The ToolBusNG's Viewer Framework

Arnold Lankamp

11 Feb 2008

Table of Contents

Introduction

The viewer is a tool that gives insight in the execution and state of the ToolBus and its connected tools. It is mainly used for debugging purposes. This document will discuss the viewer framwork of the Next Generation ToolBus and will serve as a guide for developers that wish to write their own viewer implementation.

Contrary to the viewer implementation of the old ToolBus, the new viewer implementations run as a build-in of the ToolBus; this offers the advantage of direct access to all required data, while having only a minor impact on the performance of the ToolBus.

Design

The design of the viewer framework is fairly basic as it only consists of few parts; the debug ToolBus, the IViewer interface, the IPerformanceMonitor interface and the ScriptCodeStore. These will be discussed in more detail below.

Debug ToolBus

The debug ToolBus is a specialized version of the regular ToolBus. It is able to handle breakpoints, stepping and suspension of the ToolBus and it deals with notifications that need to be passed to the attached viewer.

This is a list of all the viewer related methods:

Execution

```
void doRun()
```

Notifies the debug ToolBus that it should execute normally.

```
void doStop()
```

Notifies the debug ToolBus that it should suspend its execution.

```
void doStep()
```

Notifies the debug ToolBus that it should execute one step.

```
void doTerminate()
```

Requests the termination of the ToolBus.

Breakpoint

```
void addProcessInstanceBreakPoint(int processId)
```

Adds a breakpoint for the process instance with the given id. When the debug ToolBus executes a state element in the associated process instance, the attached viewer will be notified.

```
void removeProcessInstanceBreakPoint(int processId)
```

Removes the breakpoint on the process instance with the given id (if present).

```
void addProcessBreakPoint(String processName)
```

Adds a breakpoint for all the process instances whos type is identified by the given name. When the debug ToolBus executes a state element in one of those process instances, the attached viewer will be notified.

```
void removeProcessBreakPoint(String processName)
```

Removes the breakpoints for the process instances whos type is identified by the given name (if present).

```
void addStateElementBreakPoint(StateElement stateElement)
```

Adds a breakpoint on the given state element. When the debug ToolBus executes the given state element, the attached viewer will be notified.

```
void removeStateElementBreakPoint(StateElement stateElement)
```

Removes the breakpoint from the given state element (if present).

```
void addSourceCodeBreakPoint(String filename, int lineNumber)
```

Adds a breakpoint on the given sourcecode coordinates. When the debug toolbus executes a state element which's position information matches the sourcecode coordiates, the attached Viewer will be notified. Note that the debug toolbus assumes that line numbers start counting at zero.

```
void removeSourceCodeBreakPoint(String filename, int lineNumber)
```

Removes the breakpoint from the given source code coordinates (if present).

Performance monitoring

```
ATerm getToolBusPerformanceStats()
```

Gathers performance statistics related to JVM the current ToolBus is running in.

```
void startMonitoringTool(ATerm toolKey)
```

Initiates the monitoring of the tool associated with the given tool key (in case performance monitoring is enabled for this debug ToolBus). Note that it may take some time before performance statistics for the associated tool arrive, since they are only requested after tool interaction.

```
void stopMonitoringTool(ATerm toolKey)
```

Stops monitoring the tool associated with the given tool key.

```
void startMonitorToolType(String toolName)
```

Initiates the monitoring of the given tool type (in case performance monitoring is enabled for this debug ToolBus). Note that it may take some time before performance statistics for the given tool type arrive, since they are only requested after tool interaction.

```
void stopMonitoringToolType(String toolName)
```

Stops monitoring tools of the given type.

Process logic

```
List<ProcessInstance> getProcesses()
```

Retrieves the list of currently executing process instances.

Note that invoking any method other then those discussed above will likely induce undefined behaviour in the ToolBus.

IViewer

The viewer interface lists all types of viewer related events the debug ToolBus can fire and in which a viewer implementation may possibly be interested.

This is what the interface looks like:

```
void updateState(int state)
```

Informs the viewer about what the debug ToolBus is currently doing.

This is a complete list of possible states:

- UNKNOWN_STATE: Bogus initial state, which is also used in case the debug ToolBus gets confused (in which case it'll go to sleep). Note that this currently will never happen.
- STOPPING_STATE: Indicates that the debug ToolBus is in the process of suspending it's execution. This will be fired after a call to the doStop method.
- WAITING_STATE: Indicates the debug ToolBus is unable to execute any atoms and is waiting for either tool input, a delay to expire or a doStep or doRun event to occur.
- READY_STATE: Indicates that tool interaction occurred, meaning there probably is work to be done.
- RUNNING_STATE: Indicates that the debug ToolBus is executing normally. This will be fired after a call to the doRun method; additionally a transition from WAITING to RUNNING is also possible.
- STEPPING_STATE: Indicates that the debug ToolBus is executing one step. It's execution will be suspended once one atom has been executed. This will be fired after a call to the doStep method; additionally a transition from WAITING to STEPPING is also possible.

```
void stepExecuted(ProcessInstance processInstance, StateElement executedStateEl
```

Fired after the successfull completion of a step. The parameters indicate which state element, in which process instance was executed. Optionally, in case a communication atom has been executed, the partners paramater will indicate which other state elements were involved in the execution of this step.

```
void processInstanceStarted(ProcessInstance processInstance)
```

Fired when a new process instance is started.

```
void processInstanceTerminated(ProcessInstance processInstance)
```

Fired when a process instance is terminated.

```
void processBreakPointHit(ProcessInstance processInstance)
```

Informs the viewer that a registered breakpoint on a process or process instance was hit. The debug ToolBus will not suspend it's execution by itself; the action that will be taken is completely up to the viewer implementation. In case the execution needs to be paused this will need to be done explicitly by calling the doStop method.

```
void stateElementBreakPointHit(StateElement stateElement)
```

Informs the viewer that a registered breakpoint on a state element was hit. The debug ToolBus will not suspend it's execution by itself; the action that will be taken is completely up to the viewer implementation. In case the execution needs to be paused this will need to be done explicitly by calling the doStop method.

```
void sourceBreakPointHit(StateElement stateElement)
```

Informs the viewer that a registered breakpoint on a sourcecode coordinate was hit. The debug toolbus will not suspend it's execution by itself; the action that will be taken is completely up to the viewer implementation. In case the execution needs to be paused this will need to be done explicitly by calling the doStop method.

```
void toolbusStarting()
```

Fired right before the debug ToolBus starts executing the process logic.

```
void toolbusTerminating()
```

Fired right before then debug ToolBus shuts down.

IPerformanceMonitor

The performance monitor interface lists all types of tool related events that the performance monitor implementation needs to handle.

This is what the interface looks like:

```
void toolConnected(ToolInstance toolInstance)
```

Fired when a tool connects.

```
void toolConnectionClosed(ToolInstance toolInstance)
```

Fired when a connection with a tool is terminated.

```
void performanceStatsArrived(ToolInstance toolInstance, ATerm aTerm)
```

Fired when the performance statistics that were requested by the debug ToolBus arrived.

The ATerm containing the performance statistic information has the following layout:

```
performance-stats(tool(type(<string>),         language(<string>)),
memory-usage(heap-usage(<int>),             non-heap-usage(<int>)),
threads([<thread_name1>(user-time(<int>),      system-time(<int>)),
<thread_name2>(user-time(<int>), system-time(<int>)), ...]))
```

ScriptCodeStore

The script code store provides access to all, for the ToolBus, reachable script source code. Additionally it provides caching of loaded code.

It has two important functions:

```
String[] getScriptNames()
```

Returns a complete list of absolute paths to all, for the ToolBus, reachable scripts.

```
byte[] getCode(String scriptPath) throws IOException
```

Retrieves the source code of the script indicated by the given path.

Process logic

Every process instances represents a state machine. Such a state machine consists out of state elements that are linked together.

This is the list of methods that may be of interest to viewer builders:

ProcessInstance

```
String getProcessName()
```

Returns the name of the process instance.

```
int getProcessId()
```

Returns the process instance's unique identifier.

```
List<StateElement> getStateElementSet()
```

Returns the collection of all state elements that the process instance contains.

```
State getProcessState()
```

Returns the process instance's current state.

```
List<ATerm> getSubscriptions()
```

Returns a list containing all the types of notes the process instance is subscribed on.

```
List<ATerm> getNoteQueue()
```

Returns the collection of notes that are currently in the process instance's queue.

State

```
List<StateElement> getElementsAsList()
```

Returns the collection of state elements contained in the state.

```
String toString()
```

Returns a serial representation of the state.

StateElement

```
State getFollow()
```

Returns the follow state of the state element. This follow state contains all state elements that are possible candidates for execution after this current state element has been processed.

```
PositionInformation getPosInfo()
```

Returns the position coördinates associated with a certain state element. Position information will be discussed in more detail in the next section.

```
String toString()
```

Returns a serial representation of the state element.

```
List<ATerm> getTests()
```

Returns a collection of all test expression of the tests that are set on the state element. These test expressions restrict the conditions under which the state element is allowed to execute (in other words these represent the surrounding if-statements after compilation).

Atom

```
Environment getEnv()
```

Returns the environment that is associated with the atom.

```
int getDelay()
```

Returns the amount of time (in ms) that needs to pass by before the atom is allowed to be executed.

```
int getTimeout()
```

Returns the time (in ms) within which the atom needs to be executed before being invalidated.

Environment

```
List<Binding> getBindingsAsList()
```

Returns all bindings in the environment as a list. The bindings are key-value pair, that contain a mapping between a variable and its value.

Note that invoking any method other then those discussed above may induce undefined behaviour in the ToolBus; so caution is adviced.

Position information

During the parsing of the ToolBus scripts, position information will be added to the constructed state elements. This position information is used to relate these state elements to source code coordinates.

Positions contain the following data:

- Name of the script.
- Start line.
- Start column.

- End line.
- End column.

Note that both the line numbers and the column numbers start at 0.

One can obtain the collection of all state elements contained in a certain process instance by calling the *toolbus.process.ProcessInstance#getStateElementSet()* method, as discussed in the previous section. Not every state element may have position information on it, since some are 'made up' during compilation; all state elements that can directly be related to an expression in a ToolBus script will however have position information associated with them.

Tools

When implementing a performance viewer one has to deal with tool instances. There are only two methods on the ToolInstance class that should be used by a viewer implementation.

These are:

```
ATerm toolbus.tool.ToolInstance#getToolKey()
```

This method returns a unique key which identifies the tool instance. It consists of the name of the tool and its id (integer). It adheres to the following format: `<toolname>(<id>)`

```
toolbus.tool.ToolInstance.getToolName()
```

A convience method for retrieving the name of the tool that is associated with a certain tool instance.

Note that invoking any method other then those discussed above will likely induce undefined behaviour in the ToolBus.

Viewer template

Below follows a template which can be used to start a new viewer implementation. Developers are encouraged to copy the code below and modify it to their needs.

```
public class Viewer implements IViewer{
  private final DebugToolBus debugToolBus;
  private final ScriptCodeStore scriptCodeStore;

  public Viewer(String[] args){
    super();

    debugToolBus = new DebugToolBus(args, this, new ToolPerformanceViewer()); /
    scriptCodeStore = new ScriptCodeStore(debugToolBus); // Optional; this is o
  }

  public DebugToolBus getDebugToolBus(){
    return debugToolBus;
  }

  ...
  (Implemented IViewer methods).
  ...

  // Optional part.
  private static class ToolPerformanceViewer implements IPerformanceMonitor{
```

```
    public ToolPerformanceViewer(){
      super();
    }

    ...
    (Implemented IPerformanceMonitor methods).
    ...
  }
  // End optional part.

  public static void main(String[] args){
    Viewer viewer = new Viewer(args);
    DebugToolBus debugToolBus = viewer.getDebugToolBus();

    CommandLine.createCommandLine(debugToolBus, System.in, false); // Enable co

    debugToolBus.doStop(); // Set the initial state to stopped (recommended but

    try{
      debugToolBus.parsecup(); // Execute the parser.
      debugToolBus.prepare(); // Initialize the ToolBus.
      debugToolBus.execute(); // Execute the debug ToolBus.
    }catch(Exception ex){
      ex.printStackTrace();
    }
  }
}
```