A Variability Characterization Curve Editor

Pascal Gamper

June 30, 2005

Abstract

In the recent five years a method for real-time performance analysis for embedded systems has been developed at the Institute TIK at ETH Zurich. Variability Characterization Curves are used to describe different quantities. To create and edit the curves, a flexible software tool was asked. The curve editor provides different editors to edit a curve and import and export functionalities for the filesystem.

This report is divided into three main sections. The first section describes the theoretical background and presents the curve editor in the context. The second section shows the software architecture and the third section provides guides for developers and users.

Contents

List of Figures

Chapter 1

Introduction

1.1 Problem Description

Embedded systems can nowadays be found in almost every technical field of application. Since the time-to-market pressure in the embedded systems industry has become very high, performance analysis plays an important role in the design of embedded systems. The earlier an analysis methodology can be applied within the development process the more efficient the production becomes.

For this reason, an analysis method for embedded systems, called Real Time Calculus (RTC) has been developed at the institute TIK at ETH Zurich. To apply the research results to more realistic systems and to make the research become more popular, there is an ongoing software development in process at TIK. The Real Time Calculus is based on the Network Calculus. In each method different quantities are described by variability characterization curves.

In this semester thesis a tool should be implemented, which provides an ergonomic and intuitive graphical user interface to edit and create these curves. Additionally, it should be possible to import and export the curves from and to different file types respectively.

1.2 Approach

Although no detailed knowledge about the Real Time Calculus method was necessary to fulfill the assigned task, an insight to the theoretical part had to be gained. This had been accomplished by reading several reports about the RTC method [9][4][10]. The main analysis part of this method is done by the RTC kernel which has also been implemented as a software package.

This package had to be explored.

To accomplish the task, founded java knowledge was necessary [5]. Also some knowledge about design patterns emerged as a helpful skill [6]. Secondary the usage of several developing tools had to be learned [1][8]. For the development a subversion repository was installed on a server at the institute TIK.

The next step was the definition of specifications. What should the tool be able to do and what not? How should the user interface look like? A lot of questions about the characteristics of the software had been evaluated and discussed with the assistants. During the specification process also the idea of the software design grew. The reflections about the software design are summarized in the Chapter 4.

The main part of this thesis was the implementation of the requirements into a piece of software. The implementation was often accompanied by design and functionality changes due to a increasingly better understanding of problems related to the programming language as well as to the curve attributes.

Since the main part of the thesis was the implementation itself the tool should be useable and expandable by other developers. Therefore the documentation contains three sections concerning the software, namely a software architecture description part, a developer's and a user's guide part.

1.3 Related Work

A software framework for analysis methods of embedded systems called SymTA/S has been developed at the institute IDA of TU Braunschweig [7]. The name SymTA/S stands for Symbolic Timing Analysis for Systems. It can be used to analyze embedded systems consisting of processors and communication devices. A recent semester thesis was about integrating the analysis methods based on Real Time Calculus developed at TIK into the SymTA/S environment [2]. Some code fragments from this work could be adopted into the curve editor code.

The RTC kernel, responsible for combining and calculating the different variability characterization curves, had been implemented in Java by Ernesto Wandeler. The package provides the basic curve data structure and a lot of different methods for handling the curves. This package is mandatory for the curve editor.

Chapter 2

Theoretical Background

2.1 Overview

Performance analysis plays an increasingly important role in the design of embedded real-time systems because of many today's market-bound properties. The earlier a performance analysis can support the developing process the more efficient the design process will be. Due to this fact an analytical analysis approach for performance analysis of embedded systems, based on real-time calculus, has been developed at ETH Zurich.

An embedded system is a special-purpose information processing system closely integrated into its environment. The embedding leads towards heterogeneous and distributed systems in which the single sub-components communicate with each other via some interconnection networks. Performance and workload information of the whole system not only depends on the single computations of each sub-component but rather on the interaction of those sub-systems.

Typically, embedded systems are reactive systems. They react to stimuli coming from the environment they are connected to and with which they interact. The execution of a certain task due to a stimulus must meet some timing constraints. Not meeting the constraints can lead either to a catastrophic failure of the system (*hard real-time*) or just to mal-functioning execution (*soft real-time*).

The initial causes for an embedded system to start a process are modelled as so-called *arrival events*. In equivalence the finishing of an application can be modelled as *finishing event*. The timing behavior now can be described depending on the time interval between those two events. The characterization of a timing behavior is possible under several terms, e.g. worst and best case execution time (*WCET* and *BCET*) [3], upper and lower bounds or statistical measures. The upper and lower bounds are quantities that bound the worst and best case behaviors. Although the term *performance* is not well defined, it is usually a mixture of the achievable deadline, the delay of events or packets, and of the number of events that can be processed per time unit (throughput).

In order to determine or approximate the above quantities, several methods exist, such as formal analysis, simulation, emulation and implementation. Except the analytic methodology those possibilities should be used with care as only a finite set of possible executions can be observed.

A methodology for performance analysis of embedded systems should at least satisfy some of the following properties: correctness, accuracy, embedding into the design process and short analysis time. Another key requirement for any performance analysis method is the modularity. We can distinguish between several composition properties, such as process, scheduling or resource composition and building components.

2.2 The Performance Network Approach

The approach to performance analysis of embedded systems related to this thesis is influenced by the worst-case analysis of communication networks. The methodology from Network Calculus has been extended to Real Time Calculus in order to deal with distributed embedded systems.

Performance analysis is interested in making statements about the timing behavior for several input characteristics and therefore the concrete event streams that flow between the components must be represented in an abstract way.

The way we have created abstract event streams, we model the resources of the components as abstract resource streams. The performance network of an embedded system works as shown in Figure 2.1: On the left hand side an abstract input modelling the event streams which trigger the tasks of the applications, enters the network. On top we can see the resource modules that model the service of the shared resources. The abstract resource streams interact with the event streams on the performance modules and performance components. The performance components represent (a) the way how the timing properties of input event streams are transformed to timing properties of output event streams and (b) the transformation of the resources. The transfer function of such a component is determined e.g. by the resource sharing strategy.

The event streams propagating through the distributed architecture get increasingly complex and the standard patterns used for modelling event

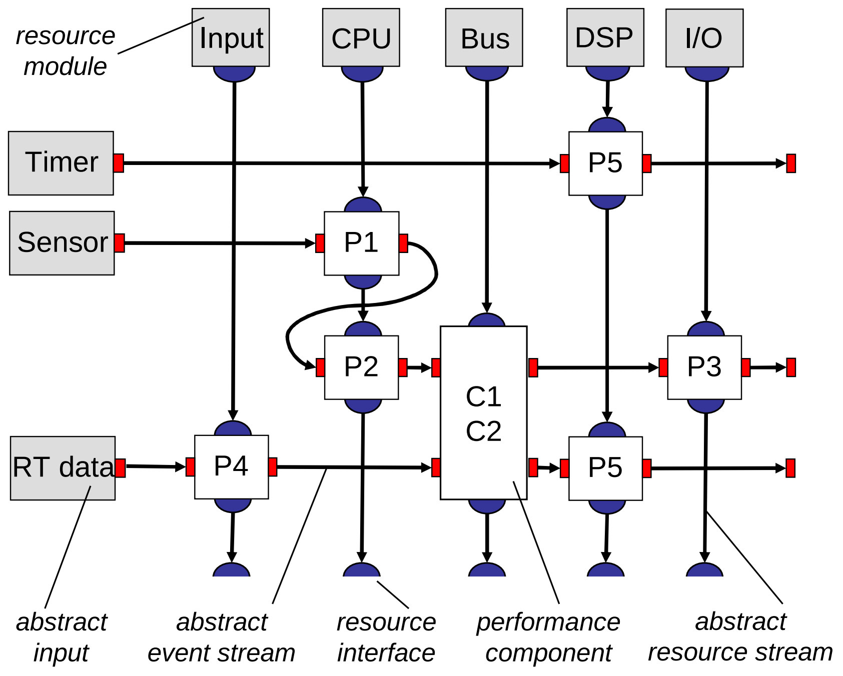
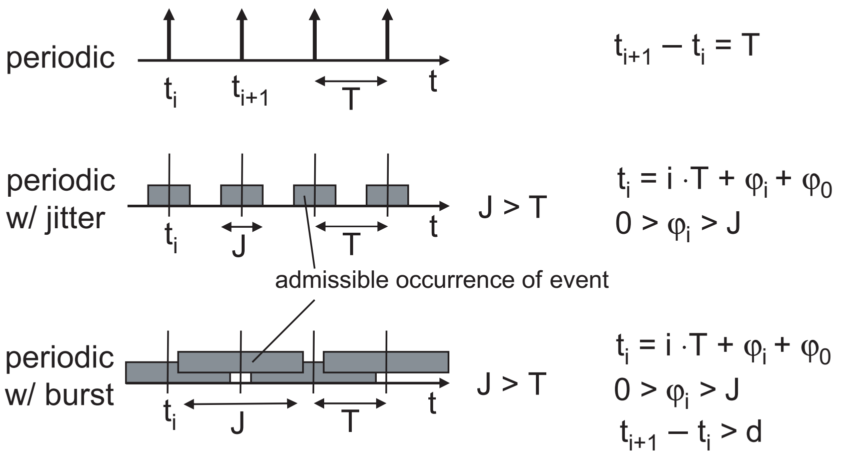


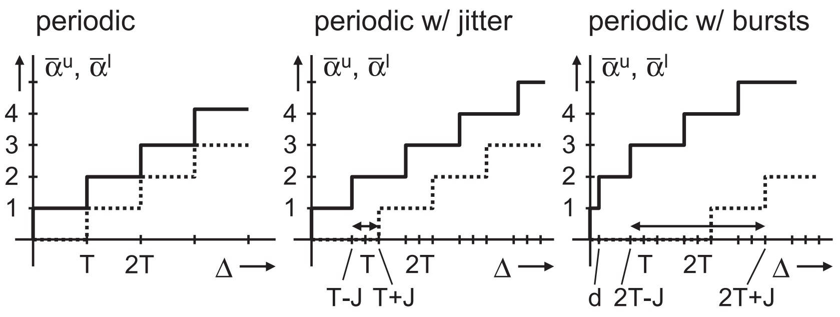
Figure 2.1: A simple performance network illustrating the abstract streams

streams like *sporadic* or *periodic* etc. cannot model the timing properties anymore. To achieve a general representation for the timing characterization of event and resource streams, the *Variability Characterization Curve (VCC)* has been introduced.

The event streams are described using *arrival curves*, which provide upper and lower bounds on the number of events in any time interval of length Δ. In particular, there are at most $\bar{\alpha}^u(\Delta) \in \mathbb{R}^{\geq 0}$ and at least $\bar{\alpha}^l(\Delta) \in \mathbb{R}^{\geq 0}$ events within the time interval $[t, t + \Delta)$ for all $t \geq 0$, $\Delta \in \mathbb{R}^{\geq 0}$. Figure 2.2 shows the patterns for typical event streams and the corresponding variability characterization curves. In a similar way, the resource streams are characterized using *service functions* $\beta^u(\Delta)$, $\beta^l(\Delta) \in \mathbb{R}^{\geq 0}$, $\Delta \in \mathbb{R}^{\geq 0}$, which provide upper and lower bounds on the available service in any time interval. The unit of service depends on the kind of the shared resource, for example instructions (computation) or bytes (communication). Note that as defined above, the arrival curves $\alpha^{u/l}$ are expressed in terms of events, while the service curves $\beta^{u/l}$ are expressed in terms of workload and service respectively. There exist a method to transform event-based curves to workload-based curves and vice-versa. Therefore the workload is often represented as γ curves. The upper and lower workload curve γ denote the maximal and minimal workload



(a) Patterns for typical event streams



(b) Curves corresponding to the event streams

Figure 2.2: Event Streams and the related curves

on a specific resource for any sequence of e consecutive events.

The mentioned curves can be computed from several sources. For e.g. bursty, periodic or sporadic event streams patterns are known and can be constructed analytically. In case of unknown arrival or service patterns, we can use a set of traces and compute the envelope. Another possibility is to derive the curves from the characteristic according to data sheets.In The combination of the resulting arrival and service curves in an appropriate way leads to the necessary information for computing all the relevant information such as the average resource loads, the end-to-end delays an the necessary buffer spaces on the event and packet queues.

2.3 VCC's in conjunction with this Thesis

For editing performance curves an editor has been developed as a plug-in for the SymTA/TS software framework [2]. Although there is the need for a more flexible tool that allows the user to edit a single VCC, no matter if it is from upper or lower type or if it is an arrival or a service curve.

A software package for real-time calculus, called RTC kernel, has been developed. The RTC kernel implements the functionality of the real-time calculus. In the RTC kernel the basic data structure and various functionality are already available.

Because this analysis methodology is still in research, one can not always deal with accomplished facts. Therefore flexibility is one of the main characteristics which this new curve editor should satisfy.

Chapter 3

Curve Editor Context

Because this software project is part of a whole research program, the context of this tool had to be discovered. As Figure 3.1 illustrates, a complete

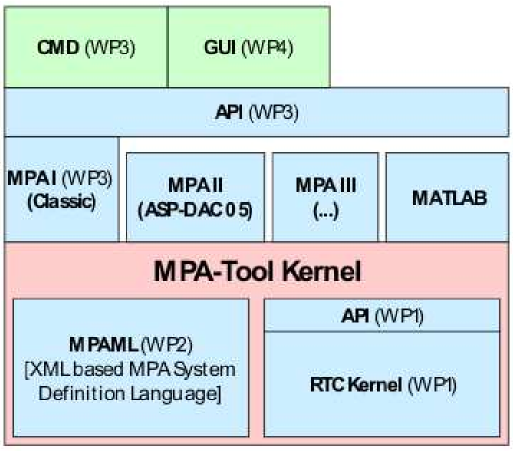


Figure 3.1: The different packages of a complete RTC software tool

software package for real time calculus is planned at TIK. The development process is divided into four work packages. Work package one (WP1) implements an efficient curve description, efficient algebraic methods and workload transformations. WP2 defines a XML based Modular Performance Analysis (MPA) System Definition Language with which systems can be modelled and analyzed. WP3 and WP4 are not defined at the time this curve editor has been written but WP4 will implement the GUI. The work package 1 which contains the RTC kernel has already been implemented. Because the data structure and several methods already existed, there have been some guidelines to follow for handling the curves.

We can see in Figure 3.1 that a graphical user interface (GUI) component stands on the top of the API. A complete GUI for all functionality is desired. Because the curve editor would probably be embedded into the GUI, the curve editor should be easily useable as a part of it. This means that the GUI of the curve editor has to be flexible in look and size and the API of this tool should be clear accessible from other programs.

Chapter 4

Design Reflections

4.1 Finding the key concept

From the beginning on it was clear that a curve should be editable in different ways. The idea was to have a textual and a graphical representation of a variability characterization curve for creating and modifying.

The question rose if every curve representation should have its own data structure or if these representations should just be a kind of view on the same curve. I have decided to the latter because

- the actual data structure for a curve has already been defined in the RTC kernel package,
- if only a single centralized data structure exists, the tool becomes more flexible and easier to expand,
- modified data is always visible for every class which knows the data structure,
- the curve managing part has the overview of all attribute changes. This way the possibility for a powerful interface is provided.

The way this key concept has been realized, is described in the Chapter 5.

4.2 Finding the right data structures

As already a curve data structure was provided by the RTC kernel package, at first glance it seemed not necessary to define another curve class. But reflections about how curves should be saved and loaded to and from the file system respectively have shown that some slight extensions would be necessary. The decision not to derive the new extended curve data structure from the existing one but to add a member which references the existing class turned out to be very lucky. This way for example an undo / redo module can just change the reference to another original curve but the extended curve itself remains the same.

I have decided to use a centralized data structure. For a centralized data structure a managing class would be reasonable. The manifest design pattern for a centralized class structure is the singleton pattern. Thus the main class `CurveEditor` is designed as a singleton. The `CurveEditor` class should not just be the center of the curve editor itself but also every other application should be able to access this class. Further reflections about the file system etc. have borne a set of functions for a general API. Figure D.1 shows a simplified class layout.

4.3 Curves and the file system

Nowadays a software tool is intuitive and ergonomic if it at least orientates itself at the defacto standard software which is - boon or bane - the windows standard. Applied to the file handling certain points had to be considered:

- If some data has not been saved, e.g. some changes on the data occurred, it is highlighted with an asterisk or a similar symbol.
- No matter what kind of data is opened, for convenience more than one item can be kept opened at the same time. Additionally the items should be treated separately.
- The supported file types should be extensible for later add-on's.

The first point can be realized with two additional attributes for the curve data structure: a field for a base file for this curve and a field which knows if changes have occurred on the curve data.

The second point makes clear that not only a single curve can be available at a time but an unknown number of curves. Thus a set of curve data structures has to be implemented. Within this curve set only one curve can be active for modifying at a time.

The last point bears the idea of modular file filter implementations which will be dynamically loaded. The modular design has proven as a very powerful way to implement several features and is presented in the next chapter. The so far presented discussions have led to the layout of the `CurveEditor` class shown in Figure D.1.

4.4 Functionality Evaluation

When the main concepts have itself tightened, the design and implementation of the framework could begin. In parallel to this process the functionality of the GUI could be evaluated. Especially for the graphical view some features had been demanded:

- As the aperiodic and the periodic part of a variability characterization curve can widely differ from each other in range in the time or even amplitude domain, they should be treated separately.
- Different zoom and coordinate shifting functions to navigate inside the coordinate system should be available.
- Different intuitive drawing modes have to be found and implemented.
- An intuitive tool supports several functions like undo / redo, (multiple) selecting, modifying selection etc.

For the text based view the curve data first was thought to be just formatted the way Matlab does. But the idea came up to implement a curve property window which shows the different attributes of a curve in a table like the way many integrated development tools (IDE's) do. So the text format changed according to the XML data structure.

4.5 Ideas for additional features

The idea for a lot of features now integrated in the curve editor arose simultaneously to the implementation. E.g. features like modular *wizards* for creating new curves or modular option dialogs came up while extensively using the eclipse developing platform.

Chapter 5

Software Architecture

5.1 Design Concept

The graphical and text based view on a curve probably remain not the only views. Therefore the views have to be loosely coupled to the framework. This way an additional view can easily be appended. Figure 5.1 demonstrates the concept of different views on the same curve residing in the framework. Further the figure shows that all opened curves are stored in a list and the

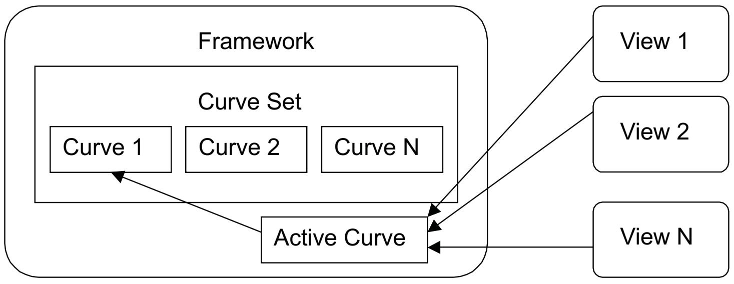


Figure 5.1: The modular views on the active curve

active curve can point on every curve in this list. Now the problem arises that if a view makes some changes on the curve, the other views have to be informed that changes occurred because they probably have to update their representation of the curve. To come by this problem, the views have to register themselves with the framework. If a view makes some changes it has to inform the framework that the curve data has changed. Then the framework will tell every view that changes occurred. A similar procedure will be used if e.g. the active curve changes its reference to another curve in the curve set.

Because the number of views is not known and the views will be loaded during the starting up of the curve editor, each view has to fulfill some requirements. To guarantee that the requirements are fulfilled every view has to derive the main classes from so-called models which are abstract classes. But the framework still has to know where to find a certain view and how the classes are named for loading it properly. Configuration files will list all the different views, called modules, and their location.

Not only modules are derived from a model and are loaded dynamically but also curve *wizards* and the file filters for exporting and importing curve data.

5.2 Package Hierarchy

Figure 5.2 shows the package hierarchy. Below a short description of each package is given.

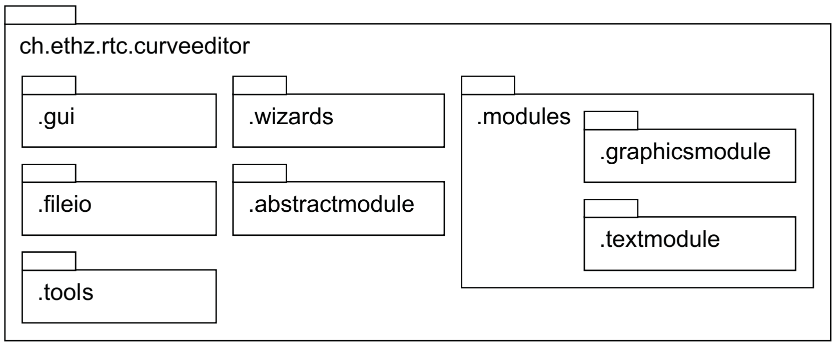


Figure 5.2: The package hierarchy

.gui The GUI package contains the actual framework. For instance the main gui classes can be found in this package as well as the core class `CurveEditor` and the extended curve class `ExtendedCurve` of the curve editor.

.tools Although most of the classes in this package are used by the framework of the curve editor, the classes could be also used by other packages for example by a module. Therefore some classes have been separated from the GUI package.

.fileio The `fileio` package contains all the classes related to the file handling. There is an abstract class model for a file importer and exporter, an importer and exporter loading and managing class, and two realized importer and exporter for Matlab and XML files respectively.

.wizards All classes necessary for the user to create a new curve are placed in this package. It contains an abstract wizard model, a wizard model loading class, a wizard dialog which displays the wizard models and two realized *wizards*.

.abstractmodule The templates for creating a new module are placed here.

.modules Every implemented module package should be placed inside this package but it is not necessary though. Two modules have been realized to use within the curve editor namely a graphical and a text based module.

5.3 Modular Framework Implementation

This section describes how the core design of the curve editor is constructed and how it works. First the core class `CurveEditor` and all the abstract models are presented and the Section 5.3.3 describes how an implementation of a model is registered. Further the Java mechanism to load classes dynamically is described and in the last section the module communication is explained.

5.3.1 The singleton class `CurveEditor`

It can exist only one instance of a singleton class. Often most of the functions are static. The `CurveEditor` class is constructed this way. The core class manages the dynamic loaded models, the opened curves and provides an API for modifying the curve list. It does not matter if a *file dialog* of the curve editor itself opens a new curve or if an extern software component tells the curve editor to open a curve because both would use a method from the API named `openCurve(Curve newCurve)`. Figure D.1 shows a simplified class layout. The communication with the models is described in the next chapters.

5.3.2 Models

Models are a set of abstract classes. By declaring an abstract class it is assured that each derived module will implement certain functionality and it has a predefined interface. As mentioned earlier the curve editor provides three different models in general. The module model which provides the

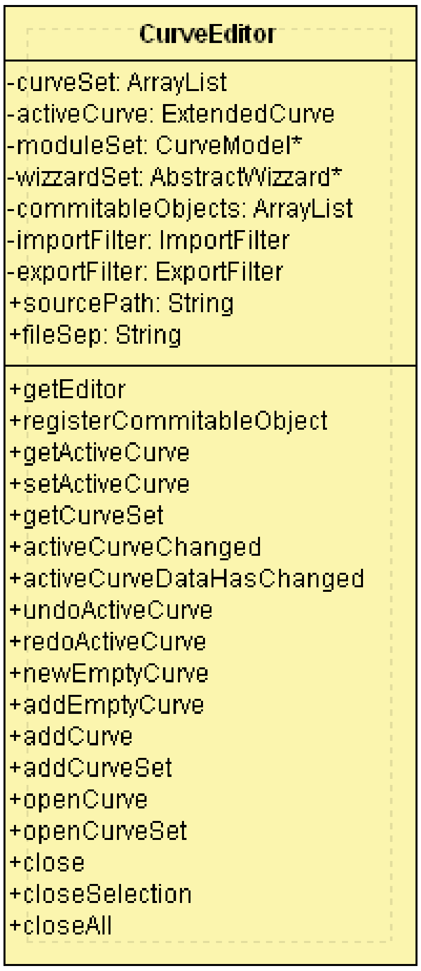


Figure 5.3: Class layout of `CurveEditor`

abstract classes for a curve view, the file filter model, subdivided in an import and export model, and a wizard model. These models are described in detail below.

Module Model

The module model consists of three classes. A valid module implements at least the two abstract classes `CurveView` and `CurveModel`. `CurveModel` could be seen as the access point for the framework because it references the other classes like the visible class `CurveView`. The class `CurveOptions` can be implemented but if it is not a template class for the options dialog will be used. Also the class `CurvePrinter` is facultative to implement. Because this class is not implemented yet it is not shown in the figure.

The classes `CurveView` and `CurveOptions` are derived from the `JPanel` class in `javax.swing`. `CurveView` represents the actual module view and `CurveOptions` the view of the options panel for this module. Figure 5.4 shows the UML diagram for the module model.

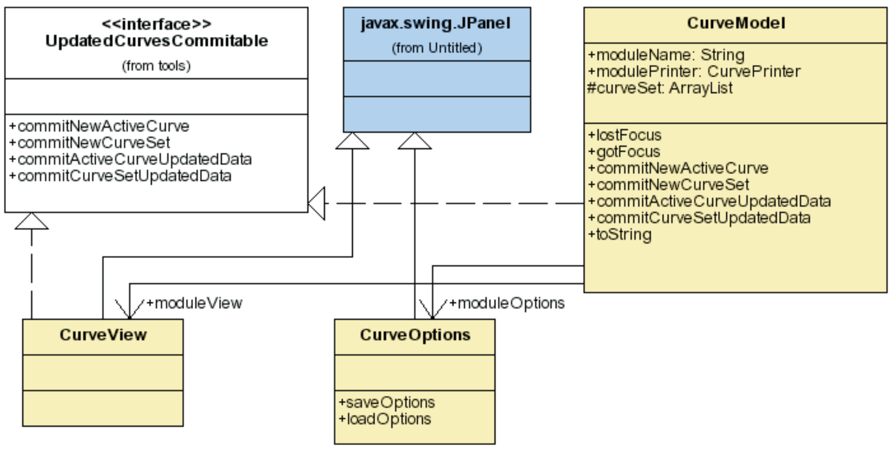


Figure 5.4: Class diagram of the module model

File Filter Model

This model is very simple. For each import and export functionality exists an abstract class respectively. If we want to write a file filter for a certain file type we just derive our filter from one of the two abstract classes. Figure 5.5 shows the class layouts.

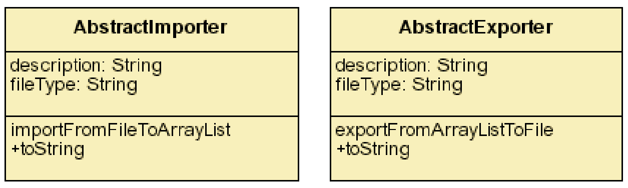


Figure 5.5: The classes of the file model

Wizzard Model

The wizzard model is an abstract class derived from `JPanel` in javax.swing. This abstract class will be displayed embedded in a dialog with other `JPanel` instances. Figure 5.6 shows the class layout.

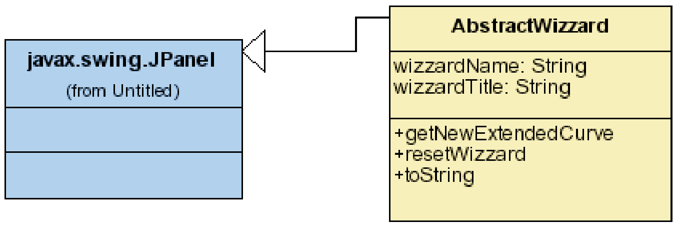


Figure 5.6: The classes of the wizzard model

5.3.3 Config Files

Text based config files are placed in a config folder in the root application folder of the curve editor. In several files the derived models can be registered and will be loaded according to the entries in the config files. If an entry leads to a loading failure a loading exception will occur. The class `LoadingException` is placed in the package `tools`. The different configuration files are `modules`, `wizards`, `importers` and `exporters`. In the developer's guide the layout of the configuration files is described in details.

5.3.4 Dynamic Class Loading

Java allows dynamic class loading. With dynamic class loading all model implementations can be loaded at run time. For each group of models (modules, wizards and file filters) exists a loader class which loads the classes registered there in the corresponding configuration files into a list. The curve editor manages the model lists, e.g. decides which of the models is active etc. The listed code fragment below shows the typical class loading procedure in the model loader classes:

```
ArrayList moduleModels = new ArrayList();
try
{
    Class modelClass = Class.forName(moduleName);
    Object modelObject = modelClass.newInstance();
    moduleModels.add(modelObject);
}
catch(ClassNotFoundException e)
{
    System.err.println(e);
}
```

Java also provides methods to load methods dynamically. This feature is used in the tools dialog. The methods are explained in the according section.

5.3.5 Module Communication

The previous subsections described the different models and the loading procedure of the models into the framework. Now we have a look at the module models and how they are used inside the curve editor. The Figure 5.7 shows the interface `UpdatedCurvesCommitable`. This interface is implemented at

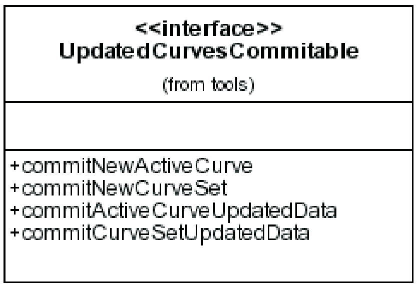


Figure 5.7: The interface `UpdatedCurvesCommitable`

least in the `CurveView` and `CurveModel` class of a module model. Implementing this interface makes it possible to register a class at the class `CurveEditor` and at the framework respectively. The framework knows which functions this interface contains and can call them if necessary.

If the interface is implemented, registering a class at the framework only needs to call the following method:

```
CurveEditor.registerCommitableObject(this);
```

Calling this method the curve editor adds the new class to the list of commitable objects. If for instance a module makes some changes on the active curve, it will call

```
CurveEditor.activeCurveDataHasChanged()
```

The curve editor commits the fact that the curve data has changed to all registered objects with the following method:

```
private static void activeCurveDataChanged()
{
    if ( activeCurve != null )
        activeCurve.updateData();
        for (int i = 0; i < commitableObjects.size(); i++)
        {
            try
            {
                ((UpdatedCurvesCommitable)(commitableObjects
                    .get(i))).commitActiveCurveUpdatedData();
            }
            catch(ClassCastException e)
            {
                commitableObjects.remove(i);
                System.err.println(e);
            }
        }
    }
}
```

Now the modules can react appropriately to the curve changes and update their states.

5.4 Graphical Module Implementation

The graphical module provides the functionality to create and modify the curves in a graphical way. As a graphical editor should provide standard features like selecting objects, moving selections etc. classes are arranged with respect to functionality. Figure 5.8 shows a simplified class diagram of the functionality related classes. View classes like the toolbar are not shown. The class `GraphicsCurveView` contains a toolbar and a drawing panel. In the toolbar a draw state is selectable and some actions like zooming or coordinate shifting are available. The draw state will be set in the class `GraphicsCurveView` and also the actions will be called from this class. The draw panel displays the active curve and reacts to user inputs. So both the view and the panel need to call graphical methods. The methods are located in a centralized class `CoordinateSystem`. This class contains other more specialized classes like the selection manager or the painter.

A variability characterization curve can contain an aperiodic part and a periodic part. The latter will be periodically continued. The two parts can

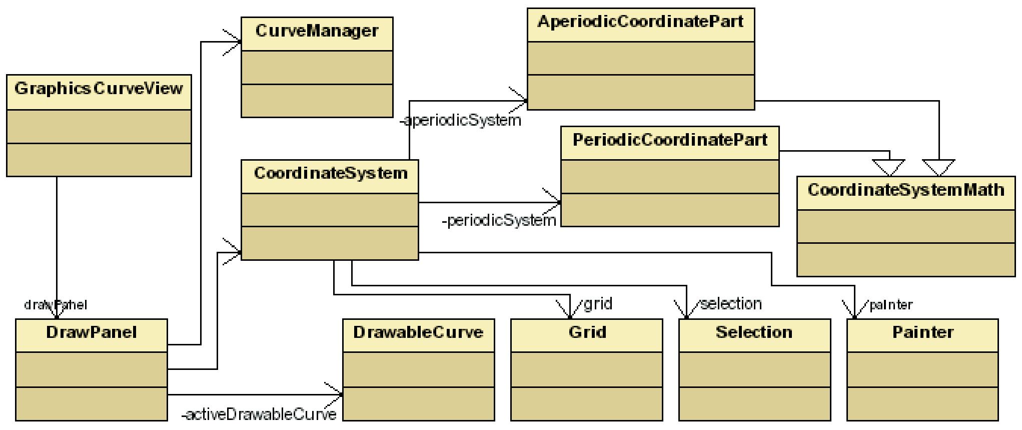


Figure 5.8: Simplified class diagram of the graphical module

largely differ in the time domain. Therefore two separate coordinate systems are useful. Common functionality for both systems is implemented in the abstract class `CoordinateSystemMath` from which the classes `AperiodicCoordinatePart` and `PeriodicCoordinatePart` are derived.

5.5 Tools Dialog Implementation

The RTC kernel package contains various functions for generating new curves or modifying existing curves. The *tools dialog* loads methods for modifying and generating curves, registered in a configuration file, dynamically and displays them in a list. In the configuration file stands the fully classified class name, the method name and the number of curves as arguments. The following code segment illustrates the dynamic method loading process for a method with two curves as arguments. The class name is stored in `fctnElement[0]` as a string and the method name in `fctnElement[1]`. Note that the code example is not complete.

```
try
{
    Class modelClass = Class.forName( fctnElement[0] );
    Class[] classArr = new Class[2];
    classArr[0] = new Curve().getClass();
    classArr[1] = new Curve().getClass();
    Method method = modelClass.getMethod(fctnElement[1], classArr);
    Object[] objArr = new Object[2];
    objArr[0]=((ExtendedCurve)cbCurveA.getSelectedItem()).baseCurve;
    objArr[1]=((ExtendedCurve)cbCurveA.getSelectedItem()).baseCurve;
    Curve cNew = (Curve)method.invoke( modelClass, objArr);
} catch ( Exception e ) {}
```

Chapter 6

Conclusions and Outlook

The standard Java libraries provide a wide range of different methods which can be used in an implementation with almost no effort. If a special-purpose implementation is needed often features of the Java Core libraries are not directly applicable according to the programmer's desire and third party libraries are too less efficient and oversized. Then the implementation takes considerably more time than estimated and more functions have to be implemented by the programmer himself. This has been experienced particularly during the implementation of the graphical module. If in this case all the common functionality known from professional developed software tools is desired, the programmer has to spend a lot of time in the implementation of those functions. Furthermore the programmer of a tool designed for special purposes has to consider a lot of user input possibilities by himself for achieving complete deterministically behavior of his software.

For the development of the curve editor sometimes highly sophisticated libraries such as the XML handling libraries could be used and sometimes functions of a lower abstraction level had to be implemented by myself, e.g. the mouse handling and the zoom functions in the graphical drawing panel.

The curve editor provides the ability to extend modules, supported file types, curve modifying methods and curve wizards. The two implemented modules, a graphical and a text based module, are stable and presumably the functionality of both is sufficient for the time being. Whereas the implemented file importers and exporters for Matlab and XML formatted text files have moderate functionality. Here some improvements could be done. A prototype for a curve checker is implemented. An implementation which tests e.g. the active curve for errors and provides applicable solutions to fix the found errors could be useful. The practical use of the curve editor will reveal where some improvements and further functionality are needed.

Furthermore, if a lot of unexperienced users should use the curve editor, the embedding of the Java Help System would be a reasonable task.

Appendix A

Developer's Guide

A.1 The `config` folder

In the `config` folder two types of files are located. For registering models the listed files below are used.

modules A line contains the fully qualified class name of the class derived from `CurveModel.java`.

wizards A line contains at most one class name appendant to a wizard which is located in the `wizard` package and derived from `AbstractWizzard`.

importers A line contains one name of an importer derived from `AbstractImporter` located in the `fileio` package.

exporters A line contains one name of an exporter derived from `AbstractExporter` that is located in the `fileio` package.

tools.config A line is divided in three parts: [full-qualified class name];[method name];[number of curves as arguments].
Example: `ch.ethz.rtc.kernel.CurveMath;RTinvBeta;2`

The files `gui.options`, `graphics.options` and `text.options` are not supposed to edit, because they are built by the preferences dialog. Nevertheless values could be changed.

A.2 Writing a file filter

A.2.1 file importer

1. Your class should be in the package fileio:
   `package ch.ethz.rtc.curveeditor.fileio;`

2. We have to derive our file importer class from `AbstractImporter.java`:
   `public class MyImporter extends AbstractImporter`

3. Overwrite the standard constructor and call the super constructor:
   `super("type","MyImporter imports curves from files.type");`
   The first field represents the exact file type extension and the second field represents a description of this importer.

4. Implement the method `public void importFromFileToArrayList( ArrayList curveSet, File file) throws IOException, FileNotFoundException`.
   This method takes a reference to the file object from which we want to import the curve data and fills the found curves into the array list. A simple example:

   ```
   public void importFromFileToArrayList(ArrayList curveSet, File
       file) throws IOException, FileNotFoundException
   {
           String line;
           String[] elements;
           BufferedReader rdr = new BufferedReader(
               new FileReader(file));

           while ((line = rdr.readLine()) != null)
           {
               if (line.startsWith("\%") || line.trim() == "")
                   continue;
               elements = line.split("=");
               curveSet.add(new ExtendedCurve(null, file,
                   CurveFactory.createFromString(elements[0]
                   .trim())));}
           }
   }
   ```

5. Insert a line in the file `importers` in the config file with the class name of your importer e.g. `MyImporter`

A.2.2 file exporter

The file exporter will be created similar to the file importer above. The only difference is the different set of methods to implement:

- `void exportFromArrayListToFile(ArrayList curveSet, File file) throws IOException, FileNotFoundException`
- `void exportFromArrayListToFile(ExtendedCurve curve, File file) throws IOException, FileNotFoundException`

A simple version for the first method could look like the following:

```
void exportFromArrayListToFile(ArrayList curveSet, File file)
    throws IOException, FileNotFoundException
{
    PrintWriter out = new PrintWriter(new BufferedWriter(
        new FileWriter(file)));
    String[] setType = new String[0];
    for (int i = 0; i < curveSet.size(); i++)
    {
        out.println(((ExtendedCurve)(curveSet.get(i)))
            .curveName+" = "+(((ExtendedCurve)(curveSet.get(i)))
            .baseCurve.toExportString()+";"));
    }
        out.close();
}
```

A.3 Writing a module

A module has not to be placed in its own package although it is at least recommended to group the modules. The following recipe is kept very simple and does not implement an option class.

1. Your classes use the package `abstractmodule` so import this package:
   `import ch.ethz.rtc.curveeditor.abstractmodule.*;`

2. Derive your classes from the classes in the `abstractmodule` package. For a valid module you have to implement the classes `CurveModel` and `CurveView`. The options class is optional and the printer class is not supported yet. Example:
   `public class MyCurveModel extends CurveModel`
   `public class MyCurveView extends CurveView`

3. In your model class call the super constructor. If you have not implemented one of the optional classes, use a `null` reference as parameter: `super("MyView", new MyCurveView(), null, null );`

4. Because the abstract class `CurveView` implements the interface `UpdatedCurvesCommitable` you have to implement this interface.

5. Now it is up to you to unleash your creativity. Note that your class `MyCurveView` is the class which will build the view for the module. It is derived from `javax.swing.JPanel`.

6. Insert a line into the file `modules` in the config folder with the fully qualified class name of your model class. Example: `ch.ethz.rtc.curveeditor.modules.mymodule.MyCurveModel`

If you would like to implement your own option class for your module make the following steps:

1. Create your class as you did with the model and view class above and derive it from `CurveOptions`.

2. Change the super constructor call in the model class. An example:

   ```
   super("MyView", new MyCurveView(), null,
       new MyCurveOptions() );
   ```

3. Implement the abstract methods

   ```
   public void saveOptions() throws IOException,
           FileNotFoundException}
   public void loadOptions() throws IOException,
           FileNotFoundException}
   ```

   The saveOptions() method will be called by the framework at the right time. The loadOptions() method is just for completeness but it is recommended to use this class to load the options from a file. You are responsible for yourself for loading the options. Note that the abstract class `CurveOptions` extends the `javax.swing.JPanel` class and so your options class does.

A.4 Writing a wizzard

1. Place your class in the package `ch.ethz.rtc.curveeditor.wizzards`.

2. Derive your class from `AbstractWizzard`.

3. Implement the abstract methods.
   `public ExtendedCurve getNewExtendedCurve()` should return the ExtendedCurve object created by this wizzard.
   `public void resetWizzard()` should reset the layout of the wizzard e.g. set all the text fields to zero or similar.

4. Your wizzard is derived from the `javax.swing.JPanel` class and embedded in a wizzard dialog. This dialog disables the OK button until you call a certain method: `WizzardDialog.setWizzardReady(true);` From this moment on the user can press the OK button and your method getNewExtendedCurve() should return a valid curve. It is recommended to disable the OK button if no valid curve would be returned. This is done with `WizzardDialog.setWizzardReady(false);`.

5. Insert a line in the file `wizzards` in the config folder with the class name of your wizzard e.g. `MyWizzard`.

A.5 Writing and registering additional methods

The tools dialog in the curve editor is able to load methods registered in a file and execute them to modify curves or create new curves. We just have to register our method located in any package in the right configuration file named `tools.config`. The method should have one of the following skeletons:

- `{void, Curve} myMethod(Curve)`

- `{void, Curve} myMethod(Curve, Curve)`

This syntax means that a method can either return a new curve object or the return type is void and that as arguments one or two curves are allowed. Have a look at the section *The config folder* for the syntax of a method registering entry.

Appendix B

User's Guide

B.1 Getting Curve Editor to work

The `curveeditor.jar` archive of the curve editor software package has to be in the same directory as a folder named `config`. In the folder `config` are two different types of files: First there are the files `modules`, `importers`, `exporters`, `wizzards`. These files are important for getting the curve editor to work. The second file types are the .options files. They are just named this way for better recognition. If the option files aren't at the right place, corrupt or incomplete, the curve editor will use default values instead.

For proper working you have to copy the `jdom.jar` library into the directory `/lib/ext` of your current java distribution. Then make sure you have the `config` folder with the `curveeditor.jar` archive in the same directory and that `config` contains the necessary files. Now just execute the `curveeditor.jar` archive.
Note: On the console it is necessary to type `java -jar curveeditor.jar`.

B.2 The GUI

Figure B.1 shows the GUI right after starting. There are three marks which show the partition of the main window.

1. The detailed functionality of the menus is described in the section *menu.*

2. Here are the different modules placed. Switch between the modules by clicking on the different tabs.

3. The curve window presents a property and text based view of the active curve and on the left side it lists all opened curves. See section *Curve Window* for a more detailed description.

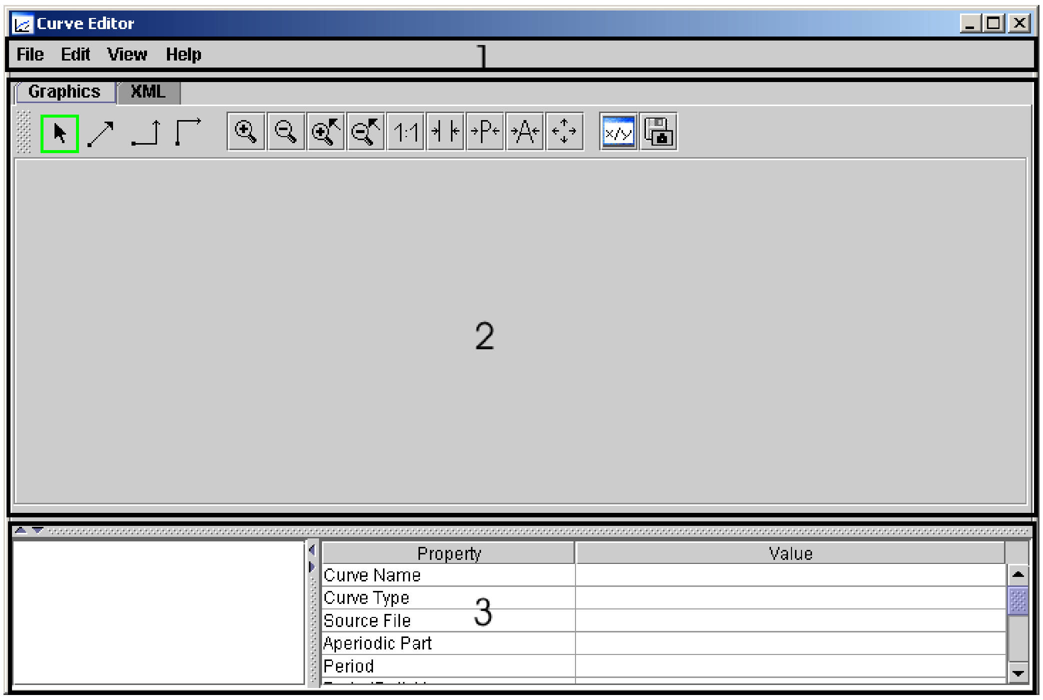


Figure B.1: The GUI without any curve opened

B.3 Opening a curve

For opening one or more curves from a file click either on the *Open Curve(s)* or *Add Curve(s) From File* menu entry in the menu *File*. The open command will close all opened curves and only load the new ones. The add command will add the curves from a file to the curves already opened. Figure B.2 shows the open entry in the file menu. If you have clicked on one of these menu entries a file open dialog appears. Select the file you want and press the open button. If the curve editor has found some valid curves in the file, the dialog of Figure B.3 will appear. Select the curves you want to open e.g. with the control key and the mouse. You also can just check the *select all* field to mark all curves. Then click *OK*. The curves will now be shown in the curve list in the curve window.

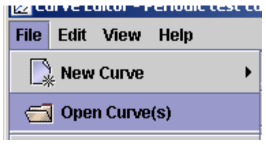
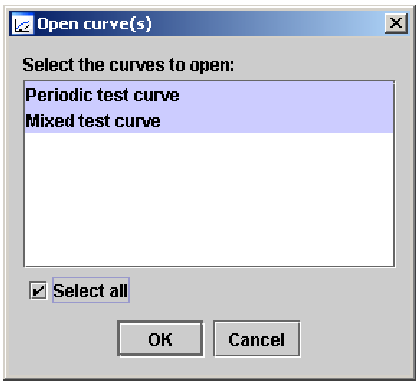


Figure B.2: Open a curve


Open curve(s)

Select the curves to open:

Periodic test curve
Mixed test curve

Select all

Figure B.3: Dialog to select the curves to open

B.4 Creating a new curve

For creating a new curve you can either choose the *New Curve* or the *Add New Curve* menu entry in the file menu. The new command will close all curves first and then create a new curve. The add command on the contrary will just add the new curve to the curves already opened. Under both menu entries you can see all the different possibilities to create a curve like Figure B.4 shows.

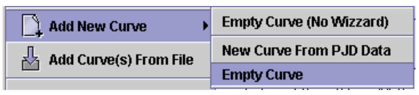
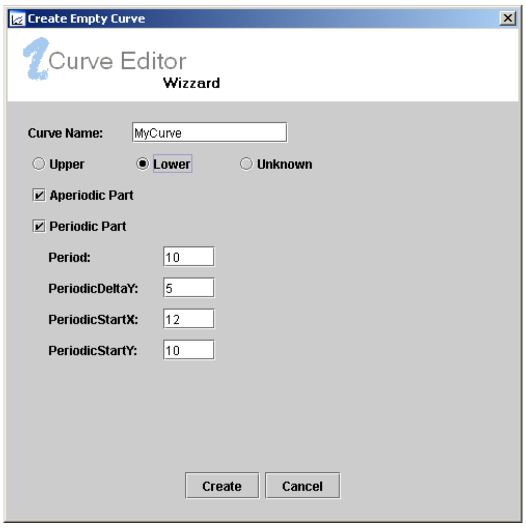


Figure B.4: The different possibilities to create a curve

Empty Curve (No Wizzard) Creates an empty curve without any content.

New Curve From PJD Data Creates a curve from the values of period, jitter and minimum inter-arrival-distance.

Empty Curve Creates an empty curve. You can set the attributes of the periodic part and the curve name. Figure B.5 shows the *Empty Curve* Wizzard.



Curve Name: MyCurve

Upper [✓ Lower] Unknown

[✓] Aperiodic Part

[✓] Periodic Part

Period: 10

PeriodicDeltaY: 5

PeriodicStartX: 12

PeriodicStartY: 10

Figure B.5: A wizzard to create empty curves

B.5 Editing a curve

If at least one curve is opened, one of the opened curves is the active curve. That means that this curve is editable. You can edit the active curve in two different ways: Either you use a module e.g. the graphical module or you use the curve window. How to use the different methods in particular is described in the according sections. When you have changed a curve it will be marked with an asterisk in the title bar of the curve editor. That means that it has unsaved changes. Also if you create a new curve which has never been saved to a file, it will be marked as unsaved. Generally the asterisk means that a curve with its present data is nowhere saved in this form. If you would close such a curve, you will be asked if you want to save the curve.

B.6 Menus

Figure B.6 shows the different menus. The help menu is not shown.

- **File Menu**

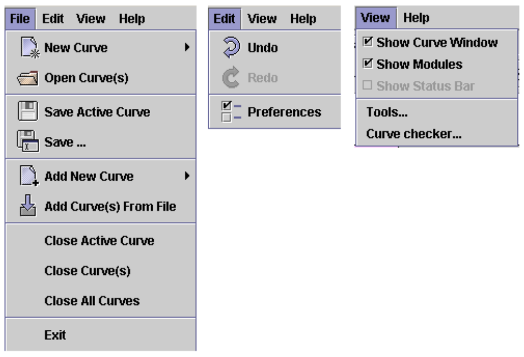


Figure B.6: The different menus (without the help menu)

New Curve Shows a list of all possibilities to create a new curve. The curves already opened will be closed.

Open Curve(s) Opens a selection of the curves found in a certain file. The curves already opened will be closed.

Save Active Curve Opens directly a save dialog for the active curve. Choose a file or enter a new file name to save the curve and press save.

Save... Opens the save dialog with the list of all opened curves. Select curves you want to save and the save mode. Then press save. A save dialog opens. Continue like above.

Add New Curve Shows a list of all possibilities to create a new curve. The new curve will be added to the curves already opened.

Add Curve(s) From File Opens a selection of the curves found in a certain file. The opened curves will be added to the curves already opened.

Close Active Curve Closes the active curve. If the curve has unsaved changes you will be asked if you wish to save the curve.

Close Curve(s) Opens a dialog to select the curves you want to close. Again you will be asked for savings if some curves have unsaved changes.

Close All Curves Closes all curves.

- **Edit Menu**

Undo Undoes the last changing on the active curve.

Redo Redoes the last undo command if some changing have been undone.

Preferences Opens the preferences dialog.

- **View Menu**

Show Curve Window Makes the curve window visible or not.

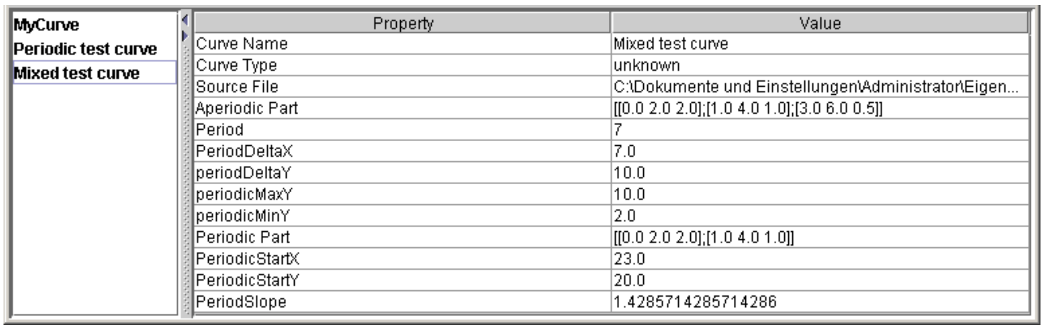
Show Modules Makes the module tabs visible or not.

Tools... Opens the tools dialog.

Curve Checker... Opens the curve checker.

B.7 Curve Window

On the left side in the curve window all opened curves are listed. The active curve is shown in a property table on the right side. Click on a curve in the list to make it the active curve. Double click on a property field to change its value. Make sure the syntax is correct when pressing the up, down or enter key. You can also just navigate with the key board in the property table and write directly into the active field. Note that some fields are not editable. The Figure B.7 shows the curve window. The illustrated curve is a good example for the syntax.



MyCurve
Periodic test curve
Mixed test curve

Property	Value
Curve Name	Mixed test curve
Curve Type	unknown
Source File	C:\Dokumente und Einstellungen\Administrator\Eigen...
Aperiodic Part	[[0.0 2.0 2.0];[1.0 4.0 1.0];[3.0 6.0 0.5]]
Period	7
PeriodDeltaX	7.0
periodDeltaY	10.0
periodicMaxY	10.0
periodicMinY	2.0
Periodic Part	[[0.0 2.0 2.0];[1.0 4.0 1.0]]
PeriodicStartX	23.0
PeriodicStartY	20.0
PeriodSlope	1.4285714285714286

Figure B.7: The curve window

B.8 Preferences Dialog

The preferences dialog shows all the loaded options in a tree. Just navigate through the tree and select the different option leafs. Look at the corresponding module descriptions on how to use the options in particular.

B.9 Graphical Module

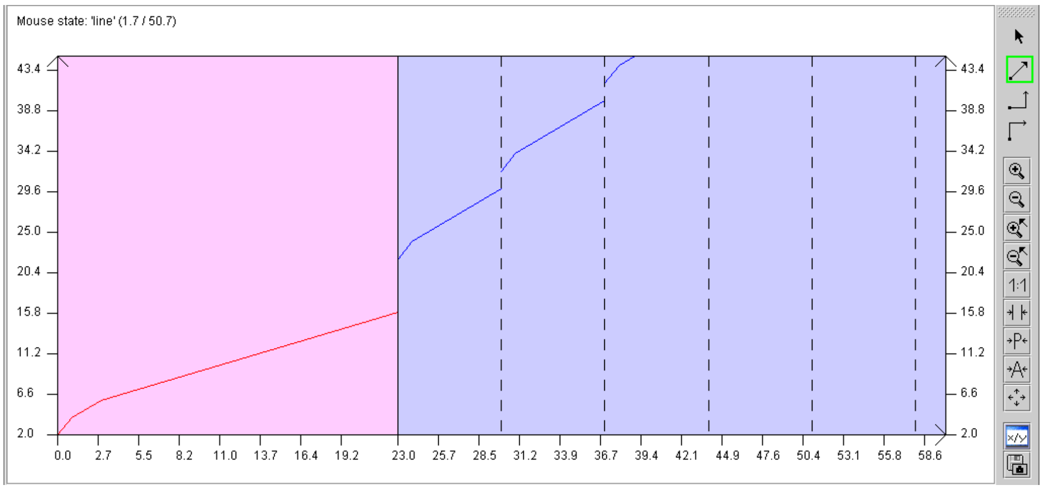

Mouse state: 'line' (1.7 / 50.7)

Figure B.8: The graphical view

B.9.1 Drawing Curve Segments

Line draw mode

Lower Manhattan draw mode

Upper Manhattan draw mode

Click on one of the three above illustrated drawing modes (line, upper or lower manhattan draw mode). Click inside the coordinate system at your desired location. On the top left corner you can see at which coordinates your mouse is. Click once more and you have created a curve segment. Repeat these steps as long as you need. To abort the current draw session just click the right mouse button inside the coordinate system. Note that if you are drawing in the aperiodic part and click on the periodic part, just this next segment will be painted and the drawing session will be aborted. The segment will be cut off at the end of the aperiodic part. This way it is guaranteed that the segments are continuous. Similar in the periodic part every segment painted outside of the period in which the painting had begun will be cut at the period border. It is recommended to draw periodic segments only in the first period. Note that mouse clicks are only valid inside the coordinate system. So if you have zoomed into one part and cannot see the right border of this part you cannot draw the segment to the outside of the coordinate system. Instead you can only draw exactly to the right coordinate border by clicking on it. Then you will have to abort the current drawing session with clicking the right mouse button inside the coordinate system.

B.9.2 Selecting Curve Segments

Selection mode

To select curve segments you have to choose the selection mode. Now click on a curve segment. It will get colored with the selection color. You can delete and add curve segments with holding the control key and clicking on the curve segments. By clicking not on a curve segment without the control key the selection will be cleared. Note that the selections are separate for the aperiodic and the periodic part. The selection behavior follows the behaviors of the standard programs.

If you have selected some segments you can move them with drag and drop. The end points of the segments can be moved each for its own. If you intend to move single end points it is recommended to just select one segment. When you are dragging segments or end points you can click the right mouse button to cancel the movement. Note that in this case the left mouse button should be released after clicking the right mouse button.

Deleting the current selection is simple. Just click the right mouse button with the mouse over a selected segment and press the delete button. Only the selections of the current curve part will be deleted. The Figure B.9 shows the button to delete the current selection.

B.9.3 Zooming and Shifting the Coordinates

The pictures (1) to (8) show the different available zoom functions with a short description.

You can shift the coordinate system in the shifting mode. Click on the button shown in picture (9). If you have selected the mode just shift the view with drag and drop.

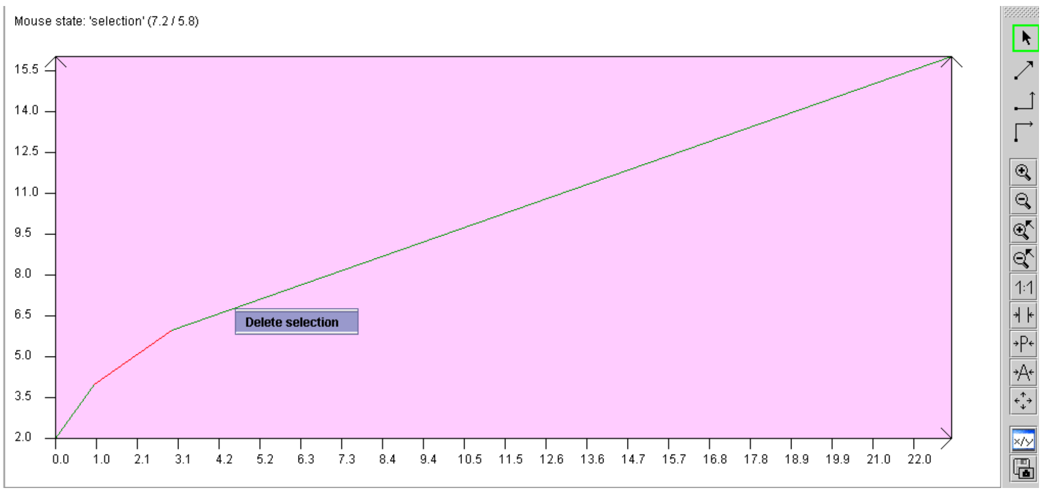


Figure B.9: The popup button to delete the selection

(1) Zooms into the view. The bottom left coordinate will remain the same.

(2) Zooms out of the view. The bottom left coordinate will remain the same.

(3) Zooms into the painted rectangle. Select this mode, click into the coordinate system and span a rectangle in the upper right direction. Then click again and the coordinates will zoom into the rectangle. Abort with the right mouse button.

(4) Zooms out the coordinate part in which the you have clicked with the mouse.

(5) Fits the window to the curve with the same scaling factor for both x and y axis.

(6) Fits the window to the curve. The axis scales have not to be the same.

(7) Fits the window to the aperiodic part.

(8) Fits the window to the periodic part.

(9) Move the Coordinate System

To determine an arbitrary view, click on the *show coordinate dialog* button shown in picture (10). In the coordinate dialog you can set the lower left and

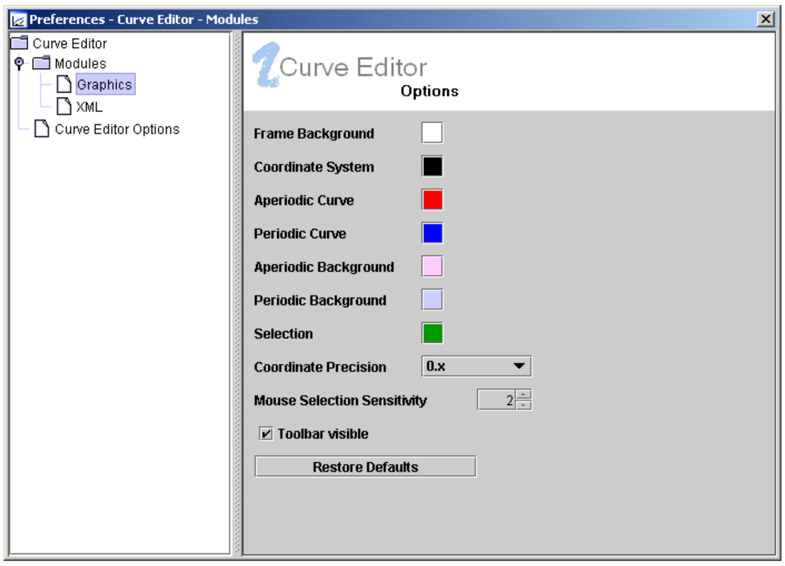
(10) Show the Coordinate Dialog

the upper right point of the coordinate system and zoom into this rectangle. If you select the *scale both axis in same ratio* box you have to specify the start point and the x coordinate of the end point. If the box is not selected you have to specify all four coordinates by yourself.

To make a screen shot of the current layout of the draw panel, click the screen shot button shown in picture (11).

(11) Export the draw panel view to an image

B.9.4 Options



Frame Background	⬜
Coordinate System	⬛
Aperiodic Curve	🟥
Periodic Curve	🟦
Aperiodic Background	🟪
Periodic Background	🟪
Selection	🟩
Coordinate Precision	0.x
Mouse Selection Sensitivity	2

☑ Toolbar visible

Figure B.10: The options dialog for the graphics module

In the options dialog you can choose the colors for the different regions of the graphical module. The coordinate precision field determines how exact user inputs in the draw panel will be handled. The segment vertices will have this coordinate precision.

B.10 XML Module

The XML module shows the active curve formatted in XML. You can edit the XML document and commit the changes. It will only be looked for changes in the text if the cursor changed its line position. At the bottom of the module you can see some parsing information. There are five types of information which are shown in the Figure B.11.

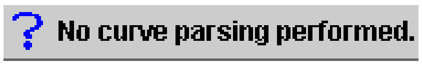
Fig B.11(a) is shown if the text had not been parsed. For example this is the case when no curve is loaded.

Fig B.11(b) is shown if the curve data has not changed and corresponds to the curve data loaded. This is the case when the curve is loaded.

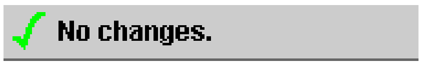
Fig B.11(c) will be shown if the text has been parsed and some valid changes occurred to the XML text. In this case you can press the key 'F1' to take over the curve changes.

Fig B.11(d) is shown if a key or a key value has invalid format.

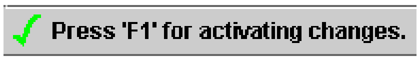
Fig B.11(e) will be shown if the XML document is not valid. This occurs e.g. if a key has no corresponding terminating element.


No curve parsing performed.

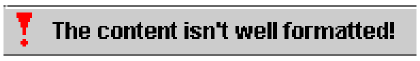
(a) No parsing


No changes.

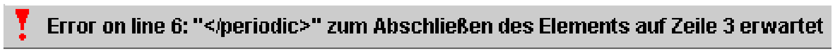
(b) No changes


Press 'F1' for activating changes.

(c) Valid changes


The content isn't well formatted!

(d) Wrong content


Error on line 6: "</periodic>" zum Abschließen des Elements auf Zeile 3 erwartet

(e) Parsing error

Figure B.11: The different parsing outputs

B.11 Tools Dialog

Select a method and the necessary number of curves for the arguments. Give the new curve a name. If the method will not return a curve object the new curve name will have no effects. Press the *execute* button.

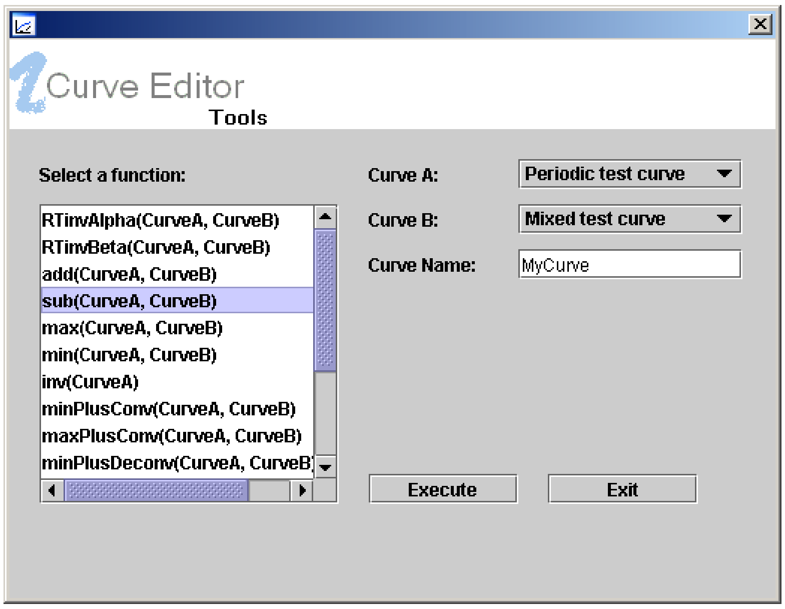


Figure B.12: The tools dialog

Appendix C

Framework API

The curve editor contains a lot of public methods. The methods which form the application program interface (API) for other software components are located in the class `CurveEditor` and are briefly described in this chapter.

static ExtendedCurve `getActiveCurve()` Returns a reference to the active curve. The active curve itself is a reference to a element of the curve set.

static void `setActiveCurve(int index)` Sets the reference of the active curve to a new element of the curve set.

static java.util.ArrayList `getCurveSet()` Returns an array list of the curve set.

static CurveEditor `getEditor()` Returns a reference to this class. Typical singleton style.

static void `registerCommitableObject(UpdatedCurvesCommitable)` Registers a class that implements the interface UpdatedCurvesCommitable.

static void `newEmptyCurve()` Clears the curve set and inserts an new empty curve.

public static void `addEmptyCurve()` Adds an empty curve to the curve set.

static void `addCurve(ExtendedCurve, boolean)` Adds a curve to the curve set. The boolean parameter indicates if the curve has unsaved attributes.

static void `addCurveSet(java.util.ArrayList, boolean)` Adds an array list of curves to the existing curve set. The boolean parameter indicates if the added curve set has unsaved attributes.

static void `openCurve(ExtendedCurve, boolean)` Opens a curve to the cleared curve set. The boolean parameter indicates if the curve has unsaved attributes.

static void `openCurveSet(java.util.ArrayList, boolean)` Clears the curve set and fills in the new curves. The boolean parameter indicates if the opened curve set has unsaved attributes.

static void `close(int indexToClose)` Removes the curve at indexToClose in the curve set.

static void `closeSelection(java.util.ArrayList curveSelection)` Removes every element in the curveSelection from the curve set if it exists.

static void `closeAll()` Clears the curve set and sets the active curve to null.

static void `curveSetHasBeenSaved(int[] indexes)` Has to be called if some curves has been saved.

static void `undoActiveCurve()` Undoes the last commited changes to active curve.

static void `redoActiveCurve()` Redoes the last undone changes.

static void `activeCurveDataHasChanged()` Call this if you have modified data of the active curve.

static boolean `hasUnsavedChanges()` Returns true if there are unsaved changes among the curves.

Appendix D

Framework Class Diagram

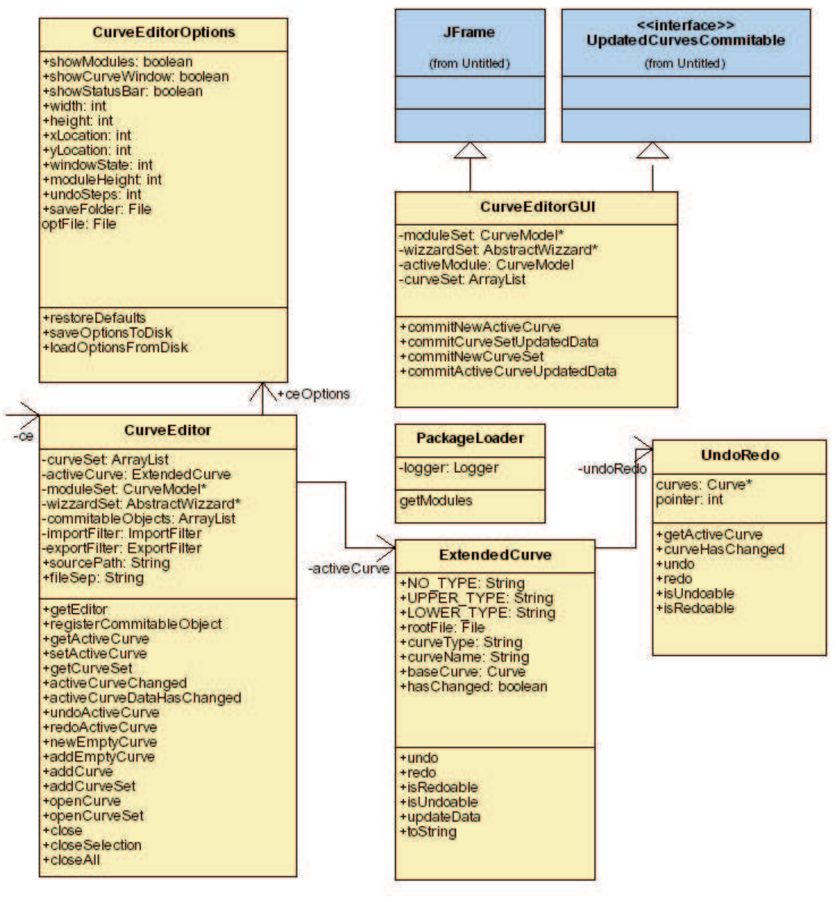


Figure D.1: Simplified class diagram of the framework

Appendix E

Known Bugs

This chapter presents a list of known bugs:

- Changes in the options dialog of the XML module become not visible until the active curve has changed.
- The undo / redo module registers changes on the curves which would not be necessary. E.g. to undo the shifting of a curve segment, more than one undo step is necessary. But changes on the curves will never get lost.
- Drawing periodic curve segments in the graphical module is recommended only in the first period. Otherwise correct handling is not guaranteed.

Bibliography

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

[2] Valerio Bürker and Roman Hiestand. Tool for performance analysis - design and integration in SymTA/S. Technical report, Institute TIK, ETHZ, 2004.

[3] G.C. Buttazzo. *Hard Real-Time Computing Systems: Predictable Scheduling Algorithms and Applications.* Kluwer Academic Publishers, Boston, 1997.

[4] S. Chakraborty, S. Künzli, and L. Thiele. A general framework for analysing system properties in platform-based embedded system designs. In *Proc. 6th Design, Automation and Test in Europe (DATE)*, pages 190–195, Munich, Germany, March 2003.

[5] Bruce Eckel. *Thinking in Java 3rd Edition.* Prentice Hall, 2005.

[6] Bruce Eckel. *Thinking in Patterns.* Prentice Hall, 2005.

[7] Rafik Henia, Arne Hamann, Marek Jersak, Razvan Racu, Kai Richter, and Rolf Ernst. System level performance analysis – the SymTA/S approach. *IEE Proceedings - Computers and Digital Techniques*, 152(02), March 2005.

[8] Mike Mason. *Pragmatic Version Control Using Subversion.* The Pragmatic Programmers LLC, 2005.

[9] Lothar Thiele and Ernesto Wandeler. Performance analysis of embedded systems. In *The Embedded Systems Handbook.* CRC Press, 2005.

[10] Lothar Thiele, Ernesto Wandeler, and Samarjit Chakraborty. Performance analysis of distributed embedded systems. *IEEE Signal Processing Magazine*, June 2005.