Distributed Processing Flow Visualization and Analysis in MSCViewer

Roberto Attias
NOSTG CRBU Advanced Tech
CISCO

October 21, 2013

Abstract

Modern core routers are heavily parallel architecture with massive computational power. Admin plane, control plane, and some data-plane functionalities are mostly software based. The software model has evolved from monolitic, single-program, to multiple processes cooperating with each other distributed across multiple nodes. More over, to avoid complex deadlock situation process communication has transitioned from a synchronous to an asynchronous model. The performance and scalability benefits of this evolution come at the cost of an increased complexity in debugging problems in execution of functionalties.

One of the critical tools for triaging is trace analysis. When functions are distributed across multiple processes and nodes trace analysis increases in complexity as execution flows need to be tracked across concurrent entities; flows are not any longer sequence of operations but rather sets of concurrent and sequential operations. A typical triage of a flow involving multiple software components requires cooperation of a number of engineers, each one familiar with a set of components and the respective traces, just to identify the component where the flow broke. This approach doesn't scale, and diverts a sizeable amount of engineering hours which could be spent in more productive ways. By capturing the most important trace events and interaction in a formal languge we can build a rich Sequence Diagram Chart, which lends itself very well to distributed system flow analysis. Visual or automatic inspection of this diagram makes it easier to identify issues, and reduces considerably triage times.

In this paper we present MSCViewer, a tool for graphical analysis of distribute entities and their cooperation in the form of dynamic sequence diagram charts. The tool, which is succesfully used to triage Panini bugs, parses traces where the most important events are captured in a formal language, building an interactive model of entities, events and interactions. The model can be inspected graphically through an interactive GUI which provides extended browsing capabilities. We also introduce a flow-specification language implemented on top of Python which allows defining expected flows in a compact form. Once defined, flows can be applied to traces to verify correctness and, in case of errors, pinpoint where the expected flow broke. Through integration with Python MSCViewer can present successful and failed flows graphically, or execute verification in batch mode. Formally capturing the knowledge about expected flows and automatically verifying them allows reducing engineers involvment at least in the first fases of triaging, where the culprit component is identified.

1 Introduction

Modern core routers are complex system where a large part of the functionalities is implemented in software. From a single-process/single-threaded model systems have evolved to multiple threads, multiple processes, and even multiple nodes operating cooperatively. Even when processes are single threaded, they often carry out asynchronous comunication and react to events, effectively handling concurrent execution contexts within a single thread. Control plane, admin plane and to some extent data plane implement multiple functionalities through transactions and interactions across different processes and threads. Some thread may cooperate to carry out multiple occurrencies of the same or different functionality. Hence, verifying the correct execution of a functionality is a complex task, typically requiring engineers to examine traces produced by various processes. Due to the complexity of the system very few engineers are familiar enough with the internals of all the processes involved, and therefore able to confidently interpret all traces. Thus triaging of bugs often requires multiple engineers attention for a considerable amount of time, diverting those engineers from more productive tasks.

This reliance on engineers for doing even the first level of triaging is very expensive and not scalable. In this paper we present a solution addressing this problem. The solution is based on emitting traces in a formal language where events and interactions are captured, and from which a model can be automatically built. Through a second formal language we support specification of expected flows for various functionalities. Finally, we provide a tool which can apply flow specifications to the model created from the traces, allowing to do both visual inspection of flows in the form of sequence diagram charts as well as GUI-less verification, for use in automatic tests.

MSCViewer is a message sequence chart visualization and analysis tool. The tool was created in NOSTG in the context of the NG-XR admin-plane development, and it is part of a set of technologies known under the name of RISE (Reusable Infrastructure Software Elements). while MSCViewer is indepenent from other RISE elements, such elements can be easily integrated in new or existing application to leverage the power of Message Sequence Chart analysis efficiently and with minimal effort. A brief overview of RISE will be provided in section 7.

2 Sequence Diagram Model

MSCViewer accepts as input text files containing statements in a Sequence Diagram Modeling Language (SDML) (see section 3) and builds a model with the following elements:

2.1 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, an ID and a set of events. Entities with the same parent entity must have a unique ID. This guarantees that the hierarchical path of entity IDs for each entity is going to be unique in the entire model.

Let's consider the example shown in table 2.1:

Name	ID	ID Path	Name Path
RP0	0	0	RP0
RP1	1	1	RP0
DS	1234	0/1234	RP0/DS
DS	1236	0/1236	RP0/DS
DS	1234	1/1234	RP1/DS

Table 1: Example of entities

Here we have three instances of a process named DS, The first two processes have ID 1234 and 1236 respectively, and they're runnning on a node whose name is RP0 and ID is 0. The third process has the same ID as the first, but a different parent. Note how each process has a unique ID path, even though some names and even name paths collide. 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. 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.

2.2 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 4.8). 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.

2.3 Interaction

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

- ev_i is called the *source event*, and the entity the event belongs to is called the *source entity*;
- ev_j is called the *sink event*, and the entity the event belongs to is called the *sink entity*
- *type* characterize the nature of the interaction.

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

Note also that the model allows source and sink 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 sink eventcaptures 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 sink events belong to the same entity is a state machine transition. In this case the source event can represent the action of leaving the from-state, while the sink event can represent reaching to-state. Note that the model allows for additional events to be present between the source and sink event (for example to represent actions taken in the transition).

In the model interactions are captured with an extra interaction ID attribute associated to the source and sink 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 only the sink to only one interaction. events with multiple outgoing interaction can be used, for example, to capture multicast messaging.

In this paper 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

2.4 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 $(ev_i \rightarrow ev_j)$
- $\exists ev_1 \ldots ev_n$ such that $ev_1 = ev_i, \quad ev_n = ev_j, \quad ev_k < ev_{k+1} \quad \forall k \in (1 \ldots n-1)$

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.

3 Sequence Diagram Modeling Language

MSCViewer is capable of loading a text file containing statements expressed in Sequence Diagram Modeling Language (SDML), intermixed with regular text. The SDML grammar can be expressed in Extended Backus-Naur Form as follows:

```
     statement = entity-dec | event-decl
   entity-decl = '@msc_entity', {key-value-pair}
    event-decl = '@msc_event', {key-value-pair}, [{source-decl | sink_decl}]
key-value-pair = identifier, '=', value
    identifier = ( letter | '_'), {letter | digit | '_'}
         value = string
   source_decl = '@msc_source' {key-value-pair}
     sink_decl = '@msc_sink' {key-value-pair}
        letter = ? [a-zA-Z] ?
         digit = ? [0-9] ?
        string = ? double-quote enclosed string ?
```

Entity declarations have a set of mandatory keys:

- `id`: the value represent an ID for the entity the event belongs to, unique across all the entities in the system.
- `display_name`: the value represent a logical name for the entity. Multiple entities can have the same display name, as long as their id is different.

Event declarations have a set of mandatory keys:

- `entity_id`: the value represents the ID of the entity the event belongs to.
- `time`: the value represents a timestamp when the event happened. The timestamp is expected to be an integer positive number.
- `label`: the value represents an event-specific label providing information about the event
- `type` the value represents a type for the event. types can be domain-specific, and correspond to a class in MSCViewer used to render the event.

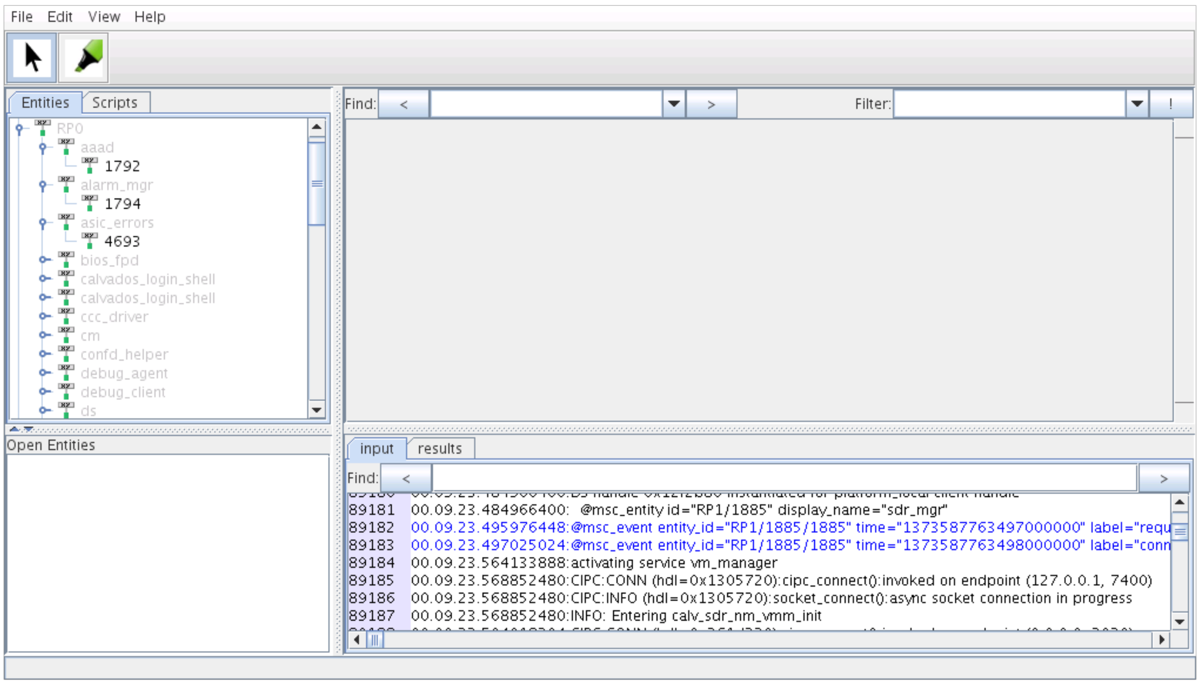
Depending on the type of the event, additional type-specific key-value pairs can be present in an event declaration.

Source Declarations and *Sink Declarations* allow pairing of events into an interaction. They both have two mandatory key-value pairs:

- `pairing_id`:the value is a string identifying uniquely the interaction this event is a source or sink for.
- `type`: the value represents a type for the interaction. types can be domain-specific, and correspond to a class in MSCViewer used to render the interaction.

When parsing the input file MSCViewer pairs into interactions events with the same pairing ID. If the same paring ID occurs more than once in the input file for a source or sink, the first source will be paired with the first sink, etc. This allows reusing the same pairing ID within an entity and over when source and sink are in the same entity.

Note that MSCViewer does not necessarily assume that source and sink events for an interaction will appear in the correct order: a topological sorting algorithm is applied to the model upon loading, as explained in section 2.4



```
00.09.23.484966400:  @msc_entity id="RP1/1885" display_name="sdr_mgr"
00.09.23.495976448:@msc_event entity_id="RP1/1885/1885" time="1373587763497000000" label="requ
00.09.23.497025024:@msc_event entity_id="RP1/1885/1885" time="1373587763498000000" label="conn
00.09.23.564133888:activating service vm_manager
00.09.23.568852480:CIPC:CONN (hdl=0x1305720):cipc_connect():invoked on endpoint (127.0.0.1, 7400)
00.09.23.568852480:CIPC:INFO (hdl=0x1305720):socket_connect():async socket connection in progress
00.09.23.568852480:INFO: Entering calv_sdr_nm_vmm_init
```

Figure 1: MSCViewer GUI main window

4 MSCViewer GUI

In this section we give a brief overview of the MSCViewer GUI, explaining the most important aspects and focusing on the provided navigation functions which allow the user to easily and effectively browse the model. Figure 1 shows MSCViewer GUI main window.

4.1 Entities Tree

In the top-left area of the window there are two tabs: *Entities* and *Scripts*.

The *Entities* tab (Figure 2) contains a tree where each node represent an entity defined in the input file. The tree reflects the hierarchy of entities. Entities may not be active, i.e. not have any event, but only act as parents for active entities, in which case they're shown in gray. Active entities are shown in black. Note that by default tree nodes are labeled with entities name, but it is possible to switch to IDs.

The *Scripts* tab contains a tree where each node represent an Python package or script. We will discuss this in detail in section 4.7

The bottom-right area of the window hosts two tabs: *Input* and *Results*.

4.2 Input View

The *Input* view shows the content of the input file loaded by the tool. when parsing the input file the tool ignores lines which do not conform to the SDDL grammar (see section **??**). Lines belonging to the language are shown in this area in blue, while ignored lines are shown in black.

4.3 Result View

The *Result* view shows HTML content typically produced by scripts executed from the *Script* view. Additional details on this view will be provided in section 6.1

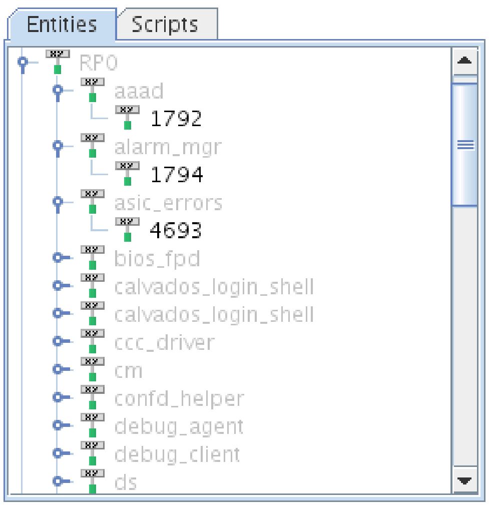


Figure 2: MSCViewer Entities view

4.4 Sequence Diagram View

the top-right part of the window hosts the *sequence diagram view*. The user can visualize any set of entities in this area, sorting them as desired and navigating through the events with mouse or cursor keys. An active entity can be opened in the view in various ways:

- by double-clicking on its node in the Entities tree;
- by double-clicking on an the line defining an event belonging to the entity in the Input view;
- by clicking on a hyperlink in a result shown in the result area;
- using cursor keys to navigate through interactions.

Figure 3 shows an example with a few entities open.

The top part, has a rectangle for each entity containing the name path of the entity. A vertical line under the box represents the existance of the entity. For entities that have a limited lifespan, the line is present only in the range of events when the entity is alive. On each life line events are drawn as they occur. Each event has an associated timestamp and an icon which is function of the event type. Interactions are usually shown as lines between their source and sink event pair, but different type of interactions are shown with different styles. For example, In Figure 3 the first entity is a state machine called `pm_service_starter:0`. at time 6:53:491 this SM instantiates a SM called `pm_role_negotiation:25`. The “creation” interaction is shown as a dotted, bold line with a green triangle to represent the sink event. As the entity starts is life at this point, the lifeline is present only below the triangle.

If an interaction involves two distinct entities, one of which is open in the view, then a stub of line is shown entering/exiting from the event on the shown entity. It is possible to open the other entity by selecting the interaction stub and choosing `Open Source` or `Open Sink` in a popup menu. In some cases only information for one of the two events belonging to an interaction may be present in the input file. This can happen, for example, if trace files are capped in size, in which case older events may be lost at the time traces are collected. Another possibility is that traces were not collect for the entire set of entities involved, in which case an entity source or sink of an event may not be present at all in the traces. When one of the two events belonging to an interaction is not present the interaction stub is drawn in a different style, to make such information apparent.

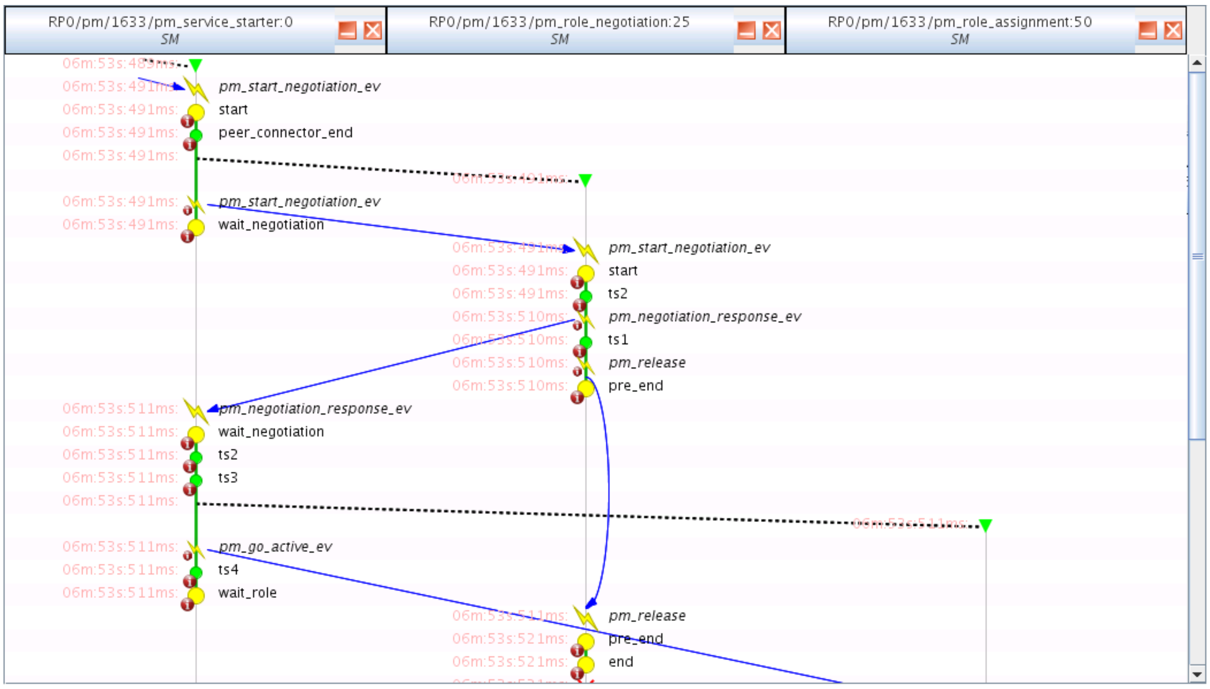


Figure 3: Sequence Diagram View

The diagram always show one event per row in the view, even when events have the same timestamp. This allows for timestamp and labels of arbitrary length never to overlap each other.

Open entities can be dragged around with the mouse to reorder them as desired. Entities can be closed via a popup menu item or the ☒ button, or reduced in size via the ▬ button

4.5 Open Entities List

The bottom-left part of the window shows a list of open entities. This list allows to quickly close a set of entities and scroll the sequence diagram view to a particular entity when too many entities are opened to fit in the window.

4.6 Model Navigation

The Sequence Diagram View allows keyboard-based navigation for efficient browsing. Once an event is selected with the mouse, the selection can be moved to the next or previous event using SHIFT ↑ or SHIFT ↓ . When an event has an outgoing or incoming interaction the selection can be moved from the event to the interaction using SHIFT ← or → , depending on the direction of the interaction. when the interaction is a stub because the other entity involved is not open, moving the selection further with the same keys causes the entity to open and the selection to move to the other event. A search box allows also to search for events based on their label, timestamp, attributes, etc.

While navigating the model the user may want to mark certain events or interactions. For this purpose a marker tool can be selected in the toolbar at the top of the GUI. the marker allows to hilight events and interactions in four different colors.

4.7 Executing Scripts in MSCViewer

MSCViewer integrates a Python interpreter which allows the user to write Python functions operating on the model and GUI and execute the function inside the viewer. This functionality is at the core of automatic flow analysis. At boot MSCViewer browses a directory containing

Python packages and scripts, importing them in the interpreter. functions decorated with the `@msc_fun` attribute are shown in the Script Tree, under their respective packages. Figure 4 shows an example where two functions were discovered.

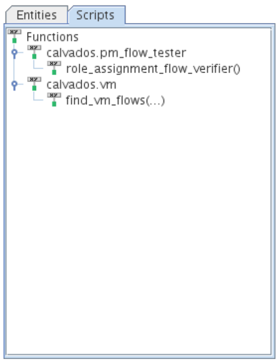
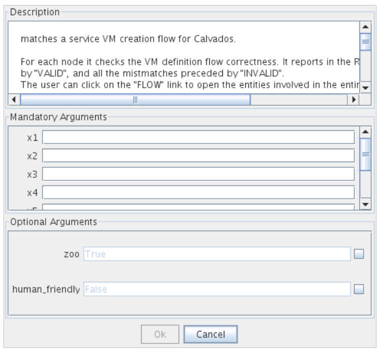


Figure 4: Scripts Tree



matches a service VM creation flow for Calvados.

For each node it checks the VM definition flow correctness. It reports in the R
by "VALID", and all the mistmatches preceded by "INVALID".
The user can click on the "FLOW" link to open the entities involved in the entir

x1
x2
x3
x4

zoo True
human_friendly False

Figure 5: Providing function arguments

Double-clicking on a function causes its execution. If the function has non-keyword arguments then the first time it is executed a window opens where the user can provide values for such arguments, as well as any possibe keyword argument, as shown in Figure 5. The provided values are remebered and used in subsequent executions. The user can bring up the parameter editing window pressing SHIFT while double-clicking on the tree node.

4.8 MSCViewer Rendering

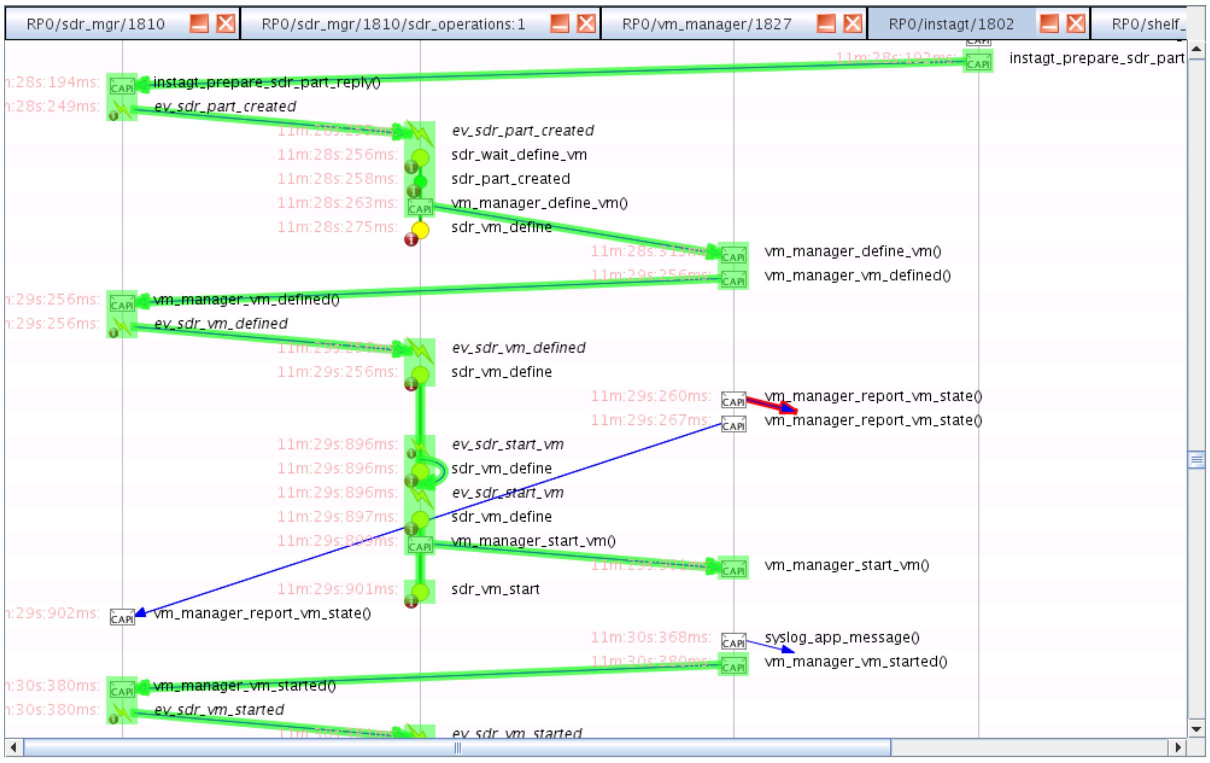


Figure 6: A flow involving multiple entities

5 Flow Modeling Language

In this section we will define a set of operators to capture expected event flows, i.e. acceptable event sequences in an input file.

Consider the sequence diagram chart snippet highlighted in figure 6.

Here multiple entities are interacting with each other in order to accomplish a task. Each event can be identified by the entity it belongs to and the label. We can use the notation `fev(`*entity*`,` *label*`)` to indicate an event occurring on an entity. For example, the first event highlighted can be represented as `fev(`“RPO/instagt/1892”`,`“instagt_prepare_sdr_part”`)`.

In the highlighted flow most of the events always happen in a well defined sequence, as they depend on sequential execution of code within an entity and transfer of control to other entities via messages. We can use the notation `fseq(`fev_1`,`$\dots$`,`fev_n`)` to represent a sequence of n events that are expected to be present always in the specified order, although not necessarily contiguous.

`fseq` has the associative property:

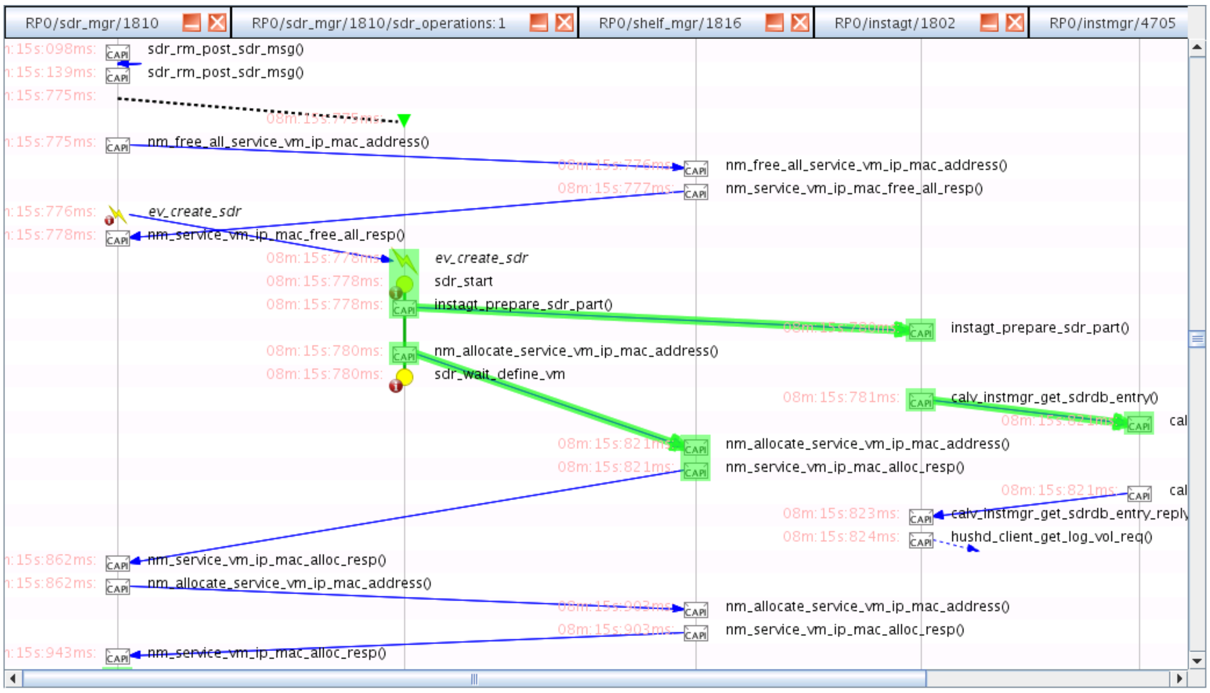
`fseq(fseq(`ev_1`,`$\dots$`,`ev_n`)`ev_{n+1}`,)` $\equiv$ `fseq(`ev_1`,`$\dots$`,`ev_{n+1}`)`

Consider now the two events

`fev(‘‘RP0/sdr_mgr/1810/sdr_operations:1’’,‘‘sdr_vm_start’’)` and

`fev(‘‘RP0/vm_manager/1827’’,‘‘vm_manager_start_vm()’’)`

In this particular execution the events happened in the listed order, however there is no explicit ordering between these two events, and their order may be the opposite one in a different run. We can use the notation `fall(`ev_1`,`$\dots$`,`ev_n`)` to indicate a set of events that must all occur, but in no particular order. `fall` has the associative and commutative properties:



RP0/sdr_mgr/1810
RP0/sdr_mgr/1810/sdr_operations:1
RP0/shelf_mgr/1816
RP0/instagt/1802
RP0/instmgr/4705
sdr_rm_post_sdr_msg()
sdr_rm_post_sdr_msg()
nm_free_all_service_vm_ip_mac_address()
nm_free_all_service_vm_ip_mac_address()
nm_service_vm_ip_mac_free_all_resp()
ev_create_sdr
nm_service_vm_ip_mac_free_all_resp()
ev_create_sdr
sdr_start
instagt_prepare_sdr_part()
instagt_prepare_sdr_part()
nm_allocate_service_vm_ip_mac_address()
sdr_wait_define_vm
calv_instmgr_get_sdrdb_entry()
nm_allocate_service_vm_ip_mac_address()
nm_service_vm_ip_mac_alloc_resp()
calv_instmgr_get_sdrdb_entry_reply
hushd_client_get_log_vol_req()
nm_service_vm_ip_mac_alloc_resp()
nm_allocate_service_vm_ip_mac_address()
nm_allocate_service_vm_ip_mac_address()
nm_service_vm_ip_mac_alloc_resp()
nm_service_vm_ip_mac_alloc_resp()

Figure 7: A flow involving multiple entities

$$\texttt{fall(}ev_1\texttt{,}ev_2\texttt{)} \equiv \texttt{fall(}ev_2\texttt{,}ev_1\texttt{)}$$

$$\texttt{fall(fall(}ev_1\texttt{,}\ldots\texttt{,}ev_n\texttt{),}ev_{n+1}\texttt{)} \equiv \texttt{fall(}ev_1\texttt{,}\ldots\texttt{,}ev_{n+1}\texttt{)}$$

In some cases we will need to specify flows where any one out of a number of events is allowed to happen. We can use the notation `fany(`ev_1`,`$\ldots$`,`ev_n`)` to indicate a set of events of which at least one has two happen. `fany` also has the associative and commutative properties:

$$\texttt{fany(}ev_1\texttt{,}ev_2\texttt{)} \equiv \texttt{fany(}ev_2\texttt{,}ev_1\texttt{)}$$

$$\texttt{fany(fany(}ev_1\texttt{,}\ldots\texttt{,}ev_n\texttt{),}ev_{n+1}\texttt{)} \equiv \texttt{fany(}ev_1\texttt{,}\ldots\texttt{,}ev_{n+1}\texttt{)}$$

With this simple building blocks it is now possible to capture fairly complex flows and their relationships. For example, consider the events highlighted in fig. 7

We can capture such flow as follows:

Listing 1: Flow specification for the flow highlighted in Fig 7

```
OP="RP0/sdr_mgr/1810/sdr_operations:1"
SM="RP0/shelf_mgr/1816/"
IA="RP0/instagt/1802"
IM="RP0/instmgr/4705"
fseq(
  fev(OP, "ev_create_sdr"),
  fev(OP, "sdr_start"),
  fev(OP, "instagt_prepare_sdr_part()"),
  fpar(
    fseq(
      fev(OP,"nm_allocate_service_vm_ip_mac_address()"),
      fev(SM,"nm_allocate_service_vm_ip_mac_address()"),
      fev(SM,"nm_service_vm_ip_max_alloc_resp()")
    ),
    fseq(
      fev(IA,"calv_instmgr_get_sdrdb_entry()"),
      fev(IM,"calv_instmgr_get_sdrdb_entry()"),
    )
  )
)
```

6 FML and Python

Each of the notations in the Flow Specification Language described in the previous section has been implemented as a Python class. For example, `fev("a","b")` represents the instantiation of a the `fev` class. The instance will have an `entity` member variable initialized to "a" and a `label` member variable initialized to "b". In the case of `fall` and `fany`, the variable number of arguments is implemented through the Python `*args` construct. In addition, class constructors have optional keyword arguments to specify uncommon characteristics of the instance: for example, the `fev` constructor has a `predicate` and `action` keywords, whose values are functions. More on these functions will be explained later.

When writing a flow as indicated in listing 1 we're actually instantiating a tree of classes. This tree has a very high resemblance to an AST: applying the tree to a list of events from a trace results in success if the events did satisfy the AST, or failure if it didn't. in case of success a Parse Tree of the matching events is built. In case of failure, only a partial tree is built, allowing to identify the missing event.

The part where this approach differs slightly from classical parsing consists in the fact that the flow elements implicitly allows for irrelevant events being present between events matching flow elements.

Also, in a regular language parsing the parser is applied to the entire input sequence. In our case instead a flow specification is matched to a sub-range of input events. Multiple different flow specifications may be applied to the same input, and a Python program can combine various flow specifications and contain some explicit Python control logic to accomplish a particular verification.

In order to apply a flow specification `f` to the input we can use the following snippet:

Listing 2: Flow specification

```
1  m = msc_get_model()
2  idx = f.match(model=m)
```

Line 1 gets a reference to the model created by MSCViewer when loading an input file. Line 2 searches for a matching of the flow specification inside the model, starting from the first event. If a matching set of events is found, the index of the last event in the model matching a `fev` is returned. In this case the events that matched can be extracted from `f` using `evs = f.get_model()`. If no match is found, an exception is thrown. The following code snippet shows the typical handling:

Listing 3: Example of Python code to apply flow spec to model

```
f = ...
m = msc_get_model()
try:
    idx = f.match(model=m)
    print "found a match!"
    m1 = f.get_model()
    # do something with model here
except:
    m1 = f.get_model()
    l = len(m1)
    if l==0:
        # no model was found. this means we reached
        # the end of the model
        print "flow not present"
        return
    else
        # got a partial match. we can do something with
        # m1 here
        print "partial match with flow"
        # idx was never returned due to the exception
        # need to try next match with first idx == smaller
        # index that matched + 1
        idx = f.get_min_model_index() + 1
```

We can now examine some practical uses of a flow definition. Consider a scenario where a system is composed by N nodes. Each node runs multiple processes interacting with each other. Let's assume, for simplicity, that three processes, called A, B and C are present in each node and participate in a node-local flow as follows:

1. A sends a message m_1 to B
2. B sends a message m_2 to A
3. B sends a message m_3 to C
4. C sends a message m_4 to A
5. when A receives both m_2 and m_4 it performs event E_1

At a first approximation this can be captured by the following flow specification:

Listing 4: Flow specification

```
fseq(
  fev("A","m1"),
  fev("B","m1"),
  fev("B","m2"),
  fpar(
    fseq(
      fev("B","m3"),
      fev("C","m3"),
      fev("C","m1"),
      fev("A","m1"),
    ),
    fseq(
      fev(A,"m2"),
    )
  )
)
```

However, if we have multiple nodes with the same processes, A B and C cannot be valid entity identifiers, as those have to be unique. A reasonable identifier can be composed by a node uinque id (for example a mac or IP address, or just a progresive number), a process name and OS process ID (unique within the node). For example, we could have: $1/A/1023$, $1/B/1025$, $1/C/1027$, $2/A/1025$, $2/B/1021$, $2/C/1025$, etc. While this choice makes it relatively easy to create unique IDs, it presents a problem in writing the flow specification. How can we capture A, B and C for the same node, given that PID can change in every trace, and how do we validate the flow on all nodes? To address this, `fev` allows for entity names (and in fact events too) to be specified as regular expression, containing also variables that can be replaced before evaluating the flow. We can rewrite listing 2 as follows:

Listing 5: Flow specification with regexp for entity names

```
A = "$node/A/[0-9]+"
B = "$node/B/[0-9]+"
C = "$node/C/[0-9]+"
fseq(
  fev(A,"m1"),
  fev(B,"m1"),
  fev(B,"m2"),
  fpar(
    fseq(
      fev(B,"m3"),
      fev(C,"m3"),
      fev(C,"m1"),
      fev(A,"m1"),
    ),
    fseq(
      fev(A,"m2"),
    )
  )
)
```

We can then rewrite the code in listing 3 as follows:

Listing 6: Example of Python code to apply flow spec to model

```
f = ...
nodes = msc_top_level_entities()
m = msc_get_model()
for node in nodes:
    vars = {'node':n.getPath()}
    f.setvars(vars)
    idx = 0
    while idx >= 0:
        try:
            idx = f.match(model=m, start_event_idx=idx)
            print "found a match!"
            m1 = f.get_model()
            # do something with model here
        except:
            m1 = f.get_model()
            l = len(m1)
            if l==0:
                # no model was found. this means we reached
                # the end of the model
                print "flow not present"
                break
            else
                # got a partial match. we can do something with
                # m1 here
                print "partial match with flow"

                # idx was never returned due to the exception
                # need to try next match with first idx == smaller
                # index that matched + 1
```

```
30                  idx = f.get_min_model_index() + 1
```

The code retrieves a list of the top-level entities in MSCViewer model (line 2), which correspond to nodes. it then iterates on the nodes (line 4), and for each creates a dictionary containing a single entry mapping the 'node' key to the path of the entity (line 5). The dictionary is then appplied to the flow definition (line 6): this will cause every appearance of the `$node` variable in the flow definition (see listing 5) to be replaced with the path of the node when the match function is applied (line 10). Note that for each iteration on a node the match function starts traversing the model from the beginning, thanks to the use of the `start_event_idx` parameter. When model traversal is completed for one node, the code procedes to the next node (line 21).

6.1 Visualizing Verification results in MSCViewer

In the previous section we described how to capture flow specifications and how to apply them to a model built from a trace. The last aspect still to explore is how to present the results of a flow verification in a useful way.

For a flow verification applied in batch mode a simple print in case of success or failure may be sufficient.

When applying a flow interactively in MSCViewer however the user should be able to easily navigate to the model elements that matched the flow definition, or to the partial match in case of failure.

In order to support this, MSCViewer API offers an `msc_results_add_flow` function which adds a representation of a flow and the associated full or partial matching model into the *result* view in the tool. In addition to that, it is possible for a script to highlight model elements, as showing in the following listing:

Listing 7: Example of Python code to apply flow spec to model

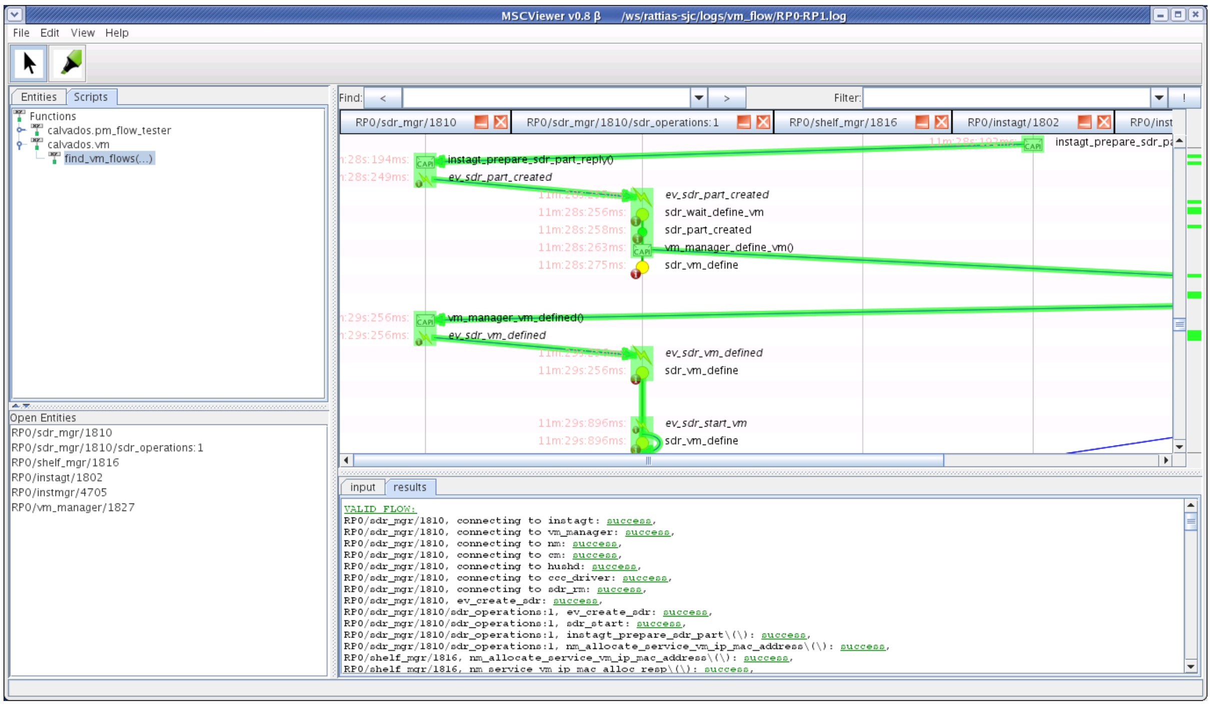
```
1  f = ...
2  nodes = msc_top_level_entities()
3  m = msc_get_model()
4  for node in nodes:
5      vars = {'node':n.getPath()}
6      f.setvars(vars)
7      idx = 0
8      while idx >= 0:
9          try:
10             idx = f.match(model=m, start_event_idx=idx)
11             m1 = f.get_model()
12             msc_flow_mark(m1, msc_color.GREEN)
13             msc_results_add_flow(f, prefix ='<span style="color:#008000;">VALID
                   FLOW:</span><br>')
14         except:
15             m1 = f.get_model()
16             l = len(m1)
17             if l==0:
18                 # no model was found. this means we reached
19                 # the end of the model
20                 print "flow not present"
21                 break
22             else
23                 # got a partial match.
24                 msc_flow_mark(m1, msc_color.RED)
25                 msc_results_add_flow(f, prefix='<span
                       style="color:#FF0000;">INVALID FLOW:</span><br>')
26
```

```
27                  # idx was never returned due to the exception
28                  # need to try next match with first idx == smaller
29                  # index that matched + 1
30                  idx = f.get_min_model_index() + 1
```

This code marks the model elements for a succesful flow match in green (line 12), and add the flow the the result view (line 13). The `msc_result_add_flow()` function creates a snippets of HTML where the specified prefix is used for a link to all the model elements associated to the specified flow. Each element will be shown also as a link to the single corresponding model elements. The user can click on the `VALID FLOW` link to open all the entities interested in the flow, or the single flow element to open just the corresponding entity.

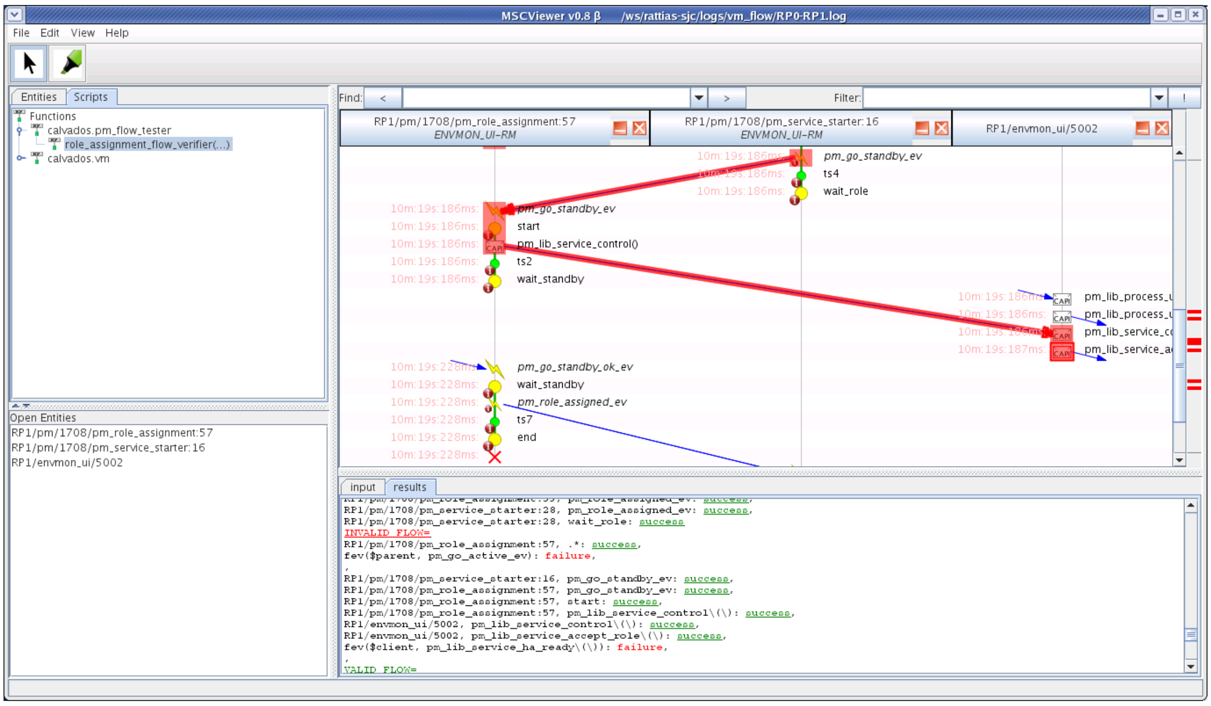
Figure 8. Show an example of output presented in the results view for a valid flow, and the coresponding model element visualized by clicking on the `VALID FLOW` link.

Figure 9. Show an example of output presented in the results view for an invalid flow, and the coresponding model element visualized by clicking on the `INVALID FLOW` link.



```
VALID FLOW:
RP0/sdr_mgr/1810, connecting to instagt: success,
RP0/sdr_mgr/1810, connecting to vm_manager: success,
RP0/sdr_mgr/1810, connecting to nm: success,
RP0/sdr_mgr/1810, connecting to cm: success,
RP0/sdr_mgr/1810, connecting to hushd: success,
RP0/sdr_mgr/1810, connecting to ccc_driver: success,
RP0/sdr_mgr/1810, connecting to sdr_rm: success,
RP0/sdr_mgr/1810, ev_create_sdr: success,
RP0/sdr_mgr/1810/sdr_operations:1, ev_create_sdr: success,
RP0/sdr_mgr/1810/sdr_operations:1, sdr_start: success,
RP0/sdr_mgr/1810/sdr_operations:1, instagt_prepare_sdr_part\(\): success,
RP0/sdr_mgr/1810/sdr_operations:1, nm_allocate_service_vm_ip_mac_address\(\): success,
RP0/shelf_mgr/1816, nm_allocate_service_vm_ip_mac_address\(\): success,
RP0/shelf_mgr/1816, nm_service_vm_ip_mac_alloc_resp\(\): success,
```

Figure 8: A valid flow shown in the result view and diagram



```
RP1/pm/1708/pm_service_starter:28, pm_role_assigned_ev: success,
RP1/pm/1708/pm_service_starter:28, wait_role: success
INVALID FLOW=
RP1/pm/1708/pm_role_assignment:57, .*: success,
fev($parent, pm_go_active_ev): failure,
,
RP1/pm/1708/pm_service_starter:16, pm_go_standby_ev: success,
RP1/pm/1708/pm_role_assignment:57, pm_go_standby_ev: success,
RP1/pm/1708/pm_role_assignment:57, start: success,
RP1/pm/1708/pm_role_assignment:57, pm_lib_service_control\(\): success,
RP1/envmon_ui/5002, pm_lib_service_control\(\): success,
RP1/envmon_ui/5002, pm_lib_service_accept_role\(\): success,
fev($client, pm_lib_service_ha_ready\(\)): failure,
,
VALID FLOW=
```

Figure 9: An invalid flow shown in the result view and diagram

7 RISE Overview

MSCViewer was designed and developed as part of a set of Reusable Infrastructure Software Elements (RISE). While no other elements are required to use MSCViewer, some of the elements have a level integration with MSCViewer and features that may facilitate the job of instrumenting an application to generate traces catering this tool.

Among the most important RISE elements:

- **CAPI** – CAPI is a generator of asynchronous, point-to-point, version change resiliant APIs for processes. Given an XML file descibing functions signature, Google Protocol Buffer specifications of exchanged messages a build tool generates a client and server side library. A server links the server library and provides callback implementations for function called by the client. The client links the client library and provides callbacks for functions called by the server. CAPI generated code takes care of marshalling, unmarshalling and dispatching. The code is also instrumented with calls to *ctrace*, a tracing infrastructure part of RISE. Part of these calls emit traces in the Sequence Diagram Modeling Language (SDML), representing interactions among entities.

- **SMIL** – SMIL is a framework for definition and generation of finite state machines. The framework includes a generator that takes an XML definition of the state machine in input and produces as output executable C code and documentation. The C code consists for the most part of functions to setup machines, feed events to advance their state, both syncrhonously or asynchronously. The model allows for various callback functions to be plugged in the model, for example to evaluate guards or execute actions when a transition is taken. The SMIL generated code is also instrumented with ctrace with some of the traces in SDML format. These traces show state machine instances as entities, and transitions as Sequence Diagram events.

- **CTRACE** – CTRACE is a framework for high performance, low-footprint tracing. Similarly to other tracing frameworks developed in CISCO such as LTRACE, CTRACE maintains circular buffers of traces, storing only data arguments and not processed format strings in memory. Through compile-time techniques CTRACE avoid the requirement of manual definition of tracepoints and decode functions affecting other tracing technologies (LTRACE), still maintaining extremely high performance. Memory footprint is minimized by using variable size records and embedding strings in the records themselves. Traces are decoded on or off target, with a single tool which parses binary dumps of the circular buffers as well as decode files automatically generated at build time.

  CTRACE offers extremely high performance, which is important due to the extra tracing requirements of MSCViewer

A Useful Links

CTRACE wiki	`http://twiki.cisco.com/Main/RiseCTRACE`
CAPI spec	`http://wwwin-eng.cisco.com/Eng/OIBU/Panini/SW_Specs/Calvados_API_Spec.docx`