Changes in the *Ptolemy II* source code

TECHNISCHE
UNIVERSITÄT
DARMSTADT

I made several changes to the source code of *Ptolemy II*. A major improvement is scriptability — the ability to control *Ptolemy II* through a script. Furthermore, I added several actors to *Ptolemy II*'s actor library and fixed some minor bugs.

1 Batch Processing

Jacl [DeJ08] is an interpreter for the scripting language Tcl. Jacl itself is written in Java. Jacl is open source, which means it can be enhanced by custom commands and integrated into other programs. Using Jacl, I created a simple program named PtolemyBatchApplication, which expects a file name as its command line parameter. The program then opens the specified file and tries to execute the file's content as a Tcl script. The interpreter recognizes the following commands in addition to the usual features of Tcl:

`ptloadmodel` loads a *Ptolemy II* model from a file,

`ptgetparameter` reads the value of a model parameter,

`ptsetparameter` sets the value of a model parameter,

`ptrunmodel` starts the execution of a *Ptolemy II* model and

`ptgenerate` creates a given number of instances of a given actor.

Furthermore, the interpreter supports a new data type named TclPtolemyActor, which is a reference to a *Ptolemy II* actor or an entire model (which is only a special case of a composite actor).

1.1 The `ptloadmodel` Command

The ptloadmodel command expects a character string parameter which contains a file name. The command opens the specified file and interprets its contents as a MoML document (*Model Markup Language*, a markup language for *Ptolemy II* models). An error message is produced if the command fails. The command returns a reference to the model.

1.2 The `ptgetparameter` Command

The ptgetparameter command expects two parameters:

1. A reference to a *Ptolemy II* model and
2. a character string containing the qualified name of a parameter within the model.

The qualified name of a parameter consists of the names of all containing actors, starting at the highest level of the hierarchy, and the name of the parameter itself, separated by dots (`.`)[1].

The command searches the given model for a parameter with the given qualified name. It then returns the parameter's value as a character string. An error message is produced if the command fails.

1.3 The `ptsetparameter` Command

The ptsetparameter command expects three parameters:

1. A reference to a *Ptolemy II* model,
2. a character string containing the qualified name of a parameter within the model and
3. another character string containing the expression to be assigned to that parameter.

The command searches the given model for a parameter with the given qualified name. It then sets the parameter's expression accordingly. An error message is produced if the command fails.

The third parameter may contain arbitrary expressions; it is not limited to constant values. When setting the value of a string parameter, the expression must start and end with double quotes (`"`). When setting the value of an array parameter, the expression must start and end with curly braces (`{`, `}`) and the array elements must be separated by commas (`,`).

The command does not return anything.

[1] Example: The qualified name of the parameter `factor` of the actor `Scale`, which is contained in the composite actor `Cavities`, is `Cavities.Scale.factor`

1.4 The `ptrunmodel` Command

The ptrunmodel command expects a reference to a *Ptolemy II* model as a parameter: The command begins the execution of the given model and blocks until the simulation has finished. Different error messages are produced if the simulation cannot be started or is terminated prematurely.

The command does not return anything.

1.5 The `ptgenerate` Command

The ptrunmodel command expects three parameters:

1. A reference to a *Ptolemy II* model,
2. a character string containing the qualified name of an actor within the model and
3. an integer number ≥ 1 specifying how many instances of the given actors shall be created.

The qualified name of an actor consists of the names of all containing actors, starting at the highest level of the hierarchy, and the name of the actor itself, separated by dots (`.`)[2].

The command searches the given model for an actor with the given qualified name. It then creates the given number of instances of that actor and connects the ports of the created instances with the same relations as the corresponding ports of the original. An error message is produced if the command fails. The command returns a list containing the names of the created instances.

The name `ptgenerate` is inspired by the generate keyword of VHDL, which creates a number of concurrent instances of a given component.

By using this command, a deficiency in *Ptolemy II* version 7 can be overcome which prevents the dynamic instantiation of continuous-time actors (MultiInstanceComposites do not work with continous-time actors, see section 3.2).

2 New Actors

With the help of several students, I created a number of new actors:

`DeadTime`

`CSVRecordReader`

`CSVRecordWriter`

`TappedCircularBuffer`

`CyclicIntegrator`

`AntiWindUpIntegrator`

These actors are discussed in the following.

2.1 The `DeadTime` Actor

This class of actors, which has been developed by Tao Guo in his diploma thesis [Guo08], represents continuous-time dead-time elements in *Ptolemy II*.

This class of actors has two parameters:

`initialState` The initial state of the output (a real number)

`deadTime` The dead-time (a positive, real number)

[2] Example: The qualified name of the actor `Scale`, which is contained in the composite actor `Cavities`, is `Cavities.Scale`

The input $x_I(t)$ at time t becomes the output $x_O(t+T_D)$ at time $t+T_D$, where T_D is the constant dead-time. The actor calls the `fireAt` method of the `Director` class in the `ptolemy.actor` package to ensure that the continuous-time model is re-evaluated at $t+T_D$. If t is a node of the numeric simulation, then $t+T_D$ is also a node. However, in a heterogeneous model, events which require the continuous-time differential equations to be re-evaluated can occur at any time. In that case, a new node may be created at time t even though there is no node at $t-T_D$. In order to be able to calculate the output $x_O(t)$ for any arbitrary t, it may therefore be necessary to interpolate between neighboring input values.

The implementation is documented in Tao Guo's diploma thesis [Guo08].

2.2 The `CSVRecordReader` Actor

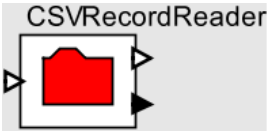

Figure 1: Vergil Icon of a CSV Record Reader

This class of actors reads data in CSV format [RFC4180] and provide these data to other actors via their output port. Another output port indicates whether further data is available or not. The file to be read is specified via a parameter. Figure 1 shows how an actor of this class is displayed in Vergil.

The entire file is read into memory at the start of the simulation in order to avoid file operations during run-time, which may slow down the simulation.

2.2.1 Ports

This class of actors has three ports:

`trigger` Whenever any token arrives at this input port, the next record is read from the file.

`output` The fields of each record are output via this output multiport whenever a token arrives at the `trigger` port. The first channel of the output contains data from the first column, the second channel contains data from the second column, and so on.

`endOfFile` A Boolean token indicating whether the end of the input file has been reached is output via this output port whenever a token arrives at the `trigger` port.

2.2.2 Parameters

This class of actors has two parameters:

`fileOrURL` This string parameter contains the path of the input file.

`numberOfLinesToSkip` This non-negative integer parameter specifies the number of lines at the top of the file which shall be skipped.

2.2.3 Source Code

The source code of this class can be found in the directory `ptolemy/actor/lib/io`. The class uses a library for handling data in CSV format which is described in section 4.

2.3 The `CSVRecordWriter` Actor

This class of actors writes data in CSV format [RFC4180]. The name of the file to be written as well as the access mode are specified using parameters. Figure 2 shows how an actor of this class is displayed in Vergil.

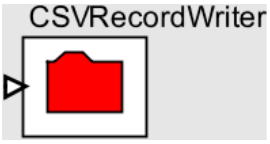

Figure 2: Vergil Icon of a CSV Record Writer

2.3.1 Ports

This class of actors has one input multiport named `input`. As soon as any tokens arrive at this port, they are being converted to character strings and a record is formed in which the token from the first channel becomes the first column, the token from the second channel becomes the second column, and so on. The actor buffers records in memory and writes them to a file from time to time because writing larger blocks of data is more efficient. The file is created if it does not yet exist at the start of the simulation. Any records remaining are flushed at the end of the simulation and the file is closed afterwards.

2.3.2 Parameters

This class of actors has four parameters:

`fileOrURL` This string parameter contains the path of the file to be written. The path may be set to `/dev/null` in order to discard output without removing the actor from the model.

`headlines` This string parameter contains a header line which will become the first line in the file.

`append` This Boolean parameter indicates what to do if the file already exists. If it is set to true, data will be appended to the end of the file; if it is set to false, the file is truncated.

`confirmOverwrite` This Boolean parameter indicates what to do if the file already exists. If it is set to true, the user will be asked for a confirmation before the file is truncated. If the output file is `stdout` or `/dev/null`, no confirmation is required.

2.3.3 Source Code

The source code of this class can be found in the directory `ptolemy/actor/lib/io`. The class uses a library for handling data in CSV format which is described in section 4.

2.4 The `TappedCircularBuffer` Actor

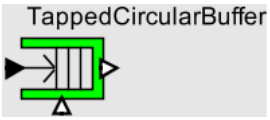

Figure 3: Vergil Icon of a Tapped Circular Buffer

This class of actors represents circular buffers with intermediate taps; they can be used to model digital filters with sparse coefficient vectors (for more details on these filters and their applications, see [Han06; The07; PGK10; Sur+11; Sam+11; Kli+07]). Figure 3 shows how an actor of this class is displayed in Vergil.

2.4.1 Ports

This class of actors has three ports:

`input` Data arriving at this input port are written to the head of the circular buffer. The oldest value is removed from the tail of the buffer.

`trigger` As soon as an integer token arrives on any channel of this port, it is used as an index into the buffer and a token containing the value at that index is produced at the `output` port. Index 0 corresponds to the most recent value, index 1 to the second most recent value, and so on.

`output` As soon as an index token arrives at the `trigger` port, the selected value is output on this port.

2.4.2 Parameters

This class of actors has two parameters:

`defaultValue` This parameter specifies the value with which the buffer is filled initially.

`capacity` This parameter, which must be a positive integer number, defines the size of the buffer.

2.4.3 Source Code

The source code of this class can be found in the directory `ptolemy/domains/de/lib`.

2.5 The `CyclicIntegrator` Actor

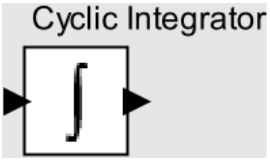

Figure 4: Vergil Icon of a Cyclic Integrator

This class of actors represents integrators whose output wraps around, which may be useful, for instance, when integrating frequencies or frequence differences to obtain phases or phase differences. In that case, the output is limited to $[-\pi, +\pi]$ (or $[0, 360^\circ]$). By computing the remainder of the division by $2 \cdot \pi$, for example, the set of real numbers $\mathbb{R}$ can be mapped to the interval $[0; 2\pi)$. Figure 4 shows how an actor of this class is displayed in Vergil.

This class of actors has three parameters:

`initialState` The initial state of the actor (a real number)

`lowerBound` The lower bound of the state (a real number); the state wraps around to the upper bound if it becomes smaller than the lower bound

`upperBound` The upper bound of the state (a real number); the state wraps around to the lower bound if it becomes smaller than the lower bound

If the lower bound is set to $-\infty$ and the upper bound to $+\infty$, this actor behaves like an ordinary integrator.

The source code of this class can be found in the directory `ptolemy/domains/ct/lib`.

2.6 The `AntiWindUpIntegrator` Actor

Figure 5: Vergil Icon of an Anti-Wind-Up Integrator

This class of actors represents integrators whose output saturates, which may be useful when modeling, for instance, PID controllers with output constraints. The output value of the integrator saturates at a given upper bound and does not increase further even if the input is positive. Likewise, the output saturates at a given lower bound and does not decrease further even if the input is negative. Figure 5 shows how an actor of this class is displayed in Vergil.

This class of actors has three parameters:

`initialState` The initial state of the actor (a real number)

`lowerBound` The lower bound (a real number) at which the actor saturates

`upperBound` The upper bound (a real number) at which the actor saturates

If the lower bound is set to $-\infty$ and the upper bound to $+\infty$, this actor behaves like an ordinary integrator.

The source code of this class can be found in the directory `ptolemy/domains/ct/lib`.

3 Bugfixes

Working with *Ptolemy II* version 7.0.1, I noticed four annoying bugs:

- The simulation step size shrinks continously during simulation, until the simulation nearly freezes.
- MultiInstanceComposites do not work with continuous-time models.
- `FirstOrderHold` elements do not behave identically to a combination of a `ZeroOrderHold` and an integrator.
- When using transfer functions in the Laplace domain, the simulation sporadically terminates with the error message "Graph is cyclic".

3.1 Shrinking Step Size

Even if a fixed-step solver is chosen in *Ptolemy II*, the simulation step size is not constant. The reason for that is that asynchronous events may occur at any point in time in between two steps. In that case, the step size is reduced appropriately and the model is re-evaluated at the time of the asynchronous event.

However, the step size was not returned to its former value afterwards, but the new, reduced step size was used instead. Sooner or later, another asynchronous event would lead to the step size being reduced again, so that the step size keeps shrinking during the course of the simulation. In extreme cases, the simulation would grind to a halt.

My solution was to change the return value of the method `integratorPredictedStepSize` of the `FixedStepSolver` class in the package `ptolemy.domains.ct.kernel.solver`:

Listing 1: Original Source Code of the `integratorPredictedStepSize` Method

```
120 public final double integratorPredictedStepSize(CTBaseIntegrator integrator) {
121   CTDirector director = (CTDirector) getContainer();
122   return director.getCurrentStepSize();
123 }
```

Listing 2: Modified Source Code of the `integratorPredictedStepSize` Method

```
120 public final double integratorPredictedStepSize(CTBaseIntegrator integrator) {
121   CTDirector director = (CTDirector) getContainer();
122   return director.getInitialStepSize();
123 }
```

This way, the step size keeps being reset to its initial value.

3.2 Dynamic Instantiation of Continuous-Time Submodels

The `MultiInstanceComposite` construct allows creating arbitrarily many instances of a given sub-model. A `MultiInstanceComposite` actor is a container for a sub-model and has a parameter `nInstances` which specifies how often this sub-model shall be instantiated. The sub-model must have a director of its own.

In *Ptolemy II* version 7.0.1, it is not possible to instantiate continuous-time sub-models this way, for the following reason: Two different directors, `CTDirector` and `CTEmbeddedDirector`, exist for continuous-time models. The former can only be used at the top level, and the latter is intended to be used within continuous-time sub-models inside other domains. Since there already is a `CTDirector` at the top level, it is not possible to use another `CTDirector` inside the `MultiInstanceComposite`. Using a `CTEmbeddedDirector` is not possible either since that director expects its container to implement the `CTStepSizeControlActor` interface, but the `MultiInstanceComposite` class does not implement that interface.

The problem no longer exists in *Ptolemy II* version 8, (available since October 2010), in which the CT domain has been replaced by the Continuous domain. Edward A. Lee wrote in the ptolemy-hackers mailing list (`ptolemy-hackers@lists.eecs.berkeley.edu`) on April 18th, 2009:

> The solution for this is to use the ContinuousDirector rather than CTDirector. However, this domain is not quite finished yet. The actor library is incomplete, for example. It has also not been thoroughly tested, and the current scheduler is probably somewhat inefficient.

Basically, CTDirector taught us quite a bit, including how to implement composable continuous-time models correctly. The approach is documented here:
`http://ptolemy.eecs.berkeley.edu/publications/papers/07/unifying/index.htm`
I hope we can finish this within the next few months...

A possible workaround is to use the scripting feature described in section 1 and in particular the `ptgenerate` command (see section 1.5) [SGK10].

3.3 First-Order Hold Elements

A `ZeroOrderHold` element has an input x_I and an output x_O which remains constant after a token has arrived at the input until the next token arrives. A `FirstOrderHold` element has two inputs x_{I1} and x_{I2} and its output x_A linearly increases at a rate of x_{I2}, starting from x_{I1}, after tokens have arrived at the inputs. Let t_1 and t_2 be two points in time at which successive tokens arrive at the inputs. For the `ZeroOrderHold`,

$$x_O(t) = x_I(t_1),\ t_1 \leq t < t_2, \tag{1}$$

and for the `FirstOrderHold`,

$$x_O(t) = x_{I1}(t_1) + x_{I2}(t_1) \cdot (t - t_1),\ t_1 \leq t < t_2. \tag{2}$$

The output $x_O(t)$ of a `FirstOrderHold` is continuous at time t_2 iff $x_{I1}(t_2) = x_{I1}(t_1) + x_{I2}(t_1) \cdot (t_2 - t_1)$. In that case, a `FirstOrderHold` should behave identically to a `ZeroOrderHold` followed by an integrator.

Apparently, this is not always the case, and the output of an integrator following a `ZeroOrderHold` is sometimes different from the output of a corresponding `FirstOrderHold`. The reason for that is probably that the `FirstOrderHold` class implements the `CTWaveformGenerator` interface, but the `Integrator` class implements the `CTDynamicActor` interface, and the director calls the different actors in different phases of the simulation.

Therefore, I prefer to use `ZeroOrderHold` elements followed by an integrator instead of `FirstOrderHolds`, which is of course only possible as long as there are no discontinuities in the signals in question.

3.4 "Graph is cyclic"

When using `ContinuousTransferFunctions`, the simulation is randomly terminated with the error message "Graph is cyclic" if an explicit solver is being used. The reason for that is that *Ptolemy II* builds a linear sub-model comprising `Add`, `Integrator` and `Scale` elements at run-time. In some cases, `Scale` elements with zero gain are being instantiated in the process, and *Ptolemy II* is not able to remove these from the model. If these elements are on the feedback path, *Ptolemy II* detects an algebraic loop, which an explicit solver cannot resolve by itself.

4 Handling of Data in CSV Format

In the context of a cooperation with the GSI Helmholtz Center for Heavy-Ion Research (GSI), I created a *Java* library to handle data in CSV format [RFC4180]. The GSI aims at standardizing their internal data formats used in RF control based on that format [KZ11]. Fig. 6 shows the class diagram of that library. The source code of this library can be found in the directory `gsi/rf/data`.

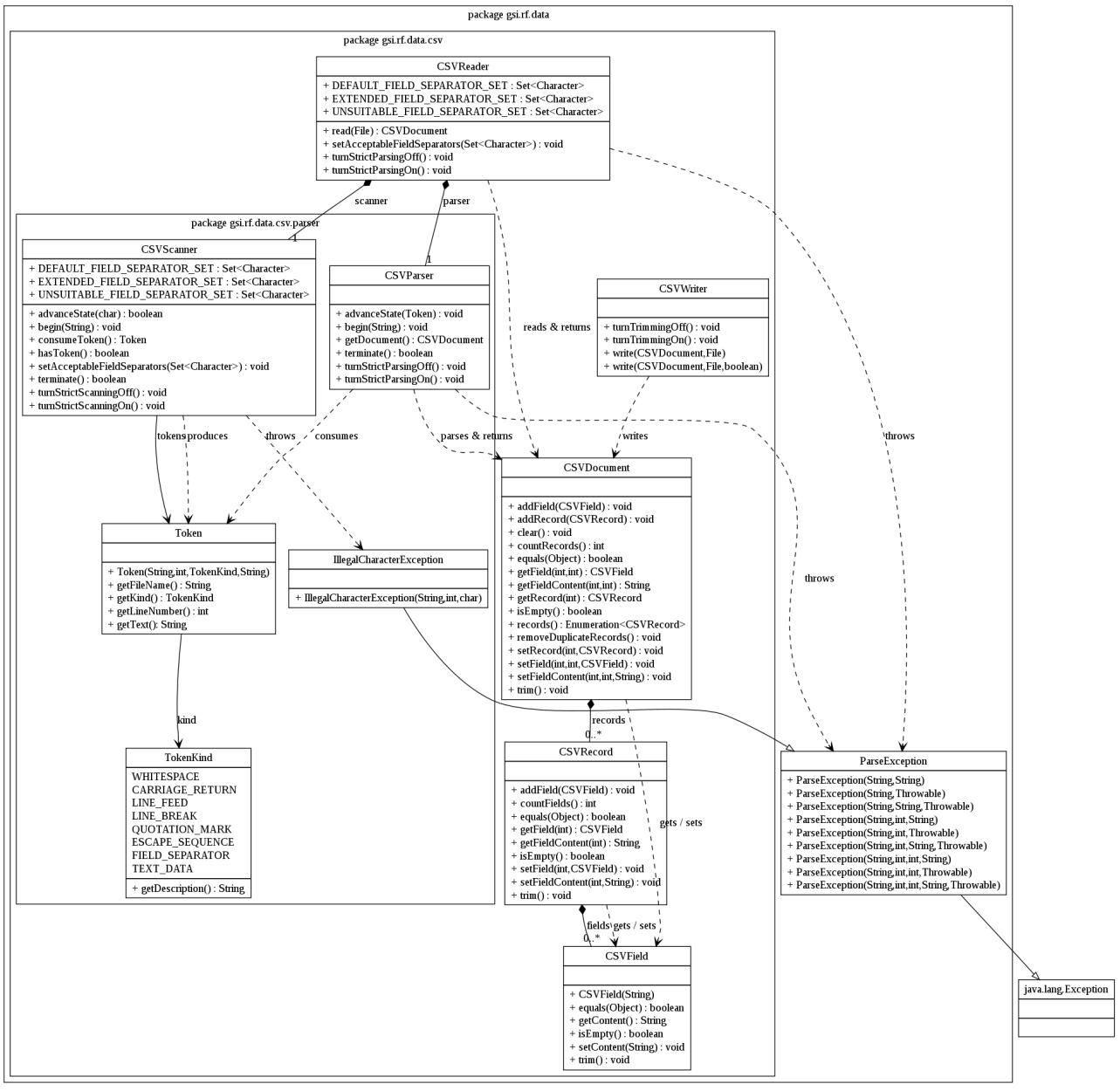

Figure 6: UML Class Diagramm for Handling of Data in CSV Format

Bibliography

[DeJ08] Mo DeJong. The Tcl/Java Project. 2008. URL: http://tcljava.sourceforge.net.

[Guo08] Tao Guo. “Analyse und Implementierung eines Lösungsverfahrens für Differentialgleichungen mit Totzeit in Ptolemy II”. Diploma Thesis. Technische Universität Darmstadt, Institute of Microelectronic Systems, 2008.

[Han06] Liang Han. “Reconfigurable FIR Filter for Large Tap Distances and Low Tap Counts”. Bachelor Thesis. Technische Universität Darmstadt, Institute of Microelectronic Systems, 2006.

[Kli+07] Harald Klingbeil, Bernhard Zipfel, Martin Kumm, and Peter Moritz. “A Digital Beam-Phase Control System for Heavy-Ion Synchrotrons”. In: IEEE Transactions on Nuclear Science 54.6 (2007), pp. 2604–2610.

[KZ11] Harald Klingbeil and Bernhard Zipfel. “Data Analysis File Formats for RF Applications”. Version 0.80. May 2011.

[PGK10] Surapong Pongyupinpanich, Manfred Glesner, and Harald Klingbeil. “Implementation of Realtime Pipeline-Folding 64-Tap Filters on FPGA”. In: Proceedings of the Conference on Ph. D. Research in Microelectronics and Electronics. 2010, pp. 1–4.

[RFC4180] Internet Engineering Task Force, ed. Common Format and MIME Type for Comma-Separated Values (CSV) Files. Request for Comments 4180 (2005).

[Sam+11] Faizal Arya Samman, Pongyupinpanich Surapong, Christopher Spies, and Manfred Glesner. “Floating-point-based hardware accelerator of a beam phase-magnitude detector and filter for a beam phase control system in a heavy-ion synchrotron application”. In: Proceedings of the International Conference on Accelerators and Large Experimental Physics Control Systems. 2011, pp. 683–686.

[SGK10] Christopher Spies, Manfred Glesner, and Harald Klingbeil. “Statusbericht zum GSI-Projekt ‘Systemmodellierung und Vernetzung der digitalen Mehr-Kavitäten-Regelung für das FAIR-Projekt’ — A Standard Scenario for Future Design Space Exploration”. Berichtszeitraum Januar bis Juni 2010. Dec. 2010.

[Sur+11] Pongyupinpanich Surapong, Christopher Spies, Manfred Glesner, and Harald Klingbeil. “Design of frequency-variable digital filters for beam phase control”. In: GSI Scientific Report (2011), p. 340.

[The07] Alexander Theisen. “Implementierung, Optimierung und Systemintegration eines FIR-Filters in VHDL”. Bachelor Thesis. Technische Universität Darmstadt, Institute of Microelectronic Systems, 2007.