Execution Manager (KIEM)

Deprecated since 0.12

This article is deprecated. The described features are no longer available in current releases.

Project Overview

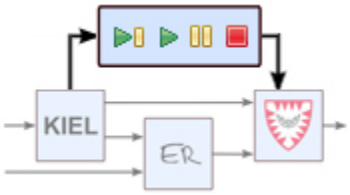
Responsible:

- Christian Motika

Related Theses:

- Christian Motika, *Semantics and Execution of Domain Specific Models – KlePto and an Execution Framework*, December 2009 (pdf)
- Sören Hansen, *Configurations and Automated Execution in the KIELER Execution Manager*, March 2010 (pdf)

KIEM - KIELER Execution Manager



Topics

This sub project implements an interface of the KIELER project for the simulation and execution of graphical domain specific models (e.g., EMF models). It itself does not do any simulation computation but bridges simulation components, visualization components and a user interface within the KIELER Eclipse rich client platform. To get a first impression about this sub project please feel free to watch the following Flash demo video.

JavaDoc

Find the official JavaDoc documantation of the KIELER Execution Manager.

Quick start

Install

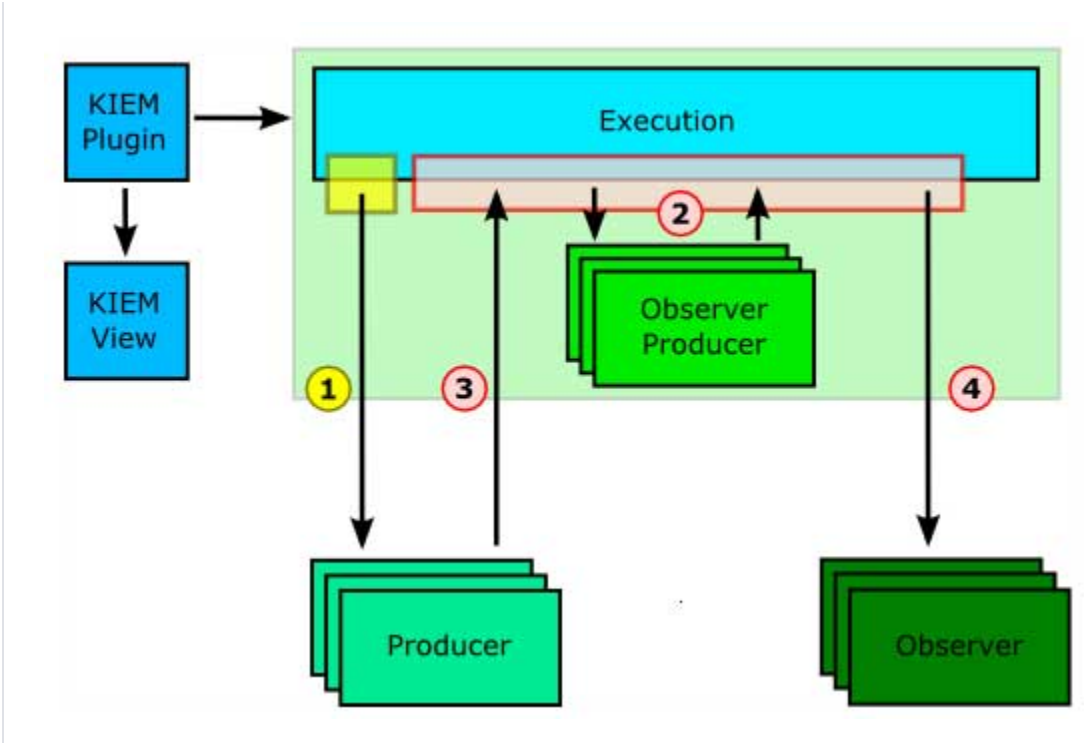
Check out the KIEM Eclipse plug-in project and enable it in your run configuration. Make sure to enable all required projects and to include all DataComponents you wish to use e.g., for visualization or simulation (s.b.).

Launch Configuration

The KIELER Execution Manager offers a launch configuration extension, see KIEM - KIEM Launch Configurations.

How does it work?

Scheduling



All components have in common that they are called by the Execution Manager in a linear order. This can be defined by the user in an execution setting and exactly reflects the order of the DataComponent list in the KIEM View shown in the example figures below. Because the execution is an iterative process --- so far only iteratable simulations are supported --- all components (e. g., a simulation engine or a visualizer) should also preserve this iterative characteristic. During an execution KIEM will stepwise activate all components that take part in the current execution run and ask them to produce new data or to react to older data. As KIEM is meant to be also an interactive debugging facility, the user may choose to synchronize the iteration step times to realtime. However, this might cause difficulties for slow DataComponents as discussed below.

All components are executed concurrently. In particular, components that cannot run concurrently due to implementation and scheduling restrictions share a common thread. Apart from that, components are executed in their own threads. For this reason, DataComponents should communicate (e. g., synchronize) with each other via the data exchange mechanism provided by the Execution Manager only to enforce thread safety. There are also additional scheduling differences between the types of DataComponents listed below. These concern two facts: First, DataComponents that only produce data do not have to wait for any other DataComponent and can start their computation immediately. Second, DataComponents that only observe data, often do not need to be called in a synchronous blocking scheme since no other DataComponent depends on their (nonexistent) output.

Extension Points and Interfaces

If you want to contribute a simulation or execution engine or any visualization facility to the KIELER project, you just need to understand about the two Extension Points that are explained below:

- de.cau.cs.kieler.sim.kiem.json.datacomponent (handles JSONObjects)
- de.cau.cs.kieler.sim.kiem.string.datacomponent (handles JSONStrings)

Both Extension Points are based on the same (abstract) super-class which is called AbstractDataComponent (AbstractDataComponent.java) and itself implements an interface called IDataComponent (IDataComponent.java). A DataComponent may handle JSONObjects (using the following JSON implementation for java: http://www.json.org/java/) directly or it may handle JSONStrings only. This is where the two Extension Points differ and what is necessary to decide prior to the implementation of a concrete JSONObjectDataComponent or JSONStringDataComponent.

If you take a look at the class diagrams of the extension package, you will see that there are 5 methods that every DataComponent needs to supply:

1. initialize()
2. wrapup()
3. step()
4. isProducer()
5. isObserver()

The first method is called during the initialization phase, that is, the execution is about to begin (but not yet has begun). This happens when for example the user clicks on play, step or the pause button in the GUI or such function is triggered by a *Master* DataComponent. This method can be used to do some initialization work that needs to be done prior to the execution, like saving opened files, doing some model transformations or resetting some data.

The second method (wrapup) is called when the execution has stopped and enables a component to provide some cleanup code that needs to be executed.

The method step() is called during the execution. It depends on the interface that is implemented what kind of data is passed as a parameter (JSONString or JSONObject) and also what (same!) kind is expected to be returned by the implementation. The parameter of the step() method contains the requested data. It depends on the following:

method to override	**standard return**	**info**
isObserver()	n/a	If returned false, the DataComponent will always get a null value as a parameter.
provideFilterKeys()	null	A String[] array of keys that the DataComponent wants to listen to should be returned. If null is returned then the DataComponent will always get all unfiltered data.
isDeltaObserver()	false	If true is returned, the DataComponent will get only the new or updated data since the last step it was scheduled (and **not** skipped, see *pure* data observers in scheduling section). If false is returned, the DataComponent will always get the complete (updated) data since the first execution step

The methods isProducer() and isObserver() must be implemented and should return a boolean value indicating whether the DataComponent wants to receive any data (s.a.) or produces data or even both. There is a possibility of so called *Initialization* DataComponents that neither receive nor produce any data but only are used during the initialization and wrap-up phase when their initialize() and wrapup() method is called.

Take in mind that if your DataComponent is not both, an observer **and** a producer of data, it's step() method will not be called in a blocking scheme during the scheduled execution (for details please see above). If this is required, then you need to set the according return values to both being true.

Components and Properties

DataComponents may provide properties that enable the user to configure them prior to an execution run. To do so, a DataComponent needs to @Overrride the method provideProperties() that returns an array of elements of the type KiemProperty. There are several build-in types like for String, Integer, Boolean values or for a list of choices, a file browser or for selecting an open editor. The following code should demonstrate the use of those build-in properties.

```
@Override
public KiemProperty[] provideProperties() {
        KiemProperty[] properties = new KiemProperty[7];
        properties[0] = new KiemProperty(
                        "state name",
                        "state");
        properties[1] = new KiemProperty(
                        "some bool",
                        true);
        properties[2] = new KiemProperty(
                        "an integer",
                        2);
        properties[3] = new KiemProperty(
                        "a file",
                        new KiemPropertyTypeFile(),
                        "c:/nothing.txt");
        String[] items = {"trace 1","trace 2", "trace 3", "trace 4"};
        properties[4] = new KiemProperty(
                        "a choice",
                        new KiemPropertyTypeChoice(items),
                        items[2]);
        properties[5] = new KiemProperty(
                        "workspace file",
                        new KiemPropertyTypeWorkspaceFile(),
                        "/nothing.txt");
        properties[6] = new KiemProperty(
                        "editor",
                        new KiemPropertyTypeEditor(),
                        "");
        return properties;
}
```

Please note that all values are the canonical string representatives and hence all property values need to be serializable. This is also required for own property types that can easily be created when deriving from the abstract class KiemProperty. A custom KiemProperty needs to implement the IKiemProperty interface and hence to provide the following two methods:

1. getValue()
2. setValue()

The Object values of the first two methods depend on the cell editor used by this property type. By default this is the TextCellEditor that handles Strings. You can @Override the provideCellEditor() method and provide another cell editor here. For example the ComboBoxCellEditor operates on integer values. Take in mind that only the String representation, that is accessible thru a call to KiemProperty.getValue()/KiemProperty.setValue() is the one that will be stored and should be unique to be distinguishable.

```
@Override
public CellEditor provideCellEditor(Composite parent) {
        cellEditor = new ComboBoxCellEditor(parent, BOOL_ITEMS, SWT.Deactivate);
}
```

With overriding the provideIcon() method you are able to provide a customized Image for the KiemPropertyType. If you return null (which is the default) the standard image will be used.

Package Organization

The following lists the most important packages and classes of the KIELER Execution Manager.

Overview

This should give an overview about the base packages of the KIEM project:

1. The **kiem package** contains the KiemPlugin activator, the basic interfaces for the extension points (API), the KiemEvents, the KiemExceptions and the externalized strings.
2. The **ui.views package** implements the tree table view and most of the gui part.
3. The **ui package** contains additional gui helpers like the text fields, icons and special SWT widgets.
4. The **properties package** contains some basic KiemPropertyTypes as well as an interface and abstract class to extend those.
5. The **internal package** accommodates some internal interfaces and abstract classes for the Extension Points. The AbstractDataComponent class may be of most interest for deciding which methods to override.
6. The **execution package** implements the scheduling and threaded execution.

Please feel free to browse the source and the JavaDoc documantation for further more detailed information.

Download

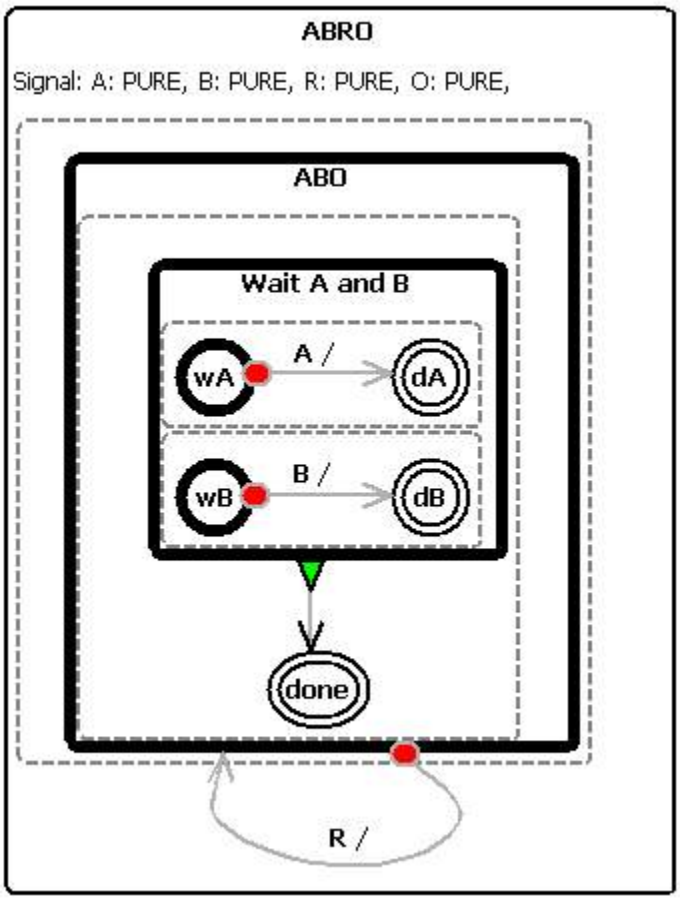
- Source: KIEM
- Source: RawTable
- Source: ABRO in JAVA
- Source: Synchronous Signal Resetter

- Source: SimpleRailCtrl Editor
- Source: SimpleRailCtrl C-Code Generator
- Source: SimpleRailCtrl Ptolemy Simulator
- Source: SimpleRailCtrl View Management

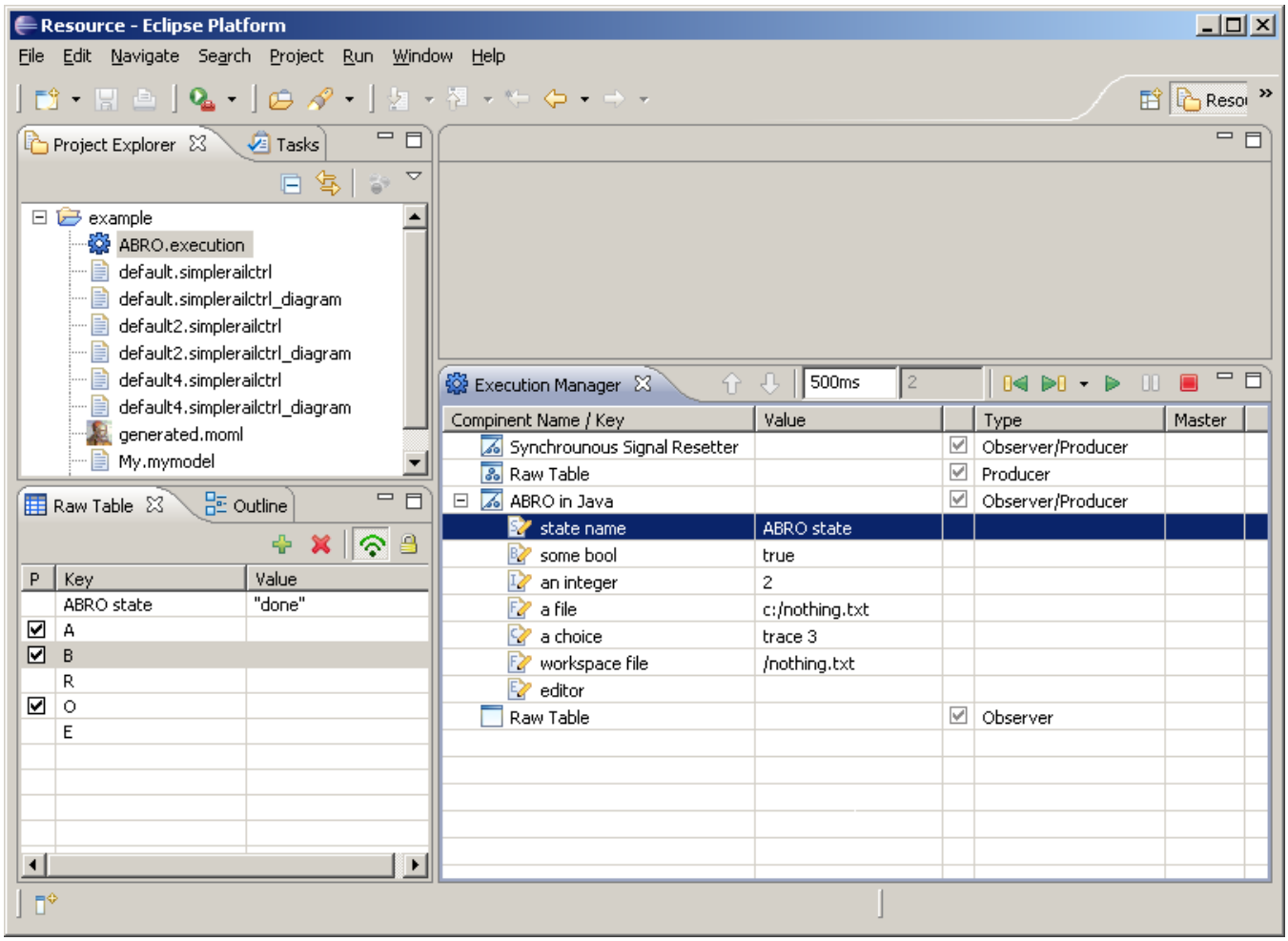
Case Studies

ABRO in Java

This illustrates the famous ABRO example, the "hello world" of the synchronous world. It is simply a Java plug-in implementing an observing and producing DataComponent that reacts to signals A, B, R with producing a present signal O whenever signal A and B just became present (in any order or even at the same time). The SyncChart then goes into the done state and is reset by signal R, i.e. it becomes ready and again waits for signals A and B. The strong abortion of the reset transition indicates that whenever R is present, in the same tick no O will be produced.

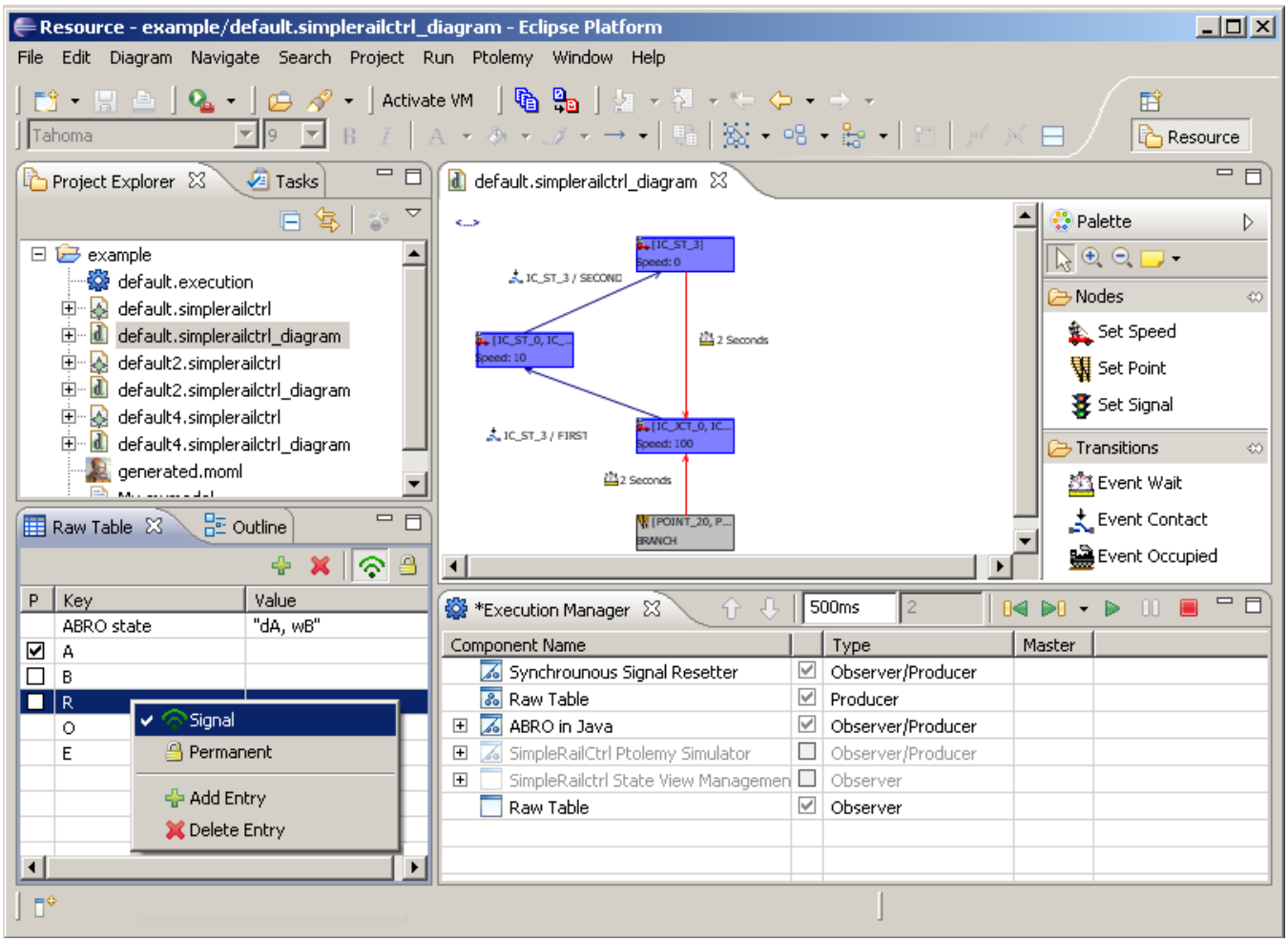


The above example observer and producer DataComponent can be executed by the execution manager. This is illustrated in the following screen snapshot. Signals can be injected (= made present) by marking the check boxes [X] in the left raw table view. Note that the variables first have to be declared as being signals by using the adequate toggle button. By using the step button of the execution manager the execution can proceed to the next step. The *Synchronous Signal Resetter* DataComponent just resets all present signals at the beginning of the next tick to be absent again so that the whole execution follows the synchronous semantics. An alternative to this could be to let all signal emitters reset their own signals with the drawback of introducing relative (macro) ticks.



Simple Rail Control

As another case study there exists a *SimpleRailControlEditor* for the model railway of the Christian-Albrechts University of Kiel. It lets you create controllers for the model railway by modeling them with a generated Eclipse GMF editor. These models can be transformed into executable C-Code by a model2text-Xpand-transformation on the one hand. On the other there exists a complete Xtend-transformation which generates executable and I/O-equivalent Ptolemy models out of them. The *SimpleRailCtrl Ptolemy Simulator* DataComponent is then capable of executing these Ptolemy models using the Triq Ptolemy Eclipse plug-in. Together with the KIELER model visualizer the active states (nodes) of the controller model can then be illustrated during the execution.



There also exists a demo video that shows the just described behavior. Whenever the execution is being initialized the Ptolemy simulator will transform the currently saved EMF model of the opened diagram into a semantically equivalent but executable ptolemy moml-File.