National Defence

National Defence Headquarters
Ottawa, Ontario
K1A 0K2

Défense nationale

Quartier général de la Défense nationale
Ottawa (Ontario)
K1A 0K2

Operational Research Integrated Graphical Analysis and Modelling Environment (ORIGAME) tutorial

Stephen Okazawa
DRWA

Director General Military Personnel Research and Analysis

Defence Research and Development Canada

Reference Document

DRDC-RDDC-2018-D173
February 2018

Abstract

This Reference Document is a tutorial that introduces new users to the ORIGAME simulation environment. It provides step-by-step instructions that will guide the reader through the creation of a simple model. Included in the instructions are detailed explanations of the concepts and features being used to accomplish specific tasks. After completing this tutorial, the reader should be able understand most of the inner workings of existing models and will be able to build new models for real applications. The ORIGAME User Manual is the complete reference for all of ORIGAME's features and is the next document that readers should consult to learn more about the software.

Résumé

Ce document de référence est un tutoriel présentant l'environnement de simulation ORIGAME aux nouveaux utilisateurs. On y explique les étapes à suivre pour créer un simple modèle. Dans les instructions, il y a des explications détaillées sur les concepts et les fonctions utilisés pour accomplir des tâches précises. À la fin du tutoriel, le lecteur devrait comprendre les rouages des modèles existants et être en mesure de créer de nouveaux modèles pour des applications réelles. Le manuel d'utilisateur du logiciel ORIGAME est un document de référence complet pour toutes les fonctions de l'outil. Les lecteurs devraient le consulter pour en apprendre davantage.

Table of contents

List of figures

1 Introduction

This Reference Document is a tutorial that introduces the reader to the main features of ORIGAME, a Python-based discrete event simulation environment developed jointly by DGMPRA and DRDC CORA [1]. The tutorial is intended to be the starting point for users encountering ORIGAME for the first time. It will guide the reader through the creation of a simple model, and insodoing introduce the reader to the software interface, the modelling concepts and software features. No prior knowledge of ORIGAME is required. However, it is assumed that the reader has ORIGAME installed on their computer.

The ORIGAME User Manual (included in the ORIGAME distribution and accessible from the Help menu in the ORIGAME application) is the complete reference document for the software. It describes the system requirements, installation procedure, and the use of all of ORIGAME's features. After completion of this tutorial, the User Manual is expected to become the user's primary reference for learning about ORIGAME in more detail.

The tutorial will begin by introducing the ORIGAME interface. It will then provide step-by-step instructions that demonstrate the basic mechanisms and concepts for creating data and code elements in ORIGAME. The tutorial then guides the reader through the creation of a simple population model, including running a simulation and plotting data collected during the run. Finally, the tutorial introduces several additional features that are useful when building models in practice.

2 Tutorial

Each of the following sections contains steps for the reader to follow including an explanation of the concepts and features being used. The example scenario that will be developed over the course of this tutorial is a very simple population model, and building this model from scratch will introduce the reader to enough of the basic concepts of ORIGAME that the reader will be able to create and understand more complex and realistic models.

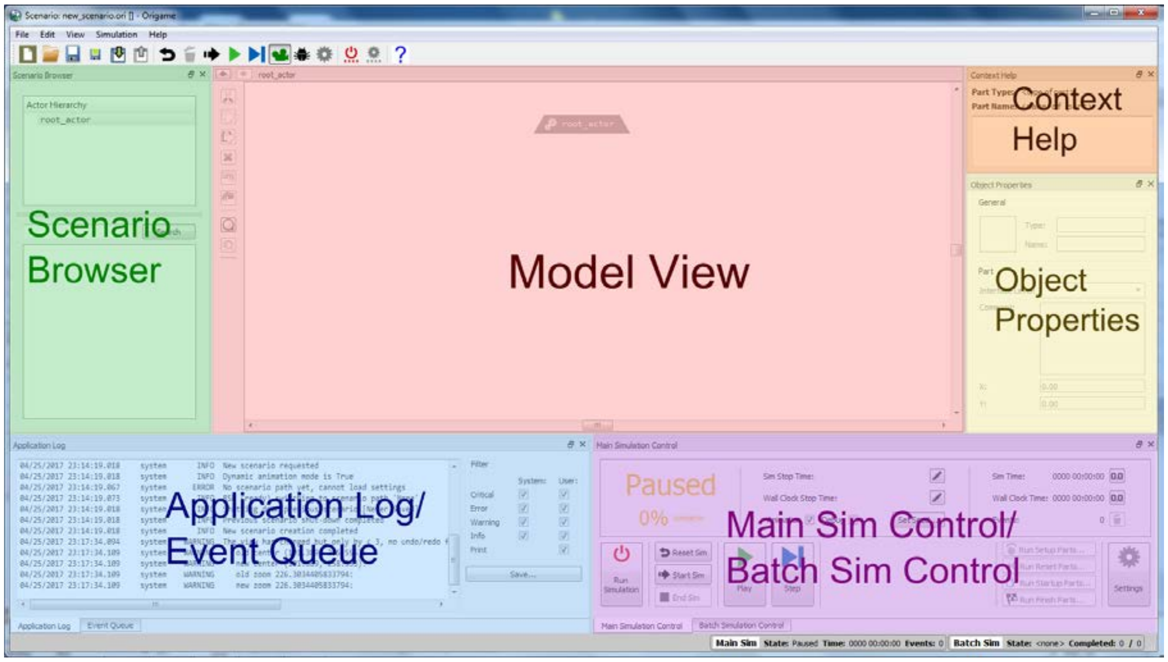
2.1 Introducing the ORIGAME interface

This section will familiarize the user with the general organization of the ORIGAME user interface.

Step 1. Open ORIGAME by double-clicking the ORIGAME desktop icon shown below.

Figure 1: *ORIGAME desktop icon.*

The ORIGAME application window will be displayed and an empty scenario will be opened. The layout of the application window upon first opening ORIGAME is shown in the figure below.



```
04/25/2017 23:14:19.018   system      INFO  New scenario requested
04/25/2017 23:14:19.018   system      INFO  Dynamic animation mode is True
04/25/2017 23:14:19.067   system     ERROR  No scenario path yet, cannot load settings
04/25/2017 23:14:19.073   system  [illegible]
04/25/2017 23:14:19.018   system  [illegible]
04/25/2017 23:14:19.018   system  [illegible] Previous scenario shut down completed
04/25/2017 23:14:19.018   system      INFO  New scenario creation completed
04/25/2017 23:17:34.094   system   WARNING  The view has changed but only by < 3, no undo/redo
04/25/2017 23:17:34.109   system  [illegible]
04/25/2017 23:17:34.109   system  [illegible]
04/25/2017 23:17:34.109   system   WARNING    old zoom 226.3034405833794:
04/25/2017 23:17:34.109   system   WARNING    new zoom 226.3034405833794:
```

Main Sim State: Paused Time: 0000 00:00:00 Events: 0 Batch Sim State: <none> Completed: 0 / 0

Figure 2: *ORIGAME application layout.*

The central area of the application window shows the *Model View*. This is where the user will build models in ORIGAME. At the top of the application are the main menu and shortcut buttons. To the left, right and below the *Model View* area are several dockable panels that contain information and controls used to build models and run simulations. Each dockable panel can be hidden by clicking the *x* in the upper right corner of the panel to create more space for the model view. Once hidden, a dockable panel can be restored by selecting it from the *view* menu. The location of each dockable panel can also be changed by dragging its title bar. The default layout of the interface can be restored by selecting *View > Restore Default View* from the main menu.

The *Scenario Browser* panel (left side of screen) displays a hierarchical view of the model and allows quick navigation and searching. The *Application Log* panel (lower left) displays log messages generated by the application and by the user's code (e.g., the output of print statements will appear here). The *Event Queue* panel (lower left, tabbed with the *Application Log*) displays all events currently on the event queue. The *Main Simulation Control* panel (lower right) displays information and controls used to run simulations. The *Batch Simulation Control* panel (lower right, tabbed with the *Main Simulation Control Panel*) displays information and controls used to run multi-replication simulations. The *Object Properties* panel (right) displays the properties of any object that is selected in the model view area. Finally, the *Context Help* panel (upper right) displays brief help information about any object that the mouse cursor hovers over.

2.2 ORIGAME basics

In ORIGAME, models are composed of parts that are linked together. There are several part types, but they can be broadly categorized into data parts which store information, code parts that execute instructions, and special parts that provide specific useful features including timing, plotting and encapsulation. This section will introduce the basics of adding parts, linking them together, and getting them to interact.

Step 1. Right-click in a blank area in the *Model View* and choose N*ew Part* > V*ariable*.

A new variable part will be added to the *Model View* as shown below. Variable parts can contain any Python object to store information in your model. By default, the variable's value is set to *None*.

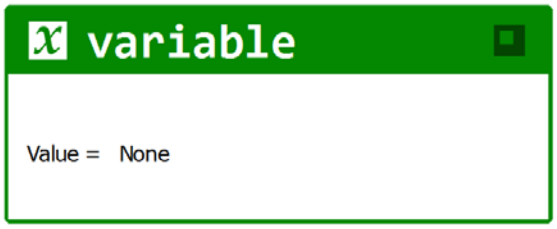


Figure 3: *Appearance of a variable part in the Model View.*

Parts can be moved by dragging them with the mouse in the *Model View*. Parts can also be resized by first clicking on them which displays a green highlight around the part, and then dragging the bottom or right sides of the highlight, or by dragging the bottom-right corner of the highlight. The highlight is darkened in areas that allow dragging to resize.

Parts can be deleted by selecting a part and pressing the *Del* key, or by right-clicking on the part and selecting *delete*.

Multiple-selection of parts is possible by holding shift and dragging a box around a group of parts or by control-clicking on specific parts. When more than one part is selected, the selected parts can be moved together by dragging any one of the selected parts, and the selected parts can be deleted by pressing the delete key.

Right-clicking on a part (or a selection of parts) also provides options to *Cut* or *Copy* the part. Right-clicking on the *Model View* canvas and selected *Paste*, will then create a duplicate of the original part (or parts) at the new location on the cavas.

The Model View can be panned left, right, up and down by dragging an empty point on the *Model View* with the mouse. The *Model View* can also be panned vertically using the mouse wheel, and horizontally by holding *Shift* and using the mouse wheel. The *Model View* can also be zoomed in and out by holding *Ctrl* and using the mouse wheel. Finally, the *Model View* can be automatically restored to a position and zoom level that contains all the parts by selecting *View > Model View > Zoom to Fit Contents* from the main menu.

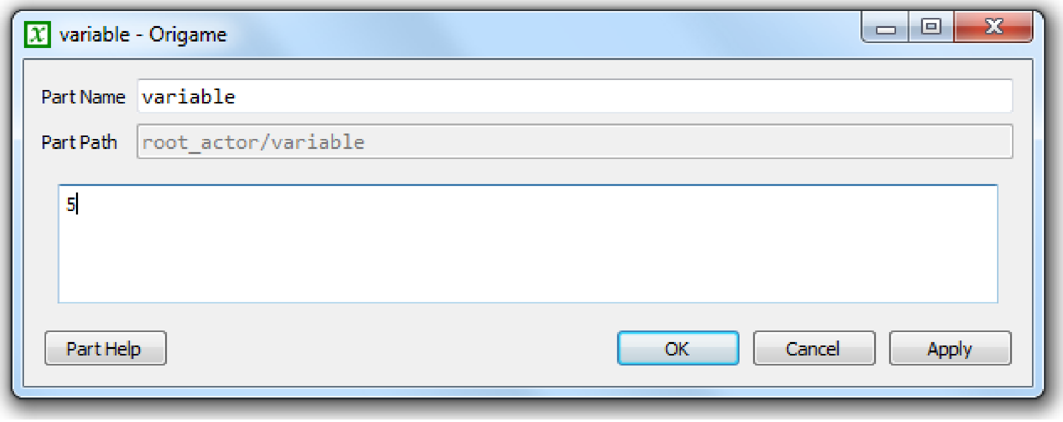
Step 2. Double-click on the variable part to open its editor.

A dialog box opens that allows the part to be edited. Each part type has a different editor, but all editors have a similar layout. The first field at the top of the editor allows the part name to be changed. The second field shows the hierarchical location of the part in the model. The next area of the editor below the path will be different depending on the type of part. In the case of variable parts, it is a text field in which any valid Python expression can be entered and the result of the expression will be stored in the variable.

At the bottom of all part editors are the standard *OK*, *Cancel*, and *Apply* buttons which follow standard windows behavior. At the bottom left of the part editors is a *Part Help* button which opens the *User Manual* to the page describing the part being edited.

Note most actions that can be performed on a part are accessible by right-clicking on the part and selecting the action from the context menu. Common actions may have a shortcut. For example, double-clicking on a part to open its editor is actually a shortcut for right-clicking on the part and selecting *Edit…*.

Step 3. Type *5* in the text field in the variable part editor (as shown below) and click OK.


variable - Origame

Part Name: variable

Part Path: root_actor/variable

5

Figure 4: *The variable part editor.*

The variable part has now been assigned the value *5*.

Step 4. Right-click in a blank area in the *Model View* and choose New *Part > Function*.

A new function part will be added to the *Model View*. Function parts are the most fundamental type of code part and will contain the underlying logic of the model. A typical model will be made up largely of various types of data parts and many function parts that manipulate the data and pass information around the model.

In order for one part to interact with another, there must be a link between the two. In this case, in order for the function part to interact with the variable part, there must be a link from the function part to the variable part. Note, links are unidirectional. If two parts each need access to the other, then two links are needed, one going in each direction.

Step 5. Hover the mouse cursor over the function part. An *arrow+* icon appears at the upper right corner of the function part (shown in the figure below). This is the *create link* icon. Click this icon to begin creating a link and click again on the variable part to complete the link. Alternatively, link creation can be initiated by right-clicking on the source part, choosing *Create Link*, and then clicking on the target part.

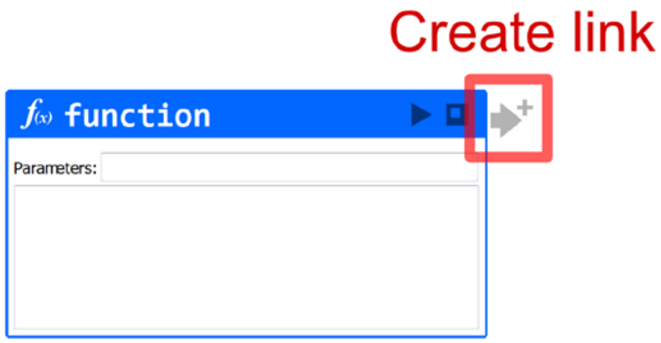


Figure 5: *The "create link" icon that appears when hovering over a part.*

A link will be created from the function part to the variable part. Note that the link has its own name. By default the link gets its name from the target part, but the link can have a name that is different from the target part. The link's name can be changed by double-clicking it.

The link's name, rather than the part name, is the name that will be used by the function part to refer to the variable part. Note this means that different parts in the same model can share the same name without causing name clashes because each part is explicitly linked to the other parts that it needs access to, and it accesses those parts using the link name rather than the part name. In other words, part names are cosmetic and have no effect on the model. On the other hand, link names do affect the model since they are referred to in the code, but the only constraint on link names is that all the links emerging from the same part must be uniquely named. Therefore, if one part is linked to two other parts having the same part name, the link names will have to be unique. ORIGAME will automatically append a number to a new link's name if needed to preserve uniqueness.

This named-link scheme is what allows ORIGAME model logic to be modular. In other words, any model logic can be copied and pasted or exported and imported into any other model and interconnected with that model. Because the links are preserved when moving or copying model logic, the transplanted logic will work in its new setting even if it contains part names that are already used in the receiving model. And the transplanted logic can be interconnected with the receiving logic simply by creating new named links.

Step 6. Double-click the function part to open its editor.

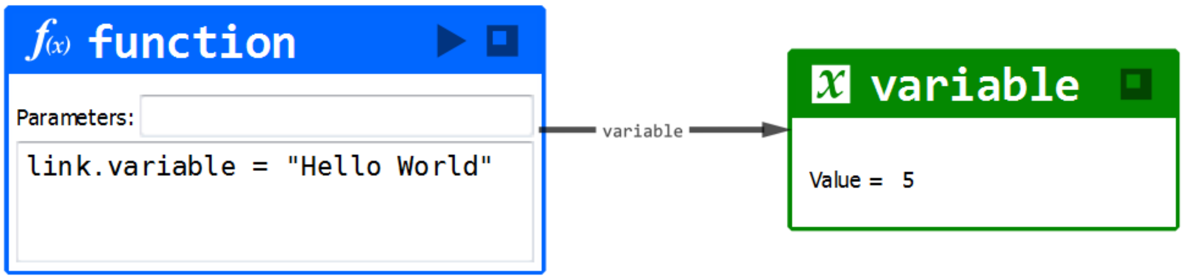
The function part editor will open. Below the standard *name* and *path* fields at the top of the editor is a *parameters* field. If the function will accept parameters, the parameter list can be entered here separated by commas, for example *arg1, arg2, arg3*. Below the parameters field is the code editing space in which the body of the function will be written.

Step 7. Type the code below in the code editing space of the function editor and click OK.

```
link.variable = “Hello World”
```

The function part in the *Model View* will be updated to display the new function body. All links that emerge from a function part can be accessed in the function body via the special object called *link*. Typing *link.* will bring up an auto-complete menu of all parts that the function part is linked to. In this case, the function's code assigns the string *Hello World* to the variable. Note that we have now defined the function, but we have not run it. Clicking OK in the editor saves the changes to the function body, but takes no further action. If the code is incomplete or contains errors, ORIGAME will not complain until the function actually runs.

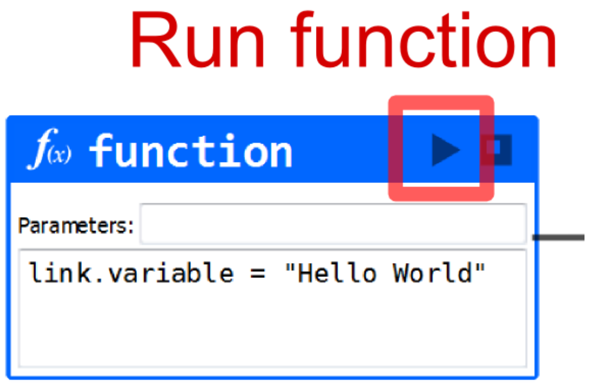
The model should now look like the following.



```
link.variable = "Hello World"
```

Figure 6: *A simple “Hello World” model.*

Step 8. Run the function by clicking the triangle shortcut button located in the upper-right corner of the function part (see figure below). Alternatively, right-clicking on function parts and selecting *Run* will also cause the function to run.



```
link.variable = "Hello World"
```

Figure 7: *The “Run” shortcut button on function parts.*

The function runs and the variable part is assigned its new value.

Step 9. Double-click the function part to open its editor again. Change the function’s name to “save”, add a parameter called *value* to the parameters field, replace the previous function body with the code below, and click OK.

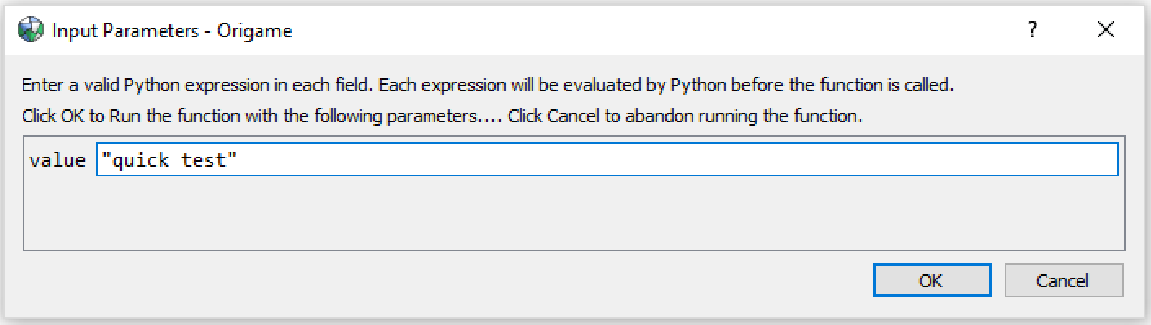
```
link.variable = value
```

The function has now been modified to accept a parameter, and it assigns whatever argument it receives for that parameter to the variable.

Step 10. Run the save function again by clicking the triangle shortcut button.

Because the save function has a parameter, attempting to run it brings up a dialog prompting the user to provide the parameter arguments.

Step 11. Enter *“quick test”* in the *value* field as shown below and click OK. Note, include the quotes when entering the parameter argument to indicate that the value is a character string.


Input Parameters - Origame

Enter a valid Python expression in each field. Each expression will be evaluated by Python before the function is called.
Click OK to Run the function with the following parameters.... Click Cancel to abandon running the function.

value "quick test"

Figure 8: *The "Input Parameters" dialog that appears when running a function with parameters.*

The function runs with the supplied argument and the variable value is updated to *"quick test"*. Note in Python, single-quotes and double-quotes can be used interchangeably as long as a single-quote is matched to a single-quote and a double-quote is matched to a double-quote to open and close strings.

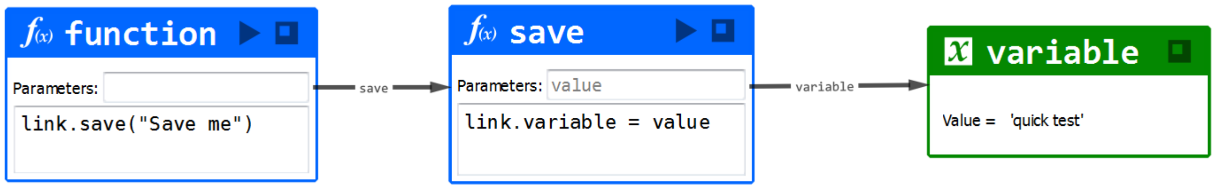
Step 12. Create a new function part and link it to the save function using the linking procedure described in Step 5.

Step 13. Double-click the new function part, enter the following code in the code editing space, and click OK.

```
link.save("Save me")
```

Using *link* to access linked parts returns a reference to the linked object. This means you can perform any action that the linked object allows. In this case, because *link.save* is a reference to a function, we can call the function and pass it the required function arguments. In general, all parts expose an application programmer interface (API) such that a function linked to that part has complete programmatic control over the part. Thus any action that can be performed via the ORIGAME user interface can also be automated in function code. In this example, we simply call the *save* function and pass the string *"Save me"* as the argument.

The model should now look like the following.



```
link.save("Save me")
```

```
link.variable = value
```

Figure 9: *A simple model that saves a value to a variable part.*

Step 14. Run the new function by clicking the triangle shortcut button.

The new function will run, which calls the save function with the argument "Save me", which assigns the value "Save me" to the variable part.

Step 15. Save the scenario by selecting *File* > *Save* from the application menu. In the dialog that opens, select a folder location, enter a file name, and click *Save*.

The scenario will be saved to a file at the specified location. ORIGAME scenario files are given the extension *.ori* and are saved in *json* format. The file will save the state of all parts in the model including the current state of any data parts.

This section demonstrated the most basic aspects of how ORIGAME works. Although we have not run a simulation, we have created data and added code that manipulates the data. These elements can be used to build almost any Python program using the features of ORIGAME. It is suggested that the user take some time to experiment with these features by creating more variables parts, linking to them from function parts and manipulating the contents of the variables. In order to write more interesting functions, the user will have to become familiar with Python. There are many free online resources and courses that teach Python, one of which is www.learnpython.org. In the next section, we will introduce the simulation features of ORIGAME.

2.3 ORIGAME simulations

This section will introduce the basic simulation features of ORIGAME by creating and running a very simple population model.

ORIGAME is a discrete event simulation (DES) environment, but there are some differences between it and other DES simulation products. In conventional DES, *entities* represents objects moving through a process, such as letters in a mail system or patients in a hospital. When an entity reaches a given step in the process, an *event* is triggered that can alter the state of the entity or other simulation variables, delay the entity, and direct the entity to the next step in the process depending on what happened in the current step.

In ORIGAME, Python functions are used to define the steps in the process (i.e., the events), and function parameters are used to pass information from one step to the next (i.e., the entities). An event occurs when a given function with a given set of input parameters is triggered. The function can perform actions as described in the previous section, and it can also schedule future events by adding another function and associated parameter values to the simulation event queue to be executed at a specified future time.

Compared to traditional DES software products, ORIGAME generalizes the concepts of events and entities in that any Python code (rather than a limited set of actions) can be used to define events, and any Python object or collection of objects (rather than a single data structure) can be used to pass information from one event to the next.

This section will introduce these features and will proceed more quickly than the previous, as it is assumed that the reader is now familiar with the operations of adding, linking, and editing parts.

Step 1. Add a data part and time part to the *Model View*.

The time part will be used to track time as the simulation runs. The data part will be used to store the data used in the model. Data parts are a slightly more advanced data structure than the variable parts used in the previous section. Data parts are a dictionary of variables, each entry consisting of a key-value pair.

Step 2. Add a function part, link it to the time part and the data part, open the function editor, name the function “reset”, enter the following code in the code editing space, and click OK.

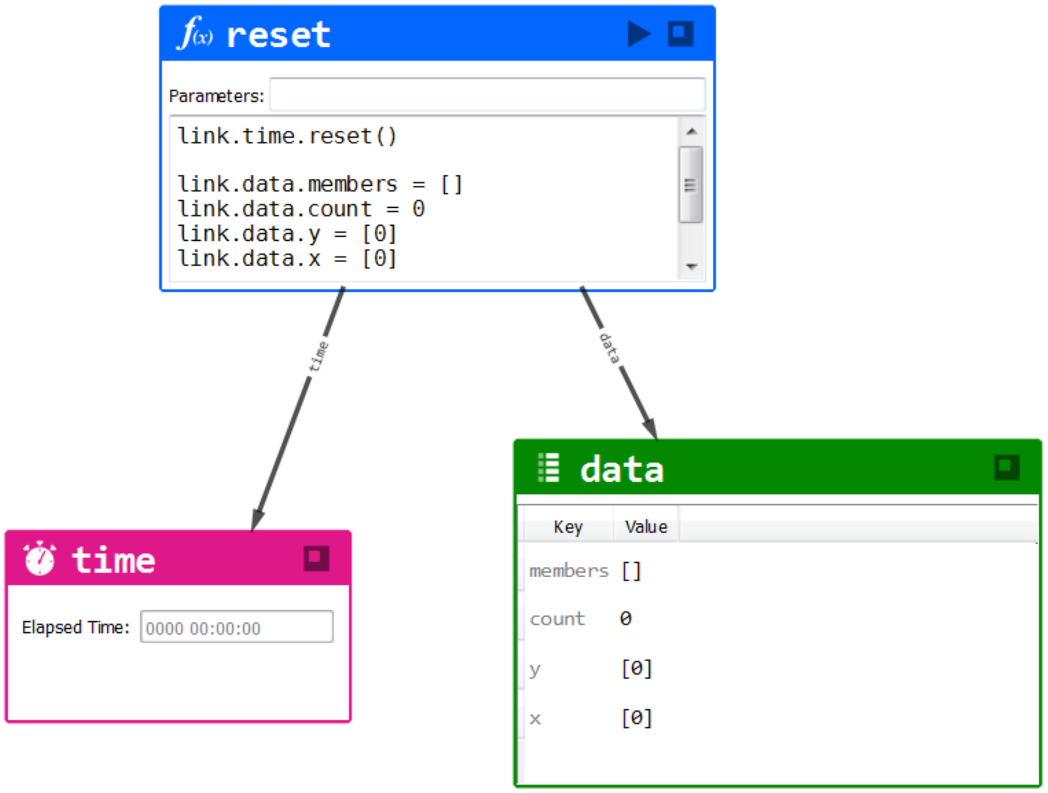
```
link.time.reset()

link.data.members = []

link.data.count = 0

link.data.y = [0]

link.data.x = [0]
```

This code will reset the time part and initialize four entries in the data part: *members*, *count*, *y*, and *x*. The square brackets are used to create Python lists. *Members* will be a list of current member IDs in the population. *Count* will track the current number of members in the member list. *Y* is a list that records how the population count changed over time. And *X* is a list of time values corresponding to the count values in the *Y* list. X and Y will later be used to plot the population size over time.

Note that in ORIGAME, time is not defined by a global simulation clock. Users must create time parts in order to keep track of time in their model. Global simulation clocks are convenient when building a single model, but ORIGAME was designed to facilitate model integration. When integrating models that have been built by different people, it is rare that they will agree on how time is used. For example, each model might assume a different simulation start date. If time is global, then there is no easy way to detect and correct the fact that the two models may be operating on different assumptions about time, and the result may be a broken model. In ORIGAME, each model includes its own explicitly defined time parts. This ensures that neither model will be broken when the two are integrated, and the user can easily see if the times are not synchronized. Synchronizing two integrated models is then usually a straightforward matter of adjusting the time parts (similar to synchronizing watches in the real world) or adjusting the logic that sets up the timing of the initial events. Because the model logic that interacts with the time parts is shown by the links, it is also easy to track down which parts of the model are dependent on timing.

Step 3. Run the reset function to initialize the time and data parts.

The model should now look like the following.



```
link.time.reset()

link.data.members = []
link.data.count = 0
link.data.y = [0]
link.data.x = [0]
```

Key	Value
members	[]
count	0
y	[0]
x	[0]

Figure 10: *A model with a Reset function.*

We will now introduce the mechanism for creating simulation events. As mentioned above, events are delayed function calls that are placed on a queue and executed in order. When adding a new event to the queue, the user specifies: 1) the function to be called, 2) the parameter arguments to be passed to the function, 3) the time at which the function call shall be executed, and 4) the priority of the event relative to other coincident events.

Adding an event to the queue is accomplished by calling the *signal* function. The name *signal* was chosen because the act of scheduling an event can be thought of as sending a message to a specified recipient, to be received at a specified time. In ORIGAME, the signal recipient is the target function to be executed, and the message is the parameter arguments that are passed to the target function. The *signal* function has the following form:

```
signal(target function, arguments, time, priority)
```

In the signal function, a target function must be specified, but the arguments, time and priority are optional parameters. The arguments parameter need only be used if the target function expects arguments to be passed. The time parameter need only be used if the event is to be delayed to a future time; if omitted, the time will be the current simulation time (zero delay). The priority parameter need only be used if there is a possibility of coincident (same time) events and it is important to control the order in which the events execute. The priority is a floating point value and higher values execute first.

The events on the queue are ordered first according to time, then according to priority, then according to first-in-first-out queuing rules. Therefore, the next event to execute is the one that is the soonest, then highest priority, then first to be added to the queue.

Note that with first-in-first-out queue rules, it is not possible to guarantee that a particular function will be the next to execute from the queue, even if the delay is zero and the priority is a high value. It is always possible that a higher priority, coincident event was previously placed on the queue. If it is necessary to guarantee that a particular function is the next to execute, the priority can be set to *ASAP*. This places the function on a special last-in-first-out event queue with zero delay that is always executed before any pending events in the regular event queue.

Step 4. Add two functions, one called *enter* and one called *exit*, add a parameter called *ID* to both functions, link both functions to the time part and the data part; link the *enter* function to the *exit* function.

Step 5. Add the following code to the *enter* function.

```
link.data.members.append(ID)

link.data.count += 1

link.data.y.append(link.data.count)

link.data.x.append(link.time.elapsed_time.days)

signal(link.exit, ID, delay(days = random.triangular(20,80,40)))
```

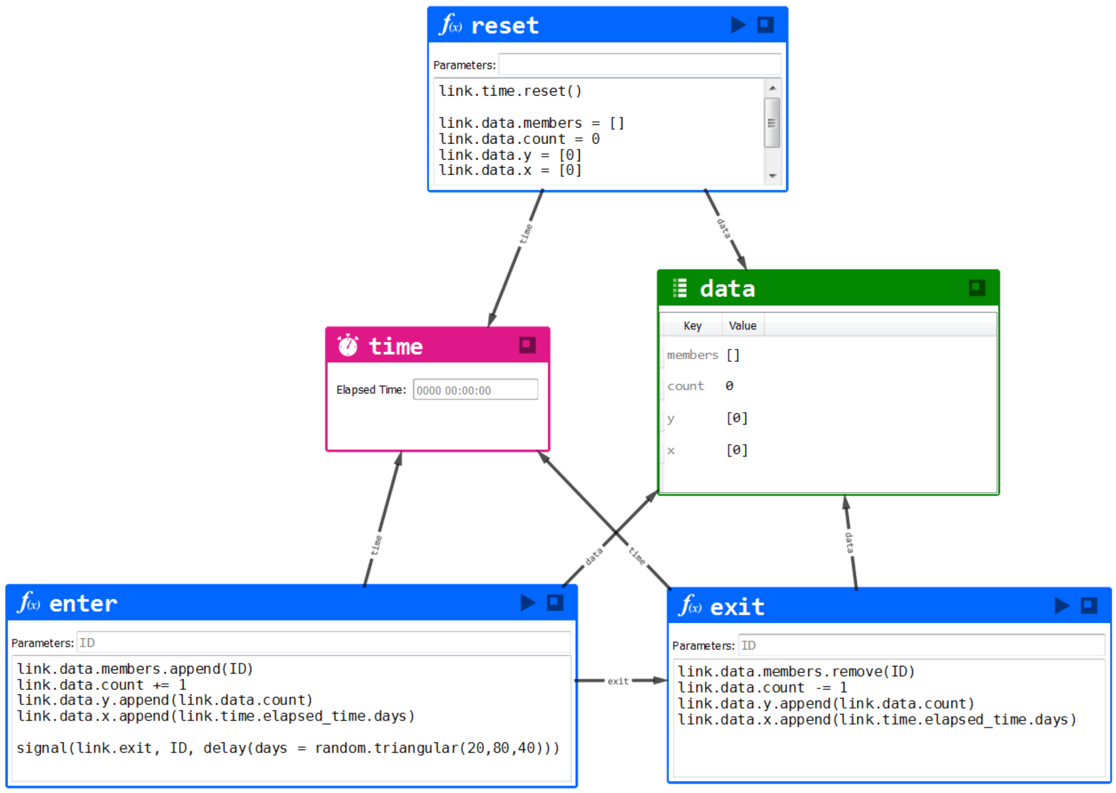
The first line adds the incoming member ID to the *members* list. The second line increments the *count*. The third line appends the current count to the *y* value list. The fourth line appends the current time in days to the *x* value list. Finally, the last line creates a future event by calling the signal function. The signal targets the exit function, passes the member *ID* as an argument, and sets the time of the event by creating a delay following a random triangular distribution with minimum 20 days, maximum 80 days, and mode of 40 days.

Step 6. Add the following code to the *exit* function.

```
link.data.members.remove(ID)

link.data.count -= 1

link.data.y.append(link.data.count)

link.data.x.append(link.time.elapsed_time.days)
```

This code removes the member ID from the *members* list, decrements the member *count*, and adds the current count and current time to the *y* and *x* value lists.

When the enter function is executed with a given member ID, the member gets added to the population and a delayed event is created that later removes this member from the population after an amount of time that follows the triangular distribution. The model should now look like the following.



```
reset
Parameters:
link.time.reset()

link.data.members = []
link.data.count = 0
link.data.y = [0]
link.data.x = [0]
```

Key	Value
members	[]
count	0
y	[0]
x	[0]

```
enter
Parameters: ID
link.data.members.append(ID)
link.data.count += 1
link.data.y.append(link.data.count)
link.data.x.append(link.time.elapsed_time.days)

signal(link.exit, ID, delay(days = random.triangular(20,80,40)))
```

```
exit
Parameters: ID
link.data.members.remove(ID)
link.data.count -= 1
link.data.y.append(link.data.count)
link.data.x.append(link.time.elapsed_time.days)
```

Figure 11: *A model that tracks entries and exits from a population.*

Step 7. Test the model by running the *enter* function. Enter a value for the *ID* when prompted.

The data part will be updated to reflect the addition of the member. Note the green arrow that appears to the left of the *exit* function. This indicates that there is a pending event on the *exit* function. It is green because it will be the next event to execute. Look at the *Event Queue* panel and note that the event appears in the list.

Step 8. Click the *Step Simulation* button in the *Main Simulation Control* panel as shown below.

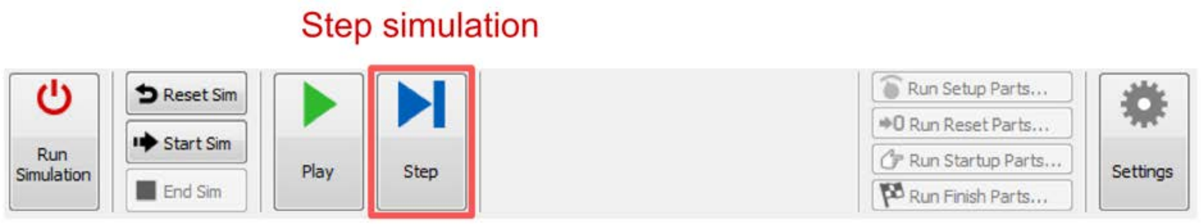


Figure 12: *The "Step Simulation" button on the "Main Simulation Control" panel.*

The *Step Simulation* button pops the next event off the event queue and executes it. Note the data part now shows that the member has come and gone. And note that the time part has advanced to the time at which the member exited.

Step 9. Create a new function part, link it to the *enter* function, name it *create*, add a parameter called ID, and enter the following code in the function editor.

```
signal(link.enter, ID)

if ID <= 100:

    signal(self, ID+1, delay(days = random.randint(0,10)))
```

This function will gradually create 100 people to become members of the population. The first line creates a signal sending the new member ID (which is itself a parameter of the create function) to the *enter* function. The *if* statement ensures that we only create up to 100 IDs. If we are still at or below ID 100, the function then signals itself, passing the next ID as the argument, and using a delay following a random uniform distribution of 0 – 10 days.

Step 10. Create a new function part, link it to the *create* function, name it *start*, and enter the following code in the function editor.

```
signal(link.create, 1)
```

This function only needs to run once to start the simulation. It executes the first signal on the *create* function, after which the *create* function repeatedly signals itself until 100 members have been created.

Step 11. Right-click on the *start* function and select *Roles…* > *Startup*.

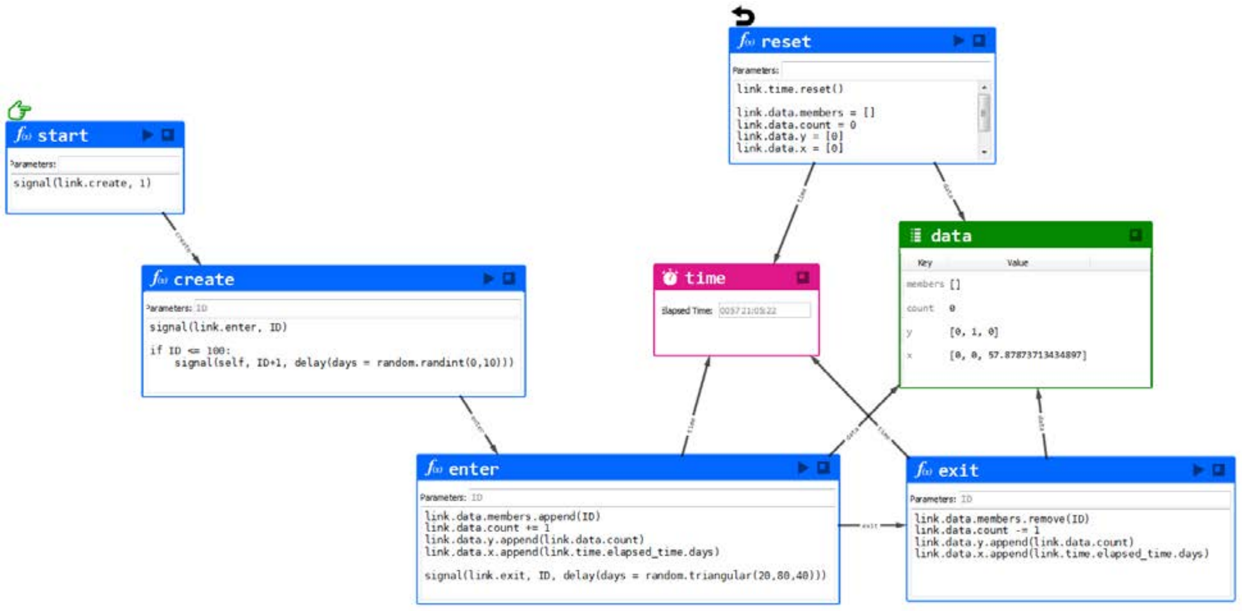
This tells ORIGAME that the simulation should begin by executing the *start* function. Note the green pointing hand icon that appears above the *start* function indicating its role as a *Startup* function.

Also note there can be multiple *Startup* functions within a model. All of the *Startup* functions will be automatically executed by ORIGAME when the simulation is started.

Step 12. Right-click on the *reset* function and select *Roles…* > *Reset*.

Similar to the *Startup* functions, all *Reset* function will be run automatically by ORIGAME when the user resets the simulation.

The model should now look like the following.



```
signal(link.create, 1)
```

```
signal(link.enter, ID)

if ID <= 100:
    signal(self, ID+1, delay(days = random.randint(0,10)))
```

```
link.time.reset()

link.data.members = []
link.data.count = 0
link.data.y = [0]
link.data.x = [0]
```

```
members []
count   0
y       [0, 1, 0]
x       [0, 0, 57.87873713434897]
```

```
link.data.members.append(ID)
link.data.count += 1
link.data.y.append(link.data.count)
link.data.x.append(link.time.elapsed_time.days)

signal(link.exit, ID, delay(days = random.triangular(20,80,40)))
```

```
link.data.members.remove(ID)
link.data.count -= 1
link.data.y.append(link.data.count)
link.data.x.append(link.time.elapsed_time.days)
```

Figure 13: *A complete model with Startup and Reset functions.*

Step 13. Click the *Reset Simulation* button in the *Main Simulation Control* panel as shown below.

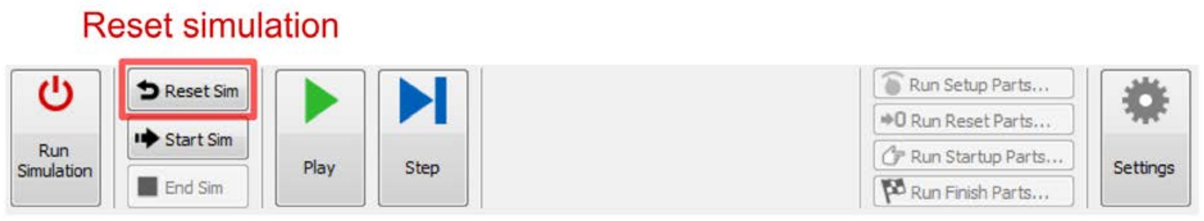


Figure 14: *The "Reset Simulation" button on the "Main Simulation Control" panel.*

This is will run all the *Reset* functions in the model. Note that the data part and time part have been restored to their initial states.

Step 14. Click the *Main Simulation Settings* button in the *Main Simulation Control* panel as shown below.

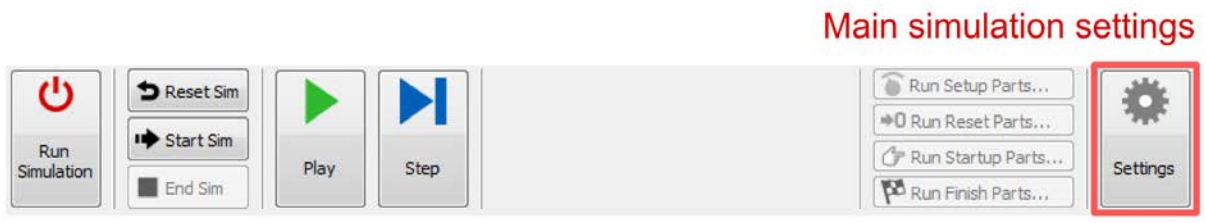


Figure 15: *The "Settings" button on the "Main Simulation Control" panel.*

A dialog opens displaying many settings that can be used to control the simulation. On the right side of this dialog is a list of actions that are performed by the application when the user clicks on the *Reset Simulation*, *Start Simulation* and *End Simulation* buttons. Note that when the *Reset Simulation* button is pressed, the application actually performs several actions in addition to running the model's *Reset* functions: zeroing the simulation time and wall clock time, clearing the event queue and applying the initial random seed. All these actions are enabled by default but can be disabled if desired. It is under the *End Simulation options* area of this dialog that the simulation's end conditions are specified. The simulation can be caused to stop at a specified simulation time or wall clock time or when there are no more events on the queue. For this example, we don't need to change any of the settings, but the reader should be aware of the options and information available in the *Main Simulation Settings* dialog. More detail on all these settings can be found in the User Manual.

Step 15. In the *Main Simulation Settings* dialog, click *Cancel* to discard changes and exit the dialog. The left-most button on the *Main Simulation Control* panel is the *Run Simulation* button which simply combines the actions of resetting and then starting the simulation in a single click. The simulation will then run until one of the end conditions is encountered. In most cases, the user would use this button to run a simulation. The individual Reset, Start and End buttons allow the user to perform these steps manually if desired, which is likely to be convenient while testing the model. In this tutorial we are performing these steps manually in order to introduce the concepts one at a time.

Step 16. Ensure that the *Animation* checkbox on the *Main Simulation Control* panel is checked, and click the *Start Simulation* button as shown below.

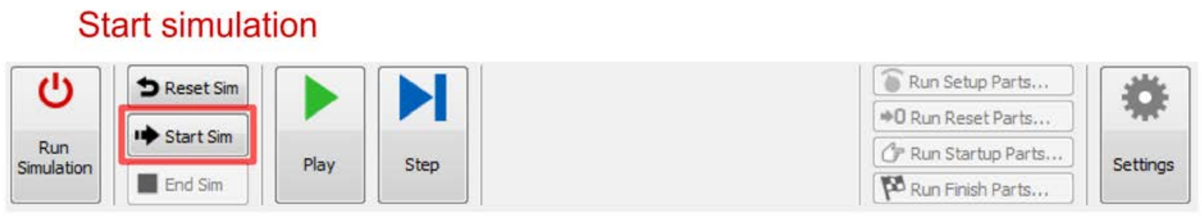


Figure 16: *The "Start Simulation" button on the "Main Simulation Control" pane.*

The *Start Simulation* button runs the model's *Startup* functions, and sets the state of the simulation to *running*. When in the *running* state, ORIGAME continuously processes events on the event queue. Because the *Animation* checkbox is checked, the model view updates as the simulation runs.

Step 17. While the simulation is still running, press the *Pause Simulation* button in the *Main Simulation Control* panel as shown below.

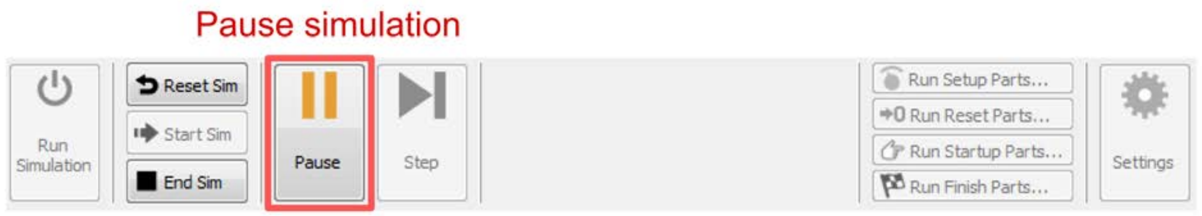


Figure 17: *The "Pause Simulation" button on the "Main Simulation Control" panel.*

This places the simulation in the *paused* state in which ORIGAME ceases processing events on the event queue. Note the arrows with numbers to the left of the functions parts which represent the current set of events that are pending in the simulation. The number indicates how many pending events are waiting at each function part. The arrow will be green if one of the function's events is the next to execute. If there are events on the queue, there will always be only a single green arrow somewhere in the model because there can only be a single next event. The arrow will be amber if one of the function's events is coincident with the next event, but will execute after the next event. The arrow will be black if all the function's events are scheduled later than the next event. The full list of pending events can be seen in the *Event Queue* panel with the next event appearing at the top of the list.

Step 18. Repeatedly press the *Step Simulation* button in the *Main Simulation Control* panel (see Step 8).

This will advance the simulation in single steps by only popping a single event off the queue and executing it. Note that it is possible to track the sequence of events by following the green event arrow.

Step 19. Press the *Play Simulation* button on the *Main Simulation Control* panel as shown below to return the simulation to the *running* state.

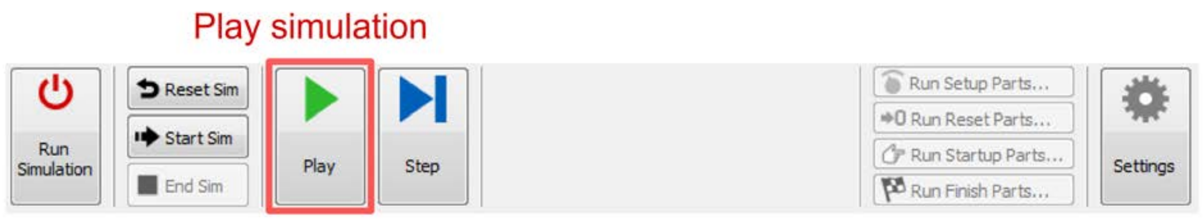


Figure 18: *The "Play Simulation" button on the "Main Simulation Control" panel.*

Note that *Play Simulation* is different from *Start Simulation*. *Play Simulation* only toggles the simulation state from *paused* to *running*. This resumes processing of events on the event queue, but if there are no events on the queue, pressing *Play Simulation* will have no effect. *Start Simulation* runs the model's *Startup* functions and then places the simulation in the *running* state as described in Step 16.

Step 20. Wait until the simulation ends. This will occur when there are no more events on the event queue which is one of the end conditions specified in the *Main Simulation Settings*.

The simulation will pause when the end condition is met. Note that the data part now indicates that there are no member IDs in the *members* list and the *count* is zero, but the *x* and *y* entries have been populated with values. The time part will also indicate that 500 to 600 days have passed since the simulation started.

Step 21. Add a plot part to the model and link it to the data part.

Plot parts are used to visualize data. Plot parts makes use of the Python Matplotlib library which is able to create an extremely diverse array of scientific visualizations. Similar to MatLab, Matplotlib plots are created by writing scripts. In this version of ORIGAME, it is necessary to consult the online Matplotlib documentation (see matplotlib.org) to learn how to write these scripts. In future versions of ORIGAME, more assistance will be provided in the plot part so that the user is not dependent on external documentation.

Step 22. Double-click on the plot part to open its editor.

The left side of the editor contains the scripting area and the right side of the editor contains a *Plot Preview*. As the user develops the script, clicking the *Refresh* button in the *Plot Preview* runs the script and displays the resulting plot. By default, the plot part includes a sample script and clicking *Refresh* will display the result. The script must implement two functions, *configure* and *plot*. The *configure* function is used to setup the axes, gridlines, labels, and other plot elements that do not depend on the actual data being plotted. The *plot* function draws the data on the axes.

Step 23. Enter the following script in the plot editor, click *Refresh* to preview the result, and click *OK*.

```
def configure():

    axes = setup_axes(1, 1)

    axes.set_ylabel('Population')

    axes.set_xlabel('Time [days]')

    axes.set_title('Population vs Time')

    axes.grid = True

def plot():

    axes.fill(link.data.x, link.data.y, 'r')
```

Step 24. Click the cycling arrows shortcut button in the upper right corner of the plot part (shown below) to refresh the plot display in the *Model View*.

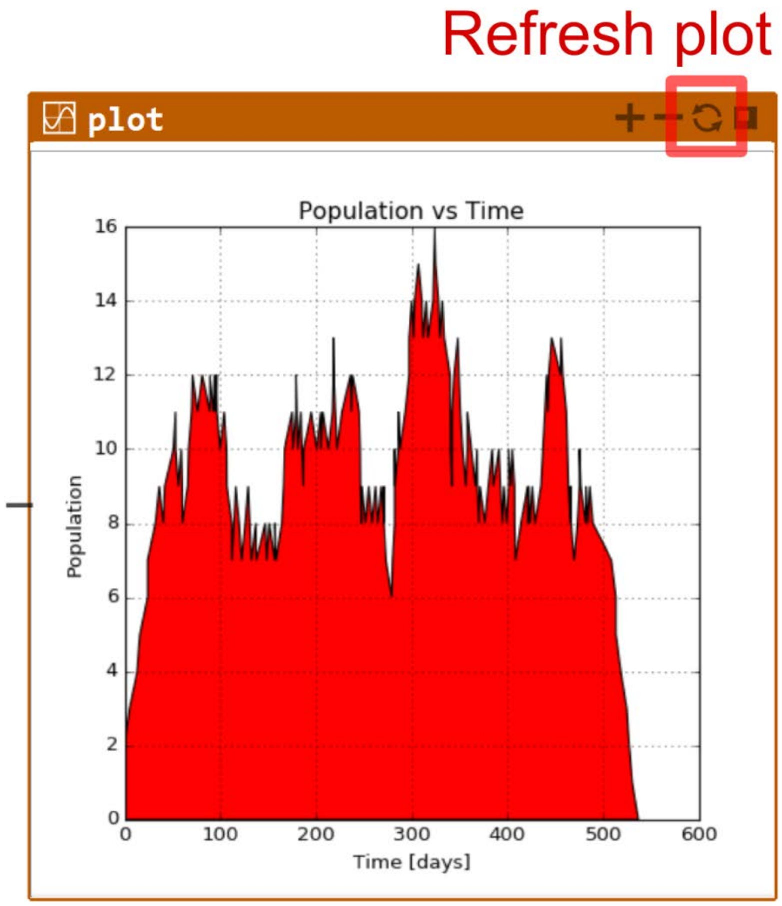


Figure 19: *The "Refresh" shortcut button on plot parts..*

The plot part will display the plot of the population over time.

Step 25. Save the scenario.

3 Additional features

The preceding sections introduced the reader to the fundamental features of ORIGAME needed to build and run simulations. This section introduces some additional features that are likely to be useful when building models in practice. Each feature is presented on its own rather than as part of a larger model, so it is not necessary proceed through the following sub-sections in order. Steps are provided to illustrate the operation of each feature.

3.1 Hubs

When the number of parts in a model starts to grow, it will eventually become impractical to create individual links between all parts that need to interact. For example, a model will likely have many parts containing data about the state of the model, and every function part may need access to many or all of these data parts. To address this situation, a hub can be used which allows a part to access many other parts via a single link to a hub that then connects to all the other needed parts. The following steps will illustrate the operation of the hub.

Step 1. Create a data part.

Step 2. Create several other parts (the part type is not important).

Step 3. Create a hub part, and link the hub to all the parts created in Steps 1 and 2. Any part that links to this hub then has access to all the parts that the hub is linked to.

Step 4. Create a function part and link it to the hub. Open the function part editor and type "*link.hub.data['apples'] = 10*" in the code editing area. Run the function and note that the function modified the contents of the data part via the hub. Note that upon typing "link.hub." in the function, the code completion hints display all parts that the hub is linked to.

3.2 Libraries

It may be useful to define several related functions and classes in a form that resembles a regular Python script. For example, helper functions that are used in various places in the model, or custom class definitions for objects that will be used in the model, may be more easily created and managed in a single Python script rather than within ORIGAME function parts. For this reason, ORIGAME includes a library part which provides a free-form text editor to write arbitrary Python scripts. Note that the library part is not meant to be a place to run Python programs (although this is possible); the purpose of the library part is to define a library of related functions and classes that can then be used within the ORIGAME model. The following steps will illustrate a simple usecase for the library part.

Step 1. Create a library part.

Step 2. Double-click the library part to open its editor and enter the following code:

```
def greeting(first, last):

    print(“Hello”, first, last)

def greet_john_doe():

    greeting(“John”, “Doe”)

def greet_group(group):

    for first, last in group:

        greeting(first, last)

def greet_team():

    team = [(“Homer”, “Simpson”),

        (“Marge”, “Simpson”),

        (“Bart”, “Simpson”),

        (“Lisa”, “Simpson”),

        (“Maggie”, “Simpson”)]

    greet_group(team)
```

Note that all these functions are related and build on each other so it is convenient to define them together in a single script in the library part.

Step 4. Create a function part, link it to the library part, open its editor and enter the following code:

```
link.library.greet_team()
```

Any part that links to the library part has access to all the objects defined in the library part.

3.3 Tables and SQL

Relational database tables are often a useful way to store simulation data. The input data for a simulation often comes from a database, so storing it in a table within the simulation is usually simpler than translating it into a conventional data structure such as a 2-dimentional list. Storing the data in a table also allows the data to be queried using SQL which may be more convenient and more powerful than the capabilities offered by other data structures. Finally, a table may scale better in terms of performance compared to other data structures when the amount of data is large.

ORIGAME integrates the SQLite relational database engine which is included with Python. The following steps illustrate the common task of import a table from an MS Access database and querying the table within ORIGAME.

Step 1. Create a table part

Step 2. Right-click on the table part and select *Import from Access…*.

Step 3. In the dialog, select an Access database file by typing the path or by clicking the "…" button and selecting the file in the file explorer. Once the file has been selected, click the *List Tables* button to retrieve the tables in the chosen Access database. Select the desired table from the list of tables, and then click the check boxes next to each the desired fields to be imported from the table. Optionally, a filter can be defined such that only records that meet the filter criteria are imported. The filter must be a valid SQL *where* clause. Finally, click OK. The Access table will be imported into the ORIGAME table part.

Step 4. Create an SQL part and link it to the table part. The SQL part allows arbitrary SQL queries to be written that can access, append to, and modify the data in table parts.

Step 5. Double-click the SQL part to open its editor. Enter the following SQL code in the editor:

```
select * from {{link.table}}
```

This query returns all records and all fields from the table created above. The double-curly-braces are used to combine ORIGAME Python code with the SQL code. In this case the Python code *link.table* is resolved to the correct table identity within the underlying SQLite relational database.

Step 6. In order to access the records returned by an SQL query, the SQL part can be called like a function. Create a function part, link it to the SQL part, and add the following code in the editor:

```
for record in link.sql():

    print(record)
```

This code calls the sql part to retrieve the resulting recordset from the query and iterates over all the records, printing each one. Note that the result of a select query, such as the one created in this example, will be a list of lists. Indexing into the query result returns the record, and indexing within the record returns the field. Thus, for example, to retrieve the 3rd field (index 2) of the 2nd record (index 1), the following code would be used within the function part:

```
result = link.sql()

print(result[1][2])
```

3.4 Sheets

The sheet part provides a data structure similar to an Excel spreadsheet. It contains a grid of cells, where each cell can be indexed using either row/column indices or Excel letter/number notation. An important use of the sheet part is to import from or export to MS Excel. The following steps will demonstrate importing data from an Excel file and accessing and manipulating the data in the sheet part.

Step 1. Create a sheet part.

Step 2. Right-click on the sheet part and select *Import from Excel*.

Step 3. In the dialog, enter the path to the desired Excel file or click the "…" button and select the Excel file in the file explorer. Once a file is selected, click the *List Sheets* button to retrieve the available sheets in the Excel file. Select a sheet from the dropdown list, and optionally enter a cell range to specify the sheet data to the imported. Finally, click OK. The Excel data will be imported into the sheet part.

Step 4. Create a function part, link it to the sheet part, and enter the following code in the function editor:

```
link.sheet[0,4] = 5

link.sheet['A4'] = 10
```

Step 5. Run the function and observe the effect on the sheet part. Note that when numerical indices are used, the format is (row index, column index) where the first row and first column have index 0. When letter/number notation is used, the behavior corresponds to Excel. Also note that if the indices correspond to a cell that is outside the sheet's row or column bounds, an exception will be raised. The number of rows and columns in the sheet can be adjusted in the sheet's editor.

Step 6. Create another function part, link it to the sheet part, and enter the following code in the function editor:

```
link.sheet['C3:D4'] = link.sheet['A1:B2']

link.sheet[0:2,2:4] = link.sheet[2:4,0:2]
```

Step 7. Run the function and observe the effect on the sheet part. This code demonstrates the use of cell ranges. The first line uses Excel cell range notation and moves a block of 2x2 cells from one part of the sheet to another. The second line uses Python slice notation to define a range and moves another 2x2 block of cells to another location on the sheet.

3.5 Actors

Actors are special parts that allow encapsulation of model logic. This means that detailed logic can be hidden within the actor, and the actor can be treated as a reusable component that can be copied and used throughout the model. Actors can contain other actors so that the model can be organized hierarchically with each level in the hierarchy being concerned with a different level of implementation granularity. The following steps will illustrate the use of Actors:

Step 1. Create an actor part. Double-clicking the actor opens its editor, but the editor only allows the properties of the actor itself to be modified, for example, to rename it. The editor does not provide access to the model logic contained inside the actor.

Step 2. Right-click on the actor and choose *Open*, or alternatively click on the down-arrow shortcut button on the actor title bar. This activates a new model view for creating model logic inside the actor. Note the model view contains a trapezoidal element labeled *actor*. One such element is always present in the model view and represents the parent actor. This is called the parent proxy, and it allows logic inside the actor to link to and interact with the parent actor part if required. The default parent actor for the entire model is called *root_actor*. Note that the actor path is displayed a the top of the model view window. In this case it displays *root_actor/actor*. Note also that the scenario browser displays the actor hierarchy and clicking on an actor in the scenario changes the current model view to the chosen actor. To return to the root_actor, either click the up-arrow shortcut on the parent proxy, click the up-arrow button on the model view title bar, or select the root_actor from the scenario browser.

Step 3. Inside the actor, create a variable part, open its editor and set its value to zero.

Step 4. Also inside the actor, create a function part, link it to the variable part, and in the function editor, change its name to *increment*, and enter the following code:

```
link.variable += 1
```

Naturally, it will be necessary for external logic to interact with, or send information to, the logic inside the actor. This is accomplished by adjusting the interface level of specific parts inside the actor. The default interface level for all parts is 0, which means the part can only be accessed inside its parent actor. This is analogous to a private member in object oriented programming. Setting the interface level to 1 allows the part to be accessed by logic outside of its parent, analogous to a public member. Setting the interface level to 2 allows the part to also be accessed by logic outside of its parent's parent. A part with a non-zero interface level is called an interface part because it interfaces with logic outside of its parent actor. It is possible to expose a part all the way up to the root_actor by adjusting the interface level. This allows any part to be linked to any other part, regardless of their relative locations in the actor hierarchy.

Step 5. Select the function part created above, and in the Object Properties window (usually found on the right side of the screen), click on the Interface Level dropdown menu and choose *1: actor* as shown in the figure below.

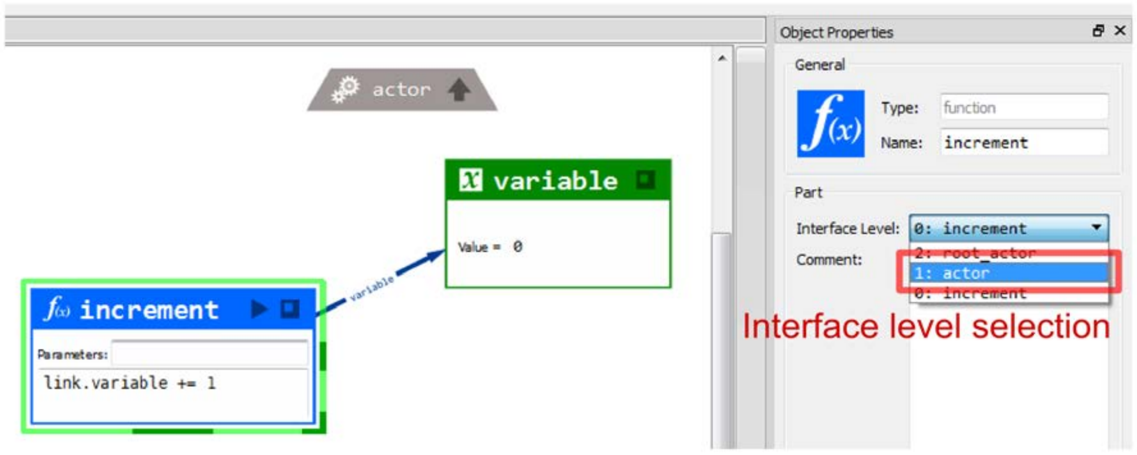
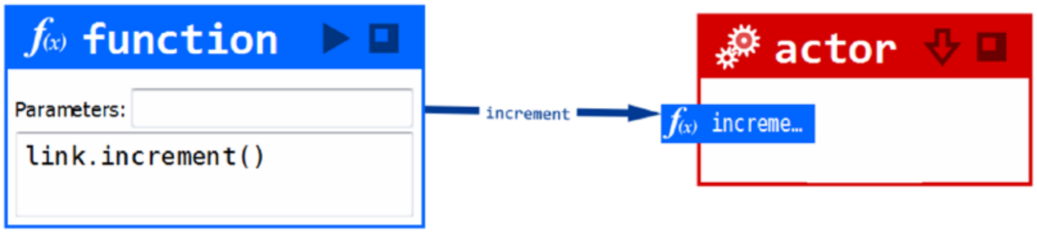


Figure 20: *Selecting a part's "Interface Level" from the "Object Properties" window.*

This sets the interface level to 1, making the part accessible from outside the actor, and adds a yellow interface bar above the function part in the model view which acts as a visual queue indicating which parts are interface parts. The left side of the interface bar displays a number indicating the interface level, and the right side of the interface bar displays two numbers that indicate the number of incoming and outgoing links that are connected to the part, but which are not visible in the current model view.

Step 6. Return to the root_actor. Note that the actor, as seen from the outside, now displays the increment function on its exterior. This representation of the part on the exterior of its parent actor is called a *port*.

Step 7. Create a function in the root_actor and link it to the port on the actor. The result should look like the following.



```
link.increment()
```

Figure 21: *Linking to an interface part visible on the exterior of an actor part.*

Step 8. Run the function a few times, and then open the actor by clicking the down-arrow shortcut on the actor title bar. Note that the variable has been incremented.

3.6 Debug mode

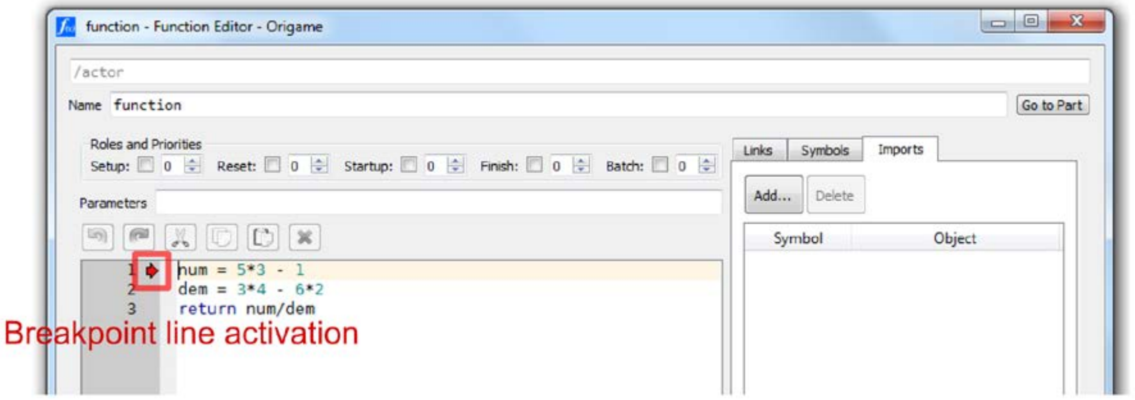
ORIGAME contains an integrated debugger which allows the code execution to be tracked line by line in order to validate the code and investigate errors. The follows steps demonstrate how to use to the debugger.

Step 1. Create a function part and add the following code in the editor:

```
num = 5*3 - 1

dem = 3*4 - 6*2

return num/dem
```

Step 2. Run the function. Note that a divide-by-zero error occurs. When errors occur in ORIGAME, a warning icon is displayed under the part that encountered the error. All active problems in the model, including Python errors, are also displayed in an *Alterts* panel usually located in the lower left docking area of the application. Any details related to an alert (such as the error message in this case) can be viewed in the alert panel by selecting the specific entry in the list of alerts. Although this particular case is very straightforward, we can use the debugger to locate the error inside the code.

Step 3. Open the function editor and click the space immediately to the right of the number 1 in the line number column (shown in the figure below). This creates a breakpoint at the first line of code in the function. Click OK to close the editor. Note that the function now displays a red stop-sign icon to indicate that it has a breakpoint. Be careful not to place the breakpoint on a comment line because code execution skips these lines and the debugger will not be triggered.



```
num = 5*3 - 1
dem = 3*4 - 6*2
return num/dem
```

Figure 22: *Inserting a breakpoint for the debugger in a function part editor.*

Step 4. Right-click on the function and choose *Run Debug*. This will run the function in debug mode, and when the breakpoint line is hit, execution pauses and a debug window opens that displays the current line in the code and provides standard debugging features.

Step 5. In the debugger, click the *Step* button twice to advance the code by two lines. Note the debugger indicates that the two local variables *num* and *dem* have been created.

Step 6. The values of the local variables (and any other Python object that is in scope) can be inspected by typing in the *Python Expression* field and hitting *enter* or clicking *Evaluate*. Use this feature to inspect the values of *num* and *dem*. Note that *dem* is zero and the next line divides by *dem* which then raises the divide-by-zero exception.

In addition to *Step*, the debugger provides standard code navigation features including *Step Into* to follow the code execution into a function call, *Continue* to resume running in debug mode from the current line, and *Stop* to stop debugging. At any time, the values of any in-scope Python objects can be inspected as described above.

It is also possible to run a simulation in debug mode, in which case the simulation will start and run as normal but will stop and open the debugging window when a breakpoint is encountered. To run the simulation in this mode, click on the *Debug* checkbox in the *Main Simulation Control* panel to enable debug mode, and then proceed to run the simulation.

4 Conclusion

This concludes the tutorial. The reader will now be familiar with the basic ingredients for creating and running simulations in ORIGAME. There are other part types and other features that this tutorial did not touch on. However, the information provided here should be sufficient to begin exploring and using the rest of what ORIGAME has to offer. It should also give the reader the tools needed to understand and to build more complex models. The *User Manual* is the next resource the reader should refer to in order to learn more about ORIGAME.

At this point, the reader is encouraged to experiment further with the part types already introduced in this tutorial and with the other part types that are available. Right-clicking on any part and selecting *Help* will open the *User Manual* to the page describing that part.

This is the first release of ORIGAME after its contract development phase. Although the software was thoroughly tested and has proven to be stable in a variety of real applications, it is possible that users will encounter problems. If bugs are discovered, or any aspect of the application does not work well, or important features are missing, please provide this (or any other) feedback to one of the following individuals:

Stephen Okazawa—ORIGAME Project Lead, stephen.okazawa@forces.gc.ca

Luminita Stemate—Director Research Workforce Analytics, luminita.stemate@forces.gc.ca

References

[1] Okazawa, S. *A Discrete Event Simulation Environment Tailored to the Needs of Military Human Resources Management.* Proceedings of the 2013 Winter Simulation Conference. R. Pasupathy, S.-H. Kim, A. Tolk, R. Hill, and M. E. Kuhl, eds. Washington, DC.

DOCUMENT CONTROL DATA

*Security markings for the title, authors, abstract and keywords must be entered when the document is sensitive

1. ORIGINATOR (Name and address of the organization preparing the document. A DRDC Centre sponsoring a contractor's report, or tasking agency, is entered in Section 8.) Director General Military Personnel Research and Analysis Defence Research and Development Canada 101 Colonel By Drive Ottawa, Ontario K1A 0K2 Canada	2a. SECURITY MARKING (Overall security marking of the document including special supplemental markings if applicable.) CAN UNCLASSIFIED 2b. CONTROLLED GOODS NON-CONTROLLED GOODS DMC A

3. TITLE (The document title and sub-title as indicated on the title page.)

Operational Research Integrated Graphical Analysis and Modelling Environment (ORIGAME) tutorial

4. AUTHORS (Last name, followed by initials – ranks, titles, etc., not to be used)

Okazawa, S.

5. DATE OF PUBLICATION (Month and year of publication of document.) February 2018	6a. NO. OF PAGES (Total pages, including Annexes, excluding DCD, covering and verso pages.) 31	6b. NO. OF REFS (Total references cited.) 1

7. DOCUMENT CATEGORY (e.g., Scientific Report, Contract Report, Scientific Letter.)

Reference Document

8. SPONSORING CENTRE (The name and address of the department project office or laboratory sponsoring the research and development.)

Director General Military Personnel Research and Analysis
Defence Research and Development Canada
101 Colonel By Drive
Ottawa, Ontario K1A 0K2
Canada

9a. PROJECT OR GRANT NO. (If appropriate, the applicable research and development project or grant number under which the document was written. Please specify whether project or grant.) 04GF07-002	9b. CONTRACT NO. (If appropriate, the applicable number under which the document was written.)
10a. DRDC PUBLICATION NUMBER (The official document number by which the document is identified by the originating activity. This number must be unique to this document.) DRDC-RDDC-2018-D173	10b. OTHER DOCUMENT NO(s). (Any other numbers which may be assigned this document either by the originator or by the sponsor.)

11a. FUTURE DISTRIBUTION WITHIN CANADA (Approval for further dissemination of the document. Security classification must also be considered.)

Further distribution done by approval from Defence Research and Development Canada's Authority

11b. FUTURE DISTRIBUTION OUTSIDE CANADA (Approval for further dissemination of the document. Security classification must also be considered.)

NONE

12. KEYWORDS, DESCRIPTORS or IDENTIFIERS (Use semi-colon as a delimiter.)

ORIGAME; Tutorial; simulation; Workforce Analysis

13. ABSTRACT (When available in the document, the French version of the abstract must be included here.)

This Reference Document is a tutorial that introduces new users to the ORIGAME simulation environment. It provides step-by-step instructions that will guide the reader through the creation of a simple model. Included in the instructions are detailed explanations of the concepts and features being used to accomplish specific tasks. After completing this tutorial, the reader should be able understand most of the inner workings of existing models and will be able to build new models for real applications. The ORIGAME User Manual is the complete reference for all of ORIGAME's features and is the next document that readers should consult to learn more about the software.

Ce document de référence est un tutoriel présentant l'environnement de simulation ORIGAME aux nouveaux utilisateurs. On y explique les étapes à suivre pour créer un simple modèle. Dans les instructions, il y a des explications détaillées sur les concepts et les fonctions utilisés pour accomplir des tâches précises. À la fin du tutoriel, le lecteur devrait comprendre les rouages des modèles existants et être en mesure de créer de nouveaux modèles pour des applications réelles. Le manuel d'utilisateur du logiciel ORIGAME est un document de référence complet pour toutes les fonctions de l'outil. Les lecteurs devraient le consulter pour en apprendre davantage