Game Environment for Command and Control Operations (GECCO)

Developer's Manual

Joel Brynielsson, Henrik Bäärnhielm, Andreas Enblom, Jing Fu Zi, Niklas Hallenfur, Karl Hasselström, Henrik Hägerström, Oskar Linde, Klas Wallenius and Jon Åslund

Department of Numerical Analysis and Computer Science
Royal Institute of Technology
SE-100 44 Stockholm
Sweden
gecco@nada.kth.se

May 14, 2001

Contents

1 Introduction

GECCO is a platform for creating and playing real-time games where units of various kinds interact with each other and the underlying ground layer. GECCO can be used to create large-scale wargame scenarios on the operative level, with a continent as the operating environment, as well as creating a fire scenario in a specific block in a certain city, where firetrucks, helicopters and ambulances have to work together to save people from a burning building. This document describes how to implement such games and scenarios using the GECCO platform. Good knowledge of the Java programming language is essential when creating a game, and also basic knowledge of the concept of finite-state automatons.

2 Terminology

Server	The GECCO server executes a scenario. It is responsible for handling the course of events in the scenario, and for dispatching the correct information to the clients connected to the server. Clients connect to the server in order to take part in a scenario. The server and the clients communicate through the TCP/IP protocol.
Client	A GECCO client is an interface for observing and commanding units of a scenario. A client can be a GUI program, presenting the current state of the scenario to a human user and letting her control the available units, or it might be an artifical intelligence client of some kind, implementing a strategy for the units it is allowed to control. From the server's point of view, there is no difference at all between the different types of client.
Game	A game is a well-defined set of unit types and actions. It is the basis on which to build a scenario. Implementing a game mainly involves programming in the Java language, subclassing different classes in the GECCO package in order to define unit types and actions.
Scenario	A scenario defines everything the server needs to execute correctly. A scenario is dependent on a game, with defined unit types and actions. In contrast with the implementation of a game, creating a scenaro is mostly done by editing text files. For example, the text files define instances of the unit types (units) implemented in the game, what roles the scenario consists of and which roles are commanders and observers for the units of the scenario, what icons to use for the unit types in a GUI client, and where to find the image to use for the map and how it should be processed.
Game/Scenario creator	A person implementing a game, a scenario or both.
Role	A role is a player in a scenario. Roles are defined in the scenario configuration text files. Clients connect to the server using one of the roles defined in the scenario. Sometimes, depending on the definition of the role in question, multiple clients can connect as the same role.
Unit type	A unit type is a definition of a unit to be used in a scenario. A unit type is defined in both the game and the scenario. The game defines the unit type implicitly with a Java class that should be used by the server when units of the unit type in question is used in the scenario text files. A unit type also has a visibility range, which means that the unit will be able to see everything (automatons and other units) within that range. One can also define properties such as health or fuel for a unit type. The scenario text files also define the available actions for the unit types. A unit type must also have an event handler, which takes care of all incoming events to units of that type.

Unit	A unit is an instance of a specific unit type. Units are defined in the scenario text files. A unit has at most one commander, but it can have any number of observers. A unit inherits the properties defined in the corresponding unit type, but it is also possible to change the value of a specific property for each unit (instance of the unit type).
Commander role for a unit	The commander role for a unit is the only role that is allowed to control the unit.
Observer role for a unit	An observer role for a unit can see everything the unit can see, i.e. all automatons and units that are inside the unit's visibility range. An observer role can also see the unit's properties.
Map	The map consists of a set of automatons. It is a bitmap picture where each pixel is represented by an automaton. This way, the map can change and evolve as the automatons change state, since the automatons affect each other. The map is parsed from a bitmap picture, for example a GIF, PNG or JPEG image. Colors in the given picture then map to different start states for the automaton in the corresponding position. The color-to-state mapping is defined in the scenario configuration text files.
Automaton	The automaton used in GECCO is a finite-state automaton. Different events affect the automaton in different ways, perhaps forcing the automaton to change state. For example, assume that the automaton X is in a state representing a forest, and the automaton Y is in a state representing a rock. An event representing fire might then force automaton X to change state to one representing a forest set on fire, but automaton Y will most probably stay unaffected by the event, still in the state representing a rock. In GECCO, adjacent automatons may affect each other. In the given example, automaton X might (after being set on fire) affect an adjacent automaton Z to start burning too.
Action	An action is a behavior, for example "Move" or "Attack". It is represented by a Java class, which executes the action. An action can take either a unit, a position on the map or nothing at all as argument. There are two types of actions in GECCO, ordinary actions and instantaneous actions. A unit can only execute one ordinary action at the time. Instantaneous actions are carried out immediately, and can thus be executed even if a unit is executing an ordinary action at the same time.
Event	All entities in a GECCO scenario can affect each other. A unit can affect other units as well as automatons. An automaton can affect adjacent automatons as well as all units that at a given moment are located on the automaton in question. Affect, in this case, means that an event is sent from one entity to another. For example, if unit X attacks unit Y, an event called "Attack" (for example) is sent to unit Y. An event always has a name and a factor, specifying the strength of the event.

Event handler	All unit types in a scenario/game must have an event handler. The event handler takes care of all incoming events to an instance of the unit type. For example, if unit Y is attacked by unit X, an event called "Attack" with factor 10 might reach unit Y. The event handler for Y's unit type then recognizes the event name, and sees that is is of strength 10 and acts accordingly. It might for example reduce the health of unit Y. Events to automatons are handled by the automaton class itself. Therefore, there is no event handler for the automaton used in a scenario/game.
Act of God	An Act of God is a predefined event in a scenario. It might for example be that ten minutes after the game has been started, an event called "Fire" will reach the automaton at a specific position, thereby starting a fire in a forest.

3 The layers of the game

A GECCO scenario consists of two layers. The automaton layer is the map of the game – the ground. Each pixel in the map is represented by an automaton. On top of the automaton layer is the unit layer. The units are controlled by the roles in the scenario. The automatons, on the other hand, are not under direct control by any role, although they can be affected by actions performed by units, which in turn are controlled by a client playing a role in the scenario.

3.1 The automaton layer

As stated earlier, the map in the game is a bitmap picture, where each pixel represents an automaton. All automatons can affect adjacent automatons in different ways, all depending on the Java implementation of the automaton. It is important to understand that the same Java class is used to represent all automatons. One doesn't specify three different Java classes to represent three different automatons for forest, water and road. They have to be merged to one single automaton class.

The color of a pixel in the map is given by the automaton state. When an automaton changes state, the new color is sent to all clients who at that moment can see the automaton in question.

3.2 The unit layer

The unit layer consists of all units defined in the scenario.

4 Interaction between components of a scenario

The different components in a scenario are units and automatons. All components can affect each other with events. An event has a name and an integer representing the strength of the event. There are two types of events in GECCO, automaton events and unit events. Unit events are sent when the affected component is a unit, and automaton events are sent when the affected component

is an automaton. The only special case is when automatons affect adjacent automatons. This is done in a different way.

4.1 Automaton events

Automaton events are sent by units and through "Acts of God", defined in the scenario configuration text files.

4.2 Unit events

Unit events are sent by units to other units, and by automatons that are in a state that may affect units, for example a burning forest.

5 Building a game

In order to create a game, one has to implement a number of Java classes. First of all, one has to implement the automaton that is to be used in the game. Second, one has to implement a Java class for all unit types that are to be used in the game. Third, an event handler for each unit type has to be implemented, and fourth, one has to implement one class for each action that one or more unit types are to perform. All classes have to be implemented by subclassing the corresponding class in the GECCO package. The classes mentioned in the following subsections all reside in the package `server.jar`.

It is advisable to read the *GECCO User's Manual*[1] and browse the class descriptions in the Javadoc documentation prior to reading the following subsections.

5.1 Automaton

In order to create an automaton, you have to subclass the class `server.automaton.Automaton`.

The automaton of a game must be able to handle all the states one wishes to represent in the map. The state itself is internally represented by an integer. It is not advisable to add non-static instance variables in the automaton class. If the map is set to a size of 1000 by 800 pixels, for example, the GECCO server will instantiate $800,000$ automatons, and a vast amount of memory will be needed to cope with the extra instance variables in each of the $800,000$ automatons.

An automaton can change state in two ways. The first way is through the means of an `AutomatonEvent`, thrown by a unit or an act of god. Therefore, every implementation of an automaton must implement a method for handling incoming events to the automaton. The second way to change state is to react upon adjacent automatons states. This is done by implementing a method called `update(..)`.

When one of the two methods mentioned above has been called and taken care of, the automaton can choose to *requeue* itself and/or all adjacent automatons. The GECCO platform holds all automatons in a large priority queue, and calls the `update(..)` method for an automaton at the time it has been scheduled. This is the way to make adjacent automatons react to changes in the surrounding environment.

For example, if automaton X starts to burn due to an `AutomatonEvent`, it knows that it is to burn down in three seconds, according to the rules of the game. But, it also knows that the fact that it is burning means that surrounding automatons which aren't burning should be updated in the near future. It therefore requeues all surrounding automatons that aren't burning already for updating in two seconds, and also requeues itself for updating in three seconds. After two seconds, the GECCO platform calls the `update` method for the surrounding automatons of automaton X. They all start to burn, affected by the fact that automaton X is burning, and requeues all their neighbours and themselves. After one more second, automaton X is updated once again, and switches from the burning state to a burnt-down state. It requeues neither itself nor the adjacent automatons, since it's no longer in a state that affects neighbouring automatons.

5.1.1 The *initialize* method

The `initialize(int initialState)` method is called once, when the automaton is instantiated. The only thing it should do, after calling the corresponding method in the superclass, is to set the current color of the automaton via the `setCurrentColor(int r, int g, int b)` method.

An example is as follows. We assume that we have a helper method which sets the current color given the state as an argument.

```
public void initialize(int initialState) {
    super.initialize(initialState);
    setColor(initialState);
}
```

This leads us to the first golden rule.

Golden rule 1 *Do* not *forget to override the initialize method and set the color using the* setCurrentColor(..) *method during initialization.*

5.1.2 The *update* method

When the `update(int[][] neighbourStates)` method is called, the states of the surrounding automatons are given in a 3 by 3 integer array. All indices that represent automatons that don't exist, i.e. are outside the map boundaries, will be set to -1.

The behavior of the automaton when the `update(..)` method is called should be as follows:

1. Check if any of the adjacent automatons is in a state that affects this automaton, for example "Fire".
2. Change state if neccessary.
3. If this automaton is in a state that should change by itself in the future, without being influenced by adjacent automatons (for example a burning forest which burns down after two seconds), this automaton should be requeued.

4. If the automaton is in a state that could affect adjacent automatons, then every adjacent automaton which is in a susceptible state should also be requeued, so that their `update(..)` methods are called in the near future.

An example of an update method follows below. We assume that we have two helper methods, one called *checkNeighbourstatesForState(..)* which checks if any of the adjacent automatons is in a certain state, and *fireReturn(..)* which requeues all adjacent automatons that are in a non-burning state.

```
public AutomatonReturn update(int[][] neighourStates) {
    switch(getState()) {
        case TREE:
            // The automaton represents a tree. If any of
            // the surrounding automatons is in the BURNING_TREE
            // state, then switch to the BURNING_TREE state.
            if (checkNeighbourstatesForState(neighourStates, BURNING_TREE)) {
                setState(BURNING_TREE);
                setColor(BURNING_TREE);
                UnitEvent event = new UnitEvent("FIRE", 10);
                return new AutomatonReturnQueueSelfAndNeighbours(2.0,
                                                                 2.0,
                                                                 event);
            }
            break;
        case BURNING_TREE:
            // The automaton represents a burning tree.
            // Use the Math.random() function to set the
            // probability of switching to the BURNT_DOWN_TREE
            // state to 0.1
            if (Math.random() > 0.1) {
                // Do not swich state, requeue self
                // and neighbours that aren't burning.
                return fireReturn(neighourStates);
            }
            // We've burnt down, switch to the BURNT_DOWN_TREE state
            setState(BURNT_DOWN_TREE);
            setColor(BURNT_DOWN_TREE);
            break;

        // Do nothing for the rest of the states
        case BURNT_DOWN_TREE:
        case MOUNTAIN:
        case GROUND:
        case WATER:
        default:
            // UNDEFINED
        }
        return new AutomatonReturnNoAction();
}
```

5.1.3 The *handleEvent* method

The `handleEvent(AutomatonEvent event)` method handles incoming events to the automaton. The behaviour when an `AutomatonEvent` reaches the automaton should be as described below.

1. Check if the incoming event affects the automaton. For example, if the event is called `FIRE`, the automaton knows from the rules of the game that if it is in a state sensible to fire, it should switch to the burning state. Check the strength (factor) of the event if necessary.
2. If the event caused a state transition, and the new state is a state that can affect other automatons, then the automaton should requeue itself and all of its neighbours.

An example follows below.

```
public AutomatonReturn handleEvent(AutomatonEvent event) {
    // Handle a FIRE event
    if (event.getEventName().equals(FIRE)) {
        if (getState() == TREE) {

            // Switch to the BURNING_TREE state
            setColor(BURNING_TREE);
            setState(BURNING_TREE);

            // Queue the automaton and all it's neighbours.
            // Create a UnitEvent to send to all units located
            // on this automaton.
            UnitEvent unitEvent = new UnitEvent("FIRE", 10);
            return new AutomatonReturnQueueSelfAndNeighbours(3.0,
                                                             2.0,
                                                             unitEvent);
        }
    }

    // If it wasn't a FIRE event, do nothing.
    return new AutomatonReturnNoAction();
}
```

5.1.4 Requeuing with *AutomatonReturn* objects

As you might have noticed, the *update* and the *handleEvent* methods use the return variable for telling the GECCO platform if and when to requeue the automaton itself and the surrounding automatons. Sometimes, a *UnitEvent* is also returned.

There are five types of *AutomatonReturn* objects. One of them always has to be returned after a call to the *update* or the *handleEvent* method. They all take doubles as arguments. The double values represent the time in seconds until the next call to the *update* method for the corresponding automaton is made. The different types are as follows:

AutomatonReturnNoAction	Neither the automaton itself nor its neighbors are requeued.
AutomatonReturnQueueSelf	The automaton itself is requeued.
AutomatonReturnQueueNeighbours	All neighbors of the automaton are requeued. The time to the next call to the *update* method is the same for all neighbors.
AutomatonReturnQueueSelfAndNeighbours	The automaton itself and all its neighbors are requeued. The time to the next call to the *update* method is the same for all automatons.
AutomatonReturnQueueSelective	The most complex return type. All adjacent automatons and the automaton itself can be requeued, at different times. The constructor takes a 3 by 3 array of doubles. The indices into the array represent the surrounding automatons. The automaton itself is the automaton in the center of the two-dimensional array. If an entry in the array is set to a positive value, the corresponding automaton will be rescheduled to wake up in the same amount of seconds as the given value. If the value in an entry is set to a negative value, the corresponding automaton will not be requeued.

It is advisable to use the *AutomatonReturnQueueSelective* return type as often as possible. It might be very costful to always requeue all adjacent automatons, instead of only requeuing the neighboring automatons that might be affected by the current state of the automaton. For example, assume that an 1000x1000 area consists of automatons where 1/6 of the automatons are in a burning state. The other automatons is in a burnt-down state. If you requeue all neighbours for each call to the *update* method, many automatons that are in a burnt-down state will be requeued as well, stealing CPU time from the rest of the server.

Golden rule 2 *When an automaton is in a state that might affect adjacent automatons, only requeue the adjacent automatons that might be affected.*

In each return type, all constructors come in two versions; one doesn't take a *UnitEvent* as an argument, and one that does. If a *UnitEvent* is given to the constructor, that event will be sent to all units located on the automaton in question. For example, assume that the automaton X is set on fire. The unit Y is located on automaton X, and doesn't move. There must be some way of continously simulate the fact that unit Y should take damage since it's located in the middle of a burning forest. This is accomplished by returning a *UnitEvent* with the *AutomatonReturn* class each time a call to the *update* method is made.

Golden rule 3 *If the automaton is in a state that may affect units, always include a UnitEvent in the return variable.*

5.1.5 State versus color

Earlier it has been said that the state of the automaton is represented by an integer. But, how does the integer relate to the color presented in the client's GUI interface? First of all, there is a private instance variable in the automaton superclass called `currentColor`. This is the color that will represent the current state in the client GUI. When the state of the automaton changes, this instance variable *must* be set to a new value in the automaton subclass. The variable is set via the `setCurrentColor` method in the superclass.

The thing is that the current color that should be presented on the client side is separate from the state integer. This is for different reasons. Although it has been stated earlier that is isn't advisable to add instance variables to the automaton due to the excessive memory usage it will imply, this might sometimes be the only way of implementing a certain behaviour. Say for example that we want each automaton to have a float or a double as an instance variable, and we want the float/double value affect the color that is representing the automaton state in the client GUI. But, the GECCO platform doesn't care for changes in instance variables it doesn't know exists. The platform doesn't send any messages to any clients until the state integer has been changed.

This gives us another golden rule.

Golden rule 4 *Always set the current color to a new value after having set a new state, otherwise the color shown in a client GUI will be the same as the old one. On the other hand, if you want the color of the automaton to change in the client GUI, you also have to change state.*

Please see appendix A for a complete automaton implementation.

5.2 Unit types/Units

As far as the implementation of a game, not a scenario, is concerned, there is not much you have to do to implement units/unit types. On the game level, all you have to do is to implement a subclass of the class `server.core.Unit` for each unit type you want to use in the game. Although it isn't technically necessary to implement one class for each unit type, it is highly recommended.

The only thing you have to do when implementing the unit classes is to decide what properties the unit should have, and which of those should be visible to the clients. Properties that you want to present in a client GUI must be set using methods in the superclass. You can only use double, integer or string values for these properties. Properties that you don't want to be presented in a client GUI is simply set as instance variables in the class.

Say for example that we want to implement a class for representing a helicopter, as in the following example:

```
package gecco.test;

public class Helicopter extends server.core.Unit {

    public int attackRange = 30;
    public double stepLength = 2.0;
```

```
    public Helicopter() {
        super();
        setProperty("Fuel", 100.0);
        setProperty("Health", 100.0);
    }
}
```

In this example we have two properties that will be shown in a client GUI, Fuel and Health, both typed as doubles. We also have two properties that won't be presented to the client, namely the attackRange integer and the stepLength double. Please see appendix B for implementations of both a helicopter and a tank unit.

There are also a vast amount of methods implemented in the unit superclass that are used by the event handler for the unit and by the actions that the unit can perform. Examples of their application follow below. Please see the Javadoc documentation for more information.

5.2.1 Unit properties

Earlier it has been stated that if a unit is to have properties which should be presented in a client GUI, you have to set them with the *setProperty* method in the unit superclass. This is not entirely true. There is another way of defining properties that should be visible to the client, namely through definitions in the scenario configuration text files. This is very convenient when you don't want every instance of the class, i.e. units in the scenario, to have the same initial value for each property. Please see the section describing how to build a scenario for a more detailed description on how to define unit properties. If you define the properties this way, you should not define them in the constructor of the unit class.

Golden rule 5 *Properties that should be visible to a client can be defined in two different ways. They can be defined in the constructor of the unit class, or they can be defined in the scenario configuration text files. The properties can be typed in three different ways; as strings, as doubles or as integers.*

Regardless of the way you define the unit properties, you can have only have three different types of properties, namely string properties, integer properties and double properties. Properties that you don't want to be shown in a client GUI are put as instance variables in the class, and can be of any Java type.

Golden rule 6 *Properties that shouldn't be visible to a client should be put as instance variables in the unit class.*

When getting and setting a property visible to the client, you must use the helper methods in the superclass.

5.2.2 Block/unblock actions

In the scenario configuration text files it is defined which actions a certain unit can execute. But, sometimes the set of actions isn't static, you might want to block or unblock a certain action. For example, assume that we have defined a

unit representing a tank, with a property called Fuel, just as in the helicopter example above. When the tank moves, i.e. an action is executed, the Fuel value is most probably decreased. When it reaches zero, you don't want the tank to be able to move, so you block the action with a call to the superclass' *addBlockedAction* method. Another example might be that a jet fighter only can carry a limited number of missiles, and when they all have been fired at a target, you don't the fighter to be able to attack another unit, and you therefore block the attack action.

Actions that have been blocked can later be unblocked, for example when the tank in the example above is refuelled. This is done with the superclass' *removeBlockedAction* method.

5.2.3 Destroy unit

Sometimes you want to mark a unit as destroyed, for example if it has been attacked by another unit, and the result is that a defined Health property reaches zero. This is done with the *markUnitAsDestroyed* method in the superclass. This will cause a complete deletion of the unit, from all parts of the game.

5.3 Event handler

Every unit in a GECCO scenario must be able to receive unit events. Therefore, you have to implement a *UnitHandler* that can handle incoming events to the units you intend to use in your game. You can build one event handler per unit class, or a generic event handler that handles incoming events to all implemented units. An event handler must be a subclass of `server.core.EventHandler`.

The event handler only has to implement one method, namely `handleEvent(UnitEvent event, Unit unit)`. This method should handle the incoming unit event for the unit given as an argument.

An example of an event handler that handles incoming events for two unit types, namely *Helicopter* and *Tank*, is given below. The rules for incoming events in the example are:

1. For the event called *FIRE*, a helicopter doesn't take damage at all. For a tank, the Health property should be decreased by 5.
2. For the event called *ATTACK*, both a helicopter and a tank should take damage with the value given as the factor in the event.
3. If a unit's Health property reaches zero, the unit should be marked as destroyed.

The implementation of the rules is as follows:

```
package gecco.test;

// Import all classes in the server.core package
import server.core.*;

public class TankAndHelicopterEventHandler extends EventHandler {

    public void handleEvent(UnitEvent event, Unit unit) {
```

```
        if (event.getName().equals("FIRE") &&
            unit instanceof Tank) {
            // Unit takes damage from fire only if it is a tank
            double health = unit.getDoubleProperty("Health");
            health -= 5.0;
            if (health > 0) {
                // Set new health property for the tank
                unit.setProperty("Health", health);
            } else {
                // The tank is destroyed
                unit.markAsDestroyed();
            }
        } else if (event.getName().equals("ATTACK")) {
            double health = unit.getDoubleProperty("Health");
            health -= event.getFactor();
            if (health > 0) {
                // Set new Health property for the unit
                unit.setProperty("Health", health);
            } else {
                // Unit is destroyed
                unit.markAsDestroyed();
            }
        }
    }
}
```

5.4 Actions

Implementing actions is perhaps the most complex part of creating a game. It is very important to fully grasp the concept of actions, and how they are executed by the GECCO server.

In the terminology list, actions are defined as different behaviors for units – tasks that can be carried out. There are two types of actions, ordinary actions and instantaneous ones. The implementation of the two types doesn't differ much, although they are quite different in other ways.

5.4.1 The *initiate* method

When designing and implementing an action, the first thing to do is to decide what kind of argument the action should take. An action can take a position on the map, a unit, or nothing as an argument. An action representing movement would probably take a position on the map as an argument, whereas an action representing an attack would take a unit as an argument.

If you need some kind of initialization before the execution of your action can begin, override the *initiate* method in the superclass that takes the correct arguments. For example, say that we want to implement an action for moving a unit. The units in the game are the ones that are implemented in appendix B.

Since we take a position on the map as an argument, we override the following superclass method.

```
public void initiate(int _actionHandle,
                     int _unitHandle,
                     double _argX,
                     double _argY) {
    super.initiate(_actionHandle, _unitHandle, _argX, _argY);

    double stepLength;
    // Fetch the stepLength from the unit
    if (getUnit() instanceof Helicopter) {
        stepLength = ((Helicopter) getUnit()).stepLength;
    } else if (getUnit() instanceof Tank) {
        stepLength = ((Tank) getUnit()).stepLength;
    } else {
        // Unknown unit class, set default stepLength value
        stepLength = 1.0;
    }
    // Calculate addX and addY
    double curX = getUnit().getX();
    double curY = getUnit().getY();
    double dx = _argX - curX;
    double dy = _argY - curY;
    double dist = Math.sqrt(dx*dx + dy*dy);
    double coeff = dist / stepLength;
    addX = dx / coeff;
    addY = dy / coeff;
}
```

What we do here is simply some precalculation. The result is put in the instance variables *addX* and *addY*, which is a two-dimensional vector which will be added to the unit's position in each iteration of the action. We also check if the unit which will execute the action is a helicopter or a tank, and set the steplength accordingly.

Golden rule 7 *When designing an action, always start by deciding what kind of argument the action should take, and override the correct initiate method in the superclass if any initialization is necessary.*

5.4.2 The *checkPoint* method for ordinary actions

The concept of ordinary actions is based on the fact that the task the action is to perform can be divided into small parts, where each part takes the action closer to its ultimate target. For an action representing movement, this means that we divide the movement from position X to position Y into many small short movements. The *checkPoint* method is the heart of the action. It is this method that is the small part that brings the action closer to its ultimate goal. This method is called repeatedly, until the action either reports back that it is done, or that it cannot continue.

Golden rule 8 *An action must be designed so that its ultimate task can be divided into many smaller tasks, where each smaller task brings the action closer to accomplishing its ultimate task. The checkPoint method takes care of the execution of the smaller task, and is called repeatedly until the action reports back that it is done or that it cannot continue.*

For our example action, the *checkPoint* method simply moves the unit one small step by adding the two-dimensional vector created in the *initiate* method to the current coordinates of the unit, moving the unit closer to the destination. When we've reached the destination, or if we're very close to it, we report back that we're done. In each call to the *checkPoint* method, we also decrease the unit's Fuel property. If the Fuel property reaches zero, we report back that we cannot continue. If we've neither reached the destination nor run out of fuel, we requeue the action, telling the GECCO server to call the *checkPoint* method in a certain number of seconds.

5.4.3 The *checkPoint* method for instantaneous actions

Instantaneous actions differ from ordinary ones by the fact that the *checkPoint* method is only called once. They can also be carried out while the unit is performing an ordinary action at the same time. Therefore, an instantaneous action can be useful for things like having a B-52 bomber drop bombs while flying over a certain area.

5.4.4 *checkPoint* return and unit messages

As stated earlier, we have three choices when returning from the *checkPoint* method. They are described below. When instantiating one of the three return classes, you always have the choice of including a unit message. A unit message is a string that will be sent to the client, with a mapping between the unit that is performing the action and the string itself. This is very convenient for explaining the current behaviour of a unit for the client, for example when the unit runs out of fuel and the movement action is aborted.

Golden rule 9 *It is advisable to include unit messages in the return class when returning from the checkPoint method. If not, there is a great chance that the user of the client won't fully understand the behaviour of the unit.*

ActionReturnCompleted	This tells the GECCO server that the action is completed.
ActionReturnError	This tells the GECCO server that the action must be aborted due to an error.
ActionReturnRequeue	This tells the GECCO server that the action is not completed, and the next call to the *checkPoint* method should be made in a certain number of seconds, given as an argument to the constructor. This return class can only be used by an ordinary action, not by an instantaneous one.

5.4.5 Examining the automaton environment

Sometimes actions must be able to inspect the automaton environment. In our movement example, we check if the automaton we move to is in a burning state. If so, and if the unit isn't a helicopter, we decrease the health of the unit. This is done with the *getAutomatonState* method in the action superclass.

Golden rule 10 *Do not forget to examine the automaton environment when executing an action.*

5.4.6 Sending events

An action must also be able to send automaton and unit events. This is done with two helper methods in the action superclass, namely *sendEventToAutomaton* and *sendEventToUnit*.

Golden rule 11 *If an action is to affect other units or automatons, for example in an action representing an attack on another unit, the way to do this is by sending unit or automaton events with the helper methods in the action superclass.*

5.4.7 The final movement action implementation

Please see appendix D.

6 Creating scenarios for an existing game

Creating a scenario far an existing game isn't as much work as creating a new game. Creating a new game means creating new functionality, creating a scenario for a game is just about using that functionality.

6.1 Configuration file format

A scenario basically consists of a few text files describing what units and roles there are, how big the map is, and stuff like that. All the configuration files share the same basic format.

6.1.1 Comments

Everything between a hash sign (`#`) or a double slash (`//`) and the end of the line is ignored by the parser. This can be used to insert comments in the configuration files. For example, on reading the lines

```
// This is the initial health of the unit
health = 100; # good health is important!
```

the parser would ignore everyting but this:

```
health = 100;
```

Additionally, everything between a slash and a star (`/*`) and a star and a slash (`*/`) is also ignored. This is useful if you want to write comments that span multiple lines:

```
/* This
   is
   all
   one
   big
   comment. */
```

6.1.2 Properties

A configuration file is just a long list of *properties*, or *name* = *value* pairs. Like this:

```
health = 100;
salary = 3445.25;name=Johnson;
```

Spaces, tabs, line breaks and other whitespace is not significant. Instead, each property is terminated by a semicolon (`;`), and whatever follows is assumed to be the name of a new property. The name and value parts are separated by an equals sign (`=`).

If you want to have a name or value containing space, equals signs, or anything else that is normally ignored or has a special meaning, you can surround it with double quotes (`"`). Like this:

```
"A Really Fancy Property Name" = "a really ugly value #/* $$,+}{  ;//=";
```

Within the quotes, everything is interpreted literally.

6.1.3 Property lists

You can associate an entire list of values with a name, like this:

```
favorite_numbers = 2, 3, 5, 7, 11, 13;
```

The values are separated with a comma (`,`). The list ends with a semicolon, as usual. In fact, the single values we've been using up until now are just lists of length one.

Associating values with the same name more than once is the same thing as associating a list of values to it; thus,

```
food = spaghetti;
food = salad;
```

is equivalent to

```
food = spaghetti, salad;
```

Use the syntax you find most pleasing to the eye.

6.1.4 Property sets

A value can be a set of properties. For example,

```
point = { x = 12; y = 10; };
```

Everything between the braces ({ and }) is interpreted just as usual, then stored as the value associated with the name preceeding the braces.

Naturally, you can create lists of property sets:

```
point = { x = 12; y = 10; };
point = { x = -12; y = 2; };
point = { x = 5; y = -39; };
```

In this case, lists created with the comma syntax are usually hard to read:

```
point = { x = 12; y = 10; }, { x = -12; y = 2; },
        { x = 5; y = -39; };
```

Finally, you can merge two property sets with a plus sign (+):

```
point = { x = 12; } + { y = 10; };
```

This is only really useful when at least one of the sets is a variable (see below).

6.1.5 Variables

Consider the following example: You want to create a list of a large number of property sets that look like this:

```
person = {
   name = "John Doe";
   position = {
      x = 24.6;
      y = 15;
   };
   type = soldier;
   subtype = "cannon fodder";
   weapon = knife, rifle, crossbow, "light sabre";
   clothes = boots, pants, jacket, "fancy hat", underwear;
   health = {
      head = 100%;
      left_arm = 100%;
      right_arm = 100%;
      left_leg = 100%;
      right_leg = 100%;
      torso = 100%;
   };
};
```

Only the name and position is different. To avoid typing the same information over and over again, you can declare a variable with all the invariant things, like this:

```
$person_defaults = {
   type = soldier;
   subtype = "cannon fodder";
   weapon = knife, rifle, crossbow, "light sabre";
   clothes = boots, pants, jacket, "fancy hat", underwear;
```

```
   health = {
      head = 100%;
      left_arm = 100%;
      right_arm = 100%;
      left_leg = 100%;
      right_leg = 100%;
      torso = 100%;
   };
};

person = $person_defaults +
   { name = "John Doe"; position = { x = 24.6; y = 15; }; };
person = $person_defaults +
   { name = "Jane Doe"; position = { x = 27.6; y = 16; }; };
person = $person_defaults +
   { name = "Richard Roe"; position = { x = 28.2; y = 15; }; };
```

The set of invariant properties is merged with the personal properties of each person by the plus, as discussed above. This can be a very useful feature.

Variables are declared just like properties, except that the name starts with a dollar sign (`$`). Once declared, the variable name can be used wherever a value is expected. Its value is whatever was assigned to it in the first place.

6.2 The configuration files

You need to supply five configuration files: `global.conf`, `roles.conf`, `unittypes.conf`, `units.conf` and `actsofgod.conf`. Below we'll cover each of them in turn; however, there are some conventions that are used in all of them.

Some values are expected to be integers, or ints. They should consist of the digits 0-9, optionally preceeded by a minus sign. `23` and `-46236124` are integers.

Some values are expected to be floating-point numbers, or doubles. They are just like integers, but optionally followed by a decimal point and more digits. `23`, `23.54` and `-46236124.563986487` are floating-point numbers.

6.2.1 Global

In the `global.conf` configuration file, you should supply two properties: `defaults` and `map`. `defaults` should be a property set containing one property, `port`. Its value, an integer, determines what port the server will listen for incoming connections on.

`map` should be a property set with the following properties:

`width` The width of the map (an integer).

`height` The height of the map (an integer).

`initial_state` The file name of a bitmap image that provides initial states for the automatons.

`class` The name of the class that determines the behavior of the automatons.

`state` A list of property sets that map colors in the bitmap image to initial automaton states. The properties in each set should be

`number` The number of the state (an integer).

`color` The color to be mapped to that state. This is a property set containing three properties, `red`, `green` and `blue`, each an integer in the range 0-255.

6.2.2 Roles

In the `roles.conf` configuration file, you should supply two properties:

`role` A list of names. The names of all roles in the scenario should be listed here.

`gods_eye` A list of names. The names of all roles with God's Eye privileges should be listed here (and not under `role`). These are just like normal roles, except they can see everything.

6.2.3 Unit types

In the `unittypes.conf` configuration file, there is only one property to define: `unit_type`. It should be a list of property sets, one for each unit type used in the scenario. The property sets should contain the following:

`name` The name of the unit type.

`image` The name of an image file. This is used to supply the clients with a graphical icon to represent this unit type.

`range` The maximum distance at which units of this type can see things (an integer).

`class` The name of the class that determines the behavior of units of this type.

`event_handler` The name of the class that handles events for units of this type.

`action` A list of property sets, one for each action this unit type can perform. Each property set must contain

- `name` The name of this action.
- `description` A short description of this action. This will be presented to the clients instead of the name.
- `class` The name of the class that defines this action.
- `argument` The kind of argument required by this action. This is also the place to tell the server if the action is an ordinary action, or an instantaneous one. For an ordinary action, the valid choices are `POINT`, `UNIT` and `VOID`, and the corresponding choices for an instantaneous action are `INSTANT_POINT`, `INSTANT_UNIT` and `INSTANT_VOID`.

`property` A list of property sets, one for each user-defined property this unit type has by default. Each property set must contain

- `name` The name of this property.
- `type` The type of this property. Can be `DOUBLE`, `INT` or `STRING`.
- `value` The default value of this property. If the type is `DOUBLE` or `INT`, the possible values of `value` are correspondingly restricted.

6.2.4 Units

In the `units.conf` configuration file, there is only one property to define: `unit`. It should be a list of property sets, one for each unit in the scenario. The property sets should contain the following:

`name` The name of the unit.

`type` The name of the unit's type.

`init_x` The initial x position of the unit (a floating-point value).

`init_y` The initial y position of the unit (a floating-point value).

`command` The name of a role, as defined in `roles.conf`. Clients playing that role are empowered to give orders to this unit. This property is optional; if it is missing, no one can order this unit around.

`observe` A list of roles, as defined in `roles.conf`. Clients playing these roles will see what this unit sees, and will also be able to see its properties. If the unit has a commander, that role automatically becomes an observer as well and should not be listed here.

`property` A list of property sets, one for each user-defined property[1]. Each property set must contain

- `name` The name of this property.
- `type` The type of this property. Can be `DOUBLE`, `INT` or `STRING`.
- `value` The value of this property. If the type is `DOUBLE` or `INT`, the possible values of `value` are correspondingly restricted.

6.2.5 Acts of God

In the `actsofgod.conf` configuration file, there is only one property to define: `event`. It should be a list of property sets, one for each event destined to occur in the scenario. The property sets should contain the following:

`type` The type of event.

`position_x` The x position of the event (a floating-point value).

`position_y` The y position of the event (a floating-point value).

`factor` The strength of the event.

`time` The time this event is destined to occur, measured in seconds from the start of the game.

[1] Units inherit all user-defined properties from their unit types. The properties defined here are added to those, or, in case of a name clash, overrides them. The properties defined in the unit type thus serve as default properties.

7 Distributing a game implementation

When delivering a game implementation, three things should be included; the general server package, the client package, and a game implementation package. Furthermore, the game implementation package should contain startup scripts that makes it easier to start the game.

The general server package contains the server components that are common to all games. It consists of a single Java Archive (jar) file, named `server.jar`.

The client package The client package contains the complete client program. Again, it consists of a single Java Archive file, named `client.jar`.

The game implementation package contains all the data specific to the game. It usually has a number of image (`.gif`, `.jpeg` and `.png`) files, a number of configuration (`.conf`) files and a Java Archive, named something like `game.jar` (where "game" is the name of the game). The game implementation should also include startup scripts and documentation about the game, containing information about how the game is started (using the startup scripts), and a general description of the game.

The startup scripts are the only things of the system that are platform-dependent, and should be supplied for those operating systems where the game is likely to run.

When creating the startup scripts for the server, keep in mind that the Java Archive (jar) files that should be loaded are the game implementation Java Archive and `server.jar`. It is important that the game implementation archive is loaded first and that the folder where the game is installed is included in the classpath. The name of the class to be executed is `server.startup.StartServer`.

Startup scripts for the client need only include `client.jar` in their classpaths. The class to execute is `client.Game`.

Make sure the documentation and startup scripts fullfill the expectations described in *GECCO User's Manual*[1].

A An automaton implementation

```
package gecco.test;

// Import the class UnitEvent
import server.core.UnitEvent;

public class AutomatonImplementation extends Automaton {

    // The only event which will affect the automaton
    final static String FIRE = "FIRE";

    // The automaton states
    final static int TREE = 1;
    final static int BURNING_TREE = 2;
    final static int BURNT_DOWN_TREE = 3;
    final static int MOUNTAIN = 4;
    final static int GROUND = 5;
    final static int WATER = 6;


    // Override the initialize method, and set the current color
    // via the setColor(..) helper method
    public void initialize(int initialState) {
        super.initialize(initialState);
        setColor(initialState);
    }


    private void setColor(int state) {
        // Check which state we're in and set the color
        // accordlingly.
        switch(state) {
            case TREE:
                setCurrentColor(10, 170, 10);
                break;
            case BURNING_TREE:
                setCurrentColor(250, 90, 0);
                break;
            case BURNT_DOWN_TREE:
                setCurrentColor(20, 45, 20);
                break;
            case MOUNTAIN:
                setCurrentColor(180, 180, 180);
                break;
            case GROUND:
                setCurrentColor(120, 100, 75);
                break;
            case WATER:
                setCurrentColor(30, 30, 200);
```

```
            break;
        default: // UNDEFINED
            setCurrentColor(200, 0, 0);
            break;
    }
}

public AutomatonReturn update(int[][] neigborStates) {
    switch(getState()) {
        case TREE:
            // The automaton represents a tree. If any of
            // the surrounding automatons is in the BURNING_TREE
            // state, then switch to the BURNING_TREE state.
            if (checkNeighborstatesForState(neigborStates, BURNING_TREE)) {
                setState(BURNING_TREE);
                setColor(BURNING_TREE);
                UnitEvent event = new UnitEvent("FIRE", 10);
                return new AutomatonReturnQueueSelfAndNeighbours(2.0,
                                        2.0,
                                        event);
            }
            break;
        case BURNING_TREE:
            // The automaton represents a burning tree.
            // Use the Math.random() function to set the
            // probability of swithing to the BURNT_DOWN_TREE
            // state to 0.1
            if (Math.random() > 0.1) {
                // Do not swich state, requeue self
                // and neighbors that aren't burning.
                return fireReturn(neigborStates);
            }
            // We've burnt down, switch to the BURNT_DOWN_TREE state
            setState(BURNT_DOWN_TREE);
            setColor(BURNT_DOWN_TREE);
            break;

        // Do nothing for the rest of the states
        case BURNT_DOWN_TREE:
        case MOUNTAIN:
        case GROUND:
        case WATER:
        default:
            // UNDEFINED
    }
    return new AutomatonReturnNoAction();
}

private AutomatonReturn fireReturn(int[][] neighborStates) {
    // Queue all adjacent automatons that is in the TREE state,
```

```
    // meaning that they should be affected by the fact that this
    // automaton is burning

    // Create an array which holds the times when the adjacent
    // automatons should be updated.
    double[][] theArray = new double[3][3];
    for (int x = 0; x < 3; x++) {
        for (int y = 0; y < 3; y++) {
            int nState = neighborStates[x][y];
            if (nState == TREE) {
                // This adjacent automaton is in the TREE state,
                // and should thus be affected. Queue the
                // automaton in 1.0 + random * 2 seconds.
                theArray[x][y] = 1.0 + Math.random() * 2;
            } else {
                // This adjacent automaton is not in
                // the TREE state, and will thus not
                // be affected. Don't queue this automaton.
                theArray[x][y] = -1.0;
            }
        }
    }
    // Now, set the time for next update of this automaton to
    // 1.5 + random * 2 seconds.
    theArray[1][1] = 1.5 + Math.random() * 2;

    // Return with the queue array, and a FIRE UnitEvent.
    AutomatonReturn aR;
    aR = new AutomatonReturnQueueSelective(theArray,
                                           new UnitEvent("FIRE", 10));
    return aR;
}


private boolean checkNeighborstatesForState(int[][] neighborStates,
                                            int checkState) {
    // A helper method for checking if any of the adjacent automatons
    // is in checkState
    for (int x = 0; x < 3; x++) {
        for (int y = 0; y < 3; y++) {
            if (neighborStates[x][y] == checkState)
                return true;
        }
    }
    return false;
}

public AutomatonReturn handleEvent(AutomatonEvent event) {
        // Handle a FIRE event
        if (event.getEventName().equals(FIRE)) {
```

```
                if (getState() == TREE) {

                    // Switch to the BURNING_TREE state
                    setColor(BURNING_TREE);
                    setState(BURNING_TREE);

                    // Queue the automaton and all its neighbors.
                    // Create a UnitEvent to send to all units located
                    // on this automaton.
                    UnitEvent unitEvent = new UnitEvent("FIRE", 10);
                    return new AutomatonReturnQueueSelfAndNeighbours(3.0,
                                                                     2.0,
                                                                     unitEvent);
                }
            }

            // If it wasn't a FIRE event, do nothing.
            return new AutomatonReturnNoAction();
    }

    public UnitEvent getUnitEventForCurrentState(String unitType) {
        // Returns a UnitEvent
        if (getState() != BURNING_TREE)
            return null;
        return new UnitEvent("FIRE", 10);
    }
}
```

B Two implementations of units

```
package gecco.test;

public class Helicopter extends server.core.Unit {

    public int attackRange = 30;
    public double stepLength = 2.0;

    public Helicopter() {
        super();
        setProperty("Fuel", 100.0);
        setProperty("Health", 100.0);
    }
}

package gecco.test;

public class Tank extends server.core.Unit {

    public int attackRange = 15;
    public double stepLength = 0.9;

    public Tank() {
        super();
        setProperty("Fuel", 100.0);
        setProperty("Health", 100.0);
    }
}
```

C An event handler implementation

```
package gecco.test;

// Import all classes in the server.core package
import server.core.*;

public class TankAndHelicopterEventHandler extends EventHandler {

    public void handleEvent(UnitEvent event, Unit unit) {
        if (event.getName().equals("FIRE") &&
            unit instanceof Tank) {
            // Unit takes damage from fire only if it is a tank
            double health = unit.getDoubleProperty("Health");
            health -= 5.0;
            if (health > 0) {
                // Set new Health property for the tank
                unit.setProperty("Health", health);
            } else {
                // The tank is destroyed
                unit.markAsDestroyed();
            }
        } else if (event.getName().equals("ATTACK")) {
            double health = unit.getDoubleProperty("Health");
            health -= event.getFactor();
            if (health > 0) {
                // Set new Health property for the unit
                unit.setProperty("Health", health);
            } else {
                // Unit is destroyed
                unit.markAsDestroyed();
            }
        }
    }
}
```

D A movement action implementation

```
package gecco.test;

// Import all classes in the server.core package
import server.core.*;

public class MovementAction extends Action {

    private double addX;
    private double addY;
    private double stepLength;
```

```
public void initiate(int _actionHandle,
             int _unitHandle,
             double _argX,
             double _argY) {
    super.initiate(_actionHandle, _unitHandle, _argX, _argY);

    double stepLength;
    // Fetch the stepLength from the unit
    if (getUnit() instanceof Helicopter) {
        stepLength = ((Helicopter) getUnit()).stepLength;
    } else if (getUnit() instanceof Tank) {
        stepLength = ((Tank) getUnit()).stepLength;
    } else {
        // Unknown unit class, set default stepLength value
        stepLength = 1.0;
    }
    // Calculate addX and addY
    double curX = getUnit().getX();
    double curY = getUnit().getY();
    double dx = _argX - curX;
    double dy = _argY - curY;
    double dist = Math.sqrt(dx*dx + dy*dy);
    double coeff = dist / stepLength;
    addX = dx / coeff;
    addY = dy / coeff;
}

public ActionReturn checkPoint() {

    // Current coordinates are to be found in the Unitclass
    // getArgumentX(),getArgumentY() are the destination coordinates
    // Property Fuel holds the amount of fuel left

    // Check if there is enough fuel to continue the movement.
    double fuel = getUnit().getDoubleProperty("Fuel");
    if (fuel <= 0) {
        // Block the Move action since we're out of fuel.
        getUnit().addBlockedAction("Move");
        // Return with a unit message explaining what
        // has happened.
        return new ActionReturnError("Unit out of fuel - cannot move.");
    }

    // Set new fuel value
    getUnit().setProperty("Fuel", fuel - 1);

    double curX = getUnit().getX();
    double curY = getUnit().getY();
    double destX = getArgumentX();
```

```
        double destY = getArgumentY();

        // Check if we're close enough to the endpoint!
        if (Math.abs(curX - destX) < stepLength
            && Math.abs(curY - destY) < stepLength) {
            // We're done, set coordinates to destination
            // coordinates and return that we're completed.
            getUnit().setPosition(destX, destY);
            return new ActionReturnCompleted();
        }

        // Step forward
        curX = curX + addX;
        curY = curY + addY;

        if (getAutomatonState((int) curX, (int) curY) == 2 &&
            !(getUnit() instanceof Helicopter)) {
            // The automaton we're moving to is on fire,
            // decrease the health for the unit.
            // If this means that the unit should be
            // destroyed, then do so.
            double health = getUnit().getDoubleProperty("Health");
            health -= 5;
            if (health > 0) {
                getUnit().setProperty("Health", health-5);
            } else {
                getUnit().markUnitAsDestroyed();
                return new ActionReturnError();
            }
        }

        // Set new coordinates
        getUnit().setPosition(curX, curY);


        // We're not done, requeue the action.
        // The time to next call to the checkPoint
        // is set to 0.3 seconds.
        return new ActionReturnRequeue(0.3);
    }
}
```

References

[1] Joel Brynielsson, Henrik Bäärnhielm, Andreas Enblom, Jing Fu Zi, Niklas Hallenfur, Karl Hasselström, Henrik Hägerström, Oskar Linde, Klas Wallenius, and Jon Åslund. *GECCO User's Manual*. Department of Numerical Analysis and Computer Science, Royal Institute of Technology, Stockholm, Sweden, May 2001.