

# Internship Report

Solal Rapaport

June 2022

## 1 Introduction

The goal is to build formulas that will allow robots, spread on a ring, to gather. We have  $k$  robots and we will use view vectors to build those formulas. The formulas will be an interpretation of the pseudo-code given in the research report [1].

The formulas we are building, will be used with formulas given in an other research report [2], and then will be tested in the acceleration algorithm using an interpolant [2]

## 2 Configuration with single multiplicity

The strategy  $\phi_{SM}$  is *true* if the given configuration has a single multiplicity and that the robot calling the strategy should move toward the robot at distance  $d_0$  :

$$\begin{aligned} \phi_{SM}(d_0, \dots, d_{k-1}) := & \\ (\bigvee_{i=0}^{k-1} (d_i = 0 \wedge \bigwedge_{j=0, j \neq i}^{k-1} (d_j > 0 \vee (d_j = 0 \wedge d_{j-1} = 0)))) \wedge & \\ (d_{k-1} \neq 0) \wedge & \\ ((d_1 = 0 \wedge d_{k-2} = 0 \wedge d_0 \leq d_{k-1}) \vee (d_1 = 0 \wedge d_{k-2} \neq 0)) & \end{aligned}$$

In order to test our strategy we need a function that will initialize our first configuration and make it one with a single multiplicity without being already a winning one. Here is its implementation in python :

```
def InitSM(p, s, t, taille_anneau):
    """
    Initialize the given configuration
    The configuration will have no multiplicity
    and won't be a winning one
    """
    tmpOr = []
    tmpAnd = []
    for i in range(len(p)):
        tmpOr.append(p[i] != p[(i+1)%len(p)])
    tmpAnd.append(Or(tmpOr))
    # we make sure there is no winning configuration
    # as a result of this function
    tmpOr = []
    for i in range(len(s)):
        tmpAnd.append(p[i] >= 0)
        tmpAnd.append(p[i] < taille_anneau)
        tmpAnd.append(s[i] == -1)
        tmpAnd.append(t[i] == 0)

    for i in range(len(p)):
        tmpAndbis = []
        tmpAndbis.append(p[i] == p[(i+1)%len(p)])
        tmpOrbis = []
        for j in range(len(p)):
```

```

        if((j != i) and (j != (i+1))):
            tmpOrbis.append(p[j] == p[i])
            tmpAndter=[]
            for h in range(len(p)):
                if h !=j:
                    tmpAndter.append(p[h] != p[j])
            tmpOrbis.append(And(tmpAndter))
        tmpAndbis.append(Or(tmpOrbis))
        tmpOr.append(And(tmpAndbis))
    tmpAnd.append(Or(tmpOr))
    return And(tmpAnd)

```

We want to test this function, we choose a configuration with a ring of size 5 and 3 robots. We use the parameters below as an input :

```

taille_anneau = 5          # Taille de l'anneau
nb_robots = 3              # Nombre de robot sur l'anneau

```

```

p = [ Int('p%s' % i) for i in range(nb_robots) ]
s = [ Int('s%s' % i) for i in range(nb_robots) ]
t = [ Int('t%s' % i) for i in range(nb_robots) ]

```

```

tabInit = InitSM(p, s, t, taille_anneau)
print("tabInit_:\n", tabInit)

```

```

solv1 = Solver()
solv1.add(tabInit)

```

```

print("solv1_:", solv1.check())
if(solv1.check() == sat):
    print("model_:\n", solv1.model())

```

The code above produces the following output :

```

tabInit :
And(Or(p0 != p1, p1 != p2, p2 != p0),
    p0 >= 0,
    p0 < 5,
    s0 == -1,
    t0 == 0,
    p1 >= 0,
    p1 < 5,
    s1 == -1,
    t1 == 0,
    p2 >= 0,
    p2 < 5,
    s2 == -1,
    t2 == 0,
    Or(And(p0 == p1, Or(p2 == p0, And(p0 != p2, p1 != p2))),
        And(p1 == p2, Or(p0 == p1, And(p1 != p0, p2 != p0))),
        And(p2 == p0,
            Or(p0 == p2,
                And(p1 != p0, p2 != p0),
                p1 == p2,
                And(p0 != p1, p2 != p1))))))
solv1 : sat
model :
[p1 = 1,
 p0 = 0,
 p2 = 0,

```

```

t2 = 0,
s2 = -1,
t1 = 0,
s1 = -1,
t0 = 0,
s0 = -1]

```

What the model shows us is a configuration where robots 0 and 2 are in position 0 and robot 1 is in position 1.

Now we implement in python the  $\phi_S M$  strategy :

```

def phiSM( distances ):
    tabAnd = []
    tabOr = []
    for i in range(len( distances )):
        tabAndBis = []
        tabAndBis.append( distances [ i ] == 0 )
        for j in range(len( distances )):
            if j != i:
                tabOrBis.append( distances [ j ] > 0 )
                tabOrBis.append( And( distances [ j ] == 0,
                    distances [ j - 1 ] == 0 ) )
                tabAndBis.append( Or( tabOrBis ) )
        tabOr.append( And( tabAndBis ) )
    tabAnd.append( Or( tabOr ) )
    tabAnd.append( distances [ - 1 ] != 0 )
    tabOr = []
    tabOr.append( And( distances [ 1 ] == 0, distances [ - 2 ] == 0,
        distances [ 0 ] <= distances [ - 1 ] ) )
    tabOr.append( And( distances [ 1 ] == 0, distances [ - 2 ] != 0 ) )
    tabAnd.append( Or( tabOr ) )
    return And( tabAnd )

```

We test this function in the same configuration than given before with the following input :

```

taille_anneau = 5          # Taille de l'anneau
nb_robots = 3              # Nombre de robot sur l'anneau

p = [ Int( 'p%s' % i ) for i in range(nb_robots) ]
s = [ Int( 's%s' % i ) for i in range(nb_robots) ]
t = [ Int( 't%s' % i ) for i in range(nb_robots) ]

d0 = [ Int( 'd%s' % i ) for i in range(nb_robots) ]
d1 = [ Int( 'd%s' % i ) for i in range(nb_robots) ]
d2 = [ Int( 'd%s' % i ) for i in range(nb_robots) ]

tabInit = InitSM( p, s, t, taille_anneau )
tabConfig1 = ConfigView( taille_anneau, nb_robots, 0, p, d0 )
tabConfig2 = ConfigView( taille_anneau, nb_robots, 1, p, d1 )
tabConfig3 = ConfigView( taille_anneau, nb_robots, 2, p, d2 )
tabPhiSM1 = phiSM( d0 )
tabPhiSM2 = phiSM( d1 )
tabPhiSM3 = phiSM( d2 )

solv1 = Solver()
solv1.add( tabInit )
solv1.add( tabConfig1 )
solv1.add( tabPhiSM1 )

```

```

print("solv1_:_" , solv1 . check ())
if(solv1 . check () == sat ):
    print("model_:\"n\" , solv1 . model ())

solv2 = Solver ()
solv2.add(tabInit)
solv2.add(tabConfig2)
solv2.add(tabPhiSM2)

print("solv2_:_" , solv2 . check ())
if(solv2 . check () == sat ):
    print("model_:\"n\" , solv2 . model ())

solv3 = Solver ()
solv3.add(tabInit)
solv3.add(tabConfig3)
solv3.add(tabPhiSM3)

print("solv3_:_" , solv3 . check ())
if(solv3 . check () == sat ):
    print("model_:\"n\" , solv3 . model ())

```

It produces the following output :

```

solv1 :   sat
model :
[d0 = 1,
d2 = 4,
p1 = 1,
p0 = 0,
p2 = 1,
d1 = 0,
t2 = 0,
s2 = -1,
t1 = 0,
s1 = -1,
t0 = 0,
s0 = -1]
solv2 :   sat
model :
[d0 = 1,
d2 = 4,
p1 = 0,
p0 = 1,
p2 = 1,
d1 = 0,
t2 = 0,
s2 = -1,
t1 = 0,
s1 = -1,
t0 = 0,
s0 = -1]
solv3 :   sat
model :
[d0 = 1,
d2 = 4,
p1 = 0,
p0 = 0,
p2 = 4,

```

$$\begin{aligned}
d1 &= 0, \\
t2 &= 0, \\
s2 &= -1, \\
t1 &= 0, \\
s1 &= -1, \\
t0 &= 0, \\
s0 &= -1]
\end{aligned}$$

What it shows us is that every robot can move in this kind of configuration. The solver find a way to place the robot calling the strategy outside of the multiplicity and then allow the strategy to be true, meaning, the robot moves toward the multiplicity.

### 3 Gathering rigid configurations

Let  $d_{ij}$  be the value  $j$  of the view vector of the robot  $i$ , and  $ds_{ij}$  the value  $j$  of the symmetrical view of the robot  $i$ . The robot is calling the strategy  $\phi_R$ , here are all the logic formulas used in order to build  $\phi_R$ :

*AllView* is true if  $d_{00}, \dots, d_{k-1k-1}$  are all the views you can obtain from a single view vector  $dist_0, \dots, dist_{k-1}$ :

$$\begin{aligned}
AllView(dist_0, \dots, dist_{k-1}, d_{00}, \dots, d_{k-1k-1}) := \\
(\bigwedge_{i=0}^{k-1} (\bigwedge_{j=0}^{k-1} (d_{ij} = dist_{(j+i) \bmod k})))
\end{aligned}$$

*IsRigid* is true if the given configuration is a rigid configuration. Meaning, all the views are distinct, and so there is no multiplicity, the configuration isn't symmetric or periodic.

$$\begin{aligned}
IsRigid(d_{00}, \dots, d_{k-1k-1}, ds_{00}, \dots, ds_{k-1k-1}) := \\
\bigwedge_{i=0}^{k-1} (\bigwedge_{j=0}^{k-1} d_{ij} \neq 0) \wedge \\
\bigwedge_{i=0}^{k-1} (\bigwedge_{l=0}^{k-1} l \neq i ((\bigvee_{j=0}^{k-1} d_{ij} \neq d_{lj}) \wedge (\bigvee_{j=0}^{k-1} d_{ij} \neq ds_{lj}) \\
\wedge (\bigvee_{j=0}^{k-1} ds_{ij} \neq d_{lj}) \wedge (\bigvee_{j=0}^{k-1} ds_{ij} \neq ds_{lj})))
\end{aligned}$$

*AllCode* is true if  $(\alpha'_r, \beta'_r)$  is the set of two natural numbers of the robot  $r$  such as  $\alpha'_r$  and  $\beta'_r$  are codes of  $r$ 's views, with  $\alpha'_r < \beta'_r$ .

$$\begin{aligned}
AllCode(d_{00}, \dots, d_{k-1k-1}, ds_{00}, \dots, ds_{k-1k-1}, \alpha_0, \dots, \alpha_{k-1}, \beta_0, \dots, \beta_{k-1}, \\
\alpha'_0, \dots, \alpha'_{k-1}, \beta'_0, \dots, \beta'_{k-1}) := \\
\bigwedge_{i=0}^{k-1} (\alpha'_i < \beta'_i \wedge (\alpha'_i = \alpha_i \vee \alpha'_i = \beta_i) \wedge (\beta'_i = \alpha_i \vee \beta'_i = \beta_i)) \wedge \\
((\alpha_0 < \alpha_1 < \dots < \alpha_{k-1} < \beta_0 < \dots < \beta_{k-1}) \wedge \\
(\bigvee_{p=0}^{k-1} (\bigwedge_{q=0}^{p-1} (d_{0q} = d_{1q}) \wedge d_{0p} > d_{1p})) \wedge \dots \wedge \\
(\bigvee_{p=0}^{k-1} (\bigwedge_{q=0}^{p-1} (ds_{(k-2)q} = ds_{(k-1)q}) \wedge ds_{(k-2)p} > ds_{(k-1)p}))) \\
\vee \\
((\alpha_0 < \alpha_2 < \alpha_1 < \dots < \alpha_{k-1} < \beta_0 < \dots < \beta_{k-1}) \wedge \dots) \vee \dots
\end{aligned}$$

*CodeMaker* is true if the configuration is rigid and if  $(a_0, \dots, a_{k-1}, as_0, \dots, as_{k-1})$  are each code of each view passed as a parameter:

$$\begin{aligned}
CodeMaker(d_{00}, \dots, d_{k-1k-1}, ds_{00}, \dots, ds_{k-1k-1}, a_0, \dots, a_{k-1}) := \\
IsRigid(d_{00}, \dots, d_{k-1k-1}, ds_{00}, \dots, ds_{k-1k-1}) \wedge \\
\exists \alpha_0, \dots, \alpha_{k-1}, \beta_0, \dots, \beta_{k-1}, \alpha'_0, \dots, \alpha'_{k-1}, \beta'_0, \dots, \beta'_{k-1}, \\
AllCode(d_{00}, \dots, d_{k-1k-1}, ds_{00}, \dots, ds_{k-1k-1}, \alpha_0, \dots, \alpha_{k-1}, \beta_0, \dots, \beta_{k-1}, \\
\alpha'_0, \dots, \alpha'_{k-1}, \beta'_0, \dots, \beta'_{k-1}) \\
(\bigwedge_{i=0}^{k-1} (\bigwedge_{j=0, j \neq i}^{k-1} ((a_i > a_j \wedge \alpha'_j > \alpha'_i) \vee (a_i < a_j \wedge \alpha'_j < \alpha'_i)))) \\
\bigwedge_{i=0}^{k-1} (\bigwedge_{j=0, j \neq i}^{k-1} a_i \neq a_j)
\end{aligned}$$

*FindMax* is true if *Max* is the highest value of the view vector passed as a parameter:

$$\begin{aligned}
FindMax(dist_0, \dots, dist_{k-1}, Max) := \\
(\bigwedge_{i=0}^{k-1} (Max \geq dist_i) \wedge (\bigvee_{i=0}^{k-1} (Max = dist_i)))
\end{aligned}$$

$FindM$  is *true* if  $M$  is the index of the robot (index in the view vector) which has the largest code of view and a neighboring robot at distance  $Max$  :

$$\begin{aligned} FindM(d_{00}, \dots, d_{k-1k-1}, a_0, \dots, a_{k-1}, Max, M) := \\ \exists r \in [0; k-1], (d_{r0} = Max \vee d_{rk-1} = Max) \wedge \\ (\bigwedge_{i=0}^{k-1} ((a_r \geq a_i \wedge (d_{i0} = Max \vee d_{ik-1} = Max)) \\ \vee (d_{i0} < Max \wedge d_{ik-1} < Max))) \wedge M = r \end{aligned}$$

$FindN$  is *true* if  $N$  is the index of the robot (index in the view vector) with the largest code of view and  $M$  as a neighboring robot at distance  $Max$  :

$$\begin{aligned} FindN(d_{00}, \dots, d_{k-1k-1}, a_0, \dots, a_{k-1}, as_0, \dots, as_{k-1}, Max, M, N) := \\ \exists (d'_0, \dots, d'_{k-1}), (d''_0, \dots, d''_{k-1}), \\ (\bigwedge_{i=0}^{k-1} ((d'_i = d_{M((i+1) \bmod k)}) \wedge (d''_i = d_{M((i-1) \bmod k)}))) \wedge \\ \exists r_\alpha, r_\beta \in [0; k-1], \bigvee_{i=0}^{k-1} (((\bigwedge_{j=0}^{k-1} (d'_j = d_{ij})) \vee (\bigwedge_{j=0}^{k-1} (d'_j = ds_{ij}))) \wedge r_\alpha = i) \wedge \\ (\bigvee_{j=0}^{k-1} (((\bigwedge_{j=0}^{k-1} (d''_j = d_{ij})) \vee (\bigwedge_{j=0}^{k-1} (d''_j = ds_{ij}))) \wedge r_\beta = j)) \wedge \\ ((a_{r_\alpha} > as_{r_\alpha} \wedge a_{r_\alpha} > a_{r_\beta} \wedge a_{r_\alpha} > as_{r_\beta} \wedge d_{r_\alpha k-1} = Max \wedge d_{r_\beta 0} = Max \wedge N = r_\alpha) \vee \\ (a_{r_\beta} > as_{r_\beta} \wedge a_{r_\beta} > a_{r_\alpha} \wedge a_{r_\beta} > as_{r_\alpha} \wedge d_{r_\beta 0} = Max \wedge d_{r_\alpha k-1} = Max \wedge N = r_\beta) \vee \\ (d_{r_\alpha k-1} = Max \wedge d_{r_\beta 0} \neq Max \wedge N = r_\alpha) \vee \\ (d_{r_\alpha k-1} \neq Max \wedge d_{r_\beta 0} = Max \wedge N = r_\beta)) \end{aligned}$$

$\phi_R$  is *true* if the robot can move toward a multiplicity. There must be only one multiplicity and the configuration must be rigid :

$$\begin{aligned} \phi_R(dist_0, \dots, dist_{k-1}) := \\ \exists d_{00}, \dots, d_{k-1k-1}, AllView(dist_0, \dots, dist_{k-1}, d_{00}, \dots, d_{k-1k-1}) \wedge \\ \exists ds_{00}, \dots, ds_{k-1k-1}, \bigwedge_{i=0}^{k-1} (ViewSym(d_{i0}, \dots, d_{ik-1}, ds_{i0}, \dots, ds_{ik-1})) \wedge \\ \exists Max, M, N, (a_0, \dots, a_{k-1}), \\ CodeMaker(d_{00}, \dots, d_{k-1k-1}, ds_{00}, \dots, ds_{k-1k-1}, a_0, \dots, a_{k-1}) \wedge \\ FindMax(dist_0, \dots, dist_{k-1}, Max) \wedge \\ FindM(d_{00}, \dots, d_{k-1k-1}, a_0, \dots, a_{k-1}, Max, M) \wedge \\ FindN(d_{00}, \dots, d_{k-1k-1}, a_0, \dots, a_{k-1}, Max, M, N) \wedge \\ \exists dm_0, \dots, dm_{k-1}, dn_0, \dots, dn_{k-1}, \bigwedge_{j=0}^{k-1} (dm_j = (\sum_{l=0}^j dm_l) \wedge dn_j = (\sum_{l=0}^j dn_l)) \wedge \\ ((\bigvee_{j=0}^{k-1} (dn_j < dm_j \wedge (\bigwedge_{i=0}^{k-1} d_{Ni} = dist_i))) \vee (\bigvee_{j=0}^{k-1} (dm_j < dn_j \wedge (\bigwedge_{i=0}^{k-1} d_{Mi} = dist_i)))) \end{aligned}$$

## 4 Gathering an odd number of robots

We are now building a strategy,  $\phi_{ON}$ , that will gather an odd number of robots on a non-periodic configuration. It is the strategy with the lowest priority, it means that the configuration won't be rigid and won't have any multiplicity.

First we build the formula,  $IsOddNonPeriodic$ , that will return *true* if the number of robots is odd and if the configuration is non-periodic :

$$\begin{aligned} IsOddNonPeriodic(dist_0, \dots, dist_{k-1}) := \\ ((k+1) \bmod 2 = 0) \wedge \\ \exists p \in [0; \frac{k-1}{2}] \\ \exists d'_0, \dots, d'_{p-1}, \bigwedge_{i=0}^{k-1} (d'_{i \bmod p} = dist_i) \wedge dist_{k-1} \neq d'_{p-1} \end{aligned}$$

Now, we build the  $\phi_{OD}$  strategy, it returns *true* if the configuration is non-rigid, non-periodic, has no multiplicity and has an odd number of robots. The robot just move in order to create a multiplicity or a rigid configuration that will lead to a rigid configuration or one with a multiplicity.

$$\begin{aligned} \phi_{ON}(dist_0, \dots, dist_{k-1}) := \\ \neg IsRigid(dist_0, \dots, dist_{k-1}) \wedge \\ IsOddNonPeriodic(dist_0, \dots, dist_{k-1}) \wedge \\ (\bigwedge_{i=0}^{k-1} dist_i \neq 0) \end{aligned}$$

## 5 Tests

In order to test the algorithm [2] we will use the python code we show you at the beginning : *InitSM* and *phiSM*.

We will use the SAT-solver to test different configurations. We will change the number of robots and the size of the ring from a test to an other.

### 5.1 Test *InitSM*

First, we test the function *InitSM* alone : can we have an initial configuration with a single multiplicity with those parameters ?

nb-robot \size-ring	2	3	4	5	6
2	Unsat	Unsat	Unsat	Unsat	Unsat
3	Sat	Sat	Sat	Sat	Sat
4	Sat	Sat	Sat	Sat	Sat
5	Sat	Sat	Sat	Sat	Sat
6	Sat	Sat	Sat	Sat	Sat

The results make sense : we can't create a multiplicity with 2 robots which is not a winning configuration. Else, even on a ring size of 2 we can have a multiplicity on one spot and only one robot on the other spot.

### 5.2 Test $\phi_{SM}$

Now we test  $\phi_{SM}$  through the algorithm [2], we also use the function *InitSM* that makes sure we have a single multiplicity at the beginning.

nb-robot \size-ring	2	3	4	5	6
3	Timeout	Timeout	Timeout	...	...
4	Timeout	...	...	...	...
5	...	...	...	...	...
6	...	...	...	...	...

Same test but with the function *Init* instead.

nb-robot \size-ring	2	3	4	5	6
3	Timeout	Loose	Loose	Loose	...
4		...	...	...	...
5	...	...	...	...	...
6	...	...	...	...	...

For  $nb_{robot} = 3$  and  $size_{ring} = 2$  we face this problem :

Traceback (most recent call last):

File "algov5.py", line 56, in <module>

```
Ip = tree_interpolant(And(Interpolant(And(tmpAndInterpolant)),
And(tmpAndContext)))
```

File "/usr/lib/python3.8/site-packages/z3/z3.py", line 8297,  
in tree\_interpolant

```
res = Z3_compute_interpolant(ctx.ref(), f.as_ast(), p.params, ptr, mptr)
```

File "/usr/lib/python3.8/site-packages/z3/z3core.py", line 4074,  
in Z3\_compute\_interpolant

```
_elems.Check(a0)
```

File "/usr/lib/python3.8/site-packages/z3/z3core.py", line 1336, in Check

```
        raise self.Exception(self.get_error_message(ctx, err))

z3.z3types.Z3Exception: b'theory not supported by interpolation or bad proof'
```



```

1  foreach synchronous winning strategy f do
2       $k = 1$ ;
3      while true do
4           $I(c) = \text{Init}(c)$ ;
5           $\text{continue} = \text{true}$ ;
6          while  $\text{continue}$  do
7               $I' = \text{Interpolant}(I(c) \wedge \text{Post}(c, c1), \text{Post}(c1, c2) \wedge \dots \wedge$ 
8                   $\text{Post}(c_{k-1}, c_k) \wedge \text{BouclePerdante}(c_k))$ ;
9              if  $I' = \text{SAT}$  then
10                 if  $I = \text{Init}$  then
11                      $\text{exit}$ ;
12                      $k = k + 1$ ;
13                      $\text{continue} = \text{false}$ ;
14                 end
15                 else
16                     if  $I' \implies I$  then
17                          $\text{exit}$ ;
18                         else
19                              $I = I \vee I'$ ;
20                         end
21                     end
22                 end
23             end
24 end

```

## References

- [1] Ralf Klasing, Euripides Markou, and Andrzej Pelc. *Gathering asynchronous oblivious mobile robots in a ring*. Tech. rep. RR-1422-07. UMR 5800 - Université Bordeaux 1, 351, cours de la Libération, 33405 Talence CEDEX, France: Laboratoire Bordelais de Recherche en Informatique, Jan. 2007.
- [2] Nathalie Sznajder and Souheib Baarir. *Algorithme d'accélération par interpolants. (French) [Acceleration Algorithm using an interpolant]*. Tech. rep. Laboratoire Informatique de Paris 6 (LIP6), Feb. 2022.