Methodologies for Discrete-Event Modeling and Simulation
Fall-2004

Battlefield Modeling and Simulation Using CD++

Rami Madhoun (100667489)
rmadhoun@sce.carleton.ca
December 10, 2004

Abstract

Cellular automata have been used as modeling and simulation technique for different natural and artificial systems. One of these systems is land battlefield where two or more armies engage in a fight and each one tries to defeat the other or defend its own land and property. Cell-DEVS was introduced to overcome one of the main limitations of cellular automata, which is the evaluation of all the cells synchronously. Hence, cell-DEVS (using CD++ toolkit) allows for more complex models to be defined with different time delays. The aim of this work is not to model all the aspects of real-life combat between two armies, since the system under consideration is highly non-linear. Rather, it aims to do two things; showing how complex –and sometimes unpredictable- behavior can emerge from relatively simple cell-DEVS rules. And, show how the performance of the model execution can be improved by using some of the new extensions to the CD++ toolkit.

Table of Contents

1. Introduction

Cell-DEVS is modeling technique that depends on splitting the system to into number of cells, each of which has a local computing function (rule) to calculate the next state of the cell. This function takes into consideration the current cell state and its neighbor's states as well, which are connected through input/output port. In this report, we try to introduce a model for a land battlefield using cell-DEVS formalism implemented with the CD++ toolkit. Two models will be considered here, one is an original model implemented using two different versions of CD++ and the second is an enhanced model incorporating more features. The main difference between the two CD++ versions is that the new one allows for having state variables and multiple input/output ports for each cell. First, we will discuss the system to be modeled and its characteristics and what assumptions are made on the system. Secondly, we will discuss briefly the implementation of the original model using the original CD++ (it was discussed in details in previous work) mentioning the disadvantages and limitations of the previous implementation. Then, the new CD++ features used to re-implement the original model will be discussed with performance analysis of the two implementations. Finally, some enhancement will be added to the models and implemented showing an overall performance analysis of the two models using the two versions of CD++.

2. System to be modeled

The system presented here is a land battlefield between two armies. Each army is composed of different soldiers and one flag. The main objective of each army is to attack/acquire the opponent's flag. To do so, the soldiers try to move towards the opponent's flag and engage in a fight if faced by enemy soldiers. The characteristics of the systems can be summarized as follows:

- A Two dimensional battlefield is considered without any airplanes or missiles.
- Each army is composed of different soldiers and one flag.
- The objective of each army is to acquire the opponent's flag.
- Each soldier can exist in one of three states: *alive*, *injured*, *dead*.
- The situation awareness of the soldier is limited to his neighborhood (no telecommunication equipment are used).
- If a soldier is in state *Alive*, and attacked by an enemy soldier, his state changes to *injured*.
- If a soldier is in state *injured* and is attacked by an enemy soldier, he becomes *dead*.
- The soldier's ability to fight is dependent on a randomly assigned factor (Fighting Ability FA). In addition, the *injured* soldier will have a less fighting ability than the *alive* one.
- *Injured* soldiers recover into *alive* state if not surrounded by enemy soldiers.
- If a soldier is not surrounded by enemy soldiers he tends to move towards the enemy's flag.
- If a soldier is surrounded by an enemy soldier/s, he engages in a fight. The outcome of this fight depends on the fighting ability (FA) of the soldiers engaged in the fight.
- The flag is acquired once an enemy soldier once an enemy soldier moves to its neighborhood.

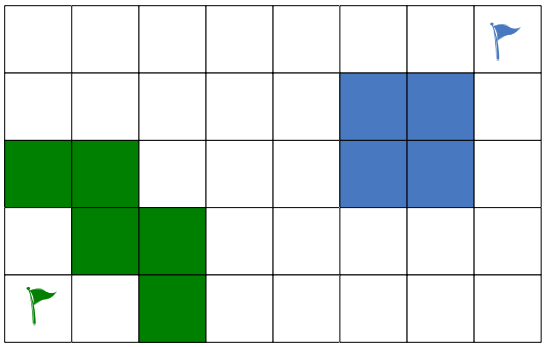


Figure-1 Possible troops allocation

3. Model definition

3.1Coupled model definition

M = < Xlist, Ylist, I, X, Y, η, N, {r, c}, C, B, Z, select >

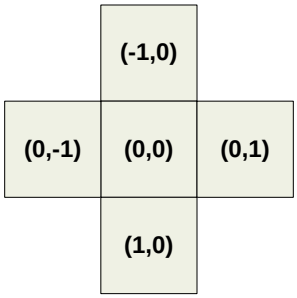


Figure-2Van Neumann neighborhood

Xlist = {Φ}
Ylist = {Φ}
I = <P^x, P^y>
$\mathbf{P^x}$ = Φ $\mathbf{P^y}$ = Φ
X = S (defined next)
Y = S (defined next)
η = 5
N = {(-1, 0), (0,-1), (0, 0), (0, 1), (1, 0)}
r = 10
c = 10

B = Φ (wrapped)
C = {C_{ij} / i ε [0, 9], j ε [0, 9]}

$P_{ij}^{Y1} \rightarrow P_{i,j+1}^{X1}$ $P_{i,j}^{X1} \leftarrow P_{i,j-1}^{Y1}$
$P_{ij}^{Y2} \rightarrow P_{i+1,j}^{X2}$ $P_{i,j}^{X2} \leftarrow P_{i-1,j}^{Y2}$
$P_{ij}^{Y3} \rightarrow P_{i,j-1}^{X3}$ $P_{i,j}^{X3} \leftarrow P_{i,j+1}^{Y3}$
$Pij^{Y4} \rightarrow P_{i-1,j}^{X4}$ $P_{i,j}^{X4} \leftarrow P_{i+1,j}^{Y4}$
$Pij^{Y5} \rightarrow P_{i,j}^{X5}$ $P_{i,j}^{X5} \leftarrow P_{ij}^{Y5}$

SELECT = {(-1,0), (0,-1), (0,0), (0,1), (1,0)}

3.2 Atomic model definition

CD = < *X*, *Y*, I, S, θ, N, d, δint, δext, τ, λ, D >

X = S
Y = S
I = < η, μ, P^x, P^y>
η = 5 , μ = 0
P^i_j = {(N_j^i, T_j^i) / ∀ j ∈ [1, 5], N_j^i ∈ [i_1, i_5] and T_j^i ∈ I*i*
}, I_i = {x / x ∈ *X* if *i* = X} or I_i = {x / x ∈ *Y* if *i* = Y}

S = {-2, -1, 0, 1, 2}
θ = {s, phase, σqueue, σ} **s** ∈ S, **phase**= {passive, active}
σqueue= { ((v1,σ1),...,(v_m,σ_m)) / m ∈ ***N*** ∧ m < ∞) ∧ ∀ (i ∈ ***N***, i ∈ [1,m]), vi ∈ V ∧ σi ∈ ***R***0 +∪ ∞}; (transport delay)
N= {(-1, 0), (0,-1), (0, 0), (0, 1), (1, 0)}
d= 100 (transport delay)
D = θ x N x d → R_0^+ ∪ ∞,

For δint, δext, τ, λ, please refer to the implementation section (section 5) and model definition files (battleX.ma, rules.inc)

4. Original model implementation using CD++

In this section, some of the implementation details will be discussed. Most of these details are common for both CD++ versions. Hence no distinction is made on whether the old or new CD++ version is used except when necessary.

4.1 Soldier/fighter status (FS)

The status of the soldier is represented as a signed integer to distinguish between the two armies (one will have positive values and the other will have negative ones). The following table describes this representation:

Status	Description
2	Fighter of army A alive
1	Fighter of army A injured
0	Fighter is dead and cell is empty
-1	Fighter of army B injured
-2	Fighter of army B alive
5	Flag of army A
-5	Flag of army B

Table-1 Soldier's status definition

4.2 Fighting ability (FA)

The fighting ability of each soldier is represented by a randomly assigned real number ranging from 0 to 1. With 0, represents no fighting ability at all (in the case of flag) and 1 with very high fighting ability. In addition, the soldier will have an effect on the enemy soldier if his fighting ability is greater than 0.5. The assignment is done using random function with a uniform distribution and is executed at two points:

- At the beginning of the battle
- After engaging in a fight with an enemy soldier

The following table describes the fighting ability factor:

Status	Fighting Ability
2	Uniformly distributed number between 0.45 → 1.0
1	Uniformly distributed number between 0.0 → 0.55
0	Fighter is dead and cell is empty 0.0
-1	Uniformly distributed number between 0.0 → 0.55
-2	Uniformly distributed number between 0.45 → 1.0
5	Does not engage in fights 0.0
-5	Does not engage in fights 0.0

Table-2 Fighting ability factors

When two or more soldiers engage in a fight, the outcome depends on the difference between their fighting abilities (FAs).

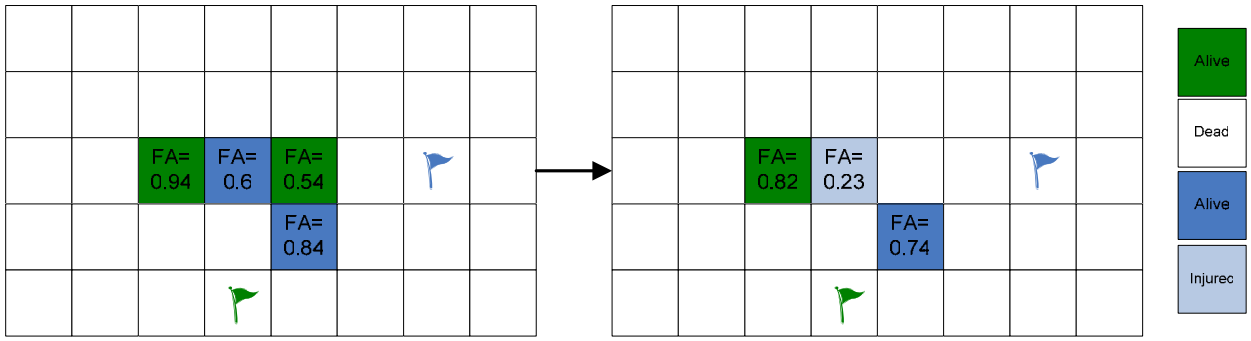


Figure-3 The effect of different FAs in a fight

4.3 Flag position

Since each soldier aims to acquire the enemy's flag, he needs to know about the flag position. This information is represented as a real number having the integer part representing the flag row number (y-coordinate) and the fractional part representing the flag column number (x-coordinate).

Row + Column/100 (ex. (row=2, column=4) → 2.04)

4.4 Moving directions

If the soldier is not surrounded by an enemy soldier, he tends to move towards enemy's flag. To do so, the soldier needs to calculate his direction in the next step to come closer to his target. This is done by comparing the current cell position of the solider with the enemy's flag position. For example, if the soldier is standing at cell (1, 1) and the enemy's flag position is at cell (3, 4); he will have two options, either to move to the east or to the south.

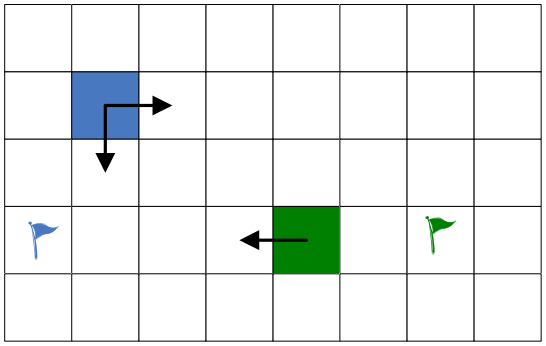


Figure-4 Movement directions

After deciding on the direction of the next step, the directions are assigned integer values according to the following table:

Direction	Value
North	10
East	20
South	30
West	40

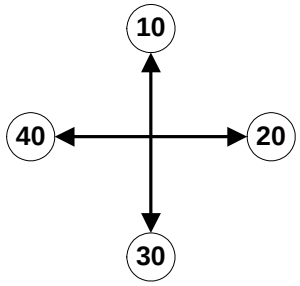


Table-3 Movement directions definition

4.5 Free-cell move-in factor

This is an integer number that is calculated for every free cell to resolve any conflict if two or more soldiers (whether enemies or friends) want to move to the same free cell.

4.5.1 Original implementation of the free-cell move-in factor

In the original implementation, this factor is evaluated as the maximum fighting ability of the soldiers surrounding the free cell. The following figure illustrates thus point.

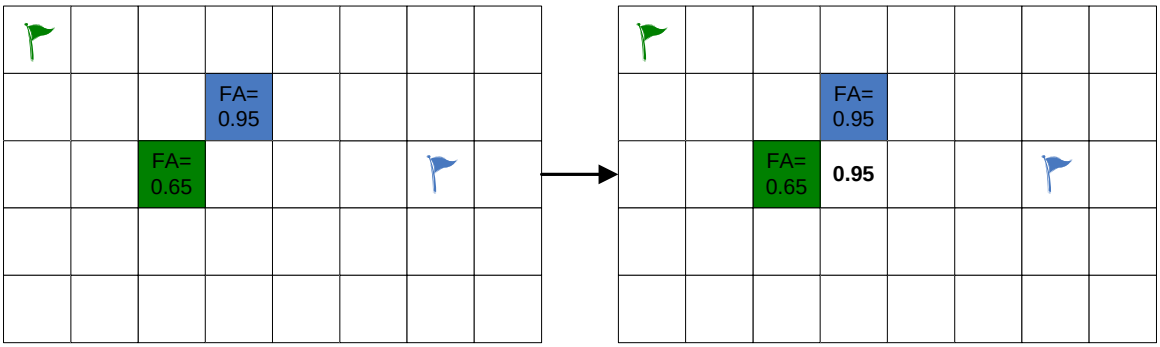


Figure-5 Free-cell move-in factor calculation (original implementation)

4.5.2 New implementation of the free-cell move-in factor

In the new implementation, this factor is calculated by searching for the maximum fighting ability (FA) of the soldiers in the neighborhood who **intend** to move to the cell. And only the one with the maximum FA will be allowed to move to the free cell. In this scenario, the free-cell move-in factor will be the direction of that soldier (the one with maximum FA) with an opposite sign to indicate that the cell will be occupied by the soldier coming from that direction. The following figure illustrates thins point.

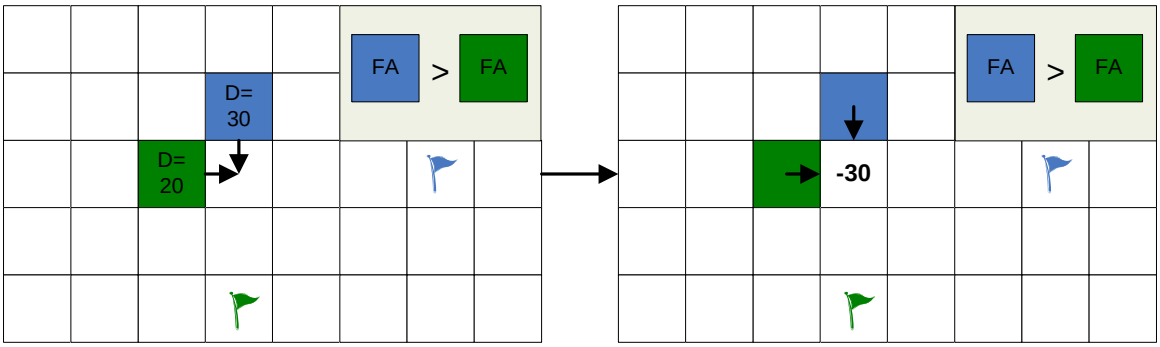


Figure-6 Free-cell move-in factor calculation (new implementation)

5. Model implementation using original CD++

The model was implemented using the original version of CD++ which only allows for one input/output port between the cell and each of its neighbors without the ability to define any variables within the cell. This has introduced some limitations when trying to simulate cells with more than one variable or input/output port. Hence, in order to overcome this limitation, each piece of information was implemented using different layer which resulted in 3-dimensional cell space to simulate the original 2-dimensional model.

5.1 Battlefield model layers

The model was implemented using the following six layers:

- Layer 0, is used to store the soldier status and allocation in the battlefield.
- Layer 1, is user to store the fighting ability factor (FA), which is used for movement and fighting rules evaluation
- Layer 2, is used to store the flag position of army B. This information is need for the all the soldiers of army A to calculate the next movement direction.
- Layer 3, is used to store the flag position of army A. This information is need for the all the soldiers of army B to calculate the next movement direction.
- Layer 4, is used to store the movement directions of each soldier.
- Layer 5, is used to store the move-in factor associated with each free cell.

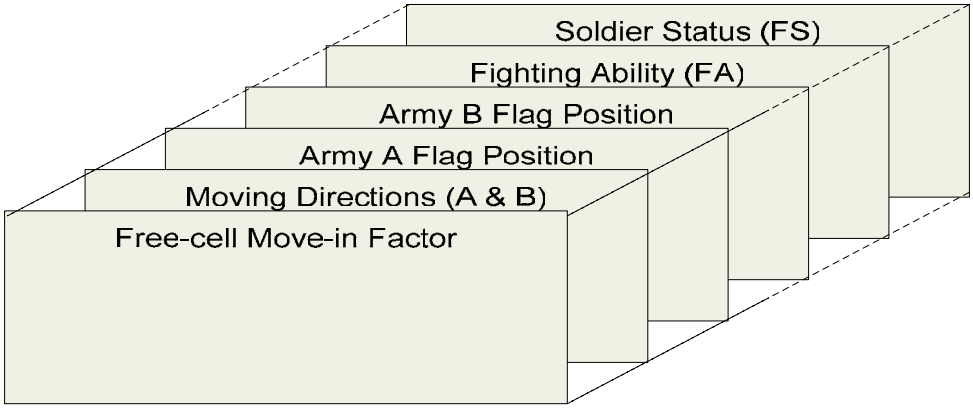


Figure-7 Cell space definition (original implementation)

In this case, each layer had some of the functionality associated with the model. For examples, the soldier allocation layer (layer 0) had the rules for evaluating the fighting outcome between soldiers; fighting ability layer (layer 1) had the rules to re-assign the fighting ability (FA) factors once the fighter status changes, and so on.

5.2 Disadvantages and limitations of the original implementation

- Each piece of information is implemented using different layer which introduces some extra complexity to the original model.
- 3-dimensional neighborhood is used which slows down the model execution
- Rule definition and debugging is time consuming
- Model expansion is difficult due to the multi-layer approach followed.

6. New CD++ extensions

As discussed earlier, the limitations in the initial version of CD++ introduces some difficulties in defining and executing complex cell-DEVS models with different time delays. Hence, a new version of CD++ was developed to overcome these limitations. The new CD++ extensions include:

- The ability to define multiple input/output ports for each cell in the cell space.
- The ability to define state variables within the cell.

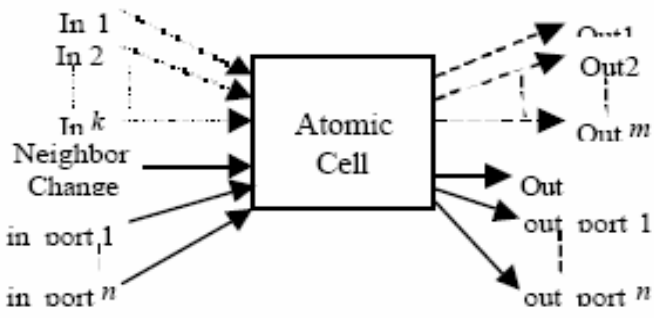


Figure-8 Multi-port cell

The input/output ports are connecting each cell to all of its neighboring cells, so it is useful to represent information that need to be transferable between different cells. However, the state variables are local to the cell, and are useful to represent any variable that does not need to be referenced from outside the cell. Both of these features were used to re-implement the original battlefield model dispensing with the need to define extra layer of cells to store new kind of information. One disadvantage of the new CD++ toolkit is that it does not allow for initial values assignment for each specific port in the cell. For example, if cell (3, 3) is initialized to the value 5, all the associated ports will have this value and hence some initialization rules need to be implemented to take care of this issue.

It is worth mentioning here that additional input/output ports will increase the messages exchanged between the cells and should be used with caution especially if the model is meant to be executed in a parallel environment.

7. Model implementation using new CD++ extensions

The original battlefield was re-implemented taking advantage of the new CD++ features. In this section, we will present the detailed implementation of the model with explanation of some of the rules used to define the model.

The model consists of two-dimensional cell space with the following input/output ports:

- **FS**: is used to represent the soldier status (i.e. alive, injured, dead)
- **FA**: is used to represent the fighting ability of the solider
- **Enemy_Flag**: is the location of the enemy flag represented in the same format explained in section 4.3.
- **Direction**: is used to represent the direction of the next move of the soldier (section 4.4).

In this implementation, no state variables were used due to the fact the all the information associated with the soldier need to be transferable once he moves from one cell to another.

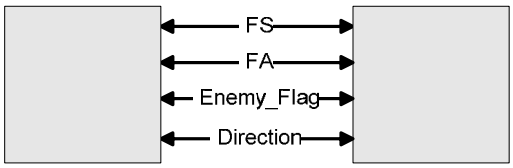


Figure-9 Multi-port connectivity between two cells

7.1 Initialization rules

These rules are used to initialize the cell ports to the proper values as CD++ does not distinguish between ports when reading the initial values file.

```
rule : { ~fs := trunc((0,0)~fs)+1 ; ~fa := abs(fractional((0,0)~fa)) ;
~direction := 0; ~enemy_flag := 1.01 ; }  0
            {  fractional((0,0)~fs) != 0 and (0,0)~fs < 0 }

rule : { ~fs := trunc((0,0)~fs) ; ~fa := abs(fractional((0,0)~fa)) ;
~direction := 0 ; ~enemy_flag := 8.08 ; }  0
            {  fractional((0,0)~fs) != 0 and (0,0)~fs >= 0 }

rule : { ~fs := 5 ; ~fa := 0; ~direction := 0;  } 0
                                    { (0,0)~fs = 5  and (0,0)~fa = 5  }

rule : { ~fs := -5 ; ~fa := 0; ~direction := 0; } 0
                                    { (0,0)~fs = -5 and (0,0)~fa = -5 }
```

The first two rules initialize the FS and FA of the soldier to the values assigned through the "battle.val" file. The integer part of the number is assigned to the FS factor and the fractional part is assigned to the FA factor. In addition, the enemy_flag value is assigned directly by editing the model definition file "battleX.ma". The next two rules reset FA and Direction values to zero for the flags of both armies since they are not going to engage in any fight.

7.2 Fighting rules

These rules assign the soldier status and his fighting ability after engaging in a fight.

```
rule : { ~fs:= 1 ; ~fa:= uniform(0,0.55) ; ~direction := 0 ;  } 100
{  (0,0)~fs = 1 and ( statecount(-1, ~fs) + statecount(-2, ~fs) ) > 0
and (#macro(fight_rule_1)) = 0  }

rule : { ~fs:= 0 ;~fa:= 0 ; ~direction := 0 ; ~enemy_flag := -1 ; } 100
{  (0,0)~fs = 1 and ( statecount(-1, ~fs) + statecount(-2, ~fs) ) > 0
and  (#macro(fight_rule_1)) < 0  }

rule : { ~fs:= 2 ; ~fa:= uniform(0.45,0.99) ; ~direction := 0; } 100
{  (0,0)~fs = 2 and  ( statecount(-1, ~fs) + statecount(-2, ~fs) ) > 0
and (#macro(fight_rule_1)) = 0  }

rule : { ~fs:= 1 ; ~fa:= uniform(0,0.55) ; ~direction := 0 ;  }  100
{ (0,0)~fs = 2 and ( statecount(-1, ~fs) + statecount(-2, ~fs) ) > 0
and  (#macro(fight_rule_1)) = -1 }

rule : { ~fs:= 0 ; ~fa:= 0 ; ~direction := 0 ; ~enemy_flag := -1; } 100
{  (0,0)~fs = 2 and ( statecount(-1, ~fs) + statecount(-2, ~fs) ) > 0
and  (#macro(fight_rule_1)) < -1 }
```

```
#BeginMacro(fight_rule_1)
(
if( ((-1,0)~fs = -1 or (-1,0)~fs = -2) and (-1,0)~fa > 0.5 and
  ((-1,0)~fa > (0,0)~fa) , -1, 0) +
if( ((0,-1)~fs = -1 or (0,-1)~fs = -2) and (0,-1)~fa > 0.5 and
  ((0,-1)~fa > (0,0)~fa), -1, 0)  +
if( ((0,1)~fs = -1 or (0,1)~fs = -2) and (0,1)~fa > 0.5 and
  ((0,1)~fa > (0,0)~fa), -1, 0) +
if( ((1,0)~fs = -1 or (1,0)~fs = -2) and (1,0)~fa > 0.5 and
  ((1,0)~fa > (0,0)~fa) , -1, 0)
)
#EndMacro
```

The macro "fight_rule_1" checks if the solider (from army A) is neighborhood by any enemy (army B) with a higher fighting ability and adds (-1) to the overall value of the macro for each such solider. The number generated by fight_rule_1 is used in the main body of the rule to evaluate the following conditions:

- If an A soldier is injured (FS=1) and is surrounded by enemy soldiers whose fighting ability are less than his, he will remain injured but will be assigned a new fighting ability.
- If an A soldier is injured (FS=1) and is surrounded by enemy soldiers whose fighting ability are higher than his, he will be dead and his fighting ability is assigned the value 0.
- If an A soldier is alive (FS=2) and is surrounded by enemy soldiers whose fighting ability are less than his, he will remain alive and assigned new fighting ability factor.
- If an A soldier is alive (FS=2) and is surrounded by enemy soldiers and only one of them has a higher fighting ability, he will be injured and assigned new fighting ability factor.
- If an A soldier is alive (FS=2) and is surrounded by enemy soldiers and more than one of them has a higher fighting ability, he will be dead and his fighting ability factor becomes zero.

The same rule is used for B soldiers when surrounded by an A army soldiers by changing the corresponding soldier status values.

7.3 Flags-under-attack rules

These rules are used to evaluate the flag's next status after being attacked.

```
rule : { ~fs := if ( (#macro(fight_rule_3)) != 0, 0 , 5 ) ; ~fa := 0 ; }
100 { (0,0)~fs = 5 and (statecount(-1,~fs) + statecount(-2,~fs)) > 0 }
```

```
#BeginMacro(fight_rule_3)
(
if( ((-1,0)~fs = -1 or (-1,0)~fs = -2) and (-1,0)~fa > 0 ,1,0)  +
if( ((0,-1)~fs = -1 or (0,-1)~fs = -2) and (0,-1)~fa > 0 ,1,0) +
if( ((0,1)~fs = -1 or (0,1)~fs = -2) and (0,1)~fa > 0 ,1,0)  +
if( ((1,0)~fs = -1 or (1,0)~fs = -2) and (1,0)~fa > 0 ,1,0)
)
#EndMacro
```

The "fight_rule_3" macro is used to check whether an A flag (FS= 5) is surrounded by enemy fighters, and is used within the main rule body to assign a values of zero to the status of the flag in such case. A similar rule is used to evaluate the status of B's flag.

7.4 Flags-not-attacked rule

If the flags are not attacked, they will simply retain their current status.

```
rule : { ~fs := 5 ; ~fa := 0 ; } 100 {  (0,0)~fs = 5 and
(statecount(-1, ~fs) + statecount(-2, ~fs)) = 0 }
rule : { ~fs := -5 ; ~fa := 0 ;  } 100 { (0,0)~fs = -5 and
(statecount(1, ~fs) + statecount(2, ~fs)) = 0 }
```

7.5 Injured-soldiers-not-attacked rule

If injured soldiers are not surrounded by enemy soldiers, they will recover and become alive again and will be assigned new fighting ability factor.

```
rule : { ~fs := 2 ; ~fa:= uniform(0.45,0.99) ; }  100
{  (0,0)~fs = 1  and (statecount(-1, ~fs) + statecount(-2, ~fs)) = 0 }

rule : { ~fs := -2 ; ~fa:= uniform(0.45,0.99); } 100
{  (0,0)~fs = -1 and (statecount(1, ~fs) + statecount(2, ~fs)) = 0 }
```

7.6 Movement-direction rules

These rules are used to calculate the direction of the soldier's next move depending on his current location and the enemy's flag position.

```
rule : {~fs := 2 ;~fa := (0,0)~fa ;~direction :=(#macrc(director_row));}
0 {  (0,0)~fs = 2 and (statecount(-1, ~fs) + statecount(-2, ~fs)) = 0
 and (0,0)~direction = 0 and (0,0)~enemy_flag >= 0 and
 cellPos(0) = trunc((0,0)~enemy_flag ) and cellPos(1) !=
 round(fractional((0,0)~enemy_flag) * 100)   }

rule : { ~fs := 2 ; ~fa := (0,0)~fa ; ~direction :=
(#macrc(director_column));} 0
{ (0,0)~fs = 2 and (statecount(-1,~fs) + statecount(-2,~fs)) = 0 and
(0,0)~direction =0 and
  (0,0)~enemy_flag >= 0 and cellPos(0) != trunc((0,0)~enemy_flag) and
   cellPos(1) = round(fractional((0,0)~enemy_flag) * 100) }

rule : { ~fs := 2; ~fa := (0,0)~fa ; ~direction :=
(#macrc(director_row_column)) ;} 0
 { (0,0)~fs = 2 and (statecount(-1, ~fs) + statecount(-2, ~fs)) = 0 and
 (0,0)~direction = 0 and
 (0,0)~enemy_flag >= 0 and cellPos(0) != trunc((0,0)~enemy_flag) and
 cellPos(1) != round(fractional((0,0)~enemy_flag) * 100) }
```

```
#BeginMacro(director_row)
(
 if ( cellPos(1) > round(fractional((0,0)~enemy_flag) * 100)  , 40,20)
)
#EndMacro
```

```
#BeginMacro(director_column)
(
 if (cellPos(0) > trunc((0,0)~enemy_flag) , 10,30)
)
#EndMacro
```

```
#BeginMacro(director_row_column)
(
if( random > 0.5 , if( cellPos(0) > trunc((0,0)~enemy_flag), 10,30),
if( cellPos(1) > round(fractional((0,0)~enemy_flag) * 100), 40,20) )
)
#EndMacro
```

- The first rule and the first macro check whether the soldier and the enemy flag are on the same row, then it assigns a direction to the right or left depending on the position of both.
- The second rule and the second macro check whether the soldier and the enemy flag are on the same column, then it assigns a direction going up or down depending on the position of both.
- The last rule and macro check if the soldier and enemy flag are on different rows and columns, then it randomly select a direction whether to move horizontally or vertically.

7.7 Movement rules

The movement of soldiers is implemented in two steps. First step is to calculate the move-in factor for each free cell and then implement the actual movement of the soldier.

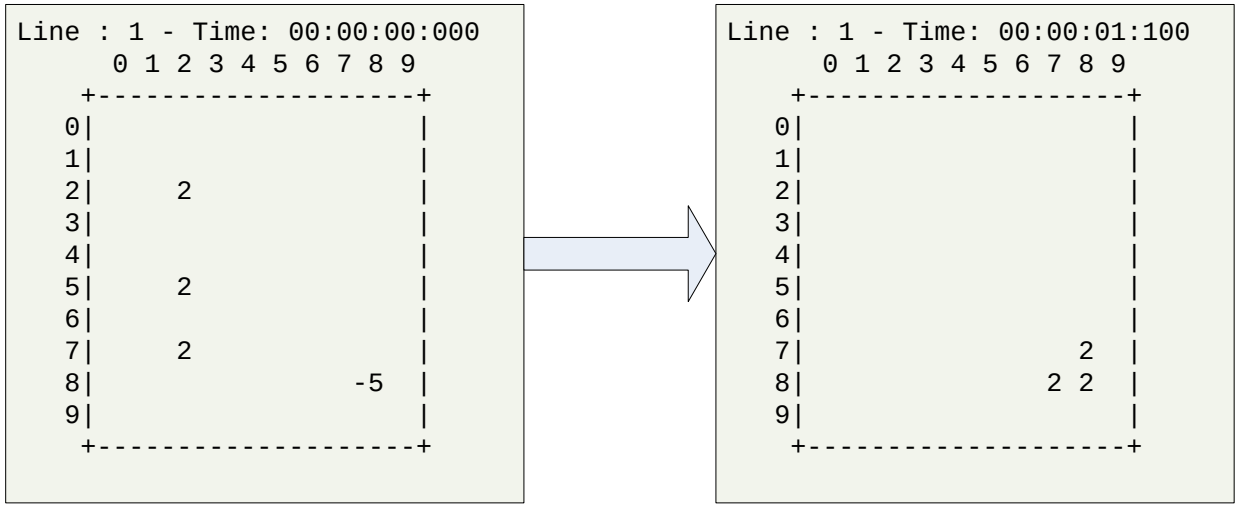
```
#BeginMacro(move_from_west_factor)
rule : {~direction := if ((0,-1)~direction = 20, -20,(0,0)~direction); }
100
 { (0,0)~fs = 0 and (0,0)~direction = 0   and
  (0,-1)~fa > if ((-1,0)~direction = 30, (-1,0)~fa, 0)   and
  (0,-1)~fa > if( (0,1)~direction = 40 , (0,1)~fa, 0) and
  (0,-1)~fa > if ( (1,0)~direction = 10, (1,0)~fa,0)  }
#EndMacro

#BeginMacro(move_from_west)
rule  : { ~fs := (0,-1)~fs ; ~fa := (0,-1)~fa ; ~direction :=0;
~enemy_flag := (0,-1)~enemy_flag ;  }  0
{  (0,0)~fs = 0 and  ( (0,-1)~fs = 2 or (0,-1)~fs = -2) and
   (0,-1)~direction = 20 and (0,0)~direction = -20 }

rule : {~fs := 0 ; ~fa := 0 ; ~direction := 0 ; ~enemy_flag := -1; } 0
{ ((0,0)~fs = 2 or (0,0)~fs = -2)  and  (0,1)~fs = 0 and
  (0,0)~direction = 20 and (0,1)~direction = -20  }
#EndMacro
```

The first rule checks to see if a free cell is neighbored by a soldier who intends to move to the east direction and has the maximum fighting ability. If this is the case, the direction of the free cell becomes the same direction of the soldier with an opposite sign (i.e. -20). The second and third rules, moves the soldier attributes (status, fighting ability…etc) to the new cell and resets these attributes of the cell originally occupied by the soldier. The same principle is followed for the other directions.

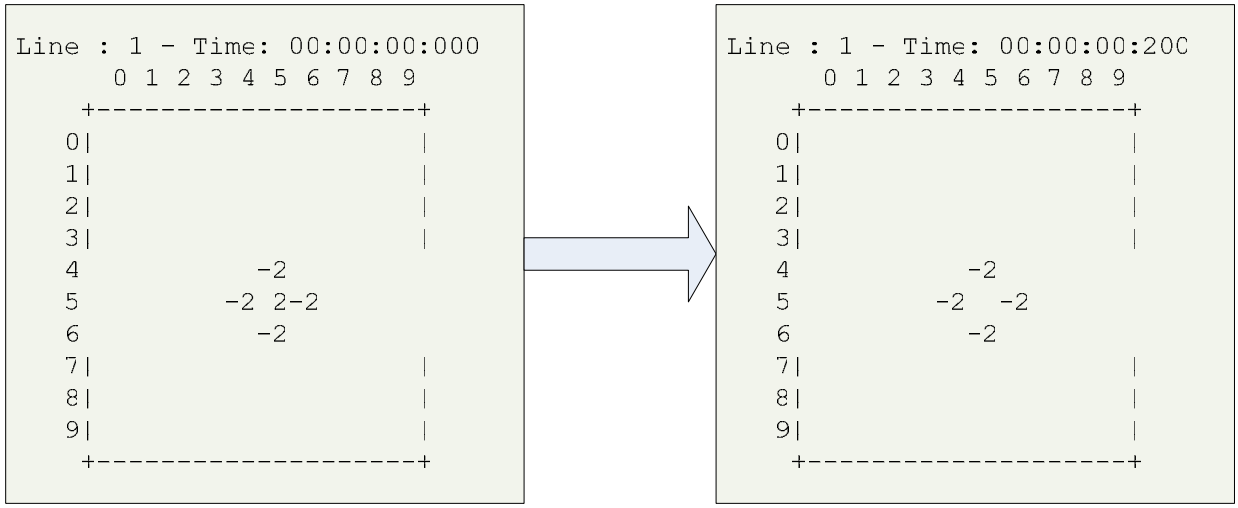
8. Testing strategy

The testing strategy followed depends on testing the model with different scenarios, each activating some specific rule/s and then testing the overall model with a scenario that activates all of the rules simultaneously. Three scenarios were used to test the model behavior

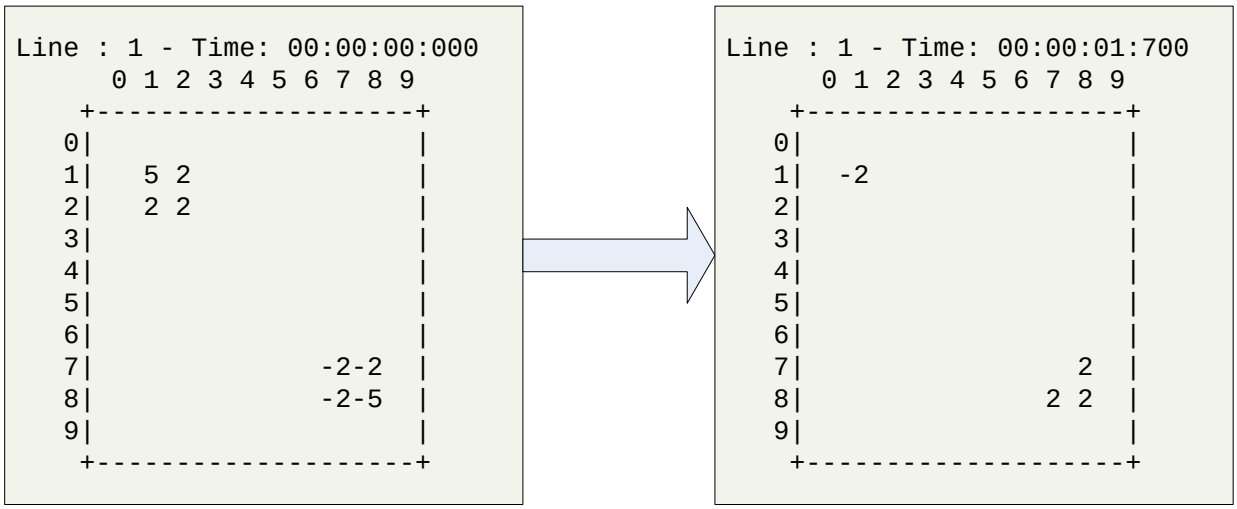
- Movement rules, in this scenario, only the movement rules are activated as the soldiers of army A move towards and acquire the B flag. The detailed output of this scenario is available in the file “battle1.drw” within the “battlefield_original.new_CD++” folder.



- Fighting rules, in this scenario the fighting rules are activated when the soldiers of both armies engage in a fight. The detailed output of this scenario is available in the file "battle2.drw" within the "battlefield_original.new_CD++" folder.



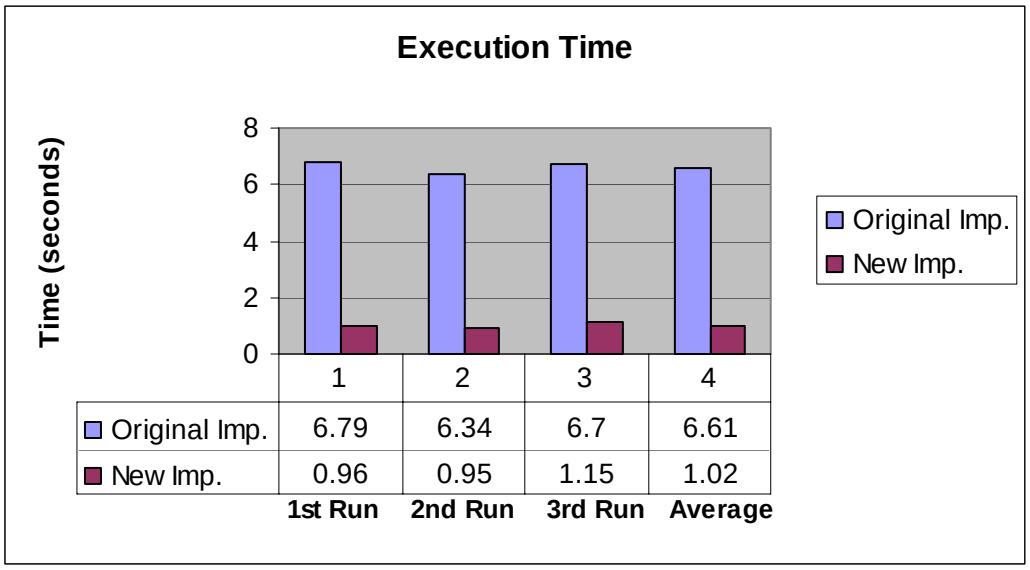
- In the last scenario, all the rules are activated to test the overall behavior of the model. The detailed output of this scenario is available in the file "battle3.drw" within the "battlefield_original.new_CD++" folder.



As shown in the previous scenarios, the model behaves exactly as expected which proves the correctness of the model.

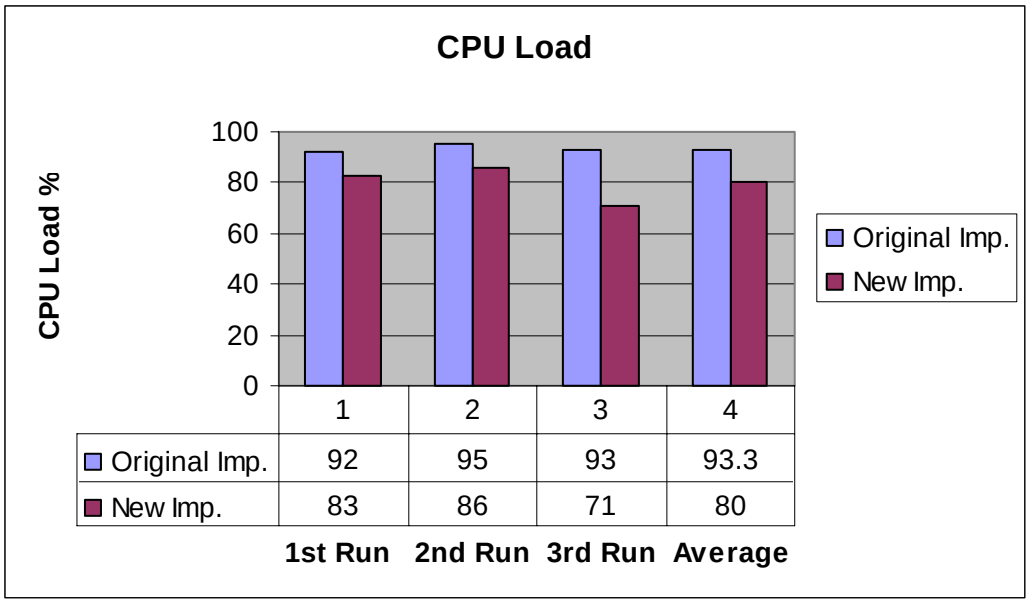
9. Performance comparison between the old and new CD++ implementations

After implementing the same model using the old and the new versions of CD++, some performance metrics (execution time, CPU load…etc) were collected to compare between the two implementations. The scenario used in these test is identical to the last scenario in the previous section. The following figures show the analysis results:



Performed on PIV machine with 512 MB RAM running Red Hat Linux 9

Figure-10 Comparison of the execution time between two different implementations



Performed on PIV machine with 512 MB RAM running Red Hat Linux 9

Figure-11 Comparison of the CPU load between two different implementations

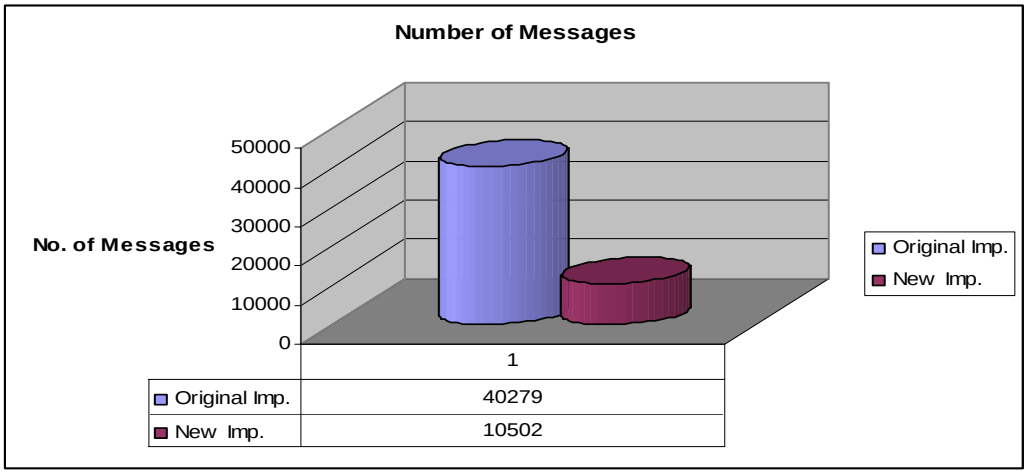


Figure-12 Comparison of the number of messages between two different implementations

The memory used by the simulator was the same for both implementations (~3.6 MB). However by comparing the execution time, CPU load, and the number of messages generates for each implementation, we will find very noticeable performance enhancement when using the new extensions offered by CD++. This enhancement is due to the fact that the cell space was simplified (2-D instead of 3-D) when implementing the model using the new CD++ features.

10. Model improvement

After implementing the original model using the new CD++ version, some extra features were added to the model to improve its behavior. These features are:

- Extending the situation awareness of the soldier (neighborhood) to include the eight surrounding cells. Hence, the soldier is able to attack and move diagonally as well as horizontally or vertically.

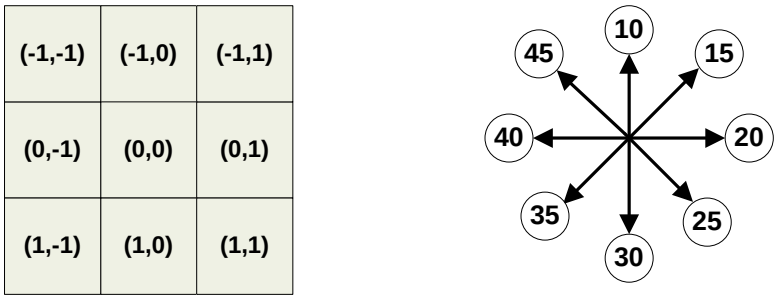


Figure-13 Extending the soldier's neighborhood to Moore's Neighborhood

- Obstacle avoidance, the soldiers are able to avoid obstacles (FS=50) while moving towards the enemy's flag.

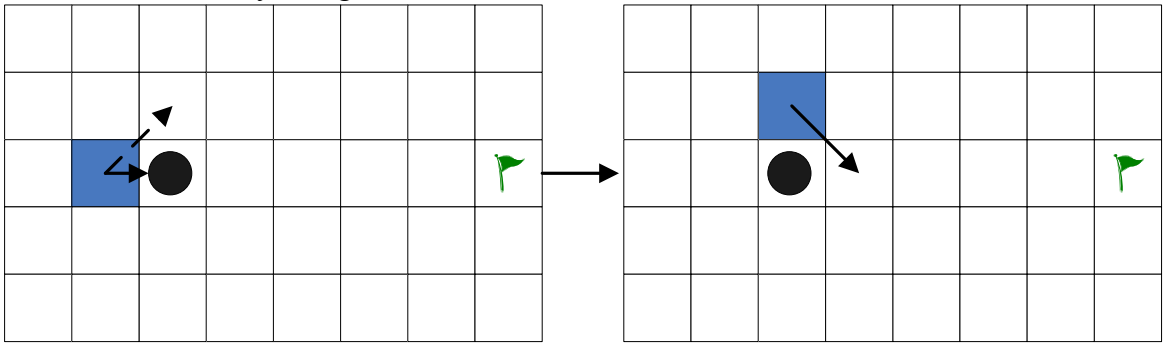


Figure-14 Obstacle avoidance example

- Courage factor (CF), this factor is used to simulate that not all the soldiers in a battlefield will have the same courage to fight the enemy. Hence, this factor will determine if the solider is going to attack the enemy or retreat towards his own base/flag.

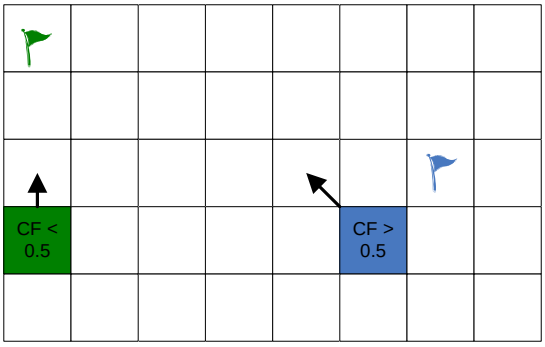


Figure-15 The effect of the Courage Factor FA on the soldier's behavior

11. New features implementation details

In order to implement the new features, some of the rules were extended to behave as expected with the new features. These rules are as follows:

11.1 Initialization rules

The initialization rules were changed to accomplish the following:

- Randomly assign the courage factor CF and then set the Target_Flag to be whether the enemy flag or the soldier's own flag.
- Assign the target flag by reading the values of A_flag and B_flag state variables. Therefore, manual manipulation of the initialization rules in the battleX.ma file is not needed anymore.

```
rule : { ~fs := trunc((0,0)~fs)+1 ; ~fa := abs(fractional((0,0)~fa)) ;
~direction := 0 ; ~target_flag := 0 ; }                { $cf :=
normal(0.5,0.15) ; } 0 {  fractional((0,0)~fs) != 0 and (0,0)~fs < 0 and
$cf = 0}

rule : { ~fs := trunc((0,0)~fs) ; ~fa := abs(fractional((0,0)~fa))  ;
~direction := 0 ; ~target_flag := 0 ; }                 { $cf :=
normal(0.5,0.15) ; } 0 {  fractional((0,0)~fs) != 0 and (0,0)~fs >= 0
and $cf = 0}

rule : {~target_flag := if( $cf > 0.5 , $B_flag , $A_flag ) ; }  0 { (
(0,0)~fs = 1 or (0,0)~fs = 2) and $cf != 0 and (0,0)~target_flag = 0 }

rule : {~target_flag := if( $cf > 0.5 , $A_flag , $B_flag) ; }  0 {
((0,0)~fs = -1 or (0,0)~fs = -2) and $cf != 0 and (0,0)~target_flag = 0
}

rule : { ~fs := 5 ; ~fa := 0; ~direction := 0 ;  } 0 { (0,0)~fs = 5 and
(0,0)~fa = 5 }
rule : { ~fs := -5 ; ~fa := 0; ~direction := 0;   } 0 { (0,0)~fs = -5
and (0,0)~fa = -5 }
rule : { ~fs := 50 ; ~fa := 0; ~direction := 0 ;  } 0 { (0,0)~fs = 50
and (0,0)~fa = 50 }
```

11.2 Fighting rules

These rules behave exactly as the previous ones except that the "fight_rule_1" and "fight_rule_2" macros were extended to consider the new cells added to the soldier neighborhood.

```
#BeginMacro(fight_rule_1)
(
if( ((-1,-1)~fs = -1 or (-1,-1)~fs = -2) and (-1,-1)~fa > 0.5 and  ((-
1,-1)~fa > (0,0)~fa) , -1,0) +
if( ((-1,0)~fs = -1 or (-1,0)~fs = -2) and (-1,0)~fa > 0.5 and ((-
1,0)~fa > (0,0)~fa) , -1, 0) +
if( ((-1,1)~fs = -1 or (-1,1)~fs =-2) and (-1,1)~fa > 0.5 and ((-1,1)~fa
> (0,0)~fa) ,-1,0) +
if( ((0,-1)~fs = -1 or (0,-1)~fs = -2) and (0,-1)~fa > 0.5 and ((0,-
1)~fa > (0,0)~fa), -1, 0) +
if( ((0,1)~fs = -1 or (0,1)~fs = -2) and (0,1)~fa > 0.5 and ((0,1)~fa >
(0,0)~fa), -1, 0) +
if( ((1,-1)~fs = -1 or (1,-1)~fs = -2) and (1,-1)~fa > 0.5 and ((1,-
1)~fa > (0,0)~fa) , -1,0) +
if( ((1,0)~fs = -1 or (1,0)~fs = -2) and (1,0)~fa > 0.5 and ((1,0)~fa >
(0,0)~fa) , -1, 0) +
if( ((1,1)~fs = -1 or (1,1)~fs = -2) and (1,1)~fa > 0.5 and ((1,1)~fa >
(0,0)~fa) , -1,0)
)
#EndMacro
```

11.3 Flags-under-attack rules

These rules behave exactly as the previous ones except that the macros “fight_rule_3” and “fight_rule_4” were extended to consider the new cells added to the soldier neighborhood.

```
#BeginMacro(fight_rule_3)
(
if( ((-1,-1)~fs = -1 or (-1,-1)~fs = -2) and (-1,-1)~fa > 0 , 1,0) +
if( ((-1,0)~fs = -1 or (-1,0)~fs = -2) and (-1,0)~fa > 0 ,1,0)  +
if( ((-1,1)~fs = -1 or (-1,1)~fs = -2) and (-1,1)~fa > 0,1,0) +
if( ((0,-1)~fs = -1 or (0,-1)~fs = -2) and (0,-1)~fa > 0 ,1,0) +
if( ((0,1)~fs = -1 or (0,1)~fs = -2) and (0,1)~fa > 0 ,1,0)  +
if( ((1,-1)~fs = -1 or (1,-1)~fs = -2) and (1,-1)~fa > 0 ,1,0) +
if( ((1,0)~fs = -1 or (1,0)~fs = -2) and (1,0)~fa > 0 ,1,0) +
if( ((1,1)~fs = -1 or (1,1)~fs = 2) and (1,1)~fa > 0 ,1,0)
)
#EndMacro
```

11.4 Movement direction rules

The same rules are used as the previous ones but the macros “director_row”, “director_column”, and “director_row_column” were changed to accomplish the following:

- Move diagonally towards the target flag, instead of arbitrarily moving horizontally or vertically (in case the soldier and the target flag are located on different rows and columns)
- Check if any obstacle is found in the direction of next step and try to find an alternate path.

```
#BeginMacro(director_row)
(
 if ( cellPos(1) > round(fractional((0,0)~enemy_flag) * 100)  , if( (0,-
1)~fs != 50,40, if((-1,-1)~fs = 0 ,45,35) ) ,
 if( (0,1)~fs != 50 ,20, if((-1,1)~fs = 0 , 15,25) ))
)
#EndMacro
#BeginMacro(director_column)
(
 if (cellPos(0) > trunc((0,0)~enemy_flag) ,if( (-1,0)~fs != 50, 10,
if((-1,-1)~fs = 0, 45,15) ),
 if( (1,0)~fs != 50, 30, if((1,-1)~fs = 0 ,35, 25) ) )
)
#EndMacro
#BeginMacro(director_row_column)
(
if( cellPos(0) > trunc((0,0)~enemy_flag)  , if( cellPos(1) >
round(fractional((0,0)~enemy_flag) * 100) ,
if ((-1,-1)~fs != 50,45, if((-1,0)~fs = 0, 10,40) ), if ((-1,1)~fs !=
50,15, if((-1,0)~fs = 0, 10,20) )) ,
if( cellPos(1) > round(fractional((0,0)~enemy_flag) * 100) ,if((1,-1)~fs
!= 50, 35, if ((1,0)~fs = 0 , 30, 40) ),
if((1,1)~fs != 50, 25,if((1,0)~fs = 0 ,30,20)  )    ) )
)
#EndMacro
```

11.5 Movement rules

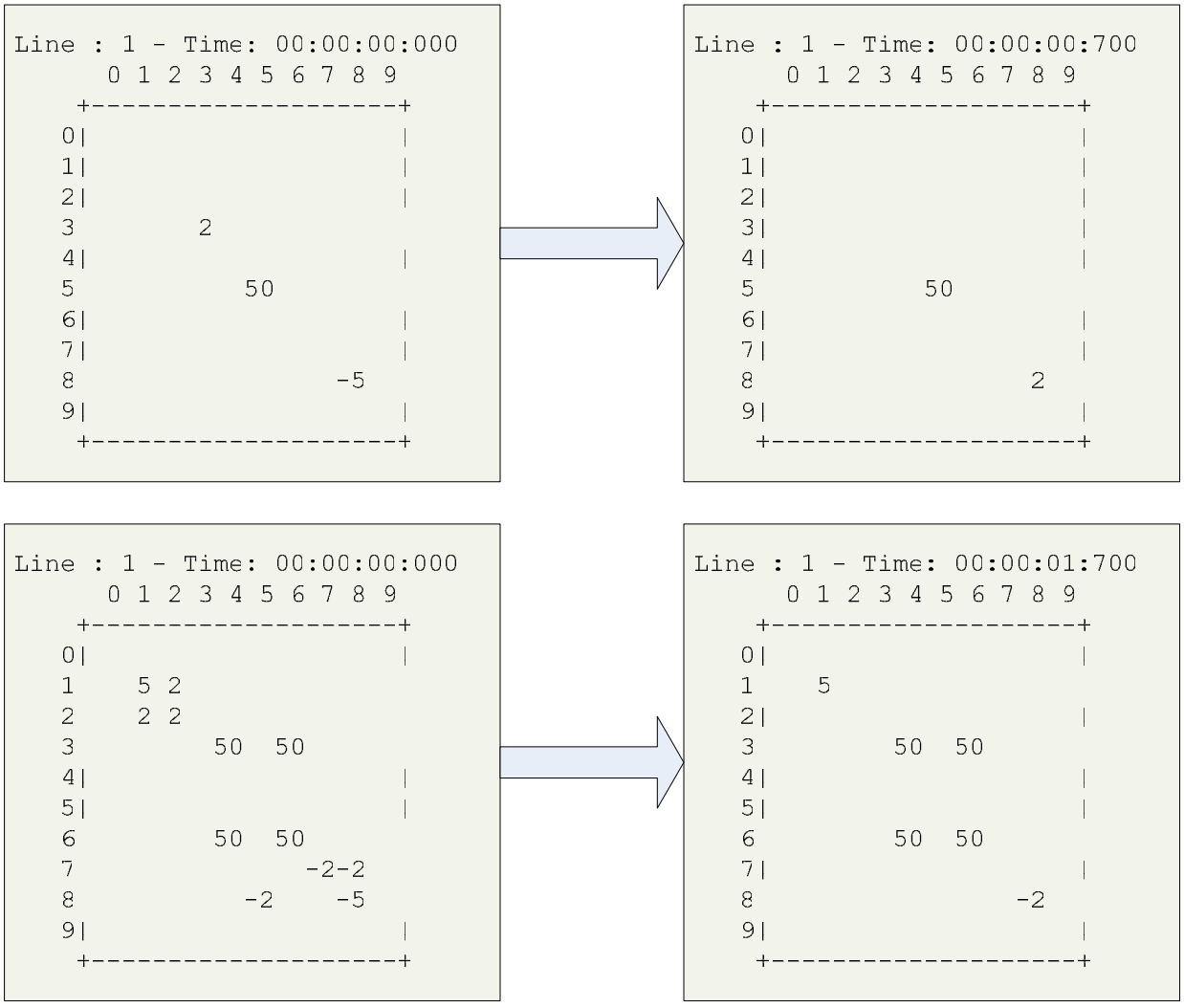
These rules depend on the same principle as the previous ones except that they were extended to consider the new cells added to the soldier neighborhood.

```
#BeginMacro(move_from_south_west_factor)

rule : { ~direction := if ( (1,-1)~direction = 15, -15, (0,0)~direction)
;  } 100 { (0,0)~fs = 0 and (0,0)~direction = 0 and (1,-1)~fa > if((-1,-
1)~direction = 25 , (-1,-1)~fa , 0) and  (1,-1)~fa >
if ((-1,0)~direction = 30, (-1,0)~fa, 0) and (1,-1)~fa > if((-
1,1)~direction = 35, (-1,1)~fa,0) and  (1,-1)~fa > if((0,-1)~direction =
20, (0,-1)~fa,0) and (1,-1)~fa > if( (0,1)~direction = 40 , (0,1)~fa,0)
and  (1,-1)~fa > if ( (1,0)~direction = 10, (1,0)~fa,0) and (1,-1)~fa >
if((1,1)~direction = 45 , (1,1)~fa, 0)  }

#EndMacro
#BeginMacro(move_from_south_west)

rule  : { ~fs := (1,-1)~fs ; ~fa := (1,-1)~fa ; ~direction :=0;
~target_flag := (1,-1)~target_flag ;  }  0 {  (0,0)~fs = 0 and  ( (1,-
1)~fs = 2 or (1,-1)~fs = -2) and (1,-1)~direction = 15 and
(0,0)~direction = -15 }

rule : {~fs := 0 ; ~fa := 0 ; ~direction := 0 ; ~target_flag := -1;  }
{ $cf := 0; }  0 { ((0,0)~fs = 2 or (0,0)~fs = -2)  and  (-1,1)~fs = 0
and (0,0)~direction = 15 and (-1,1)~direction = -15  }

#EndMacro
```

12. New features tests

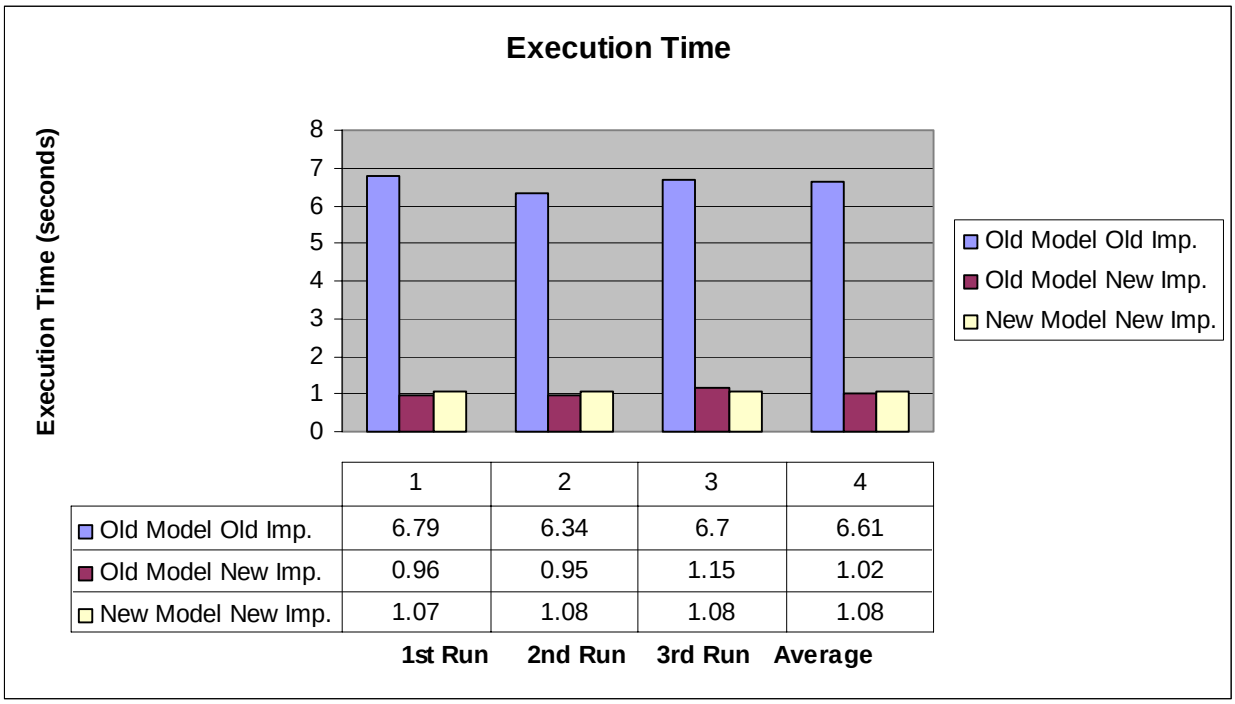
In order to test the new features incorporated in the model, two scenarios are considered here:

- The first one tests the diagonal movement and obstacle avoidance of the soldiers. The detailed output of this scenario is available in the battle1.drw file within the battlefield_new.new_CD++ folder.
- The second one, test the overall behavior of the model after incorporating the courage factor CF. The detailed output of this scenario is available in the battle2.drw file within the battlefield_new.new_CD++ folder.



13. Performance analysis of the old and improved models

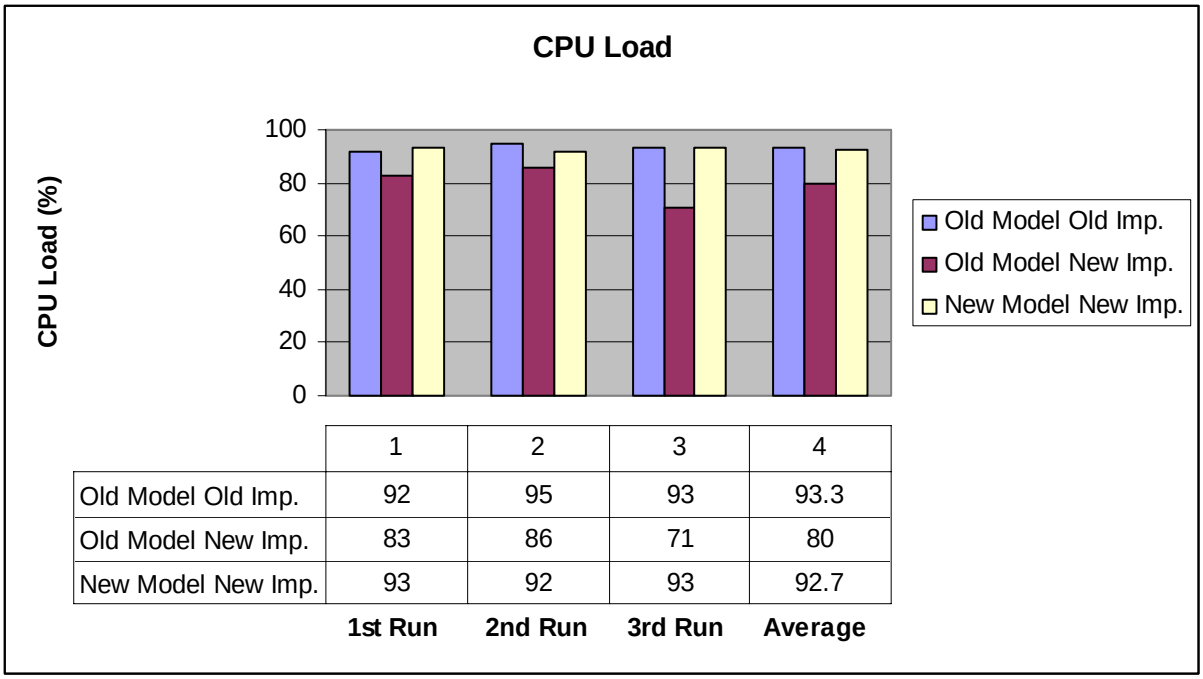
In this section, an overall performance analysis of the old model (old and new implementations) and the improved one (new implementation) is presented. The scenario used to test the performance of the improved model is identical to the scenario in the previous section.



	1	2	3	4
Old Model Old Imp.	6.79	6.34	6.7	6.61
Old Model New Imp.	0.96	0.95	1.15	1.02
New Model New Imp.	1.07	1.08	1.08	1.08
	1st Run	2nd Run	3rd Run	Average

Performed on PIV machine with 512 MB RAM running Red Hat Linux 9

Figure-16 Comparison of the execution time between three different implementations



	1	2	3	4
Old Model Old Imp.	92	95	93	93.3
Old Model New Imp.	83	86	71	80
New Model New Imp.	93	92	93	92.7
	1st Run	2nd Run	3rd Run	Average

Performed on PIV machine with 512 MB RAM running Red Hat Linux 9

Figure-17 Comparison of the CPU load between three different implementations

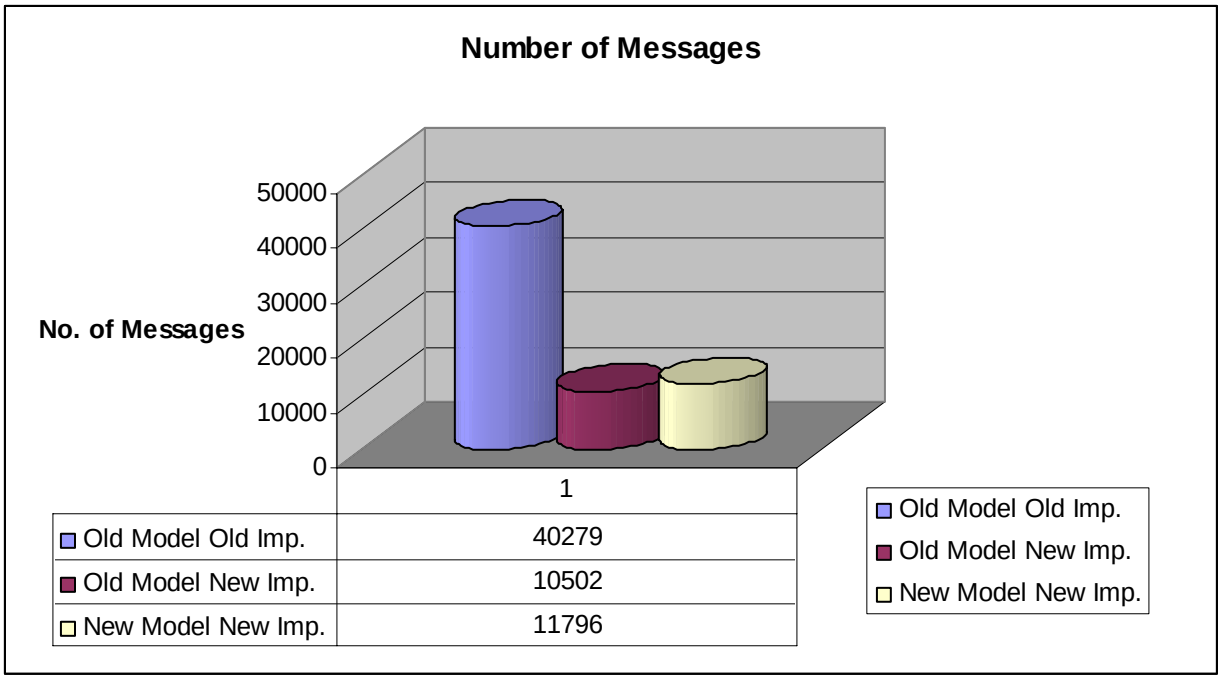


Figure-18 Comparison of the number of messages between three different implementations

The previous figures show that the new features in the model have added some overhead in terms of execution time, CPU load, and number of messages exchanged. However, this overhead is tiny and not comparable to the performance gain achieved when re-implementing the model using the new version of CD++.

14. Conclusions

The new CD++ features have added extra functionality to define and simulate complex cell-DEVS models. In addition, it simplified the model definition, and debugging dramatically. However, the fact the performance gain achieved with the battlefield model, may not be achievable in the case of simple models. Previous work (done by *Wainer, López 2003)* has shown that re-implementing the life-game model using the new CD++ version has introduced an overhead when executing the model. Hence, the model nature and specification play an important rule in determining whether the old or new implementation should be considered. In addition, one need to be careful when using multiple input/output ports as they increase the number of messages exchanged within the model which in turn affect the performance when executing the model in parallel.