# Origame User Manual (115471-005)

***Release 0.7.0 beta (2017-09-13)***

**CAE**

**Sep 13, 2017**

CHAPTER

ONE

# INTRODUCTION

Origame is an application developed by the *R4 HR* Technology Demonstration Project. It is a *DND* platform for modelling and simulation of future military *HR* requirements. It provides many convenient features that were specifically designed with this task in mind including an embedded programming language that is both powerful and friendly to non-programmers; an architecture that allows scenario logic to be easily moved, duplicated and reconnected; an integrated relational database for processing large amounts of data; a flexible signalling system for communication between simulation events; and a graphical user interface that integrates modelling, simulation and analysis activities into a single workspace.

Through these features, Origame will enable military *HR* scenarios to be developed and modified by defence scientists more quickly and easily as simulation requirements change. As a result, simulation support to decision making will be more responsive to operational and strategic needs. In addition, a key advantage of Origame is that it makes feasible the prospect of creating unified scenarios that interconnect the various subsystems of military *HR*. By capturing the interactions between these subsystems, Origame will provide foresight to *DND* decision making that is more accurate and comprehensive than previous modelling and simulation approaches allow. This Manual provides both the first time and experienced user the information they need to build and run these scenarios.

Origame is designed to operate on standard Windows-based computing platforms. *DRDC* initially developed a Prototype of this simulation environment and Origame is the result of productizing and extending the Prototype into a more robust application with a more Windows like user interface.

CHAPTER

TWO

# GETTING STARTED

## 2.1 Installation

### 2.1.1 Hardware Requirements

Origame has been designed to run on a modern *PC* or laptop with the following minimum specifications:

- Windows 7 (64bit)
- 2.5 GHz dual-core *CPU*
- 4 *GB* of *RAM*
- Dedicated video card with 1 GB of video memory
- 500 *GB* of storage

In addition, a minimum display resolution of 1152 x 864 pixels is recommended.

### 2.1.2 Software Requirements

In addition to the software necessary to run the Origame application, it is recommended that the following software be installed on the target platform:

**Microsoft Office**

- Origame cannot be installed on computers with 32–bit versions of *MS* Office also installed. However, 64–bit *MS* Office is supported but not required for installation.
- Log files are generated in tab separated value format. The application log file generated during a *GUI* session can be viewed in the application. Batch runs will generate multiple log files which are most easily read and manipulated in Excel.
- Origame Sheet Parts can be imported from and exported to *MS* Excel and Plot Parts can be exported to *MS* Excel.

**Microsoft Access**

- Origame can read from and write to *MS* Access databases.

**Notepad++**

- Origame scenario files (.ori file extension) are generated in *JSON* format. There is no need to open them in any other application, but they may be easily read using Notepad++ by more experienced users, if desired.

**Internet Explorer**

- Online help links have been optimised using Internet Explorer, but Google Chrome can also be used.

### 2.1.3 Licensing

Origame is the property of the Government of Canada, and managed by the Military Personnel Research and Analysis (*MPRA*) group at Defence Research and Development Canada (*DRDC*). It may only be accessed and used by personel of the Government of Canada, of authorized contractors, and of licensed organizations. For the latter, the terms and conditions of access and use are detailed in the **LICENSE.txt** file located in the *Application* folder of the installation package and in the Origame package folder once installed (*c:\Python35\Lib\site-packages\origame*). It can also be accessed by running the Origame *GUI* and navigating to the *Help* menu, *About...*, and clicking either of the links therein.

### 2.1.4 Installation Steps

The installation is launched by running the batch file **install_origame.bat** located in the top-level of the directory path of the installation package. By default, the installer will install Python 3.5.2 to the directory `C:\Python35`. However, this default directory can be changed by providing an optional argument, *–python-path*, along with a valid Window's path argument, as the **first** arguments to the installer. If Python is already installed in either the default directory or the one provided by the optional –python-path argument, the installer will skip the Python installation step and proceed to install Origame inside a Python virtual environment (*venv*).

By default, the *venv* is installed in the directory `C:\Origame RELEASE VERSION`. However, this location can be changed by using the argument *–venv-path* along with a valid Window's path argument. The installer can be used to install multiple self-contained copies of Origame each having their own shortcut automatically created on the desktop by specifying a different *venv* path each time it is run. Each *venv* has been configured to have access to the base Python installation's site-packages folder as well as its own. This allows Origame to import from packages installed on the user's base system as well as those installed on the *venv* for use with Origame. Some of the Origame application package dependencies include Qt 5.7, QScintilla 2.9.3, SIP 4.18.1, and Matplotlib 1.5.0.

The installation instructions for Origame are enumerated below.

1. Select the *PC* for installation that conforms to one of the three following configurations:
   (a) [Recommended] Windows with 64-bit *MS* Office installed: As Origame database features work only with 64-bit *MS* Access, Origame may be installed on Windows systems where 64-bit *MS* Office is installed.
   (b) [Recommended] Windows with no *MS* Office installed: Alternatively, on Windows systems where no version of *MS* Office is installed, it is recommended to install Origame using the optional argument *with-access* to install the included 64-bit *MS* Access database driver that enables database features.
   (c) [Not Recommended] Windows with 32-bit *MS* Office installed: Installation on a system with 32-bit *MS* Office is possible but the optional 64-bit Access database driver should **not** be installed (the default mode is not to install it) as doing so will interfere with or corrupt 32-bit *MS* Office. Under this configuration, the Origame database import and export features will not work.
2. From the Windows Start menu, type "cmd" in the Search field (alternatively search for "Command Prompt"). Right–click the search result in the list and select **Run as administrator**.
3. Change the directory to the top level of the installer package (where the **install_origame.bat** resides).
4. Type the command `install_origame.bat` along with any of the optional arguments (listed below) and hit **Enter**.
5. The installer shows the installation progress on the screen. It does not prompt the user for input.
6. Installation is complete once the message "Origame (RELEASE VERSION) successfully installed" is printed to the command prompt window.
7. Double-click the desktop shortcut to start the application or on an existing scenario ".ori" file.

Each of the optional arguments available to use with the installer are enumerated below.

1. `--python-path python_path`: By default, the installer will install Python 3.5.2 to the *python_path* `C:\Python35` if it is not already installed there. Use this argument to specify a different *python_path* to install to or to indicate to the installer where Python is already installed if it's not in the default location (see notes below).
   - Note 1: This argument, if used, must be the first argument.
   - Note 2: Only Python 3.5 is supported with installation of Origame.
   - Note 3: Only one instance of Python 3.5 can be installed on Windows. If the installer is used with this argument on a Windows system where Python is already installed, and the argument specifies a directory other than the one where Python 3.5 is currently installed, then the installer will stop after trying to "modify" the current installation and inform the user to check that Python is not currently installed.
2. `--venv venv_path`: By default, the *venv* is installed in the *venv_path* `C:\Origame RELEASE VERSION`. Use this argument specify a different *venv_path*.
3. `--with-access`: By default, the 64-bit *MS* Access database driver is not installed. Use this argument to install it (this argument should only be used on Windows systems where 64-bit or 32-bit *MS* Access are **not** installed (see installation step 1).
4. `--no-vc-redist`: This argument indicates that the Visual C redistribution should not be installed/reinstalled on the computer system.
5. `--no-file-type-assoc`: By default, .ori and .orib file types will are automatically associated with Origame. This flag will prevent that assocation during installation.

## 2.2 Assistance and Problem Reporting

The Origame application has Help information built in, as described in section *User Interface*. If further assistance is required or to report problems, please contact:

| Issue | Name | Email address | Telephone |
|---|---|---|---|
| General questions / technical questions / report problems | Stephen Okazawa | stephen.okazawa@forces.gc.ca | 613–992–0261 |
| General questions | Luminita Stemate | luminita.stemate@forces.gc.ca | 613–996–0751 |

CHAPTER

THREE

# USING ORIGAME

Origame provides a framework to create discrete event models by placing basic building blocks called *parts* in a scenario and connecting them together using *links* from one part to another. The following sections describe the application's modes of operation, and then covers basic concepts of Origame including parts, links, and other concepts including waypoints, interface ports, and simulation *events*. This section also covers the various elements of the user interface, how to open and navigate a scenario, how to build models including adding, removing, editing, and interfacing scenario parts, and concludes with running simulations in *Main* and *Batch* modes.

## 3.1 Modes of Operation

Origame can be launched in two variants: *GUI* and Console. The *GUI* variant, shown in Figure 3.1, provides a graphical user interface allowing the user to build and run scenarios. The Console variant is designed to quickly run a scenario in the background and does not provide any graphical output as shown in Figure 3.2. In order to build a scenario, the *GUI* variant is required.

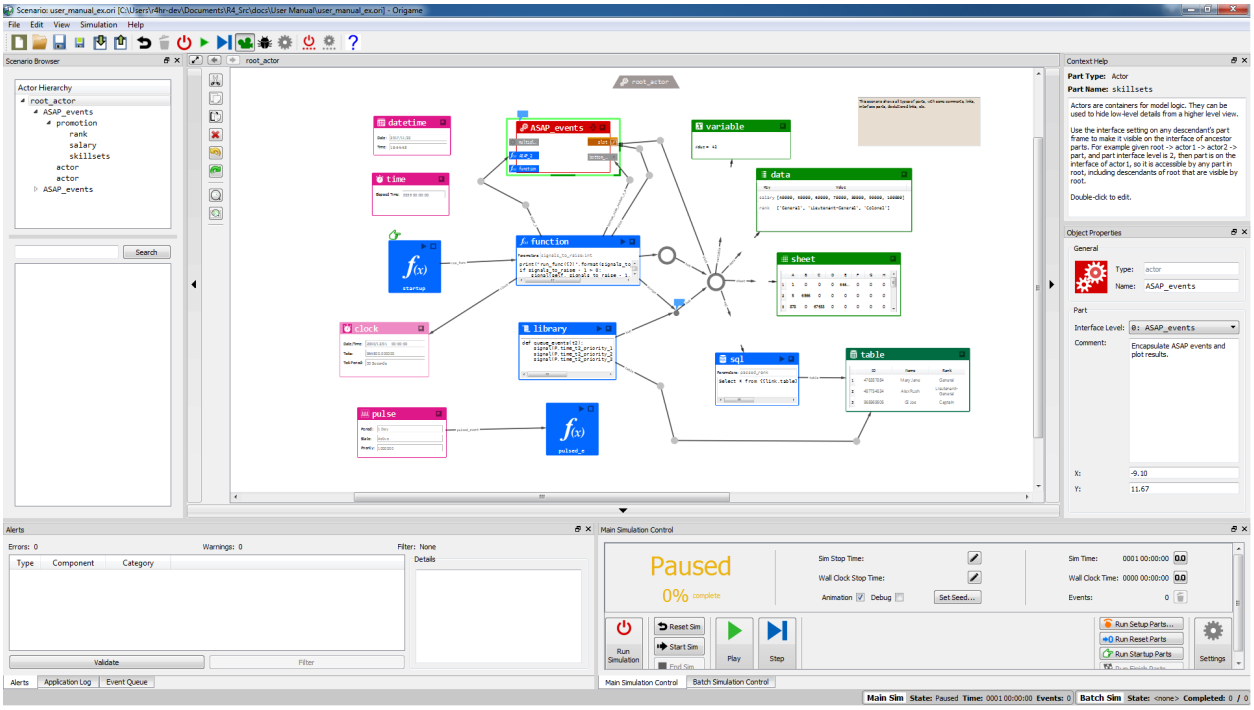


Fig. 3.1: GUI variant of Origame showing the default view of the main window.

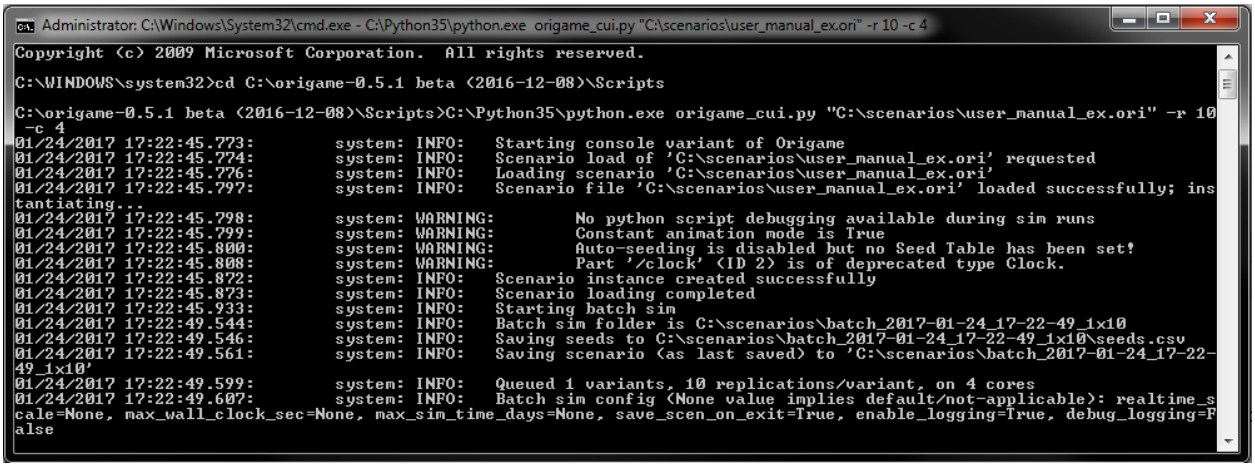


Fig. 3.2: Console variant of Origame running in the command console.

The *GUI* variant provides two modes of running a scenario: Main and Batch. In Main, only one instance of a scenario can be run. The user may also choose to turn Animation (*GUI* updates) on or off. In Batch, multiple replications of the same scenario can be run either in sequence (using one core) or in parallel (using multiple cores).

| Variant | Mode | Animation | # Replications | Best for running... |
|---|---|---|---|---|
| *GUI* | Main | On | 1 | simple scenarios which self modify and update during the run |
| | | Off | 1 | simple scenarios where the final result is displayed graphically and interim updates are not important |
| | Batch | | 1 or more | complicated scenarios, with final results written to file |
| Console | | | 1 or more | very complicated scenarios which take a long time to run, with final results written to file |

This remainder of this section describes how to use the *GUI* variant of Origame.

## 3.2 Starting Origame

Origame is launched by either of the following actions:

1. Double-clicking the Origame icon, shown in Figure 3.3,
2. Double-clicking on an existing scenario ".ori" file.

The desktop icon is created during installation as is the association with the ".ori" file extension. The icon can be pinned to the Windows Taskbar or Start menu by clicking and dragging the icon to those locations. Note that if another version of Origame is installed, even an earlier version, double-clicking a scenario ".ori" file will open automatically with the last version of Origame installed, not necessary the newest version.

When Origame starts, a splash screen displays the name of the application, its version number, licensing information, and third-party components required by Origame to run. The splash screen will close automatically after a few seconds but it can also be closed immediately by clicking on it. The information shown in the splash screen is also shown in the About box which is launched by going to the Origame *Help* menu and selecting *About*.

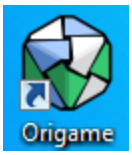


Fig. 3.3: Start Origame by double-clicking the application icon.

## 3.3 Basic Concepts

### 3.3.1 Parts

Origame allows the user to create discrete event models by creating basic building blocks called *parts* and placing them in a *Model View*. A description of each part, including how it can be used and the options available in its editor, is included in section *Part Reference*. See *Creating Parts* for information on how to create parts.

The following subsections delve into how the different part types can be further categorized. It begins with a discussion on part *frames*, a component of all parts that *may* or *may not* be visible. The special nature of *Actor parts* as a fundamental building block is then discussed, followed by the introduction of *executable parts*, *interface parts*, and finally overviews certain deprecated parts that have been made *non-creatable*.

#### Parts with Visible Frames

Parts with visible frames are displayed with a rectangular frame in which is set the part content that defines the part type as shown in Figure 3.4. The part type is signified by the icon that appears in the top left corner of the frame. The part's name appears next to this icon. Clicking the upper right short-cut on a part frame will toggle the Detail Level of the part to *minimal* so that only its icon is displayed. Clicking the short-cut again toggles the part back to *full* Detail Level. This type of part includes *Actor*, *Button*, *Data*, *Datetime*, *Function*, *Library*, *Plot*, *Pulse*, *Sheet*, *SQL*, *Table*, *Time*, and *Variable* parts.


Fig. 3.4: Visible framed data part (left) and invisible framed multiplier (right).

#### Parts with Invisible Frames

The *Info*, *Hub*, *Multiplier*, and *Node* parts do not show their frame as shown in Figure 3.4. While the Info part does have part content, the others do not and are used to route *Part* and *Frame* information between parts that are not linked directly to each other.

### Actor Parts

Actor parts are used to modularize, organize, and encapsulate related implementation details of a model into a hierarchy of parent and child actors. Figure 3.5 shows an actor part in the *Model View* with the model hierarchy of parent and child actors displayed in the *Actor Hierarchy panel* to the left. By clicking on an actor listed in the hierarchy, or the *Go Into Actor* shortcut (down arrow button) on an actor part's frame, the actor's contents can be viewed. By default, the application displays the contents of the top-level *root* actor of a scenario on application start-up. The *Model View* always displays the contents of a single actor. Additionally, actor parts facilitate link connections across the actor boundary by providing *interface ports* for parts inside the actor that have raised their *Interface Level* above the base level.

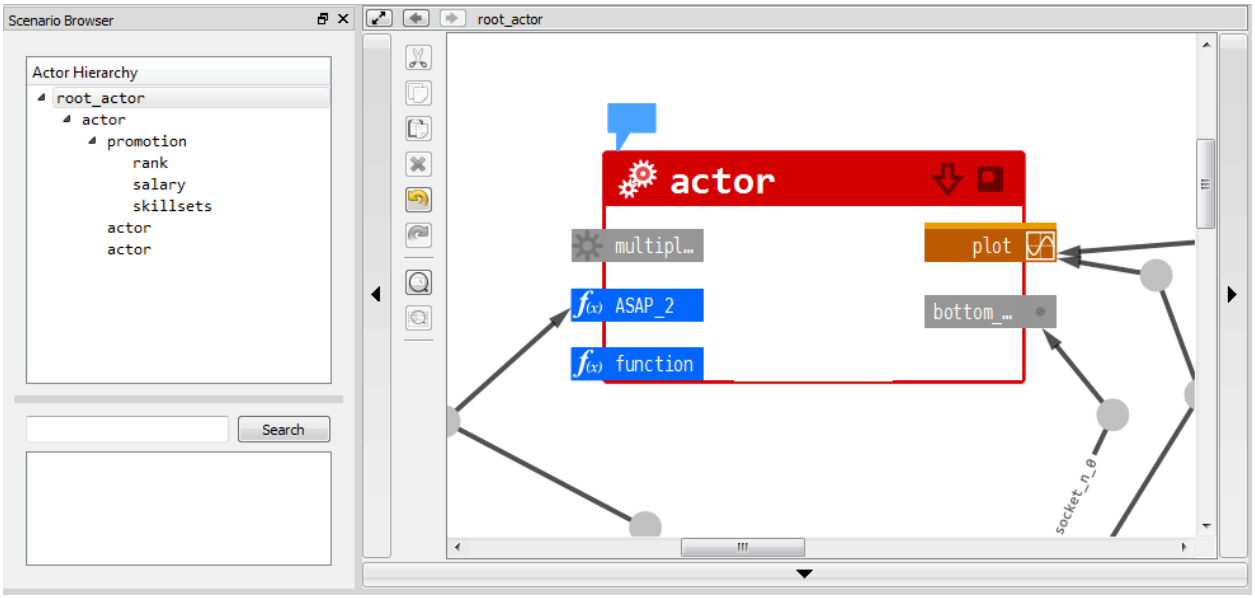


Fig. 3.5: The *Model View* showing the root actor, a child actor, and actor hierarchy in the *Scenario Browser*.

### Executable Parts

There are four types of Executable parts that include the *Function part*, *SQL part*, *Pulse part*, and *Multiplier part*. These parts can be signaled by other executable parts to define discrete simulation events that appear on the simulation *Event Queue*. If a function part is linked to another part, its script has access to that part using the *alias* specified by the link. In other words, aliases of the links that connect a function part to other parts constitute the namespace of the function part. Section *Programming Reference* describes the *API* and attributes for each part type which a function may use to access a target part's contents or properties.

Multiplier parts provide a one-to-many interface for forwarding signals. For example, a multiplier that is connected to a Function or *SQL* part, can signal any number of other parts connected to the multiplier when the Function or *SQL* part signal the multiplier. The multiplier forwards that signal to each part connected to itself.

Pulse parts that are connected to other executable parts will place those connected parts on the event queue periodically according to the pulse period that has been defined.

See *Creating, Editing and Triggering Events* for information on how to create events.

### Interface Parts

Any part can be exposed outside of its parent actor by changing a property called *interface level*. A part exposed this way is termed an Interface Part. The default level is 0 and indicates the part is not exposed outside of its parent actor: the part may only link to its parent, to/from siblings, and to/from interface parts of sibling actors. If the level is increased to 1, the part is exposed to the surroundings of its parent actor via an interface port on its parent's left or right side. These ports can be linked together which effectively links their associated parts together. Links between parts in different actors are called *elevated links*.

The maximum interface level of a part is the "distance" from the root. Hence if the root actor contains a child actor A which contains another child actor B which contains a part C, the interface level of C can be at most 3, that of B at most 2, and that of A at most 1. Actors can also be interface parts themselves, independently of the descendant parts exposed on their boundary.

When the maximum level is used on a part, that part will be visible outside the scenario when the scenario is imported into another scenario. This allows the scenario to define which parts are "public" i.e. exposed to importing scenarios that use them as a model component.

A part's boundary actor is the *ancestor actor* (up the chain of parents) beyond which the part is not visible. When the interface level of a part is 0, the boundary actor is its parent; when level is 1, boundary actor is its grand-parent; etc.

See *Linking Interface Parts* for information on how to create interface parts and elevated links.

### Non-Creatable Parts

Non-creatable parts are parts that have been deprecated in favour of another part or set of parts but continue to be supported by Origame in order to facilitate the use of scenarios that have been built using the deprecated part. As such, these parts can be loaded, saved, copied, and moved within their legacy scenario. However, new instances of the part cannot be created. When a scenario with a deprecated part is loaded, a warning will be printed to the Log panel.

As shown in Figure 3.6, non-creatable parts, such as the deprecated *Clock part*, appear slightly transparent compared to nominal, "creatable", parts. The *Clock part* had integrated a calendar, time, and tick components. The calendar and time components are now implemented as the *Datetime part* while the the tick component is now in the *Time part*. It is highly encouraged that scenarios with Clock parts replace them with these two parts.


Fig. 3.6: The non-creatable *Clock part* (left) and creatable *Time part* (right).

## 3.3.2 Links

A *link* is a directional link between two parts. Creating a link from one part to another allows the former to access and modify the latter.

The link's default name is the alias of the target part used by the source part when accessing the target part's *API* and attributes. By default, this is the name of the target part unless that name is already in use by another target connected

to the same source part, in which case an Arabic numeral will be appended to enforce uniqueness. The link name can be changed by the user as long as the name is not already used by another outgoing link from the same part. Links can also be set to appear *decluttered* by hiding their middle section, have waypoints added and removed which allow links to follow user-defined paths, and have their target part or endpoint changed as required.

When a link is created, it is automatically offset from the centerline between the two parts to allow a parallel link to be created in the opposite direction without overlapping the first link. For most parts with a single link, this offset is not noticeable. However, for small parts like the Node, Hub, and multiplier, and for ports, the offset will be noticeable if only one line is connected.

See *Linking Parts* for information on how to link parts.

### 3.3.3 Waypoints

Waypoints can be added or removed from a link and moved independently which allows a link's straight line path to be reconfigured to any path in the 2D plain required by the model designer. For example, the link can be drawn around other parts that may lie in between the two connected parts. When a waypoint is added to a link, it bisects the link creating two segments. Further waypoints can be added which continue to bisect the segment to which it is added. While waypoints separate the link into segments, the link remains a single link so selecting any segment selects the whole link. Removing all the waypoints restores the original link to a straight line.

See *Linking Parts* for information on how to use waypoints in models.

### 3.3.4 Events

The Origame simulation engine executes discrete event simulations. This means that the simulation consists of a finite number of events that occur at specific times. In a human resources context, examples of events are recruitment, promotion, starting a training activity, completing a training activity, being posted to a new position, and retirement. In general, events consist of a sequence of instructions that can modify the state of various aspects of the simulation and schedule other events to occur in the future. Once an event completes, the simulation engine selects the next pending event to process and advances the simulation clock to the time of that event.

The simulation engine processes events in the following order of execution:

1. All *ASAP* events using Last-In-First-Out (*LIFO*) execution.
2. The next event in chronological order.
3. If there are multiple events occurring at the same time, it selects the highest priority event(s).
4. If there are still multiple such events, it selects the event that has been in the queue the longest in a First-In-First-Out (*FIFO*) execution.

Once the event has been selected, it is removed from the *Event Queue*, the simulation's global time is advanced to the time of the event, and the simulation engine executes the event Function Part's script passing it the event parameters.

Parts that have corresponding events on the *Event Queue* are indicated in the *Model View* with an event *indicator* displayed at the top-left side of the part's widget. The indicator shows the number of events on the queue associated with that part. The colour of the indicator indicates if the event is the next event, concurrent with the next event, or pending.

In the run state, the simulation engine will continuously process events off of the *Event Queue*, and it will continuously refresh the simulation display in the *GUI* so that any animated effects such as the movement of parts and updating part displays will be observed. The simulation will stay in this state even if there are no events on the queue and will wait for an event to occur. In the pause state, the simulation state is frozen and the simulation engine does not retrieve events from the event queue. In this mode, the user can carry out model building actions in the *GUI* such as creating, manipulating and interconnecting parts. The user can manually step the simulation forward in the pause state. The step will retrieve and process the next event on the *Event Queue*.

Once an event has been added to the *Event Queue* it can be edited or removed from the *Event Queue panel*.

See *Creating, Editing and Triggering Events* for information on creating events.

## 3.4 User Interface

### 3.4.1 Starting and Stopping Origame

The application is launched by double-clicking the application icon that was created during installation. Alternatively, from the Windows Command Prompt enter:

```
>> "C:\Origame_VENV\Scripts\pythonw.exe" "C:\Origame_VENV\Scripts\origame_gui.py",
```

where `Origame_VENV` is the path to the Python Virtual Environment that was created during installation.

The application is closed by accessing the *Exit* option under the *File* menu or by pressing the close shortcut *X* button on the upper right of the application window. If the user tries to exit Origame with *part editors* open that have unsaved changes, a dialog will open that requires user confirmation to discard all changes before proceeding to exit, or else to cancel the exit operation and allow the user to determine which individual editor panel changes to apply and which to discard.

### 3.4.2 Scenario Files

Model files that are created, saved and loaded into Origame are called *scenario* files. Origame can save and load scenarios in the following two formats:

1. *ORI*: *JSON*-formatted files with the extension *.ori* are human-editable files.
2. *ORIB*: Binary-formatted files with extension *.orib*. can be loaded ten times faster than *ORI* files but are not human-editable.

Origame supports the application's Prototype format with the extension *.db* for loading only.

Origame assumes each scenario has its own folder as other files associated with the scenario may be saved along with the scenario .ori file. These other files include images the user may have selected for actor or button parts which are saved to the */images* subfolder that Origame automatically creates when the scenario is saved for the first time.

### 3.4.3 Menus

An overview of the Origame menu is provided in the table below.

| Menu | Description |
|---|---|
| File | File options for creating *New* scenarios, *Open* existing scenarios, *Save* scenarios under the current name, *Save As...* for new files, as well as options for saving application *Preferences*, *Importing* and *Exporting* scenarios. |
| Edit | Edit options for *Cut*, *Copy*, *Paste*, *Delete*, *Undo*, and *Redo*. |
| View | View options to control the application's various panels and icons. A *Model View* sub-menu provides controls to override part *Detail Level* that controls whether parts show full (expanded) or minimal detail in the Model View and zoom options for fitting the entire model into the view or only a selected portion. An option to *Restore Default View* will restore all application panels to their initial configuration when the application was first run. A list of checkable panel options will show or hide the corresponding panel. |
| Simulation | Simulation options for initializing, running, stepping, and debugging a scenario. *Main* and *Batch* simulation options are accessed here along with *Debug* and *Animation* modes. More information about these options is contained in section *Running Simulations*. |
| Help | Help options for accessing the *User Manual*, *Python References* and *Tutorials*, and information *About* the Origame application. |

### 3.4.4 Windows, Panels, and Dialogs

The Origame application window, shown in Figure 3.1, consists of nine panels and one toolbar. The *View* menu provides options for showing or hiding each panel.

#### Model View

The Model View, shown in Figure 3.7, is the panel used to create parts and build simulation models. It displays the contents of an actor part (the root actor at start-up) and shows the hierarchy path at the top of the panel. Refer to *Navigation* for information on how to view different actors in the model. A toolbar on the left side of the Model View contains common shortcuts to Edit and View menu options such as cut, copy, paste, delete, undo, redo, and two zoom options that include *Zoom to Fit all Contents* and *Zoom to Selection Contents*. The model view can be expanded to occupy the maximum window space available, or unexpanded, by using the three expnasion buttons located on the left, right, and bottom sides of the panel. The canvas size of the Model View is infinite in theory but will be limited by system resources in practice.

#### Scenario Browser

The Scenario Browser consists of two sub-panels that are depicted in Figure 3.8:

1. **Actor Hierarchy**: The Actor Hierarchy panel displays the parent-child relationship of all actors in the current scenario by nesting child actor parts under their parent actor. Actors are shown alphabetically and then by order of creation if they share the same name. By clicking an actor part in the hierarchy, the contents of the actor will be displayed in the *Model View*.
2. **Search**: The Search panel is used to find parts and part contents in the model. The *Search* button changes to a *Cancel* button while the search is is performed. To stop the search, press the *Cancel* button. The results are displayed as a path from the child actor in the root, down to the searched-for part or the part containing the searched-for content. Double-clicking a part in the search results will navigate the *Model View* to that part. If the searched part is removed from the scenario, the corresponding search result will change to red font and navigation to the part will be disabled. If the part is moved to a new location, the search result path will update and navigation to the part enabled.

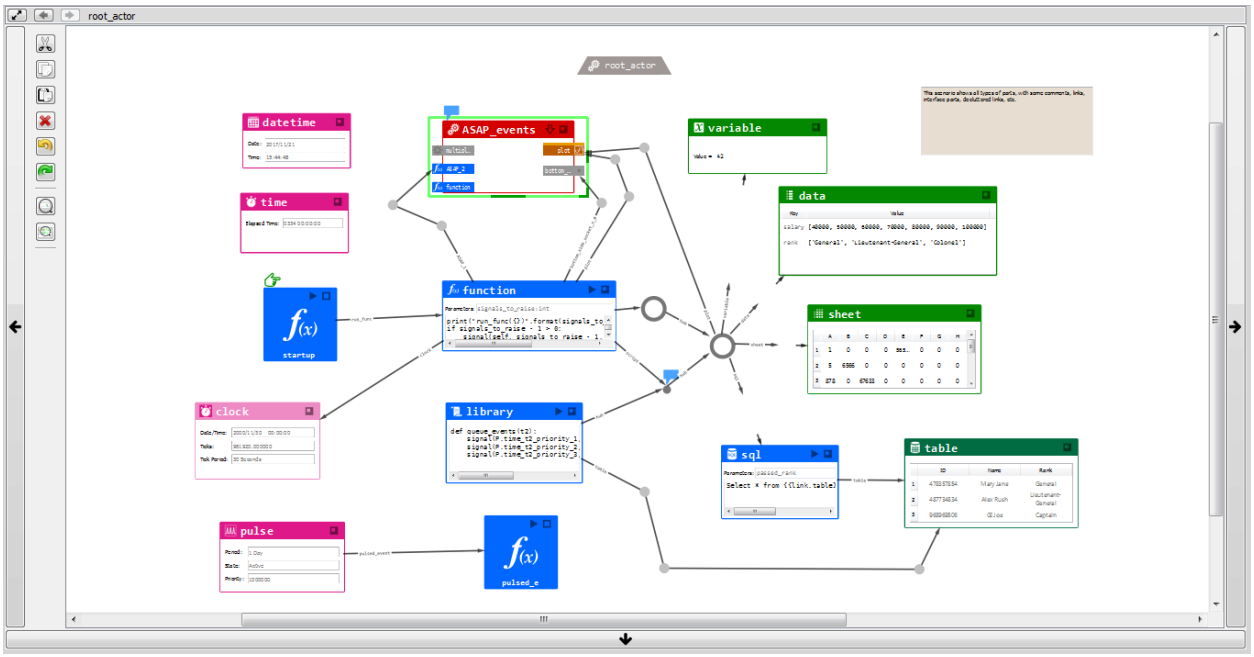


Fig. 3.7: The Model View Panel.

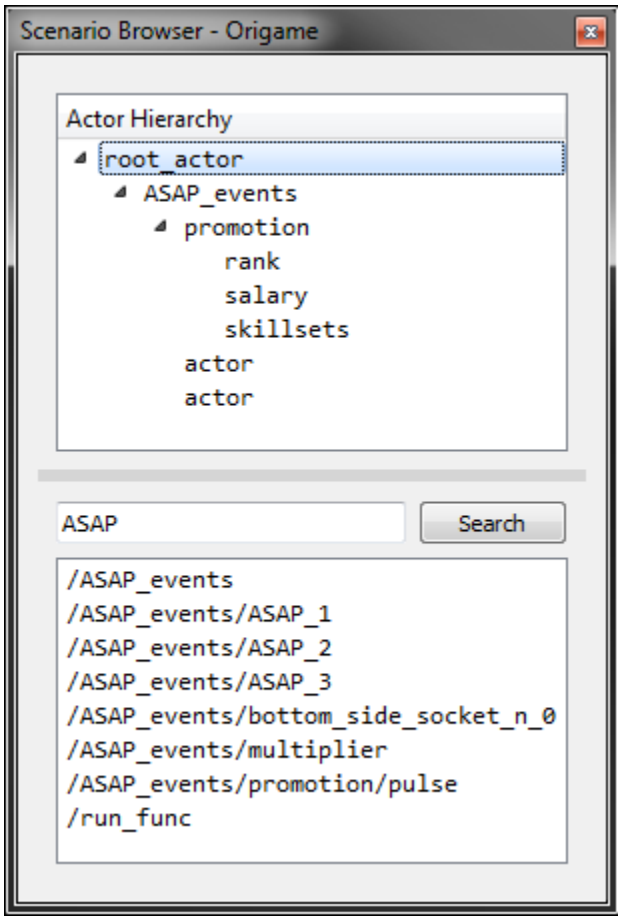


Fig. 3.8: The Scenario Browser Panel.

### Alerts

The Alerts panel, shown in Figure 3.9, displays any warnings and errors generated by the model. There are two types of alerts:

1. Automatic: alerts that are automatically generated. Every part type and scenario component can generate its own set of automatic alerts when certain operations are performance. An example is syntax errors that occur in *script-based parts* when those parts are run, or are cleared by fixing the issue and re-running the affected parts.
2. On-Demand: alerts that are generated during manually-triggered model validation. Every part type and scenario component can generate its own set of on-demand alerts during validation. Examples are missing link, unreferenced link, and datetime part synchronization warnings.

The model can be validated by clicking the *Validate* button on the Alerts panel. Model validation traverses the entire scenario parts and components to request validation. The produced on-demand alerts get displayed in the Alerts panel, and any pre-existing on-demand alerts are discarded. Clicking on an alert in the panel will display a more detailed message in the "Details" section of the panel.

With a part's alert selected, pressing the *Filter* button will cause the panel to show only alerts associated with that part. Clicking the *Un-filter* button will remove the filter. Double-clicking the alert will cause the *Model View* to show the corresponding part. The part itself will display the applicable alert indicator on its bottom left side (see *Part Indicators and Markers*). Clicking this indicator will also filter the Alerts panel to only show alerts associated with that particular part.

Actor parts display an alert indicator if any of their descendant parts (children, grand-children etc) raise an alert, as all alerts propagate up the actor hierarchy. As for other parts, clicking that indicator will filter the Alerts panel to display only alerts associated with that actor and all its descendants.

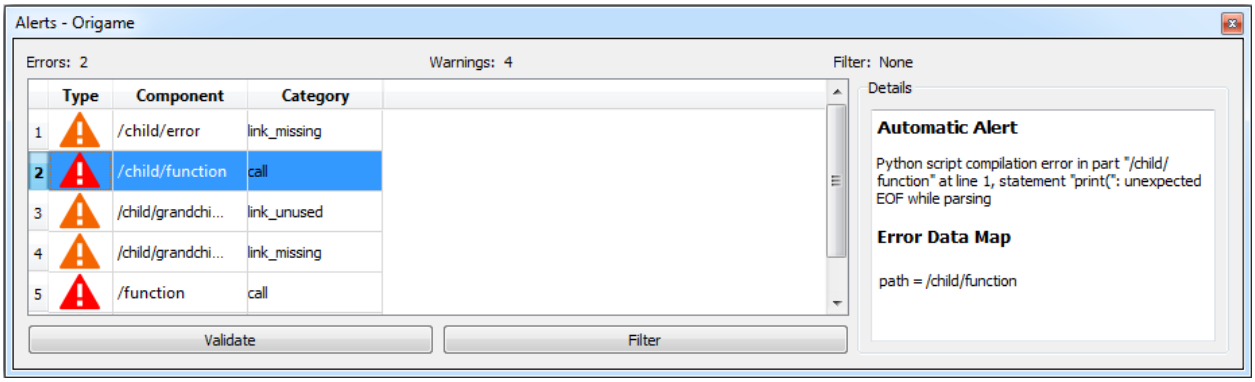


Fig. 3.9: The Alerts panel.

### Event Queue

The Event Queue, shown in Figure 3.10, shows all events that are queued to execute at a specified time and with a specified priority within the time. This panel also displays the part that the event corresponds to, the part's type, arguments, and path in the Actor Hierarchy. The panel can be used to edit and delete individual events or clear all events at once. Refer to *Creating, Editing and Triggering Events* for more information adding and changing events.

### Application Log

The Log panel is shown in Figure 3.11 and displays various log messages issued by the application. To the right of the log message display is the log operations panel that contains a toggle bar to show or hide the operations panel, the filter

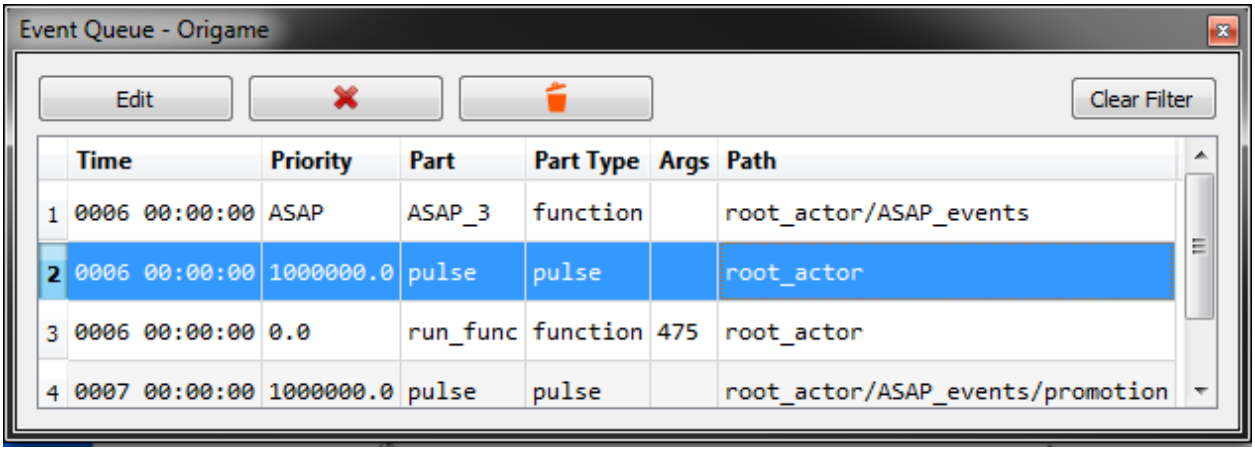


Fig. 3.10: The Event Queue panel.

selector, the log *Save...* button, and the *Hide Previous* message button. The details of the Log panel's functionality are described below.

1. Bookmark: Selecting a line in the log panel will bookmark the line by highlighting it. The log can the be scrolled with the bookmarked line remaining highlighted.
2. Highlight: By using the mouse to select a word or group of words, all instances of the selection in the log panel will be highlighted.
3. Filter: Log messages are filtered according to the filter selector. Logs are issued via two *loggers*: one is called "System" and the other is called "User". The former are issued by the application, the latter by user code. Log messages are ranked by level of importance (high to low): Critical, Error, Warning, Info, and Print. While log messages are always logged regardless of the filter settings, unchecking a box in the filter will hide all corresponding messages from the log panel, while checking a box will display them again.
4. Hide/Unhide: With a line bookmarked, clicking the *Hide Previous* button will hide all lines occurring before it and indicate how many lines are hidden. The hidden lines can be shown again by clicking *Show Hidden*.
5. Save: The log can be saved by pressing the *Save...* button in the panel.

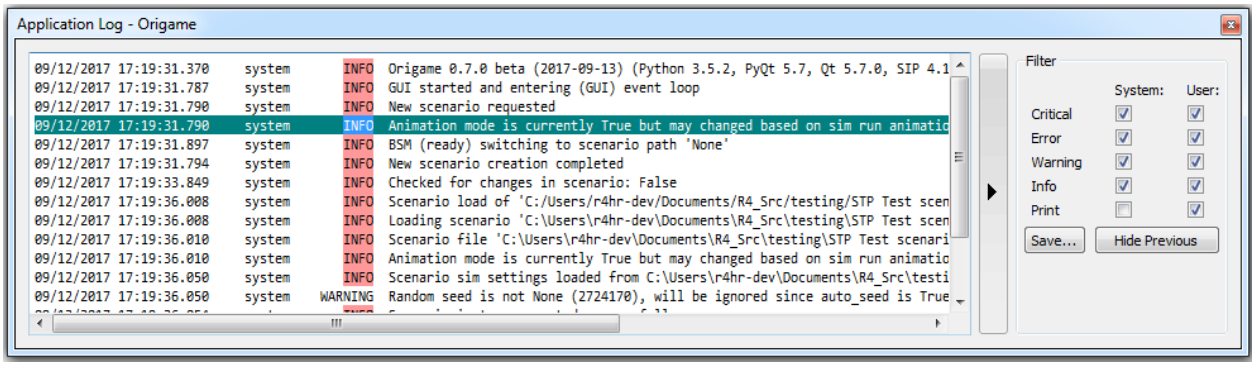


Fig. 3.11: The Application Log panel.

### Context Help

The Context Help panel, shown in Figure 3.12, displays information about the part that the mouse cursor is currently hovering over.

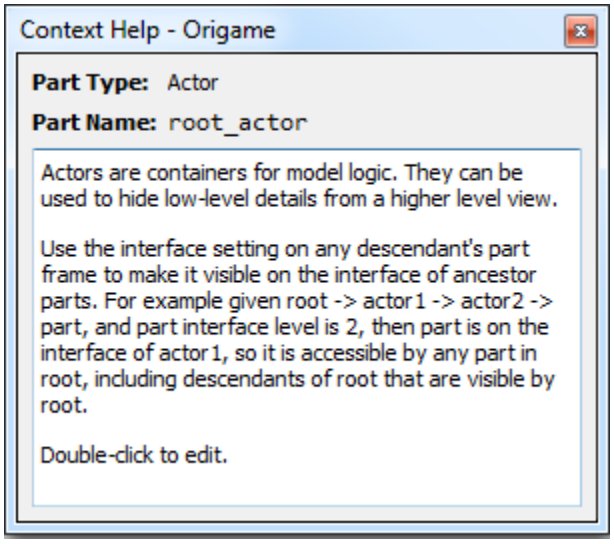


Fig. 3.12: The Context Help panel.

### Object Properties

The Object Properties panel displays the properties of the selected object and provides an interface to edit them. Objects that will display properties when selected include *parts*, *links*, and *waypoints* and are shown in the figure below. Object Type and Object Name properties are common to all objects. For selected parts and links, the Object Name is the object's name and can be edited in the panel. For selected waypoints, the name is the waypoint's *ID* number that cannot be modified. The Object Type cannot be edited for any object.

The properties displayed by each object type include:

1. *Part* properties that are displayed, shown on the left of Figure 3.13, include the interface level, comments, and position in X and Y coordinates. Each of these properties can be changed from the panel by editing the corresponding field value.
2. *Link* properties, shown in the middle of Figure 3.13, include declutter mode, source part type, target part type, and number of waypoints. Only the declutter mode can be changed from the panel by checking or unchecking the declutter checkbox.
3. *Waypoint* properties, shown on the right of Figure 3.13, include the X and Y position coordinates which can be updated by editing the values in the panel.

For changes to take effect, press *Enter*, or click away from the input field, or press the *Tab* key to change focus to a different panel field.

### Main Simulation Control

As shown in Figure 3.14, the Main Simulation Control tab contains both status read-outs and control options as well as some common main simulation settings. The main status displays the current simulation state, *Running* or *Paused*, the percent complete, and the current simulation and wall clock times. The simulation and wall clock stop times can be set from this panel, and Animation and Debug modes toggled. The random seed can be set in the scenario's random number generator from this panel as well.

On the lower left of the main control panel are controls to *Run Simulation*, *Play* (*Pause* when running), and *Step* the simulation. Control options to *Reset Sim*, *Start Sim*, and *End Sim* are also available. Four "run role" buttons on the lower right of the panel will run all parts with the corresponding role. The *Settings* button in the lower right of the panel opens the settings dialog where various main simulation settings can be specified.

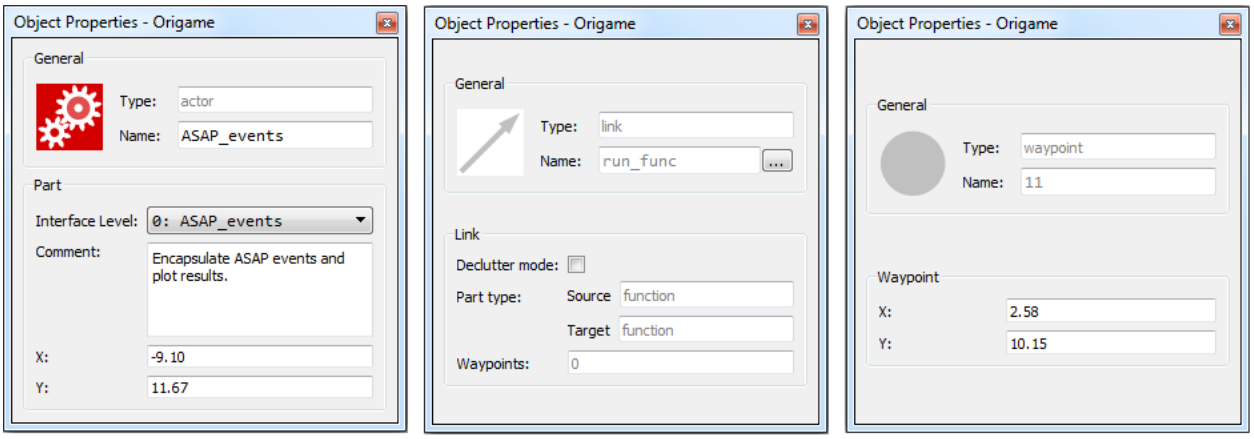


Fig. 3.13: The Object Properties panel showing (left-to-right) part, link, and waypoint properties.

Refer to *Main Simulations* for more information on using the Main Simulation Control panel to run main simulations and set main simulation settings.

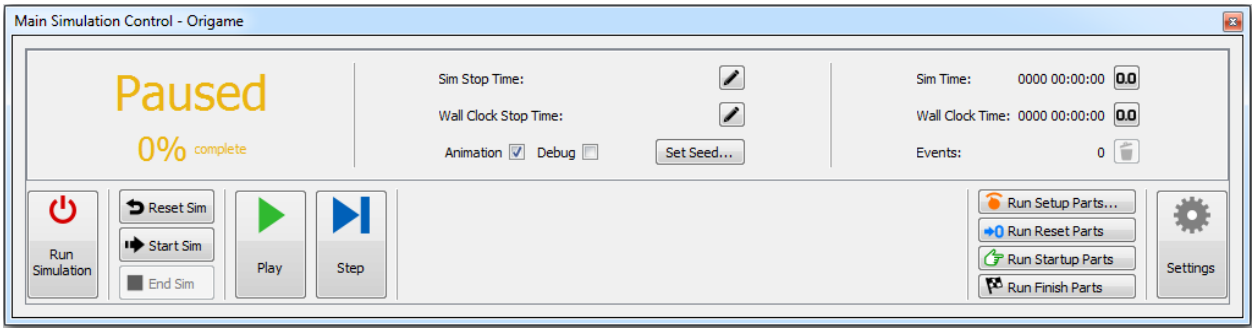


Fig. 3.14: The Main Simulation Control and Status panel.

### Batch Simulation Control

Figure 3.15 shows the Batch Simulation Control tab with both status and control components and some common batch simulation settings. The batch status displays the current batch simulation state, *Ready*, *Running*, or *Completed*, unless the simulation was aborted in which case the state will transition to *Aborted*. It will also display the percent complete, the average time per replication, the estimated time remaining, and the number of finished, running, and failed replications and variants. The number of variants, replications per variant, and number of computer cores to use may also be set from this panel.

On the lower left of the batch control panel are controls to *Run Batch Simulation* and *Pause* (*Play* when paused) the simulation are available. Additional options to *Abort* or set up a *New* batch simulation become available depending on the batch simulation state. The *Settings* button in the lower right of the panel opens the settings dialog where various batch simulation settings can be specified.

Refer to *Batch Simulations* for more information on using the Batch Simulation Control panel to run batch simulations and set batch simulation settings.

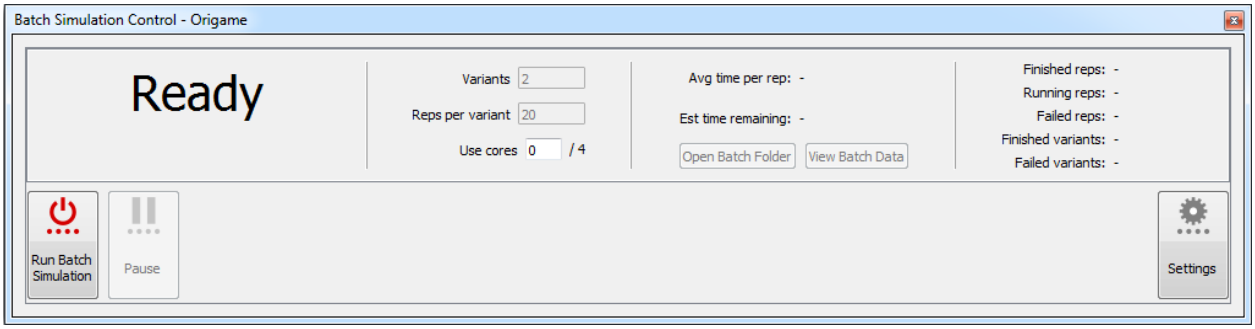


Fig. 3.15: The Batch Simulation Control and Status panel.

### Application Status

Located in the bar across the bottom of the application window, the Application Status bar, shown in Figure 3.16, displays information for the main simulation that includes the simulation state, time, and number of events, as well as for the batch simulation including simulation state, and the number of completed simulations out of the total number of simulations. The status bar also displays a business indicator bar that shows the progress of scenario processes such as loading and and saving, importing and exporting, and scenario searching.


Fig. 3.16: The Application Status bar.

### Toolbar

The toolbar is located directly under the application menu bar. As shown in Figure 3.17, it contains quick links to some of the more common options including creating a new scenario, opening a scenario, saving options, simulation options, and application help.

Fig. 3.17: The toolbar.

## 3.4.5 Navigation

Once a scenario is open, the model can be navigated in several ways. Since a scenario may include a hierarchy of Actor parts, the primary way to navigate the model should be through use of the *Scenario Browser*. This panel displays all levels of the hierarchy of parent-child actors in the model. Clicking an actor part in this panel will display the child actors contents. The actor hierarchy can be searched by using the search box located under the *Actor Hierarchy panel* to find parts or part contents in the model. For large models or models that contain large amounts of table data, the search process can take time to complete. The *Search* button changes to a *Cancel* button while the search is performed which can be pressed at any time to stop the search. Each line of the search results display a path from the child actor in the root, down to the searched-for part or the part containing the searched-for content. Double-clicking a part in the search results will display the contents of that part in the *Model View*.

Alternatively, from the current *Model View*, any child actor in that view can be chosen to display its contents by clicking the *Go Into Actor* (down arrow) button in the top-right of the actor's frame. Navigation back up to its parent

is accomplished by clicking the *Go to Parent Actor* button located on the Parent Actor Proxy widget at the center of the *Model View* and also by pressing the same button on the top-left of the *Model View's* title bar, as shown in Figure 3.18. This bar also provides *Forward* and *Backward* navigation buttons to shift through the actor hierarchy in order of the actors previously viewed.

To navigate around the *Model View*, the view can be panned and zoomed as needed. To pan the view, left-click and hold on an area of blank canvas while dragging the view horizontally or vertically. Scrolling the mouse wheel will pan the view vertically, while pressing and holding the *Shift* key while scrolling the wheel will pan horizontally (up for left, down for right). Pressing and holding the *Ctrl* key while scrolling the mouse wheel will zoom in (scroll up) or out (scroll down).


Fig. 3.18: The *Model View* title bar provides forward, backward, and 'Go to Parent Actor' navigation options.

## 3.5 Building Models

### 3.5.1 Creating Parts

Parts are created by right-clicking on a blank section of the *Model View* and selecting from the context menu,

*New Part > part*

where *part* is one of 17 part types. As discussed in *Non-Creatable Parts*, the deprecated *Clock part* cannot be created but can be loaded from, and copied within, legacy scenarios.

Once a part has been created, right-clicking on it opens a part context menu. All parts that have a visible frame have the following context-menu options:

- Toggle Contents: toggles the part between showing the *full* and the *minimal* Detail Level.
- Edit...: open the part's editor.
- Comment: toggle the visibility of the part's comment bubble.
- Cut: cut the part.
- Copy: copy the part.
- Delete: delete the part.
- Help: open the User Manual Help section for the part.

An additional *Create Link* context option is available from the following parts: *Button*, *Function*, *Library*, *SQL*, *Info*, *Hub*, *Multiplier*, *Node*, *Plot*, and *Pulse*.

The *Actor*, *Function*, *Library*, *SQL*, *Plot*, *Sheet*, and *Table*, include additional context menu options that are specific to their respective part type and are described in section *Part Reference*.

The *Hub*, *Multiplier*, and *Node* are used to route *Part* and *Frame* information between parts that are not linked directly to each other. The context menus of these parts do not contain the *Toggle Contents* or *Edit...* options.

A part or group of parts can be removed by *selecting* them and either pressing the *Delete* key or choosing the *Delete* option from the context-menu. When deleting using the *Delete* key, selected parts that are currently displayed in the view port of the *Model View* will be removed immediately, however parts outside of the viewport will prompt the user before they are removed.

A description of each part, including how it can be used and the options available in its editor, is included in the *Part Reference*.

### 3.5.2 Linking Parts

Only the following parts can create links from themselves to other parts: *Button*, *Function*, *Library*, *SQL*, *Info*, *Hub*, *Multiplier*, *Node*, *Plot*, and *Pulse*. However, all parts may be the target part of a link and therefore all parts can be linked together.

The following set of figures (Figure 3.19 - Figure 3.24) overview the link creation process by showing an example of linking a pulse interface port to a function part called *ASAP_3*.

Step 0: Start with two un-linked parts shown in Figure 3.19.


Fig. 3.19: Two un-linked parts.

Step 1: Link creation is started in one of two ways:

1. By activating the link creation shortcut on the parts upper right frame by moving the mouse near or over the part (see Figure 3.20). The shortcut appears as a grey arrow with a plus sign.
2. By accessing the part's context-menu by right-clicking the source part and selecting *Create Link*.


Fig. 3.20: Activate link creation by moving the cursor to the pulse interface port.

Step 2: Next a *target* part can be selected. As the mouse moves across the *Model View*, a temporary link will track the mouse cursor. Invalid targets will be indicated as shown in Figure 3.21.


Fig. 3.21: Invalid targets include the part creating the link (the source part), target parts already connected to the source part, and in the case of interface ports, other ports on the same parent actor (as shown here).

Step 3: Waypoints can be created during link creation by simply clicking blank canvas locations as shown in Figure 3.22. The new waypoint will be set and link creation will continue.


Fig. 3.22: Waypoints can be created during link creation.

Step 4: A link target marker will appear at the mouse cursor to indicate when the link connection is valid by hovering the mouse over a potential target as shown in Figure 3.23.


Fig. 3.23: The green link target marker appears over a valid target indicating the link can be created.

Step 5: Complete link creation by left-clicking the part. A new link is created between the parts as shown in Figure 3.24.


Fig. 3.24: New link creation complete.

Link creation can be cancelled at any time be pressing the *Esc* key or by right-clicking the mouse on blank canvas. All parts can have an unlimited number of outgoing and incoming links with the exception of the Node part which can have only **one** outgoing link.

Right-clicking a link opens the link context-menu that contains the following options:

- Rename: change the link name.
- Delete: remove the link.
- Toggle Declutter: toggle link declutter on or off.
- Go to...: opens a submenu with options for *Source Part*, *Target Part*, *Source Port*, *Target Port*.

- Insert Waypoint: adds a waypoint to the link.
- Remove All Waypoints: removes all the waypoints of this link.
- Change Link Endpoint: allows the target part to be changed.

The link name can be changed from the link's context-menu by right-clicking the link and selecting *Rename* or by double-clicking the link. The *Rename Link* dialog will open, as shown in Figure 3.25, to facilitate link renaming. The dialog displays the current name and, if the link name is referenced anywhere in the model, shows a list of parts that reference the link name in their scripts along with the actual line in the script where it is used. The new name is entered at the bottom of the dialog. The dialog automatically checks that the link name is a valid Python name and that the name is unique (each source part's outgoing link names must all be unique). If either of these conditions fails, the new name field will remain highlighted in red and the *OK* button will be disabled. Once a valid name is entered, the highlight is removed and the *OK* button is enabled. Pressing the *OK* button on the dialog will change the name of the link and all script references found in the model.

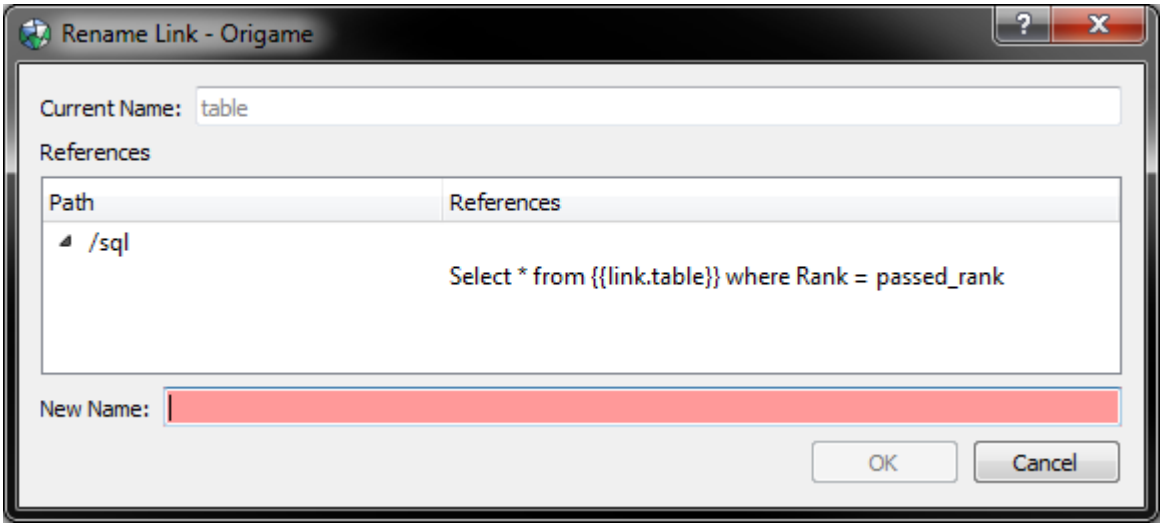


Fig. 3.25: Rename Link Dialog.

A link can be deleted by either clicking the option in link's context-menu, or by selecting it and pressing the *Delete* key, or by holding the *Ctrl-Shift* keys and clicking the link to delete.

The *Toggle Declutter* option decreases model clutter by hiding the middle section of the link. Only the start and end portion of the link will appear between two connected parts. By re-selecting the option from the context-menu the full link will be displayed.

The *Go to...* options allows navigation to the links source or target part. If the link is connecting a port or ports, then the source or target port options or both will also be available. After selecting the desired part or port, the Model View will pan to the correspoinding object.

Waypoints can be added or removed from links by selecting the *Insert Waypoint* or *Remove Waypoint* options, respectively. Each waypoint can be independently positioned allowing links to be drawn around other parts and model elements. See section *Waypoints* for more information. Selecting *Remove All Waypoints* removes all the waypoints on the selected link in one shot. Individual waypoints can be removed by accessing a waypoint's context-menu option by right-clicking it, or by selecting it and pressing the *Delete* key, or by holding the *Ctrl-Shift* keys and clicking the waypoint to delete. A group of waypoints may be removed at once by clicking the *Delete* key after selecting a group of waypoints by either using the "rubber-band" selector (hold Shift while holding the left mouse button and dragging the mouse) over a group of waypoints or by individually selecting each waypoint while holding the *Ctrl* key. The "rubber-band" must contact a waypoint first rather than a part in order for waypoints to be selected. Waypoints from several different links may be removed at once in this manner. When selected, waypoints display their X and

Y position properties in the *Object Properties panel*. Changing the X and Y coordinates in the panel will update the waypoint position in the *Model View*.

The target part of a link can be changed by selecting the *Change Link Endpoint* option from the links context-menu. Once activated, a new temporary link is drawn from the source part or the last waypoint (if any), and will track the mouse cursor while the target marker icon indicates whether the new target is valid or invalid as before. By clicking on the new valid part the new permanent link connection will be made. This operation can be cancelled by pressing the *Esc* key or by right-clicking the mouse on blank canvas either of which will restore the original link.

### 3.5.3 Linking Interface Parts

As discussed previously, *Actor parts* are used to encapsulate related implementation details of a model into a hierarchy of actors. In many cases, however, it is necessary to be able to link two parts together that are children of two different parent actors. In order to create these types of links it is necessary to make these child parts into *interface parts* by increasing the interface level of each part to the common actor part ancestor they share. After this is done, each part appears as a port on the boundary of each ancestor actor of the hierarchy up to the interface level set (the common ancestor) where the ports of each part can be linked together thereby linking the two parts.

The interface level can be set by:

1. Using the drop down menu of the *Object Properties panel*: this will be the only way to set the interface level of the part if it is at the default level of 0.
2. Right-clicking the interface port appearing on the boundary of the actor part, choosing Set Interface Level, and selecting the desired level. The Set Interface Level menu will show the interface levels and the actor name at each level, with level 0 having the part's name. The current level will be disabled, and the context level will be highlighted.

Once the interface level is increased, an interface marker bar appears above the part's widget as shown on the left side of Figure 3.26. The interface marker bar indicates the interface level, and the number of elevated incoming and outgoing links that are not visible locally. The parent actor of the interface part is shown on the right side of the figure. It shows that the part is now exposed to the surroundings of its parent actor via an interface port on its parent's left side (interface ports alternate left to right). Ports appearing with an yellow *up marker* bar indicate that they are exposed on the boundary of the currently viewed actor as well. Changing the part's interface level allows it to be linked to its grandparent, to/from siblings of its parent, and to/from interface parts of sibling actors of its parent actor.

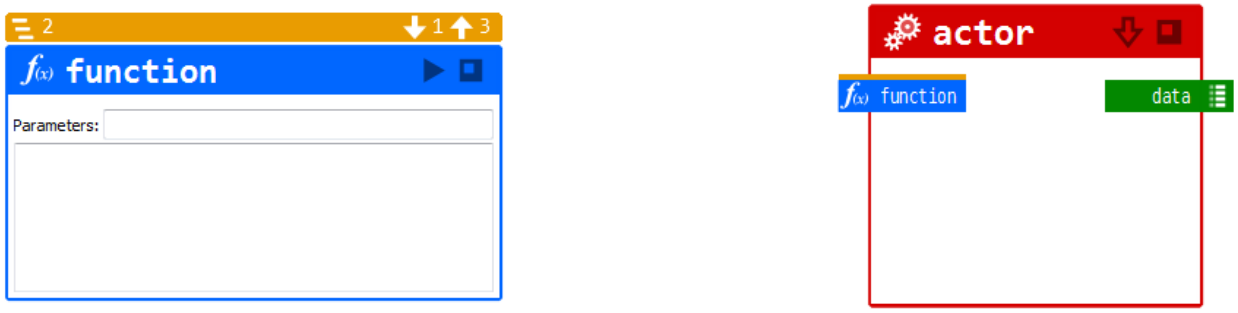


Fig. 3.26: Left: A function part with an interface marker bar. Right: The parent actor with the function interface port. The yellow *up marker* bar on the function port indicates it's also exposed on the boundary of this actor's parent.

The context menu of an interface port supports several operations:

- Set Interface Level: use the submenu to select the interface level of the part,
- Go to Ifx Port: use the submenu to navigate to an interface port at a different interface level,

- Go to Part: navigate to and select the assoicated part in the *Model View*,
- Edit: opens editor for the corresponding part,
- Switch Side: change the side of the actor on which the port is shown,
- Move Up: move the port up along the side of the actor; the port above is moved down,
- Move Down: move the port down along the side of the actor; the port below is moved up.

Valid links are links for which the source and target parts have a common ancestor that is within reach of their interface level. Example: consider the scenario shown in Figure 3.27 with parts A, B, C, D, and E. A link from root.A.B.C to root.A.D.E: the two parts have root.A as common ancestor, which is reachable by C and E if each has interface level 1 (C is exposed as a port on B and E is exposed as a port on D, both of which have A as parent). If this is not the case, such link is invalid. To make such a link valid, it is sufficient to increase the interface level of C or E or both to 1.

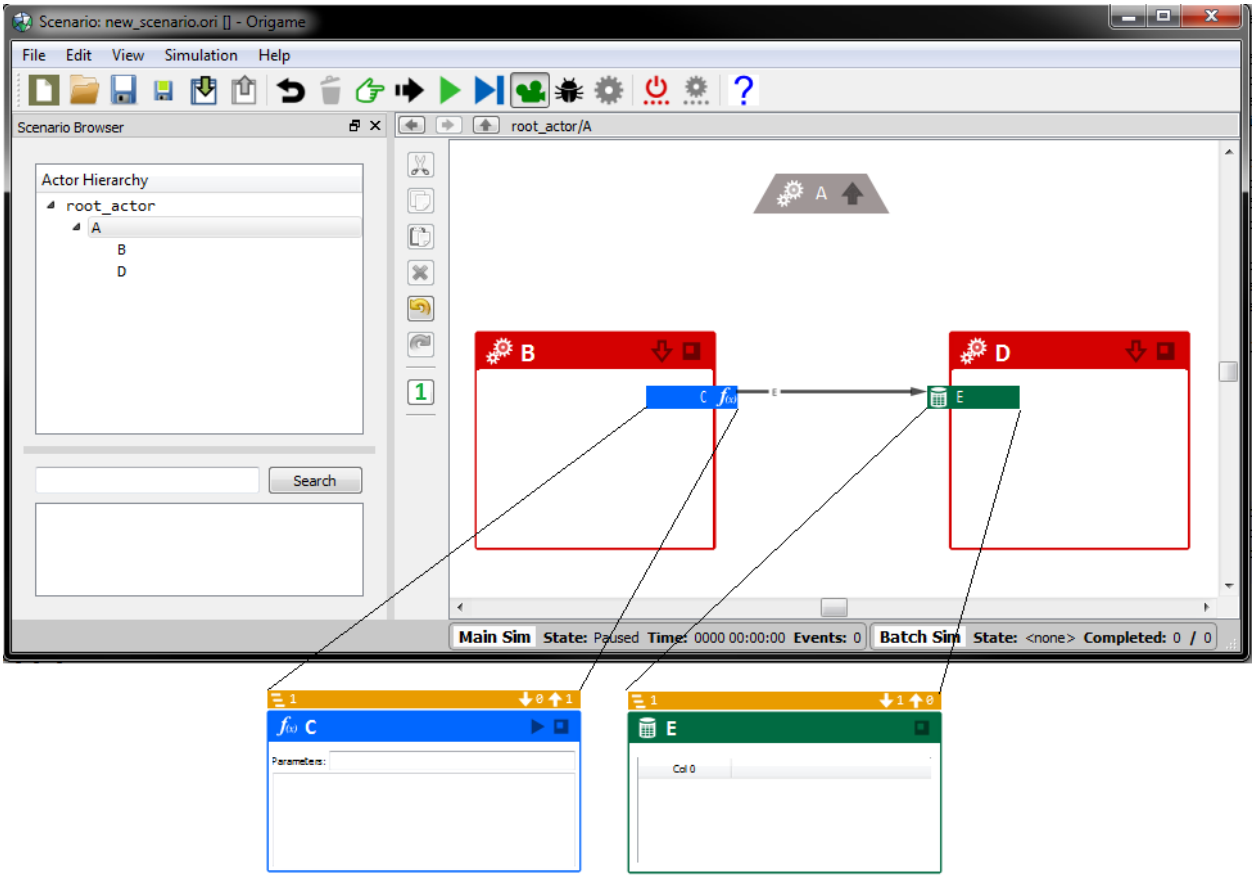


Fig. 3.27: Part C and E can be linked if each parts' interface level is increased to 1.

If a part is moved (e.g. during a cut and paste) but the resulting links are still valid, then they are automatically adjusted to maintain the connections between the two parts. However, if the part is cut and paste such that invalid links result (because of insufficient interface level on either end of the link), Origame provides a link management dialog to automatically adjust the links as the user requires. A *Move Part* dialog will open that prompts the user to decide whether to

1. Break the links since they are now invalid. The part will be pasted in the new location without any links.
2. Adjust the interface levels of the link sources and targets to keep the links valid. The part is pasted with a new interface level and all links intact.

3. Cancel the operation.

Copying a part will create a part with the default interface level (0).

### 3.5.4 Part Selection and Position

Parts and waypoints can be selected individually or as a group in a multiple-selection. Links can only be single-selected. However, a group of selected parts that are linked will be copied, cut, and pasted with the links between the parts in the selection intact.

Single part selection occurs when a single part is clicked with the mouse (left or right mouse button). Multiple parts can be selected by left-clicking each one in turn while holding down the *Ctrl* key or by invoking a "rubber band" selection box by holding the *Shift* key and left mouse button while dragging the mouse cursor diagonally across a group of parts. A green highlight around each part will indicate it is selected. Parts can be moved around the *Model View* by clicking and dragging the part to a new location. The position coordinates of an individually selected part are displayed in the *Object Properties panel*. Note that these coordinates correspond with the top-left corner of the part.

Groups of waypoints can also be selected and moved by using either *Ctrl*-click or the "rubber band" selection methods described above. However, note that groups of parts and waypoints cannot be selected together. When using either rubber band or the *Ctrl* key to select groups of items, the first item selected determines if this will be a group selection of parts or waypoints. If a part is selected first, the group selection will only allow additional parts to be selected for example. Similarly, selecting a waypoint first will permit only other waypoints to be included in the selection. To switch between selecting parts or selecting waypoints all items in the current selection must be deselected, for example, by clicking a blank point in the *Model View*.

### 3.5.5 Part Resizing

The selection highlight of framed parts also provides resizing functionality by displaying three dark green resize "grip" on the right edge, bottom edge, and lower right corner that are used to resize the part in three possible directions that include horizontal, vertical, and diagonal directions, respectively with respect to the part's source at the top-left corner of the part; see Figure 3.28. Moving the mouse over a resize grip will cause the cursor to display an arrow icon indicating the allowed direction of resizing. By clicking (and holding) one of these grips, the part can be resized by dragging the border to a new position.

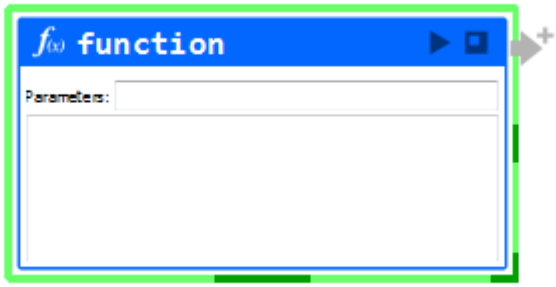


Fig. 3.28: Selected part showing horizontal, vertical, and diagonal resize grips.

### 3.5.6 Part Indicators and Markers

Several indicators and markers, shown in Figure 3.30 and Figure 3.31, are used on the parts to display special part roles and flags either during part configuration or while running a simulation. These include:

- Setup marker: appears as an orange gauge on the top-left of the part (see Figure 3.30) and flags the part as a Setup part. Parts with this role are only run when the user clicks the "Run Setup Parts..." button in the *Main Simulation Control panel*. Clicking the button opens the dialog shown in Figure 3.29). The user may enter values manually, load them via the *Load...* button or save argments to load later via the *Save...* button. Files are saved with the extension '.pca' (Part Call Arguments).

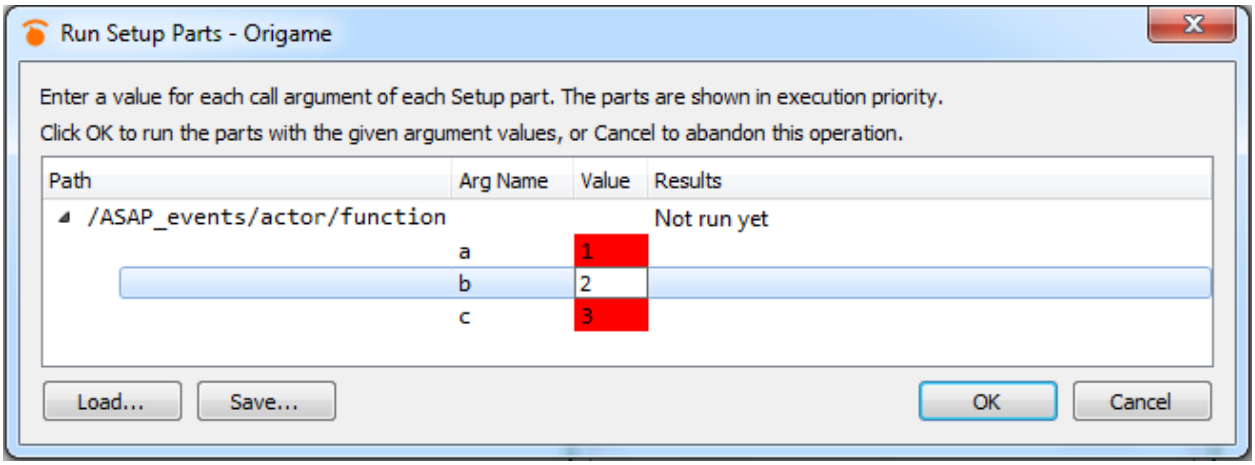


Fig. 3.29: The *Run Setup Parts* dialog allows the user to set values for each call argument to the *setup* function part.

- Startup marker: appears as a green hand pointing to the right on the top-left of the part (see Figure 3.30) and flags the part as a Startup part which runs the part immediately once the simulation has been started. Clicking the "Run Startup Parts..." button in the *Main Simulation Control panel* will also run parts with this role.
- Reset marker: appears as a blue arrow pointing at a zero on the top-left of the part (see Figure 3.30) and flags the part as a Reset part which runs the part when the simulation is reset. Clicking the "Run Reset Parts" button in the *Main Simulation Control panel* will also run parts with this role.
- Finish marker: appears as a checkered flag on the top-left of the part (see Figure 3.30) and flags the part as a Finish part which runs the part when the simulation finishes. Clicking the "Run Finish Parts" button in the *Main Simulation Control panel* will also run parts with this role.
- Breakpoint marker: appears as a red stop sign icon on the top-left of the part (see Figure 3.30) if a breakpoint has been set in the part's script (executable parts only).
- Comment indicator: appears as a blue comment bubble on the top-left of the part (see Figure 3.30) if a comment has been set in the *Object Properties panel*. Clicking the icon displays the comment above the part in a comment box.
- Warning indicator: appears as an orange triangle and exclamation point on the bottom-left of the part (see Figure 3.30) when a warning occurs running an executable part. If a part has errors and warnings, only the error indicator will be shown.
- Error indicator: appears as an red triangle and exclamation point on the bottom-left of the part when an error occurs running an executable part. If a part has errors and warnings, only the error indicator will be shown.
- Event indicator: appears as a tag on the left side of the part (see Figure 3.31) indicating the part has an associated event waiting on the *Event Queue*. The number on the indicator signifies the number of events associated with the part. The indicator also appears on all parent actors of the part. Double-clicking on an event indicator will

filter the *Event Queue* to show only the events associated with that part. To remove the filter from the queue press *Clear Filter* to display all the remaining events in the simulation. The indicator is color coded as follows:

- Green signifies that the part's event is the next event on the *Event Queue*. There can only be one part with this indicator.
- Amber signifies that the part's event is scheduled to execute concurrently with the next event, but which will execute after the next event according to the sorting of the *Event Queue*.
- Black signifies that the part's events are waiting to execute at a time later than the next event.

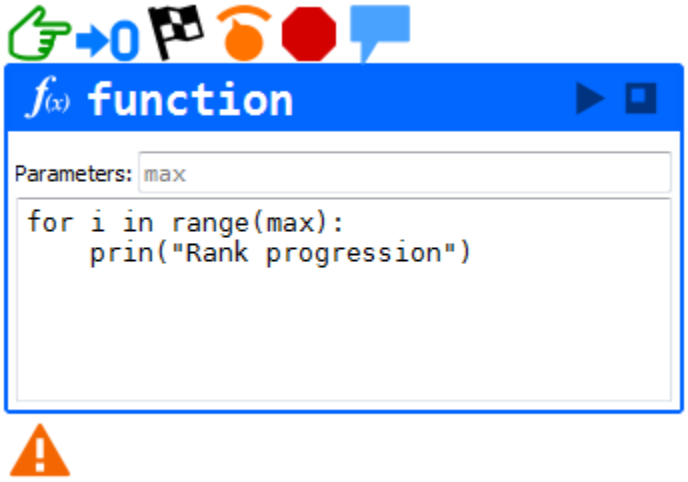


Fig. 3.30: Function part displaying (left-to-right) start-up, reset, breakpoint, and comment markers and indicators on top and warning indicator on bottom (an error indicator is a red version of the warning indicator).

### 3.5.7 Part Editing, and Undo, Redo Commands

Right-clicking a single part will select it as well as open the context menu displaying common and part-specific options including cut, copy, and delete. Right-clicking anywhere in the *Model View* with multiple parts selected opens a context menu with the same editing options that will be applied to all selected parts.

Editing options to cut, copy, paste, and delete parts are available from the following locations:

1. Application Edit menu,
2. Part context menus,
3. *Scenario Browser panel* context menu,
4. *Model View* shortcut buttons located on the left of the panel,
5. Keyboard shortcut keys.

Undo and redo options are also available from some of these locations. The options to cut, copy, and delete are only available when one or more parts are selected, otherwise, they are disabled and greyed out. Paste becomes available once a part has been added to the clipboard by a copy or paste command. Undo becomes available once the scenario being displayed is changed. Redo becomes available once undo has been invoked.

Selected parts can be removed from the scenario by pressing the *Delete* key, or by pressing 'Ctrl + X' (cut), or selecting one of the delete or cut options discussed above. When deleting using the *Delete* key, selected parts that are currently displayed in the view port of the *Model View* will be removed immediately, except for parts outside of the viewport which will prompt the user before they are removed.. Additionally, the *Scenario Browser panel* can be used to add,

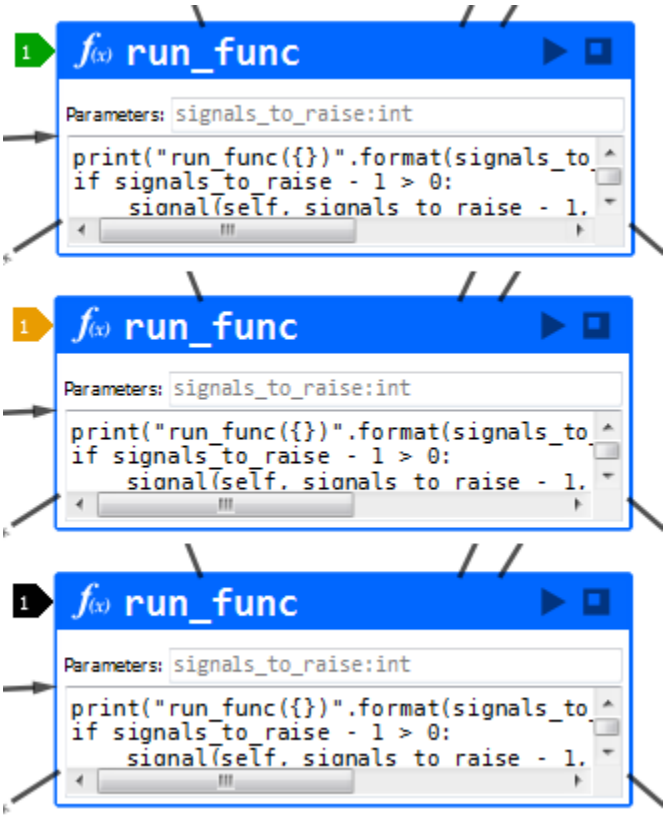


Fig. 3.31: Function parts displaying green, amber, and black event indicators each with one event in the queue.

delete, and rename actor parts by right-clicking an actor in the panel and selecting the corresponding option from the context menu. With an actor selected, pressing the *Delete* key or invoking the context menu and selecting *Delete* will remove the actor from the scenario. When an actor part is selected in the *Scenario Browser panel* by either a right or left-click, the *Model View* automatically displays the contents of the selected actor. Upon removal, the *Model View* will revert to the parent actor's contents.

Removed parts can be restored by invoking the Undo command through either the shortcut buttons on the left of the *Model View*, the application's Edit menu, or the shortcut keys 'Ctrl + Z'. If the *Model View* has been panned to a different location or the scenario navigated to view a different actor after the part was removed, the first 'undo' will navigate back to the location where the part was deleted. Undo must be invoked a second time to restore the part. Conversely, invoking Redo through any of the previously mentioned locations, or pressing the shortcut keys 'Ctrl + Y', will remove the part again.

Pasted copies of a part or selection of parts (whether copied or cut) occur at the current mouse position if it's located on the *Model View*. The paste aligns the center of the part or selection of parts with the mouse position. This allows the user to select the specific paste location in the scenario. If a subsequent paste occurs at the same mouse location (i.e. the mouse has not moved between two consecutive pastes), an offset is applied so that the pasted parts do not completely overlap and hide the original copies. If the mouse has been moved off the *Model View* when a paste occurs, the parts will be pasted at the center of the visible view and an offset applied for each subsequent paste. For context-menu paste operations, the parts are pasted where the mouse right-clicked the *Model View* to invoke the context menu.

Parts that are cut or copied may be pasted into a new or opened scenario. If parts have been placed on the application's clipboard, then initiating a *New* or *Open* scenario action will cause a prompt to display the first ten items on the clipboard (and indicate how many additional parts are also on the clipboard) and ask the user if the contents should be made available in the next scenario. If *Yes* is selected, the cut or copied parts may be pasted once the next scenario has been loaded. Selecting *No* will result in a fresh clipboard with no parts in it. *Cancel* can be selected to stop the *New* or *Open* scenario action.

All parts except the hub, multiplier, and node have a part editor that can be used to edit part properties and set part functionality. The part editor can be opened by either double-clicking the part or right-clicking and selecting *Edit...* from the context menu. The common components of all part editors are shown in Figure 3.32 and include:

- Title bar: the icon, part name, and part type,
- Top section: a read-only path to the part in the model, an editable field for the part name, and a "Go to Part" button that will navigate the Model View to the location of the associated part,
- Middle section: an area that displays the specific editable content of the part (see links below to individual part sections),
- Bottom section: a part *Help* button, an *OK* button that applies changes and closes the dialog, and *Cancel* button that discards any changes and closes the dialog (equivalent to the *X* button in the top-right of the dialog), and an *Apply* button that applies the changes made so far and leaves the dialog open to make further changes.

For information on the middle section of each individual part editor, see the part reference section for the part: *Actor*, *Button*, *Clock*, *Data*, *Datetime*, *Function*, *Info*, *Library*, *Plot*, *Pulse*, *Sheet*, *SQL*, *Table*, *Time*, and *Variable*.

If the user tries to exit Origame with part editors open with unsaved changes, a dialog will open that requires user confirmation (by pressing *Yes*) to discard all changes before proceeding to exit, else (by pressing *No*) to stop the exit operation and allow the user to determine which individual editor panel changes to apply and which to discard.

Note that changes made to part content in the editor panel are processed as a single bulk change after each *OK* or *Apply* button press. Therefore, Undo and Redo operations work on each set of bulk changes. For example, if the three changes were made and *Apply* was clicked after each change, then each Undo and Redo command will work on each change individually. However, if *Apply* was pressed after all changes were made, Undo and Redo work on all changes at once.

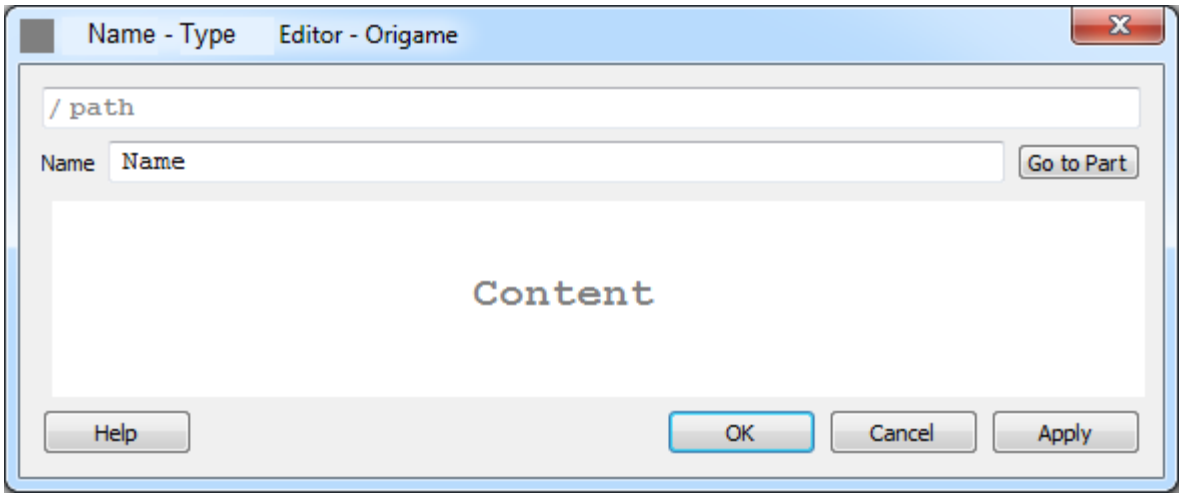


Fig. 3.32: The common components of all part editors. The "Content" label shown in the middle section indicates where the part-specific editable content would be displayed.

### 3.5.8 Script-Base Parts: Links, Symbols, and Imports

The *Function*, *Library*, *Plot*, and *SQL* parts are all script-based i.e. an embedded script can be opened in the part editor and modified so that when run, the part will generate new events, call other parts, define new functions and methods, plot simulation data, and query tables. The editor panels of these parts are discussed in their respective sections. This section covers common elements of these scripted-part editors: the Link, Symbol, and Import tabs.

Double-clicking on the any of these parts' frames or right-clicking and selecting "Edit..." opens a script editor dialog (see *Part Editing, and Undo, Redo Commands*) that displays tabs for *Links*, *Symbols*, and with the exception of the SQL part, *Imports* panels. These panels are shown in Figure 3.33.

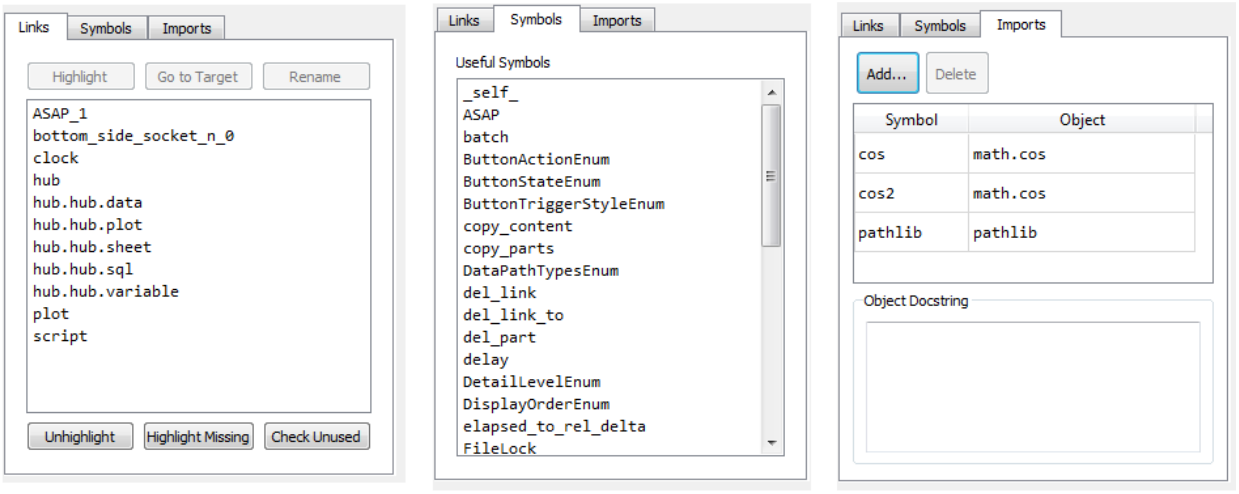


Fig. 3.33: The Links, Symbols, and Imports tabs of the script-based part editor panel.

The *Links* tab in the script editor provides various link management options that are activated after selecting a link in the list. These options include:

- *Highlight*: will highlight any usages of the link in the script in green.

- *Unhighlight*: removes the highlighted usages in the script.
- *Go to Target*: pans the Model View to center on the target part to which the link is connected.
- *Highlight Missing*: highlights script references to links that are not in the list (and therefore not in the model) in red.
- *Rename*: renames the link in the model and script (when *Ok* or *Apply* are pressed). Link *chains*, i.e. those links that are connected via multiple hub parts to their target part, cannot be renamed from the script editor.
- *Check Unused*: indicates all links in the list that are not used by the script by displaying them in a faded font.

The *Symbols* tab displays useful symbols and Python modules available to insert into the script. Clicking a symbol will display its documentation at the bottom of the editor while double-clicking it will insert it into the script body at the cursor location. The symbols displayed in the list depend on the type of part being edited.

- Function and Library parts: display the broadest range to facilitate model building and data processing and include *Path*, *batch*, *new_part*, *new_link*, *signal*, and more.
- Plot parts: contain plot-specific methods such as *figure*, *math*, *matplotlib*, *setup_axes*, and others.
- SQL parts: display two sets of lists, one for constructing SQL commands and another for Python code. The former list is displayed by default and includes *CREATE*, *DELETE*, *DROP*, *SELECT*, *WHERE* and others. To access Python-based symbols, place the cursor between a double set of curly braces, "{{}}", and the symbols list will change to show symbols specific to the Python editing context.

The *Imports* tab displays a table of symbols and objects that have been imported into the script and fasciliates importing new symbols and objects. The object is the Python entity that is imported into memory when the script is compiled, while the symbol is the alias used to reference the object in the script. When importing a new object, four cases are possible:

1. Module import: corresponds to an "import module" statement in a Python script (e.g. "import math"),
2. Module import as new symbol name: corresponds to an "import module as module_alias" statement in a Python script (e.g. "import math as math"),
3. Symbol import from module: corresponds to a "from module import symbol" statement in a Python script (e.g. "from math import cos"),
4. Symbol import from module, as new symbol name: corresponds to a "from module import symbol as symbol_alias" statement in a Python script (e.g. "from math import my_cos").

To import a new object,

- Press the *Add...* button to launch the *Import an Object* dialog shown in Figure 3.34.
- Select the module to import in the "Source Module" field.
    1. For case 1, press the dialog *OK* can be pressed to import the module as its current name.
    2. Otherwise, for case 2, enter a new symbol name to represent the module in the "As Symbol name" field. Press *OK*.
- Select an attribute to import from the module by checking "Attribute Name" and then selecting the attribute name from the drop down list. This list is automatically populated once a valid object name is entered.
    3. For case 3, the attribute name will be imported as its current name by clicking the dialog's *OK* button.
    4. Otherwise, for case 4, enter a new symbol name to represent the original symbol in the "As Symbol name" field. Press *OK*.
- The imported symbol can now be referenced in the script.

An import can be removed by selecting it in the table and clicking the *Delete* button. Clicking a symbol will display its documentation at the bottom of the tab, while double-clicking it will insert it into the script body at the cursor location.

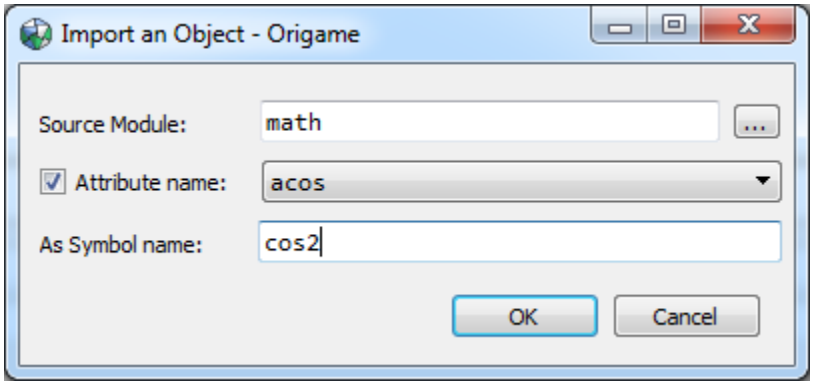


Fig. 3.34: The Import an Object dialog allows objects to be imported and their symbol specified.

### 3.5.9 Creating, Editing and Triggering Events

The *Event Queue*, shown in Figure 3.10, displays all events that are queued to execute at a specified time and with a specified priority within that time. Time can be any time in the future. The priority ranks from 0.0 (lowest) to *ASAP* (highest). This panel also displays the part that the event corresponds to, the part's type, arguments, and path in the Actor Hierarchy. Once there are events listed in the panel, the options to *Edit*, *Delete*, and *Clear* become available in the top-left of the panel. An event can be edited by double-clicking it in the Queue, or selecting it and pressing the Edit button. A dialog will open that will allow the events time, priority, and arguments to be changed. *Delete* an event by selecting it and clicking the "X" button. *Clear* all events by pressing the garbage bin button.

*Function parts* can be used to place events on the *Event Queue* programmatically (see Figure 3.10) by using the global *Signal* function in the function part script. An event is created on the queue by specifying any function *parameters* required to execute the function part's script, and scheduled according to its *date-time* and *priority* parameters. The data-time specifies a future calendar date (year-month-day, hour:minute:second) while the priority indicates the importance of the event relative to other events occurring at the same time. The priority is specified using real numbers with 0.0 being the lowest priority and all higher values indicating a higher priority (up to a maximum of 1,000,000.0). Specifying an event priority of *ASAP* will cause the event to be processed immediately and no date-time parameter is required.

If a function part is linked to another part, its script has access to that part using the *alias* specified by the link. In other words, aliases of the links that connect a function part to other parts constitute the namespace of the function part. Section *Programming Reference* describes the *API* and attributes for each part type which a function may use to access a target part's contents or properties. To run a Function part, click the run button in the upper-right corner of the part's frame in the *Model View* or select the *Run* option from its context menu. The context menu of the Function part has the following options:

- *Roles...*: set the role of this part to any combination of *Startup*, *Reset*, *Finish*, and *Setup* roles.
- *Add Event...*: opens a dialog that is used to add a new event to the Event Queue by setting function arguments (used to set function parameters), event time, and event priority.
- *Run Debug*: runs the part in debug mode which causes the process to stop at breakpoints inserted at the script line numbers.
- *Run*: runs the part.

Along with the *Function part*, the other executable parts include the *SQL part*, *Pulse part*, and *Multiplier part*. These parts can be signaled by other executable parts to define discrete simulation events that appear on the simulation *Event Queue*. *Function* and *SQL* parts contain an editable Python script that is used to signal parts or define discrete events. They can also define parameters that are used to pass information to the script. The script may be written to return a value.

Multiplier parts provide a one-to-many interface for forwarding signals. For example, a multiplier that is connected to a Function or *SQL* part, can signal any number of other parts connected to the multiplier when the Function or *SQL* part signal the multiplier. The multiplier forwards that signal to each part connected to itself.

Pulse parts that are connected to other executable parts will place those connected parts on the *Event Queue* periodically according to the pulse period that has been defined. The sequence works as follows: when the simulation is running, the pulse will place itself on the *Event Queue* at regular intervals corresponding to its pulse period. When the simulation time matches the pulse time, it will be popped from the queue and signal its connected part or parts to create a new event at *ASAP* priority. The new *ASAP* event or events are added to the queue along with the next pulse event corresponding to the next pulse time.

CHAPTER

FOUR

# RUNNING SIMULATIONS

A scenario can be run in the *GUI* variant in either a *Main* simulation where a single simulation is performed, or in a *Batch* simulation where multiple simulations are performed. A Main simulation can be run in Animated Mode, where the *Model View* may change dynamically during scenario execution, or Non-Animated Mode where the *GUI* remains unchanged. A Batch simulation is always non-animated.

Simulation scenarios can be configured to run in a combination of *variants* and in *replications*. Multiple *variants* of a scenario are configured by running with different scenario parameters while multiple *replications* are configured by running with different seed values. Each combination of variant and replication will produce a unique simulation result. For example, consider a batch simulation of an occupation modelled over a period of 20 years with three different attrition rates. The three attrition rates represent the three "variants" and if each scenario is to be simulated 100 times (i.e. 100 "replications per variant") then 100 seed values are used to generate the statistical outputs for analysis.

If multiple scenario replications or variants are required, a Batch run should be selected. A Main run is only applicable to running a scenario in *GUI* Mode for a single scenario variant/replication combination, with either Animation Mode on or off.

Each simulation run produces a *CSV* log-file automatically named "log.csv" in to the current user's AppData folder: *C:\...\AppData\Local\DRDC\OrigameGui\Logs*. The log file can be saved manually to a different location at any time by pressing the *Save...* button on the *Application Log* panel. Batch simulations produce a log-file for each corresponding run within a subfolder created in the same directory as the batch scenario file. This is described in more detail in the following sections.

The following sections discuss how to run Main and Batch simulations where multiple variants and replications of a scenario are configured.

## 4.1 Main Simulations

A single simulation can be run and controlled by using the Main Simulation controls and settings available from:

1. The *Main Simulation Control panel*,
2. The *application Toolbar*,
3. The *Simulation menu*.

The Main Simulation Control tab is shown in Figure 4.1 and contains both status read-outs and control options.

On the top left of the panel, the simulation status readout will initially be shown as *Paused* with "0% complete". When the simulation is run, the status will change to display the *Running* state. If a stop time has been specified, the percent complete will increment based on the progress of time relative to the stop time set. Two stop times can be set via the Main Simulation Control panel: *Sim Stop Time* and *Wall Clock Stop Time*. Clicking either pencil icons in the top-center of the panel will open a dialog where a stop time can be entered in units of days, hours, minutes, and seconds. If both stop times are set, the percentage complete will be based on the stop time that occurs first. In the case

of simulation stop time, the percentage complete is based on progress of simulation time relative to the simulation stop time; alternately, in the case of wall clock stop time, the percentage is based on the advancement of real-time (i.e. the computer clock) relative to the wall clock stop time. Sim and wall clock time readouts with corresponding reset buttons are located in the top right part of the panel along with a readout for the number of events on the *Event Queue* and a corresponding *Clear Queue* button.

The main control panel also allows *Animation* and *Debug* modes (which are only applicable to "Main" simulations) to be toggled and a random seed to be set for the scenario's random number generator. The latter option opens a dialog where the user-specified seed can be entered. The dialog includes directions for selecting an acceptable seed value and will indicate when an acceptable value is entered by highlighting the input field in green and enabling the *OK* button. Entering an invalid value will disable the *OK* button and highlight the input field in red. The dialog also includes a *Generate* button that will automatically generate a valid seed and a checkbox which allows the Reset Seed to be used instead.

On the lower left of the main control panel are controls to *Run Simulation*, *Play* (*Pause* when running), and *Step* the simulation. The *Run Simulation* button will run all reset parts, run all startup parts, and then run the simulation. Control options to perform these actions individually are available via the *Reset Sim* and *Start Sim* buttons, the first which resets the simulation and wall clock times to zero and runs all the reset parts, and the second which runs all startup parts and starts the simulation. The *End Sim* button runs the finish parts and stops the simulation.

Four buttons on the lower right of the panel are the "run role" buttons where the role refers to the Setup, Reset, Startup, and Finish roles. These pertain to parts with roles and are only enabled when the simulation is paused and when a part with the role exists. Clicking an enabled button will run all parts with the corresponding role.

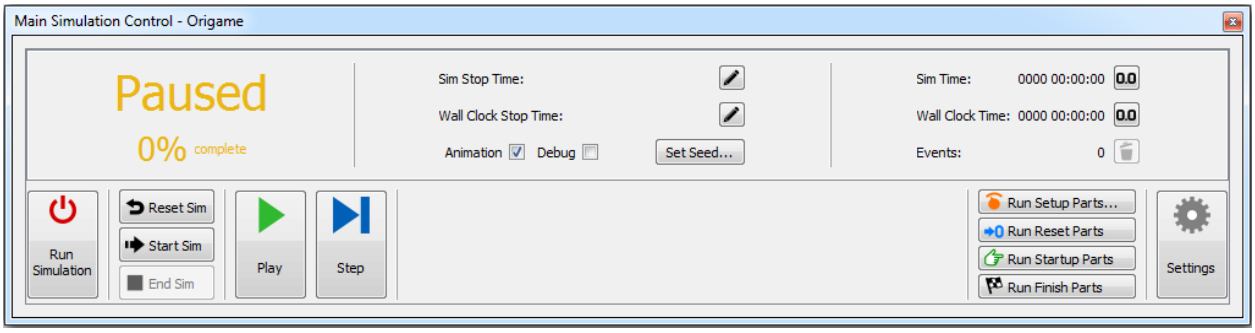


Fig. 4.1: The Main Simulation Control and Status panel.

The following table summarizes the Main Simulation Control panel status read-outs and controls.

| Main Simulation Control | Description |
|---|---|
| Sim Stop Time | Set the maximum simulation time. Simulation will advance to, but not beyond this time. Percent complete status shown based on this time if it occurs first. |
| Wall Clock Stop Time | Set the maximum wall clock time. Simulation will advance to, but not beyond this time. Percent complete status shown based on this time if it occurs first. |
| Animation | Toggles *Animation Mode*. If toggled **off**, the *GUI Model View* and *Event Queue* will appear static. If toggled **on**, then they will change dynamically as discrete events are processed. |
| Debug | Toggles *Debug Mode*. If toggled **off**, the scenario will execute and break- points that have been set will not be triggered. If toggled **on**, simulation execution will pause at each break-point as shown in Figure 4.2. |
| Set Seed | Opens a dialog that allows the user to set a seed for the application's random number generator. |
| Clear Queue Run Simulation | Removes all events from the *Event Queue*. Performs all the following actions: Run reset parts, run startup parts, run simulation. |
| Reset Sim | Reset simulation and wall clock times and run reset parts. |
| Start Sim | Run startup parts and run simulation. |
| End Sim | Run finish parts and pause the simulation. |
| Play / Pause | Toggles run the simulation / pause the simulation. |
| Step | After pausing the simulation, pressing *Step* will advance to the next discrete time step, processing the next event. This option is only available when paused. |
| Run Setup Parts... | Runs all Setup parts. Only available when paused and when Setup parts exist. |
| Run Reset Parts | Runs all Reset parts. Only available when paused and when Reset parts exist. |
| Run Startup Parts | Runs all Startup parts. Only available when paused and when Startup parts exist. |
| Run Finish Parts | Runs all Finish parts. Only available when paused and when Finish parts exist. |
| Settings | Open the Main Simulation settings panel as shown in Figure 4.3. |

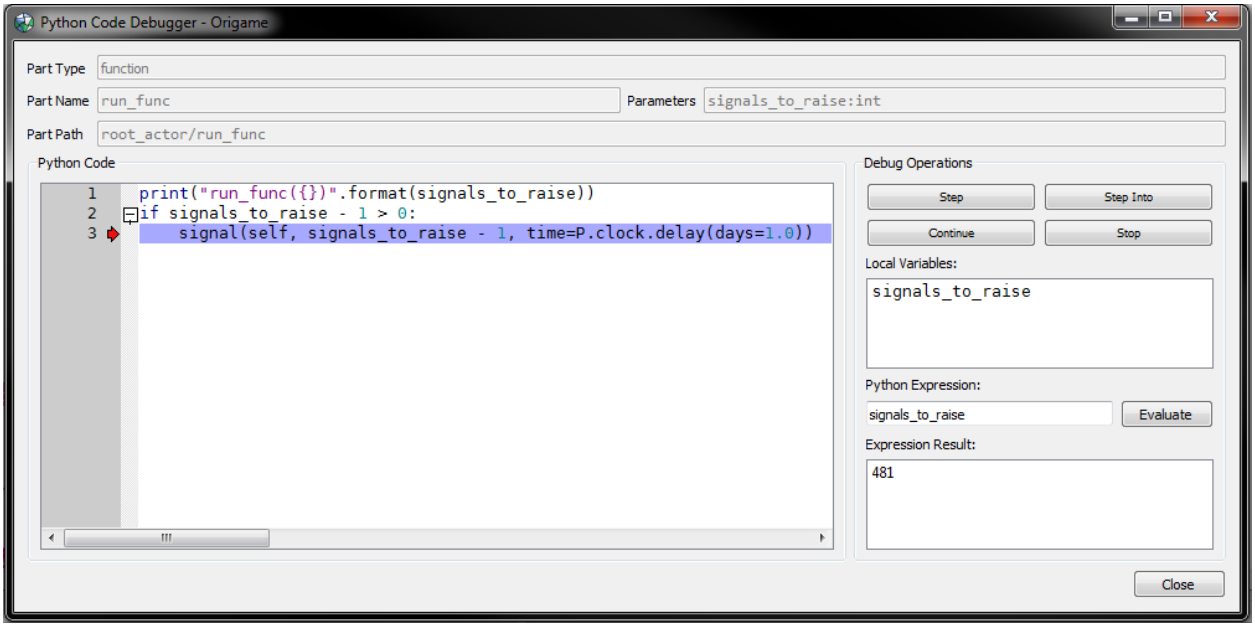


Fig. 4.2: The debug window opens automatically at break-points.

The *Settings* button in the lower right of the Main Simulation Control panel opens the the Main Simulation Settings

panel shown in Figure 4.3. The settings are described in the following table. Note that the settings are **not** saved in the scenario file but in another file in the same folder with the extension ".ssj".

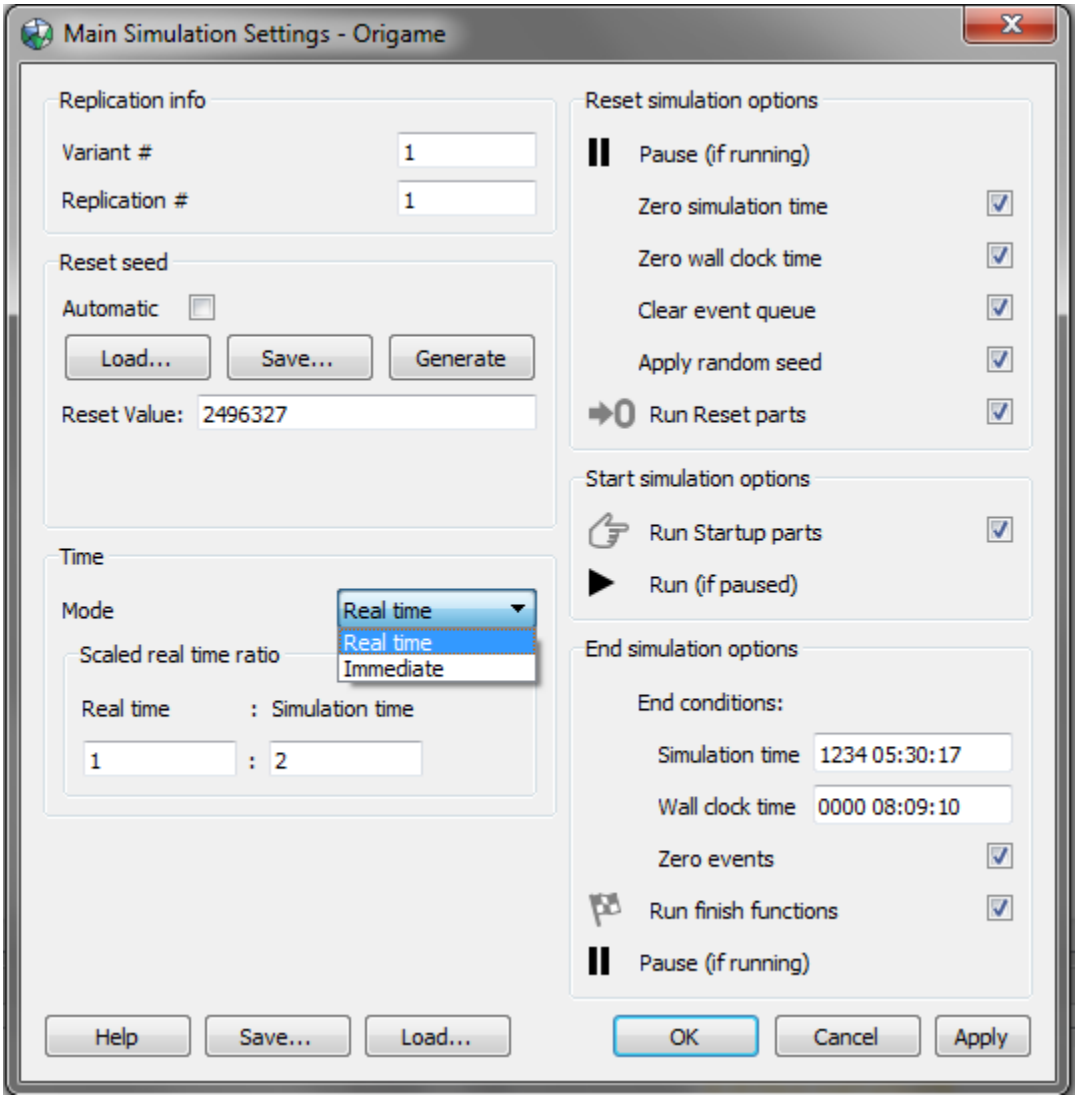


Fig. 4.3: The Main Simulation Settings Panel.

| Main Simulation Settings | Description |
|---|---|
| Variant # | A numerical *ID* for this scenario parameter variant. |
| Replication # | A numerical *ID* for this scenario seed replication. |
| Reset seed | Automatically generate, load, save, or manually generate the reset seed value. |
| Time | Select the Time mode to process events. By default, events are processed *Immediately*, one after the other. This can be changed to scaled *Real-time* according to the ratio of real-time to simulation time. For example, a ratio of 2:1 results in double-time, i.e. 1 simulation minute between events elapses in 30 seconds real-time. |
| Reset simulation options | Options for configuring how the simulation resets. Resetting the simulation will pause the simulation if it's running. It will also do the following if selected: *Zero simulation time*: reset the simulation time to zero; *Zero wall clock time*: reset the wall clock time to zero; *Clear event queue*: clear the event queue of all events; *Apply random seed*: apply a new random seed; *Run Reset parts*: run all Reset parts in the model. |
| Start simulation options | Options for configuring how the simulation starts. Starting the simulation will run the simulation if it's paused after it does the following if selected: *Run Startup parts*: run all Startup parts in the model. |
| End simulation options | Options for configuring how the simulation ends. Ending the simulation will pause the simulation if it's running after it does the following if selected: *Simulation time*: end the simulation at this simulation time; *Wall clock time*: end the simulation at this wall clock time; *Zero events*: end the simulation; *Run finish functions*: run all Finish parts in the model. |

During a GUI session, the simulation can be in either the *Paused* or *Running* states. While *Running*, the GUI has two modes of operation:

- *Animated*: the simulation engine will continuously process events off of the *Event Queue*, and it will continuously refresh the simulation display in the *GUI* so that any animated effects such as the movement of parts and updating part displays will be observed.
- *Non-Animated*: the simulation engine will continuously processes events off the *Event Queue* until there are no events remaining, however, the *GUI* display will not be refreshed. This mode will typically be used for performance reasons once a model has been tested and the user wishes to perform a simulation as quickly as possible..

While *Paused*, the GUI is always "animated". If the simulation is *Running* and *Non-Animated*, and either the user changes the state to *Paused* or toggles Animation mode to *ON*, then the current state of the scenario browser and Model View will start to be animated.

While *Running*, the simulation can be set to use either of two different time modes that include "Immediate" and "Real time":

- *Immediate*: events are processed as fast as possible.
- *Real time*: events are processed in step with with a scaled version of real time. This means that in addition to the base case where one minute of real time corresponds to one minute of simulation time, the user could also select the scale so that two minutes of real time correspond to one minute of simulation time for example. In this mode, delays in the model will correspond to real time delays as the simulation executes.

In the Paused state, the simulation state is frozen and the simulation engine does not retrieve events from the *Event Queue*. In this state, the user can carry out model building actions in the *GUI* such as creating, manipulating and interconnecting parts. The user can manually step the simulation forward in the Paused state. The step will retrieve and process the next event on the *Event Queue*. Any simulation state changes are "animated" in the GUI.

The status of the simulation can be observed in the *Main Simulation Control* panel, shown in Figure 3.14. For *GUI*-run single simulations, the panel displays the relevant updates which include the simulation *State* (as described above), the *Elapsed Sim Time*, and the *# Events* left to process.

As the simulation executes, the *GUI* displays event indicators showing which parts have pending events. The event *indicators* are displayed on the side of the part in the upper left corner.

## 4.2 Batch Simulations

A batch simulation consists of running a number of *replications* of a main simulation where each replication has its own random seed and is run in a separate process. Each replication can also be based on a different *variant* of the model where each variant has small changes made to the base model. Outputs from a batch simulation are identified by their corresponding variant and replication IDs. The outputs are placed in a folder with the timestamp and number of variants and number of replications per variant defined in folder name. Each replication has its own subfolder.

A batch simulation can be run and controlled by the *Batch Simulation* options available in one of three access points of the applications:

1. The *Batch Simulation Control panel*,
2. The *application Toolbar*,
3. The *Simulation menu*.

Figure 4.4 shows the Batch Simulation Control tab that features status and control components, and common batch settings.

On the top left of the panel, the simulation status readout will initially be shown as *Ready*. When the simulation is run, the status will change to display the *Running* state and the percent complete based upon the number of simulations completed out of the total number of variants and replications specified. The number of variants and replications can be input directly via the control panel along with the desired number of computer cores to use (one core is used per simulation). Specifying zero cores will automatically use the maximum number available.

While the batch simulation is running, the panel displays the average time per replication and the estimated time remaining for the batch run to complete. The *Open Batch Folder* button becomes enabled once the batch run starts and will open the folder to the directory containing the batch output. Also, displayed in the top right of the panel are the number of replications finished, running, and failed, and the number of variants finished and failed.

On the lower left of the batch control panel are controls to *Run Batch Simulation* and *Pause*, the latter of which is disabled while in the *Ready* state. Once the batch status has transitioned to *Running*, the *Run Batch Simulation* changes to *Abort* and *Pause* becomes enabled. Pausing the simulation will cause the *Pause* button to change to *Play*. Pressing the *Abort* button will stop the batch run after the user confirms the action from the pop-up dialog. The simulation status will then changed to *Aborted* and the *Abort* button changes to *New*. Alternatively, if the batch run completes, the status will change to *Completed* and, as before, the *New* button appears. Pressing the *New* button will reset the batch simulation to the *Ready* state after the user confirms the action from the pop-up dialog, and a new batch run can be configured and run.

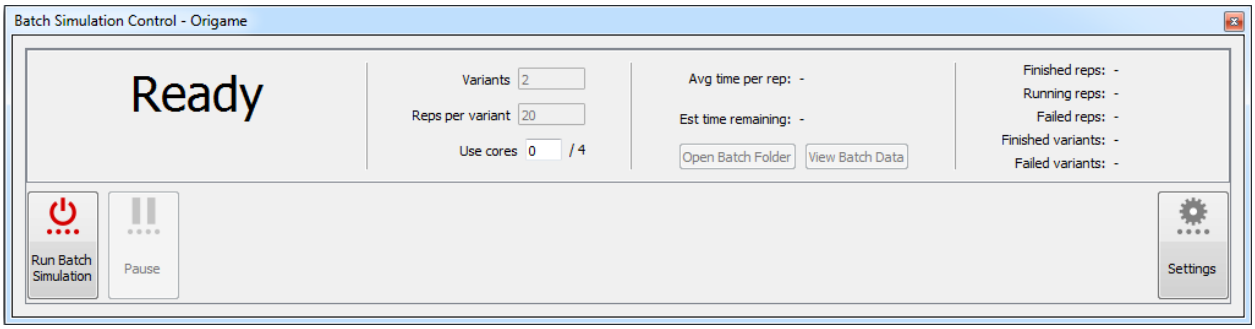


Fig. 4.4: The Batch Simulation Control and Status panel.

The following table provides a summary of the Batch Simulation Control panel.

| Batch Simulation Control | Description |
|---|---|
| Variants | The total number of variants in the batch run. |
| Reps per variant | The total number of replications per variant in the batch run. |
| Use cores | The number of cores to use (one core per simulation). Setting 0 uses all available. |
| Open Batch Folder | Opens the folder contain the batch simulation results. Enable once batch started. |
| Run Batch Simulation | Starts running the batch simulation. |
| Pause / Play | Toggles pausing and running the simulation. |
| Settings | Open the Batch Simulation settings panel as shown in Figure 4.5. |

The *Settings* button in the lower right of the panel opens the the Batch Simulation Settings panel shown in Figure 4.5. Each setting is described in the following table. Note that the settings are **not** saved with the scenario but in a separate file in the same folder with the extension ".bssj".

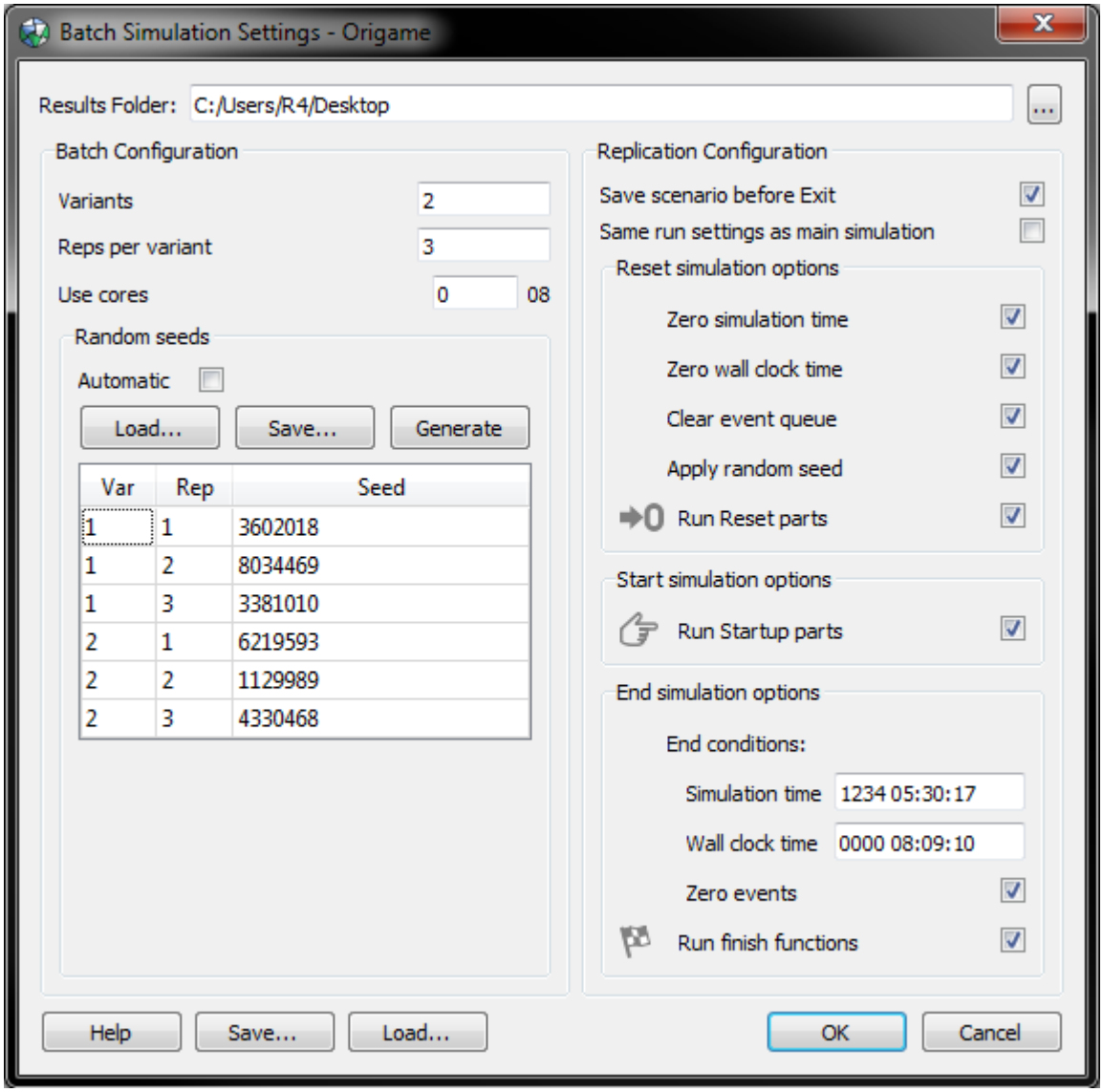


Fig. 4.5: The Batch Simulation Settings Panel.

| Batch Simulation Settings | Description |
|---|---|
| Variants | The number of variants of this scenario. The number of variants is the number of parameter sets to change in the scenario from run to run. |
| Reps per variant | The number of replications on this variant to perform. A new seed value is generated per replication per variant to create a stochastic model output. |
| Use cores | Set the number of computer cores to use of the number available. Generally, the more cores used, the faster the batch simulation is performed. Setting 0 uses all available cores. |
| Random seeds | Automatically generate, load, save, or manually generate the random seed table. |
| Save scenario before Exit | Save the scenario's final state on exit. |
| Same run settings as main simulation | Use the same settings for the following options (Reset, Start, End) as set for main simulation. |
| Reset simulation options | Options for configuring how the simulation resets. *Zero simulation time*: reset the simulation time to zero; *Zero wall clock time*: reset the wall clock time to zero; *Clear event queue*: clear the event queue of all events; *Apply random seed*: apply a new random seed; *Run Reset parts*: run all Reset parts in the model. |
| Start simulation options | Options for configuring how the simulation starts. *Run Startup parts*: run all Startup parts in the model. |
| End simulation options | Options for configuring how the simulation ends. *Simulation time*: end the simulation at this simulation time; *Wall clock time*: end the simulation at this wall clock time; *Zero events*: end the simulation; *Run finish functions*: run all Finish parts in the model. |

The status of the simulation can be observed in the *Batch Simulation Control* panel, as shown in Figure 3.15. For batch-run simulations, the panel displays the relevant updates which include the simulation state ("Ready", "Running", "Paused", "Completed", "Aborted"), percent complete, and the status of the variants and replications that includes the total number of running, finished, and failed simulations. The *Open Batch Folder* button opens the folder containing the batch simulation outputs. The folder name is a combination of the string 'batch_' + 'YYYY–MM–DD–HH–MM–SS' + 'VxR' (V is the number of variants, R is the number of replications). Inside the folder, a copy of the scenario used to perform the batch simulation is saved, along with the seeds used in a comma-separated-value file, and a subfolder for each variant-replication of the scenario that contains a copy of the scenario in the end-state of the run (if the settings are configured to save it) and the corresponding log-file.

### 4.2.1 Batch Data Processing

Origame provides the ability for function and library parts to generate and process batch data via the *batch* object available to scripts. This object provides two primary methods:

- **set_replication_data**(**data): parts that call this will generate batch data when run as part of a batch simulation. Typically, the parts that call this method would have the *Finish* role, but this is not necessary as the method can be called at any time. Each data keyword argument name is the name of a table that stores the value set by each replication. The tables are created automatically, based on the keyword argument names.

  For example, if a *Finish* function part calls set_replication_data(a=A, b=B), with A and B being some data computed by the part, then once completed table "a" will have one value of A per replication, and similarly table "b" will have one value of B per replication. All data is stored in a file called batch_data.sqlite.db in the batch folder of the batch run.

- **load_data**(variant_id): parts that call this will automatically load the batch_data.sqlite.db file found in the same folder as the scenario, and will return the data specific to the given variant. Typically such parts would have the "*Batch*" role since Origame automatically executes all *Batch* role parts at the end of a batch and saves the resulting scenario in the batch folder. Without the batch role, you will have to run the parts that call load_data() manually from the GUI to see the results of processing the batch data.

  For example, in the previous example if the scenario was built to support two variants, then data2 =

batch.load_data(2) would cause data2 to have two keys, "a" and "b": data['a'] would be a numpy array of A values (one per replication), and data['b'] would be a numpy array of all B values.

The process is illustrated in Figure 4.6 below.

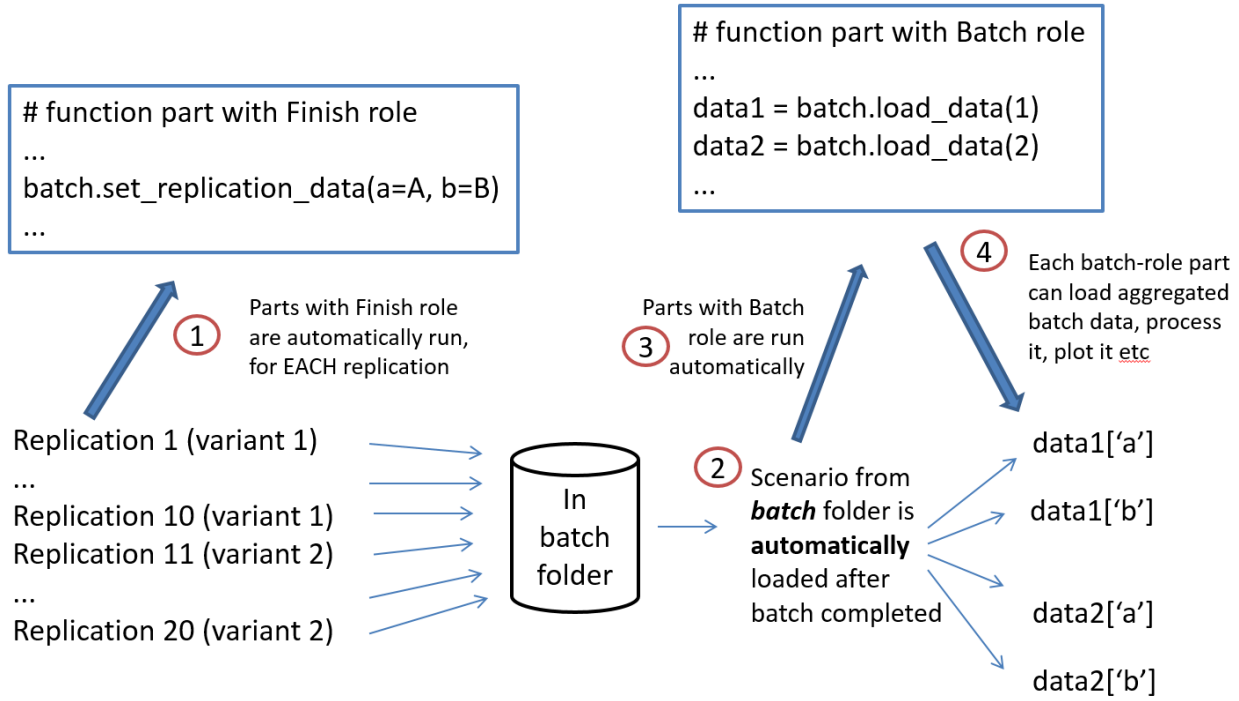


Fig. 4.6: Batch data processing overview.

There are other useful methods on the batch object available to scripts:

- **write_replication_data**(): causes the data set so far by calls to set_replication_data() in the currently running replication to be written to the batch.sqlite.db database. If this method is not called, the data is automatically saved before the replication exits. This method can be useful to save data early, for instance because a replication crashes.
- **add_allowed_data_keys**(*key_names): when this method is called, then all subsequent calls to set_replication_data() must provide valid data keys that match the given names. This method would typically be called in a Startup or Reset role function part.

While building and testing a scenario, it can be useful to test the *Finish* and *Batch* role scripts to verify that they will work, without having to run an entire batch. The following methods could be used towards this:

- **write_test_replication_data**(replication id, variant id, **data): while building the scenario and testing, this method is useful for generating data for several replications without having to run a batch simulation. The script should contain a block of code that generates "representative" or "expected" data, thus allowing you to test that the *Finish* and *Batch* role parts run correctly. Before a batch simulation, the block of code should be disabled (for example, by creating a call parameter that has a default value of False).
- **get_replication_data**(): get all the data set so far by the currently loaded scenario. For example if a scenario contains *Finish* function part A that calls set_replication_data(a=1, b=2) and *Finish* function part B that calls set_replication_data(c=3, d=4), then you may run all *Finish* parts, then run a function part that calls get_replication_data(): it will receive a dict(a=1, b=2, c=3, d=4).
- **get_key_names**(): get the keys saved so far to a batch data file. In the example for load_data(), this method would return ["a", "b"].

- **has_data**(): returns True only if the data file exists and has data in it.

Methods will by default assume that a batch data file is in the folder of the scenario loaded. They can be given a path and a path type if this is not suitable. Note that a batch folder has a copy of the scenario used for the batch run, and the batch data is in that folder. Hence running a batch part only makes sense in two situations:

- If a scenario of a batch folder is opened in the GUI, such as after having clicked "View Batch Data", then all batch data operations are using the batch data produced by the batch simulation.
- While testing the batch processing of a scenario, before running a batch simulation, batch operations are then operating on a "scratch" batch data file saved in the scenario folder; this file can be edited at will without affecting any batch run's folder.

CHAPTER

FIVE

# USING PARTS

This section describes how to use Origame parts.

## 5.1 Part Reference

Parts are the building blocks of the simulation models in Origame. There are two categories of parts: parts with visible frames and parts where the frame is invisible.

Parts with a visible frame includes the *Actor*, *Button*, *Data*, *Datetime*, *Function*, *Library*, *Plot*, *Pulse*, *Sheet*, *SQL*, *Table*, *Time*, and *Variable* parts. They contain a *content* component within which is set the part data, variables, scripts, tables, plots, etc. The title bar, visible for framed parts, identifies several characteristics of the part: the part's icon, name, and a set of shortcut buttons. Title bars with solid (opaque) colors indicate parts that can be created, loaded, and run in simulation. A semi-transparent color indicates a deprecated part (see *Non-Creatable Parts*). The part type is signified by the icon that appears in the top left corner of the frame. The part's name appears next to its icon. On the top right of the frame is a series of optional shortcuts that appear depending on the type of part, however, all framed parts have a common shortcut on the far right of the title bar that toggles the part between *full* and *minimal* Detail Level.

The *Info*, *Hub*, *Multiplier*, and *Node* parts do not show their frame. While the Info part does have part content that's used to display in-*GUI* messages, the others do not, and are used to interconnect parts via links. None of these parts can toggle their Detail Level and do not display their name on the part (the *Object Properties* panel can be used to display part name and other information for these parts, see *Windows, Panels, and Dialogs*) or have any shortcuts.

Part names need not be unique as it is purely descriptive and has no bearing on the simulation logic. Therefore, different parts can share the same name. This feature is enabled by model logic that refers to the *alias* given to a part by another part when the two parts are linked, rather than to the part names. Linked parts are represented by an arrow line connection created from one part (the source part) to another part (the target part). When this occurs, a unique alias is automatically generated and appears as the link name on the connecting link line. Alias uniqueness is automatically enforced by Origame. Thus, a model can have a single part that is linked to many other parts that each use a different alias, but may all have the same part name. This scheme allows model logic to be portable and easily integrated with other model logic because parts do not need to be protected against naming collisions.

Parts are hierarchically organized into the parent part and its associated children parts. The part frame stores this hierarchical information (parent, child, etc.) along with the following attributes common to all parts: the part's name, type, position, comments, and any connecting links. Scenarios created using the Prototype can have invisible or bold-frame parts. These properties are identified in Origame but can't be changed.

There are currently 18 part types. The rest of this section will describe each of these parts.

### 5.1.1 Actor Part

Actor parts are used to modularize, organize, and encapsulate related implementation details of a model into a hierarchy of parent and child actors. Actors act as logic containers that organize the scenario into different components. Further, since actors can contain other actors, a model hierarchy can be developed which allows the creation of model sub-systems down to the desired level of detail. The model hierarchy of parent and child actors can be viewed in the Actor Hierarchy panel. By clicking on an actor listed in the hierarchy, its contents can be viewed in the Model View. The Model View always displays the contents of a single actor. By default, the application displays the contents of the top-level "root" actor of a scenario on application start-up. An actors contents can also be viewed in the Model View by clicking the down-arrow located on an actor's frame. New actor parts can be created by right-clicking on a blank part of the Model View and selecting *New Part* > *Actor* from the context menu.

Actors can also be created by importing one scenario into another, where the imported scenario becomes a new child actor. Once created, they can help simplify old models by sub-dividing them into smaller components. This can be accomplished by cutting or copying part selections and pasting them into a new actor part.

To facilitate link connections across the actor boundary, all parts can raise their *Interface Level* which exposes the part to link connections at higher levels in a model's actor hierarchy. The interface level of a part is accessed via the Object Properties panel after the part is selected. A drop down menu indicates what the current level is and facilitates raising or lowering the interface level within the actor hierarchy. The level's value indicates the number of actors (level 1 is the part's parent actor) to which the part should be exposed. When a part's interface level is increased to level *n*, the part appears as a port on the $n^{th}$ actor and all the actors located in the hierarchy below it. See section *Interface Parts* for more information.

Other parts inside the actor can link to their parent actor by directly wiring to the actor proxy widget located at the centre of the Model View when viewing the actor's contents. Linking to the parent might be used to change the parent actor's icon or to change its position (as it appears within its parent).

Double-clicking the actor part or right-clicking and selecting "Edit..." opens an editor dialog (see *Part Editing, and Undo, Redo Commands*) that displays the content shown in Figure 5.1, that allows various properties of the part to be edited or viewed:

- selection of the part's image file (default or custom),
- image orientation.

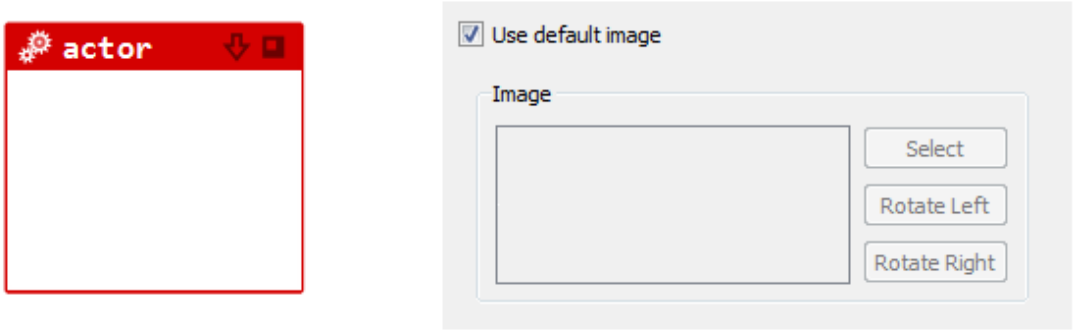


Fig. 5.1: The Actor part and editor dialog.

Images selected for the actor part are saved to the */images* subfolder located in the scenario directory.

Clicking on Part Help opens this page.

*Using Parts*

*Table of Contents*

### 5.1.2 Button Part

Buttons are used to trigger the immediate execution of the function part or parts to which they are linked. A button can be configured to work as either a push button ("momentary" type) or toggle switch ("toggle" type), and to trigger 'on' events, 'off' events, or both. These configurations can be accessed from the parts editor dialog accessed by double-clicking the part's frame. New button parts can be created by right-clicking on a blank part of the Model View and selecting *New Part* > *Button* from the context menu.

In order for the button part to trigger events, it must be linked to one or more function parts. To create a link, right-click on the button and choose *Create Link* and then click on a function part. When configured as a push-button, the button state matches the mouse click state. In the toggle configuration, the button part state toggles on each mouse down-click event. When the user left-clicks on the button part, it will immediately call any function parts to which it is linked.

**button_state parameter:** Linked function parts will be called with each button press. The function part may define the parameter "button_state" as the first parameter but this is not required. When it is defined, when the button is pressed it will pass an enumerated value *ButtonStateEnum* that is set to *pressed* (value == 0). When the user releases the left-click, the button will again call any linked function parts with the value *released* (value == 1).

If the user clicks down on the button part and drags the mouse pointer off the button part without releasing the mouse button, a *released* event will be triggered. The button part can also be configured to call any linked function parts on down press events only, or release events only. The button part uses two icons to display the button in the pressed and not pressed states.

Double-clicking on the button part's frame or right-clicking and selecting "Edit..." opens an editor dialog (see *Part Editing, and Undo, Redo Commands*) that displays the content shown in Figure 5.2, that allows various properties of the part to be edited or viewed:

- type selection: momentary or toggle,
- event selection: switch on or off,
- selection of the part's image file (default or custom) for the on and off states,
- image orientation.

Images selected for the button part are saved to the */images* subfolder located in the scenario directory.

Clicking on Part Help opens this page.

*Using Parts*

*Table of Contents*

### 5.1.3 Clock Part

The clock part has been deprecated and can no longer be created in new scenarios but continues to be fully supported in previously built scenarios. For alternative time options see *Datetime part* and *Time part*.

The clock part is used to schedule future events by defining the time at which the signals emitted by the function part or parts, linked to the clock, should be triggered. Any specified time or time delay defined in the simulation must be associated with a particular clock part. The current time, in clock ticks, and the tick rate of the clock can be set at any time. Additionally, the current calendar date (year, month, day) and time (hour, minute, second) and calendar unit (e.g. one clock tick = one day), can be set at any time. New clock parts can be created by right-clicking on a blank part of the Model View and selecting *New Part* > *Clock* from the context menu.

When the clock part is called with a time or time interval parameter, it computes a global time at which the event will be executed so that all pending events in the simulation can be sorted according to their global time. However, the user only ever interacts with times according to specific clock parts.

As an example of how to use a clock part to schedule a future event, consider a function part with alias "queue_signals" that has been linked to a clock part with alias "clock" (link from the function part to the clock part). "queue_signals"

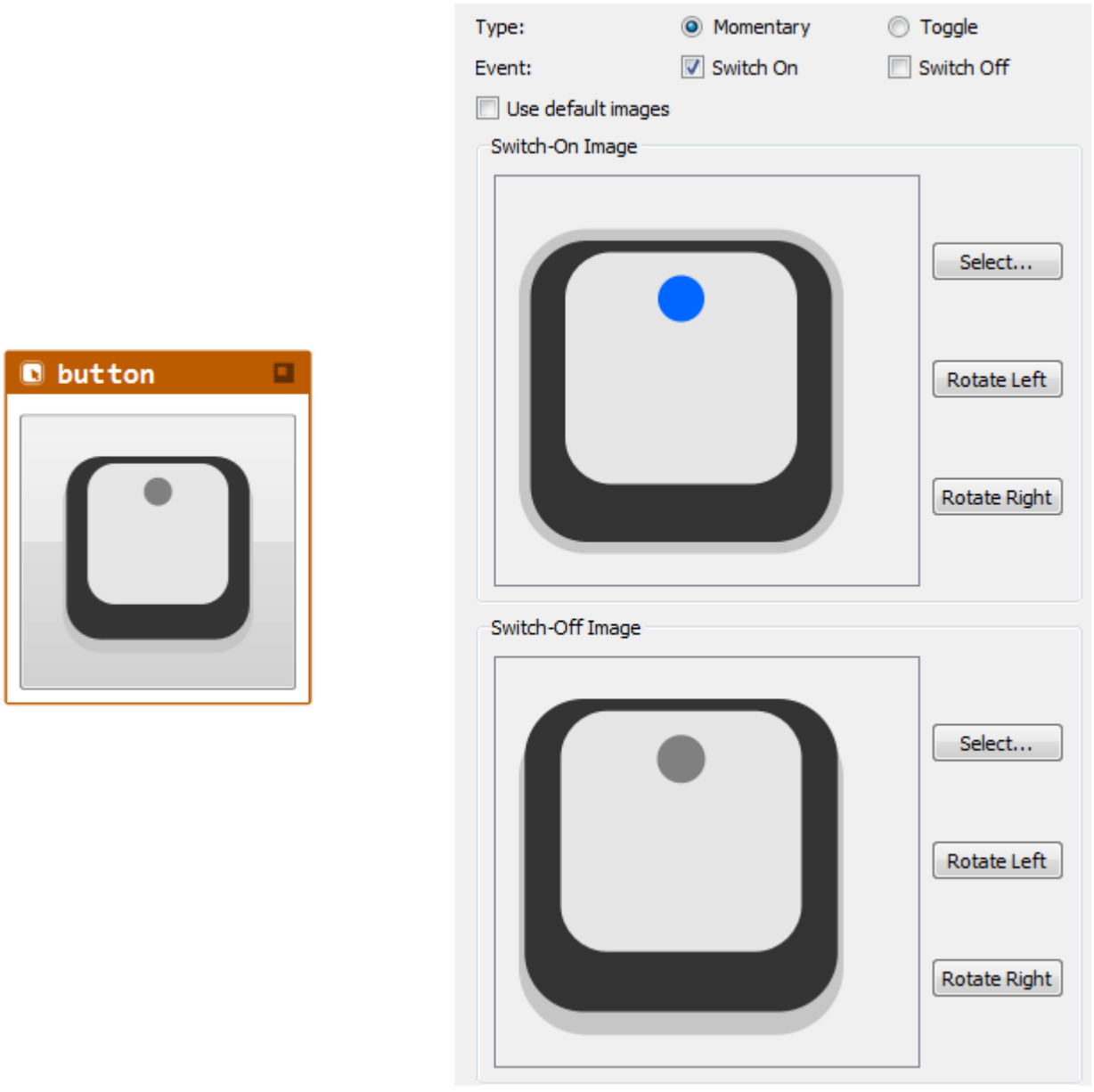


Fig. 5.2: The Button part and editor dialog.

will use the application *API* to get a future time from the clock and then use the signal() method to add events to the Event Queue that, in this case, will correspond to triggering other function parts to run at specific future times. Assume for this example that this other function part is called "foo" and has been linked to "queue_signals" (link from "queue_signals" to "foo"). The table below lists the supported call types for scheduling "foo" to execute at a future time on the clock.

| Clock call type (*API* call in "queue_signals") | Event Execution Time |
|---|---|
| signal(link.foo, args=(), time=link.clock(5), priority=0) | Execute "foo" at time 5 ticks according to "clock" |
| signal(link.foo, args=(), time=link.clock.delay(10), priority=0) | Execute "foo" after 10 ticks according to "clock" |
| signal(link.foo, args=(), time=link.clock(2012, 1, 1), priority=0) | Execute "foo" at 12:00 am on January 1, 2012 according to "clock" |
| signal(link.foo, args=(), time=link.clock(2012, 1, 1, 14, 30, 15), priority=0) | Execute "foo" at 14:30:15 on January 1, 2012 according to "clock" |
| signal(link.foo, args=(), time=link.clock.delay(months=2, days=10), priority=0) | Execute "foo" after 2 months and 10 days according to "clock" |

The clock part behaves like a real wall clock in the sense that it can be set to any time the user wants and the clock will tick forward from that time as global time advances. This allows different models to use different clocks within the same simulation scenario. The following are two examples of this type of scenario:

1. The case where each model has a clock with a different tick period: a start–up function A of model A is wired to a clock with a tick period of 1 day and signals itself to be run at clock.delay(ticks=6). Similarly, a start–up function B of model B is wired to another clock with a tick period of 7 days and signals itself to be run at clock.delay(ticks=1). Then when model A is run on its own, function part A is re–run at 6 days, then 12 days, etc. When model B is run on its own, its part B is run at 7 days, then 14 days, etc. When both sub–models are brought together, they still execute at their individual tick periods, such that both parts are re–run at 6, 7, 12, 14,... days. The order of events is preserved, as are the time deltas between the events they produce.
2. For models that use different calendar clocks, the reset function parts must be edited to set the clocks at the same time. If the models only use relative delays, no further work is necessary to use the two models in the same scenario. If one or both of the models use absolute calendar times, the model will have to be edited. This shows that absolute calendar times should only be used by reset/start–up functions for increased modularity, and tick clocks for absolute modularity.

Double-clicking on the clock part's frame or right-clicking and selecting "Edit..." opens an editor dialog (see *Part Editing, and Undo, Redo Commands*) that displays the content shown in Figure 5.3, that allows various properties of the part to be edited or viewed:

- the start date (YYYY/MM/DD),
- the start time (HH:MM:SS),
- the initial number of ticks,
- the number of weeks, days, hours, minutes, and seconds per tick.

Clicking on Part Help opens this page.

*Using Parts*

*Table of Contents*

### 5.1.4 Data Part

The data part is used to store variables. It consists of a simple container class where the variable name is stored as a 'key' that can be used to access the variable 'value'. The variable names, or keys, must be unique and valid Python names. The values can be any valid Python type such as strings, integers, floats, lists, tuples, and dictionaries. New

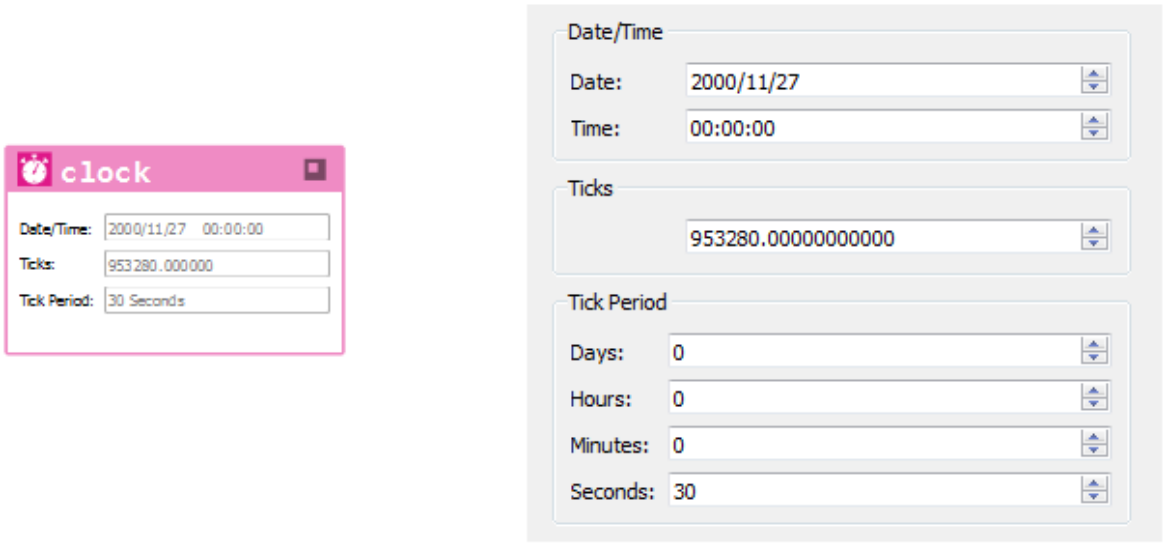


Fig. 5.3: The Clock part and editor dialog.

data parts can be created by right-clicking on a blank part of the Model View and selecting *New Part* > *Data* from the context menu.

Data elements are member variables that are added to or removed from the container class. When a function part is linked to a data part it can access the container class using the link alias. A function part can add new member variables to the data part by assigning them. For example, if a function part is linked to a data part with alias "data", a new variable name "size" can be created and assigned a value: *link.data.size = 20*. Then it can be deleted from the data part using the Python "del" function: *del link.data.size*. Alternatively, the same result can be accomplished in a parametrized function part with the arguments 'key' and 'value', and the script, *link.data[key] = value*. Running the function and providing the two parameters as *size* and *20* will add the key "size" with a value of 20.

Double-clicking the data part or right-clicking and selecting "Edit..." opens the editor dialog (see *Part Editing, and Undo, Redo Commands*) that displays the content shown in Figure 5.4, where the current keys and their corresponding values can be viewed and changed by clicking into the *Key* or *Value* field and editing the values directly. The Keys can be sorted into alphabetical, reverse-alphabetical, and as-created order by clicking the Key header name successively to toggle through each sorting option. The final sort-order selected will be propagated to the part shown in the Model View, along with all other changes, once the *Apply* or *OK* button is pressed. The dialog provides the following data editing options:

- insert before the selected row,
- insert after the selected row,
- select all,
- cut selection,
- copy selection,
- paste selection,
- delete selection,
- move selection up,
- move selection down.

Clicking on Part Help opens this page.

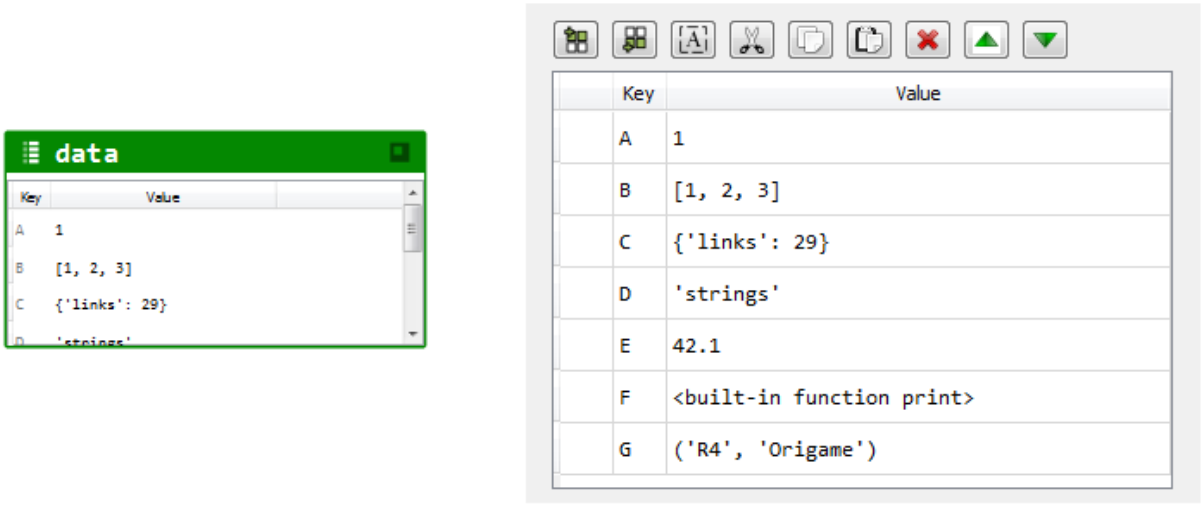


Fig. 5.4: The Data part and editor dialog.

*Using Parts*

*Table of Contents*

### 5.1.5 Datetime Part

The datetime part is used to schedule future events by defining the time at which the signals emitted by the function part or parts, linked to the datetime part, should be triggered. Any specified time or time delay defined in the simulation must be associated with a particular datetime part. The current time can be set at any time. Additionally, the current date (year, month, day) and time (hour, minute, second) can be set at any time. New datetime parts can be created by right-clicking on a blank part of the Model View and selecting *New Part* > *Datetime* from the context menu.

When the datetime part is called with a time or time interval parameter, it computes a global time at which the event will be executed so that all pending events in the simulation can be sorted according to their global time. However, the user only ever interacts with times according to specific datetime parts.

As an example of how to use a datetime part to schedule a future event, consider a function part with alias "queue_signals" that has been linked to a datetime part with alias "datetime" (link from the function part to the datetime part). "queue_signals" will use the application *API* to get a future time from the datetime and then use the signal() method to add events to the Event Queue that, in this case, will correspond to triggering other function parts to run at specific future times. Assume for this example that this other function part is called "foo" and has been linked to "queue_signals" (link from "queue_signals" to "foo"). The table below lists the supported call types for scheduling "foo" to execute at a future time on the datetime.

| Datetime call type (*API* call in "queue_signals") | Event Execution Time |
|---|---|
| signal(link.foo, args=(), time=link.datetime(2012, 1, 1), priority=0) | Execute "foo" at 12:00 am on January 1, 2012 according to "datetime" |
| signal(link.foo, args=(), time=link.datetime(2012, 1, 1, 14, 30, 15), priority=0) | Execute "foo" at 14:30:15 on January 1, 2012 according to "datetime" |
| signal(link.foo, args=(), time=link.datetime.delay(months=2, days=10), priority=0) | Execute "foo" after 2 months and 10 days according to "datetime" |

The datetime part behaves like a real wall clock in the sense that it can be set to any time the user wants and the datetime will tick forward from that time as global time advances. This allows different models to use different datetime parts within the same simulation scenario. The following are two examples of this type of scenario:

1. The case where each model has a datetime with a different delay: a start–up function A of model A is wired to a datetime and signals itself to be run at datetime.delay(days=6). Similarly, a start–up function B of model B is wired to another datetime and signals itself to be run at datetime.delay(days=7). Then when model A is run on its own, function part A is re–run at 6 days, then 12 days, etc. When model B is run on its own, its part B is run at 7 days, then 14 days, etc. When both sub–models are brought together, they still execute at their individual delays, such that both parts are re–run at 6, 7, 12, 14,... days. The order of events is preserved, as are the time deltas between the events they produce.
2. For models that use different datetime parts, the reset function parts must be edited to set the datetime parts at the same time. If the models only use relative delays, no further work is necessary to use the two models in the same scenario. If one or both of the models use absolute times, the model will have to be edited. This shows that absolute times should only be used by reset/start–up functions for increased modularity.

Double-clicking on the part's frame or right-clicking and selecting "Edit..." opens an editor dialog (see *Part Editing, and Undo, Redo Commands*) that displays the content shown in Figure 5.5, that allows various properties of the part to be edited or viewed:

- the start date (YYYY/MM/DD),
- the start time (HH:MM:SS).

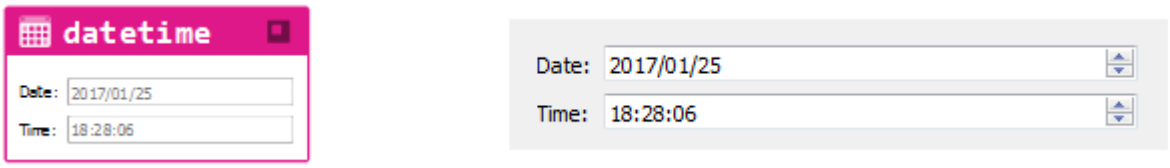


Fig. 5.5: The Datetime part and editor dialog.

Clicking on Part Help opens this page.

*Using Parts*

*Table of Contents*

### 5.1.6 File Part

The file part is used to reference a file or folder to be accessed by the model. The reference is made by specifying a path which can be either absolute or relative (i.e. a full path to a file, or just a file name). Relative paths can be specified as relative to the scenario folder by checking the "Relative to Scenario Folder" check box. Relative paths that are not specified as relative to the scenario folder are essentially relative to the current working directory, so typically such file part paths would be combined with other file parts to create absolute paths. It is the user's responsibility to make proper use of the file part paths in their model.

The file part provides a rich API for path manipulation. This API is a superset of that of pathlib.Path class from Python's standard library. For example, Path methods such as *read_text*, *write_text*, *parent*, *name*, *suffix*, *is_absolute*, *exists*, *joinpath*, *mkdir*, *cwd*, etc, are directly available from the file part object. File path manipulation based on os.path should be avoided, in favor of the Path API provided by the File part type.

Double-clicking on the file part's frame or right-clicking and selecting "Edit..." opens an editor dialog (see *Part Editing, and Undo, Redo Commands*) that displays the content shown in Figure 5.6, that allows various properties of the part to be edited or viewed:

- a path to a file or folder.
- a browse-to-File button that opens a dialog that allows a file to be selected.
- a browse-to-Folder button that opens a dialog that allows a folder to be selected.

- a check box that allows the user to specify if the path is to be treated as relative to the scenario folder. When this box is checked, the part will automatically prepend the path to the scenario folder whenever the path of the file part is used in Path operations.

Example 1: a File part that contains C:pathtofile.txt: then from a Function part linked to it via a link called "file", link.file.filepath returns a Path instance pointing to C:pathtofile; link.file.exists() tests whether that file exists on the filesystem; etc.

Example 2: a File part that contains a relative path "file.txt", and relative-to-scenario=False: with a scenario as in example 1, link.file.filepath returns "file.txt", so link.file.exists() would test whether that file exists in the folder that is the application's current working directory. Since this is rarely useful, such relative file parts would typically be used in combination with other file parts, such as an "output folder" File part linked from the Function part as "output_folder". If "output folder" contains the absolute path "C:pathtooutputfolder", then the Function part linked to "output folder" and "file" can use the expression "link.output_folder / link.file", which returns a Path instance pointing to C:pathtooutputfolderfile.txt.

Example 3: a File part that contains a relative path "output_folder", but relative-to-scenario=True: with a scenario as in example 1, link.output_folder.filepath returns "output_folder", but link.file.exists() would test whether that folder exists in the scenario folder. In combination with a file part that has "file.txt" with relative-to-scenario=False, a script could do "(link.output_folder / link.file).exists() and this would work regardless of where the scenario is located on the filesystem, because the two file parts have relative paths.

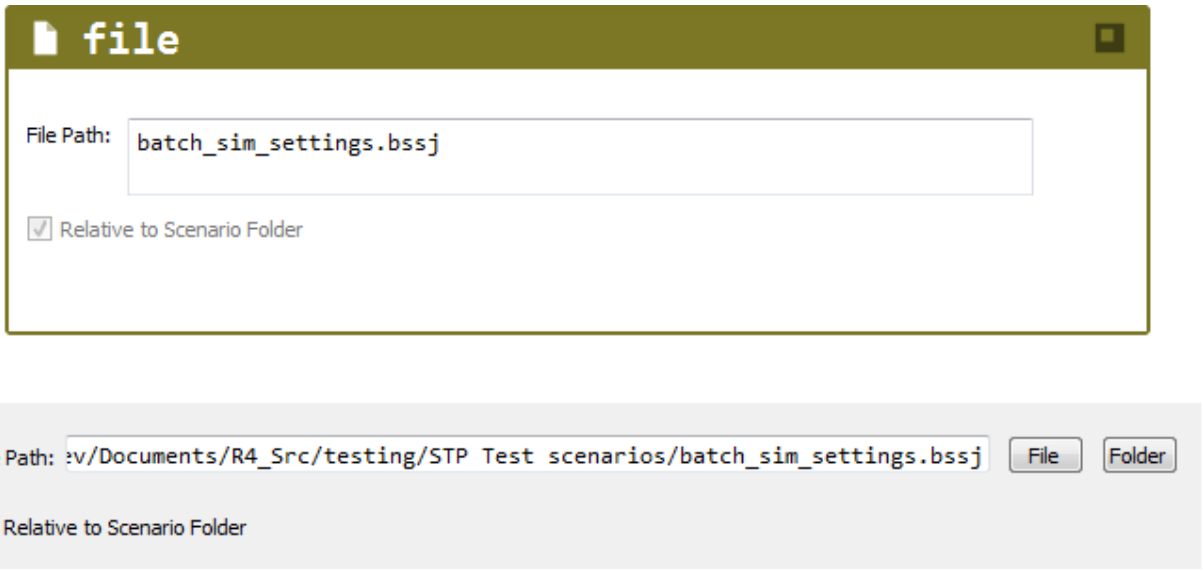


Fig. 5.6: The File part and editor dialog.

Clicking on Part Help opens this page.

*Using Parts*

*Table of Contents*

### 5.1.7 Function Part

Function parts are used to define the events that occur in the simulation and are the most fundamental part used to build models. The main component, or body, of the function part is a Python script that represents the instructions to be carried out during an event. The function part can define parameters that are used to pass information to the function body, and the function may return a value. The function part can be linked to any other part type such as a data part or

a clock part. To create a link to another part, right-click and choose "Create Link" and then click on the other part. If a function part is linked to another part, its body has access to that part using the alias specified by the link. In other words, aliases of the links that connect a function part to other parts constitute the namespace of the function part. Code completion reveals the contents of the namespace as the user writes the script.

New function parts can be created by right-clicking on a blank part of the Model View and selecting *New Part* > *Function* from the context menu. To run a function part, click the run button in the upper-right corner of the function part's frame in the Model View.

When one function part is linked to another function part, the former can call the latter as a function, optionally passing in parameters and receiving the return value. A function part can also schedule another function part to execute at any time greater than or equal to the current time. This is achieved by calling a *signal* function which has the following definition:

*signal(function, args=(), time=None, priority=0)*

where *function* is the alias of the target function part to be scheduled for future execution, *args* is a tuple consisting of the parameters to be passed to the target function part, *time* is when the target function part will execute, and *priority* is the priority of the signal. If *args* are not specified, an empty tuple will be used. If the *priority* parameter is not specified, it will default to 0. If the *time* parameter is not specified, then it will default to the current simulation time. In order to specify the *time* parameter, a clock part must be used. For example, if the function part is linked to a clock part with the alias "clock", the function body can schedule another function part (shown below using the alias "foo") to run at time 5 ticks on the clock via the *API*:

*signal(link.foo, args=(), time=link.clock(5), priority=0)*

The combination of a function part, its parameters, a clock time, and a priority constitutes a scheduled event. The *signal* function places this event on the Event Queue, a queue of all pending events in the simulation. Because the scheduled events will not execute until a future time, the return value of the scheduled events is not returned to the function that called signal.

Double-clicking on the function part's frame or right-clicking and selecting "Edit..." opens a script editor dialog (see *Part Editing, and Undo, Redo Commands*) that displays the content shown in Figure 5.7, that allows various properties of the part to be edited or viewed:

- the part's roles (Startup, Reset, Finish, Setup) and role execution order (not yet implemented)
- parameter specification,
- editing tools for undo, redo, cut, copy, paste, and delete,
- a script editor,
- a *Links*, *Symbols*, and *Imports* panel that provides link and object import management options. See section *Script-Base Parts: Links, Symbols, and Imports* for more information on these panels.

If parameters are specified in the editor, the function part will open a prompt every time the function part is run to allow the user to enter the parameter values. These values must be valid Python expressions. e.g. a string must include quotes.

Clicking on Part Help opens this page.

*Using Parts*

*Table of Contents*

### 5.1.8 Hub Part

The hub part, shown in Figure 5.8, is represented in the *GUI* as a ring, provides a single access point to a group of parts it has linked connections to. A reference to each part that the hub part is linked to is stored as a member variable in the hub where the name of each member variable is the alias name displayed on the connecting link. This enables function

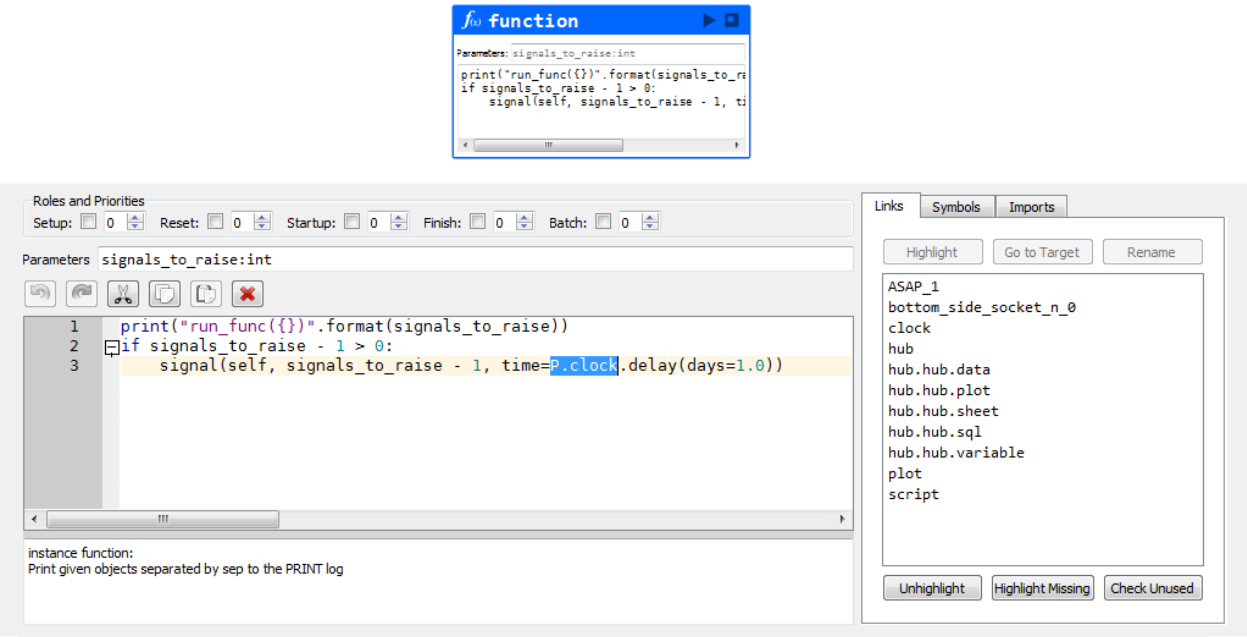


Fig. 5.7: The Function part and editor dialog.

parts that are connected to the hub to access any of the parts to which the hub is connected via dot notation. For example, if the hub part linked to a data part with alias "data" and a table part with alias "table", then a function part that is linked to the hub with alias "hub" can access "data" and "table" using dot notation as follows: "link.hub.data" and "link.hub.table".

The part frame of the hub is not visible and thus it's name is not displayed.

There is no editor for the hub part. The only attributes that can be edited are its name and position which are edited using the *Object Properties* panel.

Fig. 5.8: The Hub part.

*Using Parts*

*Table of Contents*

### 5.1.9 Information Part

The information (info) part displays arbitrary information or commentary that may be used for any reason, usually to assist with understanding how a model works. Info parts are displayed with a unique simplified visual style to distinguish them from the rest of the model logic. They can be linked to other parts to show what the comment refers to if desired, but this serves only a cosmetic purpose. If function or library part is linked to an info part, it can set the comment string that is displayed.

Double-clicking on the info part's frame or right-clicking and selecting "Edit..." opens an editor dialog (see *Part Editing, and Undo, Redo Commands*) that displays the content shown in Figure 5.9, that allows the comment input field to be edited or viewed.

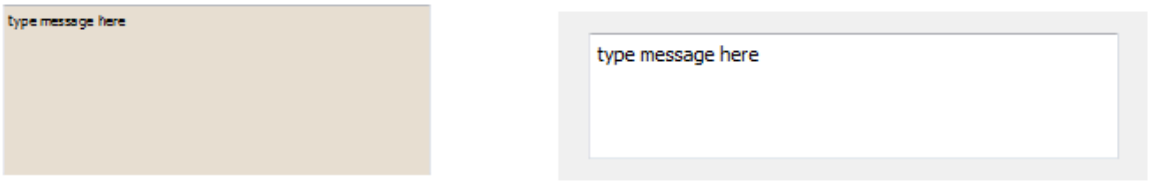


Fig. 5.9: The Info part and editor dialog.

Clicking on Part Help opens this page.

*Using Parts*

*Table of Contents*

### 5.1.10 Library Part

The library part, shown in Figure 5.10, provides the user with a part where a library of methods and attributes can be defined. These methods can be called by function parts that are linked to the script. Methods are defined using Python syntax,

```
def method_name(*args):
        # Method body
```

The Library part can be run directly by pressing the run button in the upper-right corner of the part's frame. If there are methods defined inside the script, a dialog will open and prompt the user to select which method to run, followed by another prompt to specify the method parameters (if any). The Library part can also be run automatically by linking it to a function part that can use the Library part's link alias to execute specific script methods and define their arguments programmatically.

Double-clicking on the library part's frame or right-clicking and selecting "Edit..." opens an editor dialog (see *Part Editing, and Undo, Redo Commands*) that displays the content shown in Figure 5.10, that allows various properties of the part to be edited or viewed:

- script editing tools for undo, redo, cut, copy, paste, and delete,
- a script editor,
- a *Links*, *Symbols*, and *Imports* panel that provides link and object import management options. See section *Script-Base Parts: Links, Symbols, and Imports* for more information on these panels.

For a discussion on how to use the *Links* tab, refer to the corresponding section in the *Function Part Reference*.

Clicking on Part Help opens this page.

*Using Parts*

*Table of Contents*

### 5.1.11 Multiplier Part

Multiplier parts, shown in Figure 5.11, is represented in the *GUI* as a ring with several spokes radiating from it, are used to call a group of executable parts at once. The linked parts must all accept the same number of parameters. The

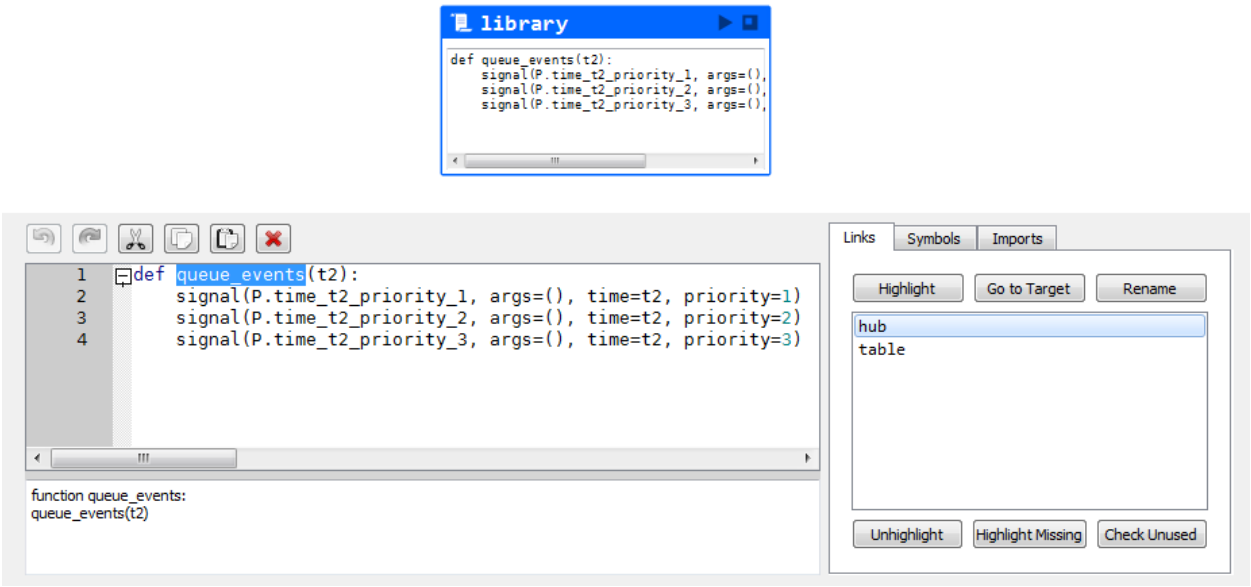


Fig. 5.10: The Library part and editor dialog.

multiplier can be linked to any executable part such as function parts, *SQL* parts, other multipliers, and node parts that are linked to function, *SQL*, or multiplier parts. When the multiplier is called or signalled (possibly including parameters) by another part, it will call every part that it is linked to, passing the incoming parameters in each call. For example, a multiplier can be used to send a reset signal throughout a simulation. There are no restrictions on the number of links that can go into or out of a multiplier part. When multiplier parts are added to the Event Queue, they are added with the time specified in the signal, but the executable parts called by the multiplier are placed on the queue with priority *ASAP*.

The part frame of the multiplier is not visible and thus it's name is not displayed.

There is no editor for the multiplier part. The only attributes that can be edited are its name and position which are edited using the *Object Properties* panel.

Fig. 5.11: The Multiplier part.

*Using Parts*

*Table of Contents*

### 5.1.12 Node Part

The node part, shown in Figure 5.12, is represented in the *GUI* as a small solid circle, is a passive part that forwards the information it receives from potentially multiple connected parts to a single part that it is connected to. They are used to break links into segments in order to create alternate arrangements of links for better visual effects.

A node can be linked only to one other part, but multiple parts can be linked to a node part. If a node part is linked to a data part containing variables "num" and "count", then a function part that is linked to the node part with alias "x" can access the data part as if it were directly linked to the data part: *x.num = x.count + 1*.

Multiple node parts may be linked together to form a network. Within a node network, the direction of wiring proceeds along the link as indicated by the link's arrowhead (i.e. node A to node B or node B to node A). Collectively a node network behaves like a single node. Thus, only a single link can connect from the node network to another part (the network exit link), but multiple links can connect other parts to the network (network entry links). Each part that has a network entry link has access to the single part that the network exit link is connected to. If a single node part or network of nodes is not linked to another part, then the node parts all reference the Python *None* value. For example, if a function part is linked to a node part with alias "x" but the node part is not linked to another part, the alias "x" will reference *None*.

The part frame of the node is not visible and thus it's name is not displayed.

There is no editor for the node part. The only attributes that can be edited are its name and position which are edited using the *Object Properties* panel.

Fig. 5.12: The Node part.

*Using Parts*

*Table of Contents*

### 5.1.13 Plot Part

The plot part is used to plot simulation data. Parts that are linked to the plot part provide it with the information to be plotted on the chart. The plot part has four shortcut buttons on the top-right of its frame that include *Zoom in*, *Zoom out*, *Update the plot*, and *Show minimal detail*. Zooming the plot part is independent of the zooming the Model View. Loading a scenario that contains plot parts will initially appear empty even if the data is available. To display the data in the part after loading the scenario the *Update the plot* short-cut button must be pressed or the context menu option selected.

Once an image is displayed, the image and data can also be exported by accessing the context menu, however, these options will only become available after the plot has been updated. For each option a dialog will open.

- Image Export: the Image Export dialog requires a file path (including the file name) to save the image to, an image resolution, and an image format. The following formats are supported: Portable Network Graphics (*PNG*), Portable Document Format (*PDF*), PostScript format (*PS*), Encapsulated Postscript Vector graphics (*EPS*), and Scalable Vector Graphics (*SVG*).
- Data Export: the data export dialog requires a file path to export the data to (in Excel *XLS* format), and the name of the sheet that will appear in the file. Currently only data from line plots, scatter plot, histograms, bar charts, and pie charts can be exported. Note that the data exported from the plot may not be identical to the original data used to make the plot. For example, pie charts compute a *wedge* as a percentage of the pie out of 360 degrees from the original data. Therefore, it is the percentage of each pie wedge that appears in the exported data. If the original data is required for export, then the source part's export feature (e.g. Sheet Part, etc.) should be used instead.

The plot part uses a script that defines three methods that control how the data is plotted: a 'configure' method that sets up the plot's axes, a 'plot' method that gets the data from connected parts and creates the plot, and optionally a 'preview' method. These can be defined in the plot part or in a function part that sets the script to the plot part. Since the script implements the Python Matplotlib package, any valid Matplotlib command can be employed in the plot part script.

An example of defining the script inside the plot part is shown below, where *sheet* represents the alias of a sheet part that contains the data to be plotted:

```
def configure():
        axes = setup_axes(rows=1, cols=1)
        axes.set_title('ORIGAME Plot')
        axes.grid(True)
        axes.set_xlabel('x')
        axes.set_ylabel('y')

def plot():
    if link.sheet.num_rows:
        x_col_data = link.sheet.col(col_idx=0)
        for col_idx in range(1, link.sheet.num_cols):
            y_col_data = link.sheet.col(col_idx=col_idx)
            axes.plot(x_col_data, y_col_data)
```

Alternately, the above could be defined by a function part as a multi-line string and then assigned to the plot part's script attribute: *link.plot.script = plot_script*. Then call the Plot part's update method to run the script and draw the plot image: *link.plot.update_fig()*.

As shown above, after the script has been defined, *update_fig()* must be called to trigger the plot part to update the plot image. The plot can also be updated by clicking on the blue update button on the plot widget, or right-clicking on the part to open the context menu, and selecting "Update Plot".

Double-clicking on the plot part's frame or right-clicking and selecting "Edit..." opens an editor dialog (see *Part Editing, and Undo, Redo Commands*) that displays the content shown in Figure 5.13, that allows various properties of the part to be edited or viewed:

- plot script editing tools for undo, redo, cut, copy, paste, and delete,
- a plot script editor,
- an *Info* panel displaying relevant information on connected *Links*, available *Functions* and *Modules*, and *Help*,
- a *Links*, *Symbols*, and *Imports* panel that provides link and object import management options. See section *Script-Base Parts: Links, Symbols, and Imports* for more information on these panels,
- a plot preview panel that displays a preview when the *Refresh* button is pressed.

For a discussion on how to use the *Links* tab, refer to the corresponding section in the *Function Part Reference*.

Clicking on Part Help opens this page.

*Using Parts*

*Table of Contents*

### 5.1.14 Pulse Part

Pulse parts can be used to place events on the event queue at regular periods. The pulse part defines a pulse period, state, and priority. When a scenario is started, the sim controller adds all pulse parts with an 'active' state to the Event Queue at time zero and with the priority defined in the pulse part. When the pulse event executes, an event corresponding to each part linked to the pulse will be added to the queue at *ASAP* priority. Each of those events then

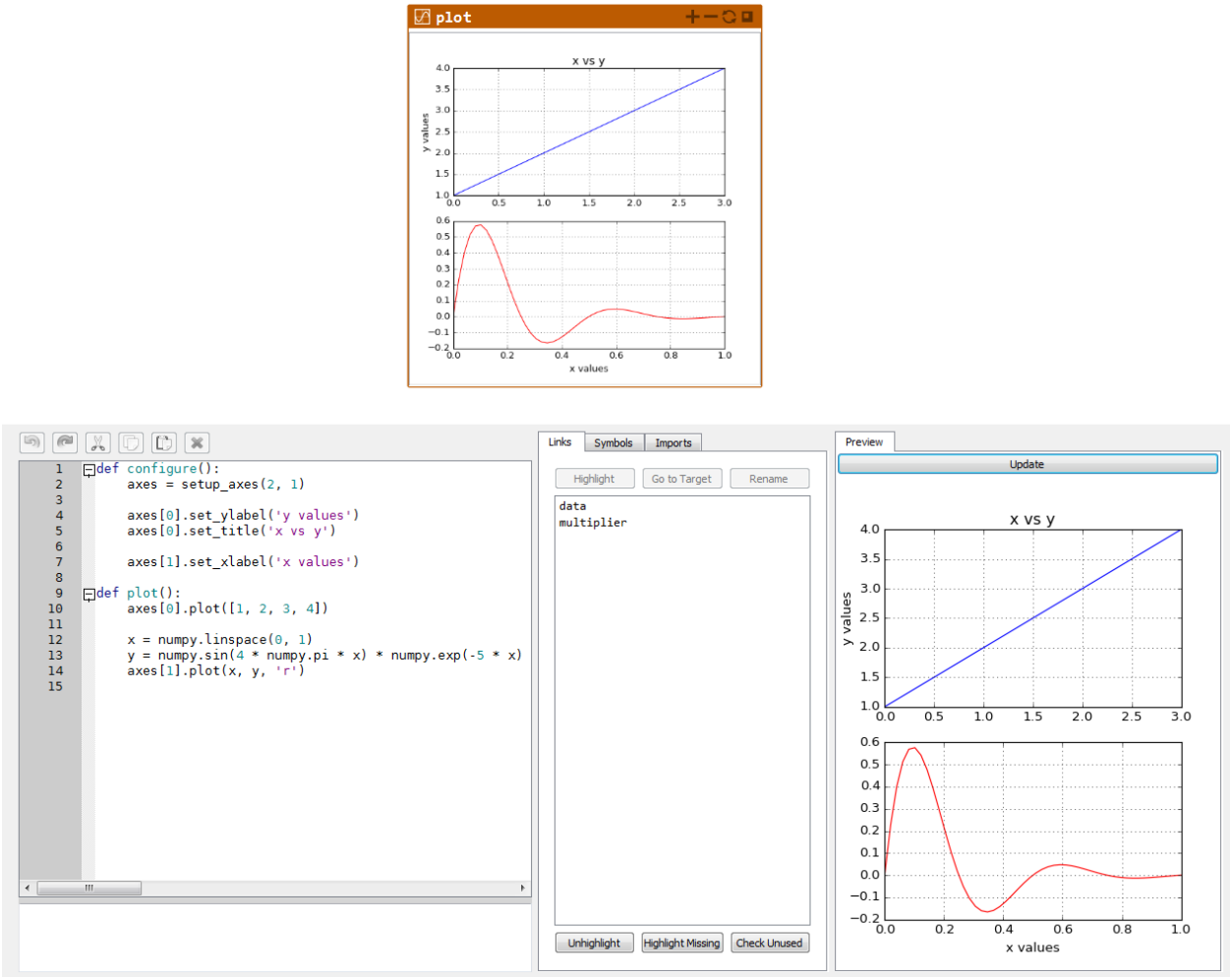


Fig. 5.13: The Plot part and editor dialog.

triggers execution of their respective parts. The pulse also adds a new event to the queue corresponding to the next pulse.

Double-clicking on the pulse part's frame or right-clicking and selecting "Edit..." opens an editor dialog (see *Part Editing, and Undo, Redo Commands*) that displays the content shown in Figure 5.14, that allows the properties of the part to be edited or viewed:

- the pulse period defined in terms of the number of weeks, days, hours, minutes, and seconds of the pulse frequency,
- the pulse state which can be either 'Active' or 'Inactive',
- the pulse priority which is specified as a floating point value ranging from 0.0 to a maximum of one million.

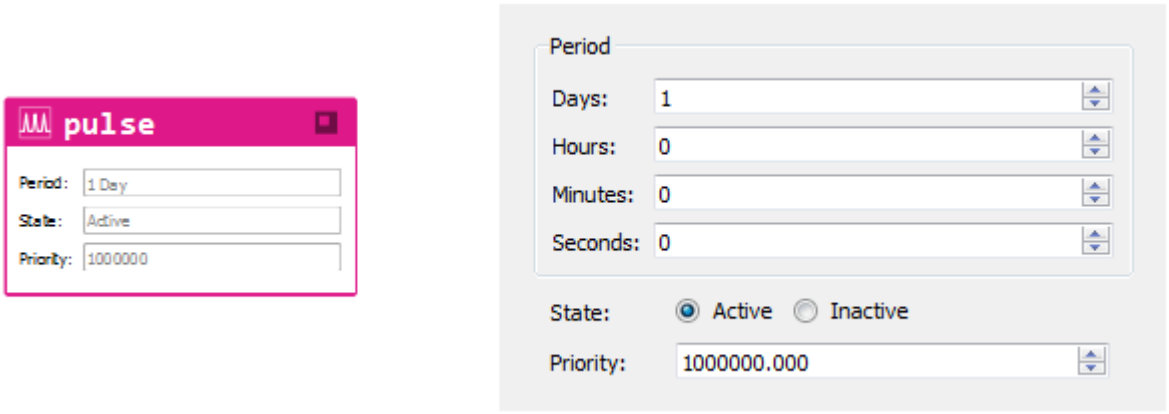


Fig. 5.14: The Pulse part and editor dialog.

Clicking on Part Help opens this page.

*Using Parts*

*Table of Contents*

### 5.1.15 Sheet Part

The sheet part is used to display and manipulate two-dimensional grids of cells similar to a spreadsheet. Cells within the sheet can be indexed by wiring to the sheet from a function part and using 0-based array indexing. For example, a function part that is linked to a sheet part with alias "sheet" can assign a new value to the cell at row 0 and column 2 with the following statement:

*link.sheet[0,2] = float(value)*,

where *value* is a parameter of the function part. Python slice notation can be used to select ranges within the sheet. For example,

*link.sheet[:2,:2] = link.sheet[:2,8:10]*,

will copy the 2x2 range from one area of the sheet to another. By default, sheet cells can also be indexed using *MS* Excel-style column-letter, row-number ranges. For example, the first-row, third-column would be indexed using *link.sheet['C1']*. Excel-style ranges, e.g. *link.sheet['A1:C8']* are also supported. Sheet columns can also be given optional custom names which may be used in place of the column letter. The "set_col_name()" function can be used set a column name. For example, if a sheet is given a custom column named "rank", the fifth element within with this column can be indexed using *link.sheet['rank5']*. Array-based indexing is enabled with the *API* command *link.sheet.set_index_style('array')*. This can be restored to Excel style with the call *link.sheet.set_index_style('excel')*.

Double-clicking the sheet part or right-clicking and selecting "Edit..." opens the editor dialog (see *Part Editing, and Undo, Redo Commands*) that displays the content shown in Figure 5.15, where the data in the sheet part can be manually edited. By clicking into the sheet cells the values can be changed directly. The dialog provides the following data editing options:

- Edit tab: tools to insert before or after (row if row selected, or column if column selected), select all, cut, copy, paste, and delete,
- Excel tab: tools for importing and exporting,
- Size tab: tools for changing sheet dimensions.

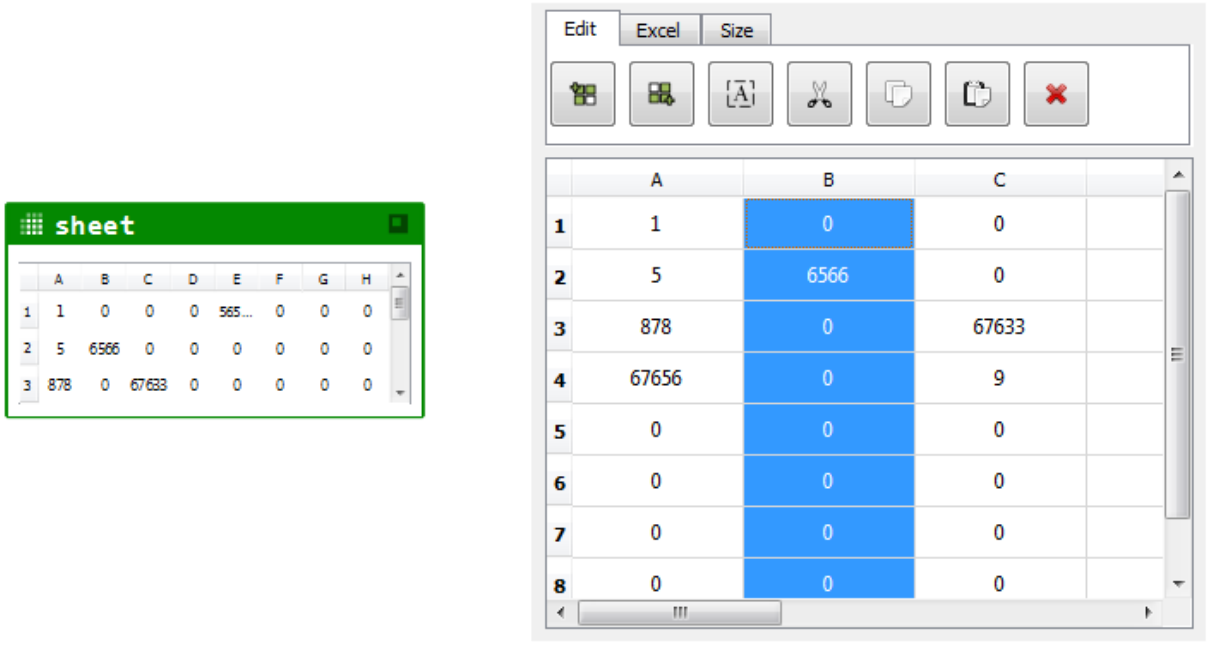


Fig. 5.15: The Sheet part and editor dialog.

Edit tab options become available or are disabled depending on how the sheet's cells, rows, or columns are selected. If nothing is selected in the sheet, all Edit tab options, except *Select All* are disabled. The following list overviews which Edit options become available depending on what is selected in the sheet:

- *Insert Before* or *Insert After*: are active only if an entire row or entire column is selected. Rows and/or columns can be selected by either clicking the row number or column header, respectively, or left-clicking and dragging the mouse along the row or column to select.
- *Cut*, *Copy*, *Paste*, *Delete*: are active if at least one cell has been selected and will also work on full or partial row or column selections. These options work on the contents of the cells. However, the exception is the *Delete* button, which will remove the selected row(s) or column(s), if a full row or set of rows, or a full column or set of columns, is selected. Otherwise, only the cell contents will be deleted (deleted cells have zeros inserted).
- *Select All*: always active.

The Excel tab provides options for importing from, and exporting to, *MS* Excel spreadsheets. These options are also available from the part's context menu.

- To import, use the dialog to select an Excel spreadsheet, and click the *List Sheets* button to list its sheets. Select a sheet to import from the drop down menu. Additionally, an optional range can be input using standard Excel notation, i.e. A1:C3 specifies column contents from A to C and row contents from 1 to 3. If you click OK,

the editor's current sheet data will be completely discarded, and the sheet populated with data from the chosen Excel spreadsheet. Both Excel *XLS* and *XLSX* formats are supported for import.

- To export, use the dialog to select a new or existing Excel spreadsheet to export to. If the spreadsheet exists, a list of its existing sheets will be displayed that allow a specific sheet to be the target of the export. A new sheet can also be selected from the list and will appear by default at the top of the list of sheets with the moniker "New". The exported contents will be inserted into the Excel sheet at cell A1 by default. The export range specifies where in the Excel sheet the sheet part's data should be inserted and can therefore be used to avoid overwriting data in existing sheets. By default, the entire contents of the sheet part will be exported, however, the range can also be used to clip the sheet part's data since only the part data that fits within the dimensions specified by the export range will be exported (the sheet data that is exported always starts at the top left and moves right and down to the end of the range). Only Excel *XLS* format is supported for export.

The Size tab provides two spin boxes to increase or decrease the sheet dimensions by using either the up or down scroll buttons on the right of the box or by typing the new value directly. The sheet dimensions will change when either *Enter* or *Return* are pressed or the mouse clicks elsewhere in the editor. **Note that if the row or column size is reduced, the contents of those cells are lost unless the editor panel is closed or cancelled and changes discarded. The values of removed rows or columns cannot be retrieved by increasing the size dimension after it has been decreased**.

Clicking on Part Help opens this page.

*Using Parts*

*Table of Contents*

### 5.1.16 SQL Part

The *SQL* part contains a structured query language (*SQL*) script. Code completion reveals the contents of the namespace as the user writes the script. The *SQL* command acts on table parts that the *SQL* part is linked to using the link aliases. The *SQL* part also has parameters that can be used in the *SQL* command. For example, consider a *SQL* part script that has a parameter "passed_country" and is linked to a table part with alias "table" and field (column name) "Country", then the *SQL* part could execute the *SQL Select* command as follows:

*Select * from {{link.table}} where Country = {{passed_country}}.*

The code referenced inside the double curly braces is interpreted as Python code which is useful for:

1. Calling other linked parts (by using the usual link.<link_name>), and
2. Referencing script parameters.

The *SQL* part can be run directly by pressing the run button in the upper-right corner of the part's frame. If there are parameters defined for the part, a dialog will open and prompt the user for the parameter values. The *SQL* part can also be run automatically by linking it to a function part that can call the *SQL* part's alias to execute the script. Additionally, the *SQL* can be linked to data or function parts which allows the part to use the alias to reference stored data or call function methods within the *SQL* statement.

An *SQL* part can also be linked to other *SQL* parts to create nested queries. In this case, the nested *SQL* part can be referenced as if it were a table. The syntax allows for parameters to be passed into the *SQL* part. As an example, a *SQL* part with alias "sql_one" is linked to a second *SQL* part with alias "sql_two", which in turn is linked to a table part with alias "table". The *SQL* statement in "sql_one" makes a query for people that are less than a specified age by querying the second *SQL* part:

*select * from {{link.sql_two("Simpson")}} where Age < {{p_max_age}}.*

The *SQL* statement in "sql_two" makes a query for a given surname by querying the table part:

*select * from {{link.table}} where Surname = {{p_surname}}.*

The parameters *p_max_age* and *p_surname* must be passed to the *SQL* parts when they are called by either another *SQL* part or a function part that is calling it. In this example, the *sql_two("Simpson")* portion of the *SQL* command will resolve to the result of the previous query passing the surname "Simpson" as a parameter. The net result of the second query is to return all records with the surname "Simpson" and whose age is less than or equal to *p_max_age*.

If a function part is linked to an *SQL* part, it can execute the *SQL* command by calling the alias of the link as a function with parameters. Calling an *SQL* part as a function from a function part will return the query result *SQL* data set.

Double-clicking on the *SQL* part's frame or right-clicking and selecting "Edit..." opens an editor dialog (see *Part Editing, and Undo, Redo Commands*) that displays the content shown in Figure 5.16, that allows various properties of the part to be edited or viewed:

- parameter specification,
- script editing tools for undo, redo, cut, copy, paste, and delete,
- a script editor,
- a *Links* and *Symbols* panel that provides link and object import management options. See section *Script-Base Parts: Links, Symbols, and Imports* for more information on these panels. Note that for symbols, if the cursor is inserted in between double curly braces (signifying Python code) then the symbols list will change to Python symbols and modules,
- a SQL preview panel that displays a preview (up to 100 rows) of the data returned when the *Refresh* button is pressed.

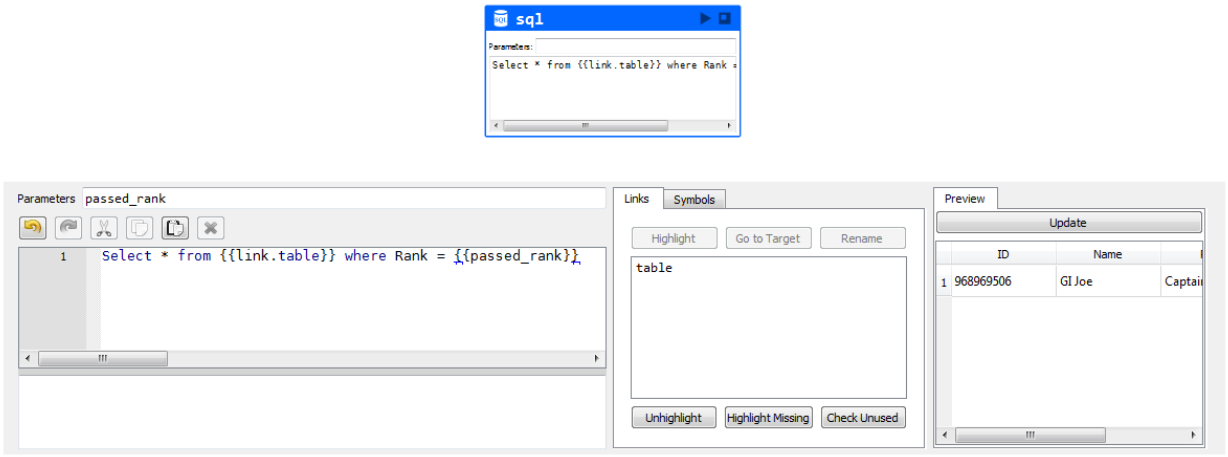


Fig. 5.16: The SQL part and editor dialog.

If parameters are specified in the editor, the *SQL* part will open a prompt every time the *SQL* part is run to allow the user to enter the parameter values. These values must be valid Python expressions. e.g. a string must include quotes.

For a discussion on how to use the *Links* tab, refer to the corresponding section in the *Function Part Reference*.

Clicking on Part Help opens this page.

*Using Parts*

*Table of Contents*

### 5.1.17 Table Part

The table part is used to display, edit, and interact with the contents of a table in an embedded relational database. The table data can be accessed and changed by using either the table *API* via a linked function part or through the

part's editor panel. The embedded database is managed by SQLite statements that are generated by table part *API* commands or user interactions with the editor panel. These commands allow the user to import from, or export to, Access database tables. They also provide an interface that can sort, filter, index, add, and remove database records and data fields. The table part also stores the names and data-types for the table fields.

Double-clicking the table part or right-clicking and selecting "Edit..." opens the editor dialog (see *Part Editing, and Undo, Redo Commands*) that displays the content shown in Figure 5.17, where the data in the table part can be manually edited. By clicking into the table fields the values can be changed directly.

Double-clicking a column header will open a dialog that can be used to change the column's name and data type. The supported types include double, integer, and text. The column type can be changed at any time and automatically convert the current column values. Likewise, any time a value is manually entered into a column, the value is converted to the type specified by the column. If the values cannot be converted, another dialog will open stating that a "Data Type Error" occurred which means that current value(s) do not match the type expected by the column. These values will be highlighted in red and must be manually changed. While Origame will continue to funtion without issue if they are not changed, an export error may occur if the table contents are later exported to an Access database. Copying and pasting values between columns will also automatically convert the copied values to the type specified by the column.

The dialog provides the following data editing options:

- Edit tab: tools to insert before or after (row if row selected, or column if column selected), select all, cut, copy, paste, delete, and sort,
- Database tab: tools for filtering, importing and exporting, and indexing,
- Size tab: tools for changing table dimensions.

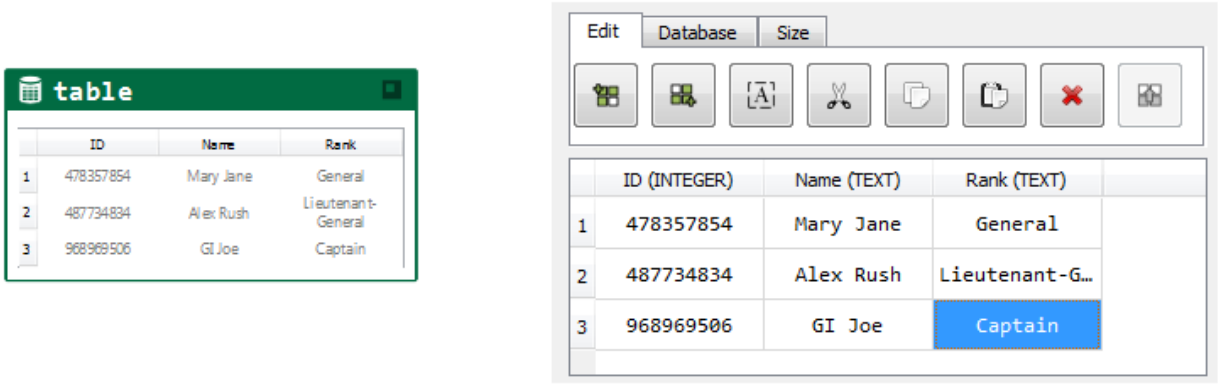


Fig. 5.17: The Table part and editor dialog.

Edit tab options become available or are disabled depending on how the table's fields, rows, or columns are selected. If nothing is selected in the table, all Edit tab options, except *Select All* are disabled. The following list overviews which Edit options become available depending on what is selected in the table:

- *Insert Before* or *Insert After*: are active immediately and, for new tables without rows or columns, will insert one row at a time only. Once a table has been initialized with a set of rows, these buttons will insert a row if a row is selected or a column if a column is selected (or group of either) before or after the selected one. Rows and/or columns can be selected by either clicking the row number or column header, respectively, or left-clicking and dragging the mouse along the row or column to select. These buttons will deactivate if no cell, row, or column are selected. However, they automatically reactivate if all rows are removed in order that they may be used to insert new rows.
- *Cut*, *Copy*, *Paste*, *Delete*: are active if at least one cell has been selected and will also work on full or partial row or column selections. While copy works on the contents of the row(s) or column(s) selected, cut will remove the selected row(s) or column(s) entirely. The delete option will do the same things as cut if a full row or set

of rows, or a full column or set of columns, is selected, otherwise, only the cell contents will be deleted. For copied contents, paste will paste the copied contents into the table (the dimensions of table must permit the entire copied selection to be pasted) starting at the location of the top left selected cell. For cut row(s) and column(s), paste will insert the cut row(s) or column(s) into the table at the selected row or column.

- *Sort*: is active only when an entire column is selected. Clicking *Sort* activates the sorting feature so that subsequently clicking on a column will toggle the sort from "as-created", to "alphabetical", to "reverse-alphabetical". If the *Sort* button is clicked again, sorting will be disabled and the column will return to "as-created". Sorted columns can be propagated to the table part in the Model View if editor changes are applied while sorting is active.
- *Select All*: always active.

While all rows may be removed from a table, it must always have one column header. If all columns are selected and Delete is pressed, all table contents will be deleted, however, the default column header will be reinserted as the first column.

The Database tab also provides the following options:

- The Filter button in the Database tab allows SQLite filter to be specified to limit the records shown in the table in accordance with the filter specified. For example, to filter all record entries for a specific column header called "Population", to a specific value, click the Filter button to open the dialog and enter the filter string that specifies which column and what value to filter for: *Population='value'*. The Refresh filter button can be pressed to reapply the filter if new records are added or existing records changed that do not satisfy the filter. The filter is applied only to the data in the editor until either the *OK* or *Apply* buttons are pressed, at which point, the filter is applied to the table part.
- The Import and Export buttons provide a dialog to facilitate importing from, and exporting to, a table in an Access database file. When importing, data will be imported into the editor panel but won't be propagated to the table part until *OK* or *Apply* is pressed. When exporting, the data in the editor will be exported to Access, not the data in the table part (which will be the same unless changes have been made to the data after the editor was opened). These options are also available from the part's context menu. When importing or exporting from the context menu, the data in the table part is affected directly. Only Access *MDB* and *ACCDB* files are supported for both import and export operations, however, only *MDB* files can be created during export if a new database file is created.
  - To import, use the dialog to select a database, and click the *List Tables* button to list its tables. Selecting a table from the drop down menu will show a list of fields in the table which can be selected or deselected for import. Additionally, an SQL "WHERE" clause can be entered in the last field in order to filter the results that are imported (the clause does not require the SQL statement "SELECT something FROM table WHERE...", only the conditions required to import specific values from specific columns). If you click OK, the editor's current table data will be completely discarded, the chosen fields are added, and the table populated with data from the chosen MS Access database.
  - To export, use the dialog to select an MS Access database to export to. If the database exists, you may also list its tables and select the table to export to. The default table shown in the list is a new table that will be created in the database. If an existing table is selected, its contents will be replaced with the exported table upon completion of the export operation. There is no way of appending data to an existing MS Access table. You may also select or deselect the fields of Table Part Editor to export. Note that if a filter has been added to the table part editor, then it will be shown in the dialog as a non-editable field and will be applied to the exported table.
- The Index button opens a dialog that enables column indexing. The dialog allows the user to build a list of indexes, where each row in the list represents a new index, and each index is a list of column names. The dialog facilitates adding and removing indexes. After the index is created, the indexed columns appear with an asterisk (*) inserted to the right of the column name.

The Size tab provides two spin boxes to increase or decrease the table dimensions by using either the up or down scroll buttons on the right of the box or by typing the new value directly. The table dimensions will change when either *Enter*

or *Return* are pressed or the mouse clicks elsewhere in the editor. **Note that if the row or column size is reduced, the contents of those fields are lost unless the editor panel is closed or cancelled and changes discarded. The values of removed rows or columns cannot be retrieved by increasing the size dimension after it has been decreased**.

Clicking on Part Help opens this page.

*Using Parts*

*Table of Contents*

### 5.1.18 Time Part

The time part is used to track the elapsed time since the time is last reset.

Double-clicking on the time part's frame or right-clicking and selecting "Edit..." opens an editor dialog (see *Part Editing, and Undo, Redo Commands*) that displays the content shown in Figure 5.18, that allows the elapsed time expressed in the days, hours, minutes, and seconds to be edited or viewed.

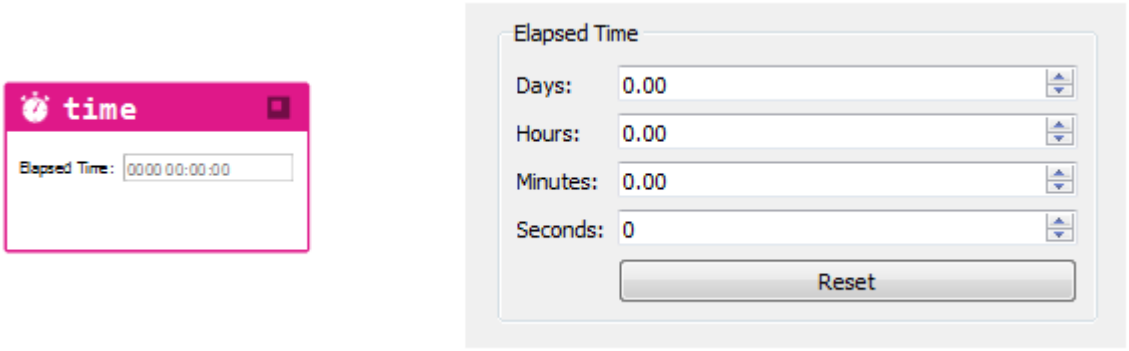


Fig. 5.18: The Time part and editor dialog.

Clicking on Part Help opens this page.

*Using Parts*

*Table of Contents*

### 5.1.19 Variable Part

The variable part is used to create and set model variables. The part's editor dialog provides a string input field that allows the user to set a new value for the variable part. For example, to a create a Python list in the variable part, the user would enter the string *['one', 'two', 'three']* in the input field. The string is then evaluated to create the Python object and store it in the variable part. The variable part can contain any valid Python object.

Double-clicking on the variable part's frame or right-clicking and selecting "Edit..." opens an editor dialog (see *Part Editing, and Undo, Redo Commands*) that displays the content shown in Figure 5.19, that allows the string value to be edited or viewed.

Clicking on Part Help opens this page.

*Using Parts*

*Table of Contents*


Fig. 5.19: The Variable part and editor dialog.

## 5.2 Programming Reference

The following *API* are globally available directly from the Function part. Some are also available to the Library, SQL, and Plot Parts.

**self.<part_api>**

- Description: This Part's functionality.
- Parameters: None.
- Returns: None.

**link.<part_alias>.<part_api>**

- Description: Access the methods, attributes, and properties of all linked parts.
- Parameters: None.
- Returns: None.

**new_part(<part_type>, [optional] <name>, [optional] <position>)**

- Description: Creation of new linked part.
- Parameters: The [str] part type, [str] part name (default=None), and [tuple] part position (default=None); e.g. <position> =(x, y, z)
- Returns: A [BasePart] new part of type <part_type>.

**del_part(<part_frame>)**

- Description: Remove the linked part.
- Parameters: The part frame expressed as either a [PartFrame] or [str]. e.g. The PartFrame can be obtained from the link by the API link._part_alias_ where "_part_alias_" is the link name pointing to the part to remove.
- Returns: None.

**get_script()**

- Description: Get the script from the part.
- Parameters: None.
- Returns: The part's [str] script.

**set_script(<script>)**

- Description: Set the part's script.
- Parameters: The part [str] <script>.
- Returns: None.

**new_link(<from_part_frame>, <to_part_frame>, [optional] <link_name>)**

- Description: Link a part to another.
- Parameters: The "from" and "to" part frames can be expressed as either a [PartFrame] or [str], a [str] link name (default=None). The PartFrame can be obtained from the link by the API link._part_alias_.
- Returns: None.

**del_link(<link_name> or <from_part_frame>, <to_part_frame>)**

- Description: Unlink two parts.
- Parameters: The [str] link name or [PartFrame] of the source (from) part, the [PartFrame] part frame of the target part. The PartFrame can be obtained from the link by the API link._part_alias_.
- Returns: None.

**del_link_to(<to_part_frame>)**

- Description: Unlink this part frame from the <to_part_frame>.
- Parameters: The [PartFrame] part frame of the target part. The PartFrame can be obtained from the link by the API link._part_alias_.
- Returns: None.

**delay(years=<years>, months=<months>, days=<days>, hours=<hours>, minutes=<minutes>, seconds=<seconds>)**

- Description: Returns the current simulation time plus the specified time. Specify at least one date or time keyword parameter to calculate a delay using date and/or time. All unused parameters default to zero.
- Parameters: The [float] delay specified with at least one of: years=, months=, days=, hours=, minutes=, seconds=.
- Returns: The [float] time in days corresponding to the current time plus the specified time delta.

**log.critical("<message>")** or **log.error("<message>")** or **log.warning("<message>")** or **log.info("<message>")** or **print("<message>")**

- Description: Message log entry creation. Log messages to the correct log type (e.g. 'critical', 'error', etc.).
- Parameters: [str] a <message> to log to the Log Panel and application log file.
- Returns: None.

**signal(<link.part_alias>, [optional] args=<(arg1, arg2, ...)>, [optional] time=<time>, [optional] priority=<priority>)**

- Description: Place an event on the Event Queue with arguments, to execute at the specified <time>, and with the specified <priority>.
- Parameters: [Part Exec] executable part, [tuple] args (default=None), [float] time in days (default=None), [float] priority (default=0.0).
- Returns: None.

**sim.<sim_state_control>**

- Description: Set simulation states or access and set simulation parameters.
- **Parameters: <sim_state_control> specifies what simulation state to set or parameter to access. These include:**
  - **Simulation state controls:**
    - *pause()*: Pause simulation; works even if already paused.

        - *resume()*: Resume simulation; works even if already running.
    - **Property 'getters':**
        - *replication_id*: Get the replication *ID*, for current variant. Smallest allowed is 1.
        - *variant_id*: Get the variant *ID*. Smallest allowed is 1.
        - *replication_folder*: Get the folder used by this simulation replication.
        - *sim_time_days*: Get the current simulation time, in days.
    - **Property 'getters' and 'setters':**
        - *runtime_animation*: Get or set the current animation setting.
        - *realtime_mode*: Get or set the real-time mode.
        - *realtime_scale*: Get or set the scale for real-time mode.
- Returns: See respective sim call listed in *Parameters*.

**get_scenario_name()**

- Description: Get the name of the scenario ORI file, or None if not saved.
- Parameters: None.
- Returns: The [str] name of the scenario.

**get_scenario_path()**

- Description: Get the path to the scenario ORI file, or None if not saved.
- Parameters: None.
- Returns: The [str] path of the scenario (not including file name).

Part-specific *API* is accessed using the following syntax:

*part.scripting_api*(*args, *kwargs)

where *part* is the Part object upon which the *API* is being called, and *scripting_api* is the specific scripting *API* invoked on the part.

A Part can be accessed from the Function Part script as follows:

- link.<*part_alias*>: for parts connected by links to the Function Part, where *part_alias* is the alias of the connected part and the name shown on the connecting link.
- link.hub.hub.<*any number of hubs*>.<*part_alias*>: a "chained link" links one part to another through a series of hub parts. The target part can be accessed by its alias from the last hub in the chain.
- <*part_object*>: for parts created inside the Function Part (e.g. by *new_part*) and stored in the object name *part_object*.

The Part Frame contains additional *API* methods which can be accessed by referencing the 'link._' followed by a linked part name:

- link.<*_part_alias_*>: for parts connected by links to the Function Part.
- link.hub.hub.<*any number of hubs*>.<*_part_alias_*>: a "chained link" links one part to another through a series of hub parts. The target part frame can be accessed by its alias from the last hub in the chain.
- <*part_object*>.part_frame: for parts created inside the Function Part.

The *API* commands available from the part frame include:

**get_detail_level()**

- Description: Get the detail level. Note: the return value is not affect by the Model View's Detail Level Override setting.
- Parameters: None.
- Returns: The [DetailLevelEnum] true detail level.

**set_detail_level(<value>)**

- Description: Set the detail level.
- Parameters: A new [DetailLevelEnum] detail level <value>.
- Returns: None.

**get_pos_x()**

- Description: Get the x–frame position (top–left corner) in global scenario coordinates.
- Parameters: None.
- Returns: An [float] x–position value along the respective axis.

**get_pos_y()**

- Description: Get y–frame position (top–left corner) in global scenario coordinates.
- Parameters: None.
- Returns: A [float] y–position value along the respective axis.

**set_pos_x(<x>)**

- Description: Set x–frame position (top–left corner) in global scenario coordinates.
- Parameters: An [float] <x> position value along the respective axis.
- Returns: None.

**set_pos_y(<y>)**

- Description: Set y–frame position (top–left corner) in global scenario coordinates.
- Parameters: A [float] <y> position value along the respective axis.
- Returns: None.

**get_name()**

- Description: Get part name.
- Parameters: None.
- Returns: A [string] value of the part's name.

**set_name(<name>)**

- Description: Set part name.
- Parameters: A [string] <name> to set.
- Returns: None.

The following sections describe the scripting *API* available from each specific Part:

### 5.2.1 Actor Part

Each Actor Part exposes the following functionality to scripting *API* by using link.<*part_alias*>.<*scripting_api*>, where <*part_alias*> is the link name, and <*scripting_api*> are the methods listed below:

**image_id**

- Property: Get the ID of the image associated with this Actor part. e.g. id = actor.image_id.

**get_image_id()**

- Description: Get the ID of the image associated with this Actor part.
- Parameters: None.
- Returns: This [int] instance's image ID.

**image_path**

- Property: Get or set the image path used by this actor. e.g. path = actor.image_path or actor.image_path = path.

**get_image_path()**

- Description: Get the image path corresponding to the image ID stored by this part or None to indicate default image.
- Parameters: None.
- Returns: The [str] path to the associated image file or None.

**set_image_path(<path>)**

- Description: Sets the image associated with the Actor part.
- Parameters: The [str] image file path of the image.
- Returns: None.

**rotation_2d**

- Property: Get or set the 2D rotation angle. e.g. rot = actor.rotation_2d or actor.rotation_2d = rot.

**get_rotation_2d()**

- Description: Get the 2D rotation angle of the part. Positive is clockwise.
- Parameters: None.
- Returns: The [float] rotation angle.

**set_rotation_2d(<angle>)**

- Description: Set the 2D rotation angle of the part. Positive is clockwise.
- Parameters: The [float] rotation angle.
- Returns: None.

**num_children**

- Property: Get the number of child parts of this actor. e.g. num = actor.num_children.

*Part API*

*Table of Contents*

### 5.2.2 Button Part

Each Button Part exposes the following functionality to scripting *API* by using link.<*part_alias*>.<*scripting_api*>, where <*part_alias*> is the link name, and <*scripting_api*> are the methods listed below:

**get_button_action()**

- Description: Get button mode (momentary or toggle).
- Parameters: None.
- Returns: The [ButtonActionEnum] with values momentary=0 or toggle=1.

**set_button_action(<mode>)**

- Description: Set button mode (momentary or toggle).
- Parameters: The [ButtonActionEnum] <mode> with values momentary=0 or toggle=1.
- Returns: None.

**get_button_trigger_style()**

- Description: Get execution mode (3 options).
- Parameters: None.
- Returns: The [ButtonTriggerStyleEnum] with value on_press=0, on_release=1, or on_press_and_release=2.

**set_button_trigger_style(<style>)**

- Description: Set execution mode (3 options).
- Parameters: The [ButtonTriggerStyleEnum] <style> with value on_press=0, on_release=1, or on_press_and_release=2.
- Returns: None.

**get_image_pressed_path()**

- Description: Get icon image path and file name (released image or pressed image).
- Parameters: None.
- Returns: The [str] path to the associated image file.

**set_image_pressed_path(<path*>)**

- Description: Set icon image path and file name (released image or pressed image).
- Parameters: The [str] <path> to the associated image file.
- Returns: None.

**get_image_released_path()**

- Description: Get icon image path and file name (released image or pressed image).
- Parameters: None.
- Returns: The [str] path to the associated image file.

**set_image_released_path(<path>)**

- Description: Set icon image path and file name (released image or pressed image).
- Parameters: The [str] <path> to the associated image file.
- Returns: None.

**get_rotation_2d_pressed()**

- Description: Get the rotation angle of the button part in the 2D view.
- Parameters: None.
- Returns: The [float] angle of the button part.

**set_rotation_2d_pressed(<angle>)**

- Description: Set the rotation angle of the button part in the 2D view.
- Parameters: The [float] <angle> of the button part.
- Returns: None.

**get_rotation_2d_released()**

- Description: Get the rotation angle of the button part in the 2D view.
- Parameters: None.
- Returns: The [float] angle of the button part.

**set_rotation_2d_released(<angle>)**

- Description: Set the rotation angle of the button part in the 2D view.
- Parameters: The [float] <angle> of the button part.
- Returns: None.

*Part API*

*Table of Contents*

### 5.2.3 Clock Part

The clock part has been deprecated and can no longer be created in new scenarios but continues to be fully supported in previously built scenarios. For alternative time options see *Datetime part* and *Time part*.

Each Clock Part exposes the following functionality to scripting *API* by using link.<*part_alias*>.<*scripting_api*>, where <*part_alias*> is the link name, and <*scripting_api*> are the methods listed below:

**delay(ticks=<ticks>)** or **delay(<ticks>)**

- Description: Get the simulation time corresponding to a specified delay using tick time. Invoke this method with just one parameter, with or without the keyword *ticks*.
- Parameters: The [float] delay in <ticks> specified with or without the keyword: ticks=.
- Returns: The [float] time in days corresponding to the current time plus the specified time delta.

**delay(years=<years>, months=<months>, days=<days>, hours=<hours>, minutes=<minutes>, seconds=<seconds>)**

- Description: Get the simulation time corresponding to a specified delay using date and time parameters. Specify at least one date or time keyword parameter to calculate a delay using date or time. All unused parameters default to zero. If only one parameter is used without a keyword, the *API* assumes ticks are specified (see previous *delay* use case).
- Parameters: The [float] delay specified with at least one of: years=, months=, days=, hours=, minutes=, seconds=.
- Returns: The [float] time in days corresponding to the current time plus the specified time delta.

**date_time**

- Property: Get or set the date_time used by this clock. e.g. date_time = clock.date_time or clock.date_time = date_time.

**get_date_time()**

- Description: Get the current data/time setting for the clock part.
- Parameters: None.
- Returns: The [datetime] object of the clock.

**set_date_time(<date_time>)**

- Description: Get the current data/time setting for the clock part.
- Parameters: The [datetime] <date_time> object of the clock.
- Returns: None.

**second**

- Property: Get or set the seconds used by this clock. e.g. seconds = clock.second or clock.second = seconds.

**get_second()**

- Description: Get the second portion of the clock's calender date/time.
- Parameters: None.
- Returns: The [int] second value of the clock.

**set_second(<second>)**

- Description: Set the second portion of the clock's calender date/time.
- Parameters: The [int] <second> value of the clock.
- Returns: None.

**minute**

- Property: Get or set the minutes used by this clock. e.g. mins = clock.minute or clock.minute = mins.

**get_minute()**

- Description: Get the minute portion of the clock's calender date/time.
- Parameters: None.
- Returns: The [int] minute value of the clock.

**set_minute(<minute>)**

- Description: Set the minute portion of the clock's calender date/time.
- Parameters: The [int] <minute> value of the clock.
- Returns: None.

**hour**

- Property: Get or set the hours used by this clock. e.g. hours = clock.hour or clock.hour = hours.

**get_hour()**

- Description: Get the hour portion of the clock's calender date/time.
- Parameters: None.
- Returns: The [int] hour value of the clock.

**set_hour(<hour>)**

- Description: Set the hour portion of the clock's calender date/time.
- Parameters: The [int] <hour> value of the clock.
- Returns: None.

**day**

- Property: Get or set the days used by this clock. e.g. days = clock.day or clock.day = days.

**get_day()**

- Description: Get the day portion of the clock's calender date/time.
- Parameters: None.
- Returns: The [int] day value of the clock.

**set_day(<day>)**

- Description: Set the day portion of the clock's calender date/time.
- Parameters: The [int] <day> value of the clock.
- Returns: None.

**month**

- Property: Get or set the months used by this clock. e.g. months = clock.month or clock.month = months.

**get_month()**

- Description: Get the month portion of the clock's calender date/time.
- Parameters: None.
- Returns: The [int] month value of the clock.

**set_month(<month>)**

- Description: Set the month portion of the clock's calender date/time.
- Parameters: The [int] <month> value of the clock.
- Returns: None.

**year**

- Property: Get or set the years used by this clock. e.g. years = clock.year or clock.year = years.

**get_year()**

- Description: Get the year portion of the clock's calender date/time.
- Parameters: None.
- Returns: The [int] year value of the clock.

**set_year(<year>)**

- Description: Set the year portion of the clock's calender date/time.
- Parameters: The [int] <year> value of the clock.
- Returns: None.

**tick_period_days**

- Property: Get or set the tick period (in days) used by this clock. e.g. tick_period = clock.tick_period_days or clock.tick_period_days = tick_period.

**get_tick_period_days()**

- Description: Get the tick period in days for this instance's tick timer.
- Parameters: None.
- Returns: The [float] tick period duration in days.

**set_tick_period_days(<days>)**

- Description: Set the tick period in days for this instance's tick timer.
- Parameters: The [float] new period duration in <days> for a time tick.
- Returns: None.

**tick_value**

- Property: Get or set the tick value used by this clock. e.g. ticks = clock.tick_value or clock.tick_value = ticks.

**get_tick_value()**

- Description: Get the number of clock ticks.
- Parameters: None.
- Returns: The [float] number of ticks.

**set_tick_value(<ticks>)**

- Description: Set the number of clock ticks. The tick value can be negative.
- Parameters: The [float] number of <ticks> to set.
- Returns: None.

*Part API*

*Table of Contents*

### 5.2.4 Data Part

Each Data Part exposes the following functionality to scripting *API* by using link.<*part_alias*>.<*scripting_api*>, where <*part_alias*> is the link name, and <*scripting_api*> are the methods listed below:

**keys()**

- Description: Get all keys.
- Parameters: None.
- Returns: A [list] list of keys in the data part.

**values()**

- Description: Get all values.
- Parameters: None.
- Returns: A [list] of values in the data part.

**items()**

- Description: Get all the items of this part in OrderedDict format.
- Parameters: None.

- Returns: An [OrderedDict] of items in the data part.

**display_order**

- Property: Get or set the DisplayOrderEnum used by this data part. e.g. order = data.display_order or data.display_order = order.

**get_display_order()**

- Description: Get display order of keys.
- Parameters: None.
- Returns: [DisplayOrderEnum] the display order with values: of_creation (0), alphabetical (1), reverse_alphabetical (2).

**set_display_order(<order>)**

- Description: Set display order of keys.
- Parameters: The [DisplayOrderEnum] display <order> with values: of_creation (0), alphabetical (1), reverse_alphabetical (2).
- Returns: None.

The following *API* uses direct assignment to, or direct reference to, the linked part using link.<part_alias> notation:

**link.<part_alias>['<key>']=<value>**

- Description: Create a new key and set its value or set a new value to an existing key. Assigned directly to the part, shown here via link connection.
- Parameters: The [dict key] <key> and [object] <value> to set to the key.
- Returns: None.

**link.<part_alias>.<key>=<value>**

- Description: Set a new value to an existing key. Assigned directly to the part, shown here via link connection.
- Parameters: The [dict key] <key> and [object] <value> to set to the key.
- Returns: None.

**link.<part_alias>['<key>']** or **link.<part_alias>.<key>**

- Description: Get value of a key. Retrieved directly from the part, shown here via link connection.
- Parameters: [dict key] the key <key>.
- Returns: The [object] <value> of the key.

**del link.<part_alias>.<key>**

- Description: Delete a key. Removed directly from the part, shown here via link connection.
- Parameters: The [dict key] <key>.
- Returns: None.

**if key in link.<part_alias>**

- Description: Test if the part has a key.
- Parameters: None.
- Returns: None.

**for key in link.<part_alias>**

- Description: Loop over a part's keys.
- Parameters: None.
- Returns: None.

*Part API*

*Table of Contents*

### 5.2.5 Datetime Part

Each Datetime Part exposes the following functionality to scripting *API* by using link.<*part_alias*>.<*scripting_api*>, where <*part_alias*> is the link name, and <*scripting_api*> are the methods listed below:

**delay(years=<years>, months=<months>, days=<days>, hours=<hours>, minutes=<minutes>, seconds=<seconds>)**

- Description: Get the simulation time corresponding to a specified delay using date and time parameters. Specify at least one date or time keyword parameter to calculate a delay using date or time. All unused parameters default to zero. If only one parameter is used without a keyword, the *API* assumes ticks are specified (see previous *delay* use case).
- Parameters: The [float] delay specified with at least one of: years=, months=, days=, hours=, minutes=, seconds=.
- Returns: The [float] time in days corresponding to the current time plus the specified time delta.

**date_time**

- Property: Get or set the date_time used by this clock. e.g. date_time = clock.date_time or clock.date_time = date_time.

**get_date_time()**

- Description: Get the current data/time setting for the clock part.
- Parameters: None.
- Returns: The [datetime] object of the clock.

**set_date_time(<date_time>)**

- Description: Get the current data/time setting for the clock part.
- Parameters: The [datetime] <date_time> object of the clock.
- Returns: None.

**second**

- Property: Get or set the seconds used by this clock. e.g. seconds = clock.second or clock.second = seconds.

**get_second()**

- Description: Get the second portion of the clock's calender date/time.
- Parameters: None.
- Returns: The [int] second value of the clock.

**set_second(<second>)**

- Description: Set the second portion of the clock's calender date/time.
- Parameters: The [int] <second> value of the clock.

- Returns: None.

**minute**

- Property: Get or set the minutes used by this clock. e.g. mins = clock.minute or clock.minute = mins.

**get_minute()**

- Description: Get the minute portion of the clock's calender date/time.
- Parameters: None.
- Returns: The [int] minute value of the clock.

**set_minute(<minute>)**

- Description: Set the minute portion of the clock's calender date/time.
- Parameters: The [int] <minute> value of the clock.
- Returns: None.

**hour**

- Property: Get or set the hours used by this clock. e.g. hours = clock.hour or clock.hour = hours.

**get_hour()**

- Description: Get the hour portion of the clock's calender date/time.
- Parameters: None.
- Returns: The [int] hour value of the clock.

**set_hour(<hour>)**

- Description: Set the hour portion of the clock's calender date/time.
- Parameters: The [int] <hour> value of the clock.
- Returns: None.

**day**

- Property: Get or set the days used by this clock. e.g. days = clock.day or clock.day = days.

**get_day()**

- Description: Get the day portion of the clock's calender date/time.
- Parameters: None.
- Returns: The [int] day value of the clock.

**set_day(<day>)**

- Description: Set the day portion of the clock's calender date/time.
- Parameters: The [int] <day> value of the clock.
- Returns: None.

**month**

- Property: Get or set the months used by this clock. e.g. months = clock.month or clock.month = months.

**get_month()**

- Description: Get the month portion of the clock's calender date/time.
- Parameters: None.

- Returns: [int] month value of the clock.

**set_month(<month>)**

- Description: Set the month portion of the clock's calender date/time.
- Parameters: The [int] <month> value of the clock.
- Returns: None.

**year**

- Property: Get or set the years used by this clock. e.g. years = clock.year or clock.year = years.

**get_year()**

- Description: Get the year portion of the clock's calender date/time.
- Parameters: None.
- Returns: The [int] year value of the clock.

**set_year(<year>)**

- Description: Set the year portion of the clock's calender date/time.
- Parameters: The [int] <year> value of the clock.
- Returns: None.

*Part API*

*Table of Contents*

### 5.2.6 File Part

Each file Part exposes the following functionality to scripting *API* by using link.<*part_alias*>.<*scripting_api*>, where <*part_alias*> is the link name, and <*scripting_api*> are the methods listed below:

**filepath**

- Property: Get or set the file path of the file part. Returns an object of type 'Path' (see pathlib package documentation). e.g. filepath = file.filepath or file.filepath = filepath.

**get_filepath()**

- Description: Get the path set by the editor.
- Parameters: None.
- Returns: The Path object to the file or folder.

**set_filepath(<path>)**

- Description: Set the path to the file or folder.
- Parameters: The <path> (Path or string object) that specifies the path to the file or folder.
- Returns: None.

**is relative_to_scen_folder**

- Property: Get or set the flag indicating if the set path is relative to the scenario folder. e.g. is_relative = file.relative_to_scen_folder or file.relative_to_scen_folder = True.

**pathlib.Path APOI**

- The file part provides access to the public API of the Path class. For example, *read_text*, *write_text*, *parent*, *name*, *suffix*, *is_absolute*, *exists*, *joinpath*, *mkdir*, *cwd*, etc.

*Part API*

*Table of Contents*

### 5.2.7 Function Part

Each Function Part exposes the following functionality to scripting *API* by using link.<*part_alias*>.<*scripting_api*>, where <*part_alias*> is the wire name, and <*scripting_api*> are the methods listed below:

**run_roles**

- Property: Get or set the [RunRolesEnum] run <roles> of the function part. e.g. roles = function.run_roles or function.run_roles = roles.

**get_run_roles()**

- Description: Get the run roles of the function part.
- Parameters: None.
- Returns: A [Set[RunRolesEnum]] of run roles.

**set_run_roles(<Set[RunRolesEnum]>)**

- Description: Set run roles of the function part.
- Parameters: A [Set[RunRolesEnum]] of run roles.
- Returns: None.

*Part API*

*Table of Contents*

### 5.2.8 Hub Part

Each Hub Part exposes the following functionality to scripting *API*:

**link.<_hub_alias_>.create_link(<part_frame>, "<part_alias>")** or **<hub_object>.part_frame.create_link(<part_frame>, "<part_alias>")**

- Description: Create link from Hub to part.
- Parameters: [Part Frame] the <part_frame> to link to, [str] a link name <part_alias>.
- Returns: [PartLink] the new link object.

**link.<hub_alias>.<part_alias>.<part_api(*args, *kwargs)>**

- Description: Get a part linked from the Hub. The Hub can access the connected parts *API* methods and change the values of part parameters.
- Parameters: arguments *args* and keyword arguments *kwargs* for the relevant part *API* methods.
- Returns: the values or objects associated with the relevant part *API* methods.

*Part API*

*Table of Contents*

### 5.2.9 Information Part

Each Information Part exposes the following functionality to scripting *API* by using link.<*part_alias*>.<*scripting_api*>, where <*part_alias*> is the link name, and <*scripting_api*> are the methods listed below:

**text**

- Property: Get or set the <text> of the information part. e.g. text = info.text or info.text = text.

**get_text()**

- Description: Get Info Part text.
- Parameters: None.
- Returns: The [str] text displayed in the part.

**set_text(<text>)**

- Description: Set Info Part text.
- Parameters: The [str] <text> to display in the part.
- Returns: None.

*Part API*

*Table of Contents*

### 5.2.10 Library Part

As a scriptable part, each Library Part shares a global set of API that is discussed in *Programming Reference*. It has no no unique API.

*Part API*

*Table of Contents*

### 5.2.11 Multiplier Part

Each Multiplier Part exposes the following functionality to scripting *API*:

**link.<multiplier_alais>**

- Description: Call all Parts linked to the Multiplier. Use as the Part Exec in Function Part's *signal* to add multiple events to the queue.
- Parameters: None.
- Returns: None.

*Part API*

*Table of Contents*

### 5.2.12 Node Part

The Node Part does not have any scripting *API*.

*Part API*

*Table of Contents*

### 5.2.13 Plot Part

Each Library Part shares a global set of API that is discussed in *Programming Reference*. In addition, the Plot Part exposes the following functionality to scripting *API* by using link.<*part_alias*>.<*scripting_api*>, where <*part_alias*> is the link name, and <*scripting_api*> are the methods listed below:

**figure**

- Property: Get the pyplot figure instance of the plot part. e.g. fig = plot.figure.

**reset_fig(<raise_on_fail>)**

- Description: Clear the figure of all plots and rerun the script's configure() function. Does NOT run the script's plot() function (press the "Update the Plot" button).
- Parameters: [optional] Flag indicates if an exception should be raised if an error occurs while resetting the figure. Default is True.
- Returns: None.

**get_axes()**

- Description: Get one or more axes from the plot's figure.
- Parameters: None.
- Returns: A [OneOrMoreAxes] is composed of either a single pyplot.Axes or a List[pyplot.Axes].

**update_fig()**

- Description: Refresh the figure plot data.
- Parameters: None.
- Returns: None.

**export_fig(<filepath>, [optional] <resolution>, [optional] <format>)**

- Description: Export a snapshot of the plot.
- Parameters: The [str] file path including file name, [int] dots per inch resolution (default=200), [str] file format (default=PNG).
- Returns: None.

**export_data(<filepath>, [optional] <sheet>)**

- Description: Export the plot data.
- Parameters: The [str] file path including file name, optional [str] name of the sheet to export to.
- Returns: A [bool] result indicating if export was successful.

**Matplotlib plot methods: axes.plot(*args), axes.hist(*args), axes.pie(*args), axes.scatter(*args)**

- Description: Set chart type for each series.
- Parameters: None.
- Returns: None.

*Part API*

*Table of Contents*

### 5.2.14 Pulse Part

Each Pulse Part exposes the following functionality to scripting *API* by using link.<*part_alias*>.<*scripting_api*>, where <*part_alias*> is the link name, and <*scripting_api*> are the methods listed below:

**pulse_period_days**

- Property: Get or set the pulse <period> of the pulse part. e.g. period = pulse.pulse_period_days or pulse.pulse_period_days = period.

**get_pulse_period_days()**

- Description: Get the pulse period.
- Parameters: None.
- Returns: The [float] pulse period in units of days.

**set_pulse_period_days(<period>)**

- Description: Sets the pulse period.
- Parameters: The [float] pulse period in units of days.
- Returns: None.

**state**

- Property: Get or set the pulse <state> of the pulse part. e.g. state = pulse.state or pulse.state = state.

**get_state()**

- Description: Get the part's state.
- Parameters: None.
- Returns: The [PulsePartState] state of the part specified as the enumeration PulsePartState which can be either 'active' or 'inactive'.

**set_state(<state>)**

- Description: Sets the pulse period.
- Parameters: The [PulsePartState] state of the part specified as the enumeration PulsePartState which can be either 'active' or 'inactive'.
- Returns: None.

**priority**

- Property: Get or set the pulse <priority> of the pulse part. e.g. priority = pulse.priority or pulse.priority = priority.

**get_priority()**

- Description: Get the pulse priority.
- Parameters: None.
- Returns: The [float] pulse priority which is specified as a floating point value ranging from 0.0 to a maximum of one million.

**set_priority(<priority>)**

- Description: Sets the pulse priority.
- Parameters: The [float] pulse priority which is specified as a floating point value ranging from 0.0 to a maximum of one million.

- Returns: None.

*Part API*

*Table of Contents*

### 5.2.15 Sheet Part

Each Sheet Part exposes the following functionality to scripting *API* by using link.<*part_alias*>.<*scripting_api*>, where <*part_alias*> is the link name, and <*scripting_api*> are the methods listed below:

**sheet_data** or **get_sheet_data()**

- Description: Get a copy of the 2-D array of sheet part data.
- Parameters: None.
- Returns: A [List[List[Any]]] list of lists containing the sheet data where each sub-list contains the data for one row.

**num_rows** or **get_rows**

- Description: Get the number of rows.
- Parameters: None.
- Returns: The [int] number of rows.

**set_rows(<num_rows>)**

- Description: Set the number of rows.
- Parameters: The [int] number of rows to set. Default is zero.
- Returns: None.

**num_cols** or **get_cols()**

- Description: Get the number of columns.
- Parameters: None.
- Returns: The [int] number of columns.

**set_cols(<num_cols>)**

- Description: Set the number of columns.
- Parameters: The [int] number of columns to set. Default is zero.
- Returns: None.

**named_cols** or **get_named_cols()**

- Description: Get a map of custom column names to column indices.
- Parameters: None.
- Returns: A Dict[str, int] containing the custom names mapped to column indices.

**col_widths** or **get_col_widths()**

- Description: Get the list of column widths of the sheet part.
- Parameters: None.
- Returns: A List[int] list of column widths.

**index_style** or **get_index_style()**

- Description: Get the name of the current index style of the sheet part.
- Parameters: None.
- Returns: A [str] name of the current index style. Either 'excel' or 'array'.

**set_index_style()**

- Description: Set the name of the current index style of the sheet part.
- Parameters: A [str] name of the current index style. Either 'excel' or 'array'.
- Returns: None.

**fill(<callback>,<cell_range>)**

- Description: Iterates over the specified 'cell_range' passing each cell row/col pair to the callback function with the result being assigned to the corresponding row/col cell in this sheet instance.
- Parameters: A user-defined [method] <callback> that takes a sheet row index [int] and column index [int] as arguments and returns an object to be set into the specified row/col cell of the sheet, a <cell_range> that is any of the following: [str] or [int] or [tuple].
- Returns: None.

**find(<item>)**

- Description: Search for a value.
- Parameters: The [object] *item* to look for.
- Returns: The [int or tuple] row, or column, or (row, column) index of the found <item>.

**repr_data()**

- Description: Get string representation of the array.
- Parameters: None.
- Returns: A [str] tabular string representation of the sheet's data.

**transpose()**

- Description: Transpose the array, inverting rows and columns.
- Parameters: None.
- Returns: None.

**add_col(<col_index>)**

- Description: Create a column.
- Parameters: The [int] column index <col_index> at which to insert the column.
- Returns: None.

**add_cols(<num_cols>,[optional] <col_index>)**

- Description: Create multiple columns.
- Parameters: The [int] number of columns <num_cols> to add, the [int] <col_index> insertion point (default=*num_cols*).
- Returns: None.

**col(<col_index>)**

- Description: Get all values in specified column.

- Parameters: The [int] column index <col_index>.
- Returns: A [list] list of row values for the column.

**del_col_name(<name>,<col_index>)**

- Description: Unname a column.
- Parameters: The [str] column <name> to remove, the [int] <col_index> of the column.
- Returns: The new Excel column [str] label or None.

**delete_col(<col_index>)**

- Description: Delete a column.
- Parameters: The [int] <col_index> of the column to delete.
- Returns: None.

**delete_cols(<num_cols>,<col_index>)**

- Description: Delete multiple columns.
- Parameters: The [int] number of columns <num_cols> to delete, the [int] index <col_index> of the first column to delete.
- Returns: None.

**set_col_name(<col_index>,<col_name>)**

- Description: Name a column.
- Parameters: The [int or str] <col_index> (numeric or Excel-style letter) to be named, the [str] new column name <col_name>.
- Returns: None.

**add_row(<row_index>)**

- Description: Create a row.
- Parameters: The [int] row index <row_index> at which to insert the row.
- Returns: None.

**add_rows(<num_rows>,[optional] <row_index>)**

- Description: Create multiple rows.
- Parameters: The [int] number of rows <num_rows> to add, [int] index <row_index> insertion point (default=*num_rows*).
- Returns: None.

**row(<row_index>)**

- Description: Get all values in specified row.
- Parameters: The [int] row index <row_index>.
- Returns: A [list] list of column values for the row.

**delete_row(<row_index>)**

- Description: Delete a row.
- Parameters: The [int] <row_index> of the row to delete.
- Returns: None.

**delete_rows(<num_rows>,<row_index>)**

- Description: Delete multiple rows.
- Parameters: The [int] number of rows <num_rows> to delete, the [int] index <row_index> of the first row to delete.
- Returns: None.

**clear()**

- Description: Clear the data in the Sheet.
- Parameters: None.
- Returns: None.

**resize(<num_rows>,<num_cols>)**

- Description: Change the number of rows and columns in the Sheet.
- Parameters: The [int] new number of rows <num_rows> to set, [int] the new number of columns <num_cols> to set.
- Returns: None.

**set_data(<data>)**

- Description: Set all values (overwrite) in the Sheet.
- Parameters: The [SheetPart or TablePart or int or float or str or tuple or list] new data to put into the sheet.
- Returns: None.

**copyfrom(<other_sheet>)**

- Description: Copy data from another Sheet.
- Parameters: A [SheetPart] sheet part to copy.
- Returns: This [SheetPart] instance as a copy of <other_sheet>.

**read_excel(<xls_file>,<xls_sheet>,<xls_range>,[optional] <accept_empty_cells>)**

- Description: Import an Excel file into the Sheet.
- Parameters: The [str] file path, the [str] worksheet name, the [str] Excel-style sheet range, a [bool] option to accept empty cells (default=False and sets empty cells to 0, otherwise set True to accept empty cells).
- Returns: None.

**write_excel(<xls_file>,<xls_sheet>)**

- Description: Export the Sheet to an Excel file.
- Parameters: The [str] file path, the [str] worksheet name.
- Returns: None.

The following *API* uses direct assignment to, or direct reference to, the linked part using link.<part_alias> notation:

**link.<sheet_alias>[<row>,<col>]**

- Description: Get values from the Sheet.
- Parameters: The [int, int] <row> and <col> indexes.
- Returns: A [object] value at the index provided.

**link.<sheet_alias>[<row>,<col>]=<value>**

- Description: Set values in the Sheet.
- Parameters: The [int, int] <row> and <col> indexes and the [object] <value> to set.
- Returns: None.

**if link.<sheet_alias1> == link.<sheet_alias2>**

- Description: Test if two Sheet Parts contain the same value.
- Parameters: None.
- Returns: None.

**if link.<sheet_alias1> !=link.<sheet_alias2>**

- Description: Test if two Sheet Parts differ by at least one value.
- Parameters: None.
- Returns: None.

**for value in link.<sheet_alias>:**

- Description: Loop through all values in a Sheet by row.
- Parameters: None.
- Returns: None.

*Part API*

*Table of Contents*

### 5.2.16 SQL Part

A *SQL* part can be run by calling it from within the function part to which it is linked. It returns a *SQL* data set object which contains the result of the *SQL* query.

**link.<sql_alias>(<parameters>)**

- Description: Execute the query.
- Parameters: The [object] list of <parameters> required by the *SQL* part.
- Returns: A [object] *SQL* data set.

The *SQL* data set's structure is a grid consisting of a list of lists. Two methods can be directly invoked on the Sql data set object:

1. *get_table_name* which returns the name of the dynamically created table as a string,
2. *get_data* which returns a list of lists representing the *SQL* query result.

The *SQL* data set object can also be treated like other Python objects such as testing if something is **in** the object, testing equality or inequality, getting or setting a subset of data in the object via an index (the index take the following forms: [int], [str], [int, int], [int, slice], [slice, int], or [slice, slice]), iterating over items in the object, converting the data inside the object to a string using Python's *repr* method, and testing the length of the object using Python's *len* method (len will return the number of rows unless there is only one row, in which case it will return the number of of columns).

Each *SQL* Part exposes the following functionality to scripting *API* by using link.<*part_alias*>.<*scripting_api*>, where <*part_alias*> is the link name, and <*scripting_api*> are the methods listed below:

**sql_script**

- Property: Get or set the <script> used by this SQL part. e.g. script = sql.sql_script or sql.sql_script = script.

**get_sql_script()**

- Description: Get the *SQL* script.
- Parameters: None.
- Returns: The [str] script that defines the *SQL* statements.

**set_sql_script(<script>)**

- Description: Set the *SQL* script.
- Parameters: Parameters: The [str] <script> that defines the *SQL* statements.
- Returns: None.

*Part API*

*Table of Contents*

### 5.2.17 Table Part

Each Table Part exposes the following functionality to scripting *API* by using link.<*part_alias*>.<*scripting_api*>, where <*part_alias*> is the link name, and <*scripting_api*> are the methods listed below:

**database_table_name**

- Property: Get the name of the table in this instance of the SQL database of this table part. e.g. name = table.database_table_name.

**get_database_table_name()**

- Description: Get the name of the table of this instance in the SQLite database.
- Parameters: None.
- Returns: The [str] name of the database table.

**indices**

- Property: Get or set the name of the table in this instance of the SQL database of this table part. e.g. name = table.database_table_name.

**get_indices()**

- Description: Get the indexes and the associated columns.
- Parameters: None.
- Returns: A Dict[str, List[str]] dictionary containing key value pairs where the key is the index name and value is a list of columns.

**set_indices(<col_indices>)**

- Description: Set the indices on this table. Drops any previous indices.
- Parameters: A Dict[str, List[str]] <col_indices> to set.
- Returns: None.

**data**

- Property: Get or set the data of the table part. e.g. data = table.data or table.data = data.

**get_all_data(<flag_omit_rec_id>, <flag_apply_filter>)**

- Description: Get all of the data (records) in this Table Part.

- Parameters: A [bool] <flag_omit_rec_id> to specify if record ID's should not be included in the returned data, a [bool] <flag_apply_filter> to specify whether the current filter should be applied on the data.
- Returns: A List[Tuple[Either[str, int, float]]] of data, where each data record is a tuple containing values of type string, integer, and/or float.

**set_all_data(<data>)**

- Description: Set all of the data (records) in this Table Part. Replaces all current data from this instance of the table part with the new <data>.
- Parameters: A List[Tuple[Either[str, int, float]]] of <data> to set into the part. The list contains all records, where each record is a tuple containing values of type string, integer, and/or float.
- Returns: None.

**delete_data(<where>)**

- Description: Delete records given the optional <where> clause. If the <where> clause is not supplied, then this method will delete all data in the table. Otherwise, only the data matching the specified "where" criteria will be deleted.
- Parameters: An optional SQL [str] <where> clause to restrict the number of records removed.
- Returns: None.

**column_types**

- Property: Get the data types of columns in this Table Part. e.g. types = table.column_types.

**get_column_types()**

- Description: Get the data types of columns in this Table Part.
- Parameters: None.
- Returns: A List[str] containing the data types in this Table Part.

**column_names**

- Property: Get the names of the columns in this Table Part. e.g. names = table.column_names.

**get_column_names()**

- Description: Get the names of the columns in this Table Part.
- Parameters: None.
- Returns: A List[str] of column names.

**get_column_names_and_types()**

- Description: Get the columns and types in a list. i.e. ["Col1 varchar(200)", "Col2 real(100)", "Col3"].
- Parameters: None.
- Returns: A List[str] of column names and types.

**set_column_names_and_types(<names_and_types>)**

- Description: Get the names of the columns in this Table Part.
- Parameters: A List[str] <names_and_types> of column names and types.
- Returns: None.

**arrange_columns(<column_names>)**

- Description: Re-arrange the columns in this Table Part. If a column does not appear in <column_names> but is part of the current table schema, then the column and data is not fetched from this table.
- Parameters: A [str] of comma delimited <column_names>.
- Returns: None.

**add_column(<name>,[optional] <type>,[optional] <size>)**

- Description: Add one or more columns.
- Parameters: [str] the <name> of the column, [str] the <type> of column (default=None), [int] the <size> of the column (default=None).
- Returns: none.

**drop_column(<name>)**

- Description: Remove a column.
- Parameters: [str] the <name> of the column.
- Returns: none.

**rename_column(<current_name>,<new_name>)**

- Description: Change a field name.
- Parameters: [str] the <current_name>, [str] the <new_name>.
- Returns: none.

**set_column(<fields>)**

- Description: Add columns to this Table Part. The fields must be valid specifications accepted by the SQLite "CREATE TABLE" statement.
- Parameters: A [str] of <fields> expressed as a comma delimited list of column names and types. e.g. "ID INTEGER, Name TEXT, Age REAL...".
- Returns: None.

**filter_string**

- Property: Get and set the filter string for filter requests. e.g. filter = table.filter_string or table.filter_string = filter.

**filter(<table_filter>)**

- Description: Set the filter string to be applied when fetching table data in other calls such as get_all_data().
- Parameters: A [str] <table_filter> which must be a valid SQL filter string.
- Returns: None.

**get_filter_string()**

- Description: Get the filter string for filter requests.
- Parameters: None.
- Returns: A [str] <filter> that represents the filter that will be applied in other calls such as get_all_data().

**set_filter_string(<filter>)**

- Description: Set the filter string for filter requests.
- Parameters: [A [str] <filter> that represents the filter that will be applied in other calls such as get_all_data().
- Returns: None.

**update(<field_value_pairs>,[optional] <where>)**

- Description: Change a field value. Note that it is up to the user to ensure that the where clause is correctly constructed.
- Parameters: A new[str] <field_value_pairs> of the form "field1=value1, field2=value2...", a [str] <where> clause to restrict the number of records retrieved (default=None).
- Returns: None.

**select([optional] <fields>,[optional] <where>,[optional] <limit>, [optional] <select_raw>)**

- Description: Select rows that match specified criteria.
- Parameters: A [str] <fields> specifying a list of fields to select (default='*' all fields), a [str] <where> clause to restrict records retrieved (default=None), a [int] maximum <limit> on the number of records returned (default=None), a [bool] flag to choose a return value of either a list of tuples (set True) or a SqlDataSet (default=False).
- Returns: A List[Tuple(Any)] if <select_raw> is True, else an [SqlDataSet] SQL data set. See *SQL Part*.

**insert(<record>)**

- Description: Append a new record.
- Parameters: A new Tuple[Either[str, int, float]] <record> where each record is a tuple containing values of type string, integer, and/or float.
- Returns: None.

**count(<where>)**

- Description: Count rows that match specified criteria.
- Parameters: A [str] <where> clause to restrict data retrieved.
- Returns: The [int] number of records matching the <where> clause.

**import_from_access(<file_path>,<from_table_name>)**

- Description: Import from *MS* Access.
- Parameters: The complete [str] <file_path> to the Access database, the [str] <from_table_name> to import the data from.
- Returns: None.

**export_to_access(<file_path>,<to_table_name>)**

- Description: Export to *MS* Access.
- Parameters: The complete [str] <file_path> of the Access database, the [str] <to_table_name> to export the data to.
- Returns: None.
- *Note 1: The full table is exported even if a filter is applied.*
- *Note 2: The Access file to export to must exist as a new one will not be created.*

**create_index(<column_names>)**

- Description: Create a Table index.
- Parameters: A comma-delimited [str] of <column_names> to index. e.g. "Col1, Col2, ..."
- Returns: none.

**drop_index(<column_names>)**

- Description: Drop a Table index.
- Parameters: A comma-delimited [str] of <column_names> in the index to drop.
- Returns: None.

*Part API*

*Table of Contents*

### 5.2.18 Time Part

Each Time Part exposes the following functionality to scripting *API* by using link.<*part_alias*>.<*scripting_api*>, where <*part_alias*> is the link name, and <*scripting_api*> are the methods listed below:

**elapsed_time**

- Property: Get or set the elapsed time of the time part. Returns an object of type 'relativedelta' (see dateutil package documentation). e.g. delta = time.elapsed_time or time.elapsed_time = delta.

**reset**()

- Description: Resets the elapsed time to zero.
- Parameters: None.
- Returns: None.

*Part API*

*Table of Contents*

### 5.2.19 Variable Part

Each Variable Part exposes the following functionality to scripting *API*:

**link.<var_alias>**

- Description: Get the Python expression.
- Parameters: none.
- Returns: [str] the string returned by assignment.

**link.<var_alias>= <value>**

- Description: Set the Python expression.
- Parameters: [object] a <value> to set to the variable part. The value set is the objects string representation (using "repr").
- Returns: none.

**obj**

- Property: Get or set the object, which is usually set by the script. e.g. the_obj = variable.obj or variable.obj = the_obj.

**get_obj**()

- Description: Get the object, which is usually set by the script.
- Parameters: None.
- Returns: The [object] to get. The value set is the objects string representation (using "repr").

**set_obj()**

- Description: Set the object, which is usually set by the script.
- Parameters: The [object] to set in the variable part. The value set is the objects string representation (using "repr").
- Returns: None.

*Part API*

*Table of Contents*

CHAPTER

SIX

# USING THE ORIGAME CONSOLE VARIANT

The ORIGAME Console is run from the Windows Command prompt. The general usage of ORIGAME Console is as follows:

**usage: origame_con.py [–h] [–b] [–l] [–dev_log] [–dev_no_dep_warn]** [–dev_log_raw_events] [–dev_no_wiring_fixes_on_load] [–t MAX_SIM_TIME_DAYS] [–x MAX_WALL_CLOCK_SEC] [–f REALTIME_SCALE] [–s SEED_FILE_PATH] [–v NUM_VARIANTS] [–r NUM_REPLICS_PER_VARIANT] [–c NUM_CORES] scenario_path

Each positional argument is described in the table below. The mandatory positional argument is the *scenario_path* that identifies where the scenario definition is and what the file is called. An example use case is provided below, where the Windows Command prompt is opened in the directory **python34\scripts**:

> Python.exe origame_con.py c:\myscripts\myscenario.ori –v23 –r4

The above example would run the scenario file "myscenario.ori" for 23 variants and 4 replications.

If ORIGAME Console is run on a *PC* where **only** Python34 is installed, *Python.exe* can be called without a prefixed path to its location in the file hierarchy, as in the above example. However, if multiple versions of Python are installed, then the full path to Python34 must be included. For example:

> C:\Python34\Python.exe origame_con.py c:\myscripts\myscenario.ori –v23 –r4

The mandatory and optional positional arguments are described in the following table. This table can be displayed in the Command Window by typing:

> Python.exe origame_con.py –h

| Positional Arguments: | Description |
|---|---|
| scenario_path | Mandatory. Scenario definition file pathname. Can be relative path. |
| –h, —help | Optional. Show this help message and exit. |
| –b, —batch_replic_save | Optional. Turn off saving of final scenario state by each batch replication. |
| –l, —logging | Optional. Turn logging off. |
| —dev_log | Optional. Developer option to log at system debug level. |
| —dev_no_dep_warn | Optional. Developer option to turn off logging of prototype *API* deprecation warnings. |
| —dev_log_raw_events | Optional. Developer option to turn on logging of sim event push/pops raw data. |
| —dev_no_wiring_fixes_on_load | Optional. Developer option to turn off fixing of invalid links on scenario load. |
| –t MAX_SIM_TIME_DAYS, —max_sim_time_days MAX_SIM_TIME_DAYS | Optional. Simulation cut-off time in days. Default runs till no events (or max real-time reached, if set; whichever occurs first). |
| –x MAX_WALL_CLOCK_SEC, —max_wall_clock_sec MAX_WALL_CLOCK_SEC | Optional. Real-time cut-off time in seconds. Default runs till no events (or max sim time reached, if set; whichever occurs first). |
| –f REALTIME_SCALE, —realtime_scale REALTIME_SCALE | Optional. Turn on real-time mode, using the given scale factor. Default is as-fast-as-possible. |
| –s SEED_FILE_PATH, —seed_file_path SEED_FILE_PATH | Optional. Seed file pathname. Can be relative path. Default causes seeds to be randomly generated. |
| –v NUM_VARIANTS, —num_variants NUM_VARIANTS | Optional. The number of scenario variants to be run. Default runs a single variant. |
| –r NUM_REPLICS_PER_VARIANT, —num_replics_per_variant NUM_REPLICS_PER_VARIANT | Optional. The number of scenario replications to be run for a scenario variant. Default runs a single replication for a variant. |
| –c NUM_CORES, —num_cores NUM_CORES | Optional. The maximum number of cores to utilize for the configured scenario run. Zero distributes batch replications across all available cores. Default number of cores is 1. |

CHAPTER

SEVEN

# GLOSSARY

**ACCDB** (Microsoft) Access Database

**ASAP** As Soon As Possible

**API** Application Programming Interface

**CPU** Central Processing Unit

**CSV** Comma Separated Values

**DND** Department of National Defence

**DRDC** Defence Research and Development Canada

**EPS** Encapsulated Postscript

**FIFO** First-In-First-Out

**GB** Gigabyte

**GUI** Graphical User Interface

**HR** Human Resources

**ID** Identification

**JSON** JavaScript Object Notation

**LIFO** Last-In-First-Out

**MDB** Microsoft Access Database

**MPRA** Military Personnel Research and Analysis

**MS** Microsoft

**ORI** Origame file

**ORIB** Origame binary file

**PC** Personal Computer

**PDF** Portable Document Format

**PNG** Portable Network Graphics

**PS** PostScript

**R4** Right Person, Right Qualification, Right Place, Right Time

**RAM** Random Access Memory

**SVG** Scalable Vector Graphics

**SQL** Structured Query Language

**venv** (Python) Virtual Environment

**XLS** Excel Spreadsheet

**XLSX** Excel Spreadsheet