APPENDIX I

We used a genetic algorithm GA [20] to search the parameter space of the DS-M for configurations which deliver the best fits to the data. We chose a GA since they tend to perform well when the feedback information on performance includes variation [20], as is the case here where properties are calculated from several stochastic simulation runs. The GA (Algorithm 1) starts with a set of k randomly generated individuals (states), called the population. Each individual encodes a set of model parameters using binary strings of '0' and '1'.

Algorithm 1 Genetic algorithm: the general procedure.

```
1: function GeneticAlgorithm(population 'pop', fitness function 'FitnessFN')
2:     current-pop ← pop
3:     while there is still a significant performance increase do
4:         new-population ← empty set
5:         for all individuals in current-pop do
6:             x ← RandomSelectIndividual(current-pop,FitnessFN)
7:             y ← RandomSelectIndividual(current-pop,FitnessFN)
8:             if with crossoverprobability 'pCross' then
9:                 child ← Crossover(x,y)
10:            else
11:                child ← RandomSelectIndividual(x,y, FitnessFN)
12:            end if
13:            if with probability 'p' then
14:                child ← Mutate(child)
15:            end if
16:            add child to new-population
17:        end for
18:        current-pop ← new-population
19:    end while
   return the best individual(s) in the population
20: end function
```

Next, each individual is tested for its performance, which is the fitness value returned by F(D,W) (i.e. how well the fires produced by the model, given the parameters encoded by the individual, match the field data). The next generation of states is then generated from the current population. For this, two individuals are chosen at random such that the likelihood of a state being chosen is proportional to its fitness.

Algorithm 2 Crossover operation on 'genome' strings.

```
1: function Crossover(individuals x,y)
2:     n← Length(x)
3:     c←random number from 1... n
   return Append(Substring(x,1,c),Substring(y,c+1,n))
4: end function
```

The binary strings which encode the corresponding parameter values of the two states are then chopped at a randomly chosen point and new strings are formed (Algorithm 2). Each new string contains a part of both parent strings. The resulting strings are randomly mutated, with a probability 'p' at each location. The 'reproduction' procedure is repeated until we have 'k' offspring states. In our version, we allowed the individual with the best performance to directly enter the next generation unmodified. The procedure continues with this new generation as for the first until the performance of the populations does not increase significantly anymore. The best values are noted and tracked during the entire process. We used a population size of 30 individuals (representing parameter sets); a number of 800 individuals were evaluated in total. Each individual was run for 15,000 steps before taking 2,000 fire samples in a ten-step interval. The parameters were encoded in binary strings with random mutation (rate 0.05) and a crossover rate of 0.7 with roulette wheel selection [20].

APPENDIX II

In the following we document the algorithms which were used in the analysis of burn clusters in the model.

Algorithm 3 Determine fire cluster after initial spark.

```
Require: Cell c has been struck by lightning and ignited.
 1: c.removeTree()
 2: queue.add(c)
 3: while queue is not empty do
 4:     c←queue.poll()
 5:     for all von Neumann neighbors n of c with n ∉ queue do
 6:         if n is occupied by a tree  then
 7:             n.removeTree()
 8:             queue.add(n)
 9:         end if
10:     end for
11: end while
```

Algorithm 3 is used to determine the cells which belong to a cluster of burned area. The outer perimeter of the burned area and the island rims, i.e. the cells of an island which border directly to the recently burned area, are obtained by Algorithm 4.

Algorithm 4 Find all cells belonging to a rim.

```
Require: List rimCells contains all cells which were asked in the burn process,
    yet did not ignite.
 1: rim ← new List()
 2: queue ← new Queue()
 3: c ← arbitrary cell from rimCells
 4: rimCells.remove(c)
 5: rim.add(c)
 6: queue.add(c)
 7: while queue is not empty do
 8:     d ← queue.poll()
 9:     for all von Neumann neighbors n of d do
10:         if n is in rimCells then
11:             queue.add(n)
12:             rimCells.remove(n)
13:             rim.add(n)
14:         end if
15:     end for
16: end whilereturn rim
```

The problem remains to separate the outer perimeter from the island rims. We use the assumption that the largest rim is the perimeter.

Algorithm 5 Find all cells belonging to an island.

```
Require: Queue q and list island contain all cells of an island perimeter.
1: while q is not empty do
2:     c ← queue.poll()
3:     for all von Neumann neighbors n of c do
4:         if  n is not in the rim AND n is not burned
5:             q.add(n) then
6:             island.add(n)
7:         end if
8:     end for
9: end while
```

Finally, algorithm 5 finds all cells which belong to an island once its rim has been determined.