SEVENTH FRAMEWORK PROGRAMME

EvoEvo Deliverable 4.3

EvoMachina user documentation

Due date:	M18 (April 2015)
Person in charge:	Susan Stepney
Partner in charge:	University of York (UoY)
Workpackage:	WP4 (A computational EvoEvo framework)
Deliverable description:	User documentation for initial release of the EvoMachina system

Revisions:

Revision no.	Revision description	Date	Person in charge
0.1	Initial release for comment	5/7/16	Tim Hoverd (UoY)

Table of Contents

1. Introduction

This document describes user documentation for the initial release of *EvoMachina*, a prototype implementation of the Metamodel developed and described as deliverable 4.1.

The metamodel introduced a model for genetic algorithms that was based explicitly on a biological cell's reproductive machinery. The central notion of the model is the explicit use of *machines* to represent those parts of the cell machinery that carry out operations within the cell and which have an underlying structure which defines the details of their behaviour.

The rest of this document defines the platform model[1] that satisfies the requirements of the metamodel and summarises how the initial prototype implementation works. Subsequently, we present some results based on specialising the platform model implementation into a specific domain with a specific, and well-known, problem to solve. This specialisation forms initial guidelines for users who wish to use EvoMachina in other domains of application.

2. The *EvoMachina* platform model

The metamodel is a model that describes those aspects of an EvoMachina implementation that are common to all such implementations. That is, it describes the generic, shared, mechanisms. These mechanisms must be implemented in some manner and the platform model described here is one such model.

2.1. Structures and Machines

The core idea contained in both the metamodel and the platform model is that of a *machine* that is an special case of a *structure* which contains some information that defines the behaviour of the machine. This and related notions are summarised in the UML class diagram in Figure 1. This model shows that:

1) A Structure is some sequence of *Pearls* each of which is drawn from a particular *Domain*. Moreover, all of the pearls that form the code of a structure are required to be drawn from the same domain. Pearls are not mutable and in some implementations it might well be the case that a specific domain contains *prototype* implementations of each possible Pearl.
2) A Machine is a special sort of structure that has some behaviour, exemplified by the presence of a *doIt* operation, the implementation of which will actually carry out some function that is the specific machine's role.
3) Many machines, for example one tasked with translating a transcription unit into a functioning machine, might well have a specific *source* structure as the structure upon which it operates. In this example the *Translator* machine would have a *source* structure that was the transcription unit in question.

[1] P. S. Andrews, S. Stepney, T. Hoverd, F. A. C. Polack, A. T. Sampson, J. Timmis, 2011, CoSMoS process, models, and metamodels. *CoSMoS 2011*, Luniver Press, pp.1-13.

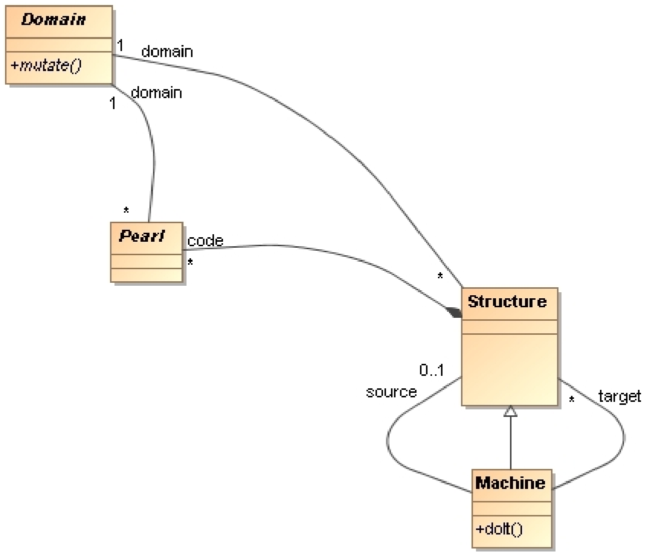


Figure 1: Structures and Machines

4) Similarly, some machines might produce a structure as a result of their activity and this is shown as the *target* structures in the figure.

5) The *Domain* represents the entire world of a set of related *Pearls*. As such, the notion of how to mutate a sequence of *Pearls*, likely the *code* of *Structure*, is implemented by the *Domain* itself and represented here as the *mutate()* operation. Furthermore, the domain acts as a factory for new machines of the appropriate sort. Although the mutation operation itself is defined within the Domain, it is the case that that mutation is invoked by the *Machine*ry of an individual.

2.2. Spaces and Individuals

Structures, and the specialised structures that are machines, always exist in the context of some space. Figure 2 shows the basics of this relationship.

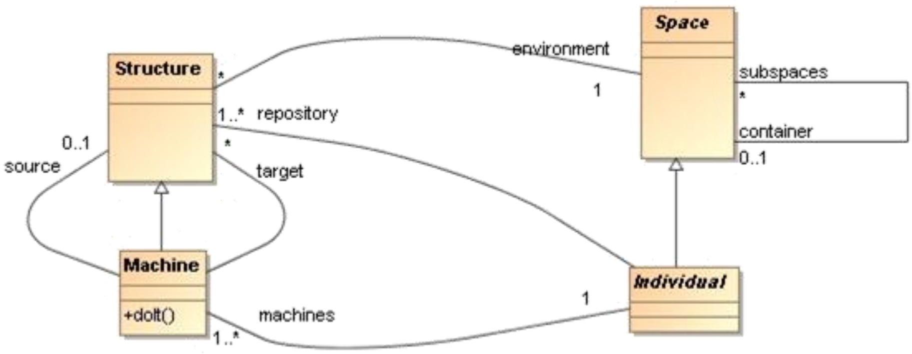


Figure 2: Structures within Spaces

Here a *Space* is the *environment* within which a collection of *Structures* exist. Specifically, Structures cannot exist outside of an environment. A space can have a hierarchical structure with a container and a set of possible subspaces.

There is a specific type of Space, Individual, which represents a set of structures, some of which will be machines, that are cooperating to solve some task. That task might just be surviving in some environment or it could be a specific computation that is a component of an overall genetic algorithm. More particularly, an Individual has a collection of structures as its repositories which is the set of structures that are templates for the machines that can, and may, be expressed within the context of that Individual.

2.2.1. Searchable spaces

A particular problem being solved using the EvoMachina framework will exist as a space that contains (directly or indirectly) Individuals. In order to guide implementation an interface SearchableSpace is defined that specifies the behaviour required of such spaces.

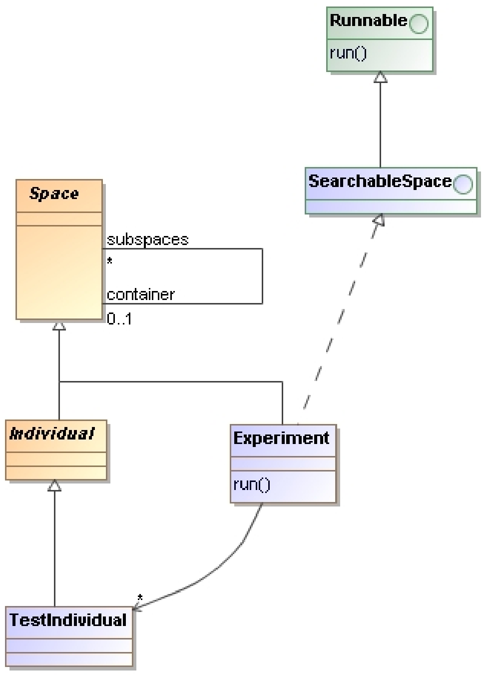


Figure 3: SearchableSpaces

SearchableSpace extends the standard Runnable interface meaning that experiment classes also have to be able to be run() as a separate thread.

2.3. Types of machine

As machines provide the behaviour of an Individual, from replication of that individual to solving the task that is embodied by the individual, there is a need for various types of machine. As shown in Figure 4, this is achieved by simply specialising the Machine class and overriding the functionality of the doIt operation.

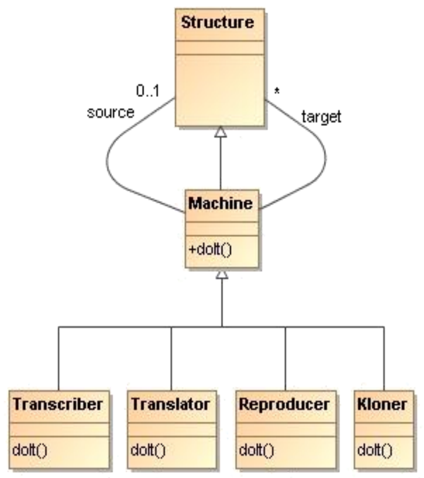


Figure 4: Machine varieties

2.4. Domains and machine factories

A domain must act as a factory for new instances of specific types of machine. This is shown diagrammatically in Figure 5.

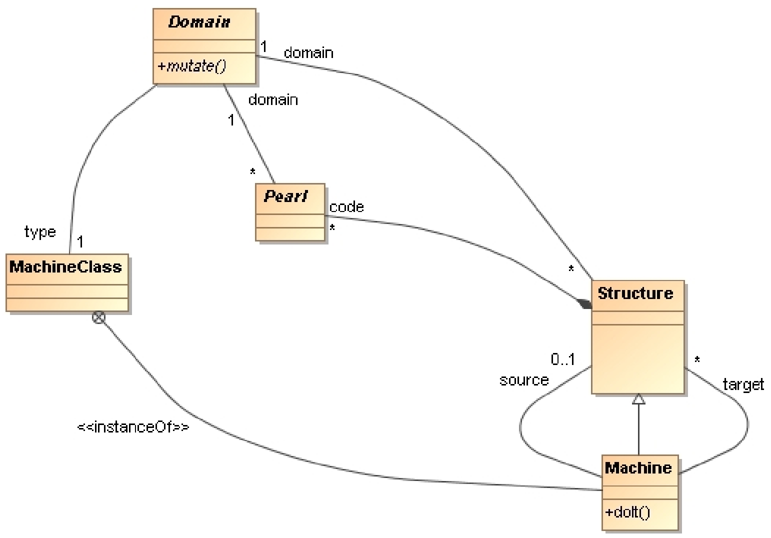


Figure 5: Domains as machine factories

That is, each domain has an instance of the *class* of the type of machine for which a particular structure is the machine template. The domain can use this class, by reflection, to construct instances of the required type of machine as they are required to be expressed by the Translation mechanism.

3. Implementation decisions

In order to implement the model as already outlined, a collection of implementation decisions have to be taken. The most important of these are described here:

3.1. Java

It is necessary to choose a programming language that has wide applicability, is easy for users to come to terms with and which is mostly insensitive to the implementation platform. The decision was therefore taken to use Java, albeit of the most recent (at the time of writing) release: Java 8. Java 8 provides a good degree of platform independence and a number of sophisticated facilities that allow for concise programming. It's also available for just about every execution platform available.

3.2. Simple concurrency model

It's expected that implementations of many real world problems using EvoMachina will result in an execution context with a very large number of Individual objects competing for machine resources. All modern execution contexts will be multi-core environments and, as such, managing the concurrent use of those processor cores will be essential to achieving efficient use of the machine. It would be possible to use many different mechanisms to manage this concurrency but many of them would obscure the essentially independent nature of each executing Individual object. As each Individual is modelled to some extent after a biological cell, which themselves act independently, then retaining that independence is an important part of this model.

As such, the EvoMachina implementation uses Environment Orientation[2] to simplify the concurrency model. Essentially, each Individual executes independently and the world within which they live manages the interdependencies such as competition for potentially shared resources. This matches what happens in the biological world where organisms are independent and inter-relationships are mediated by the (in this case real) world of physics.

3.3. Class library

The ideal ultimate implementation of EvoMachina would be something like a shared API or REST website so that users could easily access that implementation without having to incorporate it into their own code.

However, such a sophisticated approach has not, yet, been implemented. The current implementation is just as a Java class library, encapsulated in a .jar file, and it's expected that users will encode their own implementations as extensions of those standard classes. Such an approach has the potential downside of users needing to extend the functionality of EvoMachina superclasses. The current approach is to ignore this particular issue and to look, in the future, to a more robust implementation model such as the possibilities mentioned above.

3.4. Command line

Whilst an ideal implementation would implement sophisticated visualisations allowing users to interact directly with a running experiment the current EvoMachina implementation does none of this. In order to allow for simple execution it is easy to build command line executables using property files for definition of many run time parameters. A logging framework, based on the standard java.util.logging framework provides for extensive logging to files or, ultimately, other data sources. Currently, analysis of the behaviour of an implementation must be done by textual analysis of the contents of log files using a tool such as *awk*.

[2] Hoverd, T. & Stepney, S. (2015), 'Environment orientation: a structured simulation approach for agent-based complex systems.', *Natural Computing* **14** (1) , 83-97.

4. Example implementation

In order to explore the capability of the current implementation this section includes many details of an example implementation. This implementation searches for solutions to the Travelling Salesperson Problem (TSP). This is a problem well known to computer science and, as such, there are known heuristics for solving the problem, or at least converging on a good enough solution.

The TSP problem is that given a set of locations of cities which a putative salesperson has to visit just once, what is the ideal route around those cities that minimises the distance that the salesperson has to travel?

So, for example, a typical TSP problem might be to find the shortest route that visited all the state capitals of the "lower 48" US states. (That is, the contiguous states.) Test data is readily available with which to set up this particular problem as well as many others. The data used in the example discussed here was obtained from a website[3] that provides many different TSP data sets.

A TSP problem is essentially one of taking a list of cities and permuting it in many ways until a good solution is found. However, it's not clear how to perform this permutation. In the implementation described here the approach taken is to permute a list of cities using the Lin-Kernigham heuristic[4]. This is a *k-opt* generalisation of the approaches known as 2-opt, 3-opt, and so on. The heuristic is basically to cut a defined journey around all of the cities into a number of pieces (the value of *k*) and to reassemble the pieces in all possible ways and then to re-evaluate the time taken to traverse the newly assembled journeys.

The following sections of this document describe many aspects of the implementation of the TSP problem, using essentially class specialisation, within the EvoMachina framework.

4.1. Implementation details

The basic idea behind the implementation described here is to represent a single possible solution to a TSP problem as an Individual of some sort. That individual would support an operation that provided the journey time that it represented. These individuals will exist in a world where a number of similar individuals exist, each representing a possibly different route around the cities of the problem. These individuals will compete to take over more of the world with successful individuals being replicated into new parts of the world and less successful ones being left to die off. Such replications will mutate the route used, possibly leading to a better solution.

The world used for the implementation described here is a toroidal space. That is, every square site in that world has neighbours on all four sides. Each site in the toroidal space is constrained to having only one, or no, resident individuals. The described implementation handles concurrency in the following manner:

1. The toroidal space handles competition between individuals and as such is scheduled to periodically carry out competitions.
2. Each site in the space is periodically scheduled to run any resident individual.
3. When an individual is run it:

[3] https://people.sc.fsu.edu/~jburkardt/datasets/tsp/tsp.html
[4] Lin, Shen; Kernighan, B. W. (1973). "An Effective Heuristic Algorithm for the Traveling-Salesman Problem". *Operations Research* **21**, 498–516.

   a. Decides whether to commit suicide which is done on the basis of the number of times it's been run (essentially how old it is) and how many times it's replicated (which is some measure of its success).
   b. If it's still alive it looks to see if there an adjacent empty site (using a Von Neumann neighbourhood) and, if so, tells the containing world that the empty site so found should be competed over.

These actions are carried out in a specified number of threads using Java thread scheduling features, in particular those provided by the standard *ScheduledExecutorService* class.

Given this structure, a particular TSP problem can be solved by seeding one or more individuals into the toroidal space with initial random routes and then letting those individuals replicate across the space. Success, or otherwise, is supported by the world itself providing the current "best" individual when required.

The following sections describe the specific extensions to various parts of the framework, that have been done to support the TSP implementation.

4.1.1. Spaces

The TSP implementation realises the *SearchableSpace* interface, as shown in Figure 4 to represent the space within which a particular problem is solved. This class is *ToroidalTSP2DSpace*. This space contains a number of implementations of the basic class *Site (*the *Journey2DSite* class*)*, each of which contains, or not, an instance of the class *Journey*, a *Space* which represents a particular route around the cities of the problem

4.1.2. Domains

The TSP implementation includes two subclasses of the *Domain* class that provide for TSP specific functionality. These are:

CityType

This domain represents the set of all possible cities around which the TSP must make their journey. All cities in the implementation are instances of the application specific class *City*. A *CityType* object knows, in a particular experiment, what the distance between any pair of cities is. It is assumed that the distance from city A to city B, for example, is the same as that from B to A. One part of the initialisation of a particular problem is setting up the *CityType* instance with all the required inter-city distances. Also on initialisation of a *CityType* instance a mutation operator must be provided that implements the Lin-Kernigham heuristic as described above.

KlonerDomain

This domain is the set of all possible *k* in a *k-opt* mutation operator. The current implementation uses all integers between 2 and 10 in this domain with the particular value used for a particular *Kloner* machine being determined by which *Pearl*s in the machine's code as set to be coding, as opposed to non-coding.

4.1.3. Machines

The set of machines available in this implementation is as follows:

Transcriber

This machine is responsible for taking one of the structures in an Individual's repository and transcribing it into a new *Structure* representing a transcription unit. In this initial implementation the only modification done here is to excluded non-coding *Pearl*s from the transcription unit.

Translator

This machine is responsible for taking a transcription unit and generating the appropriate type of machine with the code of the transcription unit as the new machine's code. The new machine is created by using the domain of the transcription unit which, by reflection, provides a facility to construct a new machine object of the appropriate class.

Reproducer

This machine, which is an essential part of all *Individual*s, is responsible for coordinating the efforts of other machines in order to replicate an individual space. This approach is a deliberate shortcut with the replicating encoded explicitly. It's to be expected that in future releases of EvoMachina the approach taken here will be more closely matched to the underlying biological inspiration and there will be little need for this type of machine.

Kloner

This machine is used explicitly by the reproduction process implemented by the *Reproducer* machine. Its action it to replicate the repository of the "parent" individual into that of the "daughter" individual. However, this replication is potentially inaccurate in that each template in the repository is copied using the mutate() operation of its associated *Domain* object. The method invoked here may well mutate the code of the template in some manner.

TSPCalculator

This machine represents the fitness function of the containing *Individual*. Its code is the sequence of cities in a particular possible solution to the overall problem. This machine supports a specific operation to provide the journey time of the calculator machine in question. The method that implements this operation uses the facilities provided by the *CityType* domain to provide the answer and this operation is used by the specific individual space which contains this type of machine.

4.1.4. Main class

In addition to all the classes described above a particular command line invocation of the problem is encapsulated as the Main class in the package that contains all the TSP classes. The entire source code of the main method of that class is shown below, with interspersed commentary to describe its function.

Initialisation requires that the user invoke the application with the name of a Java properties file that describes various execution parameters:

```
public static void main(String[] args) {
    if (args.length != 1) {
        System.out.println(String.format("Usage: java -classpath <classpath> %s <property...",
                                         Main.class.getName()));
        System.exit(1);
    }
    // Create the domain:
    SetUp();
```

The setup method, elsewhere in the Main class, sets up the necessary domains for the implementation including the *CityType* domain which knows the distances between the various cities in the problem.

The properties file is embedded in a singleton class, Simulation, that is used elsewhere in the code to determine the value of execution time parameters. The parameters in this file are used to create the world within which the problem will be solved. Here it is using a two dimensional toroidal space, each cell of which contains an instance of the class *JourneySite* which can contain an instance of the *Journey* class itself.

```
    // Load the Properties files into the Simulation class that is referred to by...
    try {
        Properties properties = new Properties();
        properties.load(new FileInputStream(args[0]));
        Simulation.SetProperties(properties);
    } catch (IOException e) {
        e.printStackTrace();
        System.exit(2);
    }
    // Create the toroidal TSP world. In this case it's 9x9 cells by default:
    int xSize = Simulation.GetValue("xSize", 9);
    int ySize = Simulation.GetValue("ySize", 9);
    ToroidalTSP2DSpace world = new ToroidalTSP2DSpace(xSize, ySize);
```

Here the *KlonerDomain* is created and an initial journey, with a random route through the world, is created. This particular *Journey* object uses a *Kloner* object which mutates the *k* in the *k-opt* used to mutate the journey itself.

```
    // Create a single journey in the centre of the new world
    _KDomain = new KlonerDomain("Kloner domain", Kloner.class);
    List<Pearl> route = CreateInitialRandomRoute();
    Space js = world.getSubspace(xSize/2, ySize/2);
    Journey j = MakeJourneyWithMutatingCopier(route, js, _CityDomain, _KDomain);
```

Here the executor service, which will manage a number of separate threads, is created and each site in the world, and the world itself, are submitted to the executor service to manage.

```
    // Create scheduled executor and give it all the sites in the world to look after:
    ScheduledExecutorService executor =
                   Executors.newScheduledThreadPool(Simulation.GetValue("numThreads", 20));
    world.getSubspaces()
         .forEach(s -> executor.scheduleWithFixedDelay(
                               (Journey2DSite)s,
                               0,
                               Simulation.GetValue("siteExecutionDelayInMicroseconds",
                                                   100),
                               TimeUnit.MICROSECONDS));

    // Allow the world itself to be run and retain a Future object:
    ScheduledFuture result = executor.scheduleWithFixedDelay(
                               world,
                               0,
                               Simulation.GetValue("worldExecutionDelayInMicroseconds", 50),
                               TimeUnit.MICROSECONDS);
```

Here the execution is run, periodically printing out the current best journey, until either the best journey is better than some threshold or a maximum execution time is reached.

```
    Optional<Individual> best;
    long time = System.currentTimeMillis();

    do {
        try {
            Thread.sleep(Simulation.GetValue("reportingDelayInMilliseconds", 100));
        } catch (InterruptedException e) {}
        best = world.best();
        String message;
        if (best.isPresent()) {
            message = String.format("Best of %d is: %s",
                                    world.numJourneys(),
                                    best.get().getContainer().get());
        } else {
            message = String.format("Best of %d is: nobody",
```

```
                                        world.numJourneys());
        }
        System.out.println(message);
    } while ((best.isPresent() ? ((Journey)(best.get())).journeyTime() : 10000000) >
                               Simulation.GetValue("targetTime", 35500)
             &&
             System.currentTimeMillis() < (time + Simulation.GetValue("totalRunTimeInMilliseconds",
                                                                       60000))
             &&
             !result.isDone());
```

Finally, the executor is shutdown and the final result both logged and printed out.

```
    if (result.isDone()) {
        try {
            Object r = result.get();
            System.out.println("result is " + r);
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
    executor.shutdownNow();
    try {
        executor.awaitTermination(5, TimeUnit.SECONDS);
    } catch (InterruptedException e) {}

    Machine.FlushLogger();
    _Logger.info("Completed, best is " + best.get().getContainer().get());
    Machine.FlushLogger();

    System.out.println("Best: " + best);
}
```

4.2. Properties file

This section contains details of the properties file which must be provided to execute the example EvoMachina code. The properties file is in the format of a normal textual Java properties file. For example, a part of one file could be:

```
logFileName = evoevocl.log
logLevel = FINE
numThreads = 120
targetTime = 34500
```

The complete set of possible properties is as follows:

logFileName

The name of a file, possibly including a path, to which the EvoMachina code will log events.

logLevel

The name of the logging level. The set of names is defined in the standard Java class *java.util.logging.Level*.

numThreads

The number of virtual machine threads that should be scheduled for this problem by the Java *Executor* used. This number should be at least, and probably greater, than the number of hardware threads available due to the number of processor cores present.

targetTime

The targetTime for journeys by the travelling salesperson.

xSize, ySize

The sizes of the two dimensions of toroidal world used in terms of the number of sites in that world. So, having a value of 9 for each property would imply there were 81 sites in that world.

programmedDeath

true if the code of the *Journey* objects should determine whether, or not, a specific *Journey* should die.

deathByOldAge

true if *Journey* objects die as a consequence of being run, and replicated, a certain number of times. If false then there is a probability that a *Journey* will be killed off on every run of the *Individual*.

replicationMultiplier

A number that is multiplied by the number of times that an individual *Journey* has been replicated. If the number of times a *Journey* has been run is greater than the result, then the *Journey* will be killed off.

minRunCount

If the deathByOldAge parameter is true, then this is number of times that a Journey must be run before being killed off. If deathByOldAge is false, and programmedDeath is true, then this is indicative of the probability of a Journey being killed on each run. That is, if the value is 100 then the chance of a Journey being killed on each iteration is 0.01.

minKOpt

The minimum value of *k* in *k-opt* as is used by the code

maxKOpt

The maximum value for k that may be used.

initialKOpt

The initial value for the *k* in *k-opt*.

siteExecutionDelayInMicroseconds

The time in miliseconds between individual *Site*s in the world becoming due for execution. Whether they are executed or not depends on whether other run clients have been executed, the number of threads available and the performance of the host machine.

worldExecutionDelayInMicroseconds

The time in microseconds between executions of the overall world object. The same considerations as those for siteExecutionDelayInMicroseconds apply here.

reportingDelayInMilliseconds

The delay between console reports about the progress of the overall execution.

totalRunTimeInMilliseconds

The total time that an execution will run, in milliseconds, before being terminated.

numToroidalJourneys

The initial number of journeys that will be populated, at random positions, in to the toroidal space.

4.3. Execution and logging

The EvoMachina code is provided in the jar file which may be executed, on a machine that has Java installed, using the command line:

```
java -jar <path to jar file> <path to execution properties file>
```

This will cause the code to search for a route around the state capitals of the lower 48 US states stopping either when an acceptable solution has been found or the execution exceeds some value as determined by the properties file. Along the way the code will generate output on the console of this form:

```
Best of 6 is: Journey2DSite: <1>  [5:3] [Journey: [0, 148179, 5] (1,2,3,4,5,36,14,15,16,35,32
Best of 32 is: Journey2DSite: <1>  [4:1] [Journey: [0, 135732, 4] (1,2,3,4,5,6,7,8,9,45,23,24
Best of 67 is: Journey2DSite: <1>  [6:3] [Journey: [0, 128766, 5] (1,2,3,4,5,28,29,13,12,11,1
Best of 80 is: Journey2DSite: <66>  [6:3] [Journey: [1, 128766, 5] (1,2,3,4,5,28,29,13,12,11,
Best of 72 is: Journey2DSite: <16>  [6:4] [Journey: [0, 126291, 5] (1,2,3,4,5,28,29,13,12,11,
Best of 76 is: Journey2DSite: <45>  [5:4] [Journey: [0, 117255, 4] (1,2,3,4,5,6,7,8,9,45,23,2
Best of 73 is: Journey2DSite: <8>  [1:3] [Journey: [0, 108651, 2] (1,2,3,9,8,7,6,5,4,45,23,29
Best of 76 is: Journey2DSite: <0>  [2:4] [Journey: [1, 108000, 2] (1,2,3,9,38,37,46,47,21,20,
```

Each line here is a report of the best solution at a particular point in time. Each row here consists of a summary of how many individuals are in the current (9x9) toroid and the result of invoking the Java toString() operation on the best site in the current world. So, in the first row above, the components are:

6
: There are currently 6 solutions in the toroid.

<1>
: This particular site has been run() 1 time since its contents last replicated.

[5:3]
: This site is at position (5,3) in the toroid.

[0
: This journey has been replicated 0 times

148179
: The journey time represented by this journey

5
: The value of *k* in *k opt* in this journey. Note that this value is the *k* in the code of the Kloner machines in this Journey. This value might well be different from the value in the journey that was the parent of this one, but this is the value that will be used when mutating the route of the current journey.

(1,2,3,4...
: This is the sequence of the cities in this specific object.

In addition to this output, running the jar file will also result in the generation of a log file. This may contain a very large amount of data recording every single replication and suicide event. The "level" of the data recorded[5] in the log file, and the name of the log file, may be changed using the parameters in the properties file.

4.3.1. Logging

The log file produced by the TSP code includes the following components. Below there is an (often truncated) example of a log file line and an explanation of the underlying cause.

[5] Note that if a very large amount of the data is logged, then it has the parallel effect of making the entire execution run as a single thread, as flushing the logger takes overall precedence.

```
Jun 28, 2016 12:23:35 PM EvoEvo.york.machineMetaModel.Machine Initialise
```

In general each record in the log file appears on a separate line. Each such line is preceded by another line that contains information about what specific method is being run and at what time it was run. In this case the static initaliser component of the *Machine* class is being executed.

```
INFO: Initial population: 20
```

This record, which is produced in the INFO log level and all finer ones, records the initial Journey population of this run.

```
FINE: {1467113015468} Adding destination Journey2DSite: <0>  [3:1]  as replication target of source site Journey2DSite: <1>  [2:0] [Journey: [0, 163706, 5] (1,11,...)]
```

This line is produced when some *Journey* informs its containing world that it would like to be considered for replication into another site. The other information provided is:

1. The result of *System.currentTimeMillis()*.
2. The description of the target site (in this case [3:1] which is currently empty).
3. The description of the site wishing to replicate, this includes the exact information discussed earlier in this section.

```
FINE: {1467113015605} Committing suicide: Journey2DSite: <3>  [7:8] [Journey: [0, 170924, 5] (1,10,11,41,19,18,39,42,31,22,5,21,...)]
```

Here the journey in a particular site is committing suicide. This particular execution is using the programmedDeath concept discussed in section 4.2. Similar lines appear if programmedDeath is not in use, but the line records that the journey is being killed by code outside the *Journey* itself.

```
FINE: {1467113015618} Replicated parent Journey2DSite: <0>  [2:0] [Journey: [1, 163706, 5] (1,11,12,43,5,36,...)] into child Journey2DSite: <1>  [1:0] [Journey: [0, 165450, 4] (1,11,12,...22,1)]
```

Here a particular site (the one in [2:0]) is being replicated into the adjacent site at [1:0]. As usual, the same information is recorded for each Journey and here it can be seen that the route has been mutated and, in fact, produced a new Journey that is actually worse than the parent. (A journey time of 165450 whereas that for the parent was 163706.)

4.4. Refinement of the TSP approach

As a further example of the use of the EvoMachina framework, this section includes details of the extension of the existing example to support a new underlying mechanism: the use of the Microbial GA[6] approach to optimisation.

Configuration parameter

In order to configure EvoMachina/TSP so that the Microbial genetic algorithm is used a new property is added so as to configure the code. As such, the properties file contains:

```
programmedDeath = false
```

This parameter tells the EvoMachina/TSP code that the usual "programmedDeath" approach to death of Journey individuals is not to be used.

[6] Harvey, I. (2009), The Microbial Genetic Algorithm., *in* George Kampis; István Karsai & Eörs Szathmáry, ed., 'ECAL (2)', Springer, pp. 126-133 .

Remove programmed death

The code for the Journey class, the specialisation of Individual, is modified to that the Journey objects do not decide for themselves whether, or not, to die when they are run in the case where programmedDeath is set to false:

```
        boolean shouldDie = false;
        if (Simulation.GetValue("programmedDeath", false)) {
            // This code, which decides whether a Journey should die, is
            // ignored if programmedDeath is set to false. By default, it's
            // assumed to be true
        }
```

Create a new test

It's essential that all extensions of the framework are properly tested. So, the first thing to do is to write a test, and ensure that it fails.

The code for this test is all shown here. The first thing to do is to define that a new test exists using the annotation that allows the TestNG code to see that it's a test:

```
// Annotation that defines the next method to be a TestNG test with a
// specific priority
@Test (priority = 100)
```

At the start of the code for the test, which will be done in a "bucket" space which is just a collection of Journey objects with no notion of position in a world, the space is created (which is a new class MicrobialGATSPSpace) and populated with a specified number of Journey objects, each with a random initial route. The number of journeys created is defined by the numMGAJourneys parameter in the execution properties.

```
// Define new test method:
public void singleThreadedSearchInBucketSpaceUsingMicrobialGA()
{
   // Create SearchableSpace for MGA approach using new space class:
   SearchableSpace world = new MicrobialGATSPSpace();

   // Create a complete set of journeys in the new space:
   int numJourneys = Simulation.GetValue("numMGAJourneys", 100);
   KlonerDomain kDomain = new KlonerDomain("Kloner domain", Kloner.class);
   for (int i = 0; i < numJourneys; i++)
   {
      List<Pearl> route = createInitialRandomRoute();
      this.makeJourneyWithMutatingCopier(route, world, _cityDomain, kDomain);
   }
```

Now we assert that the world has the expected number of journey objects:

```
   assertEquals(world.numIndividuals(),
                numJourneys,
                "Number of journeys in MicrobialGA");
```

Now we loop around until a specific time has elapsed or the best journey is better than some threshold. On each iteration we ask the world object to provide the current best result and print out that result, using the toString() method implemented by the Journey class, every time that a predefined number of iterations has elapsed.

```
   Journey best;
   long time = System.currentTimeMillis();
   int searchCount = 0;
   do
   {
      best = (Journey)world.search().get();
      if (searchCount++ % 100 == 0)
      {
         System.out.printf("Best of %d is: %s%n", world.numSubspaces(), best);
      }
   } while (best.journeyTime() > Simulation.GetValue("targetTime", 35500)
           &&
```

```
            System.currentTimeMillis() <
                (time + Simulation.GetValue("totalRunTimeInMilliseconds",
                                            60000)));
```

The test concludes by logging the result, taking care to flush the logger queue, and asserting that the world still contains the same number of Journey objects and that the result of execution has shown at least some convergence on the expected solution:

```
    Machine.FlushLogger();
    _logger.fine(String.format("{%d} Completed, best is %s",
                               System.currentTimeMillis(), best));
    Machine.FlushLogger();

    assertEquals(world.numSubspaces(),
                 numJourneys,
                 "Number of journeys in MicrobialGA at end");
    assertTrue(best.journeyTime() <
                    Simulation.GetValue("microbialGASuccessThresholdTime",
                                        50000),
              "MicrobialGA threshold time");
    System.out.println("Best: " + best);
```

Execute new test

Executing the new test, as expected as the new space class has not been defined over and above anything necessary for the code to compile, causes and immediate failure by exception. In this case, the failure is due to the fact that a default implementation of the search() operation required by the SearchableSpace interface answers null.

Implement MicrobialGATSPSpace class

This new class is an extension of the basic Space class, and implements the methods required by the SearchableSpace interface. Constructing an instance of the class just ensures that it's the "topmost" space in a hierarchy:

```
/** A Searchable space that implements the Microbial GA algorithm for finding
TSP solutions */
public class MicrobialGATSPSpace extends Space implements SearchableSpace {
   public MicrobialGATSPSpace() {
      super(Optional.empty());
   }
```

One method provides the currently "best" solution within the space, which just compares all the subspaces of the object, which are all Journey objects, finding the one that provides the minimum journey time:

```
@Override
public Optional<Individual> best() {
    Optional<Space> best = this.getSubspaces()
                               .stream()
                               .min((j1, j2) ->
                                     (int)(((Journey)j1).journeyTime() -
                                          ((Journey)j2).journeyTime()));
    return Optional.of((Individual)best.get());
 }
```

That does not provide a searching capability though and the test, when executed, merely loops around providing the details of the initially best solution:

```
Best of 100 is: Journey: [0, 135815, 5] (1,2,3,4,5,6,7,8,9,10,11,12,13,14
Best of 100 is: Journey: [0, 135815, 5] (1,2,3,4,5,6,7,8,9,10,11,12,13,14
Best of 100 is: Journey: [0, 135815, 5] (1,2,3,4,5,6,7,8,9,10,11,12,13,14
Best of 100 is: Journey: [0, 135815, 5] (1,2,3,4,5,6,7,8,9,10,11,12,13,14

java.lang.AssertionError: MicrobialGA threshold time expected
                                           [true] but found [false]

Expected :true
Actual   :false
```

Implementing the search() operation in the SearchableSpace interface provides the searching capability:

```
@Override
/** Implement the microbial GA algorithm which is to:
 *    1) Randomly select a pair of individuals
 *    2) Remove the least fit of that pair and replicate the other
 *  which leaves the population of the space the same as it
 *  was at the start of the process  */
public Optional<Individual> search() {
    // Select two individuals:
    Journey j1 = (Journey)this.getSubspace(
                                  ThreadLocalRandom
                                       .current()
                                       .nextInt(this.numSubspaces()));
    Journey j2;
    do {
        j2 = (Journey)this.getSubspace(
                              ThreadLocalRandom
                                    .current()
                                    .nextInt(this.numSubspaces()));
    } while (j1.equals(j2));

    // Work out which is the best and which the worst of the two:
    Individual best = (j1.journeyTime() < j2.journeyTime()) ? j1 : j2;
    Individual worst = best.equals(j1) ? j2 : j1;

    // Remove the worst from the container and replicate the best:
    this.removeSubspace(worst);
    this.addSubspace(best.replicate());

    // Answer the best journey, which may now be better than when we started:
    return this.best();
}
```

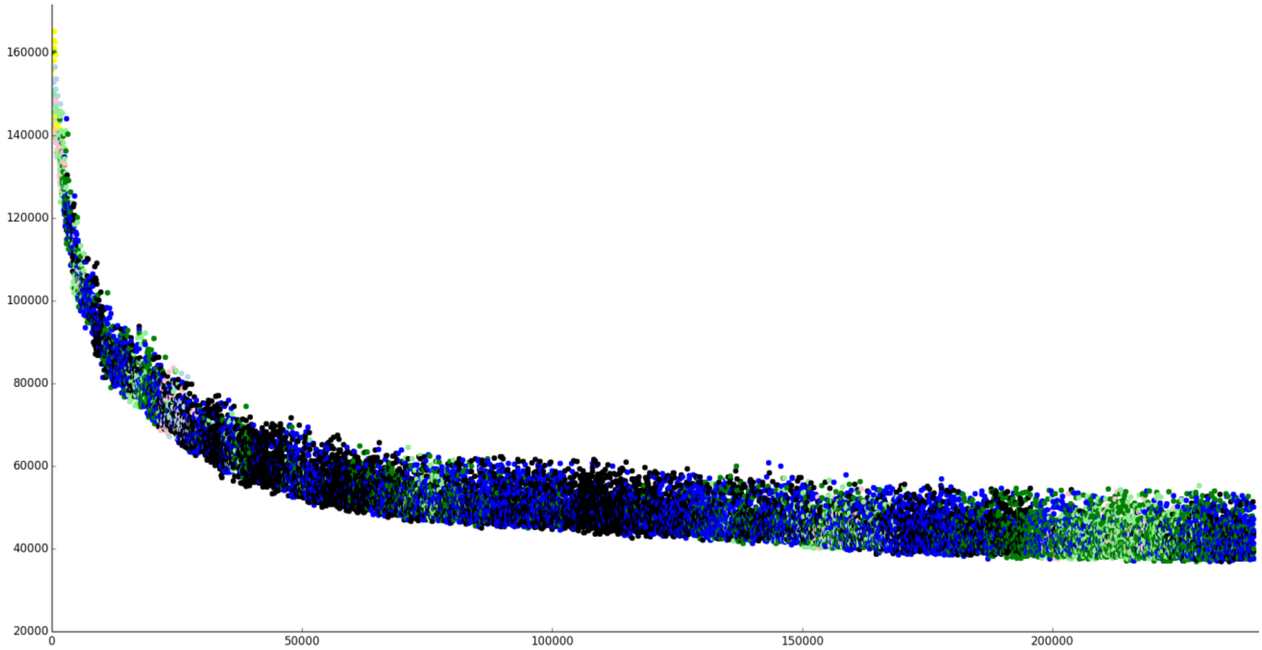
Now, when executing the test useful results are returned:

```
Best of 100 is: Journey: [13, 36234, 2] (1,9,40,11,23,14,34,3,22,16,41,2,4,26,10,24,45,35,42,
Best of 100 is: Journey: [13, 36234, 2] (1,9,40,11,23,14,34,3,22,16,41,2,4,26,10,24,45,35,42,
Best of 100 is: Journey: [14, 36234, 2] (1,9,40,11,23,14,34,3,22,16,41,2,4,26,10,24,45,35,42,
Best of 100 is: Journey: [8, 36234, 3] (1,9,40,11,23,14,34,3,22,16,41,2,4,26,10,24,45,35,42,2
Best of 100 is: Journey: [10, 36234, 3] (1,9,40,11,23,14,34,3,22,16,41,2,4,26,10,24,45,35,42,
Best of 100 is: Journey: [14, 36234, 2] (1,9,40,11,23,14,34,3,22,16,41,2,4,26,10,24,45,35,42,
Best of 100 is: Journey: [12, 36234, 3] (1,9,40,11,23,14,34,3,22,16,41,2,4,26,10,24,45,35,42,
Best of 100 is: Journey: [12, 36234, 2] (1,9,40,11,23,14,34,3,22,16,41,2,4,26,10,24,45,35,42,
Best of 100 is: Journey: [14, 36234, 2] (1,9,40,11,23,14,34,3,22,16,41,2,4,26,10,24,45,35,42,
Best of 100 is: Journey: [15, 36234, 2] (1,9,40,11,23,14,34,3,22,16,41,2,4,26,10,24,45,35,42,
Best of 100 is: Journey: [16, 36234, 2] (1,9,40,11,23,14,34,3,22,16,41,2,4,26,10,24,45,35,42,
Best of 100 is: Journey: [18, 36234, 2] (1,9,40,11,23,14,34,3,22,16,41,2,4,26,10,24,45,35,42,
Best of 100 is: Journey: [21, 36234, 2] (1,9,40,11,23,14,34,3,22,16,41,2,4,26,10,24,45,35,42,
Best of 100 is: Journey: [12, 36234, 3] (1,9,40,11,23,14,34,3,22,16,41,2,4,26,10,24,45,35,42,
Best of 100 is: Journey: [15, 36234, 4] (1,9,40,11,23,14,34,3,22,16,41,2,4,26,10,24,45,35,42,
Best of 100 is: Journey: [16, 36234, 4] (1,9,40,11,23,14,34,3,22,16,41,2,4,26,10,24,45,35,42,
Best of 100 is: Journey: [18, 36234, 4] (1,9,40,11,23,14,34,3,22,16,41,2,4,26,10,24,45,35,42,
Best of 100 is: Journey: [14, 36234, 4] (1,9,40,11,23,14,34,3,22,16,41,2,4,26,10,24,45,35,42,
Best of 100 is: Journey: [16, 36234, 4] (1,9,40,11,23,14,34,3,22,16,41,2,4,26,10,24,45,35,42,
Best: Journey: [0, 35414, 2] (1,9,40,11,23,14,34,3,22,16,41,2,4,26,35,45,24,10,42,29,5,48,32,

===============================================
Default Suite
Total tests run: 1, Failures: 0, Skips: 0
===============================================
```

4.5. TSP results

Producing bulk results can be done by analysing the log files produced by EvoMachina. This plot, produced using an awk script to analyse the log file and some subsequent Python code, shows how the value for the journey time decreases over time. This plot shows how the value descreases for each new instance of Journey with time, in milliseconds on the x axis. The colour of the blobs indicates the value of *k* used to replicate the individual with darker colours representing smaller values of *k.*



5. Specialisation for a new problem

The process that is expected to be following for configuring the EvoMachina framework to a new problem is outlined here. Note that all metamodel classes in the framework are in the package `EvoEvo.york.machineMetaModel`. Similarly, all class in the example TSP implementation, which includes the TestNG test suite, are in the package `EvoEvo.york.tspTest`.

The expected, although probably not necessary, procedure is as follows. Many details can be seen by inspecting the source code, and the generated javadoc, for the `EvoEvo.york.tspTest` package.

1. Create a new package to contain all classes that use, and are extensions, of the metamodel.
2. Write TestNG tests to exercise the new domains, machines and spaces. Ensure that they all fail.
3. Write extension classes for:
    a. The necessary domains and pearls.
    b. The necessary machines, over and above the standard ones such as Translator.
    c. The spaces and related topics such as site and searchable spaces
4. Execute all tests, including those defined for the metamodel classes themselves, and ensure they all pass. This will likely require some iteration of this list.
5. Write a harness, as exemplified by the TSP main class, for execution of the model.