Formele specificatie van Groep 6
OGO 2.2

Hugo Snel
Jules Wulms
Marc van Meel
Saskia van Doormalen
Benjamin van der Hoeven
Frederique Gerritsen
Maikel Heetveld

12 maart 2012

1 Inleiding

Dit verslag is een uitwerking van de formele specificatie van de informele specificatie, zoals te lezen in *"Report on OGO 2.2 Software specification Assignment 2a"*, beschreven door OGO 2.2 groep 6, 13 februari 2012.

We beginnen met te definiëren hoe het spel eruit zit in termen van zijn individuele componenten (Player, Board, Controller). Daarna wordt ingegaan op hoe deze tot uitdrukking komen in de buitenwereld middels het View-component.

Ten slotte worden de mogelijke acties van de spelers, de verandering van de waterhoogte en natural events beschreven.

2 Pieces en Players specificatie

Dit onderdeel beschrijft de *Player*s en *Piece*s van het spel. Een *Player* bestuurt zijn *Piece*s, die allemaal van hetzelfde type *Animal* zijn. Er zijn twee *Player*s die elk een ander soort *Animal* besturen. *Player*s kunnen een verplaatsing van een *Piece* aanvragen aan het *Board*, waarop dat zal antwoorden met een positief danwel negatief antwoord. Als het *Board* de aangevraagde verplaatsing niet goedkeurt, dan blijft het *Piece* op zijn huidige positie staan. Een *Player* kan de speelstukken van de andere *Player* doden, door op dezelfde *Tile* te gaan staan als het *Piece* van de andere *Player*. Als dit gebeurt, dan wordt het *Piece* van de speler die al op de *Tile* stond, verwijderd van het *Board*. De overwinnaar zal dan de positie van het zojuist vermoordde *Piece* innemen.

> *Animal* == *Fox* | *Dolphin*

Een *Animal* is hetzij een *Fox*, hetzij een *Dolphin*. Dit geeft de diersoort van een *Piece* aan. Het soort *Animal* van een *Piece* wordt gedurende het spel niet meer veranderd. Elk speelstuk heeft precies één diersoort. *Fox*es kunnen zich alleen verplaatsen in de buurt van of op land, en *Dolphin*s kunnen zich alleen verplaatsen in de buurt van of in water. Als een *Fox* meer dan twee *Tile*s is verwijderd van land, dan zal deze zich niet meer kunnen verplaatsen, totdat de waterhoogte zodanig veranderd, dat de *Fox* weer op twee of minder *Tile*s van het land is verwijderd. Dit gaat net zo in het geval van een *Dolphin*, waarbij die uiteraard in de buurt van water moet zijn in plaats van land.

Piece

$type : Animal$
$x : \mathbb{Z}$
$y : \mathbb{Z}$

Een *Piece* bevat een type *Animal*, en een (x, y) positie op het *Board*. Dit zijn de speelstukken waarmee de *Player*s het spel spelen. De *Piece*s worden verplaatst door de spelers, met als doel op de locatie van *Piece*s van de andere *Player* te komen teneinde deze te vermoorden en op te eten. Uiteindelijk verliest de *Player* die al zijn *Piece*s verloren heeft. Het is dus de taak om deze *Piece*s zo lang mogelijk in leven te houden, en tegelijkertijd te proberen de *Piece*s van de andere speler op te eten.

Player

$Pieces : \mathbb{P} Pieces$

$\forall p_1, p_2 : Pieces \bullet p_1.type = p_2.type$

Een *Player* heeft een aantal *Piece*s. Deze *Piece*s zijn allemaal van hetzelfde type *Animal*. Alle *Piece*s van een *Player* zijn, zoals eerder gezegd, van hetzelfde type *Animal*. Eén van de *Player*s heeft alleen *Piece*s van het type *Fox*, de andere heeft alleen *Piece*s van het type *Dolphin*. De *Dolphin*s moeten proberen om alle *Fox*es op te eten, en de *Fox*es proberen alle *Dolphin*s te eten.

MoveRequest

$\Xi Player$
$!p : Piece$
$!x, !y : \mathbb{Z}$

$\exists piece \in Pieces \bullet !p = piece \land \|!x - piece.x\| \leqslant 1 \land \|!y - piece.y\| \leqslant 1$

Dit schema beschrijft de manier waarop een *Player* een verplaatsing van één van zijn *Piece*s aan kan vragen. De *Player* geeft een *Piece* op die hij wil verplaatsen en een (x, y) coördinaat om het *Piece* naar toe te verplaatsen. Dit (x, y) coördinaat mag maximaal één *Tile* van de huidige positie van het *Piece* dat verplaatst moet worden, verschillen.

DoMove

$\Delta Player$
$?p : Piece$
$?x, ?y : \mathbb{Z}$

$Pieces' = Pieces \cup \{(?s.type, ?x, ?y)\} \setminus \{?s\}$

Dit beschrijft het daadwerkelijk verplaatsen van een *Piece* van een *Player*. Het *Piece* dat verplaatst wordt, wordt uit de set van *Piece*s verwijderd. Daarna wordt er een nieuw *Piece* toegevoegd aan deze set, met hetzelfde type *Animal* als het originele *Piece* – de types van de *Piece*s van een *Player* moeten immers gedurende het hele spel ongewijzigd blijven – en de nieuwe (x, y) coördinaten.

DoKill

$?x, ?y : \mathbb{Z}$
$\Delta Player$

$\exists p \in Pieces \mid p.x = ?x \land p.y = ?y \land Pieces' = Pieces \setminus \{s\}$

Hier wordt een *Piece* gedood en uit de set van *Piece*s verwijderd. Het *Piece* dat op de opgegeven (x, y) coördinaten staat, wordt verwijderd, en dus wordt de nieuwe set van *Piece*s van de betreffende *Player* de oude set min het zojuist vermoordde *Piece*.

3 Board specificatie

De specificatie van het bord bestaat uit twee klassen, met namen *Tile* en *Board*. De klasse *Board* heeft een aantal operaties die het bord veranderen en/of een output genereren.

3.1 Tile

Tile

$h : \mathbb{Z}$
$x, y : \mathbb{N}$
$isWater : boolean$

$-50 \leqslant h \leqslant 50$

De klasse *Tile* zal verderop onder andere gebruikt worden om een *board* van *tiles* en de plaatsen van *bridges* en *caves* te specificeren. Naast de x- en y-waarden die moeten worden bijgehouden voor een *tile*, moet ook de hoogte h ervan en of het onder water staat ($isWater$) worden geregistreerd. De hoogte van een *tile* zit tussen -50 en 50.

3.2 Board

Board

$B : \mathbb{P}\, Tile$
$caves : \mathbb{P}\, Tile \times Tile$
$bridges : \mathbb{P}\, Tile \times Tile$
$w, wl, wu : \mathbb{Z}$
$nEvent : \mathbb{Z} \times \mathbb{Z}$
$pieces : \mathbb{P}\, Piece$

$wl \leqslant w \leqslant wu$
$\forall (b_1, b_2) \in (caves \lor bridges) \mid b_1 \in B \land b_2 \in B$
$\forall (x, y) \in nEvent \mid -20 \leqslant x \leqslant 20 \land 0 \leqslant y \leqslant 5$
$\forall\, p \in pieces \mid (\exists\, tile \in B \mid p.x = tile.x \land p.y = tile.y)$
$\forall\, t, t' \in B \mid t' \in Neighbours(tile) \Rightarrow \mid t.h - t'.h \mid \leqslant 2$
$\sum_{b \in B} b.h = 0$
$\forall (x, y) \in \mathbb{N} \times \mathbb{N} \mid (nEvent = (x, y) \Rightarrow (wl = -15 + x \land wu = 15 + x))$
$\forall\, b_1, b_2 \in B \mid (b_1 \neq b_2 \Rightarrow \neg(b_1.x = b_2.x \land b_1.y = b_2.y))$

De klasse *Board* heeft naast alle informatie over het speelbord ook de operaties die op dat speelbord uitgevoerd kunnen worden.
Variabele B representeert een spelbord van *tiles* die hierboven gespecificeerd zijn.

$$\forall\, t, t' \in B \mid t' \in Neighbours(tile) \Rightarrow \mid t.h - t'.h \mid \leqslant 2$$

Elke *neighbour* van een *tile* mag een maximaal hoogteverschil van 2 hebben met de *tile* zelf.

$$\forall (b_1, b_2) \in (caves \lor bridges) \mid b_1 \in B \land b_2 \in B$$

Daarnaast zijn er tuples van *tiles* die de begin- en eindtiles van *bridges* en *caves* representeren. De tiles waarnaar deze *bridges* en *caves* refereren moeten uiteraard bestaan.

$$wl \leqslant w \leqslant wu$$

De integers w, wl en wu zijn respectievelijk de huidige waterhoogte, minimale waterhoogte en maximale waterhoogte. De huidige waterhoogte moet tussen de minimale en maximale waterhoogten in liggen, die normaal gesproken -15 en 15 zijn.

De waarden van wl en wu kunnen veranderd worden tijdens een natural event, *nEvent*. Hierbij is x de waarde waarmee de waterhoogten worden verhoogd of verlaagd en y representeert het aantal dagen dat een event nog duurt.

$$\forall (x, y) \in nEvent \mid -20 \leqslant x \leqslant 20 \land 0 \leqslant y \leqslant 5$$

Een event kan de waterhoogten veranderen met minimaal -20 en maximaal 20. Daarnaast duurt een natural event nog maximaal vijf dagen. Als de waarde van y gelijk is aan nul, dan is er op dat moment geen natural event gaande.

$$\forall (x, y) \in \mathbb{N} \times \mathbb{N} \mid (nEvent = (x, y) \Rightarrow (wl = -15 + x \land wu = 15 + x))$$

Ongeacht er een natural event plaatsvindt, moet de invariant hierboven gelden. Dit betekent dat wl en wu hun standaardwaarden van -15 en 15 aannemen zodra er geen natural event plaatsvindt, maar veranderen naar $-15 + x$ en $15 + x$ bij een natural event met waarden (x, y).

$$\forall\, p \in pieces \mid (\exists\, tile \in B \mid p.x = tile.x \land p.y = tile.y)$$

De set van alle speelstukken over in het spel wordt weergegeven door *pieces*. Voor elke *Piece* in *pieces* moet er een *tile* bestaan in B met dezelfde x-, en y-coordinaten.

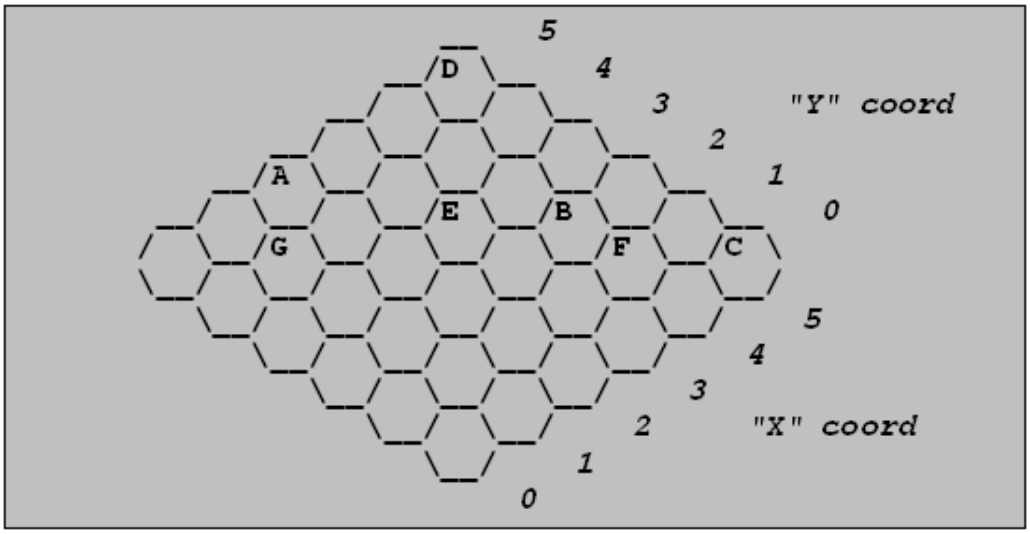
3.2.1 Neighbours

Neighbours

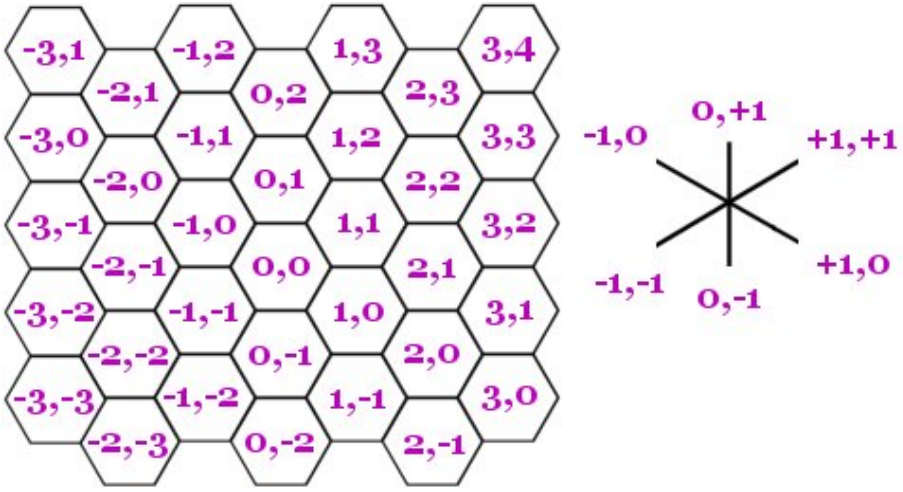
$\Xi Board$
$!neighbours : \mathbb{P} tile$
$?t : Tile$

$$\begin{aligned}
&\forall\, b \in B \mid (((t.x - 1 = b.x \land t.y - 1 = b.y) \lor \\
&\qquad (t.x - 1 = b.x \land t.y = b.y) \lor \\
&\qquad (t.x = b.x \land t.y + 1 = b.y) \lor \\
&\qquad (t.x + 1 = b.x \land t.y + 1 = b.y) \lor \\
&\qquad (t.x + 1 = b.x \land t.y = b.y) \lor \\
&\qquad (t.x = b.x \land t.y - 1 = b.y)) \Rightarrow b \in !neighbours) \\
&\qquad\quad \land \\
&\qquad (\neg((t.x - 1 = b.x \land t.y - 1 = b.y) \lor \\
&\qquad (t.x - 1 = b.x \land t.y = b.y) \lor \\
&\qquad (t.x = b.x \land t.y + 1 = b.y) \lor \\
&\qquad (t.x + 1 = b.x \land t.y + 1 = b.y) \lor \\
&\qquad (t.x + 1 = b.x \land t.y = b.y) \lor \\
&\qquad (t.x = b.x \land t.y - 1 = b.y)) \Rightarrow b \notin !neighbours)
\end{aligned}$$

De neighbours functie retouneert alle tiles die aangrenzend zijn aan de geparametiseerde tile. Een rastercoördinatenstelsel is onhandig voor het weergeven van hexagonale tiles. Daarom is gekozen voor hexagonale grid-coördinaten.



Hiermee kunnen buurtiles als volgt eenvoudig gedefinieerd worden.



In tegenstelling tot dit voorbeeld hebben onze tiles enkel positieve coordinaten, maar voor de definitie van buurtile maakt dit niet uit. De Z-specificatie kijkt voor alle tiles in B of hij aan een van de zes neighbour definities voldoet. Zo ja, dan wordt hij geretouneert in $!neighbours$, zo niet, dan niet.

3.3 Waterhoogte

3.3.1 Natural Event

NaturalEvent

$\Delta Board$

$$((\exists\, x \in \mathbb{Z} \mid nEvent = (x, 0)) \Rightarrow \exists\, e \in \mathbb{Z} \mid (e, 5) = nEvent' \land -20 \leqslant e \leqslant 20 \land e \neq 0)$$
$$\underline{\vee}$$
$$(0, 0) = nEvent'$$

Deze operatie op *Board* maakt het mogelijk om, als een event afgelopen is, een nieuw event te maken. In een dergelijk geval van een nieuw event krijgt *nEvent* een nieuwe waarde tussen de -20 en 20 (maar niet nul), die nog vijf dagen duurt. We krijgen dan een nieuwe versie van het board,

waarbij de rest van alle variabelen hetzelfde blijft. De kans dat er een nieuw event komt is 5%, waarbij de kans op elk getal tussen de -20 en 20 uniform verdeeld is. Let op: *nEvent* wordt niet verandert, als het aantal dagen dat het huidige event nog duurt, nog niet 0 is. Verder staat $\veebar$ voor XOR.

3.3.2 Tide

Tide

$\Delta Board$

$$\begin{aligned}
&\exists\, e \in \mathbb{Z} \mid -5 \leqslant e \leqslant 5 \land e \neq 0 \land \\
&\qquad (e + w < wl \Rightarrow w' = wl) \land \\
&\qquad (e + w > wu \Rightarrow w' = wu) \land \\
&\qquad (wl \leqslant w + e \leqslant wu \Rightarrow w' = w + e)
\end{aligned}$$

Elke beurt van het *board* is er sprake van een *tide*, het getij, waardoor de waterhoogte wordt veranderd. De *tide* kan een waarde tussen -5 en 5 (zonder nul) hebben. Dus de waterhoogte moet elke beurt veranderen.

$$(e + w < wl \Rightarrow w' = wl) \land (e + w > wu \Rightarrow w' = wu)$$

Het is mogelijk dat de tide opgeteld bij de waterhoogte kleiner is dan de minimale waterhoogte. In dit geval stellen we de nieuwe waterhoogte w' gelijk aan de minimale waterhoogte. Als de tide opgeteld bij de waterhoogte groter is dan de maximale waterhoogte, wordt dit op dezelfde manier afgehandeld; w' is dan gelijk aan de maximale waterhoogte. Ook geldt er dat de kans op elk getal tussen de -5 en 5 uniform verdeeld is.

3.3.3 Flood

Flood

$\Delta Board$

$$\begin{aligned}
&\#B' = \#B \\
&(\forall\, b \in B \mid ((\exists\, c \in Neighbours(b) \mid b.isWater \neq c.isWater) \Rightarrow \\
&\qquad (\exists\, b' \in B' \mid (b.h > w \Rightarrow \neg b'.isWater) \land \\
&\qquad\qquad (b.h < w \Rightarrow b'.isWater) \land \\
&\qquad\qquad (b.h = w \Rightarrow b'.isWater = b.isWater) \land \\
&\qquad\qquad (b'.x = b.x) \land \\
&\qquad\qquad (b'.y = b.y) \land \\
&\qquad\qquad (b'.h = b.h))) \\
&\quad \land \\
&\quad ((\neg \exists\, c \in Neighbours(b) \mid b.isWater \neq c.isWater) \Rightarrow \\
&\qquad (\exists\, b' \in B' \mid (b'.x = b.x) \land \\
&\qquad\qquad (b'.y = b.y) \land \\
&\qquad\qquad (b'.h = b.h))))
\end{aligned}$$

Nadat de tide, en dus de waterhoogte is veranderd, wordt gekeken of er veranderingen zijn in welke vakjes nu wél of niet onder water staan. Er zit een restrictie aan de snelheid waarin het water zich verplaatst over het land, namelijk dat dit maar met 2 vakjes per beurt kan, in de richting waar het zich opschuift.

$$\forall\, b \in B \mid (\exists\, c \in Neighbours(b) \mid b.isWater \neq c.isWater) \Rightarrow$$

Hier kijken we voor alle tiles of ze een buurtile hebben waar water staat als de tile land is en andersom. De functie Flood wordt zo opgesteld, zodat hij tweemaal aangeroepen kan worden. Dit

vergemakkelijkt het specificeren, aangezien we nu enkel naar de neighbors kunnen flooden. Door Flood tweemaal aan te roepen beslaan we dus ook de tweede buren van een tile. Als de eerste existentiekwantor geldt, dan moet ook het volgende waar zijn.

$$
\begin{aligned}
&(\exists\, b' \in B' \mid (b.h > w \Rightarrow \neg b'.isWater) \ \wedge \\
&\quad (b.h < w \Rightarrow b'.isWater) \ \wedge \\
&\quad (b.h = w \Rightarrow b'.isWater = b.isWater) \ \wedge \\
&\quad (b'.x = b.x) \ \wedge \\
&\quad (b'.y = b.y) \ \wedge \\
&\quad (b'.h = b.h)))
\end{aligned}
$$

Als de hoogte van een tile hoger is dan de waterhoogte, dan is het water op die tile weggespoeld in de afterstate. Maar als de hoogte van de tile lager is is dan de waterhoogte, dan is de tile in de afterstate natuurlijk overspoeld. Mocht het zo zijn dat de hoogte van een tile gelijk is aan de waterhoogte, dan verandert de tile niet. Verder willen we hier ook nog bevestigen dat de x-waarde, y-waarde en de hoogte van een tile niet veranderd zijn in de afterstate.

$$
\begin{aligned}
&(\neg \exists\, c \in Neighbours(b) \mid b.isWater \neq c.isWater) \Rightarrow \\
&\quad (\exists\, b' \in B' \mid (b'.x = b.x) \ \wedge \\
&\qquad (b'.y = b.y) \ \wedge \\
&\qquad (b'.h = b.h)))
\end{aligned}
$$

Het moet natuurlijk ook zo zijn dat de tiles die niet aan de vorige existentiekwantor voldeden, geen andere waarden aannemen in de afterstate.

$$\#B' = \#B$$

Het zelfde geldt voor de bovenstaande regel, hiermee willen we voorkomen dat het aantal tiles in de afterstate B' verschilt van het aantal tiles in de huidige state B.

3.4 Moves

Het aanvragen van moves, het verifiëren en het uitvoeren daarvan is een erg groot proces. Dit in één enkele operatie te zetten zou, naast erg onoverzichtelijk, ook dubbelop zijn. Om dit te voorkomen zijn er hulpfuncties ingebouwd die eerst uitgelegd zullen worden alvorens de verificatie en uitvoering van moves uiteen wordt gezet.

3.4.1 In Range

InRange

$\Xi Board$
$!isInRange : boolean$
$?p : Piece$

$$
\begin{aligned}
&((?p.species = fox) \Rightarrow \\
&\qquad !isInRange = (\exists\, t \in B \mid ?p.x = t.x \wedge ?p.y = t.y \wedge \\
&\qquad (\exists\, n \in Neighbour(t) \mid n.x = ?x \wedge n.y = ?y) \wedge \\
&\qquad (\exists\, n_1 \in Neighbour(t) \mid (\exists\, n_2 \in Neighbour(n) \mid (n_2.isWater = false \vee n_1.isWater = false))))) \\
&\quad \wedge \\
&((?p.species = dolphin) \Rightarrow \\
&\qquad !isInRange = (\exists\, t \in B \mid ?p.x = t.x \wedge ?p.y = t.y \wedge \\
&\qquad (\exists\, n \in Neighbour(t) \mid n.x = ?x \wedge n.y = ?y) \wedge \\
&\qquad (\exists\, n_1 \in Neighbour(t) \mid (\exists\, n_2 \in Neighbour(n) \mid (n_2.isWater = true \vee n_1.isWater = true)))))
\end{aligned}
$$

Deze hulpfunctie kijkt of het voor een Piece mogelijk is om te kunnen bewegen. Om te kunnen bewegen moet een dolphin maximaal twee tiles van water af zijn, en een fox maximaal twee tiles van land af. Zodra ze verder zijn dan dat kunnen zij niet meer bewegen, wat uiteindelijk zal resulteren in een ongeldige en dus onuitgevoerde move.
We zullen nu de InRange functie uitleggen in het geval dat $?p.species = fox$, aangezien het dusdanig veel overeenkomt met het geval van $?p.species = dolphin$ dat het overbodig zou zijn om beide gevallen apart uit te leggen.

$$!isInRange = (\exists\, t \in B \mid ?p.x = t.x \land ?p.y = t.y \ \land$$
$$(\exists\, n \in Neighbour(t) \mid n.x = ?x \land n.y = ?y) \ \land$$
$$(\exists\, n_1 \in Neighbour(t) \mid (\exists\, n_2 \in Neighbour(n) \mid (n_2.isWater = false \lor n_1.isWater = false)))$$

De variabele !isInRange wordt true als de tile in B, met gelijke coördinaten als de input, een neighbour, of een neighbour van een neighbour heeft die land is. Dit is in overeenkomst met de eerder genoemde regels. Voor een dolfijn geldt hetzelfde, maar dan met water.

3.4.2 Occupied By Same Animal

OccupiedBySameAnimal

$\Xi Board$
$?p : Piece$
$?x, ?y : \mathbb{N}$
$!isOccupied : boolean$

$$!isOccupied = \exists\, p \in Pieces \mid p.x = ?x \land p.y = ?y \land p.species = ?p.species$$

Uiteraard mag de tile waar een Piece $?p$ naartoe wil gaan niet bezet zijn door een Piece van zijn eigen soort. Zodra dit het geval is, is een move naar deze tile mogelijk en is $!isOccupied$ true.

3.4.3 Make Move

MakeMove

$\Delta Board$
$?p : Piece$
$?x, ?y : \mathbb{N}$
$!killed : boolean$

$$\begin{aligned}
&((?p.species = fox \Rightarrow \\
&\quad ((\neg OccupiedBySameAnimal(dolphin, ?x, ?y) \Rightarrow \\
&\qquad (!killed = false \land Pieces' = Pieces \cup \{(fox, ?x, ?y)\} \setminus \{?p\})) \\
&\quad \land \\
&\quad (OccupiedBySameAnimal(dolphin, ?x, ?y) \Rightarrow \\
&\qquad (!killed = true \land Pieces' = Pieces \cup \{(fox, ?x, ?y)\} \setminus \{(?p), (dolphin, ?x, ?y)\}))))) \\
&(?p.species = dolphin \Rightarrow \\
&\quad ((\neg OccupiedBySameAnimal(fox, ?x, ?y) \Rightarrow \\
&\qquad (!killed = false \land Pieces' = Pieces \cup \{(dolphin, ?x, ?y)\} \setminus \{?p\})) \\
&\quad \land \\
&\quad (OccupiedBySameAnimal(fox, ?x, ?y) \Rightarrow \\
&\qquad (!killed = true \land Pieces' = Pieces \cup \{(dolphin, ?x, ?y)\} \setminus \{(?p), (fox, ?x, ?y)\})))))
\end{aligned}$$

Naast een zet goedkeuren, moet het board zelf ook nog aangepast worden. Om dit te doen werd *MakeMove* in het leven geroepen. Allereerst wordt gekeken of er een dier van het andere soort (van $?p$) op de tile staat waar $?p$ heen wil lopen (met coördinaten $?x$, $?y$). Als dit zo is, wordt het dier vermoord, uit de set van *pieces* gehaald en true als output voor $!killed$ gegeven. In deze operatie definiëren we (Dier, $\mathbb{N}$, $\mathbb{N}$) als een manier om een object van klasse *piece* te maken.

Ook hier zullen we enkel de formele specificatie voor een zet van $?p.species = fox$ uitleggen, aangezien het dezelfde werking heeft voor een $?p.species = dolphin$, maar dan andersom.

$$\neg OccupiedBySameAnimal(dolphin, ?x, ?y) \Rightarrow$$
$$(!killed = false \land Pieces' = Pieces \cup \{(fox, ?x, ?y)\} \setminus \{?p\})$$

Zoals hier te zien is, gebruiken we de functie OccupiedBySameAnimal om te controleren of er een dolphin op de bestemde tile staat. Als dit niet het geval is, dan $!killed = false$ en in de nieuwe set $Pieces'$ is de meegegeven Piece $?p$ weggehaald, maar eenzelfde soort Piece met coördinaten van de bestemming is toegevoegd.

$$OccupiedBySameAnimal(dolphin, ?x, ?y) \Rightarrow$$
$$(!killed = true \land Pieces' = Pieces \cup \{(fox, ?x, ?y)\} \setminus \{(?p), (dolphin, ?x, ?y)\})$$

Dit geeft het geval weer waarin er wel een dolphin op de bestemming staat. De variabele $!killed$ is dan true en de nieuwe set $Pieces'$ heeft, naast het verplaatsen van de fox, ook de desbetreffende dolphin verwijderd.

3.4.4 Shortcut Possible

ShortcutPossible

$\Xi Board$
$?p : Piece$
$?x, ?y : \mathbb{N}$
$!isPossible : boolean$

$$((?p.species = fox) \Rightarrow$$
$$!isPossible = (\exists\, t_1 \in B \mid ?p.x = t_1.x \land ?p.y = t_1.y \land$$
$$\exists\, t_2 \in B \mid (t_2.x = ?x \land t_2.y = ?y \land \neg t_1.isWater \land \neg t_2.isWater \land$$
$$\neg OccupiedBySameAnimal(fox, ?x, ?y) \land \exists\, b \in bridges \mid (b = (t_1, t_2) \lor b = (t_2, t_1)))))$$
$$\land$$
$$((?p.species = dolphin) \Rightarrow$$
$$!isPossible = (\exists\, t_1 \in B \mid ?p.x = t_1.x \land ?p.y = t_1.y \land$$
$$\exists\, t_2 \in B \mid (t_2.x = ?x \land t_2.y = ?y \land t_1.isWater \land t_2.isWater \land$$
$$\neg OccupiedBySameAnimal(dolphin, ?x, ?y) \land \exists\, b \in bridges \mid (b = (t_1, t_2) \lor b = (t_2, t_1)))))$$

Zoals de naam het misschien al zegt, wordt hier bepaald of er een shortcut mogelijk is op een bepaalde tile. Met shortcut bedoelen we een cave of bridge. Aangezien een fox alleen een bridge en een dolphin alleen een cave mag gebruiken, moet hier rekening mee gehouden worden. Om die reden wordt een *Piece* als input gegeven. Daarnaast kan zo'n shortcut alleen gebruikt worden door een stap naar het uiteinde van de bridge of cave te maken, wanneer het *Piece* aan de ingang van de bridge of cave staat. De meegegeven x- en y-coördinaten moeten dus gelijk zijn aan de tile waar het uiteinde naar refereert.
We leggen hier alleen de formele specificatie voor fox uit omdat het bijna hetzelfde is voor dolphin.

$$!isPossible = (\exists\, t_1 \in B \mid ?p.x = t_1.x \land ?p.y = t_1.y \land$$
$$\exists\, t_2 \in B \mid (t_2.x = ?x \land t_2.y = ?y \land \neg t_1.isWater \land \neg t_2.isWater \land$$
$$\neg OccupiedBySameAnimal(fox, ?x, ?y) \land \exists\, b \in bridges \mid (b = (t_1, t_2) \lor b = (t_2, t_1)))$$

$!isPossible$ geeft aan of een afkorting via een bridge mogelijk is, wat alleen kan als beide tiles niet onder water staan en er geen fox staat op de andere kant van de bridge. Het laatste stukje geeft aan dat er daadwerkelijk een bridge moet zijn, het tuple van tiles moet zijn opgeslagen in het bord.

3.4.5 Request Move

RequestMove

$\Delta Board$

$!movedXkilled : boolean \times boolean$

$?p : Piece$

$?x, ?y : \mathbb{N}$

$$(InRange(?p) \land \neg OccupiedBySameAnimal(?p, ?x, ?y)) \Rightarrow (!movedXkilled = (true, MakeMove(?p, ?x, ?y)))$$
$$\land$$
$$(ShortcutPossible(?p, ?x, ?y) \land \neg OccupiedBySameAnimal(?p, ?x, ?y)) \Rightarrow (!movedXkilled = (true, MakeMove(?p, ?x, ?y)))$$
$$\land$$
$$(\neg(InRange(?p) \land \neg OccupiedBySameAnimal(?p, ?x, ?y) \land ShortcutPossible(?p, ?x, ?y))) \Rightarrow (!movedXkilled = (false, false))$$

Deze operatie maakt gebruik van de functies die hierboven uitgelegd staan. Het wordt gebruikt door de controller om bij het *board* een zet aan te vragen. Vervolgens wordt gekeken of de zet geldig is en zo ja, dan wordt die ook uitgevoerd. Om een zet aan te vragen wordt een *piece*, en een x-, en y-coördinaat gevraagd, waarvoor geldt dat het gevraagde *piece* verplaatst wordt naar de gegeven x- en y-coördinaten. Vervolgens wordt een tuple van booleans terug gegeven, waarbij de eerste weergeeft of de zet is uitgevoerd en de tweede of er een dier is vermoord. Zodra bekend is dat een zet geldig is, wordt *MakeMove* gebruikt om het uiteindelijk in het *board* te verwerken. *MakeMove* geeft ook een boolean terug die wordt gebruikt in de tuple van boolean om te definiëren of er een dier is vermoord.

$$(InRange(?p) \land \neg OccupiedBySameAnimal(?p, ?x, ?y)) \Rightarrow (!movedXkilled = (true, MakeMove(?p, ?x, ?y)))$$

Als een Piece $?p$ volgens InRange kan bewegen en als de bestemmende tile niet bezet is door eenzelfde dier, dan is het een geldige move en wordt deze uitgevoerd.

$$(ShortcutPossible(?p, ?x, ?y) \land \neg OccupiedBySameAnimal(?p, ?x, ?y)) \Rightarrow (!movedXkilled = (true, MakeMove(?p, ?x, ?y)))$$

Hier beslaan we het geval van een cave of een bridge. Dus als een afkorting via een dergelijk medium mogelijk is en er staat op de bestemmende tile geen dier van hetzelfde soort als $?p$, dan is de move geldig en wordt deze uitgevoerd.

$$(\neg(InRange(?p) \land \neg OccupiedBySameAnimal(?p, ?x, ?y) \land ShortcutPossible(?p, ?x, ?y))) \Rightarrow (!movedXkilled = (false, false))$$

Het kan ook zo zijn dat een speler een ongeldige move aanvraagt. Als Piece $?p$ niet kan bewegen volgens de functie InRange, als er een zelfde soort dier als $?p$ op de bestemming staat of als er geen mogelijke afkorting via een bridge of een cave mogelijk is, dan hebben we te maken met een ongeldige move. In dat geval zijn beide booleans van $!movedXkilled$ false, aangezien ook geen *piece* gedood kan worden als het *piece* niet kan bewegen.

3.5 Number of Moves

getNrMoves

$\Xi Board$

$!moves : \mathbb{N}$

$!moves = \lceil (\#Pieces/4) \rceil$

Dit geeft het aantal moves weer dat een speler mag doen per beurt. Deze operatie rekent het automagisch uit.

4 View en Controller specificatie

4.1 View

View

$board : Board$

De *View* zorgt ervoor dat de huidige spelsituatie op het scherm verschijnt. Het *View* bevat daarom enkel *Board*. Deze entiteit heeft dus enkel een update schema nodig om goed te kunnen functioneren. In de MSC in Figuur 2 is te zien hoe de *View* de huidige situatie van het *Board* krijgt.

UpdateView

$\Delta Viewb? : Board$

$board' = b?$

Het *View* wordt door de *Controller* up-to-date gehouden door op een specifiek moment, zoals we eerder hebben gespecificeerd, het *Board* naar het *View* te sturen met deze operatie.

4.2 Controller

De *Controller* is de allesomvattende klasse. Alle onderdelen van het spel, zoals de *Player*s, *Piece*s, *Board* etcetera, worden geïnitialiseerd door de *Controller*. Deze is ook het doorgeefluik voor de communicatie tussen *Player*s en het *Board*.

4.2.1 Z-specificatie

Controller

$b : Board$

$p_1, p_2 : Player$

$v : View$

$\forall\, piece_1 \in p_1.Pieces, piece_2 \in p_2.Pieces \bullet piece_1.type \neq piece_2.type$

$b.Pieces = p_1.Pieces \cup p_2.Pieces$

$v.board = b$

Dit schema zorgt ervoor dat de types *Animal* van beide *Player*s verschillend zijn. Ook wordt de set van alle *Piece*s in het spel gedefiniëerd door de set van *Piece*s van *Player* 1 gecombineerd met die van *Player* 2. De *Controller* zorgt er verder voor dat de *View* de op dit moment correcte versie van het *Board* heeft.

Init

$n? : \mathbb{N}$
$board' : Board$
$Player'_1 : Player$
$Player'_2 : Player$
$view' : View$

$\#Player'_1.Pieces = \#Player'_2.Pieces = n?$
$board'.w = 0$
$board'.wl = -15$
$board'.wu = 15$
$\forall\, v, v' \in board'.B \mid v \neq v' \land v.h \geq 0 \land v'.h \geq 0 \bullet Connected(v, v')$

De initialisatie van de *Controller* bestaat uit de volgende onderdelen: eerst worden de *Players* geïnitialiseerd met $n?$ Pieces. De initiële waterhoogte is nul, de ondergrens voor waterhoogte is -15, en de bovengrens is 15.

Verder geldt dat, aan het begin van het spel, alle land tiles hetzij direct aan elkaar liggen, hetzij er een pad is via onderling verbonden stukken land van elke willekeurige land tile naar elke andere willekeurige land tile. Deze eigenschap wordt formeel gedefiniëerd in het schema *Connected*.

Connected

$v?, w? : Tile$
$!o : Boolean$

$((v? \in Neighbours(w?) \lor \exists\, v' \in Tile \mid v'.h \geq 0 \bullet$
$\quad (v' \in Neighbours(w?) \land Connected(v', v?))) \Rightarrow (!o = true))$
$\lor$
$(\neg(v? \in Neighbours(w?) \lor \exists\, v' \in Tile \mid v'.h \geq 0 \bullet$
$\quad (v' \in Neighbours(w?) \land Connected(v', v?))) \Rightarrow (!o = false))$

Dit schema retourneert `true` als er een pad is via land van vakje $v?$ naar vakje $w?$, en `false` als dat niet het geval is. Dit werkt alleen tijdens de initialisatiefase, aangezien dan elk vakje met hoogte $\geqslant 0$ gegarandeerd land is. Het gaat als volgt te werk: eerst controleert het schema of $v?$ en $w?$ direct naast elkaar liggen, dat wil zeggen, $w?$ zit in de verzameling van *Tile*s die grenzen aan $v?$. Als dit het geval is, dan wordt direct een waarde $!o =$ `true` geretourneerd.

Als dit niet het geval is, dan wordt gekeken of er een tussenliggende *Tile* v' is met hoogte $\geqslant 0$ (dus v' is land), waarvoor geldt dat v' direct verbonden is met $w?$, en er een recursief pad is zodanig dat uiteindelijk $v?$ en v' verbonden zijn. Als hieraan voldaan is, dan is er dus een pad van $v?$ naar $w?$ over land.

Als beide gevallen niet waar zijn, dan wordt de outputwaarde $!o =$ `false` geretourneerd.

4.2.2 State Diagram en Message Sequence Chart

In deze sectie behandelen we de communicatie tussen de uitgewerkte classes. In een state diagram zullen we laten zien hoe we de klassen combineren om een ronde in het spel uit te voeren. Hoe een beurt van een specifieke speler in zijn werk gaat, laten we afzonderlijk zien in een MSC.

Het State Diagram in Figuur 1 laat zien hoe de verschillende classes van de Controller samenwerken tijdens een ronde van het spel. De *Player*s en het *Board* zijn aan de start van het spel *idle*, aangezien ze op de *Controller* moeten wachten voor ze iets mogen doen. We gaan er voor het modelleren in dit state diagram vanuit dat de *View* op elk moment een *Outdated* Board kan hebben, maar de *Controller* zorgt dat na bepaalde acties het *View* weer up-to-date wordt. Hier komen we later nog op terug.

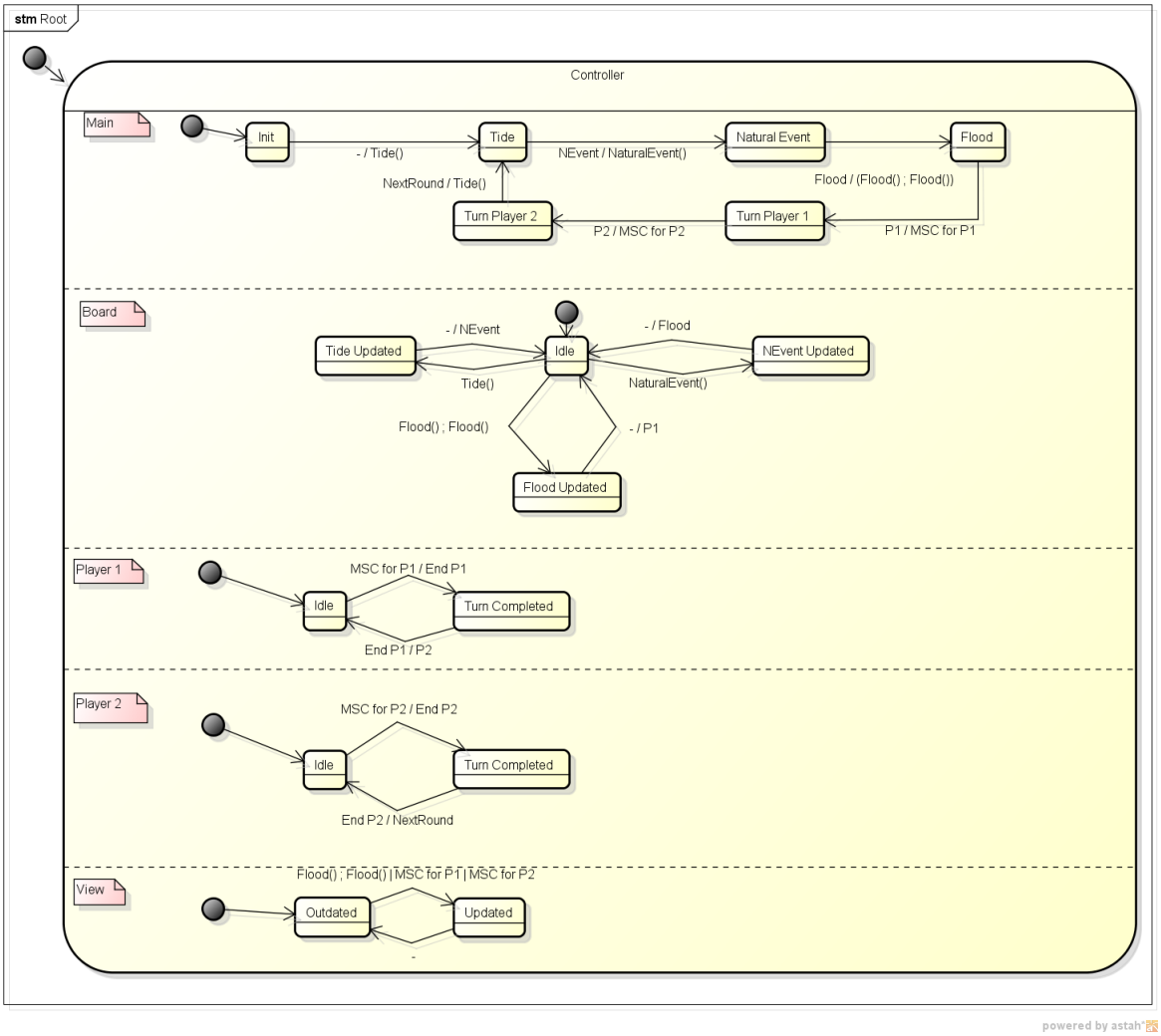
Wanneer de *Controller* alles geïnitialiseerd heeft, kan het spel beginnen. Aan het begin van elke ronde wordt de *Tide* geüpdatet (met operatie *Tide*() uit *Board*) en het *Natural Event* worden aangepast (met *NaturalEvent*() uit *Board*). Nu alle aanpassingen aan de waterhoogte berekend zijn,

kan de controller zorgen dat *Flood* wordt uitgevoerd om het water te laten voortbewegen. Aangezien *Flood* het water maximaal 1 vakje laat voortbewegen, is het water pas op de juiste manier verandert wanneer *Flood* twee keer is aangeroepen (compositie van operatie *Flood*(): *Flood*(); *Flood*(). Nu is het updaten van het *Board* afgelopen en moet het *View* geüpdatet worden, vandaar dat een van de mogelijke events die het *View* in de *Updated* state brengt *Flood*(); *Flood*() is.

Nu het *Board* is aangepast, kunnen de *Player*s aan de beurt komen. Eerst is *Player* 1 aan de beurt en daarna *Player* 2. Na een beurt van een *Player* wordt ook het *View* weer up-to-date gemaakt, zoals.

Wanneer de beurten van de spelers voorbij zijn, begint een nieuwe beurt met het updaten van de waterhoogte van het *Board*. Elke ronde van het spel heeft dus de volgende volgorde: Update water(hoogte) van *Board*, vervolgens is *Player* 1 aan de beurt, en als laatste is *Player* 2 aan de beurt.

De events, die de beurt van de *Player*s aangeven, heten *MSC for P*1/2. De beurt van de *Player*s worden later uitgelegd in een MSC en dit event refereert naar de events die plaatsvinden in de MSC.



Figuur 1: State Diagram voor de controller

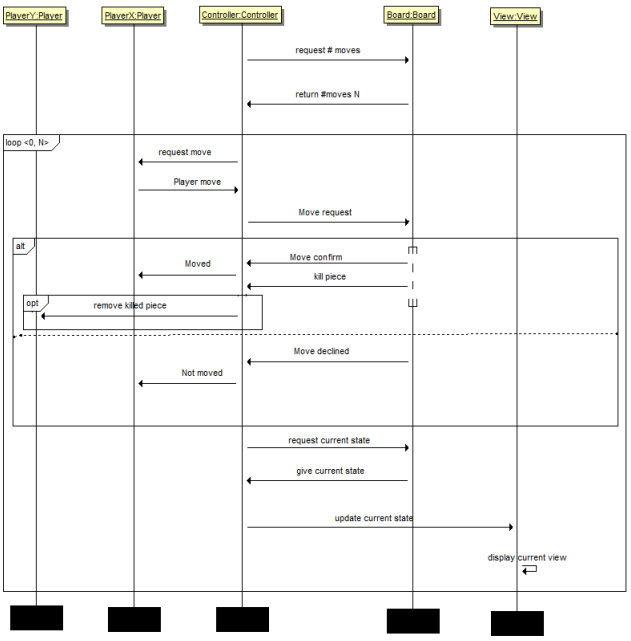
De Message Sequence Chart in Figuur 2 laat de communicatie zien tussen *Player*s, het *Board*,

de *Controller* en de *View*, tijdens een beurt van een *Player*. Wanneer *Player* 1 aan de beurt is, is hij in de MSC *PlayerX* en zal *Player* 2 gelijk zijn aan *PlayerY*. Dit is precies omgekeerd wanneer *Player* 2 aan de beurt is.

Een beurt start, wanneer de *Controller* aan het *Board* vraagt hoeveel moves een speler nog mag doen. De *Controller* doet dit door *getNrMoves*() uit *Board* aan te roepen en zal als output het aantal moves krijgen dat een speler mag doen. Nu zorgt de *Controller* er met een loop voor dat de *Player* inderdaad zoveel zetten mag doen.

Elke zet bestaat uit het aanvragen van een zet bij de *Player* door middel van *MoveRequest* van die *Player*. De *Player* geeft dan een speelstuk en coördinaten ($!p$, $!x$ en $!y$) terug die als input dienen voor de aanvraag van een zet, die de *Controller* bij het *Board* doet (met $RequestMove(!p, !x, !y)$). Het *Board* zal nu aangeven of de move daadwerkelijk mag plaatsvinden of niet, vandaar het alternatieve scenario wanneer een move niet mag worden uitgevoerd. *RequestMove* geeft als output of de move geldig is en of een speelstuk is gedood. Wanneer een move geldig (move confirm) is zal de *Controller* de *Player* die de zet deed updaten door middel van $DoMove(!p, !x, !y)$ en wanneer een stuk gedood is (kill piece) zal de andere *Player* geüpdatet worden met $DoKill(!x, !y)$.

Nu een hele zet van een *Player* voorbij is moet de *Controller* de huidige status van het *Board* worden doorgegeven aan het *View*, met $UpdateView(b)$. Het hele process dat hierboven beschreven is wordt herhaald voor het aantal zetten dat een *Player* kan doen. Een uitzondering hierop geldt, wanneer een move ongeldig was. In dat geval zal die iteratie in de loop niet meetellen, aangezien de loop het aantal zetten voorstelt en de zet niet uitgevoerd kan worden.



Figuur 2: Message Sequence Chart voor de controller

5 Conclusie

Tot slot kunnen we stellen dat we met deze formele specificatie aan alle gevraagde eisen voldoen van de informele specificatie. Het board en de players werken precies via de opgelegde regels en de controller zal de gevraagde taak goed uitvoeren. Verdere verbeteringen kunnen betrekking hebben op de intelligentie van de *Player*s, deze bewegen nu naar een random neighbor vakje, en in sommige

gevallen iets verder weg. Deze intelligentie zal ook foute moves en moves via bridges/caves kunnen genereren en was voor de gegeven informele specificatie goed genoeg. Vervolgprojecten zouden zich kunnen richten op deze AI verbeteringen voor *Player*. Naast deze verbetering zou er ook nog extra werk gedaan worden om verschillende vaste bordtypen te genereren. Voor nu voldoet de specificatie echter aan alle gestelde eisen.