TMF Architecture

Version 0.1
François Chouinard

1 Introduction

This document provides a high-level description of the main components of the Eclipse Tracing and Monitoring Framework (TMF) and their interrelations. Each major component is described in more details in a separate design document.

1.1 Purpose and Features

The purpose of TMF is to:

1) Facilitate the integration of tracing and monitoring tools into Eclipse
2) Provide out-of-the-box generic functionalities/views
3) Provide extension mechanisms of the base functionalities for application specific purposes

TMF additional features:

1) Ability to interact with local and remote tracing tools
2) Ability to handle concurrent, possibly live, trace streams
3) Ability to handle traces that exceed available memory
4) Facility to integrate external, host-based libraries and analysis tools
5) Facility to integrate custom trace parsers

1.2 Framework Aspects

There are five main aspects to TMF:

1) Tool Discovery – locate trace providers and identify their capabilities
2) Tool Control – control the execution of the tool and retrieve the trace data
3) Data Retrieval and Storage – import the collected trace data and store it locally
4) Analysis and Visualization – massage the trace data into useful information
5) Application Integration – add a new application, transformation, etc, to the trace handling infrastructure

2 Tool Discovery

Tool Discovery refers to the ability to discover available tracing tools on a network and to retrieve their capabilities. This information can then be used to generically control the tracing tool.

This section was scoped out because TM/RSE and TM/TCF provide a similar set of features. In the advent that RSE and TCF would not be sufficient, this decision will be revisited.

3 Tool Control

Tool Control refers to the ability to configure and control the tracing tool.

This section was scoped out because TM/TCF provides a similar set of features. In the advent that TCF would not be sufficient, this decision will be revisited.

4 Data Retrieval and Storage

Data Retrieval and Storage refers to the ability to import the collected data from the target node and store it locally (on the host) in a format that can be handled by the framework.

This section was scoped out because TM/TCF provides a similar set of features. In the advent that TCF would not be sufficient, this decision would be revisited.

Note: Monitoring which, in the general sense, involves a relation between the Analysis and the Control functions, is not addressed in this version of the document.

5 Analysis and Visualization

Analysis and Visualization refers to the ability to interpret, analyze and visualize the trace data. This is the core part of the framework.

This section deals with the organization and responsibilities of the components of the analysis and visualization aspect of the framework. Figure 1 shows the high-level structure[1] of this aspect.

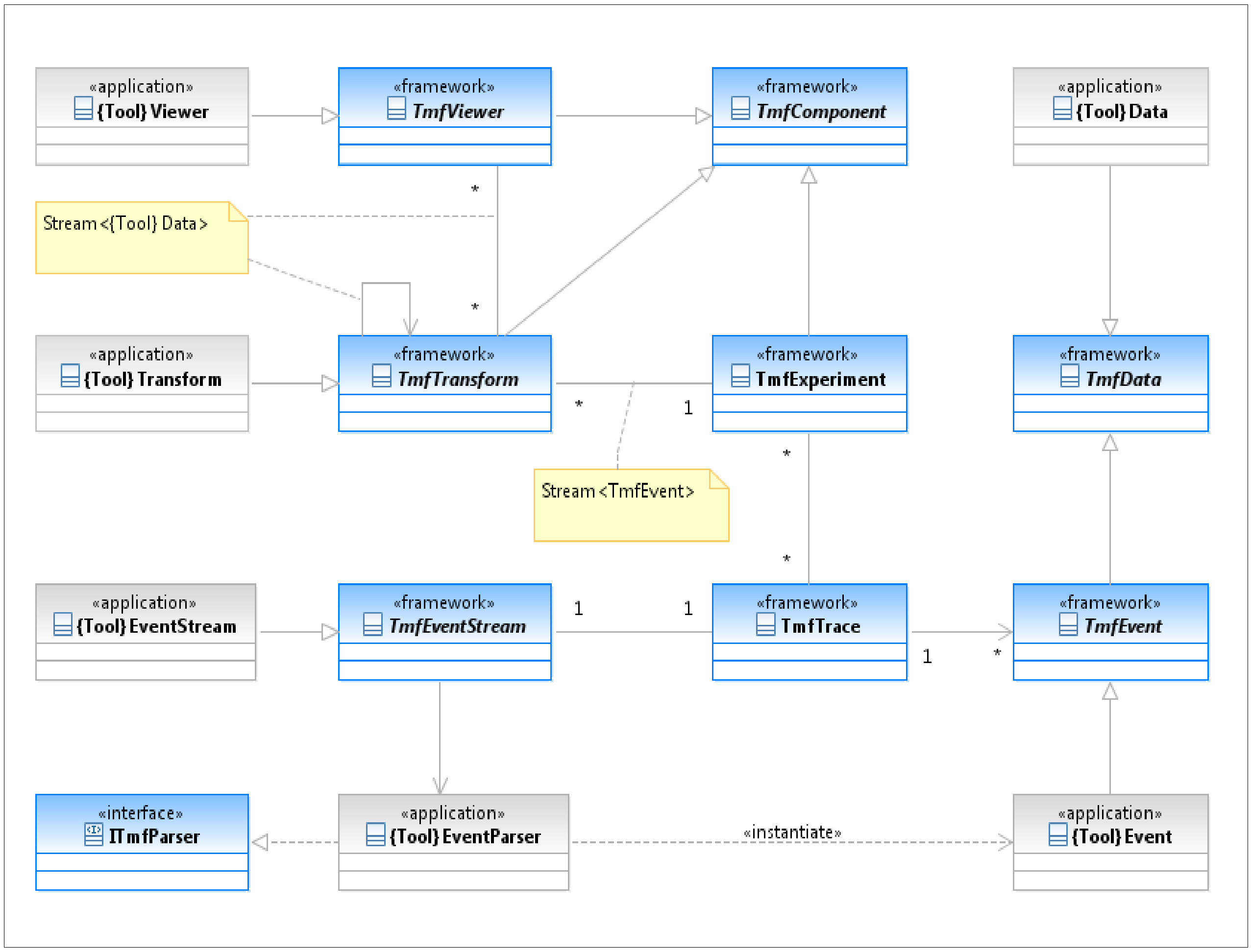
«application» {Tool} Viewer
«framework» TmfViewer
«framework» TmfComponent
«application» {Tool} Data
Stream<{Tool} Data>
«application» {Tool} Transform
«framework» TmfTransform
«framework» TmfExperiment
«framework» TmfData
Stream<TmfEvent>
«application» {Tool} EventStream
«framework» TmfEventStream
«framework» TmfTrace
«framework» TmfEvent
«interface» ITmfParser
«application» {Tool} EventParser
«instantiate»
«application» {Tool} Event
*
1

Figure 1 TMF Analysis and Visualization Structure

5.1 Static Structure

There is a one-to-one association between a TmfTrace and an EventStream. The base functionality of the stream handling is provided by TmfEventStream while the EventStream extension provides the specifics of the stream access mechanism.

1 The classes with the "framework" stereotype belong to the framework while the ones marked "application" are provided by the designer (or application/tool integrator).

It is expected that the framework will provide a set of standard stream adapters, e.g. random-access file, socket, DB, ..., in the form of a toolbox (not shown).

The TmfEventStream invokes the associated ITmfParser, implemented by EventParser, to obtain the next Event, an extension of TmfEvent. The TmfEvent provides a number of standard fields. Among these fields, the timestamp is central to the framework since it is used as the basis for locating and correlating events. The TmfTrace keeps track of the Events of a given stream in a time-ordered manner.

A TmfExperiment presents a unified view of a set of one or more TmfTraces for analysis and visualization purposes. A key responsibility of the TmfExperiment is to keep the proper ordering of the (virtually) merged trace events, against an experiment reference clock, by managing the event timestamps synchronization. A TmfExperiment can hold multiple TmfTraces and a TmfTrace can belong to multiple TmfExperiments.

A TmfTransform, implemented by an actual Transform, handles the analysis and transformation of a stream of TmfData (TmfEvents or a flavor of application Data) into a stream of application Data. The abstract TmfTransform class handles the client subscription and data flow. The application Transform handles the actual data transformation. TmfTransforms can be pipelined.

A TmfViewer, implemented by a Viewer, subscribes to one or more TmfTransforms[2] output stream and displays the TmfData in the way it sees fit, driven by the user. The abstract TmfViewer class handles the subscription to transforms and the data flow. The actual Viewer handles the data display and the user interaction.

TmfEvent and the application Data both extend the base class TmfData. This is just an artefact to standardize the data flows between the framework components.

TmfExperiment, TmfTransform and TmfViewer extend TmfComponent which handles the inter-component communication, client subscription and global events dispatch.

5.2 Data Flow

Figure 2 illustrates the main data flow between the trace and the viewer.

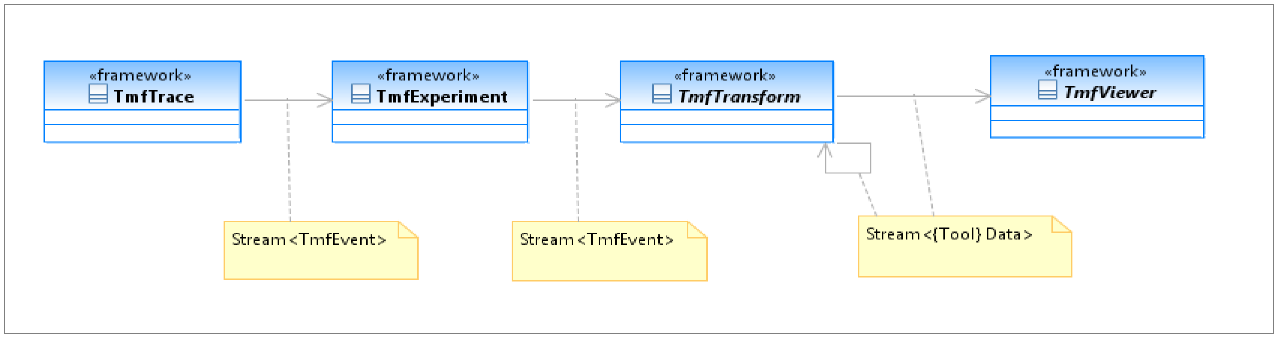

Figure 2 TMF Data Flow

2 If the viewer subscribes to multiple transforms, it is its responsibility to ensure synchronization of the various information flows.

The general structure of the framework is a pipeline: the data flows within the framework and is transformed is successive stages:

1) From raw data to a sequence of standard events (stream → trace → experiment)
2) From internal events to application data (experiment → transform [→ transform]*)
3) From application data to display (transform → viewer)

Also note that the ability to handle traces larger than the available memory is a critical constraint on the framework architecture. This has a number of important consequences for the model components:

1) No component can hold more than a reasonably small amount of data at any time
2) Data has to be streamed, a small chunk at a time, between components
3) Components have to be designed so they produce their result in one sequential pass over the incoming data

5.2.1 Inter-component streaming

The data streaming between the components (item 2 above) is achieved by implementing a generic, request-based, concurrent data handling scheme. To use it, the requesting client classes have to:

1) Instantiate a TmfDataRequest for the desired data type
2) Submit the request to the target request handler
3) Implement the methods that handle the incoming data (i.e. process the results)

The framework implements these handlers in its key components, namely: TmfEventStream, TmfTrace, TmfExperiment and TmfTransform. The functionality is inherited by the extending classes which have to implement the specifics of the data handling. See Figure 3 for the static structure and Figure 4 for an example of a simple data request processing sequence involving two request handlers.

5.2.2 TmfEventStream, TmfTrace and TmfExperiment

These three classes implement ITmfRequestHandler and manipulate TmfEvents. Their purpose is to convert the stream(s) of raw data into a time-ordered sequence of standard internal events that can then be processed by the TmfTransforms.

To minimize I/O and improve performance, these classes implement TmfEvent caching as well as request coalescing.

5.2.3 TmfTransform

TmfTransform is also an ITmfRequestHandler. However, the result of a transformation, i.e. the output stream data type, is application-dependent and not known in advance (except for the standard TmfTransforms delivered with the framework).

TmfTransforms really are both request handlers and request clients. By enforcing that the data types of the input and output stream be extensions of TmfData, the streams can be handled generically by the TmfTransform abstract class. The application Transform classes have to implement the specifics of their transformation.

The astute reader will have understood that pipelining the data into a chain of transformations simply becomes a matter of instantiating a suite of application Transforms, specify their input and output streams data types, and hook them in the proper order[3].

5.2.4 TmfViewer

A TmfViewer is a request client only and acts as an information sink. It can be treated as a simplified case of a TmfTransform where the input stream data type is not known in advance, except for the standard views delivered with the framework.

TmfViewer offers support for integrating filtering and searching, if needed.

TmfViewer also offers support synchronization i.e. when the active time range or the current time is modified, the viewers are notified and can refresh their display accordingly.

Finally, it is expected that the framework will provide a set of standard widgets that can be re-used to assemble a view or parts of a view (not shown).

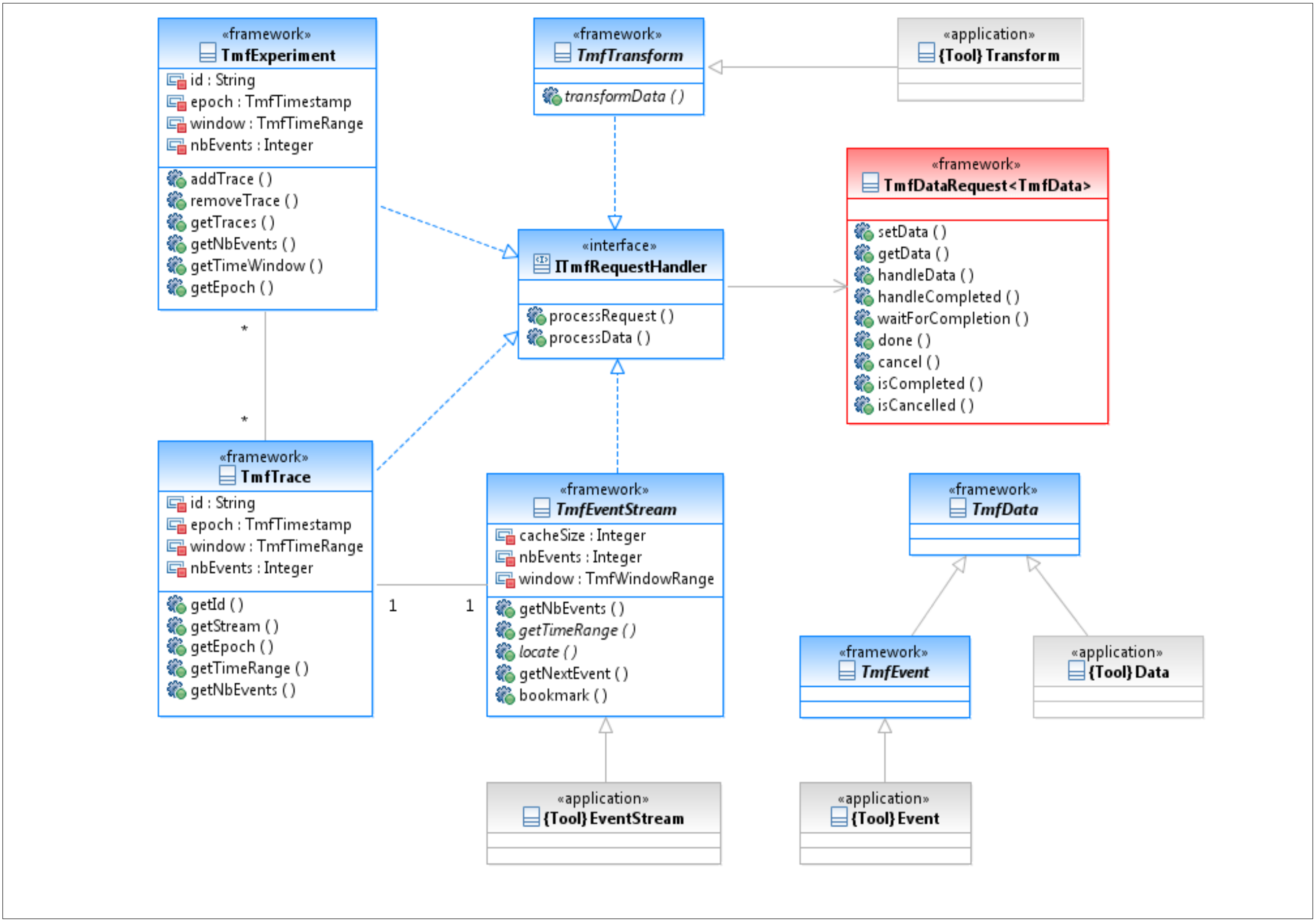

Figure 3 TMF Request Handling Model

3 It is planned to eventually enrich this model with (1) a "T-Junction" component, and (2) a non-viewer data end. This would allow the integration of orthogonal functions e.g. intermediate results storage in a database.

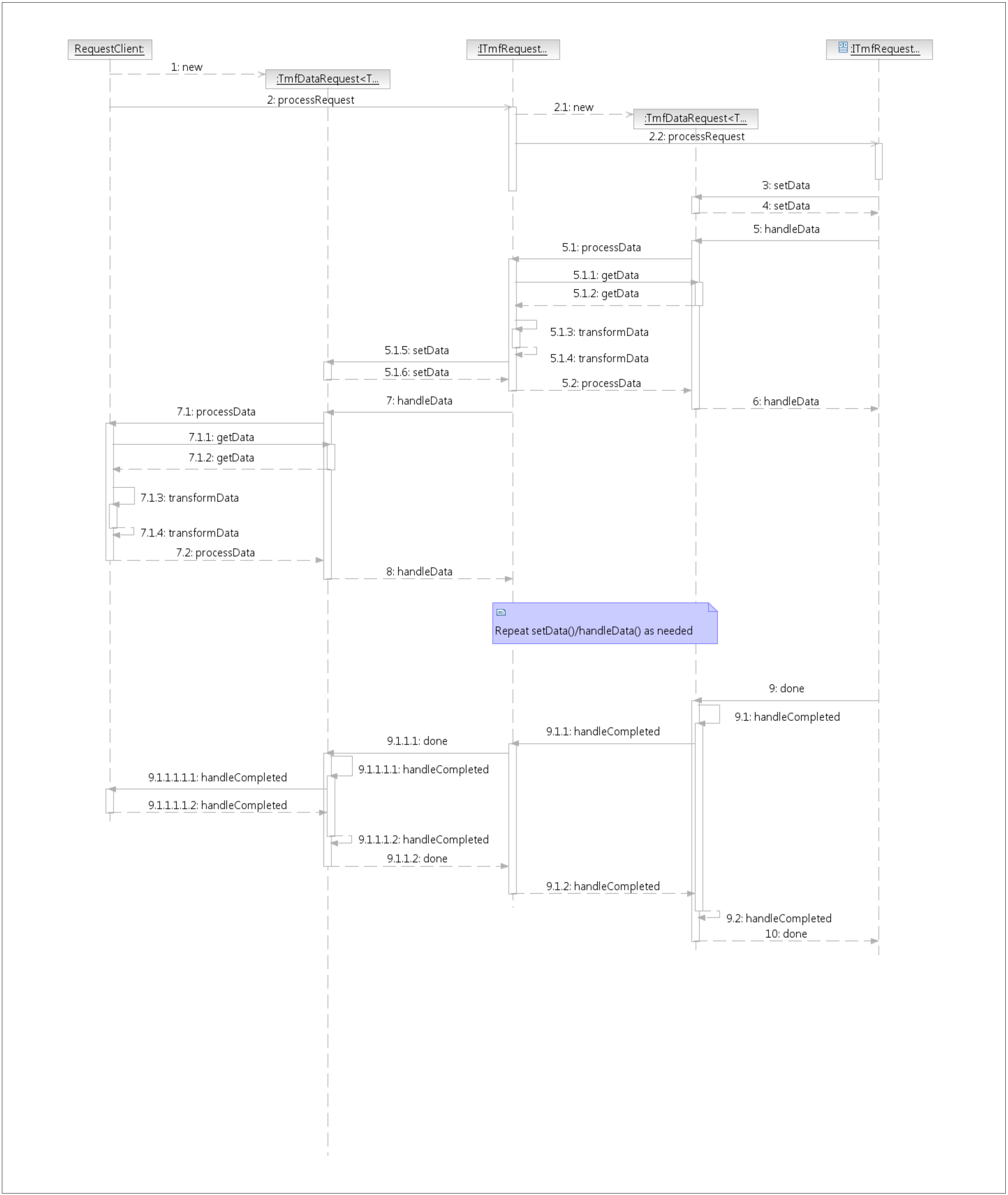

Figure 4 TMF Sample Data Request Handling

5.3 Inter-Component Communication

As a result of some trigger, an event might need to be dispatched to some or all interested components. For example, the TmfViewers can optionally be time-synchronized, and changing the time range or the current time in one view could trigger an automatic update of all the synchronized views.

Asynchronous inter-component communication is event-based and the framework distinguishes two event scopes: global and limited. The components base class, TmfComponent, provides a separate scheme for each scope.

The global scope events are handled by the TmfComponent base class. It is aware of all the instantiated components and can dispatch any event to all components. The components interested in a particular event type only have to provide an event handler for that event type.

The limited scope events are implemented using a subscription scheme where a component interested in a particular event from a target (other) component can simply add itself to the list of interested parties at the target component. On termination, the component removes itself from the target's list.

5.4 Component Management

The TmfComponent base class holds a registry of the available concrete components and the data type(s) that they provide (their outflow). In the case of TmfViewers, the outflow data type is simply 'null'. This registry is compiled at startup time using static initialization methods.

When a component is instantiated, it notifies TmfComponent which updates the component state (i.e. loaded/not loaded/...) in its registry.

5.4.1 Static Component Instantiation

With this scheme, the components are instantiated statically at startup. The application simply has to provide an initialization sequence where the components are instantiated in the correct order.

5.4.2 Dynamic Component Instantiation

With this scheme, when a given component is instantiated, it also requests TmfComponent to provide the component that generates the input it requires i.e. the producer of a specific data type flow. If the provider already exists, it is returned. Otherwise, it is instantiated by TmfComponent. Since it is possible that the newly instantiated provider has a dependency itself, the scheme is repeated as needed.

Eventually, a stream of TmfEvents will be required and the active experiment, if any, is supplied by TmfComponent (switching experiment generates a global event that should be caught and addressed by the components)[4].

4 The current version of the framework doesn't handle multiple, simultaneous experiments with multiple instances of the viewers (for side-by-side comparisons). This requires a few modifications, one of them being the introduction of a context object.

A number of conditions can occur with this scheme:

1. There is no data provider for the requested data type. This is an error condition that generates an exception.
2. There is more than one data provider for the requested data type. The user has to choose among the list of possible providers.

An interesting consequence of this scheme is that a data consumer doesn't have to know the class type of the data provider i.e. replacing a transformation by another (implemented by a different class) is practically transparent for the consumer.

5.4.3 Mixed Approach

The static and dynamic schemes can easily be combined: an application could instantiate none, some, or all of its components either up front or on demand.

5.4.4 Component Disposition

When a component usage count drops to zero (no more subscribers), it can dispose of itself automatically or stay alive, waiting for new subscribers. This could be useful when e.g. a transform requires a lot of computations and elects to cache its results.

The disposition of the component after its last subscriber left (keep, delete) is kept in a TmfComponent registry field initialized statically by the component itself. (preference?)

6 Tool Integration

Tool Integration refers to the ability to integrate a tracing tool in the framework.

For the analysis and visualization, a tool integrator would typically have to provide:

1) A stream adapter
2) A stream event parser
3) One or more transforms
4) One or more viewers

- Eclipse extension points
- Wizards
- TBC