4200, Saint-Laurent blvd., #680
Montreal, Quebec, Canada
H2W 2R2
Montreal: +1 514-394-0737
Fax: +1 514-394-0741
Toll-free (Canada/USA): 1-888-666-5494
E-mail: alexandre.montplaisir@efficios.com
http://www.efficios.com

Design Proposal: Refactoring of Trace Compass Control Flow View

Document prepared by : Alexandre Montplaisir
Date : May 15th, 2016

Introduction

This report presents a design proposal consisting of refactoring the existing Control Flow View[1] of the Trace Compass[2] trace viewer. The main idea is to split the model of the view into separate, non-UI components.

The report is intended for the Trace Compass development community and other interested parties. It's aimed to be a “Request for Comments”, so that the elements of this report can be discussed and iterated on until we reach a solution that is acceptable to all parties. After that, EfficiOS will be able to move forward with the implementation work.

The first section of the report will present the current implementation and highlight some of its limitations. We will then spell out the design goals of the new proposal, followed by the expected advantages.
Then we will specify the scope of the proposal, which will in fact indicate what this proposal *is not*.
Finally we will lay out some architecture details.

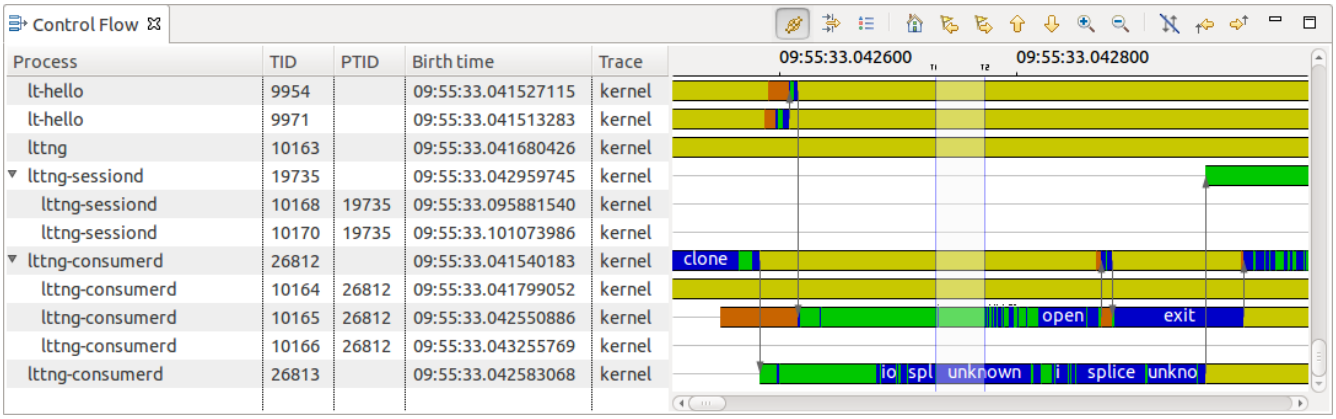

Illustration 1: The Control Flow View

1 http://archive.eclipse.org/tracecompass/doc/stable/org.eclipse.tracecompass.doc.user/LTTng-Kernel-Analysis.html#Control_Flow_View
2 http://tracecompass.org/

Overview

Current situation

The Control Flow View currently resides in the *org.eclipse.tracecompass.analysis.os.linux.ui* plugin in the Trace Compass source tree. It extends the *AbstractStateSystemTimeGraphView* class, which itself is in the core *org.eclipse.tracecompass.tmf.ui* plugin.

The view, along with its super-classes, contain the logic to query the state system for the required information, as well as doing the relevant UI operations to draw the state intervals on the screen. Some of it is separated in different classes and subclasses, for example *AbstractTimeGraphView* vs *AbstractStateSystemTimeGraphView*. But since the inheritance pattern is used, the resulting view object ends up inheriting of all that behavior and all those responsibilities.

This situation results in very big objects whose responsibilities are very wide. This makes the code complex, but more importantly, hard to modify. Adding new features can prove rather difficult, and often requires updating a plethora of related classes and views.

Proposal

We hereby propose a redesign of these classes to split these responsibilities into separate components. The high-level summary is as follows:

- Implement a model layer, which will produce model, or "render" objects.
  - These render objects will contain all the information required to populate the view for a given zoom level and set of filters.
- All the state-system-querying logic would be contained into the model layer, so that the view does not have to know about the data back-end.
- Finally, this model layer shall reside in the *core* plugins, so that it is independent from any view implementation.

Advantages

The expected advantages of this approach are:

- Better separation of responsibilities between layers
  - The model layer handles the state system queries, and produces model objects. The view layer just has to draw the model objects onto the screen.
- Increased flexibility with respect to the view implementations
  - Will allow implementations in different GUI toolkits
  - Opens the door to operations like exporting to SVG or PNG without calling any UI components, which means it could be done directly from the command-line.
- Better maintainability
  - Changing something in one layer should not affect the other, since they will be isolated by a new interface.
- Easier to add new features
  - New features for time graph views could be added to the model layer first, before the view implementation even supports it.
  - In the long term, changing the view implementation (to use OpenGL for example) would be much easier to do than it is today.

Example implementation in the Trace Compass code base

Some of the ideas expressed so far have previously been implemented in the so-called "LAMI" charts, where a core model object (*LamiChartModel*) contains all the view configuration, and the UI object (*LamiXYChartViewer*) receives a model object, and takes care of drawing it onto the screen.

For reference, these two classes are part of the *org.eclipse.tracecompass.analysis.lami.core* and *.ui* plugins, respectively.

Scope

This section will specify the scope of the current proposal. In other terms, it will identify points that are explicitly *not* covered.

The end goal is to implement a Control Flow View, based on a separated model as explained earlier, that will keep all the features of the current one, with **one** exception: the extra columns in the table on the left side of the view. This will be explained and justified below.

The current proposal covers up to the following points:

Only the Control Flow View for now

There are many other time graph views, as well as abstract implementations. The new model-plus-view concept will be applied to a new Control Flow View implementation. Additional views, as well as separation into abstract and implementing classes, are planned to be done separately.

Only one column on the left-hand side

As mentioned at the beginning of this section, we are planning to remove the concept of table from the left side of the view, and keeping only a single column. This brings the view closer to what the current Resource View looks like.

If we look at the contents of the current columns, and their planned fate:

- The TID will be part of the main column, along with the process name.
- The "PTID" will become redundant with the new tree organization, see below.
- The process birth times are useless for users, and will be removed.
- The trace name could instead be moved to be a tree element, a bit like the Resource View does.

Since the "time graph" part gets aligned with other time-aligned views, those columns often end up hidden under the time graph anyway, because users make that part bigger to see more of the interesting information.

Doing so will also avoid bugs like this one: https://bugs.eclipse.org/bugs/show_bug.cgi?id=490240

Thread information that is still deemed useful, like PPID, could be shown in a mouse-over pop-up instead.

It is currently possible to sort the tree using any of the columns. Sorting by both process names and TID should remain possible. Since there will be no columns, a separate View Properties dialog shall be added to configure the tree sorting order. This dialog could also be used for an eventual filter configuration.

New tree hierarchy

Along with the removal of the additional columns, we also propose reworking the hierarchy of the elements in the first column.

The current implementation shows the threads according to their parent-child relationships. Discussions with some users revealed that this is not particularly useful. In addition, traces don't usually have enough information to display the parent-child relationships of all threads, so many entries end up being shown all at the top level, like a flat list.

At the other extreme, a very long hierarchy can end up offsetting the text labels very far to the right, taking over space that could be used for the time graph (or simply going under it and requiring a scroll bar).

What we propose instead is to organize the threads by their thread groups, which corresponds to user-space processes. This will bring the view closer to how the *htop* tool shows processes and threads:

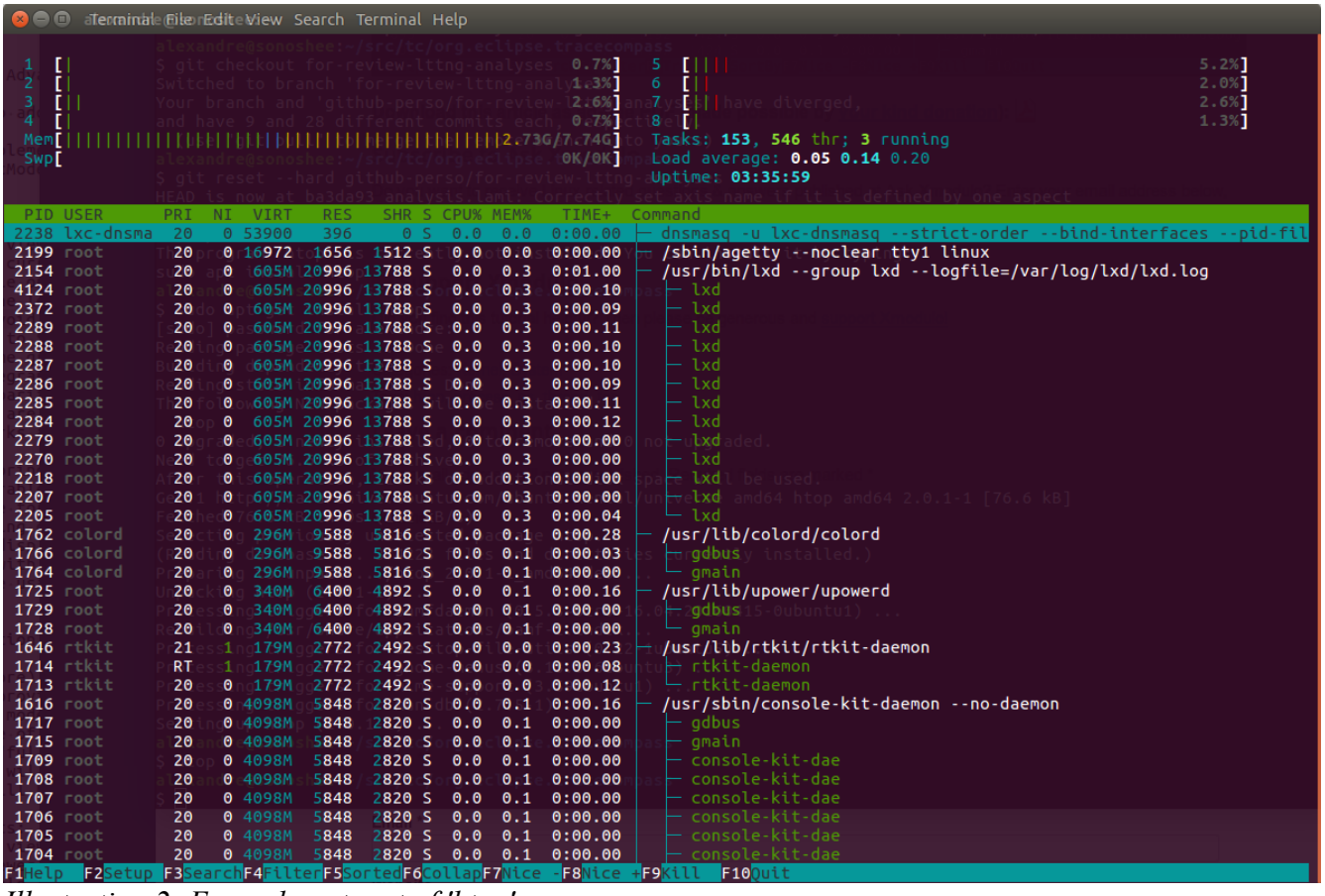

Illustration 2: Example outpout of 'htop'

The main thread of a process is shown on the first level, and other threads for the same process are shown as children of the main thread. Although note that *htop* also shows parent-child threads as sub-

tree elements. But here we suggest only using two levels, for threads of the same process. Doing so will keep the tree width under control, and will avoid shifting the text labels too far to the right due to very long hierarchies.

The model elements per se shall not be restricted to only two levels. But it should be a design guideline for views to not expand the tree to an arbitrary number of levels.

A future possible improvement that was brought up is to change the first-level elements to be simple aggregators of the states of the threads underneath it. That way, a collapsed tree element could still show an aggregate state of all the threads of this process.
The current proposal does not include doing this change, but only to have main threads and sub-threads like *htop* does. However it leaves the door open for this future improvement.

Flattening of the tree

Just like the current Control Flow View, it should remain possible for the user to completely flatten the tree, thus showing all the threads in the system as a flat list. Other view implementations could define their own criteria for flattening.

In this mode, filtering the entries in the view will allow showing only the given threads, and not their parent processes.

View model specific to a state system back-end

The view model we propose to introduce shall be specific to using a state system as a data back-end. By "state system" we mean the data structure present in the *org.eclipse.tracecompass.statesystem.** plugins.

Currently, most time graph view implementations in Trace Compass, including the Control Flow View, are based on state system back-ends. As such, it makes sense for a first iteration to assume a state system as a data back-end.

If the need ever arises, it would be possible to extract a new interface on the model side: a generic "time graph model provider" could be defined, with sub-implementations specific to different data back-ends. But again, the views per se would be completely oblivious to how the data is gathered into the model.

Feature parity with the existing Control Flow View

As we mentioned previously, we aim to keep the same feature set as the current Control Flow View (with the exception of the table columns). There are two more features that were not covered so far: arrows between states, and the display of bookmarks.

The Control Flow View currently displays arrows to represent the execution flow of the CPUs, with the

option to hide them. Such arrows could also be used to represent IPC or wakeups.
In the new paradigm, the arrows shall be part of the view model, and the view shall be able to read and display them. This will allow defining arrows a bit more generically, so that future views can re-use this feature.

On the other hand, the bookmarks are not part of the view's model. The bookmarks themselves are handled elsewhere in the framework, and the view simply paints an overlay to show them. In the scope of this proposal, the view model should not know about bookmarks at all. However the view itself should be able to display the bookmark overlay, similarly to what is done today.

Architecture and Interfaces

This section will list the main components of the new model layer, along with their proposed API. This is not a definitive nor exhaustive list, but rather a general idea of how the components will work and interact with each other.

The class names shown here are very simple, real class names will follow a better naming scheme, probably including the *ITmf-* prefix.

We will first present the model objects, then the UI objects, followed at the end by a small section related to the handling of time synchronization signals.

Model Architecture

Model provider

The model layer, which will be completely contained inside core plugins, will consist of a *model render provider*, which will query the state system it it assigned to, and produce *render* objects.

```
public interface ModelRenderProvider {
    Render getRender(TmfTimeRange range);
}
```

There will be one model render provider per view. The view will request renders to the model for a given visible time range. In the future, a set of filters could also be passed to the *getRender()* method to restrict the rendering according to filters defined by the user.

Tree Model

As part of the model, a *TreeModel* object will represent the tree elements shown on the left side of the view. This tree technically only changes when new trace events are read and new elements (processes in the case of the Control Flow View). At some point it stops changing.

This object shall have a defined iteration order, which will correspond to the ordering of the entries in the view.

It is not obvious at this time if the elements per se should be organized as a List or as a tree of *TreeElements*, each one having pointers to its immediate children.

View configuration

The mapping between states and their graphical representations, like colors, line thicknesses, etc. shall be part of the model and stored as a separate configuration object.

Initially, this configuration will be hard-coded, but a future planned improvement is to allow the user to customize this configuration, for example assigning different colors to each state.

Having this configuration at the model level ensures that the user-defined settings will be applied to all views, for example both to a view in Eclipse or to a SVG/PNG export.
It also means that the *Render* object presented below will contain all the information required to get a complete UI render. If the user eventually modifies the view configuration, then the view will have to request a new *Render* to reflect the new settings.

Note that some configuration options can be applied directly on the fetched data, like changing color or line thicknesses. Whereas other options may require re-fetching data from the state system, like adding new attributes to the display. It should be possible to differentiate between the two by using separate classes, like *PresentationConfiguration* and *DataConfiguration*.

Renders

The render objects will be *immutable* objects, which mean their contents should never change after being created. If the view wants to display the data for another time range, it should request a new render.

The render object will be the analogous of the *LamiChartModel* mentioned earlier. It shall contain all the required information for the view, which means the elements of the tree and the list of state intervals for each element. This includes the location, color and tool tips of each displayed state:

```
public interface Render {
    TreeRender getTreeRender();
    List<List<StateInterval>> getStateIntervals();
    List<List<DisplayedEvent>> getDisplayedEvents();
    Collection<ArrowSeries> getArrowsSeries();
}
```

The *TreeRender* will simply be a "snapshot" of the *TreeModel* part of the model presented earlier. If the tree render did not change since the previous render object was requested, the view can check if the two trees are *equals*, and only redraw the tree if needed.
The *TreeRender* is an immutable object. If the tree model's changes, a new *TreeRender* object has to be created.

Event

The *Event* object would be an intermediate model element that represents a two-dimensional position along the two axes of the view, being the time and the attribute. It will hold a back-reference to the *TreeElement* representing its attribute.

It should not be confused with *ITmfEvent*, the one here is a much simpler object!

```
public interface Event {
    long getTimestamp();
    TreeElement getTreeElement();
}
```

StateInterval

The *StateInterval* objects are the equivalent of the current *ITimeEvent* objects used in time graph views. One state interval shall indicate where it starts, where it ends, the color[3] it should be displayed in, any tool tip information it should show, and any other relevant information.

The indexes of the first List should match the iteration order of the tree elements in the *TreeRender* object. As we just explained, it will be useful to check quickly if the *TreeRender* changed or not, so this is why they are passed as separate objects, and not inside a single List of tuples.
The second List will contain the various state intervals this particular attribute goes through.

The interface of a *StateInterval* itself would look like this:

```
public interface StateInterval {
    Event getStartEvent();
    Event getEndEvent();
    String getStateName();
    Color getColor(); // tuple of 3 integers
    Map<String, String> getProperties();
    int getLineThickness(); // Example future addition
}
```

Here, both *Events* are expected to belong to the same *TreeElement*. To enforce this we can have the classes's constructors take two timetamps and one *TreeElement*, instead of two separate *TreeElement* objects.

The *getProperties()* method can be used to return additional information, which can then be displayed on mouse-over or in the Properties View, for example.

3 Since we are in a core plugin, the "color" cannot be a SWT RGB(A) object or such. At this level it should be independent from even SWT. The notion of color should probably be defined manually by using three integers, for R/G/B channels.

DrawnEvent

A *DrawEvent* object would be similar to *StateInterval*, but would represent a single time stamp and not a range. It could be used to display circles or points in the view to highlight points of interest.

Its interface would be similar to that of *StateInterval*, except there would be no notion of time range, only a single *Event*.

```
public interface DrawnEvent {
    enum SymbolStyle { CIRCLE, CROSS, STAR; } // etc

    Event getEvent();
    String getEventName();
    Color getColor();
    SymbolStyle getSymbolStyle();
    Map<String, String> getProperties();
}
```

Arrow series

Finally, the *Arrow* objects will represent actual arrows that should be drawn on top of the state intervals. They will be organized into *ArrowSeries*, since there could be multiple series of arrows for a given view. An eventual drop-down menu could allow selecting which series should be displayed, similarly to bookmarks.

The API of those classes will look like:

```
public interface ArrowSeries {
    enum LineStyle { FULL, DOTTED, DASHED; } // etc

    String getSeriesName();
    Color getColor();
    LineStyle getLineStyle();
    Collection<Arrow> getArrows();
}

public interface Arrow {
    Event getStartEvent();
    Event getEndEvent();
}
```

Unlike with state intervals, here both *Events* of an *Arrow* will usually not have the same *TreeElement* root.

Note that one of the end points of an Arrow can be null, but not both. This could be used to represent

start or end points that are outside of the render's range. In such cases a special symbol for sinks or sources could be used.

Cache of Render objects

The model provider is free to keep a cache of Render objects, so that often-requested renders do not have to be rebuilt every time. For instance, a render for a completely zoomed-out view could be kept in memory at all times, so that the *Show full time range* action can always be done very quickly.

This is similar to the current concept of zoom levels in the time graph views. The render of the full trace time-range is always kept in memory, and is used to populate the view quickly whenever the full trace is selected as the visible range.

The current proposal aims to keep this functionality, so that a "full-trace" render is kept in memory. But in the future, it would also be possible to implement additional optimizations, like pre-caching renders for other zoom levels or for areas of the trace close the currently active one, so that zoom and scroll operations done by the user appear to be instantaneous.

Model providers per view or per view type

As with most views in Trace Compass currently, the Control Flow View is a singleton, meaning there can be only one instance of the view at a time. For now, since there is only one view, there will be only one *ModelProvider*.

A planned improvement is to remove this restriction, so that there can be multiple instances of one view. It is not clear yet if there should be one *ModelProvider* per view instance, or one per view *type*. This will have to be decided on when the non-singleton views are implemented.

User Interface Architecture

For the UI side of things, we plan on using a view and viewer pattern, which extend *TmfView* and *TmfViewer* respectively, as is done with many other views elsewhere.

To ease integration into the current object model, the view shall instantiate its own *ModelProvider* upon creation. Other than that, the view will not have any knowledge of the data structures underneath, and will only talk to the model using the *ModelProvider* interface. This means that as features are added into the model before they are implemented by the view, the view can simply ignore them.

The view will also keep its current render available. This render can be used for intermediate displays before the new, full render is available. For example, if the user zooms into the view, we can use the data from the current render to show zoomed-in levels, while waiting for the complete render to arrive. This is also done similarly in the current implementation.

Note about time synchronization signals

Time synchronization signals are sent from the framework when the user selects a new time range in any of the time-aware views. This ensures that all time-aware views are constantly showing the same visible and selected time ranges.

Under the new paradigm, we propose that the view retains these responsibilities. Upon reception of a signal, it would request a new render from the model provider. Upon reception of the render, it would update its display to show the new information. It would work as a *pull* mechanism, instead of the alternative where the model would *push* its updates to the view.