MSCViewer V2.1.0
User Manual

Roberto Attias

Contents

1 Introduction

1.1 MSCViewer

MSCViewer is a tool supporting visualization and analysis of message sequence charts intended primaritly for debugging of logs/traces obtained from concurrent systems.

The tool loads a text file and interprets lines containing well known keywords representing events and interactions occurring in entities. Lines not containing the known keywords are ignored. From the interpreted lines a model is constructed in memory. In the GUI the tool shows the list of all entities for which events exist. From this list the user can select a subset to be shown in a sequence diagram chart. Through keyboard or mouse commands the user can easily navigate through events and interactions, rearrange entities, search, and perform a number of other actions for browsing the diagram.

In addition to using the tool for visualization of traces for existing systems, during a design phase the user can write a concise description of one or more flow[1] in the input language, load it in the tool, and export an image to be used for documentation. Later on, the implemented system should produce diagrams similar to the one defined at design time, hence this approach can be used to confirm the system behaves as designed.

In order for a system to produce the required data for MSCViewer, the program code need to be instrumented with syslogs, printf, traces or any other mechanism avaliable, using the format expected by the tool for the messages. The user can run the application and generate the input file for MSCViewer, possibly by merging together multiple generated files.

Visual inspection of the system behavior from execution logs is useful, but doesn't scale for debugging, as it requires a human operator familiar with the expected behavior. When applications are large enough, their development might be distributed across multiple teams, and possibly even geographically, with only few members fully aware of cross-component flows. To address this problem MSCViewer is integrated with a Python interpreter and supports execution of scripts which can browse the model and perform any sort of validation. In addition, a set of classes is provided to support a concise description of expected flows. With these mechanism the user can create script to be executed in the tool on the model created from execution logs, automatically validating their correct-

[1] a formal definiton of flow will be provided later in this document. For the time being consider it a sequence of events across one or more entities

ness. Scripts can also be executed from command line, without any GUI visualization, supporting batch validations and product smoke-testing.

1.2 Message Sequence Charts and Flows

A large class of products and application is composed by multiple concurrent entities. For example a web application may be composed by a front-end running in a browser, a backend web server and a database. The user interacts with the frontend, which in turn interacts with the backend. The backend interacts with the database. We refer to elements involved in the interactions as *Entities*, and the sequence of interactions among entities required to perform a functionality as *Flow*.

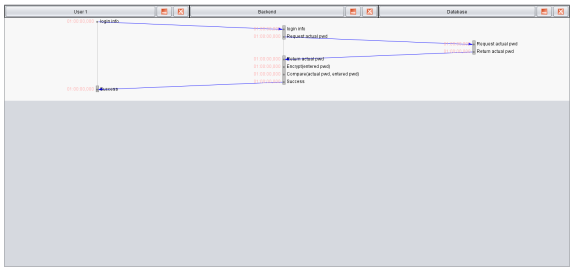


Figure 1.1: A simple flow example

Fig 1.1 shows, as an example, the flow involving a user logging into a server which stores users information in a database. In complex concurrent system multiple homogeneous or heterogeneous flows spawning tens or hundreds of entities may be in execution at any point in time.

When designing, implementing and debugging such a complex system, flows play a very important role in the entire development cycle.

In the design phase is fairly common to represent expected flows in a design document through some variation of Message Sequence Charts [2], such as the one shown in fig 1.1.

In the implementation phase the developer can sprinkle calls to a tracing/logging [3] infrastructure around the main points of flows.

At debug time the user can then browse the various log files and mentally reconstruct and verify flows. If not properly supported by a tool this exercise can be complex and time-consuming, as the user has to reassemble flows identifying relevant information often diluted in hundreds of lines of unrelated logs. To add to this complexity, flows may not always be purely sequential like the one shown in fig 1.1.

For example, fig 1.2 shows a user requesting the instantiation of a virtual machine to an hypotetical VM Manager. the VM Manager requests asynchrously resources (disk space, CPU, memory) to a Resource Manager and an image for the VM to a VM Repository. When the resources and VM image are provided, VM Manager spawns the VM.

Note that the flow could also happen in different ways. For example, the response from the Image Repo could reach VM Manager before the response from the Resource Manager. The implication of this consideration is that verifying whether a flow happened

[2]for more information on Message Sequence Chart see http://en.wikipedia.org/wiki/Message_sequence_chart
[3] for more information about tracing vs. logging see http://en.wikipedia.org/wiki/Tracing_(software)

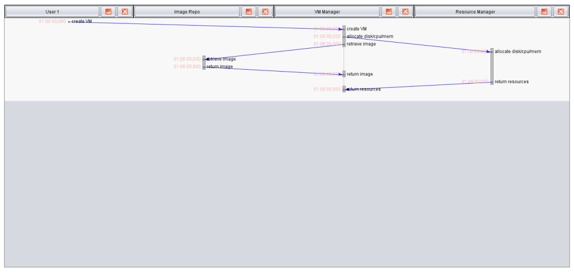

User 1
Image Repo
VM Manager
Resource Manager
01:00:00,000 create VM
01:00:00,000 create VM
01:00:00,000 allocate disk/cpu/mem
01:00:00,000 retrieve image
01:00:00,000 allocate disk/cpu/mem
01:00:00,000 retrieve image
01:00:00,000 return image
01:00:00,000 return image
01:00:00,000 return resources
01:00:00,000 return resources

Figure 1.2: A flow with two concurrent branches

as expected doesn't always amount to simply checking whether certain string appear in a specific total order in the logs. Sometimes events (i.e. log lines) have only a partial order, or some parts may be optional, for example in case of an *if* vs. *else* branch taken in the code.

Capturing flows at design time is very important to properly document how a system composed by multiple concurrent elements operate. Validating flows in logs of runtime executions is crucial to guarantee a complex system is operating as expected. MSCViewer was designed to support this.

1.3 History and Acknowledgements

Work on MSCViewer was started around 2010 at CISCO. The idea was to develop a tool for usage during development of the control plane for the NCS-6000, CISCO last-generation core router, specifically to address the problem of automatically identifying potential problems in flows and reducing the triaging time and amount of resources required. While the tool was started as a personal initiative, it has found good adoption by some communities in the company.

Before 2010, I had worked on other tools with some common aspects, so MSCViewer can be considered the result of an effort started around 2001 (no code or IP from those early efforts has been used in MSCViewer).

As of Sept. 2014 CISCO has gracefully consented to release MSCViewer in the open-source.

I want to thank the following people for their ideas, suggestions, feedback and support during development of MSCViewer:

- Akash Deshpande and Marco Zandonadi, whose collaboration in CISCO and before has helped shaping MSCViewer to its current form;
- Edward Conger, for his suggestions, observation and his ability to push the boundary of MSCViewer and other tools I developed;
- the CISCO managment, in particular Sunil Khaunte, Feisal Daruwalla, Satish Gannu, Sohyong Chong and Robert Krohn, for their support during the developement of MSCViewer and for facilitating its release as open source.

2 Getting Started

In this chapter we will start familiarizing with MSCViewer concepts through some examples. The files used in the examples are located in the `examples/` directory of the distribution.

2.1 Your First Sequence Diagram

The following is a simple input file for MSCViewer:

```
@event { "entity":"producer", "label":"start"}
@event { "entity":"producer", "label":"produce", "src":"1"}
@event { "entity":"consumer", "label":"consume", "dst":"1"}
```

Each line defines an *event*, i.e. an occurrence of something relevant in an entity. The syntax consists in the `@event` token followed by a JSON object (see http://json.org/ for more information about the JSON format).

Line 1 defines a *local* event, which is not part of any interaction with other entities. The value of the `@entity` key indicates the entity this event belongs to. There is no explicit sytax to define an entity: as it parses events, MSCViewer creates entities the events belong to.

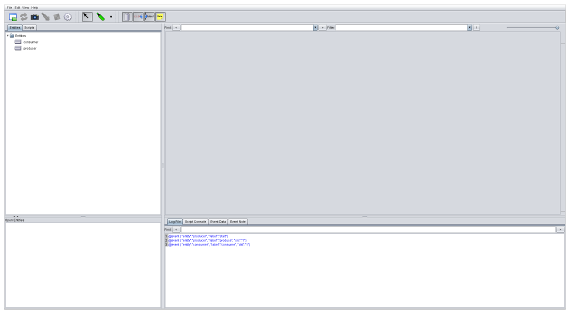
Line 2 defines an event which is part of an interaction. More specifically, this event is the *source* (or *cause*) for the interaction, as indicated by the presence of the `src` key. Line 3 defines a *destination*, or *effect*, as indicated by the presence of the `dst` key. An interaction is an ordered pair composed by a cause and an effect event.

2.2 Running MSCViewer

We can now start MSCViewer with this input file. Assuming the current directory is the top directory of the distribution, run:

```
bin/mscviewer examples/lst1.msc
```

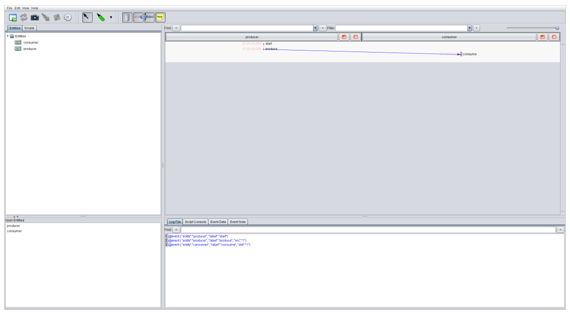
Fig. 2.1 shows the GUI in its initial state. The bottom-right area shows the content of the input file, while the top-left tree shows the various entities. Double-click on the



```
@event{"entity":"producer", "label":"start"}
@event{"entity":"producer", "label":"produce", "src":"1"}
@event{"entity":"consumer", "label":"consume", "dst":"1"}
```

Figure 2.1: MSCViewer GUI with `lst1.msc` loaded.

`producer` and `consumer` entities to open them in the diagram area, in the top-right part of the window (fig. 2.2). Each entity is shown in a column, with events interleaving each other across entities. Columns can be sorted by dragging them from their header (the part showing the entity name), and closed by clicking on the ☒ button. Clicking on an event selects it; the selection can be moved to the previous/next event for an entity by pressing respectively the SHIFT ↑ and SHIFT ↓ key combinations. To navigate from a source or destination event to the paired event in an interaction use the SHIFT ← or SHIFT → .



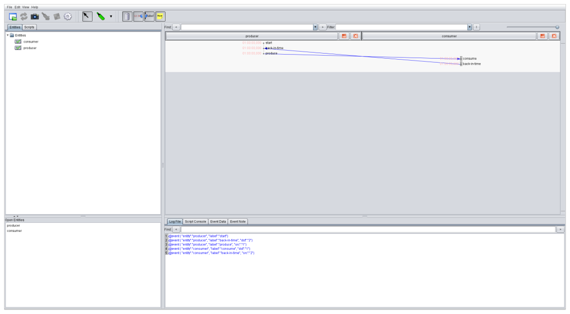
```
@event{"entity":"producer", "label":"start"}
@event{"entity":"producer", "label":"produce", "src":"1"}
@event{"entity":"consumer", "label":"consume", "dst":"1"}
```

Figure 2.2: The user has opened all the entities in the diagram

Try closing the `consumer` entity by clicking on the ☒ button. As the destination entity is now closed, the interaction outgoing from the producer is shown as an arrow stub. In this case, it is pretty obvious who the destination entity is, but in more complex scenarios there can be hundreds of entities, and it wouldn't be easy to know which one to open from the entities tree. In this situation the keyboard navigation cames handy. Select the interaction stub, then press SHIFT → : "pushing" on the tip/base of an interaction stub for which the destination/source entity is not open causes the entity to be opened.

2.3 Topological Sorting

In the diagram in fig. 2.2 events appear in the same order in which they were in the file. However, this is not always the case. Select the `File/Open...` menu item, and from the file selection widget open the `lst2.msc` file. As shown in the input area, this file is the same as `lst1.msc`, except that the last two lines have been swapped. Despite the fact that the destination for the interaction is listed before the source, opening the two entities will show the same diagram as before: MSCViewer tries to preserve cause-effect ordering among events by applying a topological sort to the events as they're loaded. It is possible to write input files with create circular dependencies of events, resulting in the impossibility of performing a topological sorting. In this case MSCViewer reports an error upon loading the file and skips the sorting. This case is captured in `lst3.msc`, shown in fig. 2.3



```
@event { "entity":"producer", "label":"start"}
@event { "entity":"producer", "label":"back-in-time", "dst":"2"}
@event { "entity":"producer", "label":"produce", "src":"1"}
@event { "entity":"consumer", "label":"consume", "dst":"1"}
@event { "entity":"consumer", "label":"back-in-time", "src":"2"}
```

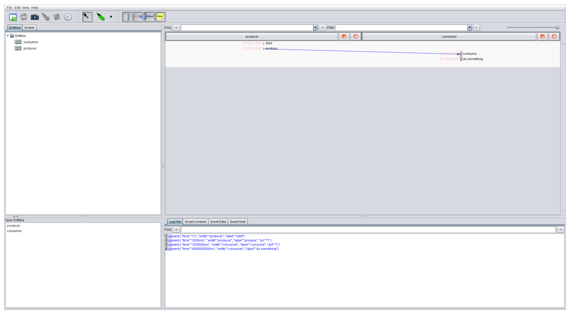
Figure 2.3: Diagram with circular dependencies caused by interactions

2.4 Timestamps

Timestamps can be associated to events throught the `time` key. The value can assume different formats, as shown in `lst4.msc`:

```
@event { "time":"1s", "entity":"producer", "label":"start"}
@event { "time":"2000ms", "entity":"producer", "label":"produce", "src":"1"}
@event { "time":"3000000us", "entity":"consumer", "label":"consume", "dst":"1"}
@event { "time":"4000000000ns", "entity":"consumer", "label":"do something"}
```

. Regardless of the unit specified in input, MSCViewer converts internally the timestamp into nanoseconds, and visualizes them in a format that can be chosen by the user. The default format is `hh:mm:ss,ms`, but it can be changed through the the `Edit/Preferences...` menu item. Fig 2.4 shows timestamps visualized near events.



```
@event { "time":"1s", "entity":"producer", "label":"start"}
@event { "time":"2000ms", "entity":"producer", "label":"produce", "src":"1"}
@event { "time":"3000000us", "entity":"consumer", "label":"consume", "dst":"1"}
@event { "time":"4000000000ns", "entity":"consumer", "label":"do something"}
```

Figure 2.4: Diagram with timestamps associated to events.

2.5 Associating Data to Events

To associate some data to an event, use the `data` key. The value is a JSON object that will be visualized in the bottom-right area, in the `Data` tab. For example, loading the following input file and selecting the third event results in the screen as shown in fig. 2.5:

```
@event { "time":"1s", "entity":"producer", "label":"start",
    "data":{"somekey":"somevalue", "anotherkey":"anothervalue"}}
@event { "time":"2s", "entity":"producer", "label":"produce",
    "data":{"bytes":[10,20,30]}, "src":"1"}
@event { "time":"3s", "entity":"consumer", "label":"consume",
    "data":{"foo":"bar", "subobj":{"some":"stuff", "someother":"stuff"}},
    "dst":"1"}
@event { "time":"4s", "entity":"consumer", "label":"do something"}
```

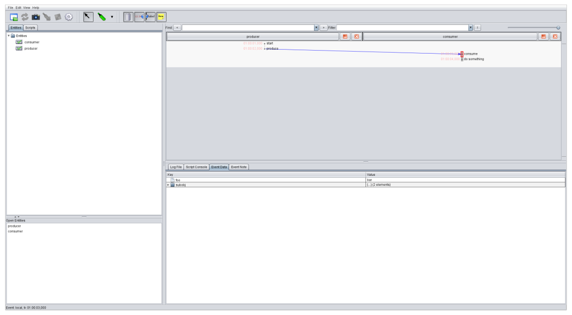


Figure 2.5: The Data view showing data associated to the selected event

2.6 Orphaned Interactions

When collecting traces from a system it is possible that either the event representing a cause or the one representing an effect for an interaction goes missing. This can be caused by multiple reasons, such as the fact that a trace has wrapped around (typically traces don't grow indefinitely, but rather wrap around, loosing older information), or that traces for one of the two entity involved in the interaction are not available. We will refer to interactions for which only one of the two events is available as *Orphaned Interactions*. MSCViewer visualizes orphaned interactions as dashed arrow stubs outgoing or incoming into the only available event.

3 MSCViewer Model

This chapter provides a more formal definition of the MSCViewer model.

3.1 Entity

An *entity* is an autonomous execution entity such as a thread or state machine. In MSCViewer entities can be defined hierarchically. For example, in a router a node entity might contain a process entity, containing a thread entity, containing a state machine entity. An entity is characterized by a name, a unique ID and a set of events. The ID of an entity is composed by the ID of the parent, a slash (`/`) and an identifier. In other words, entity IDs have a namespace similar to a filesystem path. Let's consider the example shown in table 3.1:

ID	Name
0	Node0
1	Node1
0/1234	Node0/DS
0/1236	Node0/DS
1/1234	Node1/DS

Table 3.1: Example of entities

Here we have two nodes, named `Node0` and `Node1`. The entity IDs are respectively 0 and 1, while their names are `Node0` and `Node1`.

On the first node two processes are running, the first with PID 1234, the second with PID 1236. The IDs for the corresponding entities are `0/1234` and `0/1236`. The processes are instances of the same executable called DS, so the entity names are `Node0/DS` and `Node0/DS`.

On the second node a single instance of `DS` is running, with ID `1/1234` and name `Node1/DS`

It's worth mentioning that MSCViewer does not have any semantic knowledge of concept such as nodes, processes, or threads, and the model could be used to represent any kind of problem domain. MSCViewer understands only the concept of hierarchical IDS and names where elements in the hierarchy are separated by slashes.

Some entities can have an extra attribute characterizing them as clock sources. This indicates that the entity and all its descendants are on the same clock domain, hence sharing the same clock. As an example, a group of computational nodes may have a shared system clock (via software or dedicated hardware), in which case a system root entity could be used to represent this. In other scenarios a set of nodes may have a loose clock synchronization (for example via NTP), in which case entities representing nodes could be clock sources for children representing processes running on the node.

3.2 Event

An event is an occurrence of some relevance happening to an entity. Examples of event are the state transitions in state machines or receiving or sending a message. An event is always characterized by a timestamp, a label, and a type. The timestamp is relative to the clock source the entity belongs to. The label is a human-readable message providing information about the event. They type is an identifier representing formally the specific type of the event, used for example to select how to visually render the event in the sequence diagram (see **??**). In addition to these mandatory attributes, an event may have a dictionary of extra attributes. Rendering code or model browsing code can retrieve these attribute to perform special tasks.

3.2.1 Events Partial Order

Theoretically all events within an clock domain are fully ordered in a "happened-before" relationship by their timestamp. In reality however high event granularity, low clock resolution, or the presence of multiprocessing capabilities can result in multiple events with the same timestamp. For example a trace may have been generated for a computational node where have two processes have an event happening exactly at the same timestamp. This means that model events are a Partially Ordered Set where $ev_i < ev_j$ if at least one of the following conditions is true:

- ev_i and ev_j belong to the same clock domain and $t(ev_i) < t(ev_j)$
- $\exists$ an interaction I such that $(ev_i \to ev_j)$
- $\exists ev_1 \dots ev_n$ such that $ev_1 = ev_i, \quad ev_n = ev_j, \quad ev_k < ev_{k+1} \quad \forall k \in (1 \dots n-1)$

Upon parsing the model from the input file MSCViewer applies the Kahn topological sorting algorithm according to the partial order specified above. In the algorithm however the set S set is replaced with a FIFO where elements are added in the order in which they are defined in the input file. This, and the assumption that the input file was sorted on timestamps, guarantees that events with the same timestamp and no incoming edges in the graph will not be shuffled around.

The sorted events output by the algorithm are stored in an array which reflects the visualization order.

The topological sorting is particularly useful when the input data is produced by entities with different clock sources. For example, an input file may be produced by joining traces collected on two separate Linux nodes. Even when nodes are synchronized through NTP, there may be a small drift resulting in the tracing of a source event having a timestamp larger than the corresponding to the tracing of the destination event. A simple merging of the two traces based on timestamp would produce a model where cause and effect are in the opposite order (with the arrow representing the interaction going from bottom to top). The topological sorting takes care of this problem. Note that the timestamps associated to the events are not modified, hence while the arrow flows from top to bottom, the source will still show up with a timestamp greater than the destination. Computing the actual offset between the two nodes is not trivial if not impossible, and anyway correcting the timestamp may confuse the user.

3.3 Interaction

An interaction is a tuple $(ev_i, ev_j, type)$ where:

- ev_i is called the *source* or *cause* event, and the entity the event belongs to is called the *source entity*;

- ev_j is called the *destination* or *effect* event, and the entity the event belongs to is called the *destination entity*

- *type* is an identifier characterizing the nature of the interaction (for example `message`, or `creation`)

In the model adopted interaction implies a causality between the source and destination event, hence $(ev_i, ev_j, T) \not\Leftrightarrow (ev_j, ev_i, T)$.

Note also that the model allows source and destination event to belong to the same entity.

An example of interaction type is a message-passing. In this case the source event captures the sending of the message, while the destination event captures the reception. Another example is a state machine instantiation: in this case the creating entity would have an "sm instantiation" event, while the created state machine would have a first event corresponding to its birth. In this case, clearly, the state machine entity would be a children of the creator entity. An example where source and destination events belong to the same entity is initiation and completion of a DMA operation. Note that the model allows for additional events to be present between the source and destination event for an interaction.

In the model interactions are captured with an interaction ID attribute associated to the source and destination events. This attribute has the same value (a unique id) for events belonging to the same interaction.

The model allows for an event to be the source of multiple interactions, but the destination to only one interaction. events with multiple outgoing interaction can be used, for example, to capture multicast messaging.

In this document we will in some cases use the notation $(A_{e_1} \rightarrow B_{e_2})$ to represent an interaction between an event e_1 on entity A and an event e_2 on entity B

3.4 Input Language

MSCViewer can load a single file containing description of events for multiple entities and build a model in memory. The input file is composed by lines of text. MSCViewer ignores any line not containing the `@event` or `@entity` word. This allows to load trace files which might contain a mix of trace lines catered to MSCViewer and unrelated trace lines.

An event is described in the input language by a line containing the `@event` followed by a JSON object. If the reader is not familiar with this format, it is explained here: http://json.org/

MSCViewerinterprets certain keys-value pairs in the JSON objects in order to build the entity/event/interaction model.

For events, the following keys are interpreted:

Key:	`entity`
Type:	String
Mandatory:	y
Example:	`"id":"node0/producer_1"`
Description:	an identifier for the entity this event belongs to. identifiers can contain slash to indicate hierarchy of entities.
Key:	`label`
Type:	String
Mandatory:	y
Example:	`label":"operation competed"`
Description:	a label specifying some immediate info about this event. In MSCViewer the label is shown on the right of the icon representing the event.
Key:	`time`
Type:	String
Mandatory:	n
Example:	`"time":"1542532s"`
Description:	a timestamp for the event. It's an integer number followed by a unit qualifier: "s" (seconds), "ms" (milliseconds), "us" (microseconds) or "ns" (nanoseconds). The timestamp is interpreted as elapsed time since the Unix Epoch . If the timestamp is not present inside the JSON object, it is expected to be present before it in one of the formats supported by syslog. time is shown in MSCViewer at the right of the event icon.
Key:	`type`
Type:	String
Mandatory:	n
Example:	`"type":"Timer"`
Description:	If present, the value specifies a distinct type for the event. Events of different types can be rendered by different icons in MSCViewer. Icons can be provided by a user in a directory in the form of PNG images whose name matches the type name. This allows to plug domain-specific representation for events.

Key:	`data`
Type:	JSON Value
Mandatory:	n
Example:	`"data":{"v1":10, "v2":20}`
Description:	Specifies some data associated to the event. For example, if the event corresponds to the sending of a message, the value here could be a JSON representation of the message. in MSCViewer data is shown in a table in the data section when the event is selected.

Key:	`src`
Type:	JSON Value
Mandatory:	n
Example:	`"src":"producer_1/123"`, `"src":"{"id":"producer_1/123", "color":"00FF00"}`
Description:	Interactions are pairs of events. In a pair, one event contains the src key, while the other contains the dst key, both with the same value. Interactions are visualized in MSCViewer as arrows going from the source to the destination event. The file format doesn't mandate a particular format for the value, but the value should be unique for an interaction (more on this in the interaction section of this document. In the simplest case the value can be a string (which is a valid JSON value). When certain aspects of the rendering need to be controlled, the value can be a JSON object with an "id" key identifying the unique interaction, and other keys (such as "color") identifying rendering aspects.

Key:	`dst`
Type:	JSON Value
Mandatory:	n
Example:	`"dst":"producer_1/123"`, `"dst":"{"id":"producer_1/123", "color":"00FF00"}`
Description:	Interactions are pairs of events. In a pair, one event contains the src key, while the other contains the dst key, both with the same value. Interactions are visualized in MSCViewer as arrows going from the source to the destination event. The file format doesn't mandate a particular format for the value, but the value should be unique for an interaction (more on this in the interaction section of this document. In the simplest case the value can be a string (which is a valid JSON value). When certain aspects of the rendering need to be controlled, the value can be a JSON object with an "id" key identifying the unique interaction, and other keys (such as "color") identifying rendering aspects.

The following table shows the keys interpreted by MSCViewer for entities:

Key:	`id`
Type:	String
Mandatory:	y
Example:	`"id":"/node0/1984"`
Description:	The unique identifier for the entity

Key:	`name`
Type:	String
Mandatory:	y
Example:	`"id":"/node0/producer_1"`
Description:	A name for the entity. Names don't need to be unique and provide a more human-readable representation for the entity

4 MSCViewer GUI

This chapter provides a description of all the GUI elements in MSCViewer.

4.1 The Entities Tree

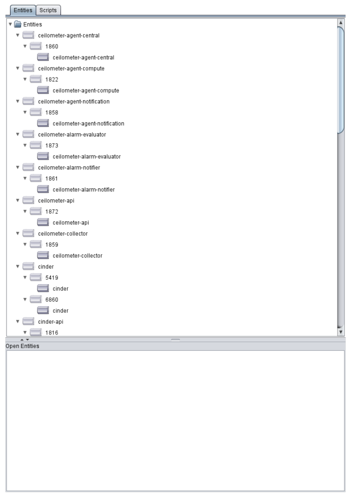


Figure 4.1: The Entities tree

The Entities Tree is visible on the top-left are of the GUI once an input file is loaded and the *Entities* tab is selected. This tree shows all entities defined in the input file. An entity is defined if a `@entity` line is present or if the entity is specified in the `entity` entry of a `@event`line. Elements of an entity ID separated by slashes constitute nodes in the tree. Internal nodes of the tree may have event of their own, if they appear as the terminal element in the `entity` entry of at least one event, or not, in which case they just represent some relationship among the child nodes. In the latter case the node is visualized in a lighter shade.

For example, a system may be composed by multiple processes running in different hosts. The user may decide to have a log file for each process, and use a `host-id/process-id` notation for entity events. In this case, Intermediate tree nodes for host-id will be created, but clearly there will be no events associated to them.

Double-clicking on a node corresponding to an entity with at least one event opens the entity in the Sequence Diagram. Double-clicking on a node corresponding to an entity already opened in the Sequence Diagram causes it to be removed from the diagram. The following legend shows the various representations of the tree nodes:

▭	entity with at least one event, currently closed in sequence diagram
☑	entity with at least one event, currently open in sequence diagram
▯	entity with no event, can't be opened in sequence diagram

4.2 The Sequence Diagram

The Sequence Diagram ais situated on the top-right part of the GUI. Entities can be opened in this area by various user actions:

- double-clicking a node in the *Entity Tree*, or pressing enter open the selected entity
- double-clicking on a line in the *Log File View*
- programmatically, by Python scripts.

Open entities can be rearranged by dragging their `Header`, the rectangle containing the entity name.

Width for an entity column can be increased or reduced by selecting the header and pressing repreatedly the $\boxed{+}$ or $\boxed{-}$ keys.

4.3 The Log File View

The Log File View is a tab in the bottom-right part of the GUI. This view shows the original log file that was parsed to create the sequence diagram model. Lines that contributed event information are highlighted in blue, while other lines are shown in black. Clicking on an event line selects the correspoding event, provided that the owning entity was open in the sequence diagram area. Selecting an event in the sequence diagram causes the corresponding line in the Source View to be selected. Double-clicking on an event line in the Source View causes the corresponding entity to be opened in the sequence diagram and the event selected.

Clicking and dragging in the Log View causes text to be selected. When the mouse button is released the currently selected test is copied in the system clipboard.

4.4 The Event Data View

Events may have associated structured data. The data associated to the selected event is shown in the Data View, which is located inside a tab in the bottomr-right part of the GUI.

4.5 The Notes View

In order to facilitate a debugging session the user can associate notes to events. A note associated with an event is shown as a small yellow square on the event. Selecting the event and right-mouse-clicking the user can checkmark the "Note Sticker" menu item to make the note visible as a sticker in the sequence diagram. To create a note, the user selects the event and edit the note in the Notes View, which is situated in a tab in the bottom-right part of the GUI.

4.6 The Tool Bar

The toolbar contains the following buttons:

Actions

Loads a new input file. Pressing this button causes a file chooser to open. Selecting a file and pressing ok results in the file being loaded.

Reload the present file. This command can be used if changes are made to the file after it was loaded. Note that all currently opened entities are closed before the new file is loaded.

Captures a screenshot of the sequence diagram. The user can select whether to capture the entire model (regardless of what entities are opened in the diagram), The entities currently opened, or just the entities for which at least one event is highlighted.

Removes any highlight set on events or interactions, whether they're on open entity or not.

Re-run the latest script that was executed. Scripts can be executed by double-clicking on nodes in the tree shown in the Script tab, but this button provides a convenient shortcut to rerun the latest script.

Opens a window where various program options can be configured.

Cursor Tools

Chooses the select tool associated to the mouse pointer. The select tool allows to select an event or interaction.

chooses the green marker for highlighting events. When this tool is selected clicking on an event causes it to be highlighted in green.

Same as the green highlighter, but with blue color.

Same as the green highlighter, but with yellow color.

Same as the green highlighter, but with red color.

Visualization Options

	Shows/hides blocks.
12:34	Shows/hides event timestamps
label	Shows/hides event labels
Note	Shows/hides notes associated to events

5 Customizing MSCViewer

5.1 Preferences

a number of parameters affecting the GUI can be modified in the `Preferences` dialog accessible via the `Edit/Preferences...` menu item. These include:

- how to interpret event timestamps where units were not specified
- the visualized format for timestamps in the sequence diagram
- the format for entities headers in the sequence diagram
- colors of various GUI elements

When any of the parameters is modified the change is immediately visible in the GUI. If the dialog is closed by pressing the `Ok` button, then those changes become permanent. Pressing the `Cancel` button or closing the dialog via the OS-specific window-close button results in the original parameters being restored.

The original parameters can also be restored without closing the dialog by pressing the `Reset` button.

5.2 Sessions

While `Preferences` control program parameters irrespective of the loaded log files, some set of parameters can be associated to specific Log files loaded in the tool. These set of parameters, referrred to as *Sessions*, are saved in XML files which can be reloaded later or even transfer across different machines. Session information includes:

- Entities open in the sequence diagram
- currently selected event or interaction
- Notes associated to events
- markers associated to events

5.3 Passing Resource Sets to MSCViewer

At the time of execution MSCViewer can be passed one or more directories, each one specifying a set of resources. Resource directories are passed using the `-r` option. Directory paths should be separated using the OS-specific path separator character (`:` on Linux/OS-X and `;` on Windows). Each directory can contain one or both the following sub-directories:

- `renderer`: this directory can contain images in `.png` format. These images are loaded by MSCViewerand become available as renderer for events whose type matches the file names (without the extension).
- `script`: this directory can contains Python scripts. These scripts are parsed by MSCViewerand specially decorated functions become available for execution in the GUI. For more information on Python scripts see chapter 6.2.

Consider the following example on Linux:

```
mscviewer -r food;animals test.msc
```

where `test.msc` contains

```
@event { "entity":"food", "type":"bananas", "label":"bananas"}
@event { "entity":"food", "type":"cake",  "label":"cake"}
@event { "entity":"animals", "type":"bee", "label":"bee"}
@event { "entity":"animals", "type":"snail", "label":"snail"}
```

and the current directory has the following directory structure:

```
.
├── animals/
│   └── renderer/
│       ├── bee.png
│       └── snail.png
└── food/
    └── renderer/
        ├── bananas.png
        └── cake.png
```

This loads a sequence diagram where icons for various events are picked up from the `renderer` subdirectories.

This example is present in the `examples/custom-icons` directory of the distribution, together with a `run.sh` script that runs it regardless of the OS in use.

6 Python Integration

One of the most powerful aspects of MSCViewer is the integration with a Python interpreter which allows to access programmatically both model as wel as GUI.

One use of this feature consist in writing validation scripts that can browse model data and report whether a particular computational flow is present or not, it respects certain timing constraints, etc. Because the script is written once and then applied to traces produced by running the system multiple times, this scales the effort of an activity that otherwise would have to be performed by one or more human beings. A tutorial on how to create flow validation scripts using the APIs described in this chapter is provided in chapter 6.3

Python scripts can also be used to drive the GUI. For example, the screen captures in this manual are all generated automatically a build time by a Python script which loads a model, open some entities in the sequence diagram, captures screenshots and saves them.

The interaction between the user Python script and the tool is done through APIs collected in a few Python files the script can import, and described in details in 6.2.

6.1 *Hello World* of Python Integration

Let's start with the smallest possible *Hello World* program in Python (`examples/hello.py`):

```
print 'Hello World'
```

This can be executed by MSCViewer as follows:

```
$ mscviewer -b examples/hello.py examples/lst1.msc
 Hello World
```

Here we're invoking MSCViewer in batch mode, passing the `hello.py` script and a log file. The log file is not even used in this case, and the script is simply passed to the embedded Python interpreter which runs it, producing the output on the console. Not particularly exciting.

Let's examine next how we can interact with the model created from the log file, as demonstrated by `examples/info.py`:

```
from msc.model import *
```

```

model = events()
for ev in model:
    en = event_entity(ev)
    en_name = entity_path(en)
    label = event_label(ev)
    print 'entity=', en_name, 'label=', label
```

```
$ bin/mscviewer -b examples/info.py examples/lst1.msc
 entity= producer label= start
 entity= producer label= produce
 entity= consumer label= consume
```

This example uses the Python API to interact with the model created from the log file, printing out for each event the entity path and the label. The extensive set of APIs provided to interact with the model as well as the GUI is documented in section 6.2.

In addition to executing a single Python program in batch mode, it is possible to define any number of Python functions that can be explicitly invoked by the user within the GUI. To demonstrate this, let's consider the `examples/hello1/hello.py` file:

```
from msc.gui import *


@msc_fun
def hello_world():
  print 'Hello World'


if __name__ == '__main__':
    hello_world()
```

In order for functions to be visible by the gui we need to place the Python file declaring them into a package, which we can achieve by creating an empty `__init__.py` file in the directory containing the file. For this reason, the `examples/hello1` directory has the following structure:

```
hello1
 ├── __init__.py
 └── hello.py
```

We can now run MSCViewer as follows:

```
mscviewer -p examples/hello1 examples/lst1.msc
```

Once the GUI comes up, click on the `scripts` tab. The tool will parse the Python modules available in the path, and the visualized tree should contain a `hello1` node representing the package. Expanding such node should reveal a leaf node representing the `hello_world()` function. Notice that in order for a function to show up in the GUI tree the function has to be decorated by the `@msc_fun` decorator. This allow to have various utility functions in the file that do not show up as first class citizens in the GUI.

In order to execute the function, just double-click on its node in the tree. In this particular case the function takes no arguments, so it will be immediately executed producing the expected `Hello World` output on the console.

Let's now consider `examples/hello2/hello.py`:

```python
from msc.gui import msc_fun
from msc.utils import msc_print

@msc_fun
def hello_something(text):
  msc_print('Hello '+text)

@msc_fun
def hello_world():
    hello_something('World')

@msc_fun
def hello_world_or_not(text='World'):
    hello_something(text)

if __name__ == '__main__':
    hello_world()
```

This example provides three functions that will be listed in the GUI:

- `hello_something()` expects an argument of type string and prints the concatenation of `'Hello '` followed by such string. The first time we double-click on this node a dialog will open allowing us to provide a value for the argument. We can type `'World'` (including the apices) and press `OK`. This time the output will not be produced on the console, but rather in the `Script Console` tab in the bottom-right are of the GUI. Double-clicking on the same node again will not open the dialog again, as MSCViewer remembers the previosly passed value and uses it again. In order to open the dialog again and change the value keep the SHIFT key pressed while double-clicking on the node.

- `hello_world()` calls the previous function passing the value of `'World'` for the argument.

- `hello_world_or_not()` has a keyword parameter. Keyword parameters have a default value, hence double-clicking on the node executes the function with such default value for the parameter. Again, to change such value keep the SHIFT key pressed while double-clicking.

It should be noted that this particular example follows a common pattern which allows a script to be executed in batch mode, or to provide functions to be called in interactive mode. The pattern consists in the presence of an `if __name__ == '__main__'` statement which is executed when in batch mode, calling a function with default values. Also, the pattern makes use of the `msc_print` function to produce output: this function will direct its output to the GUI `Script Console` view when in interactive mode, while falling back to printing on the console when in batch mode. We can run the script in batch mode with

```
mscviewer -b examples/hello2/hello.py examples/lst1.msc
```

6.2 Pyhton API

The interaction between the user Python script and the tool is done through APIs collected in a few Python files the script can import. The Python language doesn't have a concept of private vs. public API, and the technology used to integrate the Python interpreted in MSCViewer (which is written in Java) results in all the Java classes being actually visible and accessible from the Python code. However the user should restrict itself to using the API documented in the next few sections, as this provides better guarantees of not being modified in the future. No such claims are made agains any of the Java classes.

The API is currently divided in the following modules:

- `msc.model` – provides API to access entities, events and interactions in the model
- `msc.gui` – provides API to access the GUI
- `msc.utils` – provides some utility API
- `msc.graph` – provides API to access the graphing capabilities (experimental).
- `msc.flowdef` – provides API to implement compact definition of expected flows and apply them to the current model

6.2.1 Model API

This module contains the Pthon API to access MSCViewer models (entities, events, interactions).

Even though Python scripts may have access to the full Java API directly, this module defines a stable interface that will guarantee compatibility in the future. Internal Java APIs may, and most likely will, be subject to change, so the user is strongly advised against its direct usage.

`entities(root_only=False)`

returns an iterator on all entities in the model.

Parameters

`root_only(boolean)`

if True, only top-level entities are iterated upon

Example

```
for en in entities():
            print entity_name(en)
```

`entity_count(root_only=False)`

returns a count of the entities in the model

Parameters

`root_only`(boolean)

if True, only top-level entities are counted

Example

```
printf "total entities: ",entity_count(), ", top-level: ",
    entity_count(True)
```

`entity_first_event_index(en)`

returns the index of the first event for this entity, or -1 if the entity has no events

Parameters

`en`(Entity) : the entity

`entity_id(entity)`

returns the unique ID of this entity

`entity_last_event_index(en)`

returns the index of the first last for this entity, or -1 if the entity has no events

Parameters

`en`(Entity) : the entity

`entity_path(en)`

returns the pathname of the entity.

Parameters

`en`(Entity) : an entity

`event_at(idx=0)`

returns the idx-th event in the model.

Parameters

`idx`(int)

index of the event, should be between 0 (included) and event_count() (excluded).

Example

```
for i in range(event_count()):
            print event_label(event_at(i))
```

`event_count()`

returns the count of all events in the model.

Example

```
for i in range(event_count()):
            print event_label(event_at(i))
```

`event_entity(ev)`

returns the entity this event occurred on.

Parameters

`ev(Event)` : an event

Example

```
for ev in events():
            print entity_path(event_entity(ev))
```

`event_index(ev)`

returns the index of the event in the model.

Parameters

`ev(Event)` : an event

`event_interactions(ev, outgoing=True)`

returns interactions incoming or outgoing from the event, depending on the value of the outgoing parameter.

Parameter

`ev(Event)` : an event

`outgoing(boolean)`

if True, outgoing interactions are returned, if False incoming interactions are reported (default=True).

Example

```
print "event has ", len(event_interactions(ev, False), "incoming and",
    len(event_interactions(ev), "outgoing transitions"
```

`event_is_block_begin(ev)`

returns True if this event begins a block, False otherwise.

Parameters

`ev(Event)` : an event

`event_label(ev)`

returns the label for the event.

Parameters

`ev(Event)` : an event

Example

```
for ev in events:
            print event_timestamp(ev)
```

`event_marker(ev)`

returns the current marker associated to the event, or None

Parameters

`ev(Event)` : an event

`event_predecessor(ev, same_entity=False)`

returns the event preceding this event.

If same_entity is False returns the event preceding this event in the model, otherwise returns the event preceding this event within the same entity.

Parameters

`ev(Event)` : an event

`event_successor(ev, same_entity=False)`

returns the event following this event.

If same_entity is False returns the event following this event in the model, otherwise returns the event following this event within the same entity.

Parameters

`ev(Event)` : an event

`event_timestamp(ev)`

returns the timestamp for the event. The timestamp is in the form of an integer number of nanoseconds.

Parameters

`ev(Event)` : an event

Example

```
for ev in events:
            print event_timestamp(ev)
```

`event_type(ev)`

returns the type of the event as a string.

Parameters

`ev(Event)` : an event

`events()`

returns an iterable on all events in the model.

Example

```
for ev in events():
            print event_label(ev)
```

`interaction_events(inter)`

returns a tuple containing the *from* and *to* event for the interaction.

Parameters

`inter``(Interaction)` : an interaction

`interaction_from_event(inter)`

returns the *from* event for the interaction.

Parameters

`inter``(Interaction)` : an interaction

`interaction_to_event(inter)`

returns the *to* event for the interaction.

Parameters

`inter``(Interaction)` : an interaction

`interaction_type(inter)`

returns the type of the interaction.

Parameters

`inter``(Interaction)` : an interaction

`interactions()`

returns an iterable on all interactions in the model

6.2.2 GUI API

This module contains the Pthon API to interact with MSCViewer GUI.

Even though Python scripts may have access to the full Java API directly, this module defines a stable interface that will guarantee compatibility in the future. Internal Java APIs may, and most likely will, be subject to change, so the user is strongly advised against its direct usage.

`capture_diagram(file_path)`

captures a PNG of the current content of the sequence diagram in the specified file

Parameters

`file_path``(String)`

path of the fiel to save the PNG image to

`capture_gui(gui_element_name, file_path)`

captures a screenshot of the specified GUI element in the specified file. This function is intended mostly for internal use to generate screen captures for the documentation

Parameters

`gui_element_name``(String)` : name of the GUI element

`file_path``(String)`

path of the fiel to save the PNG image to

`data_show()`

makes the data tab is visible

`error(model_el, msg)`

reports an error. In GUI mode the result is reported in the a popup window. In batch mode the error is printed on the console

---

`event_select(ev)`

Selects the specified event in the GUI

Parameters

`ev``(Event)` : the event to be selected

`event_selected()`

returns the event currently selected in the GUI, or None.

`expand_entity_tree()`

fully expands the entity tree so all leafs are visible

`graph_show(graph, type)`

(EXPERIMENTAL) Shows a graph in the GUI

`interaction_selected()`

returns the interaction currently selected in the GUI, or None.

`load(file_path)`

loads a new model file in mscviewer.

Parameters

`file_path``(String)` : path of the model file to be loaded

---

msc_fun(f)

Functions that should be visible in the GUI under the *Script* tree should be decorated with msc_fun.

Example

```
@msc_fun
    def myfunct():
        pass
```

---

open(entity_id)

opens in the sequence diagram the entity by the given name, returning the Entity object

Parameters

entity_id(String)
The fully-qualified id of the entity to be opened

---

progress_done(pr)

closes the window reporting progress

---

progress_report(pr, cnt)

Reports a progress.

Parameters

pr(handle) : a handle returned by progress_start()
cnt(int)
a value between *emminand*emmax, the values that were passed to progress_start()

---

progress_start(msg, min, max)

starts a reporting of progress. In GUI mode this will cause a progress window to show up. min and max are the range the progress is expected to cover. Returns a handle to be used with progress_report()

Parameters

msg(String)
message to be shown int the progress window
min(int) : initial value for progress
max(int) : final value for progress

---

results_show()

makes the results tab is visible

---

set_left_right_split_pane_divider_location(percent)

re-arranges the split between the entity tree area and the diagram/result area

Parameters

percent(int)

the percentage (horizontally) of the window that should be occupied by the Entity tree

---

set_right_split_pane_divider_location(percent)

re-arranges the split between the sequence diagram area and the result/data area

Parameters

percent(int)

the percentage (vertically) of the window that should be occupied by the sequence diagram

6.2.3 Flows API

This module contains the Pthon API to create flow definitions.

Even though Python scripts may have access to the full Java API directly, this module defines a stable interface that will guarantee compatibility in the future. Internal Java APIs may, and most likely will, be subject to change, so the user is strongly advised against its direct usage.

---

class fall

fall(*args)

class representing an concurrent flows. for a *fall* to match a sequence of events in the model, all of the flows must match

Parameters

***args**(flows) : the concurrent flows

Example

Suppose the model has a sequence of events on label *e1..e3* on entity *A* and on labels *e4..e6* on entity *B*. The following flow definition would match all those events, regardless of the order between events on *A* and *B*:

```
fall(fseq(fev("A", "e1"), fev("A", "e2"), fev("A", "e2")),
          fseq(fev("B", "e4"), fev("B", "e5"), fev("B", "e6"))
```

get_min_model_index()

returns the minimum among the index of the events this flow definition has been annotated with.

---

class fany

fany(*args)

class representing an alternative of flows. for a *fany* to match a sequence of events in the model, at least one of the sub-flow must match a set of events.

Parameters

`*args(flows)` : the flows alternatives

Example

Suppose the model has a sequence of events on entity *A* and on labels *e1..e5*. The following flow definition would match events *e4* and *e5*:

```
fany(fseq(fev("A", "e1"), fev("A", "e6")),
          fseq(fev("A", "e4"), fev("A", "e5"))
```

`get_min_model_index()`

returns the minimum among the index of the events this flow definition has been annotated with.

`class fev`

`fev(entity, label, predicate=None, predicate_arg=None, action=None, action_arg=None, descr=None)`

Instances of this class represent a single event.

`get_min_model_index()`

returns the minimum among the index of the events this flow definition has been annotated with.

`get_model(arr=[], dbg=False)`

returns a tuple of model element that have annotated this flow (see flow_match())

`class fseq`

`fseq(*args)`

class representing a sequence of flows. for a *fseq* to match a sequence of events in the model, each sub-flow must match a set of events, and the first matching event in a subflow must follow (but not necessarily immediately) the last matching event in the previous flow.

Parameters

`*args(flows)` : the flows to be matched in sequence

Example

Suppose the model has a sequence of events on entity *A* and on labels *e1..e5*. The following flow definition would match events with odd indices:

```
fseq(fev("A", "e1"), fev("A", "e3"), fev("A", "e5"))
```

`get_min_model_index()`

returns the minimum among the index of the events this flow definition has been annotated with.

`fint(src, dst, label, src_predicate=None, src_predicate_arg=None, dst_predicate=None, dst_predicate_arg=None, src_action=None, src_action_arg=None, dst_action=None, dst_action_arg=None, src_description=None, dst_description=None, src_flow=None, dst_flow=None)`

returns a flow specification for an interaction.

Parameters

`src``(String)` : regexp for source entity ID

`dst``(String)` : regexp for destination entity ID

`dst``(String)`
regexp for label associated to source and destination event

`src_predicate``(function)`
optional boolean function to be invoked on the source event in addition to matching entity and label

`src_predicate_arg``(any)`
a value passed to the src_predicate function

`dst_predicate``(function)`
optional boolean function to be invoked on the destinatione vent in addition to matching entity and label

`dst_predicate_arg``(any)`
a value passed to the dst_predicate function

`src_description``(String)`
optional human readable representation of the source event

`dst_description``(String)`
optional human readable representation of the destination event

`src_flow``(String)`
An interaction creates a new branch on a flow. This element allow to specify the branch continuing on the source entity

`dst_flow``(String)`
An interaction creates a new branch on a flow. This element allow to specify the branch continuing on the destination entity

Example

Suppose entity *S* sends a message to entity *D* on label *msg*. After that *S* has an event with label *e1* and *D* has an event on label *e2*. This can be expressed as follows:

```
fint("S", "D", "msg",
        src_flow=fev("S", "e1"),
        dst_flow=fev("S", "e2"))
```

6.2.4 Graph API

This module contains the Pthon API to interact with MSCViewer graphing capabilities.

Even though Python scripts may have access to the full Java API directly, this module defines a stable interface that will guarantee compatibility in the future. Internal Java

APIs may, and most likely will, be subject to change, so the user is strongly advised against its direct usage.

6.3 Flow Definition

MSCViewer provides a powerful infrastructure supporting automated flow validation. This means that a developer can write once a concise Python script describing an expected flow, and apply such definition to models built from traces generated a runtime by a system. This is the basis for flow-level verification automation, which is crucial to scale at least the first level of triaging for a system.

6.3.1 Your First Flow Definition

File `examples/flw1.py` contains a simple flow definition script. Let's look at it in detail:

```
 1  from msc.flowdef import *
 2  from msc.gui import msc_fun
 3  import sys
 4
 5  def get_flow():
 6      return fseq(
 7          fev("foo", "event_a"),
 8          fev("foo", "event_b")
 9          )
10
11  @msc_fun
12  def find_flow():
13      f = get_flow()
14      idx = 0
15      found = False
16      while idx >= 0:
17          try:
18              idx = f.match(start_event_idx=idx)
19              if idx >= 0:
20                  print "found match!"
21                  found = True
22          except FlowError, err:
23              if not found:
24                  print "No match found (", str(err), ")"
25              break
26      return
27
28
29  if __name__ == "__main__":
30    find_flow()
```

Lines 1, 2 import some packages required for flow definition. These are defined in Python files contained in the `resources/default` directory of the MSCViewer install directory.

Lines 5-7 define a function returning a flow definition. The flow definition is specified as a sequence (`fseq`) of two events (`fev`). The two parameters for each `fev()` element are respectively the entity id and the label of the events. A model containing an ordered sequence of two events on an entity named *foo*, the first one with label `event_a` and the second with label `event_b` does contain the flow specified by function `get_flow()`. Function `get_flow()` returns the defined flow as an object.

Function `find_flow()` at line 12 calls such function to retrieve the flow definition and repeatedly calls function `match()` on it. the `match()` function attempts to match the flow definition to the model currently loaded in MSCViewer, starting with the model event with the index specified as the `start_event_idx` parameter. In case of match, the function returns the index of the last event in the model matching one of the elements in the flow definition, plus one. If the entire flow definition is not found in the model, then the function throws a FlowError. If passed a starting event index corresponding to the total number of events the function returns -1.

The `examples/flw1-match.msc` is an example of a log file containing the specified flow:

```
@event { "entity":"foo", "label":"event_a"}
@event { "entity":"foo", "label":"event_b"}
@event { "entity":"bar", "label":"event_x"}
```

However, note that the `examples/flw1-match.msc` file also contains the specified flow:

```
@event { "entity":"foo", "label":"event_a"}
@event { "entity":"bar", "label":"something"}
@event { "entity":"foo", "label":"something else"}
@event { "entity":"foo", "label":"event_b"}
```

This is the case because `fseq` requires events to be in the specified order, but not to be contiguous in the file, or even within the entity.

We can test the scripts by running MSCViewer in batch mode:

```
$ mscviewer -b examples/flw1.py examples/flw1-match.msc
 Found match!
$ mscviewer -b examples/flw1.py examples/flw2-match.msc
 Found match!
$ bin/mscviewer -b examples/flw1.py examples/flw1-nomatch.msc
 No match found ( element (foo, event_a) not found in model. )
$
```

6.3.2 Model Elements for a Flow

In addition to knowing that a certain flow is present in a model, you may want to discover which `fev` elements match which mdodel events. This can be achieved using the `get_model_map()` function, as illustrated by the `examples/flw2.py` example:

```
from msc.flowdef import *
from msc.model import *
from msc.gui import msc_fun
import sys
```

```

def get_flow():
    return fseq(
        fev("foo", "event_a"),
        fev("foo", "event_b")
        )

@msc_fun
def find_flow():
    f = get_flow()
    idx = 0
    found = False
    while idx >= 0:
        try:
            idx = f.match(start_event_idx=idx)
            if idx >= 0:
                print "found match!"
                found = True
                m = f.get_model_map()
                for fl in m:
                    ev = m[fl]
                    print str(fl), ':', event_line(ev)
        except FlowError, err:
            if not found:
                print "No match found (", str(err), ")"
            break
    return


if __name__ == "__main__":
  find_flow()
```

The `get_model_map()` function returns a map where each `fev` in the flow definition is a key, and the event matching it is the value associated to the key. Various pieces of information can be stracted from the event using the `event_` API defined in the `msc.model` module. In this example, we print the line number where the event is defined in the input file:

```
$ bin/mscviewer -b examples/flw2.py examples/flw1-match.msc
 found match!
 fev(foo, event_b) : 2
 fev(foo, event_a) : 1
$ bin/mscviewer -b examples/flw2.py examples/flw1-match2.msc
 found match!
 fev(foo, event_b) : 4
 fev(foo, event_a) : 1
```