Christian-Albrechts-Universität zu Kiel

Bachelor Project

Configurations and Automated Execution in the KIELER Execution Manager

cand. inform. Sören Hansen

February 12, 2010

Department of Computer Science
Real-Time and Embedded Systems Group

Prof. Dr. Reinhard von Hanxleden

Advised by:
Christian Motika

Eidesstattliche Erklärung

Hiermit erkläre ich an Eides statt, dass ich die vorliegende Arbeit selbstständig verfasst und keine anderen als die angegebenen Hilfsmittel verwendet habe.

Kiel,

_____

Contents

List of Figures

Verzeichnis der Auflistungen

Abbreviations

API Application Programming Interface

GUI Graphical User Interface

IDE Integrated Development Environment

KIEL Kiel Integrated Environment for Layout

KIELER Kiel Integrated Environment for Layout Eclipse Rich Client

KIEM KIELER Execution Manager

KIEMAuto Automated Executions for the KIEM

KIEMConfig Configurations for the KIEM

OS Operating System

RCA Rich-Client Application

UI User Interface

1 Introduction

1.1 KIELER Framework

- layouter, structure based editing
- syncchartsimulater
- execution manager

1.1.1 Model Files

1.2 The KIELER Execution Manager

Execution Manager is used in KIELER as a framework to plug-in DataComponents for various tasks. Examples are:

- Simulation Engines
- Model Visualizers
- Environment Visualizers
- Validators
- User Input Facilities
- Trace Recording Facilities

These DataComponents can be executed using a graphical user interface (GUI). The scheduling order can also be defined by this GUI as well as other settings like a step/tick duration and properties of DataComponents. For information about KIELER Execution Manager (KIEM) see the wiki[1].

[1] `http://rtsys.informatik.uni-kiel.de/trac/kieler/wiki/Projects/KIEM`

1.2.1 DataComponents

DataComponentWrappers

1.2.2 Execution

1.3 Outline of this Document

The first part of this document is about the implemention of the Configuration plugin for the KIEM. The second part discusses the implementation of the Automated Execution plugin for the KIEM. Both parts will have the same structure described below. Each part starts with a detailed outline of the problems that are to be solved by this thesis. The next section introduces technologies used to solve the problem. After that there will be a section about the concepts of how to solve the problem and some design decisions that were made at a very early stage in the development. Sections 5 and 6 are about the actual implementation with section 5 outlining the changes that were made to the KIEM plug-in itself and section 6 describing the newly created Configurations for the KIEM (KIEMConfig) plug-in. The final section of each part will summarize the results of the thesis and outline a few projects that could be based on it.

Part I

Configuration Management

2 Used technologies

- before explaining solutions
  - short explanation about the technology in question
  - small sketch to get an idea of the context
  - full explanation goes beyond the scope of this work

2.1 Eclipse

- KIELER part of Eclipse framework
  - Java IDE but also lots of other languages (C++, Latex, ...)
  - can build other IDEs with it
  - citation: IDE an for anything, and nothing in particular [2]

2.1.1 Plug-ins

- KIEM itself is a plug-in, this work plug-in dependent on KIEM
  - different components of an Integrated Development Environment (IDE)
  - can operate by themselves or depending on other plug-ins
  - in addition to Application Programming Interface (API) of each plug-in: extension point mechanism
  - extension points to interact with other plug-ins
  - extension point mechanism used by runtime environment to check which plug-ins to load
  - plug-ins can be used to plug into the standard Eclipse feature

Extension point mechanism

2.1.2 Preference Pages

- One of the plug-ins used in this work in that fashion is the *org.eclipse.ui.preferencePage* plug-in. It is used to create new preference pages at a specific location inside the normal tree of preference pages accessible through Window>Preferences. The programmer only has to take care of the contents of the actual page and not worry about additional buttons or integrating it into the PreferenceDialog.

Preference Store

- Closely coupled with the preference pages is the Eclipse preference store. It is basically a text file for each plug-in where the plug-in can deposit simple Strings

under a given key to ensure that information is kept between execution runs.

For additional information about eclipse see the official Eclipse website[1] or literature [3]

[1] www.eclipse.org

3 Problem Statement

The objective of this project is to improve the configurability of the Execution Manager as outlined in the diploma-thesis by Christian Motika[1] :

> KIEM currently does not have a preference page to save additional settings like DataComponent timeouts. Also execution schedulings might be similar for a common diagram type.
>
> It may improve the usability further to allow the user to customize execution schedulings for specific diagram types. An interface for these kind of settings could be realized as an Eclipse preference page.

This task will be explained in more detail in this chapter. It will start by introducing solutions to the problem of how save the new configuration properties into the existing execution files. In addition the chapter will explain ways to enable the user to set up a series of default configurations. The last section of this chapter will explore possibilities of how to make it easier to load previously saved schedules.

3.1 Configurations

Currently every property in the Execution Manager has a hard coded default value. There is a text box for setting the aimed step duration for the currently loaded execution file but that value is lost once a new execution file is loaded. To solve this problem an extension to the Execution Manager should attempt to provide the following:

1. Find a way that execution files can store values like the aimed step duration and the timeout. This mechanism should be implemented in a way that ensures that old files can be upgraded and new files are still valid in instances of the Execution Manager that don't use the configuration plug-in.

2. Find a way to load the configurations into the different parts of the Execution Manager as soon as an execution file is loaded from the file system.

3. Ensure that the user can edit all properties and maybe even create his own custom properties. This should be implemented in a way that doesn't clutter up the current user interface too much.

3.1.1 Default Configuration

The different properties stored in each execution file might sometimes not suit the users current needs and he might want to use a default value for some properties without having to manually set them in each new configuration. The solution could be implemented using the preference mechanism provided by Eclipse.

1. There should be a way to set the default properties for all Execution Manager properties and possibly for user defined properties as well.
2. It should be possible for the user to set up which of these properties should override the value stored in the execution file and which should only be used if the execution file doesn't contain one.

3.2 Easier configuration loading

The last objective of this thesis is to make it easier to load execution files. Currently all execution files are stored in the workspace at a place of the users choice. In a very large workspace it can be very hard to find the execution file that you need for your current simulation. The list of recently used documents that Eclipse provides is of little use since all opened documents are placed there not just execution files. This problem leads to the in the following tasks:

1. Finding a way to track recently used execution files and make it easier for the user to load them without the need to locate them inside his workspace.
2. In addition to tracking recently used execution files the user might want to have a way to get a list of execution files that work for the currently active editor. This list should be sorted with the most likely candidates at the top to allow less experienced users to select an execution file that will most likely work.

4 Concepts

The solution to the problems outlined in Chapter 3 can be achieved with the help of the Eclipse plug-in technology described in Chapter 2.

This chapter explains the solutions in the same structure as Chapter 3. That means that it will start by introducing solutions to the problem of how save the new configuration properties into the existing execution files. In addition the chapter will explain ways to enable the user to set up a series of default configurations. The last section of this chapter will explore possibilities of how to make it easier to load previously saved schedules.

4.1 Configurations

The first approach to save additional configuration information in the execution files (see Section 1.2.2) would be to actually modify the format of those files and write the information into them. This would most likely be the easiest approach but would destroy backward compatibility of those files since the basic Execution Manager would not know how to deal with the modified files.

The approach taken in this thesis is based on the list of DataComponents (see Section 1.2.1). Each execution file carries a list of its own DataComponents and their properties to ensure that the component are properly initialized the next time the execution is loaded. That list can be loaded even if there are DataComponents contained in it that are not present in the current runtime configuration. The Execution Manager will show a warning indicating that it doesn't know the given component but proceed to load the rest of the execution.

These DataComponents basically just carry a list of KiemProperties which are basically (key, value) pairs. Hence, they are suited well for storing simple information like the value of a text field.

To solve the problem a new type of DataComponent will have to be constructed and registered through the extension point in the Execution Manager . This ensures that the component can be added to any execution file. The Execution Manager ensures that all properties contained in our component will be saved with the execution file and loaded the next time the file is opened. After that the configuration plug-in has find the component inside the DataComponent list, get the properties saved inside it and set them inside the Execution Manager.

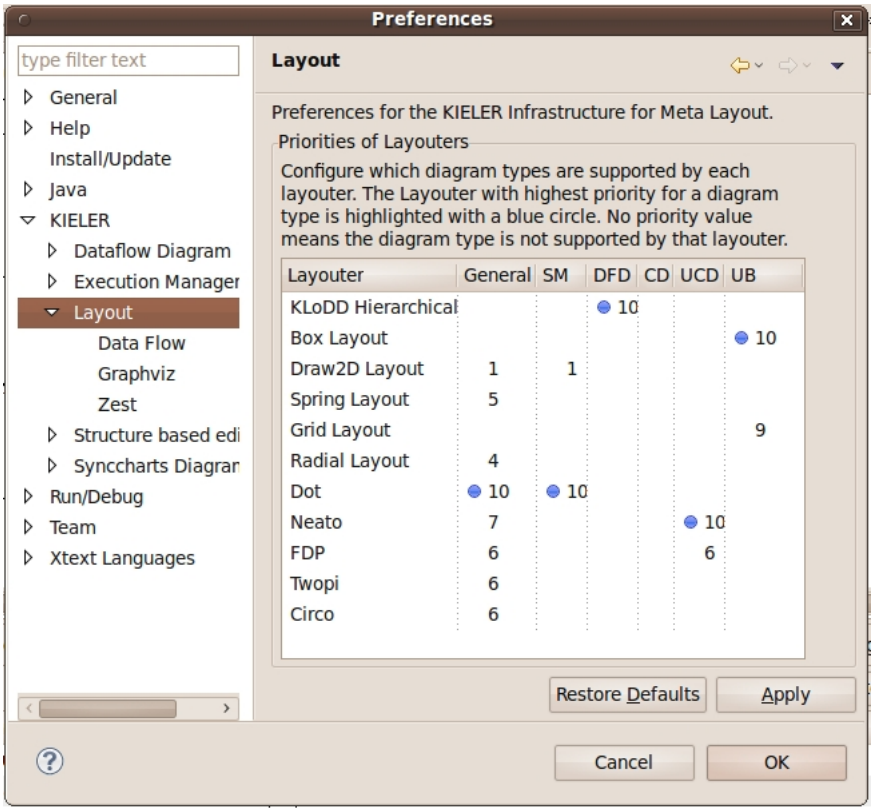

Figure 4.1: Layout Preference Page by Miro Spönemann

4.1.1 Default Configuration

In order to provide a place to manage the default configurations we will be using the Eclipse preference pages.

The root page for the Execution Manager should contain the basic settings for the KIEM itself like the aimed step duration, timeout and the view elements of the Configuration plug-in.

Then we will construct another page for managing the different schedules and their editors. For that we will adapt the LayoutPreferencePage by Miro Spönemann as seen in Figure 4.1. The original preference page displays a table where different layouters can be assigned priorities for different diagram types. This is similar to the problem that needs to be solved in this thesis. The diagram types can be mapped to our editors and the layouters will be replaced by the list of saved schedules. That way the modified preference page can be used to assign priorities to schedules for different editors. The priorities can then be used to sort the schedules matching the currently opened editor.

The last page will be used to allow the user to set up his own properties and give them default values.

The values entered in those pages will be stored inside the Eclipse preference store (see Section 2.1.2).

4.2 Easier Configuration loading

For easier configuration loading we will add two ComboBoxes to the existing tool bar inside the KIEM view.

One of them will display the list of recently used schedules the other one the list of schedules that work for the currently active editor.

As soon as the user opens a new execution file through the normal workspace view we will be notified of that event by the Execution Manager. We will then create a new schedule and store the path to the execution file in it along with the editor that was opened at the time that the schedule was created. The new schedule will also be added to the list of recently used schedules that we maintain through the use of the Eclipse preference store.

When the user selects one of the previously saved schedules in one of the ComboBoxes we will retrieve the saved path and offer it to the KiemPlugin to load it in the hopes that the execution file is still in the same location and wasn't deleted, renamed or moved by the user.

5 Code Changes in the Execution Manager

Although the project attempts to realize most of the objectives without changing the Execution Manager itself minimal adaptations were necessary. This mostly involves adding new methods to the API in order to gain access to previously hidden properties.

Also some changes had to be performed where properties were loaded from hard-coded default values. These were refined and will now only be used if the KIEMConfig plug-in is not registered to supply previously saved properties.

However all changes that were made to the KIEM plug-in were merely additions and won't break any plug-ins relying on the old implementation.

5.1 Schema Files and Interfaces

In order to provide additional functionality for other plug-ins we choose the extension point mechanism described in Section 2.1.1. This is done is order to retain the old functionality of the plug-in while on the other hand giving options to ask extending plug-ins for their contributions.

The extension points are described in more details in the next sections. They consist of a schema file for defining the extension point and an interface that contains the methods that extending components have to supply.

5.1.1 Toolbar Contribution Provider

The purpose of the tool bar contribution provider is to allow other plug-ins to put items onto the tool bar of the Execution Manager. There are two reasons for using the extension point mechanism rather than making the tool bar available and have other plug-ins put their contributions directly on it:

1. At the moment the tool bar and all contributions are created dynamically. Switching the entire native tool bar of the Execution Manager to adding the actions to the tool bar through a predefined Eclipse extension point would require major code changes and have major drawbacks.

2. A programmatic approach gives control over the contributions to the Execution Manager. This means that the order of the native Execution Manager buttons is always the same and in the same place. It also means that the Execution Manager can choose to ignore contributions if the tool bar gets too crowded.

Figure 5.1: The interface for Tool bar Contribution Providers

The schema file for components that want to add contributions to the tool bar is quite simple. It only requires them to implement the interface shown in figure 5.1. The implementing class provides an array with all ControlContributions they want to add to the tool bar. A ControlContribution for a tool bar can be almost any widget like Labels, Buttons, ComboBoxes, ...

When the Execution Manager starts to build the views tool bar it will perform the following steps:

1. The contributors will be asked for the list of controls that they want to contribute.

2. That list will be added to the Execution Manager's tool bar.

3. After that the Execution Manager will add its own controls to the tool bar.

This order causes the tool bar to have the native elements always in the same order. The contributed elements will be added from left to right in the order that they occur in the array. However there is no guarantee on the order in which the extending plug-ins are asked. Figure 5.2 shows the Execution Manager tool bar with the two combo boxes belonging to KIEMConfig contributed through the extension point. Figure 5.3 shows the same tool bar without the contributions.

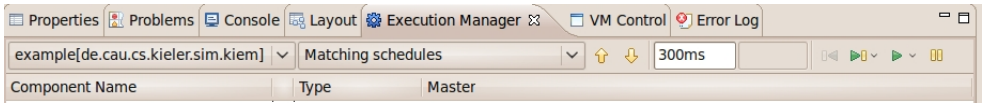

Figure 5.2: The Execution Managers Tool bar with two contributed ComboBoxes

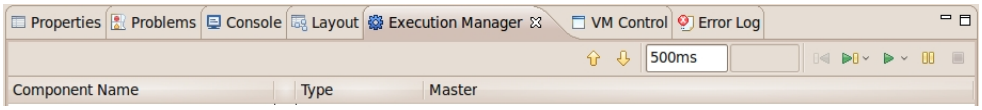

Figure 5.3: The Execution Managers Tool bar without contributions

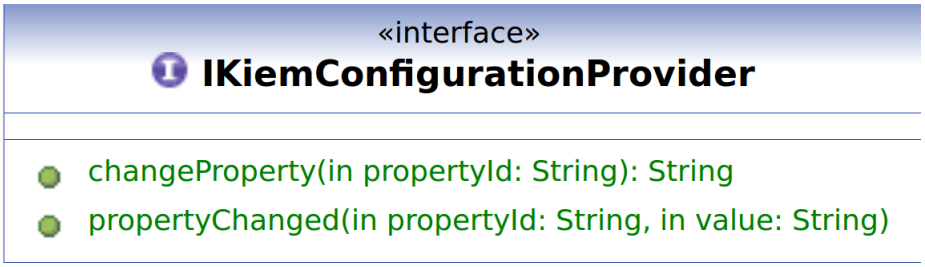

Figure 5.4: The Interface of the Configuration Provider

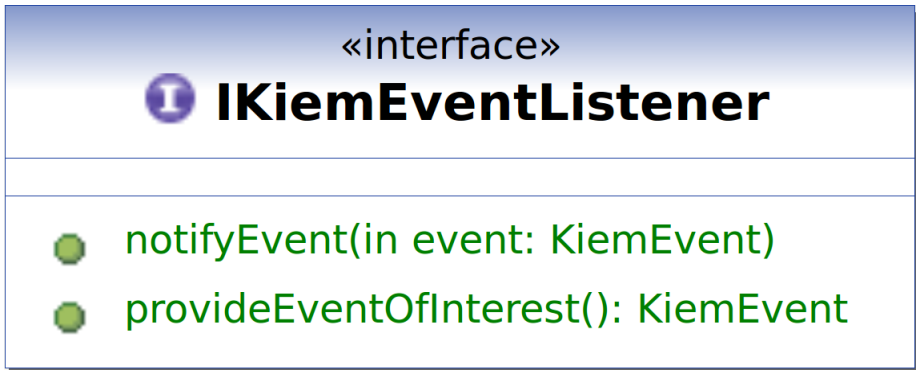

Figure 5.5: The Interface of the Event Listener

5.1.2 Configuration provider

The purpose of the configuration provider is to allow internal attributes of the Execution Manager to be stored in another plug in.

This is achieved by another extension point to allow any plug-in to listen to changes in the Execution Manager's attributes. It also means that there may be multiple plug-ins that provide values for properties and not all plug-ins may have the value for a property needed by the Execution Manager. Through the plug-in mechanism the KIEM can ask all providers for values and choose the one he would like to use.

The two methods from the interface shown in Figure 5.4 work in the following way:

5.1.3 Event Listener

The main purpose of the Event Manager is to inform DataComponents of events happening in the Execution Manager during execution. This behavior has been modified to include events that occur while the execution is not running. This modification lead to the creation of another extension point in order to allow other plug-ins to be notified on any of these events as well. The classes implementing the interface (see Figure 5.5) required by this extension point will be notified on any event that happens inside the Execution Manager.

int provideEventOfInterest() : This method is directly derived from the method with the same name in the AbstractDataComponent class of the KIEM plug-in. It is called by the EventManager to determine which events the implementing class is interested in. This is done to improve efficiency and not flooding components with events they are not interested in.

Based on the response, the EventManager puts the component into lists along with the DataComponentWrappers already inside.

void notifyEvent(KiemEvent event) : This method is called by the EventManager when something happens inside the Execution Manager that the implementing classes might be interested in.

5.2 KIEMPlugin.java

The main Activator class contains almost the entire API of the KIEM. Therefore any additions to that has to performed in this class which means that most of the adjustments were made here.

5.2.1 Listener

The following methods were added to communicate with the plug-ins registered through the ConfigurationProvider extension point (see Section 5.1.2).

notifyConfigurationProviders(String propertyId, String value) : This method can be called by any class inside the Execution Manager itself. It should be called when the user changes a property through any of the elements on the Graphical User Interface (GUI). The method will then inform all listeners that the property identified by the given identifier was changed to the new value.

String getPropertyValueFromProviders(String propertyId) : This method allows the Execution Manager to retrieve a previously saved value. The KIEM will ask all plug-ins registered on the extension point if they can provide a value for the given identifier. Plug-ins that can't provide the value will indicate this by throwing an Exception. The KIEM will then take the first value he receives without getting an Exception and assign it to the internal property.

Integer getIntegerValueFromProviders(final String propertyId) : This method is a convenience method for the one described above. It will try to parse an integer from the retrieved String and return it or return null if no integer could be parsed.

5.2.2 Getters and Setters

An example for the use of the methods described in the last section can be found in the Getters and Setters for the different properties in the Execution Manager (for an

Auflistung 5.1: Example of modified Getter and Setter

```
1  public int getAimedStepDuration() {
2     int result = this.aimedStepDuration;
3     Integer value = getIntegerValueFromProviders
4        (AIMED_STEP_DURATION_ID);
5     if (value != null) {
6        result = value;
7     }
8     return result;
9  }
10
11 public void setAimedStepDuration(final int stepParam) {
12    this.aimedStepDuration = stepParam;
13    // if executing, also update current delay
14    if (execution != null) {
15       this.execution.setAimedStepDuration(stepParam);
16    }
17    this.getKIEMViewInstance().updateViewAsync();
18    notifyConfigurationProviders
19       (AIMED_STEP_DURATION_ID, stepParam + "");
20 }
```

example see Figure 5.1). These were changed in order to use the new methods but are still able to fall back on hard-coded default values if no configuration plug-in is registered.

5.2.3 Open File

The method that takes care of loading an execution file was split. This was done to allow other plug-ins to pass an IPath object directly to the method and perform a load of that file without having to go through the User Interface (UI). This method was also shifted around a little in order to detect missing execution files before the load enters the UIThread. This was necessary to make is possible for the callers of the method to catch the resulting exception. The method also had to be modified in order to be able to take files that are not inside the workspace but were added through an extension point. The changed part of the openFile method is shown in Listing 5.2. The last change to that method concerns the event listener. When the user opens a file through the Eclipse workspace without using the KIEMConfig plug-in the plug-in still has to be informed. This happens through the use of the Event Manager that notifies all listeners upon the successful completion of the loading operation.

5.3 KIEMView

The changes described in Section 5.2 were mostly concerned with the configuration management and loading of new execution files. This section will mostly deal with the changes that were necessary to enable the adding of new items to the tool bar.

Auflistung 5.2: The head of the modified openFile() method

```
1  public void openFile(final IPath executionFile,
2     final boolean readOnly) throws IOException {
3     String fileString = executionFile.toOSString();
4     final InputStream inputStream;
5
6     if (fileString.startsWith("bundleentry:/")) {
7        String urlPath = fileString.replaceFirst
8           ("bundleentry:/", "bundleentry://");
9        URL pathUrl = new URL(urlPath);
10       inputStream = pathUrl.openStream();
11    } else {
12       URI fileURI =
13          URI.createPlatformResourceURI(fileString, true);
14       // resolve relative workspace paths
15       URIConverter uriConverter =
16          new ExtensibleURIConverterImpl();
17       inputStream = uriConverter.createInputStream(fileURI);
18    }
19
20    Display.getDefault().syncExec(new Runnable() {
21    ...
```

- changes to tool bar creation
- provide means to set view as dirty when changes happen
- refresh method to reload values, not needed before because no change except by user

6 Kiem Configuration Plug-in

This chapter describes the contents and functionality of the newly created plug-in to solve the problems described in Chapter 3. A new plug-in was created in order to improve modularity within the Kiel Integrated Environment for Layout Eclipse Rich Client (KIELER) framework. Putting the code into the KIEM plug-in itself would have meant that there would have been no way to separate the two projects.

The sections in this chapter describe the different parts of the KIEMConfig plug-in. The whole plug-in is structured according to the Model-View-Controller pattern. The first section will describe the data storing classes which constitute the model. The second section will describe the different manager classes which are essentially the controller of the entire plug-in. This section will also look at the API that the KIEMConfig plug-in provides to other plug-ins. The last section will describe the classes that render the preference pages and other view elements.

6.1 Data Classes and Utilities - the Model

This section will describe the different classes that are responsible for storing all data that the plug-in needs at runtime.

6.1.1 ConfigDataComponent

This extension to the AbstractDataComponent of the KIEM is responsible for solving the problem described in Section 3.1 and implements the behavior described in Section 4.1. The component is a DataComponent like all others used in the KIEM. It is registered through the extension point that allows new DataComponents to appear in the list of available components. However unlike the usual DataComponent that is responsible for simulating a model during an execution run its main function is to store the configuration of the KIEM.

Like all other DataComponents the ConfigDataComponent contains an array of KIEMProperties. These properties contain a String key which should be non-null and unique and a value which can be of various types. However for the purpose of storing configuration elements only the String value will be used.

The new DataComponent also provides additional methods in order to make accessing and manipulatin the array more convenient:

KiemProperty findProperty(String key) : This method iterates through the array and attempts to find the KiemProperty that has exactly the provided key. Since the keys are assumed to be unique the first match is returned by

this method. If there is no property with the given key the method will throw an Exception.

void removeProperty(String key) : This method attempts to remove the property identified by the given key from the array. It does this by converting the array to a list, locating and removing the specified property and then converting the list back to an array. This procedure may not be as efficient as manually constructing the new array but it still performes the operation in linear time. Furthermore it makes the method easier to understand than the alternative.

KiemProperty updateProperty(String key, String value) : This method updates the property identified by the key with a new value. It first checks if the property already exists and if it does its value is updated. If a property with the specified key doesn't exist a new one is created and the provided value stored inside.

In addition to those methods the ConfigDataComponent also keeps a reference to its DataComponentWrapper (see Section 1.2.1). This is necessary in order to retrieve the properties from the wrapper right after the execution file was loaded and to write them back into the wrapper before the file is saved.

TODO:

- can be added and removed by the user to update old files or downgrade new ones
- used as current/default configuration

6.1.2 EditorDefinition

The EditorDefinition class is responsible for storing information about the editors that are known to the KIEMConfig. Each instance of this class stores the information about a single editor. This is necessary in order to successfully operate a list of execution files that work for the currently active editor.

String editorId : The identifier for the given editor. This attribute is a unique non-null String by which any editor can be identified. For example the standard Java editor has the id *org.eclipse.jdt.ui.CompilationUnitEditor*.

String name : The name of the editor. This is the human readable name given to the editor by the plug-in that defines the editor. Storing this attribute may seem redundant since the names of the editors can be retrieved through an Eclipse mechanism if the editor id is known. However there is no guarantee that a previously saved editor id exists in the currently active application in which case the name of the editor can't be retrieved.

boolean isLocked : This attribute is responsible for showing that the editor can not be removed. The reason that an editor might become read only will be explained in Section 6.2.3.

6.1.3 ScheduleData

The ScheduleData class is responsible for tracking the different execution files that are known to the KIEMConfig. A ScheduleData object is the representation of a single execution file. These objects are used to main he lists of recently used schedules and of those that match the currently opened editor. It contains the following attributes:

- The most important attribute is the path at which the execution file that this instance should represent is located. The path is used to trigger the loading of the file inside the Execution Manager. It is also used to determine whether a newly loaded execution file is already known. The path also doubles as the unique identifier for the schedule since there can't be two files at the same physical location.

- The ScheduleData object also stores a list of priorities for all known editors. This is necessary in order to determine whether or not a given schedule can be used with the currently opened editor and which position it should have in an ordered list. To make accessing and manipulating this list easier it simply uses an instance of the ConfigDataComponent. The component already has methods for accessing the array inside and can be easily stored and loaded.

- Like the EditorDescription a ScheduleData also contains a boolean **isLocked**. ScheduleData object with that attribute set to *true* can't be modified or removed (see Section 6.2.3).

MAYBE TODO:

- getting/setting supported priority, adding/removing editors

6.1.4 Tools

The Tools class holds a host of useful methods and attributes that are used in various parts of the plug-in.

Attributes

First of all it contains messages and tooltips that are used in more than one class. This ensures that the appearance of the different messages is unified across the entire plug-in. It also makes it easy to change these messages or combine different partial messages to new ones.

The class also holds the different identifiers for the properties that are used in the plug-in. This is done to avoid bugs due to mistyping an identifier which is likely to happen if it is stored in two different places.

Methods for Parsing and Serialization

All of the manager classes in the KIEMConfig need to save their properties into the Eclipse preference store. In order to have the information stored in a structured way an XML like format was chosen. As this requires the keys and values to be formatted in a certain way the Tools class provides methods to format the Strings in the required way.

String putValue(String key, String value : Converts the (key, value) pair into a formatted String for saving into the Eclipse preference store. The resulting String has the following format: *<[key]>[value]</[key]>*.

String putProperty(KiemProperty property : Convenience method for transforming a KiemProperty object into a formatted String. This method exists because most of the items serialized in this way are of that type. The resulting String has the following format: *<KIEM_PROPERTY> <Key> [property.key] </Key> <Value> [property.value] </Value> </KIEM_PROPERTY>*.

The methods described above provide all the necessary facilities for the KIEMConfig to save its preferences into the Eclipse preference store. In order to retrieve these properties the Tools class provides another set of methods. These methods take an input String and try to parse the saved properties.

String getValue(String key, String input) : This method retrieves the value enclosed by tags with the given key. The retrieved value can either be an atomic String that can directly be assigned to a property or another series of values enclosed in their tags. The method will always look for the outermost tags inside the input String. The method returns null if there are no tags with the provided key inside the input String.

KiemProperty getKiemProperty(String input) : This convenience method tries to retrieve the (key, value) pair that constitutes a KiemProperty object from an input String.

String[] getValueList(String key, String input) : Since there sometimes is the need to store an entire list of entities the Tools class provides a method to convert an entire list back to the individual Strings. The method iterates over the input String and extracts all elements that are enclosed in tags with the specified key.

Methods for Dialogs

The Tools class also contains methods for easily displaying error and warning dialogs. These methods take the information, add the own plug-in id and forward the information to the error handling facilities inside the Execution Manager itself.

6.1.5 MostRecentCollection

The MostRecentCollection is a new collection type that is can be used for simulating the behavior found in 'Open recent' menu item of almost any text editing application. To avoid the list growing too long it can be given a maximum capacity. After that capacity is reached the oldest entry will be deleted when a new one enters the list. The default implementation of the collection uses an ArrayList to store the data but any other list works as well. Most operations are directly delegating to the operations of the underlying List. The only exception is the add(item : T) method that works in a different way:

1. It checks if the item is already in the list and removes it. This is done to ensure that already added items don't appear twice in the list.
2. It adds the item at the highest index to the end of the list and increments the index of all other items.
3. The element at the head of the list is overridden by the new item.
4. Optionally the last item is removed if the list has grown beyond the capacity.

The collection also provides an additional method that is used to replace an item in the list by another one. This is used when files are renamed and the name of the ScheduleData inside the list has to be updated.

This collection is used to track the most recently used schedules and display them in the corresponding ComboBox.

6.2 Manager Class - the Controller

The manager classes are responsible for the control flow inside the plug-in. They gather information from the view, the Eclipse preference store and the Execution Manager and create and update a model using the classes described in Section 6.1. There are multiple managers each with a different task:

- The **Configuration Manager** is responsible for maintaining the configuration saved in each execution file and the default configuration saved in the preferences store.
- The **Schedule Manager** is responsible for keeping track of the different execution files and updating the information inside the ScheduleData objects.
- The task of the **Editor Manager** is to

6.2.1 Abstract Manager

All of the managers share some common features that each of them must provide. Some of those features are handled almost the same or exactly the same in each

manager. This lead to the creation of an abstract super class for all managers (see Figure that takes care of the basic tasks.

The first task is to allow other classes to register as a listener to the manager. Some of the classes in the KIEMConfig have to perform updates when a value inside the model changes. It is the managers responsibility to inform the listeners when such a change was completed successfully.

The second task is to provide the subclasses with facilities to easily access the Eclipse Preference Store. Whenever a value is requested by any part of the controller or another plug-in and a manager didn't access the preference store yet it has to gain access to the store and retrieve the information belonging to it. Furthermore when the user explicitly wants to save the preferences or the workbench is shutting down the data contained in the model has to be saved into the Eclipse Preference Store. For an example of a saved configuration see Appendix 13.2).

6.2.2 Configuration Manager

The Configuration Manager basically handles all the problems described in Section 3.1.

TODO:

- manages current/default configuration, supply filtered lists for preference components
- find current configuration and wrapper in data component wrapper list
- get values for properties (where to look, ignored keys, save to current when found)
  - check if there is a current configuration, check if the key is not on the ignored key list, try to find in current configuration
  - on failure: try to find in default configuration, update current configuration, return found value
  - on failure: if default value supplied, update default configuration, update current configuration, return default value
  - on failure: throw exception
- add/remove properties, update values
  - try to update saved values
  - if it doesn't exist create it
- restore default values

6.2.3 Schedule Manager

The Schedule Manager is the second of the two large managers. It is responsible for managing the ScheduleData object, the execution files and provide the methods for

solving the problem described in Section 3.2. These responsibilities can be broken down into different parts:

1. Gather the different types of lists of schedules.
2. Manage the different ScheduleData objects and provide methods to add, remove and change them.
3. Provide a means to trigger the loading of an execution file in the Execution Manager.
4. Track the locations of the execution files if the user modifies them.
5. Loading the default schedules.

Provide the Schedule Lists

The Schedule Manager stores all ScheduleData objects in one list that is saved and loaded through the abstract super class. However different components of the KIEM-Config or other plug-ins need different views on that list. Some components may not want to display all schedules or have the list sorted in a certain way. To provide these different views the Schedule Manager contains several methods:

List<ScheduleData> getAllSchedules() : Returns the list of all schedules. This method triggers a load through the super class if no load has been performed yet. It also triggers a load of the default schedules described at the end of Section 6.2.3.

List<ScheduleData> getMatchingSchedules(String editorID, String editorName) : This method is responsible for constructing the list that will be displayed in the ComboBox that shows the list of schedules matching the currently active editor. First it tries to find an EditorDefinition with the given editor id. If that fails a new EditorDefinition is created and added it to the list of known editors. After that the method searches through the list of all schedules and extracts those that have a positive priority for the given editor. The list is then sorted and returned with the editor with the highest priority appearing at the lowest index.

List<ScheduleData> getRecentSchedules() : This method constructs the list of recently used schedules. This list is used in another ComboBox to allow the user to easily load his last used schedules. In order to realize this feature the Schedule Manager keeps a list of all execution file locations that were recently accessed using the list described in Section 6.1.5. When the method is called it iterates over the list of locations and tries to find a schedule for each location. Schedules that match a location in that list are added to the resulting list. Entries in the list of locations where no schedule can be found will be removed from the list as they are no longer valid.

List<ScheduleData> getImportedSchedules() : Returns the list of default schedules. This feature will be described in detail at the end of Section 6.2.3.

Schedule Management

Open a Schedule

Tracking Execution Files

Default Schedule

An additional requirement that came up through the development of this project was that it would be desirable to provide the user with an initial set of execution files that are not located inside the users workspace.

For that purpose the new extension point seen in Figure 6.1 was created.

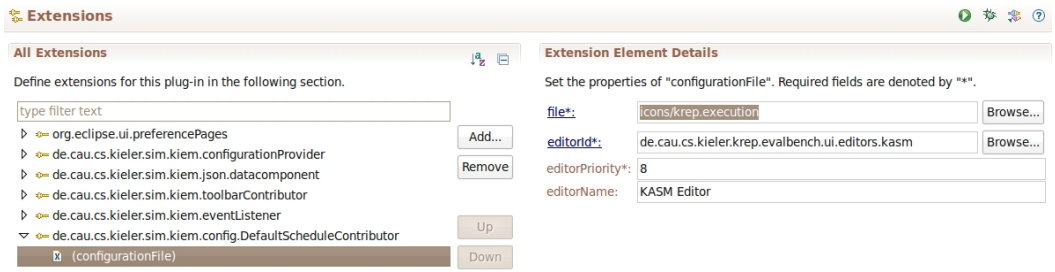

Figure 6.1: The Default Schedule extension point with example

TODO:

- all schedule data
- track recently used schedule Ids
- ask KIEMPlugin to load a saved schedule, deal with error
- add/remove schedules, update schedule priorities
- handle user load/save events

6.2.4 Editor Manager

As described in Section 6.1.2 the editor names and editor ids have to be saved since they might not be available in the current runtime environment. The Editor Manager is responsible for managing the list of all known editors. It also contains facilities around the default editor.

EditorDefinition addEditor(EditorDefinition editor : Adds a new editor to list of editors and return the added editor. If an editor with that editor id already exists in the list it is not added to prevent duplicates. In this case the

already existing editor is returned instead. Since some methods may want to work on the editor that was just added they can work on the returned reference instead of having to perform a find to get the correct object.

EditorDefinition findEditorById(String id) : This method searches through the list of available editors to retrieve the editor with the given id. Since editors ids are assumed to be unique the first match is returned. This method is for example used to build the list of schedules that work with the currently active editor.

void removeEditor(EditorDefinition editor) : At some point the user may decide that one of the editors is no longer used. In this case the editor is simply removed from the list of available editors. However if the removed editor is the default editor (see Section 6.2.4) the method also has to choose a new default editor. The editor of choice for simplicity is the first editor in the list. If the last editor is removed a hard-coded default editor is restored.

When the user is finished changing the editors the manager can use the facilities in the super class to write the list of known editors to the Eclipse preference store.

Default Editor

The KIEMConfig contains a facility for showing the list of schedules that match the currently active editor. However there may be situations no editor is opened. Since the user should be able to keep working with his schedules in this situation there is the need to define a default editor. This editor will be assumed open when in fact no editor is active.

Although the actual storing of the default editor happens inside the Configuration Manager the Editor Manager still supplies facilities to get and set the default editor. This arrangement is more intuitive than having the methods inside the Configuration Manager.

6.2.5 Contribution Manager

The Contribution Manager is responsible for maintaining the view elements (see Section 6.3.3) that are not part of any of the preference pages. To accomplish this the manager has two tasks:

1. The manager must create the view elements and store them. As described in Section 5.1.1 the Execution Manager will ask the KIEMConfig for the list of items it wants to contribute to the tool bar. The manager then has to create the list with the saved view elements and forward it through the extension point.

2. As the user might want to hide the new elements the Contribution Manager also has to keep track of the visibility of the elements. Showing and hiding the components is realized in the following way:

- When the Execution Manager requests the list of control contributions the Contribution Manager checks whether or not the given view element should be visible. If it should not be shown on the tool bar it is not added to the list and thus never reaches the tool bar.
- When the user changes the visibility of a given view element the manager first updates its own representation of that information and triggers a save into the Eclipse preference store through the use of the facilities in the super class. After that the manager triggers a refresh in the Execution Manager view which causes the method in the extension point to be called which then receives the changed list.

MAYBE TODO: Advanced user mode

6.2.6 Property Usage Manager

As described in Section 3.1.1 the user might not want to use the properties saved in the currently loaded execution file but rather the default values entered through the preference page. The Property Usage Manager is responsible enabling the Configuration Manager to realize this feature.

To accomplish this task the manager contains a list of property keys for those values that should be taken from the default configuration. The list can be changed by any other class when the user changes the preferences on which properties should be in it. The list is also stored and loaded through the use of the facilities in the Abstract Manager.

6.2.7 Implementing Classes

- tool bar provider: link to contribution manager, forward array of contributions
- configuration provider: link to configuration manager, forward requests
- event listener: handles events received by the configData component listen to load/save events, can be disabled when KIEMConf is about to trigger load/save

6.3 Preference Pages - the View

- place to easily configure settings, all KIELER preferences in one place
- integrated into eclipse look and feel

6.3.1 Configuration page

- changing default configuration for internal properties
- check boxes for changing visibility of the combos

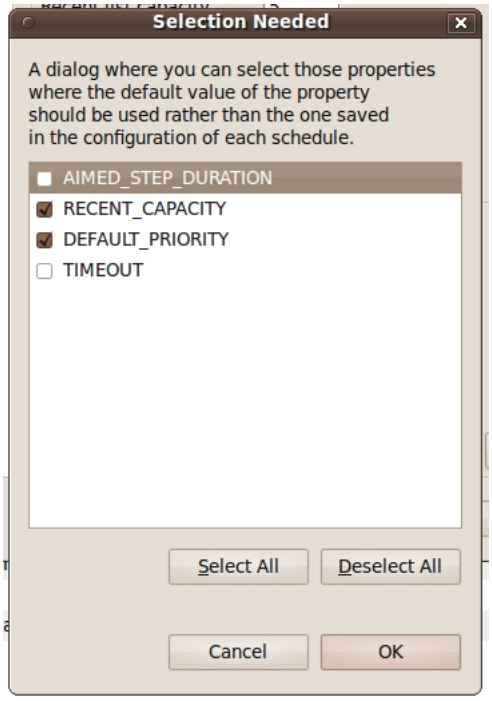

Figure 6.2: The Property Usage Dialog

User defined properties page

- adding/removing properties
- modified from msp, table view with providers

Property usage dialog

This dialog shown in figure 6.2 is used for selecting which properties should always be taken from the default configuration rather than the configuration component contained in every .execution file. The dialog used for this is a ListSelectionDialog which just receives the list of all keys as input and the list of PropertyKeys from the PropertyUsageManager as default selection. After the user is finished with selecting attributes and hit the 'Ok' Button the dialog passes the new list of selected items back to the PropertyUsageManager.

6.3.2 Scheduling page

This preference page is used to manage the schedules and the editors that they belong to. This page is basically a modified version of the LayoutPrioritiesPage by msp.

- table with schedules / editors and their priorities

- modified from msp , editors = diagram types, schedules = layouters
- modify priorities
- add/remove editors, remove schedules, selecting default editor

Adding and removing editors, Selecting a default editor

On the scheduling preference page there are routines for adding and removing editors as well as selecting a default editor. All of these actions use the same basic method for displaying an ElementListSelectionDialog 6.3 that takes a list of editor ids and returns the one selected by the user.

- The editor adding dialog gets a list of all editors currently registered on the active workbench. The user can select a single editor which is then added to the table.
- The editor removal dialog gets a list of all editors currently available for assignment of support properties. The editor selected by the user is removed from the table. It is also removed from all schedules. This is done to prevent the schedule objects from growing to monstrous size over time when editors are getting added and removed.
- The default editor selection dialog gets the same list as the removal dialog. The selected editor is then set as default editor. The default editor is used when there is no currently active editor on the workbench. It is used:
  1. to determine which editors to show in the Matching combo box.
  2. when a new schedule is created as an editor id.

6.3.3 Configuration Selector

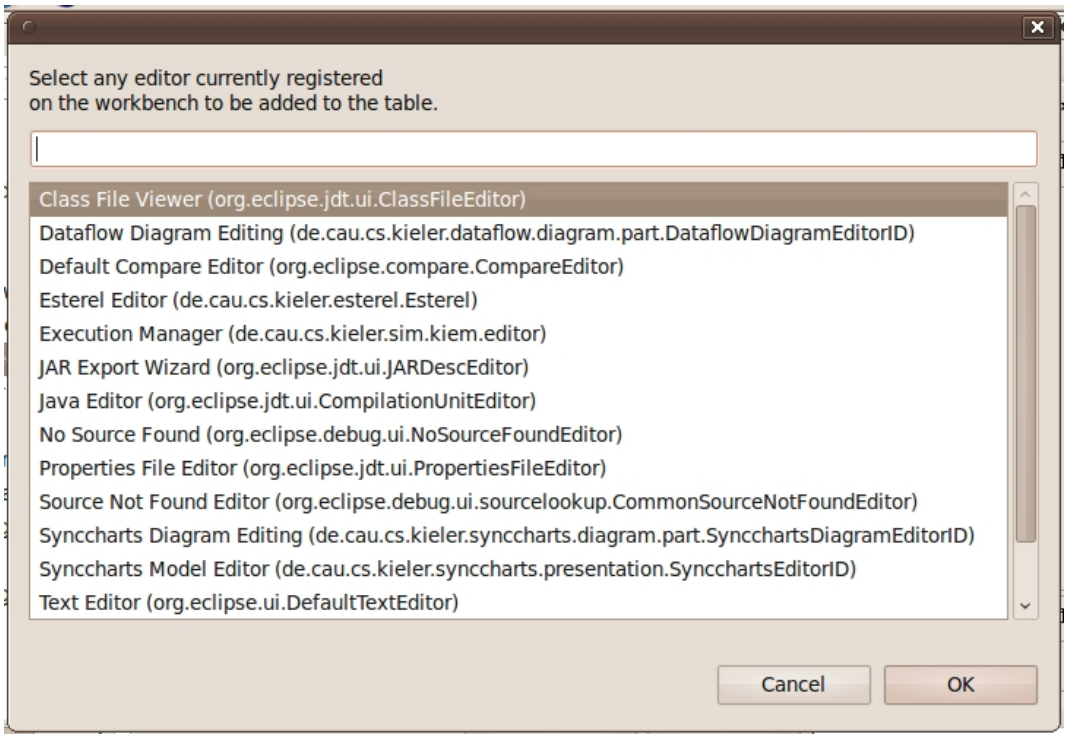

Figure 6.3: The Editor Selection Dialog

7 Conclusion

As stated in Chapter 3 the problem consists of two parts:

1. Find a way to add configurations to the existing execution files. Additionally find a way to allow the user the set up default preferences.
2. Make it easier to load previously saved schedules without having to locate the execution file in the workspace.

The following sections will summarize the results and provide some ideas for future work.

7.1 Results

- problems solved

7.2 Further Improvements

Although all initial goals of this thesis were more than met there are still some features that could not be added due to the lack of time.

Eclipse Runtime Mechanism

The Eclipse framework provides a very comprehensive system to run different modules. This is used to execute Java programs or to start a new Eclipse application but there are a variety of other applications as well.

The entire Execution Manager could be refactored into using that runtime mechanism instead of setting up a run through the now present KIEM view. This means that the table that shows the DataComponents has to be moved to a new runtime page.

The controls for pausing, resuming, stopping and stepping through the execution have to be moved somewhere else as well, possibly the DataTable.

Improve Storage Options

Currently DataComponents as well as the preference mechanism only allow the use of Strings to store the preferences. This means that all primitive data types can more or less be stored by conversion to a String. However more complex objects can't be stored without serializing them into a String and parsing them again on load.

A future project could try to find a way to overcome that limitation by allowing objects that implement the Serializable interface to be stored as well.

Part II

Automated Execution

8 Used Technologies

In addition to the technologies used in the first part of this thesis (see Chapter 2) other Eclipse technologies will be used as well.

The next sections will describe the technologies and give some examples of their usage in the standard Eclipse application. The technologies described here are the following:

The Job: The Eclipse Job is a mechanism for very long running tasks.

The Wizard: The wizard is a method for helping the user to set up complex tasks.

8.1 The Job

The Eclipse Job API provides the means to schedule very long running tasks. It uses a Thread to run the actual task and contains a ProgressMonitor to show the progress of the task. Since it is a task that can run independently of the current state of the workspace it can also be run in the back ground if the user desires it. An example for the use of jobs in the normal Eclipse architecture is the SVN commit operation seen in Figure 8.1.

8.2 Eclipse Wizards

Wizards are used to guide the user through the process of creating complex items by taking the information in a structured way and then generating the item from it. One example inside the Eclipse Architecture is the Java Class Creation Wizard (see Figure 8.2) In theory it is possible to open a text file and enter all the information manually. However if the wizard is used the user only has to select the class he wants to extend and the interfaces he wants to implement, activate one check box and then the wizard will create the class body, all required methods and comments for each element (see Listing 8.1) This makes it very easy for even inexperienced users to create new classes without knowing the exact syntax.

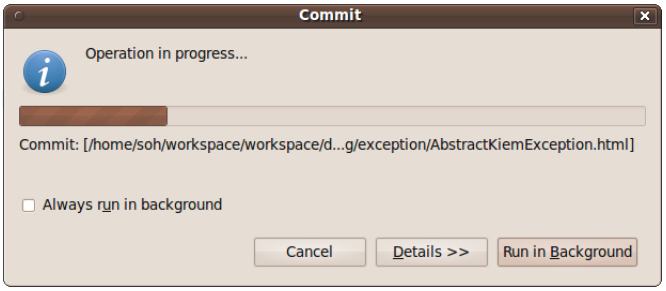

Figure 8.1: The SVN commit job

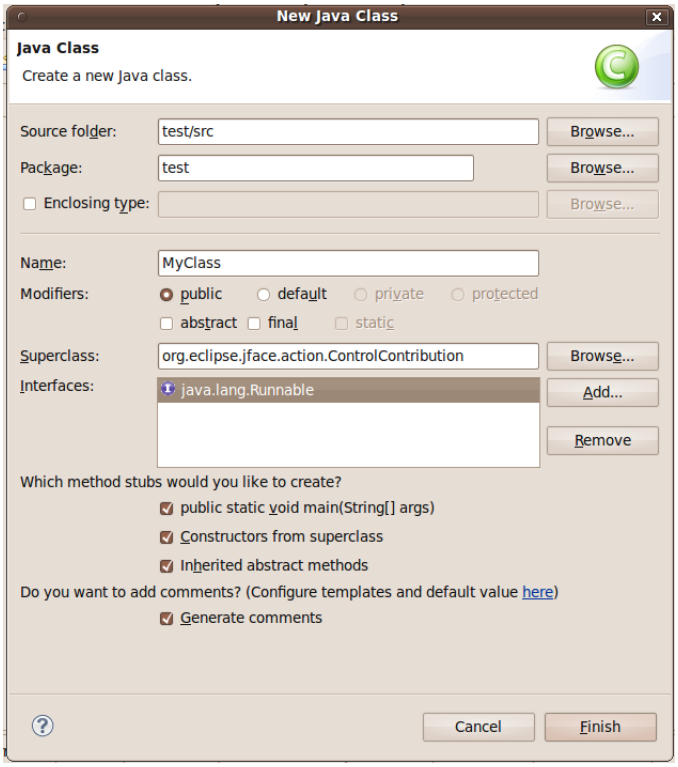

Figure 8.2: The Class Creation Wizard

Auflistung 8.1: Code generated by the wizard

```
1  package test;
2
3  import org.eclipse.jface.action.ControlContribution;
4  import org.eclipse.swt.widgets.Composite;
5  import org.eclipse.swt.widgets.Control;
6
7  public class MyClass extends ControlContribution
8     implements Runnable {
9
10    /**
11     * Creates a new MyClass.java.
12     *
13     * @param id
14     */
15    public MyClass(String id) {
16       super(id);
17    }
18
19    /**
20     * {@inheritDoc}
21     */
22    @Override
23    protected Control createControl(Composite parent) {
24       return null;
25    }
26
27    /**
28     * {@inheritDoc}
29     */
30    public void run() {
31    }
32
33    /**
34     * @param args
35     */
36    public static void main(String[] args) {
37    }
38 }
```

9 Problem Statement

The objective of this project is to find a way to automate the execution runs of the Execution Manager as described by Christian Motika in his diploma-thesis[1].

Currently the Execution manager works in a way that the user manually sets up a new execution run or loads a saved execution file. The DataComponents then have to gather all information they need themselves like model files, trace files and so on. Since there is no generic way to do that, this information is either hard-coded into the components or entered manually through the properties. After that the user has to manually control the execution. The execution runs until the user or a component stops it. The user then has to manually set up another execution run, possibly even rewriting his components if the model files and trace files are hard-coded or manually change the properties. This is very unsatisfactory if you have a large number of model files that should be tested with a one or more execution files and possibly hundreds of trace files (see Figure [TODO]). Performing runs like that manually is completely out of the question as even with the automation in place it would take several hours.

The task resulting from this problem can be broken down in 4 parts which are explained in detail in the following sections:

1. The setup of an automated run by the user.
2. The input of all the necessary information.
3. The control flow of the automated run.
4. The gathering of information after the run has finished and the display of that information.

9.1 Setting up a Run

The first objective is to find an easy way for the user to efficiently set up an automated run. This involves selecting the model files and execution files needed for the automation as well as entering initial properties.

9.2 Input for the Automation

The second objective is to enable the components to receive inputs. Each component should receive all information it needs prior to each execution run in order to make the components more dynamic. This mechanism would ensure that components can

be written in a more generic way than is currently possible. We will have to define an API for this information passing process as well as an API to trigger an automated execution from other plug-ins.

9.3 Automate the execution

The third objective is to automate the control flow of the execution itself. This would involve the following:

1. Loading the desired execution files, model files and trace files.
2. Determining how many steps should be performed and running the execution up the desired step.
3. Gathering the information produced by the components.
4. Properly shut down the execution so that a new one can be started.

9.4 Output execution results

The last objective is to display the information in a meaningful way. This should involve at least two methods of output:

1. A formatted string possibly in an XML fashion that can be parsed and used by other plug-ins for automated analysis.
2. Some graphic component that will display the information in a way that is easy to read for most users.

10 Concepts

10.1 Setting up a Run

There are several possibilities of how to solve the problem of accumulating large amounts of information prior to a long running action. The first possibility would be to have the user enter the paths to the necessary files in text files, parse those files and start a run with the parsed information. While this is a good method for performing static runs from a console environment it has several disadvantages inside the GUI of an Eclipse Rich-Client Application (RCA):

- Manually entered file names in a text file are prone to have erroneous information. It is very hard to manually enter the correct file name of any file and the entered location only works on one Operating System (OS). Aside from that it takes a long time to manually enter the possible vast amount of files used.
- There is no way to quickly adjust the file if other models or execution files should be used.
- It also means more files cluttering up the workspace.
- It is not very intuitive and the user has to know the exact syntax that the execution needs.

Another approach is the selection of the files through a dialog. Here the first option is to write a new dialog from scratch. While this option ensures flexibility since only the elements that are really needed are on it in exactly the way they are needed some disadvantages come along:

- It involves a lot of work since every widget has to be manually placed on the dialog.
- It involves even more work to get the layout of the dialog just right.

A more comprehensive approach would be to use one of the dialogs provided by Eclipse specifically the wizard type dialog. Eclipse itself uses a host of wizards as explained in Section 8.2. The wizard has several advantages over the other methods explained here:

- Even inexperienced users can be guided to set up a valid execution run.
- The entered information is most likely valid since the wizard only displays valid files.
- It is quicker to program and easier to adjust than any of the other methods.

10.2 Input for the Automation

In order to send information to the DataComponents (see Section 1.2.1) the first decision must consider the form of the information that will be supplied. The chosen form is that of a list of (key, value) pairs. It allows for most flexibility while still being very generic and simple to read and write on. This list of properties will at least include the path to the model file (see Section 1.1.1) in order for components to be executed with several different model files without having to alter the code between runs.

The next decision involves how the components are getting the information.The first possibility would be to have the component ask the plug-in for the information in question. The upside of this would be that components are sure to get all the information they need before the execution can start since they can keep asking for it. However this would likely mean that the component has to poll multiple times as it has no knowledge about when the required information will be available which constitutes additional workload. Furthermore this situation would likely mean that multiple components might request information at the same time. This means that there would be the need for substantial synchronization mechanisms to ensure consistency of data.

Therefore the way chosen in this thesis is that the Execution Manager will inform interested components about all properties that were accumulated and then starts the execution run. This ensures that a run is started in any event and keeps communication between the components and the manager simple.

10.3 Automate the execution

Automating the execution itself requires the plug-in to interact with the Execution Manager. There is already an API defined for loading an execution file by supplying a path so that is what will be used in this project. Then it is necessary to initialize the execution and step through it using the API methods provided in the Execution Manager. For this the EventListener extension point (see Section 5.1.3) of the Execution Manager can also be used in order to determine when a step has finished executing and a new one can be dispatched. After the execution is finished all components should be called again to be given a chance to provide information for the display in the next step. This information will be gathered in the same form and way as described in Section 10.2.

10.4 Output execution results

On the subject of displaying the information several options are available.

The first option would be to open a dialog once the execution has finished and display the results in a tabular manner. This has the advantage that the users attention is immediately caught when the run finishes. However pop-up boxes should

be used only when something very important happens and only with small messages as they tend to interrupt the users work flow. Aside from that the user might want to look at the results from the execution and compare them with the actual model. An opened dialog is usually something sitting in front of the rest of the IDE and that user wants to get rid of as fast as possible.

The option taken in this project will utilize the view mechanism provided by Eclipse. This approach ensures that the user can place the displayed results where he wants them to be. It also makes it possible for the user to run an execution multiple times and compare the results by having them displayed in multiple views or next to each other in the same view. Another advantage of the view concept is that it provides a tool bar for adding actions. The user might want to control the automated run during its execution or interact with the displayed results. While a static dialog would have difficulties providing the control elements for those actions a view can easily display them in the tool bar.

11 Code changes in KIEMPlugin

In order for the Automated Executions for the KIEM (KIEMAuto) to operate successfully there were no changes necessary beyond the ones described in Chapter 5.

The KIEMAuto makes use of the event listener extension point (see Section 5.1.3) in order to get notified about events occurring during the current execution run.

It also uses the tool bar contributor extension point (see Section 5.1.1) to add new control elements to the Execution Managers tool bar.

12 The Automated Executions Plug-in

- new plug in - handles the setup, control flow and display of automated execution - consists of 3 parts explained in detail below - wizard, manager, view

12.1 Automated Component

An automated component is any DataComponent (see Section 1.2.1) that wants to interact with the automated execution plug-in. As seen in the diagram automated components have to implement three methods:

12.1.1 provideProperties()

This method enables components to receive information prior to each execution run. The list is implemented as an array of key, value pairs stored inside KiemProperty objects. At the every least the list contains the location of the model file and the index of the currently running iteration (that is how many time the current model has already been executed with the current execution file). This allows components to load additional files that are always in the same path as the execution file and determine which of those to load based on the iteration index.

12.1.2 int wantMoreSteps()

This method is called before the Automation Manager performs the first step. All components will be asked how many steps they are likely to need for their execution run. The maximum of these values will be taken and the execution will perform the requested number of steps. After that the components will be asked again and so on. The process stops when all components answer with zero.

12.1.3 int wantsMoreRuns()

This method works analog to the wantMoreSteps() method in the context of entire execution runs. It is used to determine how many iterations should be performed with the given combination of execution file and model file.

12.2 Automated Producer

This interface extends the AutomatedComponent interface. In addition to the inherited methods it provided one additional method. This method is called after an

iteration has finished and asks the components if they want to publish any information about the results of their execution. This information is gathered by the plug-in and the accumulated results are either passed to the calling plug-in or displayed in the specially designed view (see chapter about the view).

12.3 Automation Wizard

12.3.1 File Selection Page

- wizard is used to set up the execution run - extends ResourceImportWizard for displaying a folder/file structure for selecting files from - easily usable, select whole folders, filter file types - can be given an initial selection, on close will save the selection, store it in preference store and restore it on load - additional dialog for selecting execution files that are not in the workspace but imported - for simplicity assume that files ending .execution are execution files and all other selected ones are model files, wizard can not check if valid since formats are not known - only allow user to proceed if at least one execution and one model file is selected

12.3.2 Property Setting Page

- set up the additional arguments passed to the execution - simple adding and removing of key, value pairs - same as file page, on close results are saved to preference store and restored for initial selection on next open

12.3.3 Information Processing

- gather execution files and model files from file selection page - gather properties from property page - store information for next open - invoke the execution manager

12.4 Automation manager and Automation Job

- handles control flow during the automation - takes information from either call through the API or wizard

12.4.1 Automation Manager

- handles the overall control flow - takes the execution files, model files and properties as argument - if progress monitor is registered it is informed about the progress of the evaluation - The control flow:

- iterate over all execution files
- open execution file
- tell view to set up display for the first execution file

- iterate over all model files
- get first model file from list
- ask components how many more runs they need, take maximum and perform runs before asking again
- pass model file, execution file and index of iteration
- initialize the execution
- pass properties to components, at least receive model file and iteration
- start worker thread that listens for monitor canceling, step timing out
- ask components how many more steps they need, take maximum and perform steps before asking again
- perform one step, lock self inside semaphore, stay locked until either worker thread or event listener notifies (step done)
- when no component wants more steps pause
- gather information from all IAutomatedProducers
- tell view to show information for this iteration
- stop execution inside the KIEM and perform cleanup
- proceed to next iteration
- inform monitor of progress
- proceed to next model
- proceed to next execution
- when done inform monitor of done and terminate the job

12.4.2 Automation Job

- workbenchjob with progressmonitor - used to display the progress in the progress view and a dialog with progress bar - long running task, doesn't want to block the rest of the workbench - triggers execution in the manager

12.5 Automation View

- displays the information in a structured way - start a new table for each execution file - one row for each iteration with each model file - prerequisite needed here: always the same outputs throughout the entire execution file - first columns display model file name, iteration index and current status

12.5.1 Tool bar

- button to start the wizard - button for clearing the view - text field showing the step that was just processed

13 Conclusion

As stated in Chapter 9 the problem consists of four parts:

1. Find an easy way for the user to set up an automated run.
2. Input the information provided by the user into the data components.
3. Design a control flow for an automated run.
4. Organize the output of the automated run and display it to the user.

The following sections will summarize the results and provide some ideas for future work.

13.1 Results

- setup using the wizard
- input using new interfaces
- control flow with job mechanism
- output using a view with tables and methods to export

13.2 Further improvements

- allow the user to select which columns to export/display
- implement macro step support as soon as KIEM does

Appendix

Appendix A

```
#Wed Feb 10 16:18:38 CET 2010
eclipse.preferences.version=1
SCHEDULE_CONFIGURATION=
  <SCHEDULE_DATA>
    <LOCATION>/de.cau.cs.kieler.sim.kiem/example.execution</LOCATION>
    <CONFIG_DATA_COMP>
      <KIEM_PROPERTY>
        <Key>de.cau.cs.kieler.syncchartsdiagram.part.
            SyncchartsDiagramEditorID</Key><Value>5</Value>
      </KIEM_PROPERTY>
    </CONFIG_DATA_COMP>
  </SCHEDULE_DATA>
  <SCHEDULE_DATA>
    <LOCATION>/test/noname2.execution</LOCATION>
    <CONFIG_DATA_COMP>
      <KIEM_PROPERTY>
        <Key>de.cau.cs.kieler.syncchartsdiagram.part.
            SyncchartsDiagramEditorID</Key><Value>2</Value>
      </KIEM_PROPERTY>
    </CONFIG_DATA_COMP>
  </SCHEDULE_DATA>
DEFAULT_CONFIGURATION=
  <KIEM_PROPERTY>
    <Key>AIMED_STEP_DURATION</Key><Value>500</Value>
  </KIEM_PROPERTY>
  <KIEM_PROPERTY>
    <Key>TIMEOUT</Key><Value>5000</Value>
  </KIEM_PROPERTY>
EDITOR_IDS=
  <EDITOR>
    <EDITOR_NAME>Syncchartsu Diagram Editing</EDITOR_NAME>
    <EDITOR_ID>de.cau.cs.kieler.syncchartsdiagram.part.
        SyncchartsDiagramEditorID</EDITOR_ID>
  </EDITOR>
```

Auflistung 1: Example for a configuration saved into the Eclipse preference store

Index

Bibliography

[1] Christian Motika, Semantics and Execution of Domain Specific Models, 2009.

[2] Object Technology International, Inc. Eclipse Platform Technical Overview, 2003.

[3] Eric Clayberg and Dan Rubel. Eclipse Plug-ins. Addison Wesley, 2009.