 4	[StCrParsing]	STATECRUNCHER Parsing
Related report 5	[StCrTest]	STATECRUNCHER Test Models
Related report 6	[StCrFunMod]	State-based Modelling of Functions and Pump Engines

References

[McMorran] Mike McMorran and Steve Powell
Z Guide for Beginners
Blackwell Scientific Publications, 1993. ISBN 0-632-03117-4

[Hayes] Ian Hayes (editor)
Specification Case Studies
Prentice Hall 1987, ISBN 0-13-826579-8

[UML] The Object Management Group website is: http://www.omg.org
UML specifications are available from this website.

[Z font] Shareware by Lubos Mikusiak, lmikusia@ingr.com, available from the site http://www.informatikforum.org

