Christian-Albrechts-Universität zu Kiel

Bachelor Project

Configurations and Automated Execution in the KIELER Execution Manager

cand. inform. Sören Hansen

February 18, 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

Listings

List of Tables

Abbreviations

API	Application Programming Interface
CAU	Christian-Albrechts-Universität zu Kiel
CSV	Comma-Separated Values
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
KIML	KIELER Infrastructure for Meta Layout
KLePto	KIELER leveraging Ptolemy
KSBasE	KIELER Structure Based Editing
OS	Operating System
RCA	Rich-Client Application
SVN	Subversion
UI	User Interface
XML	Extensible Markup Language

1 Introduction

The purpose of this thesis consists of two parts. The first part is to find an easier way and more flexible way to deal with execution files in the Kiel Integrated Environment for Layout Eclipse Rich Client (KIELER) Execution Manager (see Section 2.2). The second part is to find an easy way to automatically do long execution runs inside the KIELER Execution Manager (KIEM).

1.1 KIELER Framework

Since the project is part of the KIELER[1] framework a short introduction is in order.

KIELER is an open-source project for model design, simulation and analysis. It is developed by the Real-Time and Embedded Systems Group[2] of the Department of Computer Science[3] of the Christian-Albrechts-Universität zu Kiel (CAU)[4].

It contains facilities to edit different types of graphical models (e.g. synccharts) and has several tools to make the editing easier. The KIELER Structure Based Editing (KSBasE)[5] project for example contains several features where the user can edit successor states to already existing states in the diagram or adding new inner states without having to perform all the necessary actions himself. Another project[6] automatically layouts any diagram with a variety of algorithms in order to improve readability.

The KIELER Execution Manager described in the next section is used for simulating the model files created by KIELER.

1.2 Outline of this Document

The first part of this document is about the implemention of the Configurations for the KIEM (KIEMConfig) plug-in.

It starts with an introduction into the technologies that were used to solve the problem as well as an overview of similar projects. The part continues with a detailed description of the problem followed by a chapter about a conceptual solution to those problems. The next chapter is about the modifications that had to be made to the existing Execution Manager. In the following chapter a detailed description of the

[1]http://www.informatik.uni-kiel.de/rtsys/kieler/
[2]http://www.informatik.uni-kiel.de/rtsys/
[3]http://www.informatik.uni-kiel.de
[4]http://www.uni-kiel.de
[5]http://rtsys.informatik.uni-kiel.de/trac/kieler/wiki/Projects/KSBasE
[6]http://rtsys.informatik.uni-kiel.de/trac/kieler/wiki/Projects/KIML

implementation of the new features will be given. The last chapter will summarize the results of this thesis and outline a few projects that could follow up on it.

The second part discusses the implementation of the Automated Executions for the KIEM (KIEMAuto) plug-in. It follows the same structure as the first part.

Part I

Configuration Management

2 Used Technologies

Before the problem can be explained the technologies in question and the terminology that is used in the rest of this document is in order must first be introduced. This should only serve as a brief outline since a full explanation goes beyond the scope of this thesis.

2.1 Eclipse

Since the KIELER project and thus the Execution Manager is build upon the Eclipse framework a short introduction into Eclipse is necessary.

The basic function of Eclipse is as an Integrated Development Environment (IDE) for Java. It provides a host of facilities that makes it easier for the user to create their own Java applications. A few examples for these facilities are:

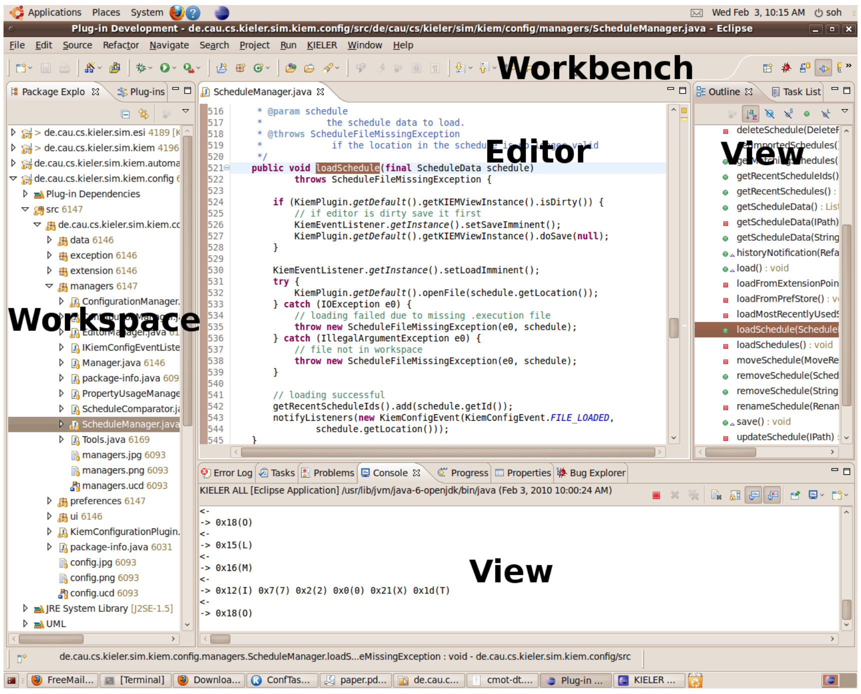
- Syntax highlighting to make the source code easier to read.
- Automatic completion of partial commands to ensure correctness and make it easier to write code.
- Content assist to create better code and remove errors.
- Several wizards for class creation and other tasks.

However since Eclipse is an open-source project there are also plug-ins (see Section 2.1.1) for variety of other things. For example the language isn't limited to Java. There are also plug-ins for C++, LaTeX, Visual Basic and several other programming languages.

Through the use of different modeling framework Eclipse can also be used as an IDE for IDEs. That makes Eclipse "an IDE for anything, and nothing in particular" (2).

The terminology used for the different basic parts of Eclipse can be illustrated based on Figure 2.1:

- The main window of Eclipse is called the *workbench*. The workbench consists of the different editors and views.
- The files that the user operates on are located in the Eclipse *workspace*.
- An *editor* is a component that allows the user to display, enter and modify information. Editors are used to modify a specific file type. There can usually be multiple instances of the same editor. An example for an editor would be the Java editor which is used to create and edit Java source files.



```java
     * @param schedule
     *            the schedule data to load.
     * @throws ScheduleFileMissingException
     *             if the location in the schedule is no longer valid
     */
    public void loadSchedule(final ScheduleData schedule)
            throws ScheduleFileMissingException {

        if (KiemPlugin.getDefault().getKIEMViewInstance().isDirty()) {
            // if editor is dirty save it first
            KiemEventListener.getInstance().setSaveImminent();
            KiemPlugin.getDefault().getKIEMViewInstance().doSave(null);
        }

        KiemEventListener.getInstance().setLoadImminent();
        try {
            KiemPlugin.getDefault().openFile(schedule.getLocation());
        } catch (IOException e0) {
            // loading failed due to missing .execution file
            throw new ScheduleFileMissingException(e0, schedule);
        } catch (IllegalArgumentException e0) {
            // file not in workspace
            throw new ScheduleFileMissingException(e0, schedule);
        }

        // loading successful
        getRecentScheduleIds().add(schedule.getId());
        notifyListeners(new KiemConfigEvent(KiemConfigEvent.FILE_LOADED,
                schedule.getLocation()));
    }
```

```
KIELER ALL [Eclipse Application] /usr/lib/jvm/java-6-openjdk/bin/java (Feb 3, 2010 10:00:24 AM)
<-
-> 0x18(O)
<-
-> 0x15(L)
<-
-> 0x16(M)
<-
-> 0x12(I) 0x7(7) 0x2(2) 0x0(0) 0x21(X) 0x1d(T)
<-
-> 0x18(O)
```

Figure 2.1: The Eclipse workbench window

- An Eclipse *view* is the other component located on the workbench. Views are only used to display content that was created elsewhere. Unlike editors there is usually only one instance of any view. One of the views shown in the figure is the class outline view. It shows all methods and attributes of the Java class in the currently active editor.

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

2.1.1 Plug-ins

The building blocks of any Eclipse application are the so called *plug-ins*. They consist of any number of Java classes and additional meta information. The Java classes describe the behavior of the plug-in and define its Application Programming Interface (API). The meta information is not written in Java but uses an Extensible Markup Language (XML) notation instead. The meta information contains the following information:

- What other plug-ins does the plug-in depend on? This information is necessary to determine which other plug-ins have to be loaded or when to refuse loading the plug-in because of missing dependencies.

[1]www.eclipse.org

- What *extension points* does the plug-in offer? These are part of the API and will be described below.
- What functionality does it add to the plug-ins which it extends?

Plug-ins encapsulate their internal behavior and can be accessed through the API and the *extension points*. They provide a specific functionality that can be reused as long as the dependencies are met. As such an Eclipse application consists of a mosaic of different plug-ins that can be exchanged at will.

Eclipse can not only be used to create plug-ins that can be used in an Eclipse instance but can also compile a set of plug-ins into a standalone application - the so called Rich-Client Application (RCA). This RCA contains a minimal set of plug-ins to provide the Eclipse look-and-feel and the plug-ins created by the user.

Extension Point Mechanism

The extension point mechanism is one of the key features of plug-in development in Eclipse. It extends the API provided by the public methods of the different Java classes inside the plug-in. An extension point definition consists of a tree of different configuration elements. Each configuration element has different attributes some of which can be optional while other are mandatory. These attributes can be anything from a String, a file to a Java class that has to extend one class and implement a specific interface.

A plug-in that wants to add their functionality to an already existing plug-in through the use of an extension point has to provide the mandatory attributes defined in the specifications.

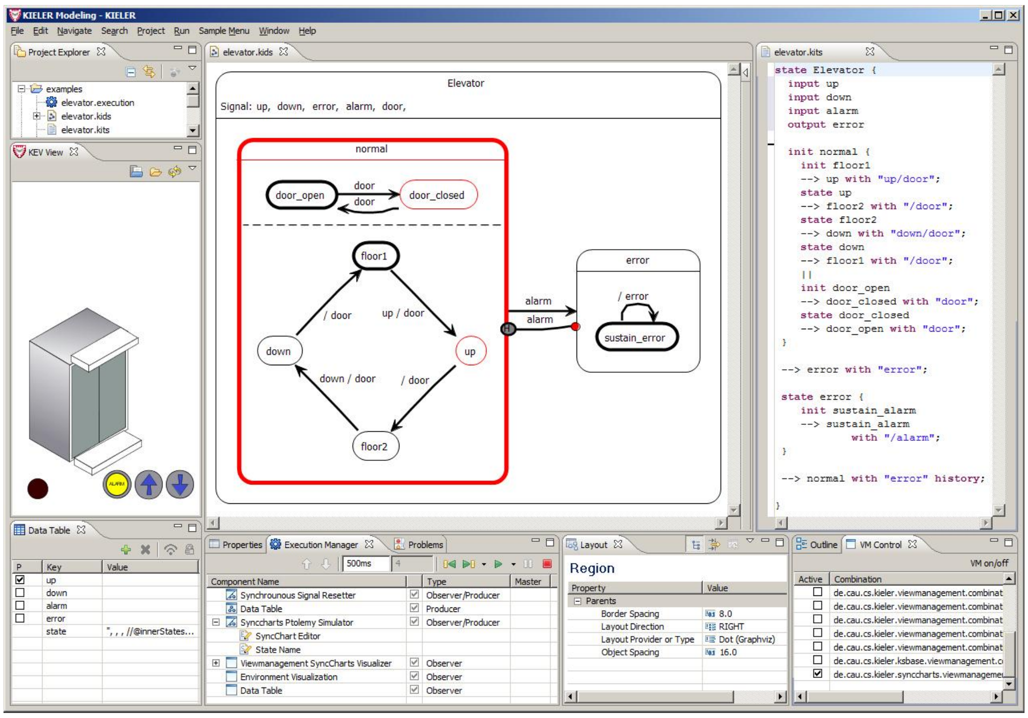
Eclipse itself already provides many extension points to extend the functionality of the workbench. For example, if a plug-in wants to add a new editor to the workbench it has to extend the *org.eclipse.ui.editors* extension point. It then has to provide an identifier and a name as well as a class that implements the *org.eclipse.ui.IEditorPart* interface. It can also specify an icon and a file extension for files which should be opened with the new editor.

When an Eclipse application is started with this plug-in Eclipse will automatically make sure that the new editor can be used to open the specified file type. The programmer only has to concern himself with the area of the editor itself without worrying about it being added at all the necessary places inside the Eclipse architecture.

2.1.2 Preference Pages

A special example of plug-in usage within the Eclipse framework itself 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.



```
state Elevator {
 input up
 input down
 input alarm
 output error

 init normal {
   init floor1
   --> up with "up/door";
   state up
   --> floor2 with "/door";
   state floor2
   --> down with "down/door";
   state down
   --> floor1 with "/door";
   ||
   init door_open
   --> door_closed with "door";
   state door_closed
   --> door_open with "door";
 }

 --> error with "error";

 state error {
   init sustain_alarm
   --> sustain_alarm
          with "/alarm";
 }

 --> normal with "error" history;

}
```

Data Table

P	Key	Value
☑	up	
☐	down	
☐	alarm	
☐	error	
	state	", , , //@innerStates...

Component Name		Type	Master
Synchrounous Signal Resetter	☑	Observer/Producer	
Data Table	☑	Producer	
Syncharts Ptolemy Simulator	☑	Observer/Producer	
SyncChart Editor			
State Name			
Viewmanagement SyncCharts Visualizer	☑	Observer	
Environment Visualization	☑	Observer	
Data Table	☑	Observer	

Region

Property	Value
Parents	
Border Spacing	8.0
Layout Direction	RIGHT
Layout Provider or Type	Dot (Graphviz)
Object Spacing	16.0

Active	Combination
☐	de.cau.cs.kieler.viewmanagement.combinat
☐	de.cau.cs.kieler.viewmanagement.combinat
☐	de.cau.cs.kieler.viewmanagement.combinat
☐	de.cau.cs.kieler.viewmanagement.combinat
☐	de.cau.cs.kieler.viewmanagement.combinat
☐	de.cau.cs.kieler.ksbase.viewmanagement.c
☑	de.cau.cs.kieler.synccharts.viewmanageme

Figure 2.2: The Execution Manager during a simulation.

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.

2.2 The KIELER Execution Manager

Execution Manager shown in Figure 2.2 is used in KIELER as a framework to plug-in DataComponents (see Section 2.2.2) 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 KIEM see the wiki[2].

2.2.1 KiemProperty

The basic KiemProperty object holds a (key, value) pair of type String. It is used to store information inside the DataComponents. There are also advanced KiemProperties which can contain other types of values like integers, booleans or files.

2.2.2 DataComponents

A DataComponent is an entity that has a specific task during a the course of an execution. The DataComponents are scheduled in a specific order and can either receive or produce information or both. During each step of the execution each DataComponent is asked to perform their computations for the step. Every DataComponent contains a list of KiemProperties that are used to allow some configuration of the component after it has been loaded.

DataComponentWrappers

A DataComponentWrapper is an object that wrapps around a DataComponent. The wrappers are stored in the schedule of an execution file.

2.2.3 Execution

The files used by the Execution Manager are called *execution files*. These files contain the list of DataComponents with all internal data and the specific order. That list is called a *schedule*. The schedule can be used to start an *execution* in the Execution Manager. An execution consists of a initialization, stepping and wrap-up.

2.2.4 Model Files

A model file is not a concept of the Execution Manager as such but rather a general concept. However since this thesis only uses model files in the context of the KIEM it seems appropriate to explain them here.

A model file can be any file that contains the structure and behavior of a semantic entity. This can be something as simple as a text file containing a list of elements in a certain order. Most model files that are used in the KIELER project are diagrams that consist of nodes and links. These *syncacharts* are build in a hierarchical structure and are therefore stored as a tree of different elements. For an example of a simple syncchart see Figure 2.3.

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

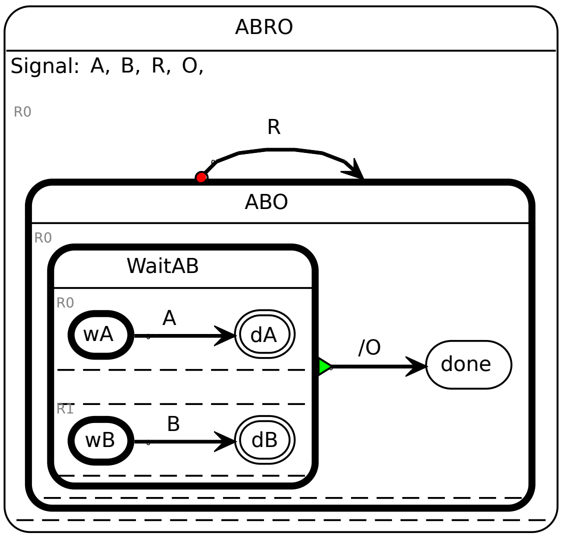


Figure 2.3: An example for a simple syncchart diagram.

2.3 Related Work

There are of course a huge amount of other applications and projects that deal with configuration management. In order to get some idea of similar projects a few examples from Eclipse itself and the KIELER project will be used.

2.3.1 Configurations

One of the tasks that will be explained in Section 3.1 is to add new configuration information to existing execution files. This task can be compared to the KIELER Infrastructure for Meta Layout (KIML) project.

KIML is used to automatically layout existing diagrams. Since diagrams basicly consist of nodes and connections generic algorithms can be used to layout almost any type of diagram. However the user still has some control over the details of the automatic layout process. The user can configure details like the distance between different elements, the direction of the layout or the layouter that should be used.

This meta-information has to be stored somewhere. The approach used in KIML was to create a separate file. The file has the same name as the diagram file and contains all the layout information needed for it. The information can be easily reset by deleting the separate file.

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 Schedule 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.

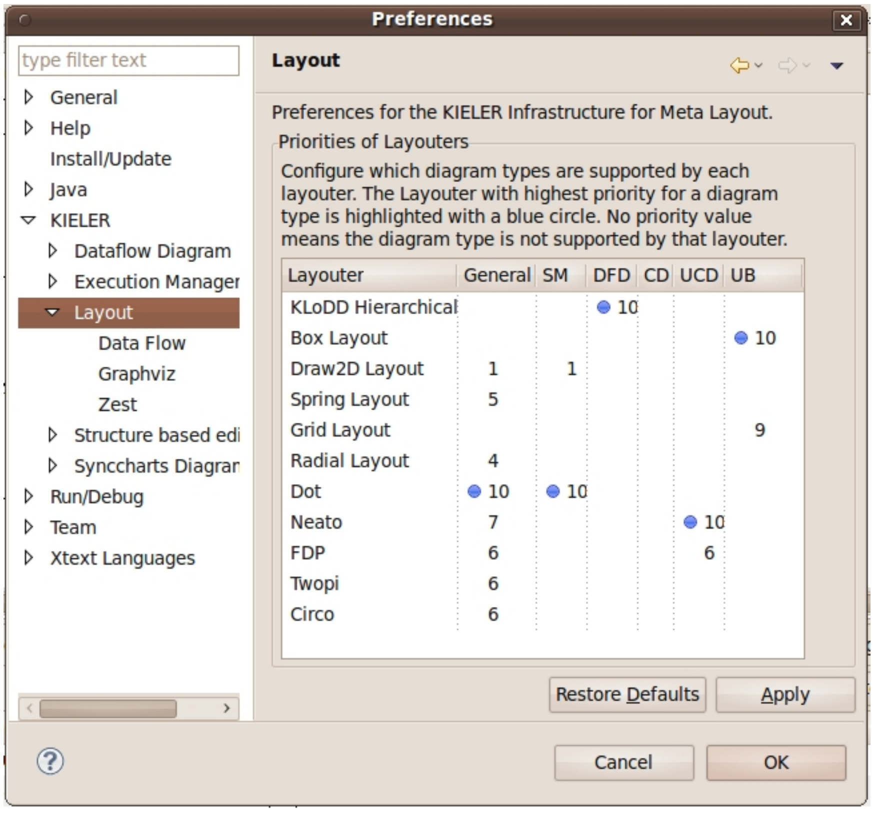
4.1 Configurations

The first approach to save additional configuration information in the execution files (see Section 2.2.3) 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 2.2.2). 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 carry a list of the KiemProperties described in Section 2.2.1. The properties can carry a list of (key, value) pairs which means 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.



Preferences for the KIELER Infrastructure for Meta Layout.

Priorities of Layouters

Configure which diagram types are supported by each layouter. The Layouter with highest priority for a diagram type is highlighted with a blue circle. No priority value means the diagram type is not supported by that layouter.

Layouter	General	SM	DFD	CD	UCD	UB
KLoDD Hierarchical			● 10			
Box Layout						● 10
Draw2D Layout	1	1				
Spring Layout	5					
Grid Layout						9
Radial Layout	4					
Dot	● 10	● 10				
Neato	7				● 10	
FDP	6				6	
Twopi	6					
Circo	6					

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 the Eclipse preference page mechanism will be used (see Section 2.1.2).

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.

The next page that needs to be constructed will be used for managing the different schedules and their editors. For that the LayoutPreferencePage by Miro Spönemann as seen in Figure 4.1 will be slightly adapted. 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 directly mapped to the list of 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 Schedule Loading

To allow the user to easily load previously saved execution files two ComboBoxes will be introduces to the Execution Managers tool bar.

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 the KIEMConfig plug-in will be notified of that event by the Execution Manager. A new object will then be created which represents the execution file and contains its path. This schedule object will also be added to the list of recently used schedules that is maintained through the use of the Eclipse preference store.

When the user selects one of the previously saved schedules in one of the ComboBoxes the saved path will be retrieved and relayed to the Execution Manager 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.

Listing 5.1: The interface for ToolbarContributionProviders

```java
public interface IKiemToolbarContributor {

    /**
     * The plugin will add the components from left to right in the order that
     * the contributors are stored in the extension registry. KIEM's own
     * controls will be added after the contributed components have been added.
     *
     * The array should contain the components in the order that they are
     * supposed to be added, null values will be ignored.
     *
     * @param info
     *            may hold some information.
     * @return the list of controls that should be contributed.
     */
    ControlContribution[] provideToolbarContributions(Object info);
}
```

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 Listing 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 for example Labels, Buttons or 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 5.1 shows the Execution Manager tool bar with the two combo boxes belonging to KIEMConfig contributed through the extension point. Figure 5.2 shows the same tool bar without the contributions.

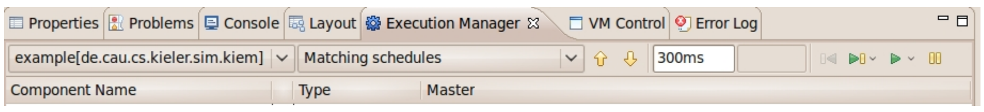


Figure 5.1: The Execution Managers Tool bar with two contributed ComboBoxes

Listing 5.2: The Interface of the Configuration Provider

```java
public interface IKiemConfigurationProvider {

    /**
     * Ask the component to give a new value for the property specified by the
     * id. If multiple components are registered on this extension point only
     * the first value that was successfully retrieved will be used. All other
     * providers will not be asked.
     *
     * @param propertyId
     *            the id of the property to change.
     * @return the new value of the property.
     * @throws KiemPropertyException
     *             if the propertyId was not found.
     */
    String changeProperty(String propertyId) throws KiemPropertyException;

    /**
     * Notify the listener that the user changed the property specified by the
     * id.
     *
     * @param propertyId
     *            the id of the property.
     * @param value
     *            the new value of the property.
     */
    void propertyChanged(String propertyId, String value);
}
```

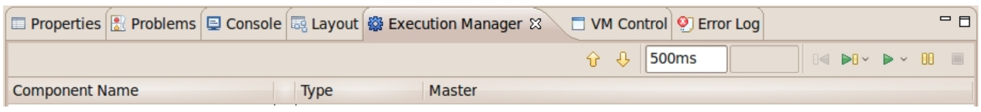


Figure 5.2: The Execution Managers Tool bar without contributions

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 Listing 5.2 work in the following way:

String changeProperty(String propertyId) throws KiemPropertyException :

This method will be called by the Execution Manager whenever a property has to be loaded where other plug-ins are encouraged to provide their value. When a plug-in is asked for a value it can do one of two things:

1. It can either provide a value for the property. Any value is acceptable here, even *null*. If one plug-in provides any value at all the other plug-ins will not be asked. The reason behind this arrangement is that the Execution Manager most likely can't decide which value has more validity anyway if more than one plug-in answers.
2. If it can not provide a value the declared Exception should be thrown in which case the Execution Manager will move to the next plug-in. The reason for not using the *null* value to encode no valid value is that *null* might be the intended value.

void propertyChanged(String propertyId, String value) :

Notifies all listeners that a property was changed somewhere in the Execution Manager. This will be called for example when the user changes the step duration through the input field on the Execution Managers tool bar.

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 List 5.3) 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.

Listing 5.3: The Interface of the Event Listener

```java
public interface IKiemEventListener {

    /**
     * This is the basic notify method that is called by KIEM whenever an event
     * occurs for which this EventListener is registered (see
     * {@link #provideEventOfInterest()}).
     *
     * @param event
     *            the KiemEvent with additional attached information, depending
     *            on the specific event
     */
    void notifyEvent(final KiemEvent event);

    /**
     * Return a KiemEvent type (integer value) that represents a number of
     * events this component wants to listen to.
     *
     * A KiemEvent can be a combination of several events. The simplest way to
     * register for two events that e.g., indicate a step-command and the
     * removal of the component is to have the following code:
     *
     * public KiemEvent provideEventOfInterest() {
     * int[] events = {CMD_STEP, DELETED}
     * return new KiemEvent(events);
     * }
     *
     * @return the KiemEvent type indicating the events of interest
     */
    KiemEvent provideEventOfInterest();
}
```

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.

Listing 5.4: Example of modified Getter and Setter

```java
public int getAimedStepDuration() {
   int result = this.aimedStepDuration;
   Integer value = getIntegerValueFromProviders
      (AIMED_STEP_DURATION_ID);
   if (value != null) {
      result = value;
   }
   return result;
}

public void setAimedStepDuration(final int stepParam) {
   this.aimedStepDuration = stepParam;
   // if executing, also update current delay
   if (execution != null) {
      this.execution.setAimedStepDuration(stepParam);
   }
   this.getKIEMViewInstance().updateViewAsync();
   notifyConfigurationProviders
      (AIMED_STEP_DURATION_ID, stepParam + "");
}
```

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 example see Figure 5.4). 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 5.5. 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

Listing 5.5: The head of the modified openFile() method

```java
public void openFile(final IPath executionFile,
   final boolean readOnly) throws IOException {
   String fileString = executionFile.toOSString();
   final InputStream inputStream;

   if (fileString.startsWith("bundleentry:/")) {
      String urlPath = fileString.replaceFirst
         ("bundleentry:/", "bundleentry://");
      URL pathUrl = new URL(urlPath);
      inputStream = pathUrl.openStream();
   } else {
      URI fileURI =
         URI.createPlatformResourceURI(fileString, true);
      // resolve relative workspace paths
      URIConverter uriConverter =
         new ExtensibleURIConverterImpl();
      inputStream = uriConverter.createInputStream(fileURI);
   }

   Display.getDefault().syncExec(new Runnable() {
   ...
```

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.

The tool bar of the Execution Manager is created in a programmatic way instead of through the use of the corresponding Eclipse extension point. This means that the only way to place additional controls onto the tool bar is to modify the code in order to make use of the Toolbar Contribution Provider extension point described in Section 5.1.1. For the full code of the modified method see the Appendix.

6 The KIEMConfig 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 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 manipulating 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 performs 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 2.2.2). 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.

The ConfigDataComponent is not only used to store the properties of the currently active configuration that each execution file carries. However it is also used to store the default configuration that is saved in the Eclipse preference store. This is done because both instances are closely linked and have the same requirements (see Section 6.2.2).

The default behavior of the Configuration Manager is to add a new ConfigDataComponent to each execution file that it encounters. However as this feature can be turned off the user also has can upgrade old files or downgrade new ones by manually adding and removing the ConfigDataComponent.

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).

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 tool tips 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 keep track of the different known editors.
- The **ContributionManager** is used to manage the controls that are placed on the tool bar in the Execution Manager.

- The **PropertyUsageManager** is responsible for managing the keys of those properties where the default configuration is used rather than the current configuration.

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 the Appendix.

6.2.2 Configuration Manager

The Configuration Manager basically handles all the problems described in Section 3.1. This means that the Configuration Manager has two responsibilities:

1. It manages the configuration contained in the currently opened execution file and all properties contained in it. It is also responsible for deciding whether or not the preferences stored in that configuration should be used or the default preferences instead.

2. It manages the default configuration that the user can access and modify through the preference pages (see Section 6.3). For all of the predefined properties it also has to hold and manage the hard-coded default values.

Currently Loaded Configuration

The first thing the Configuration Manager has to do when a request for the value of a property is made is to locate the ConfigDataComponent that contains the property.

It first takes a look at the list of keys where the default configuration should be used. If this is the case the task is quite simple and the default configuration is loaded from the preference store and used. If the current configuration should be used the task is a little more difficult. The Configuration Manager then has to look at the DataComponentWrapperList inside the Execution Manager where all components for the currently opened execution file are stored. If the list already contains a wrapper with the ConfigDataComponent inside that component is used. Otherwise a new ConfigDataComponent is created, initialized with the default values from the default configuration and then added to the list of the current execution file. Since this feature can be disabled by the user the Configuration Manager can encounter execution files that have no ConfigDataComponent or where the component simply doesn't contain the requested value. In this case the default configuration has to be used.

If the default configuration couldn't supply a value the last possibility is that the caller passed a non-null default value for the given property. In this case the default value is returned to the caller.

If no value could be retrieved in the way described above there is no way to get a valid value for the requested key. In this case the Configuration Manager notifies the caller through an exception of these circumstances.

However if the value was expected in the current configuration but not found the Configuration Manager assumes that it should have been in there. To remedy that situation the Configuration Manager will try to add a new property to the current configuration with the value that the method will return (either the one retrieved from the default configuration or the default supplied by the caller).

Default Configuration

The first responsibility of the Configuration Manager with respects to the default configuration is to manage the hard-coded default values. The whole idea of the KIEMConfig is to avoid using hard-coded values and retrieve user defined values. However the Execution Manager and the KIEMConfig rely on certain values to be present and even though the user is encouraged to change them they still have to be present before the user enters them for the first time. Furthermore the user may want to revert back to sensible default values which should be provided by the plug-in itself.

This means that the plug-in contains a list of hard-coded default values for the needed properties. It also supplies methods to access these properties and restore the default values by writing their values into the default configuration.

The next feature that the default configuration supplies has to do with adding new ConfigDataComponents to the list inside the Execution Manager. Since the Configuration Manager knows which properties will be taken from the current configuration it can already make sure that the component contains some value. This is done through calling `void initWithDefaults(AbstractDataComponent dataComponent)` with the newly created component. This causes the default values for all properties that are likely to be taken from the current configuration to be added to it.

The Configuration Manager also supplies different views on the default configuration.

1. **KiemProperty[] getDefaultConfig().getProperties()** : This method simply returns all properties stored in the default configuration. This is the list

actually written to the Eclipse preference store.

2. **KiemProperty[] getInternalDefaultProperties()** : This method returns the list of properties that are needed to operate the Execution Manager and the KIEMConfig. The motivation for this method is that the keys and types for these values are already known. That means that a view that modifies these properties can be designed in a more user-friendly way than would have been possible otherwise. Furthermore the Configuration Manager is guaranteed to have hard-coded default values for these properties.

3. **KiemProperty[] getExternalDefaultProperties()** : This method returns the complement of the internal properties with respect to the entirety of the default properties. These properties are those that the user defined himself.

However neither of the last two lists will return the default editor as that falls into the responsibility of the Editor Manager which is described below.

The Configuration Manager also supplies methods to add new properties to the default configuration, remove properties and update the value of specific property.

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 six 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. Deal with loads and saves triggered through the normal workspace interface.

4. Provide a means to trigger the loading of an execution file in the Execution Manager.

5. Track the locations of the execution files if the user modifies them.

6. 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 KIEMConfig 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.

Listen to User Events

It has to be assumed that the user creates his own schedules and saves and loads them through the workspace explorer. Since the Schedule Manager should attempt to track all available execution files it has to be informed about these changes and act on the notification. In order to get notified the Event Listener extension point of the Execution Manager is used (see Section 5.1.3). The listener will dispatch and event that contains the location of any saved or loaded execution file as soon as the user performs the corresponding action. When the Schedule Manager receives such an event it will perform the following steps:

1. Try to find a schedule with the provided location. If a schedule can be found there is nothing to be done and the method skips to the last step.

2. If no schedule was found with the given location a new one has to created. In order for that the method first has to check if there is an active editor and if the editor is known to the Schedule Manager. If that is the case the method skips ahead to step 4.

3. If the editor is unknown it is created. If no editor is active the default editor will be used.

4. The new schedule is created with the editor determined in the previous steps. Since it is assumed that the created schedule works with the currently active editor a default priority is inside to that editor in the newly created schedule.

5. Since the file operation constitutes a use of the given schedule the list of recently used schedules has to be updated. The used schedule will be added to the top or moved to the top if it's already inside.

Open a Schedule

The Schedule Manager also provides a way to trigger a load in the Execution Manager. This is done through the modified method described in Section 5.2.3. The reason for this is that the Schedule Manager may try to load schedules where the files no longer exist or that the Execution Manager is unable to load. The method mentioned above will throw Exceptions in order to perform the caller of these circumstances and the Schedule Manager will inform its callers.

As described in the last section the Schedule Manager will be informed of a load through the Event Listener interface and will act on it. However if the load is triggered by the Schedule Manager itself that behavior would not be desirable. To avoid the Schedule Manager informing itself the method makes sure that the next event indicating a loaded file will be ignored.

Since the load triggered by the Schedule Manager constitutes an access to that file the associated schedule is added to the list of recently used schedules. Furthermore all listeners on the Schedule Manager will be notified since view updates may be necessary.

Tracking Execution Files

As described above new schedules will be created when the user manually loads or saved an execution file. However in some cases the user may also choose to remove, rename or move execution files. Since the Schedule Manager relies on the path of the execution file to load the schedules it runs into trouble if the path is changed outside its control.

The only solution in this case is to prompt the user to select a new path for an execution file that is no longer at the expected location. An alternative would be to simply delete the schedule and all assigned priorities.

However if the user performs a remove, rename or move through the context menu inside the Eclipse workspace window Eclipse itself provides a way to get notified of these changes. Any class can register themselves as listener by calling : `RefactoringCore.getHistoryService().addHistoryListener(IRefactoringHistoryListener listener)`. Whenever the removal, renaming or moving of a file completes all listeners

```
<extension
        point="de.cau.cs.kieler.sim.kiem.config.
            DefaultScheduleContributor">
    <configurationFile
            file="testFiles/krep.execution">
        <editor
                id="de.cau.cs.kieler.krep.evalbench.ui.editors.kasm"
                name="KASM Editor"
                priority="8">
        </editor>
    </configurationFile>
</extension>
```

Listing 6.1: Example implementation of the Default Schedule extension point.

will be notified through a call to `void historyNotification(RefactoringHistoryEvent event)`. The event contains different information based on the type of the operation. Depending on the type of operation that was performed the Schedule Manager takes one of the following actions:

Delete : One or more files were deleted by the user. In this case the event contains the list of files that was deleted. Since the user has no more use for the deleted files he probably has no use for the schedule and the saved priorities as well. The Schedule Manager attempts to find the ScheduleData object associated with the given file and removes it.

Rename : The user changed the name of a file. The event provides the old file name and the new name of the file. In order to keep track of the file the Schedule Manager tries to find the ScheduleData associated with the old file name and changes the name to the new file.

Move : One or more files were moved by the user to a new location. The event contains the list of files that were moved and the destination path. The Schedule Manager tries to find a Schedule for each of the files involved and changes its name to reflect the new location.

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 Listing 6.1 was created.

As soon as the Schedule Manager loads the schedules from the Eclipse preference store it also triggers a load from the components registered on this extension point. The new schedule will be constructed with the location provided through the corresponding field in the extension point. After that the Schedule Manager iterates through the list of child elements of a given execution file and adds all the editors and their priorities to the schedule and the Editor Manager.

Since the schedules and editors added in this way are supposed to be a kind of factory defaults they are not supposed to be removed or changed. The Schedule Manager sets a field in both the newly created ScheduleData objects and the EditorDefinitions. It is the responsibility of the different view elements not to modify entities marked in this way.

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:

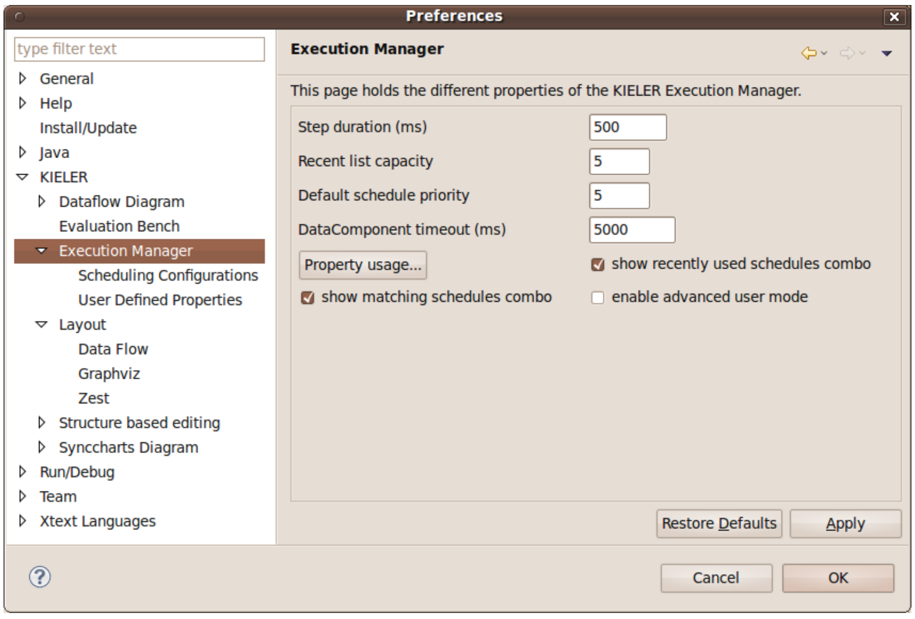
1. The manager must create the view elements and store them. As described in Section 5.1.1 the Execution Manager will ask the KIEMConfig for the list of items it wants to contribute to the tool bar. The manager then has to create the list with the saved view elements and forward it through the extension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

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.


Preferences

type filter text

- General
- Help
- Install/Update
- Java
- KIELER
  - Dataflow Diagram
  - Evaluation Bench
  - Execution Manager
    - Scheduling Configurations
    - User Defined Properties
  - Layout
    - Data Flow
    - Graphviz
    - Zest
  - Structure based editing
  - Syncchart Diagram
- Run/Debug
- Team
- Xtext Languages

Execution Manager

This page holds the different properties of the KIELER Execution Manager.

Step duration (ms)	500
Recent list capacity	5
Default schedule priority	5
DataComponent timeout (ms)	5000

Property usage... [✓] show recently used schedules combo

[✓] show matching schedules combo [ ] enable advanced user mode

Figure 6.1: The main preference page of the Execution Manager

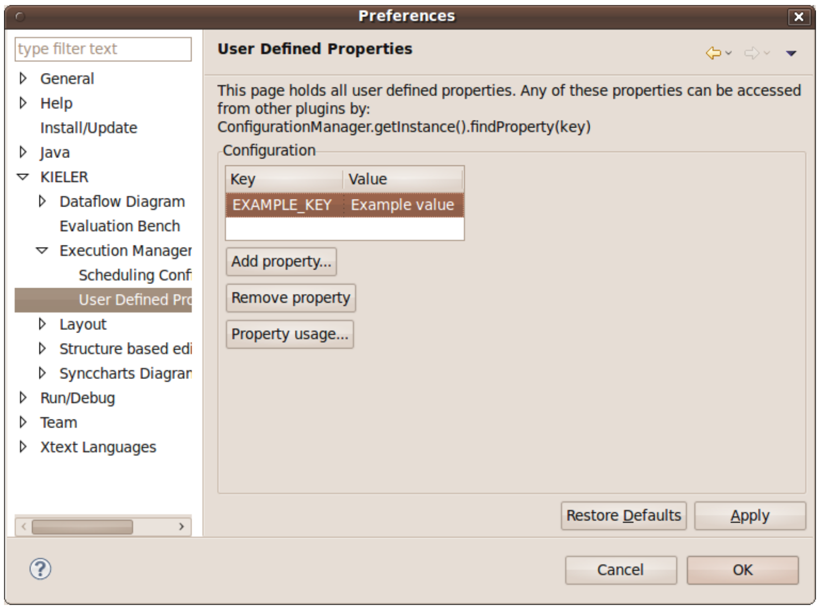
6.3 Preference Pages - the View

The view part of this project mostly consists of the preference pages for setting up the different aspects of the KIEMConfig. These preference pages use the technology described in Section 2.1.2 which integrates them into the rest of the preference page framework. The root page for the Execution Manager is added into the already existing tree of preference pages for the rest of KIELER in order to make them easier to find.

6.3.1 Configuration Page

On the main preference page of the Execution Manager shown in Figure 6.1 the user can set up most of the default properties. The user can also change the visibility of the ComboBoxes that display the recently used and matching schedules. The last CheckBox is for enabling the advanced user mode.

In the normal user mode the ConfigDataComponent described in Section 6.1.1 is not visible to the user. It is also automatically added to any new file that is loaded into the Execution Manager. While this behavior is fine for the average user an advanced user may want to have a little more control. An advantage of having the component visible is that the user can reinitialize the current configuration with the values in the default configuration by removing and adding the component in the list.



User Defined Properties

This page holds all user defined properties. Any of these properties can be accessed from other plugins by:
ConfigurationManager.getInstance().findProperty(key)

Configuration

Key	Value
EXAMPLE_KEY	Example value

Figure 6.2: The page for defining custom properties

User-Defined Properties Page

Since there is a page to modify the internal properties of the Execution Manager there also exists a preference page where the user can define, edit and remove his own properties (see Figure 6.2). These properties can then be accessed by any DataComponent (or any other plug-in) through the KIEMConfig's API.

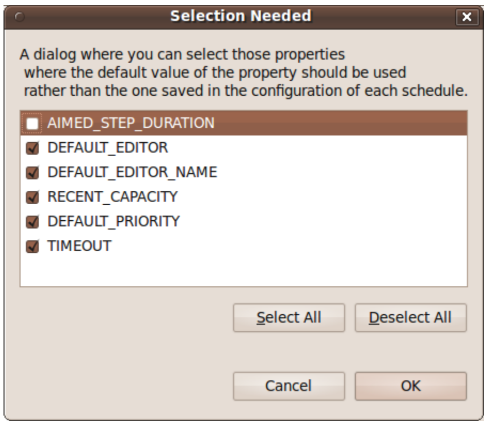
Since not much can be said about the nature of the user defined properties there is no real format that can be chosen for an individual property. Thus all properties are simply displayed in a table with a key and a value column.

The user is only allowed to edit the value column of previously defined properties. This restriction is necessary to keep the user from accidentally changing keys that are required by another users DataComponent.

However the user can remove a property that is no longer needed or define his own properties with a custom key.

Property Usage Dialog

Both of the previously described pages contain a button for accessing the Property Usage Dialog. This dialog (see figure 6.3) 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



A dialog where you can select those properties
where the default value of the property should be used
rather than the one saved in the configuration of each schedule.

- ☐ AIMED_STEP_DURATION
- ✓ DEFAULT_EDITOR
- ✓ DEFAULT_EDITOR_NAME
- ✓ RECENT_CAPACITY
- ✓ DEFAULT_PRIORITY
- ✓ TIMEOUT

Figure 6.3: The Property Usage Dialog

items back to the PropertyUsageManager.

6.3.2 Scheduling Page

This preference page (see Figure 6.4 is used to manage the schedules and the editors that they belong to. As mentioned in Section 4.1.1 this page is basically a modified version of the LayoutPrioritiesPage by Miro Spönemann (see Figure 4.1).

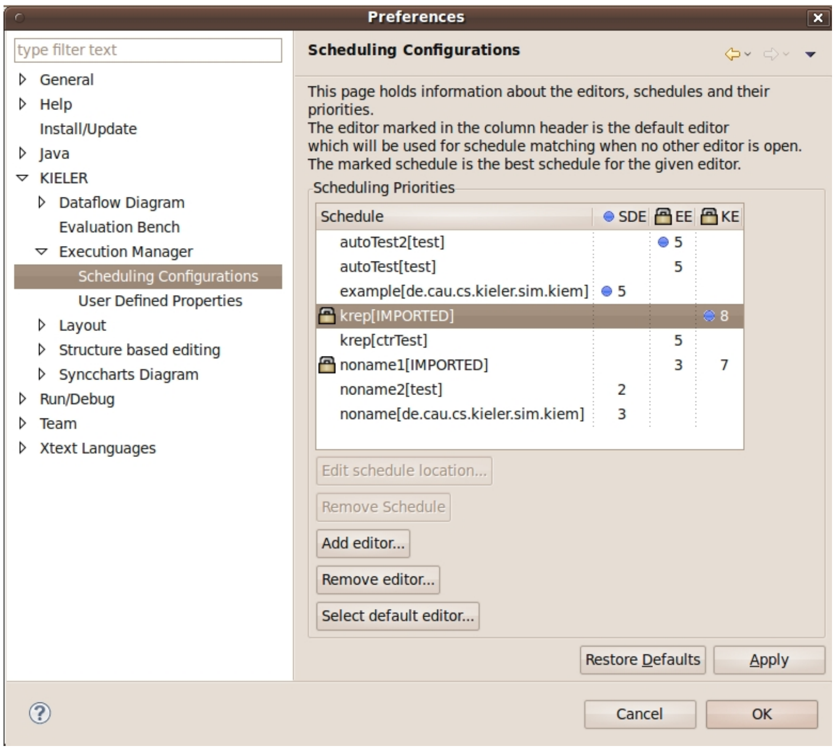
The page is divided into two parts. The top part shows the table, the bottom part the buttons for manipulating the table entries.

The table column headers show the abbreviated names of the editors (the tool tip of each header will show the full name). Each column represents the priorities that the different schedules have for this particular editor. When the editor is active in the workbench view the list of matching schedules will be sorted in the order of these priorities. The user can directly edit these properties in the table. For easier readability the best schedule for each editor is marked with a dot and the table is sortable by clicking any of the column headers.

The editors and schedules that have a padlock next to their name are the ones imported through the default schedule extension point described in Section 6.2.3. These editors and schedules are not supposed to be edited or removed since they represent a factory default setting. The view realizes this by graying out the corresponding buttons when an imported schedule is selected.

All other schedules can be removed by simply clicking the appropriate button. The button for editing the location of a schedule opens a dialog showing the currently active workspace and allows the user to select a new execution file that should be associated with the selected schedule. This feature is necessary in case the user moves a schedule through his file browser instead of the refactoring facilities on the workbench.

The editor marked with the dot is the default editor (see Section 6.2.4.



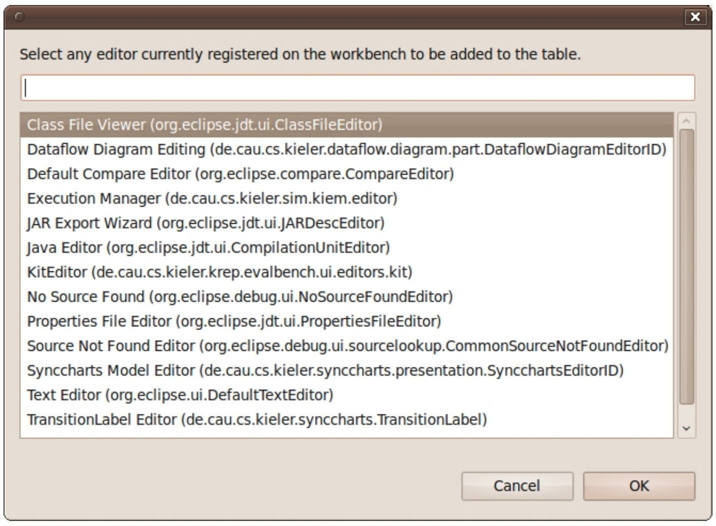
Scheduling Configurations

This page holds information about the editors, schedules and their priorities.
The editor marked in the column header is the default editor which will be used for schedule matching when no other editor is open.
The marked schedule is the best schedule for the given editor.

Scheduling Priorities

Schedule	● SDE	🔒 EE	🔒 KE
autoTest2[test]		● 5	
autoTest[test]		5	
example[de.cau.cs.kieler.sim.kiem]	● 5		
🔒 krep[IMPORTED]			● 8
krep[ctrTest]		5	
🔒 noname1[IMPORTED]		3	7
noname2[test]	2		
noname[de.cau.cs.kieler.sim.kiem]	3		

Figure 6.4: The page for managing the schedules and editors


Select any editor currently registered on the workbench to be added to the table.

```
Class File Viewer (org.eclipse.jdt.ui.ClassFileEditor)
Dataflow Diagram Editing (de.cau.cs.kieler.dataflow.diagram.part.DataflowDiagramEditorID)
Default Compare Editor (org.eclipse.compare.CompareEditor)
Execution Manager (de.cau.cs.kieler.sim.kiem.editor)
JAR Export Wizard (org.eclipse.jdt.ui.JARDescEditor)
Java Editor (org.eclipse.jdt.ui.CompilationUnitEditor)
KitEditor (de.cau.cs.kieler.krep.evalbench.ui.editors.kit)
No Source Found (org.eclipse.debug.ui.NoSourceFoundEditor)
Properties File Editor (org.eclipse.jdt.ui.PropertiesFileEditor)
Source Not Found Editor (org.eclipse.debug.ui.sourcelookup.CommonSourceNotFoundEditor)
Synccharts Model Editor (de.cau.cs.kieler.synccharts.presentation.SyncchartsEditorID)
Text Editor (org.eclipse.ui.DefaultTextEditor)
TransitionLabel Editor (de.cau.cs.kieler.synccharts.TransitionLabel)
```

Figure 6.5: The Editor Selection Dialog

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 6.5 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 unnecessarily large 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.

6.3.3 Schedule Selector

The Schedule Selector is the only view element isn't shown on the preference pages. It is used to construct and manage the two ComboBoxes for loading schedules that are known to the KIEMConfig. Each of the two ComboBoxes seen in Figure 6.6 has its own task.

The Matching Selector : This ComboBox displays the schedules that can be used with the currently active editor. The list is sorted by the priorities assigned

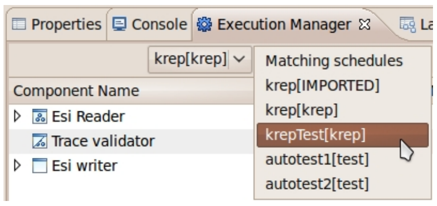


Figure 6.6: The Schedule Selection ComboBoxes

through the preference page. It displays a header showing its purpose in order to distinguish it from the other ComboBox.

The Recently Used Selector : This ComboBox displays the list of recently used schedules. The list is ordered in the same order that the schedules have been accessed through the Execution Manager. The maximum size of the list can be configured through the preference page. The ComboBox displays the name of the currently loaded execution file regardless of how it was loaded. This was done because up till now there actually was no way to tell which execution file was loaded in the Execution Manager.

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

The first task was to implement a way for execution files to carry configuration properties. This was solved by creating a new type of DataComponent that stores all properties and is accessed by the KIEMConfig when values for the properties are needed. The newly created component is automatically saved with any execution file that wants to use the new feature.

Part of the first task involved enabling the user to manage a default configuration for the different properties. This was realized through the creation of a set of preference pages. These pages allow the user to manage the properties and even override the ones stored in the execution file.

The last objective concerned the actual loading of the execution files. A way had to be found in order to make loading the files easier and provide filtered perspectives on all known files. This was accomplished by providing the user with two ComboBoxes that contained a subset of all known execution files. One ComboBox is used to load the most recently used files while the user contains files that probably work for the currently active editor.

7.2 Future Work

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.

7.2.1 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.

7.2.2 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 *Serialization* interface to be stored as well.

7.2.3 Use Advanced KiemPropertyTypes

Currently the KIEMConfig can only handle KiemProperties of the basic type that contains a String value. However there is a whole range of different KiemPropertyTypes such as integer, files or choice. An enhancement to the KIEMConfig would be to utilize the full potential of these types.

This would mean rewriting much of the already existing code and completely refining the page where the user can define their properties.

7.3 Summary

All in all the initial problem of providing a means to storing additional configuration information in an execution file as well as providing a mechanism for providing a default configuration was solved. In addition to that the second objective which involved finding a way to make it easier to load previously saved schedules was also completed.

However there is still a lot of basic for further work on the concepts of this project and on the concept of the Execution Manager in general.

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.

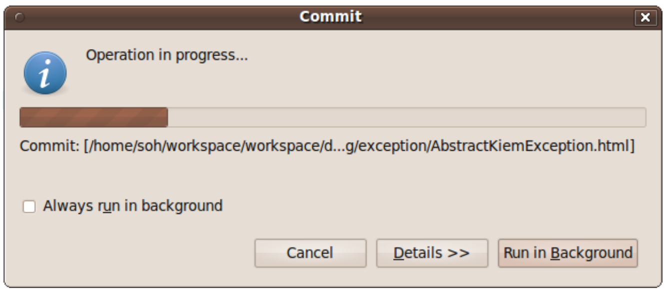
The next sections will describe the technologies and give some examples of their usage in the standard Eclipse application. The technologies described here are the following:

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

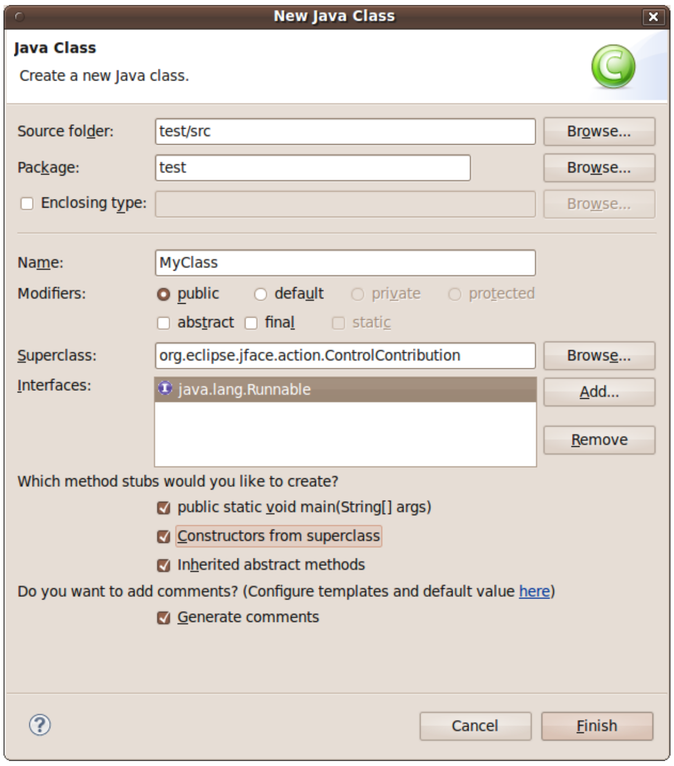
The Eclipse Job API provides the means to schedule very long running tasks. It uses a Thread to run the actual task and contains a ProgressMonitor to show the progress of the task. Since it is a task that can run independently of the current state of the workspace it can also be run in the back ground if the user desires it. An example for the use of jobs in the normal Eclipse architecture is the Subversion (SVN) commit operation seen in Figure 8.1. An SVN commit involves sending a possibly large amount of files to a remote location over a network connection. That operation might take a very long time. Furthermore there is no reason to prevent the user from continueing to work while the files are send which means that the job can be run in the back ground.



Operation in progress...

Commit: [/home/soh/workspace/workspace/d...g/exception/AbstractKiemException.html]

Figure 8.1: The SVN commit job


New Java Class

Java Class

Create a new Java class.

Source folder: test/src

Package: test

☐ Enclosing type:

Name: MyClass

Modifiers: [✓ public] default private protected

☐ abstract ☐ final ☐ static

Superclass: org.eclipse.jface.action.ControlContribution

Interfaces: java.lang.Runnable

Which method stubs would you like to create?

☑ public static void main(String[] args)

☑ Constructors from superclass

☑ Inherited abstract methods

Do you want to add comments? (Configure templates and default value here)

☑ Generate comments

Figure 8.2: The Class Creation Wizard

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. A wizard is basically a multi-page dialog with each page representing one step in the creation of the desired item.

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.

```java
package test;

import org.eclipse.jface.action.ControlContribution;
import org.eclipse.swt.widgets.Composite;
import org.eclipse.swt.widgets.Control;

public class MyClass extends ControlContribution
   implements Runnable {

   /**
    * Creates a new MyClass.java.
    *
    * @param id
    */
   public MyClass(String id) {
      super(id);
   }

   /**
    * {@inheritDoc}
    */
   @Override
   protected Control createControl(Composite parent) {
      return null;
   }

   /**
    * {@inheritDoc}
    */
   public void run() {
   }

   /**
    * @param args
    */
   public static void main(String[] args) {
   }
}
```

Listing 8.1: Code generated by the wizard

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.

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

This chapter presents the conceptual solution to the problems described in Chapter 9.

It will follow the same structure to make it easier to follow. This means that the chapter will start by offering a few possible options for the user to set up an automated execution. It will continue by providing alternatives of how DataComponents receive the necessary information prior to each run. The next section explains how the actual control flow throughout the automation will be handled. In the last section some possibilities for user-friendly output are presented.

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 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 2.2.2) 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 2.2.4) 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

As a basic control flow for the automation the following procedure is chosen:

1. The automation will iterate over all supplied execution files. These are likely the most time consuming to load which means loading an execution file multiple times should be avoided.

2. With each execution file the automation will iterate over all model files. Model files are costly to load as well however each load of an execution file would mean that all DataComponents would not be able to store their saved properties either way which in turn means they would have to reload the model anyway. For this reason the model files will be loaded once for every execution file.

3. With each model file the automation will perform multiple runs. That means starting a run in the execution manager and performing a certain number of steps and the stopping the execution again. With each run the components get the chance to execute a few steps with different properties, for example trace files. These runs will be called iterations from this point forward.

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. As described in Section 2.1 a view is used to display content that was created elsewhere. Therefore it is the logical choice for displaying the information created by the automated execution. 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 the Execution Manager

In order for the KIEMAuto to operate successfully there were no changes necessary beyond the ones described in Chapter 5. However the newly created plug-in uses many of these features and could not achieve its objectives without them.

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. This involves receiving information about the completion of a step or user- and error-triggered termination or the currently active execution.

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

This chapter describes the Automated Execution plug-in that is responsible for handling the setup, control flow and display of an automated execution run. Although the plug-in is structured according to the model-view-controller pattern this is not the structure that this chapter will to describe it. Instead this chapter will follow the structure already used in the previous chapters. Which means the chapter will consist of four parts:

1. The setup of an automated execution run with a description of the wizard.
2. The input to the automated execution. This section will describe the newly created interface.
3. The control flow of the automation with the Automation Job and the Automation Wizard.
4. The output of the automation run. In this section the new view and the results structure will be described.

12.1 Automation Setup - The Wizard

As described in Section 10.1 an Eclipse wizard will be used to set up the automated run. For easy access the button for opening the wizard ( ) is located on the tool bars of both the Execution Manager and the Automated Execution view.

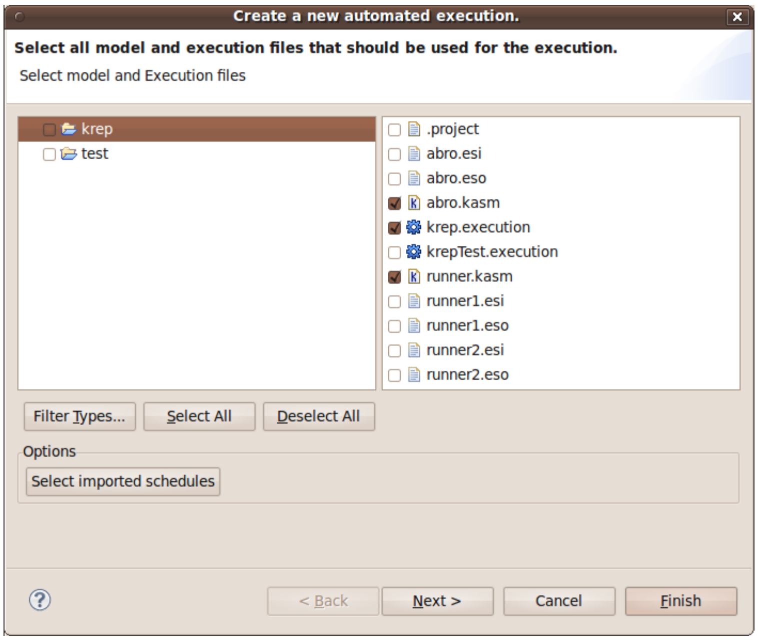
The Automation Wizard consists of two pages:

1. The File Selection Page for selecting all files that should be involved in the automated run.
2. The Property Setting Page for defining custom properties that the components should receive prior to each iteration.

12.1.1 File Selection Page

The File Selection Page shown in Figure 12.1 is used to select the model files and execution files that should be used for the automated run.

Since Eclipse already provides a variety of pre-made wizard pages it can be avoided to write a page for a complex task like this from scratch. The wizard that can be modified to fit the needs of the task at hand is the standard Eclipse Resource Import Wizard. It is normally used to select a number of files and folders for import into



Create a new automated execution.

Select all model and execution files that should be used for the execution.

Select model and Execution files

- ☐ krep
- ☐ test

- ☐ .project
- ☐ abro.esi
- ☐ abro.eso
- ☑ abro.kasm
- ☑ krep.execution
- ☐ krepTest.execution
- ☑ runner.kasm
- ☐ runner1.esi
- ☐ runner1.eso
- ☐ runner2.esi
- ☐ runner2.eso

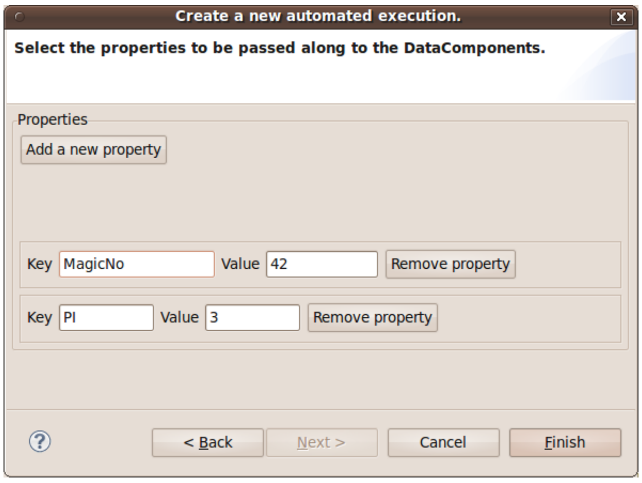
Figure 12.1: The Wizard Page for selecting the input files for an automated run.

the workspace. It provides a structured view where entire folders can be selected, files can be filtered by their type and additional space is available for other buttons. In this project the files will be 'imported' into the automated execution. This is similar enough to make it possible to use the wizard with a few modifications and extensions.

As it should be as easy as possible to set up a run for the user it would be desirable that he doesn't need to select all the files each time the wizard is opened. For this reason the selection will be saved into the Eclipse preference store every time the wizard is closed. The next time the wizard is opened the selection only has to be retrieved from the preference store and passed to the Resource Import Wizard super class.

The KIEMConfig allows for execution files to be linked into the workspace through an extension point. This is a useful feature for adding factory defaults and as such KIEMAuto naturally wants to be able to use these execution files as well. However since they these files are not in the workspace they can't be selected through the main area of the wizard page. In order to select these files anyway a simple list selection dialog can be accessed through the button at the bottom of the page. The dialog displays all schedules imported through the extension point and allows the user to select any number of them.

The hard part is how to figure out if the user has selected valid files for an automated run. Recognizing selected execution files can simply be done by looking at the file extension. However determining whether the user selected valid model files that will work with the selected execution files is somewhat difficult. One possibility


Create a new automated execution.

Select the properties to be passed along to the DataComponents.

Properties

Add a new property

Key MagicNo Value 42 Remove property

Key PI Value 3 Remove property

< Back Next > Cancel Finish

Figure 12.2: The Wizard Page for setting up user defined properties.

would be to use the priority system in the KIEMConfig in order to determine the validity of the combinations. However this would assume that all selected execution files are known to the plug-in and that the user set priorities for each of them. At this point it is simply assumed that all selected files that have an extension other than 'execution' are model files. Precautions to avoid running invalid combinations of model files and execution files are described in Section 12.2.

The dialog will only allow the user to proceed if at least one execution file or imported schedule and one model file is selected. Otherwise an error marker will be displayed in the page header.

12.1.2 Property Setting Page

The second wizard page shown in Figure 12.2 allows the user to enter some custom properties for the automated run. Unlike the first page this one doesn't extend a particular wizard page but rather the generic wizard page that only supplies the header and the button bar at the bottom of the page.

The user can add an arbitrary number of panels for adding new (key, value) pairs through the button at the top of the page. These values will be added to the list that is passed to all DataComponents before each iteration and the values can be retrieved by looking for the key.

Since these properties are completely optional there are requirements for finishing this page. Which means that due to the wizard nature of the dialog the user doesn't even have to look at that page at all but can finish directly from the file selection page.

As with the file selection page the user input will also be stored as soon as the wizard closes and the properties restored on the next opening of the wizard. The upside of which is that the user saves considerable work by only having to enter the properties once. However the downside is that the user might not realize that the properties are still set if he finishes the wizard directly from the first page. The only way to avoid this situation would be to force the user to look at the second page. This is however very undesirable since the second page is likely to be used only by advanced users anyway and the average user wouldn't want to see it.

12.1.3 Information Processing

After the user is finished with the wizard all information has to be collected in order to set up the automated run. This involves the following steps:

1. Gather the model files selected on the file selection page.
2. Get the paths for the execution files. This might involve retrieving the paths from the schedules imported through the extension point if the user selected any of those.
3. Get the list of (key, value) pairs that should be added to the automated run from the second wizard page.
4. Invoke the Automation Manager described in Section 12.3 with the gathered parameters.

12.2 Automation Input

In order to provide the DataComponents with input prior to each part of the automated run new interfaces had to be created in order to interact with the components.

12.2.1 Automated Component

An automated component is any DataComponent (see Section 2.2.2) that wants to interact with the automated execution plug-in. An automated component has to provide the following methods:

String[] getSupportedExtensions() :
: This method is used in order to avoid the automated run encountering errors while trying to simulate invalid combinations of model files and execution files. As soon as any execution file is loaded the method will be called on each of the implementing classes. The classes should answer with a list of file extensions of the model files that they can simulate. Model files that no component in the currently active execution file can simulate will be skipped.

void setParameters(List<KiemProperty> properties) throws KiemInitializationException :
: 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. 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. The custom properties that the user defined through the wizard for example are also added here. If the component encounters an error during at this point because for example a model file could not be loaded it should respond by throwing the declared Exception.

int wantsMoreSteps() :

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.

int wantsMoreRuns() :

This method works analog to the wantsMoreSteps() 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.

12.2.2 Automated Producer

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

List<KiemProperty> produceInformation() :

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 Section 12.4).

The component is free to publish any amount of information using any of the different KiemProperty types. However the number and order of the properties in the list has to stay the same through all iterations with one execution file. This restriction is necessary in order to be able to build one large table rather than many small ones.

12.3 The Automated Run

The Automation Manager is the key part of the automated execution. It manages the entire control flow through the automated execution and its public methods are part of the plug-ins API. The reason for this is that the automated run can be initiated without the use of the wizard by any other plug-in.

The next sections will give a detailed description of the control flow during the automated execution. The API methods to initiate a new automated execution are located inside the Automation Manager. However since those methods immediately create a new Automation Job and schedule it right away the first section will explain the Automation Job.

Section 12.3.2 will then proceed to describe the entire control flow that is managed by the Automation Manager.

In Section 12.3.3 the Cancel Manager will be described which is responsible for triggering a premature termination of the entire automated execution or parts of it.

The last section will describe the modifications to the ErrorHandler in order to ensure a smooth run of the automation.

12.3.1 Automation Job

The Automation Job is used to run the automated execution in. As a WorkbenchJob it can execute parallel to the normal operation of the GUI without blocking it. It also comes with a progress monitor that is updated by the Automation Manager through the course of the automated execution. Since an automated run can take a very long time the user can also tell the job to run in the background while still being able to get feedback about it through Eclipse's progress view.

Upon creation the Automation Job takes all the parameters necessary for the automated execution. After that it opens Execution Manager view in order to load all necessary plug-ins before the actual run starts. The job then creates a new thread and initiates the automated execution inside the Automation Manager. At the end it tells the calling thread that it's returning asynchronously in order not to block any callers.

The dialog showing the progress monitor is not only used to get feedback about the progress of the task it can also be used to cancel the execution prematurely for example if the user realizes that he selected the wrong files.

12.3.2 Automation Manager

The Automation Manager is responsible for handling the entire control flow during the automated execution. It takes the execution files, model files and other properties as arguments and organizes the entire run based on the available information. During the run the Automation Manager also collects the information that will be displayed as results by the view.

1. The Automation Manager starts by initializing the progress monitor that displays the progress of the automated execution. The displayed message of the monitor is set up and the total length of the execution is calculated in order for the progress bar to be displayed.

Since a priori only the number of execution files and model files is known these are the only variables that can be used. If more than one execution file and one model file are used the progress monitor will display a progress bar that advances with each completed model file. However this method would be meaningless if only one execution file and one model file are used. In this case the progress bar will just show that progress is occurring at all.

2. In this step the Automation Manager stores the path of the execution file that is currently opened inside the Execution Manager. The reason for this is that the user might want to continue working on that file after the automated execution has finished. If the file wasn't stored he would be left with the last execution file that the automated run simulates. The file is restored in Step 20.

3. The next step is to initiate the Cancel Manager described in Section 12.3.3. This also involves registering as listener on the modified ErrorHandler (see Section 12.3.4) to prevent any Exception from interfering with the automated run.

4. Since all parameters for the automated execution have been set up now the Automation Manager can start working on the first execution file. The first thing to do here is to open the execution file inside the Execution Manager. If this fails the automation writes an entry into the error log and proceeds to the next execution file. Otherwise it proceeds to the next step. If all execution files have finished the automation will skip to Step 19.

5. The next step is to initiate all variables that are needed for the currently active execution file. The manager first asks the view to create a new table to display the results that the execution will produce. After that all components will be asked for the list of model file extensions that they support (see Section 12.2.1).

6. With the information about the supported model file extensions gathered in the previous step the automation can start to examine the first model file. If any component in the current execution file supports the given model file the automation proceeds to the next step. Otherwise it skips to the next model file. If all model files on the current execution file have finished it will proceed to Step 4 with the next execution file.

7. The first thing that happens when the Automated Manager decides that a model file should be simulated is that all Automated Components will be notified with the list of parameters.

   This method iterates over all components and provides them with the list of user defined properties that were passed to the automated execution. The list also includes the path of the current model file and the index of the current iteration. Some components may choose to write information back into the list in order to communicate with other components. Therefore the list will be passed to all components twice if the size changed between after the first notification.

8. Since the components now have all the necessary information for the first iteration they might already be able to approximate how many iterations they want to execute. Therefore all components will be asked through the methods provided in the interface and the maximum value used.

   At this point all components may decide that they don't need any runs at all. The automation will then add a new result to the panel displaying that fact. After that the automation will continue on Step 6 with the next model file.

   If any component answers with a non-zero value the automation will move to the next step and perform at least that many iterations, baring user cancellation of course.

9. Since set up of the model file has now finished the automation now checks if an iteration should be executed. It first checks whether the number of remaining iterations has reached zero in which case no component requested another run. It then checks whether the current model file was skipped through the Cancel Manager (see Section 12.3.3). If both checks turn out to be negative it proceeds to the next step after decrementing the amount of remaining iterations. If one of the checks hold true it continues on Step 6 with the next model file.

10. In this step the Automation Manager sets up all parameters that are needed for the current iteration.

    It starts by updating the text in the progress monitor in order to reflect the current status of the automation. The displayed text shows the current model file and execution file, the iteration index and how many iterations are to be expected based on the value retrieved from the components.

    After that the manager adds a new row to the resulting table inside the view (see Section 12.4). This has to be done at this stage in order to show the status of the currently executing iteration before it has finished.

    Before the execution can proceed the components first have to be passed the list of parameters including the current iteration index. This process is performed by the same method as described in Step 7.

    After all the preparations have been completed the automation proceeds to the next step.

11. The next step is to access the Execution Manager itself in order to initialize the execution. The Execution Manager will set up all DataComponents and create the thread in which the execution is run.

    If an error occurred at this stage the automation will skip to Step 16. An error can be caused by any of the DataComponents during their initialization phase or by the Execution Manager itself if the execution file can be loaded but not executed.

If the initialization was successful the Automation Manager continue to the next step.

12. Before the automated execution can begin to step through the execution it has to ask the components if any steps should be performed. Due to some results from previous iterations the components might decide that they don't need to simulate a particular trace file, for example. In this case they need an opportunity to notify the Automation Manager of these circumstances which will be done in this step.

    The Automation Manager will ask each Automated Component how many steps they expect to need for their execution (see Section 12.2.1). The maximum of all components will be taken and at least that many steps will be performed before asking again.

13. With the information of the previous step the Automated Manager will determine whether or not the execution should resume. It first checks if the number of requested steps has reached zero. After that it checks if the Cancel Manager requests a cancellation of the current iteration. If one of these conditions hold true the automation will skip to Step 16. Otherwise it will continue to the next step.

14. The Automation is now ready to perform a step in the Execution Manager. However it first resets the timeout in the monitoring thread inside the Cancel Manager. The thread will abort the current step if the timeout is reached.

    After that the Automation Manager tells the Execution Manager to perform a step. Since that method returns asynchronously in order not to block any callers. The Automation Manager will lock itself inside a Semaphore after the asynchronous return.

    It will wait inside the Semaphore until one of the following events occur:

    - The Cancel Manager determines that a timeout is reached, an error occurred or the user cancels the current iteration. In this case the automation will skip to Step 16.
    - The Execution Manager has finished processing the step and notified the Automation Manager through the extension point described in Section 5.1.3. In this case the automation proceeds to the next step.

15. After the step was successfully executed the Automation Manager has to check if more steps should be performed. If at this stage the number of remaining steps has reached zero and the iteration was not canceled by the Cancel Manager the Automated Manager will ask all Automated Components how many more steps they need to perform (see Step 12).

If more steps should be performed the Automation Manager will proceed to Step 13. Otherwise the current iteration will be paused and wrapped up in the next step.

16. This phase will be executed if the iteration was completed successfully or an error occurred. In any case some wrap-up has to be performed in order to allow the next iteration to be performed or to tidily shut down the entire execution.

    The cleanup will start by terminating the thread that watches for timeouts during the currently executing step and user cancellations. This has to be done because the wrap-up code should not be interrupted.

    After that all Automated Producers (see Section 12.2.2) will be asked to publish their results. The accumulated results together with the index of the last step will be forwarded to the view in order to update the table (see Section 12.4).

    As soon as the results were retrieved from all components the current execution inside the Execution Manager is no longer needed. The Automation Manager triggers the synchronous stop of the paused execution. This will cause the Execution Manager to terminate all worker threads and get ready for the next execution that has to be run.

17. If the iteration completed without an error the status will be set to “Done”. If the counter for the remaining iterations has reached zero the components will be asked if they want to perform more iterations by the same method as in Step 8.

    If more iterations should be performed the iteration index is incremented and the automation proceeds to Step 9. Otherwise the Automation Manager proceeds to finishing the current model file in the next step.

18. Since all iterations for the current model file have been finished the automation will perform the progress monitor of that fact which will cause a progress bar to advance.

    If there are more model files to simulate under the current execution file the automation will return to Step 6.

    item At this point the automation has finished with the last model file under the current execution file. If there are more execution files to be loaded the Automated Manager will return to Step 4.

19. Since the entire automated execution has now finished another wrap-up stage has to be initiated.

    First the Automation Manager will remove the listener on the ErrorHandler that was added on Step 3. Since the automation has finished there is no further need to block pop-up messages and unnecessarily interfering with the error messages of other plug-ins should be avoided.

After that the progress monitor will be informed that the job was finished. This will cause the dialog to disappear.

20. The last thing to do before the Automation Job can terminate is that the file stored in Step 2 is opened in the Execution Manager. This is done in order to allow the user to continue his previous work.

After the Automation Manager has finished the entire automated execution the user can look at the complete results in the view or export them into an external format (see Section 12.4).

12.3.3 Cancel Manager

The Cancel Manager is responsible for initiating a premature termination of any part of the automated execution. There can be multiple reasons for such a process to be necessary. For example, the user is monitoring the automation himself and decides that he wants to skip a part of it or cancel the entire operation. Another reason would be that an error occurred or a timeout was exceeded. These cases can't be detected by the Automation Manager itself. Therefore the Cancel Manager launches another thread that keeps checking for any of the cancellation criteria to hold. However the manager will not perform a hard cancellation of the Automation Manager's thread. This option was not chosen because no cleanup could be performed and it would make skipping only certain parts impossible.

Skipping only certain parts of the entire automation is necessary in case an error occurs or the user wants to skip a certain model file, for example, because it was added by mistake. The Cancel Manager provides the following cancellation option which are supported by actions on the tool bar (see Section 12.4.1):

1. **Skip iteration** : During an automated run the user might realize that an iteration has somehow locked up or isn't aborting because the components keep requesting more steps. This option performs a deferred termination of the current iteration and proceed to the next index.

2. **Skip model file** : This option will skip to the next model file after aborting the current iteration. The reason for the user wanting to cancel the current model file might be that the components keep requesting additional runs indefinitely. Another reason would be that the model file keeps producing faulty results due to the trace files missing and the user doesn't want to wait for it to fail on the remaining files.

3. **Skip execution file** : Sometimes the user may even want to skip an entire execution file if at some point it can be expected that it won't produce any useful results.

4. **Cancel automated execution** : The option initiates a deferred termination of the entire execution. This has the same functionality as pressing "Cancel" inside the progress monitor dialog.

12.3.4 Modified Error Handler

One of the problems in the early stages of development was that an automated run wouldn't complete because of errors in some of the DataComponents.

This was caused by the fact that the Automation Manager operates only indirectly on the execution that is running inside the Execution Manager and thus will not receive any thrown Exceptions. Any uncaught Exception in the Execution Manager itself or in any of the DataComponents will cause the ErrorHandler of the current Eclipse application to be invoked.

The ErrorHandler is responsible for dealing with all errors that may have to be brought to the users attention. It contains facilities that will allow any plug-in to dispatch an error with a predefined option of how to handle it. These options include showing the error in the error log view, opening a dialog or opening a dialog and block the GUI until the user acknowledges the error. During a very long running automated run the last option is very undesirable as it means that the automation suddenly requires user interaction which of course defeats the whole point. This is even more annoying for the user since the majority of Exception during an automated run that cause the GUI to block are not critical Exceptions.

For example, one of the DataComponents is analyzing a model file with a given set of trace files. For some reason one of the trace files is missing which causes an Exception to be thrown inside the component which throws it to the Execution Manager that called the component. The Execution Manager doesn't deal with the RuntimeException and throws it up to the Eclipse GUI which responds by invoking the ErrorHandler and blocking the automated execution from continuing.

If the Exception could have been caught in the Automation Manager it would simply mean that the iteration with that particular trace file would have to be skipped and the next trace file should be used. The user then could have received the error at the end of the run or look it up in the error log.

In order to remedy that situation the default ErrorHandler used by Eclipse is replaced with a different one. The modified ErrorHandler allows listeners that implement the interface shown in Listing 12.1 to register to it. Whenever a plug-in asks the ErrorHandler to deal with an error it will first notify all listeners of the error that occurred. The listeners then can decide if they want to modify the status of the given error. The ErrorHandler accumulates all requests from the listeners and computes the new status. If no listeners are registered or all listeners answered with “Don't care” the error will be handled with its old status. If the listeners did request a status change all requested status changes will be applied. This means if one listener asks to only log the error while another listener requests a pop-up the pop-up dialog is shown and the error is logged.

However there are some errors that will be immediately handled without asking the listeners first. Those are fatal errors that will likely cause the entire application to enter an undefined state and where the best course of action usually is to shut down the application entirely.

In the context of the automated execution the error handler will be asked to only

```java
public interface StatusListener {

    /**
    * Indicates that the component doesn't care about the style of the
    * error.
    */
    int DONT_CARE = -1;

    /**
    * Reroute the exception to the given listener. If the listener wants to
    * modify the style it should return the style that it wants the
    * exception to have. If the component doesn't care about that
    * particular exception it should return the
    * {@link StatusListener#DONT_CARE} style.
    *
    * @param statusAdapter
    *            the status adapter
    * @param style
    *            the style
    * @return the style that the status should have
    */
    int reroute(StatusAdapter statusAdapter, int style);
}
```

Listing 12.1: The interface for listeners on the ErrorHandler.

log the errors while the automated run is in progress.

12.4 Automation View

The Automation View seen in Figure 12.3 is used to display the results of an automated execution is a structured way.

However it would be very undesirable if the user had to wait for the entire automated run to finish before viewing any information. Therefore the view displays all information that is currently available.

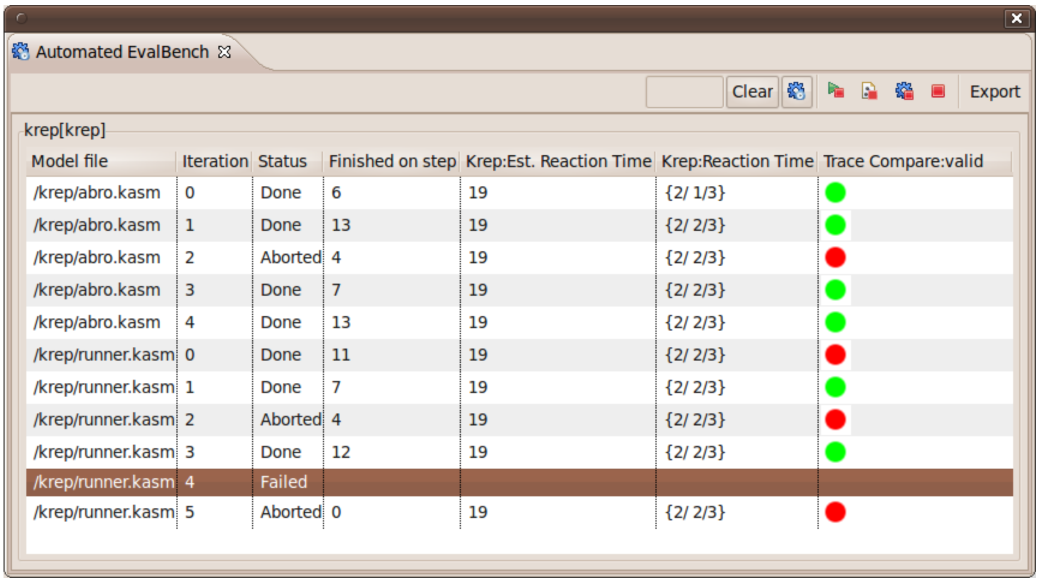
The information is gathered from the output of the Automated Producers and also includes status information produced by the automation itself. The first four columns contain this status information:

1. The first column shows the name of the model file that was simulated. This might be the same for multiple subsequent rows as a model file might be simulated with more than one iteration.

2. The next column shows the index of the iteration that produced the given result.

3. The third column displays the status that the execution that produced the result is currently in:

    - **Created** : The current iteration has been created and is preparing to execute. It is for example waiting for trace files to be opened or preliminary



krep[krep]

Model file	Iteration	Status	Finished on step	Krep:Est. Reaction Time	Krep:Reaction Time	Trace Compare:valid
/krep/abro.kasm	0	Done	6	19	{2/ 1/3}	🟢
/krep/abro.kasm	1	Done	13	19	{2/ 2/3}	🟢
/krep/abro.kasm	2	Aborted	4	19	{2/ 2/3}	🔴
/krep/abro.kasm	3	Done	7	19	{2/ 2/3}	🟢
/krep/abro.kasm	4	Done	13	19	{2/ 2/3}	🟢
/krep/runner.kasm	0	Done	11	19	{2/ 2/3}	🔴
/krep/runner.kasm	1	Done	7	19	{2/ 2/3}	🟢
/krep/runner.kasm	2	Aborted	4	19	{2/ 2/3}	🔴
/krep/runner.kasm	3	Done	12	19	{2/ 2/3}	🟢
/krep/runner.kasm	4	Failed				
/krep/runner.kasm	5	Aborted	0	19	{2/ 2/3}	🔴

Figure 12.3: Automation View showing the result of an automated execution.

calculations.

- **Running** : The iteration is execution in the Execution Manager and performing the desired number of steps.
- **Done** : The iteration completed successfully.
- **Aborted** : The iteration was aborted by the user or another component.
- **Failed** : The iteration failed to complete because of an error.

4. The last column contains the step that the execution finished on.

The following columns contain the information gathered from the Automated Producers. In order to make the information easier to understand the column header not only displays the name of the attribute but also the name of the component that produced the given value.

If more than one execution file is simulated a new table will be created below the previous tables. This is necessary because another execution file might contain different components which produce different lists of results. These results would not fit into the table of the previous executions. Another possibility would be to add additional columns to the existing table. However this could increase the width of the table too much for the user to still effectively read. Furthermore if the two execution files don't share any components a shared table would look more confusing than helpful.

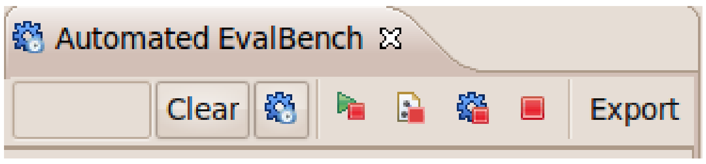


Figure 12.4: The tool bar in the automation view.

12.4.1 Tool bar

The tool bar on the Automation View contains several actions for controlling the automated execution before, during and after its run (see Figure 12.4). The different controls have the following functionality (left-to-right):

1. **Current Step Field** : During the automated run this field displays the currently executing step. It is basically the same control as on the tool bar of the Execution Manager itself. The control was duplicated in this place in order to avoid having to switch between the two views. This means that information about the currently running execution is displayed in the Automation View.

2. **Clear** : When the user initiates multiple automated runs after one another the results are all displayed in the same view. This is the intended behavior as it should give the user the ability to compare automated runs with different inputs. However if the view gets filled with too many results the user needs an easy way to clear the view which is realized through this button.

3. **Automation Wizard** : The next button is used in order to launch the Automation Wizard. The button exists both here and on the Execution Manager's tool bar in order to allow easy access to the automation.

4. The next four buttons are used for performing one of the skipping actions supported by the Cancel Manager (see Section 12.3.3):

   Perform a deferred termination of the current iteration and proceed to the next index.

   Abort the current iteration and all subsequent ones on the current model file skipping to the next model file.

   Skip to the next execution file.

   🟥 Initiate a deferred termination of the entire execution. This has the same functionality as pressing "Cancel" inside the progress monitor dialog.

5. **Export** : The export button is used for opening a dialog to export the currently displayed results to an external format. This feature is explained in Section 12.4.2.

```
krep[krep]
"Model file","Iteration","Status","Finished on step","Krep:Est. Reaction Time","
    Krep:Reaction Time","Trace Compare:valid"
"/krep/abro.kasm","0","Done","6","19","{2/ 1/3}","true"
"/krep/abro.kasm","1","Done","13","19","{2/ 2/3}","true"
"/krep/abro.kasm","2","Aborted","4","19","{2/ 2/3}","false"
"/krep/abro.kasm","3","Done","7","19","{2/ 2/3}","true"
"/krep/abro.kasm","4","Done","13","19","{2/ 2/3}","true"
"/krep/runner.kasm","0","Done","11","19","{2/ 2/3}","false"
"/krep/runner.kasm","1","Done","7","19","{2/ 2/3}","true"
"/krep/runner.kasm","2","Aborted","4","19","{2/ 2/3}","false"
"/krep/runner.kasm","3","Done","12","19","{2/ 2/3}","true"
"/krep/runner.kasm","4","Failed","","","",""
"/krep/runner.kasm","5","Aborted","0","19","{2/ 2/3}","false"
```

Listing 12.2: Example of a table exported to CSV.

12.4.2 Exporting Results

As described in Section 12.4 a number of tables is used to display the results in the view. However these tables are not persistent i.e. they are removed when the program is closed. In order to keep the results for analysis a method had to be found to export them into an external format.

The button on the tool bar first opens a dialog that shows all available types that the results can be exported to. The next window prompts the user to enter file names for the exported files. Since the exported files are supposed to be used in other applications as well the standard OS file chooser is used instead of the workspace file chooser.

To illustrate the results of such a transformation the table shown in Figure 12.3 is transformed in both formats. The resulting Comma-Separated Values (CSV) file can be seen in Listing 12.2. The same results exported to LaTeXcan be viewed in Table 12.1.

				Krep		Trace Compare
Model file	Iteration	Status	Finished on step	Est. Reaction Time	Reaction Time	valid
/krep/abro.kasm	1	Done	13	19	2/ 2/3	true
	2	Aborted	4	19	2/ 2/3	false
	3	Done	7	19	2/ 2/3	true
	4	Done	13	19	2/ 2/3	true
/krep/runner.kasm	0	Done	11	19	2/ 2/3	false
	1	Done	7	19	2/ 2/3	true
	2	Aborted	4	19	2/ 2/3	false
	3	Done	12	19	2/ 2/3	true
	4	Failed				
	5	Aborted	0	19	2/ 2/3	false

Table 12.1: Example of a table exported to LaTeX.

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

The first objective of the project was to solve the set up of an automated run by the user. The initial idea of using a script-based approach and providing the necessary files as lists in text files was not pursued. Instead a more user-friendly solution was found through the use of a wizard.

The next objective was to find a way to input information into the DataComponents in order for them to be designed in a more generic way. This was achieved by designing new interfaces that allow components to receive a list of properties prior to each part of the automated execution.

The main part of the task involved creating a manager that guided the control flow of an automated execution. The Automation Manager described in Section 12.3.2 loads all the necessary files, sets up the different outputs, steps through the execution to the desired step and includes error management facilities as well.

The last part was to find a user-friendly way to display the information generated by the automated execution. The problem was solved by creating a view that displayed the information in a set of tables. These tables are structured in a way to easily compare the results of different model files and iterations. To allow use of the generated tables in another context methods were implemented in order to allow the generation of external formats (namely CSV and $\LaTeX$).

13.2 Future Work

Although all initial objectives were achieved there is still room for additional improvements. These improvements which could be the subject of further study will be explained in this section.

13.2.1 Scripting

Currently the automated execution can be triggered through the use of a wizard and will run inside the Eclipse workbench and display the results in a graphical view.

In order to allow other plug-ins or external applications to trigger an automated execution an interface would have to be defined. This could involve the creation of a scripting language that passes all the needed parameters to the automated execution. It would also involve retrieving those results and importing them into the calling application.

The existing code already supports most of the requested features however there is no interface to access those features from outside of Java.

13.2.2 Exports

Currently the only possibility to use the collected data in another application is by exporting the entire table into CSV or LaTeX.

This process could be improved, for example by allowing the user to select the table columns and rows that he wants to export instead of exporting the entire table. Furthermore export to additional formats could be implemented. It could even be possible to interface the entire automation with a database application or remote system.

13.3 Summary

All in all the initial problem of providing a framework for setting up automated execution runs was solved.

Further testing and use of these features may however reveal new ways that the plug-in can extend its functionality.

Appendix

```java
/**
 * Builds the local tool bar for the KiemView part.
 */
private void buildLocalToolBar() {
    IActionBars bars = getViewSite().getActionBars();
    IToolBarManager manager = bars.getToolBarManager();
    // first remove all entries
    manager.removeAll();

    // call soh's extension point
    addExternalContributions(manager);

    manager.add(getActionUp());
    manager.add(getActionDown());
    manager.add(new Separator());
    manager.add(getAimedStepDurationTextField());
    manager.add(getStepTextField());
    manager.add(new Separator());
    manager.add(getActionStepBack());

    [...]
}

/**
 * Add components contributed by other plugins through the
 * ToolBarContributor extension point.
 *
 * author soh
 * @param manager the manager where to add the components
 */
private void addExternalContributions(final IToolBarManager manager) {
    IConfigurationElement[] contributors = Platform.getExtensionRegistry()
            .getConfigurationElementsFor(
                    "de.cau.cs.kieler.sim.kiem.toolbarContributor");
    for (IConfigurationElement element : contributors) {
        try {
            IKiemToolbarContributor contributor = (IKiemToolbarContributor) (
                element
                    .createExecutableExtension("class"));

            ControlContribution[] contributions = contributor
                    .provideToolbarContributions(null);

            if (contributions != null) {
                for (ControlContribution contribution : contributions) {
                    if (contribution != null) {
                        manager.add(contribution);
                    }
                }
            }
        } catch (CoreException e0) {
            e0.printStackTrace();
        }
    }
}
```

Listing 1: Example for the use of extension point code in the modified creation of the Execution Manager's tool bar.

```
#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.synccharts.diagram.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.synccharts.diagram.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>Synccharts Diagram Editing</EDITOR_NAME>
  <EDITOR_ID>de.cau.cs.kieler.synccharts.diagram.part.
      SyncchartsDiagramEditorID</EDITOR_ID>
 </EDITOR>
```

Listing 2: 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.