DIGITAL SYNTHESIS BY PLUG-IN METHOD IN JAVA MEDIA FRAMEWORK ENVIRONMENT

Jiri Schimmel

Department of Telecommunications
FEECS
Brno University of Technology
`schimmel@utko.fee.vutbr.cz`

Rostislav Fitz

Department of Computer Science and Engineering
FEECS
Brno University of Technology
`rfitz@asyc.cz`

ABSTRACT

This paper deals with the implementation of real-time digital musical sound synthesizers by the Plug-In method in the Sun Microsystems Java Media Framework environment. This environment use the Plug-In technology as well as the DirectX or VST environments, but the implementation methods are different.

1. JAVA MEDIA FRAMEWORK

Java is object-oriented multi-platform programming language developed by the Sun Microsystems company that is used mainly for Internet applet production.
The Java Media Framework (JMF) is an application programming interface (API) for incorporating time-based media into Java applications and applets. The JMF 1.0 API (the Java Media Player API) enabled programmers to develop Java programs that presented time-based media. The JMF 2.0 API extends the framework to provide support for capturing and storing media data, controlling the type of processing that is performed during playback, and performing custom processing on media data streams. In addition, JMF 2.0 defines a plug-in API that enables advanced developers and technology providers to more easily customize and extend JMF functionality.

1.1. High-Level Architecture

Devices such as tape decks and VCRs provide a familiar model for recording, processing, and presenting time-based media. When you play a movie using a VCR, you provide the media stream to the VCR by inserting a video tape. The VCR reads and interprets the data on the tape and sends appropriate signals to your television and speakers. JMF uses this same basic model. A data source encapsulates the media stream much like a video tape and a player provides processing and control mechanisms similar to a VCR. Playing and capturing audio and video with JMF requires the appropriate input and output devices such as microphones, cameras, speakers, and monitors.
A data source encapsulates the media stream much like a video tape and a player provides processing and control mechanisms similar to a VCR. Playing and capturing audio and video with JMF requires the appropriate input and output devices such as microphones, cameras, speakers, and monitors.
Data sources and players are integral parts of JMF's high-level API for managing the capture, presentation, and processing of time-based media. JMF also provides a lower-level API that supports the seamless integration of custom processing components and extensions. This layering provides Java developers with an easy-to-use API for incorporating time-based media into Java programs while maintaining the flexibility and extensibility required supporting advanced media applications and future media technologies.

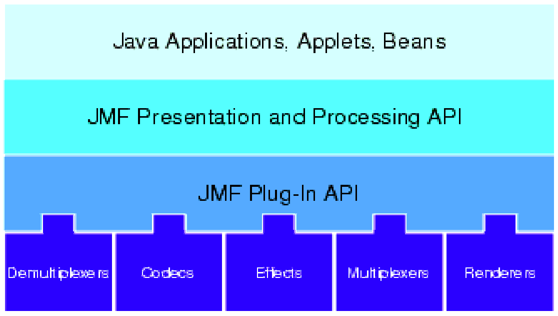

Java Applications, Applets, Beans
JMF Presentation and Processing API
JMF Plug-In API
Demultiplexers
Codecs
Effects
Multiplexers
Renderers

Figure 1. *High-level JMF architecture*

1.1.1. Managers

The JMF API consists mainly of interfaces that define the behaviour and interaction of objects used to capture, process, and present time-based media. Implementations of these interfaces operate within the structure of the framework. By using intermediary objects called *managers*, JMF makes it easy to integrate new implementations of key interfaces that can be used seamlessly with existing classes. JMF uses four managers: *Manager, PackageManager, CaptureDeviceManager*, and *PlugInManager*.
To write programs based on JMF, you'll need to use the *Manager create* methods to construct the *Players*, *Processors*, *DataSources*, and *DataSinks* for your application. If you're capturing media data from an input device, you'll use the *CaptureDeviceManager* to find out what devices are available and access information about them. If you're interested in controlling what processing is performed on the data, you might

also query the *PlugInManager* to determine what plug-ins have been registered.

1.1.2. Media Streams

A media stream is the media data obtained from a local file, acquired over the network, or captured from a camera or microphone. Media streams often contain multiple channels of data called tracks. For example, a Quicktime file might contain both an audio track and a video track. A track's type identifies the kind of data it contains, such as audio or video. The format of a track defines how the data for the track is structured. Media streams can be categorized according to how the data is delivered:

- *Pull* - data transfer is initiated and controlled from the client side.
- *Push* - the server initiates data transfer and controls the flow of data.

1.2. Media Processing and Presentation

Most time-based media is audio or video data that can be presented through output devices such as speakers and monitors. An output destination for media data is sometimes referred to as a data sink. In most instances, the data in a media stream is manipulated before it is presented to the user. The tracks are then delivered to the appropriate output device. If the media stream is to be stored instead of rendered to an output device, the processing stages might differ slightly.

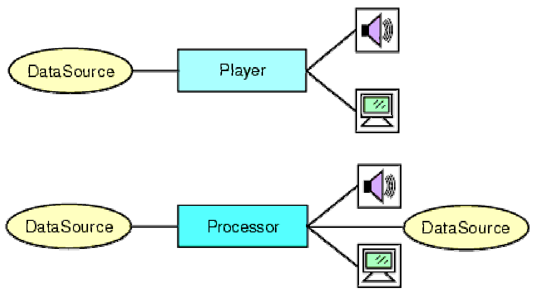


Figure 2. *JMF processor model and JMF player model*

In JMF, the presentation process is modelled by the *Controller interface*. *Controller* defines the basic state and control mechanism for an object that controls, presents, or captures time-based media. It defines the phases that a media controller goes through and provides a mechanism for controlling the transitions between those phases. The JMF API defines two types of *Controllers*: *Players* and *Processors*. They are constructed for a particular data source and are normally not re-used to present other media data.

- *Player* processes an input stream of media data and renders it at a precise time. A *DataSource* is used to deliver the input media-stream to the Player.
- *Processors* can also be used to present media data. A Processor is just a specialized type of *Player* that provides control over what processing is performed on the input media stream.

A *Processor* can send the output data to a presentation device or to a *DataSource*. If the data is sent to a *DataSource*, that *DataSource* can be used as the input to another *Player* or *Processor*, or as the input to a *DataSink*. While the processing performed by a *Player* is predefined by the implementor, a *Processor* allows the application developer to define the type of processing that is applied to the media data. This enables the application of effects, mixing, and compositing in real-time. The processing of the media data is split into several stages: demultiplexing, pre-processing, transcoding, post-processing, multiplexing and rendering (see Fig. 3).

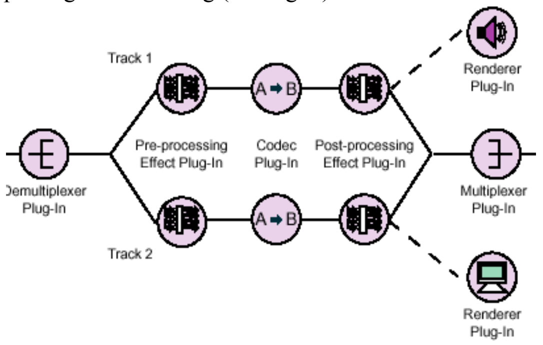


Figure 3. *Processor stages*

Demultiplexer extracts individual tracks of media data from a multiplexed media stream, *Mutliplexer* performs the opposite function, it takes individual tracks of media data and merges them into a single multiplexed media stream. *Codec* performs media-data compression and decompression. Each codec has certain input formats that it can handle and certain output formats that it can generate. *Effect* filter modifies the track data in some way. Effect filters are classified as either pre-processing effects or post-processing effects, depending on where they are applied. *Renderer* is an abstraction of a presentation device.
For more information about JMF 2.0 see [1].

2. ANALOG MODELLING SYNTHESIS

The synthesisers are electronic musical instruments that simulate sounds of acoustic musical instruments using various methods of sound synthesis or create wholly new sounds that we could not find in nature. Analog modelling synthesizers are digital electronic musical instruments that simulate vintage analogue synthesisers that have been the first electronic instruments. These instruments consist of particular blocks whose mutual connections and parameters adjusting, which generate various sounds, are user-controlled. The sound generator of analog synthesizers combines the additive and the subtractive synthesis with the PWM, RM, and FM modulation synthesis (see [3] for details).

2.1. Analog Modelling Synthesizer Architecture

Every analog modelling synthesizer consists of 4 blocks: sound generator, effect processor, control logic, and MIDI interface. MIDI (Musical Instruments Digital Interface) is proposed for communication between digital musical instruments. It transfers all information that describes sound. The synthesizer control logic generates control signals for all the other synthesizer units according to data from the MIDI interface.

2.2. Additive Synthesizer

The additive synthesizer consists of an even number of *Voltage-Controlled Oscillators* VCO (most often two or four). The oscillator pairs function as input signals of the RM module and at the same time they can be switched to the FM mode and create one FM operator (see Fig. 4). The voltage-controlled oscillator inputs are signals that determine the oscillator frequency, i.e. tone pitch and pulse width when generating the square signal for the pulse width modulation (PWM). The outputs are signals of both oscillators and the ring modulator output signal.

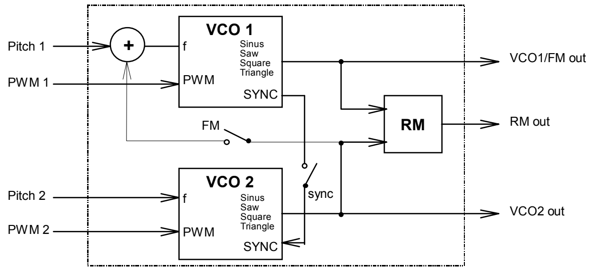


Figure 4. *Structure of a block of synthesizer oscillators*

The oscillators generate basic signal waveforms, such as the sinus, square, saw, and triangle signal in the acoustic signal frequency range. The Simulink model of that oscillator is in Fig. 5. The Trigger signal, signal for frequency control, and signal for the control of the width of square signal (pulse width modulation) are the inputs.

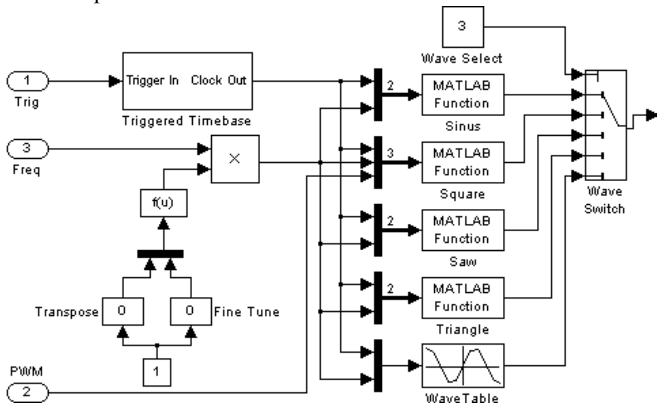


Figure 5. *Simulink model of sound generator oscillator*

The oscillator parameters are the *transposition* and the *fine tune*. They are designed for mutual harmonic and non-harmonic de-tuning of oscillators. The transposition sets harmonic de-tuning in the +/-24 semitone range, the fine tune sets non-harmonic de-tuning in the +/-50 cent range. Since 1 cent = 1/100 of semitone, the following equation holds for the resulting frequency of the oscillator

$$f = f_0 . \sqrt[12/s]{2} . \sqrt[12/c]{2} \,/\, 100^{c} = f_0 \cdot \frac{2^{\frac{s+c}{12}}}{100^{c}} \qquad (1)$$

where f_0 is the original frequency, s is the transposition size in semitone, and c is fine tune size in cent. If the synthesizer consists of two oscillators only, transposition and fine tune settings with one oscillator only is sufficient.

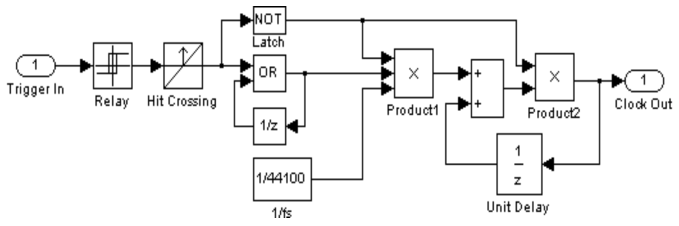


Figure 6. *Simulink model of time base for sound oscillators*

The time behaviours of generators are time-dependent, therefore they need *time bases*. Time base generates a linear time vector with steps of $1/f_S$. Function generators need a triggered time base that generates the time vector in the required time interval only. Sound generators need special type of time base, because the *Trigger* signal (see Fig. 5), which is determined from key playing, ends after the *Note Off* MIDI event is received, i.e. after the key is released. But the oscillators have to generate sound also hereafter because the sound still runs in the *Release* period. The time base according to Fig. 6 is suitable for these purposes because it runs also after the impulse on the *Trigger* input terminates and it resets itself with the rising edge, i.e. when a new tone is played.

2.3. Subtractive Synthesizer

The subtractive synthesizer block follows this amplifier. It is a bank of parametric, *Voltage-Controlled Filters* (VCF). These filters are connected in series or in parallel. The most often used filters are the low-pass and the high-pass filter with adjustable resonance, and the band-pass and the band-stop filter with adjustable bandwidth.

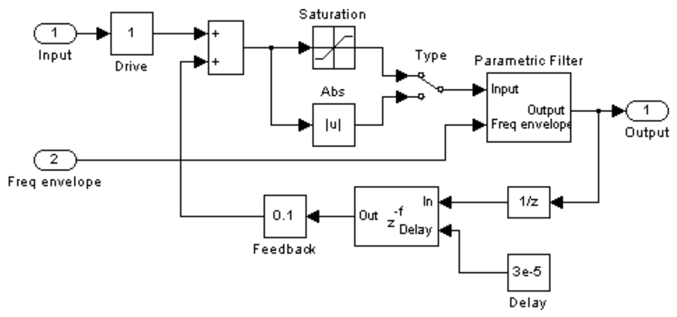


Figure 7. *Parametric filter in circuit with distortion*

Signal is admitted into the time-varying IIR filter. The coefficients of this filter are computed on the basis of slider settings that represent individual filter parameters. Various types of filters differ only in their transfer function, whose numerator

and denominator are composed according to the three parameters that correspond to the frequency, gain, and bandwidth/slope settings. More information about parametric filters and their transfer functions can be found in [2] and [4].
To producing further sound colours some synthesizers use a filter in circuit with distortion with feedback, similar to guitar distortion [2][5]. The Simulink model of that filter is in Fig. 7.
Two distortion types are used: limitation and rectification of signal. The number of filter parameters increases by two, *Drive* and *Feedback*, which set the distortion amount. This expands the synthesizer sound capability and the results are very interesting.

2.4. Analog Modelling Synthesizer Model

The Simulink model of a simple analog modelling synthesizer is in Fig. 8. The synthesizer is single-channel and monophonic. It contains two oscillators with ring modulator and noise generator, one low-frequency oscillator (LFO), which controls oscillator pulse width modulation, two ADSR envelope generators (AEG and FEG), one parametric filter with distortion, and modulation effect. The *MIDI source* block, which is the model of a very simple sequencer, works as a MIDI note source. It converts information about the beginning, length, number, and *Velocity* of note to MIDI events, which it sends at the moment of note beginning and note end. The LFO, AEG, and FEG blocks are described in detail in [2].

3. DIGITAL SYNTHESIZER IMPLEMENTATION IN JMF ENVIRONMENT

3.1. Implementing JMF Plug-Ins

If you extend JMF functionality by implementing a new plug-in, you can register it with the *PlugInManager* to make it available to *Processors* that support the plug-in API. To use a custom *Player, Processor, DataSource*, or *DataSink* with JMF, you register your unique package prefix with the *PackageManager*.
When implement digital musical effects we need to create the custom *Effect Plug-In*. When we implement digital synthesis algorithms, we need to create the custom *Effect Plug-In* as well as the custom *DataSource* and custom *Multiplexer* Plug-In.

3.1.1. Implementing Protocol Data Source

A *DataSource* is an abstraction of a media protocol-handler. You can implement new types of *DataSources* to support additional protocols by extending *PullDataSource*, *PullBufferDataSource*, *PushDataSource*, or *PushBufferDataSource*. If you implement a custom *DataSource*, you can implement *Demultiplexer* and *Multiplexer* plug-ins that work with your custom *DataSource* to enable playback through an existing Processor, or you can implement a completely custom *MediaHandler* for your *DataSource*.
A *DataSource* manages a collection of *SourceStreams* of the corresponding type. When you implement a new *DataSource*, you also need to implement the corresponding source stream: *PullSourceStream, PullBufferStream, PushSourceStream, or PushBufferStream*.
So that the *Manager* can construct your custom *DataSource*, the name and package hierarchy for the *DataSource* must follow certain conventions. The fully qualified name of your custom *DataSource* should be:

```
<protocol package-prefix>.media.protocol.<protocol>.DataSource
```

The *protocol package-prefix* is a unique identifier for your code that you register with the JMF *PackageManager* as a protocol package-prefix. The protocol identifies the protocol for your new *DataSource*.

3.1.2. Implementing Effect Plug-In

An *Effect* plug-in is actually a specialized type of *Codec* that performs some processing on the input *Track*. *Effect* is a single-input, single-output processing component and the data manipulation that the *Effect* performs is implemented in the *Process* method. When you implement an *Effect*, you need to:

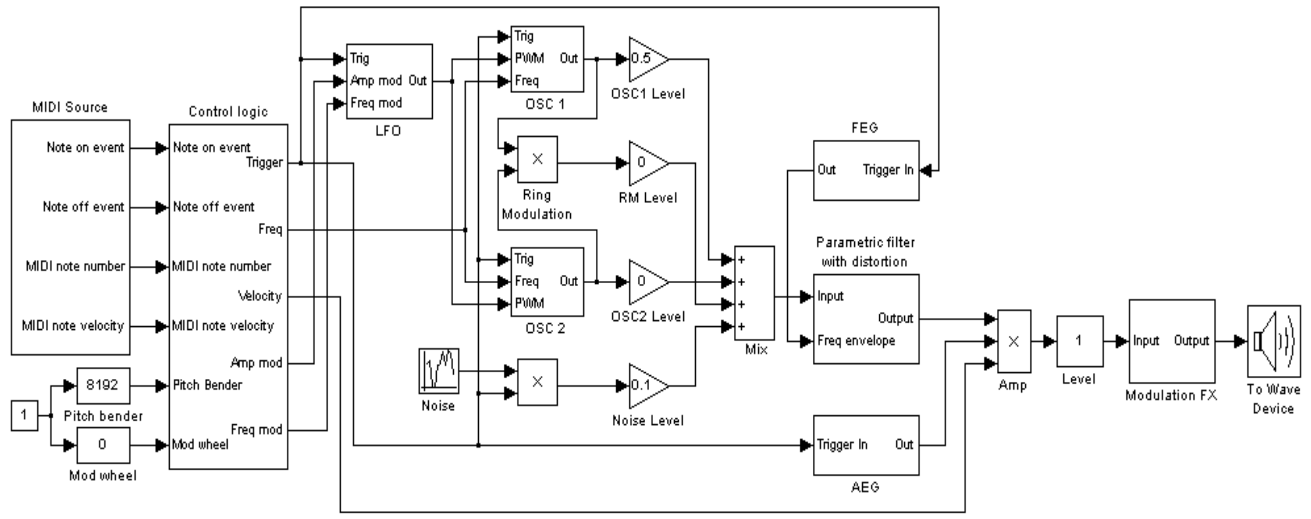


Figure 8. *Simulink model of analog modelling synthesizer*

- Implement `getSupportedInputFormats` and `getSupportedOutputFormats` to advertise what input and output formats the effect supports.
- Enable the selection of those formats by implementing `setInputFormat` and `setOutputFormat`.
- Implement `process` to actually perform the effect processing.

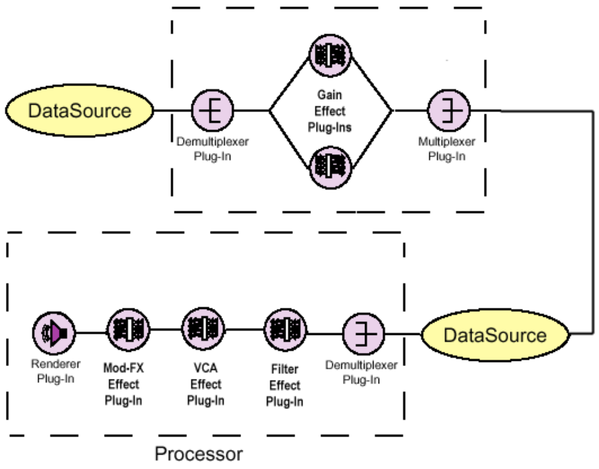


Figure 9. *JMF model of digital synthesizer according to Fig. 8*

3.1.3. *Implementing Multiplexer Plug-In*

A *Multiplexer* takes individual tracks of media data and merges them into a single multiplexed media-stream. A *Multiplexer* is a multi-input, single-output processing component. It reads data from a set of tracks and outputs a *DataSource*. The main work performed by a *Multiplexer* is done in the implementation of the *process* method. The *getDataSource* method returns the *DataSource* generated by the *Multiplexer*. When you implement a *Multiplexer*, you need to:

- Implement `getSupportedOutputContentDescriptors` to advertise what output formats the *Multiplexer* supports.
- Enable the selection of the output format by implementing `setOutputContentDescriptor`.
- Implement `process` to actually merge the individual tracks into an output stream of the selected format.

3.2. Implementing Digital Synthesizer using JMF Plug-Ins

When realizing digital synthesizer in the JMF environment we divided the model according to Fig. 9 into three parts: the oscillators and LFO part, the mixer part, and the filter/VCA/effect processor part. The oscillators are actually signal sources, i.e. a data stream. That is why we used a *PullBufferStream* interface when we implemented these oscillators and the LFO.

```
public class GeneratorPullBufferStream implements
PullBufferStream
```

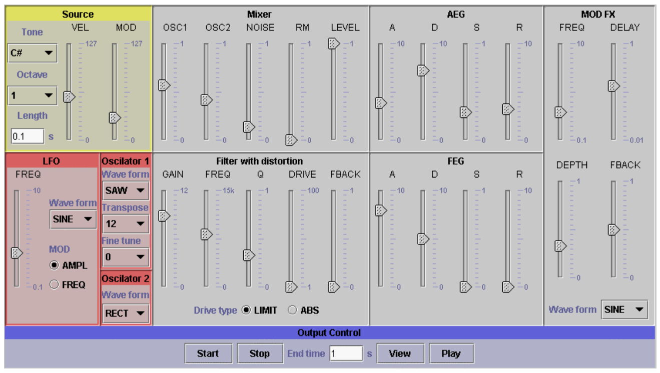
A *Read* method is implemented in this class, which is called when data are read from the stream. Within this method performs signal generation is performed.

```
public void read(Buffer buffer)
```

Streams also contain information about the data format. The mixer block is formed by the *Processor*, whose input is a *GeneratorDataSource* data source. This class extends the *PullBufferDatSource*

```
class GeneratorDataSource extends PullBufferDataSource
```

A *GetStreams* method is implemented in this class, which refers to *GeneratorPullBufferStream* objects. The *Processor* contains a demultiplexer, two effects plug-Ins, and a multiplexer. The *Processor* selects the demultiplexer according to the input streams from the *DataSource*. The *Demultiplexer* divides the input stream into two tracks that contain output signals of both generators from *GeneratorPullBufferStream*. The *Effect Plug-Ins* only adjusts the amplitude of these signals. The output multiplexer merges both tracks and sums them with the noise generator and RM outputs.
The other parts of the synthesizer model according to Fig. 9 are parts of the other *Processor*. A demultiplexer is on its input again, this time with one output track. Individual blocks are realized via the effect plug-ins while a *Renderer* plug-in is on the output. We obtain a suitable *Renderer* for the given data format (if it is registered) from the *Plug-In Manager*. It can be a *JavaSoundRenderer* for raw audio data, which is part of the com.sun.media package.



Tone: C#
Octave: 1
Length: 0.1 s
LFO Wave form: SINE
MOD: [✓ AMPL] FREQ
Oscilator 1 Wave form: SAW
Transpose: 12
Fine tune: 0
Oscilator 2 Wave form: RECT
Drive type: [✓ LIMIT] ABS
Wave form: SINE
End time: 1 s

Figure 10. *Graphic user interface of the JMF digital synthesizer*

REFERENCES

[1] Java Media Framework Application Programming Interface Guide. *Sun Microsystems, Inc. 1999.*

[2] Schimmel, J., Fitz, R., and Oboril, D. "Digital Synthesis and Processing by Plug-In Method in VST and Java Media Framework Environments". Student Member Scholarly Works at the 110th International Convention of The AES, Student Delegate Assembly, at Amsterdam, 2001.

[3] Schimmel, J. "Modern Types of Analog Modelling Synthesizers". *Proceedings of ATP 2000 conference, Brno, 2000. (in Czech)*

[4] Smekal, Z. et al., Partial research report on the solution of international project No OC G6 10 for the year 1999, 1999.

[5] Smekal, Z. et al., Partial research report on the solution of international project No OC G6 10 for the year 2000, 2000.