Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Institut für
Technische Informatik und
Kommunikationsnetze

Developement of a Simulink Toolbox for the analysis of Real-Time Embedded Systems

Semesterthesis

Authors: Marcel Breu, Remo Niedermann
Supervisor: Prof. Dr. Lothar Thiele
Tutors: Simon Perathoner, Nikolay Stoimenov

Zurich, January 8, 2008

Acknowledgment

At first we want to thank Prof. Dr. Lothar Thiele for the possibility to write our semester thesis at his institute at the Computer Engineering and Networks Lab of the Swiss Federal Institute of Technology (ETH) Zurich.
Further we are very grateful to Simon Perathoner and Nikolay Stoimenov because of their continuous help and support as supervisors during the whole semester thesis.

Contents

1. Introduction

1.1. Motivation

Embedded systems are special-purpose information processing systems that are closely integrated into their environment. An embedded system is typically dedicated to a specific application domain, and knowledge about this domain and the system's environment are used to develop customized and optimized system designs (in contrast to general-purpose computing systems like personal computers).
The embedding into a larger product leads to distributed components that need to communicate with each other trough some interconnection network.
Embedded systems are typically reactive systems i.e. they are driven by events from their physical environment to which they are connected through sensors and actuators. The pace of execution is therefore determined by the environment and as a result many embedded systems must meet real-time constraints. A correct result arriving too late or even too early is wrong for a real-time system. If such a wrong result can lead to a catastrophic failure, the real-time constraint for the system is called hard.
From the above assessments it becomes clear that distributed embedded real-time systems are inherently difficult to design and to analyze. Particularly, as often not only the availability and the correctness of the computation are of major concern but also the timeliness of the computed results. So for real-time systems the focus is on the analysis of timing aspects. In particular, a designer is interested in best-case and worst-case delays as he needs to know whether the end-to-end latency constraints are fulfilled or not. Further the loss of data due to buffer overflows is mostly not accepted. Therefore it is essential for the designer to know if the allocated buffer sizes can accommodate the worst-case buffer requirement of the system.
There are several different approaches to analyzing or estimating the performance of distributed embedded systems. An important distinction is between formal performance analysis methods and simulation based approaches. The former determine hard guarantees of the performance of a system, while the latter can not due to insufficient corner case coverage. This work is based on an analysis method called Modular Per-

formance Analysis with Real Time Calculus (MPA-RTC) [1]. With this framework it is possible to compute hard upper and lower bounds to various performance criteria [2]. The MPA-RTC framework has been implemented in Java and is mainly intended to be used as a MATLAB toolbox. For a user this means that he has to model the system by writing MATLAB-code. It is a modular approach where each HW/SW component is modeled independently with a MATLAB function. The inputs and outputs of the functions get names and the connections between the components are then made by using the right output names as inputs for the next component (Figure 1.1 shows on the right side a MATLAB model). This can be quite a tedious task and it is error-prone and time-consuming. A much more intuitive approach would be to have graphical blocks with inputs and outputs (like the one in Figure 1.1 on the left side) and then the connections could be made just by dragging the outputs of one block to the inputs of the next. Simulink allows to build such graphical models and this work will describe how a graphical user interface for the MATLAB RTC toolbox has been implemented in Simulink.

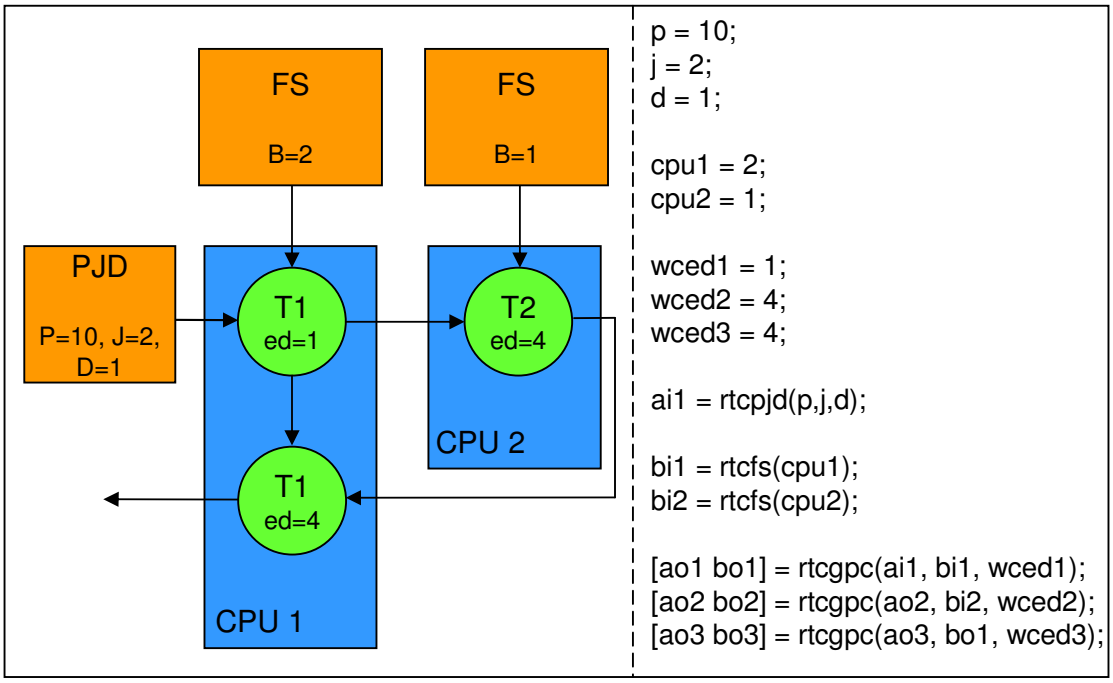


Figure 1.1.: Performance models of the same system made with a graphical representation on the left and MATLAB code on the right

1.2. Related Work

SymTA/S is another tool for performance analysis with a graphical user interface. The name stands for Symbolic Timing Analysis for Systems. The SymTA/S software and technology was developed at the Institute of Computer and Communication Network Engineering at the Technical University of Braunschweig [4].
In contrast to the MPA framework the SymTA/S approach is limited to a few classes of input event models [5].

1.3. Contribution

A basic prototype of a graphical user interface for the RTC Toolbox in form of a Simulink library already existed but this prototype was not complete and many problems were not solved. The contribution of this work is the design of a new GUI which extends the functionality of the prototype by the following points:

- Almost all functions of the RTC toolbox are implemented as Simulink blocks
- The precision (in terms of number of segments) for the representation of Variability Characterization Curves can be adjusted dynamically
- The library is able to model cyclic systems.
- The appearance of the blocks has been improved (i.e. blocks can have an arbitrary number of input ports and the port labels will change accordingly)

1.4. Structure of this work

- In Chapter 2 the theoretical background of Modular Performance Analysis with Real-Time Calculus is presented. In particular the concepts of variability characterization curves and of performance models are described.
- Chapter 3 is the main part of this work, it explains how exactly the functionality of the RTC toolbox was implemented in Simulink.
- Readers who just want to work with the library are advised to read Chapter 4 because this is a guide on how to use the library.
- The last chapter contains the conclusions of the work and some points for future improvements of the library.

2. Modular Performance Analysis with Real Time Calculus

2.1. Theoretical Background

Modular Performance Analysis with Real Time Calculus (MPA-RTC) [1] is a framework for performance analysis of distributed embedded systems that has its roots in network calculus [3]. MPA-RTC analyzes the flow of event streams through a network of computation and communication resources in order to derive hard upper and lower bounds to various performance characteristics of a distributed embedded system, such as end-to-end delays or buffer requirements.
The central idea of this method is to build a so-called performance model which is an abstraction of the concrete system. The performance model consists of components that are communicating with each other through arrival and service curves (denoted together as variability characterization curves).
The next sections describe the parts of such a performance model: first the variability characterization curves, how a component processes these curves, the resource sharing between different components and the analysis of such networks.

2.1.1. Variability Characterization Curves

The timing characterization of event and resource streams is based on variability characterization curves (VCC) which are an abstraction of the classical representation such as sporadic, periodic or periodic with jitter. While the number of events and the available resources are represented in the time domain, the VCC's are represented in a time-interval domain. Figure 2.1 depicts this step from one domain to the other: The curves in the time-interval domain are created by moving a window of size S over the time domain and counting the minimal and maximal number of events in all of these windows.
The event streams are described using arrival curves $\alpha^u(\Delta)$, $\alpha^l(\Delta) \in \mathbb{R}^{\geq 0}$, $\Delta \in \mathbb{R}^{\geq 0}$ [5] which provide upper and lower bounds on the number of events in any time interval

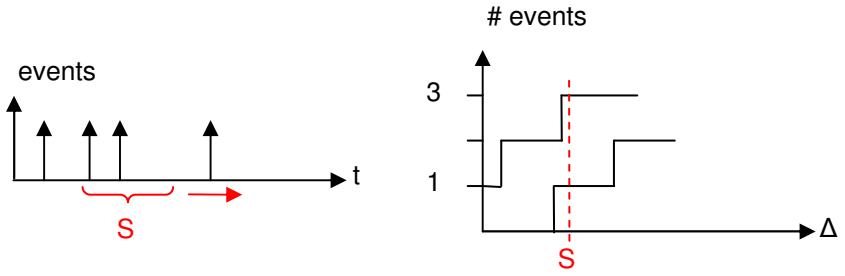


Figure 2.1.: From event streams to VCCs

of length Δ. In particular, if $R[s,t)$ denotes the number of events that arrive in the time interval $[s,t)$, then the following inequality is satisfied:

$$\alpha^l(t-s) \leq R[s,t) \leq \alpha^u(t-s) \forall s < t$$

In a similar way, the resource streams are characterized using service functions $\beta^u(\Delta)$, $\beta^l(\Delta) \in \mathbb{R}^{\geq 0}$, $\Delta \in \mathbb{R}^{\geq 0}$. If $C[s,t)$ denotes the number of processing or communication units available from the resource over the time interval $[s,t)$, then the following inequality is satisfied:

$$\beta^l(t-s) \leq C[s,t) \leq \beta^u(t-s) \forall s < t$$

These curves can be obtained from different sources: perhaps the pattern of the event or resource stream is known, e.g. bursty or periodic or they can be derived from a finite set of event traces. In other cases, one can derive the curves by deriving the bounds from the characteristics of the generating device or the hardware component [6]. Figure 2.2

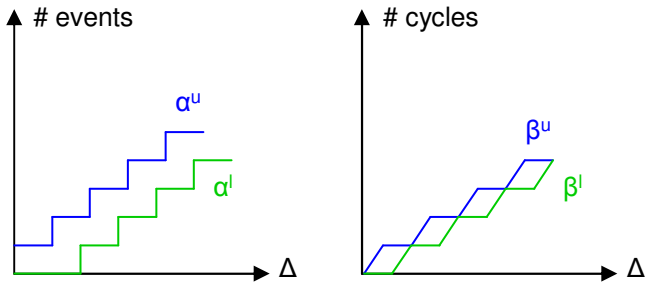


Figure 2.2.: An example of an event and a resource stream: left) periodic with jitter, right) TDMA

shows an example of a periodic event stream with jitter and a resource curve from a TDMA resource.

2.1.2. Analysis of a Greedy Processing Component

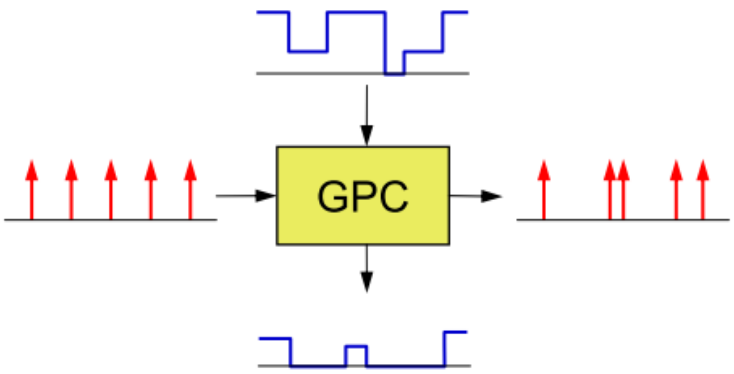


Figure 2.3.: A greedy processing component

A Greedy Processing Component as in Figure 2.3 (taken from [7]) realistically models the semantics of many HW/SW tasks. In particular, an incoming event stream represented as a pair of arrival curves α^l and α^u, flows into a FIFO buffer in front of the processing component. The component is triggered by these events and will process them in a greedy manner while being restricted by the availability of resources, which are represented by a pair of service curves β^l and β^u. On its output, the component generates an outgoing stream of processed events, represented by a pair of arrival curves $\alpha^{l'}$ and $\alpha^{u'}$. Resources left over by the component are made available again on the resource output and are represented by a pair of service curves $\beta^{l'}$ and $\beta^{u'}$. The output curves are given by [7]:

$$\begin{aligned}
\alpha^{l'} &= \left[\left(\alpha^l \oslash \beta^u\right) \otimes \beta^l\right] \wedge \beta^l \\
\alpha^{u'} &= \left[(\alpha^u \otimes \beta^u) \oslash \beta^l\right] \wedge \beta^u \\
\beta^{l'} &= (\beta^l - \alpha^u)\overline{\otimes}0 \\
\beta^{u'} &= (\beta^u - \alpha^l)\overline{\oslash}0
\end{aligned}$$

The definition of the used operators is given in the appendix.

2.1.3. Resource sharing

The service inputs and outputs in the system performance model are interconnected to reflect the resource sharing policies in the system. If for example several tasks of a system are allocated to the same resource these tasks share this resource according to a scheduling or arbitration policy. In the performance model, this scheduling or arbitration policy on a resource can then be modeled either by the way the abstract resources β are connected among the different abstract tasks or by additional scheduling blocks. One of the simplest resource sharing strategies to implement is fixed priority: this is done by connecting the service curves of two performance modules serially, so the second module can only use the service that is left-over from the first one. This scheme is depicted in Figure 2.4 (taken from [7]).

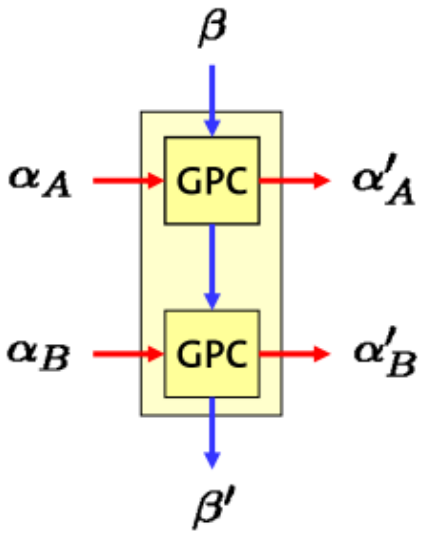


Figure 2.4.: Modeling of the fixed priority resource sharing scheme

2.1.4. System Performance Analysis

After interconnecting all components the performance model captures all the information that builds the basis for performance analysis. Various performance criteria, such as end-to-end delay guarantees or buffer requirements can be computed analytically in the performance model. The performance analysis of a distributed embedded system is done by combining the analysis of the single processing components of a performance model. The maximum delay d_{max} experienced by an event at a system module and the

maximum size of the buffer b_{max} can be bounded by the following inequalities:

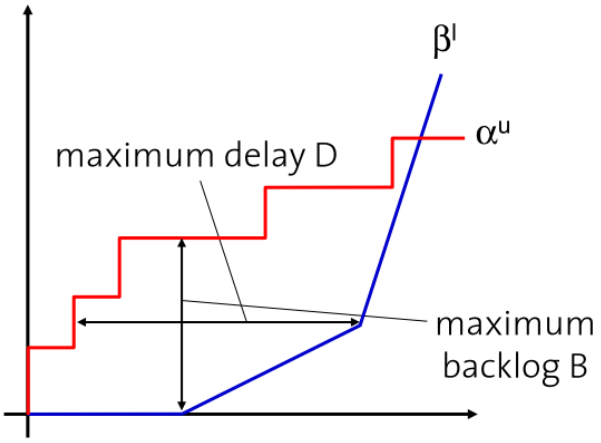


Figure 2.5.: Maximal delay and buffer requirement obtained from arrival and service curves

$$
\begin{aligned}
d_{max} &\leq \sup_{\lambda \geq 0} \left\{ \inf \left\{ \tau \geq 0 : \alpha^u(\lambda) \leq \beta^l(\lambda + \tau) \right\} \right\} \stackrel{def}{=} Del(\alpha^u, \beta^l) \\
b_{max} &\leq \sup_{\lambda \geq 0} \left\{ \alpha^u(\lambda) - \beta^l(\lambda) \right\} \stackrel{def}{=} Buf(\alpha^u, \beta^l)
\end{aligned}
$$

A graphical interpretation is that b_{max} is the maximal vertical distance and d_{max} the maximal horizontal distance between the upper arrival curve and the lower service curve as can be seen in Figure 2.5 (taken from [7]).
The total end-to-end delay or the total buffer requirement is then computed by adding the delay or buffer requirement of the single components. However, another computation is also possible when an event stream is processed by a sequence of components and when the components use the same shared memory. Then it is possible to exploit a phenomenon known as „Pay Bursts Only Once“ (this means that the worst-case scenario of all tasks will not coincide [8]) and the end-to-end delay and the total buffer requirement can be tightened to [2]:

$$
\begin{aligned}
d_{max} &\leq Del(\alpha^u, \beta_1^l \otimes \beta_2^l \otimes \ldots \otimes \beta_n^l) \\
b_{max} &\leq Buf(\alpha^u, \beta_1^l \otimes \beta_2^l \otimes \ldots \otimes \beta_n^l)
\end{aligned}
$$

2.2. Software Architecture

The Real-Time Calculus (RTC) Toolbox is a free MATLAB toolbox for system-level performance analysis of distributed real-time and embedded systems. The RTC Toolbox is based on an efficient representation of Variability Characterization Curves (VCC's) and implements most min-plus and max-plus algebra operators for these curves. On top of the min-plus and max-plus algebra operators, the RTC Toolbox provides a library of functions for Modular Performance Analysis with Real-Time Calculus [9].

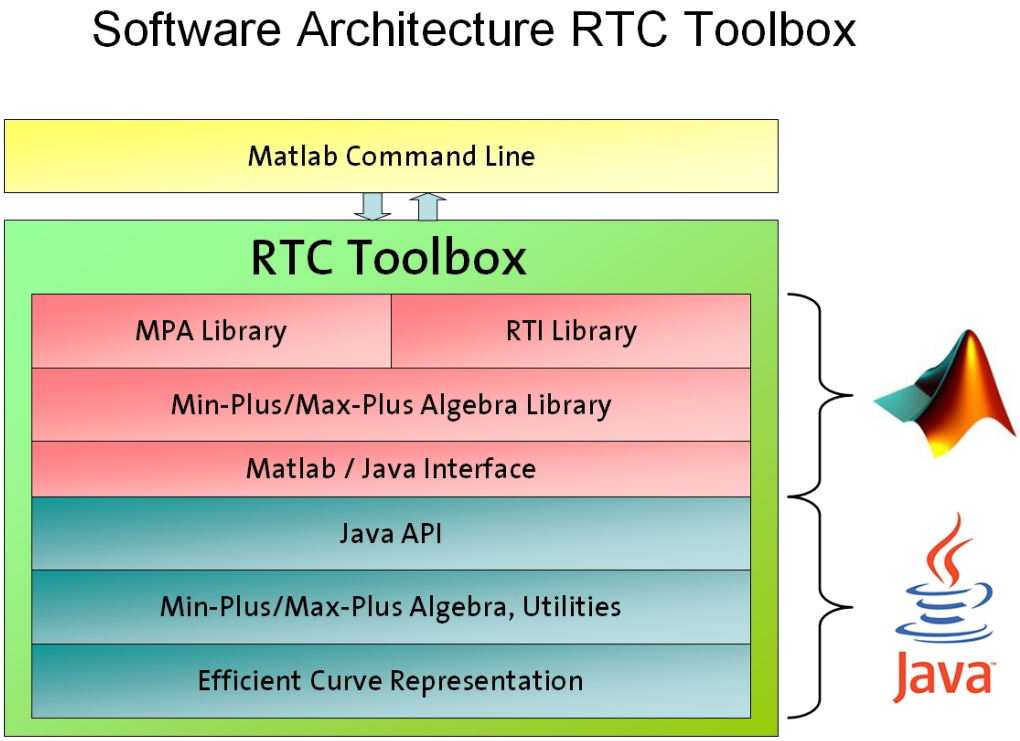


Figure 2.6.: The architecture of the RTC Toolbox

3. Implementation

This chapter describes the actual implementation of the graphical user interface in Simulink. The first section lists the requirements that the GUI should fulfill. In section 3.2 the Software structure is presented. Section 3.3 explains the problem with the curve representation in Simulink and the solution to it. In section 3.4 some important points of the implementation will be discussed in detail and the following two sections deal with the problem of cyclic dependencies. At the end of this chapter in section 3.7 the testing of the blocks is described.

3.1. Requirements

1. The Simulink GUI should work with MATLAB R2007b and the current latest version of the RTC Toolbox.

2. The Simulink GUI should only be a separate front end to the RTC Toolbox. It means that the RTC Toolbox should be able to work as before without the Simulink GUI. There should be a clear separation between the GUI and the Toolbox. The interface of interaction between the two should be clearly defined. Any needed Java methods should be added to the file MatlabUtil.java

3. The user should be able to model any system with the Simulink GUI that is possible to be modeled with the RTC Toolbox including cyclic systems.

4. The results of the analysis should be the same regardless of whether the user uses the Simulink GUI or not.

5. Create a structured library of components in Simulink following the guidelines where RTC MATLAB functions are grouped as:

   Curve generation functions: rtcbd, rtccurve (create any curve), rtcfs, rtcpjd, rtcps, rtctdma

   Analysis: rtcbuf, rtcdel

Processing and scheduling components: rtcedf, rtcfifo, rtcgpc, rtcgsc

Functions for complex task activations: rtcor, rtcand

Operations on curves and curve sets: rtcaffine, rtcceil, rtceq, rtcfloor, rtch, rtcmax, rtcmaxconv, rtcmaxdeconv, rtcmin, rtcminconv, rtcmindeconv, rtcminus, rtcplus, rtcrdivide, rtcsteps, rtctimes, rtcuminus, rtcv

Display components: display (for displaying a scalar or textual representation of a curve), plot (for graphically representing curves), plotgsc, plotgpc

6. Create a short tutorial for using the Simulink GUI.

3.2. Software Architecture

The Software Architecture is structured in three levels: Simulink, MATLAB and Java (Figure 3.1). A Simulink block will call a MATLAB function which then will call a function from the Java-Core to do the whole computation. Basically in the MATLAB RTC toolbox curves are represented as Java objects and the operations on curves are implemented in the Java-Core. Simulink, on the other hand, cannot handle Java objects so it is necessary to convert these curves into matrices. This is done by using special MATLAB functions (more details can be found in the next Section). The new RTC Library in Simulink consists of six categories: curve, analysis, processing and scheduling components, operations and display components and each of them contains blocks (e.g. the category processing and scheduling components contains the GPC, GSC, FIFO and EDF blocks).
Every block implementation consists of two parts:

- A Level-2 M-File S-Function: These M-Files contain the behavioural logic of the blocks. The block input and output ports are defined, the calculation is done with the input data and the result of the calculation is assigned to the output ports.
- A block mask: Block Masks are custom user interfaces with an own parameter dialog box and a self-defineable visual appearance. In the library they are used to provide a custom help message and to allow the user to easily enter the parameters of a block.

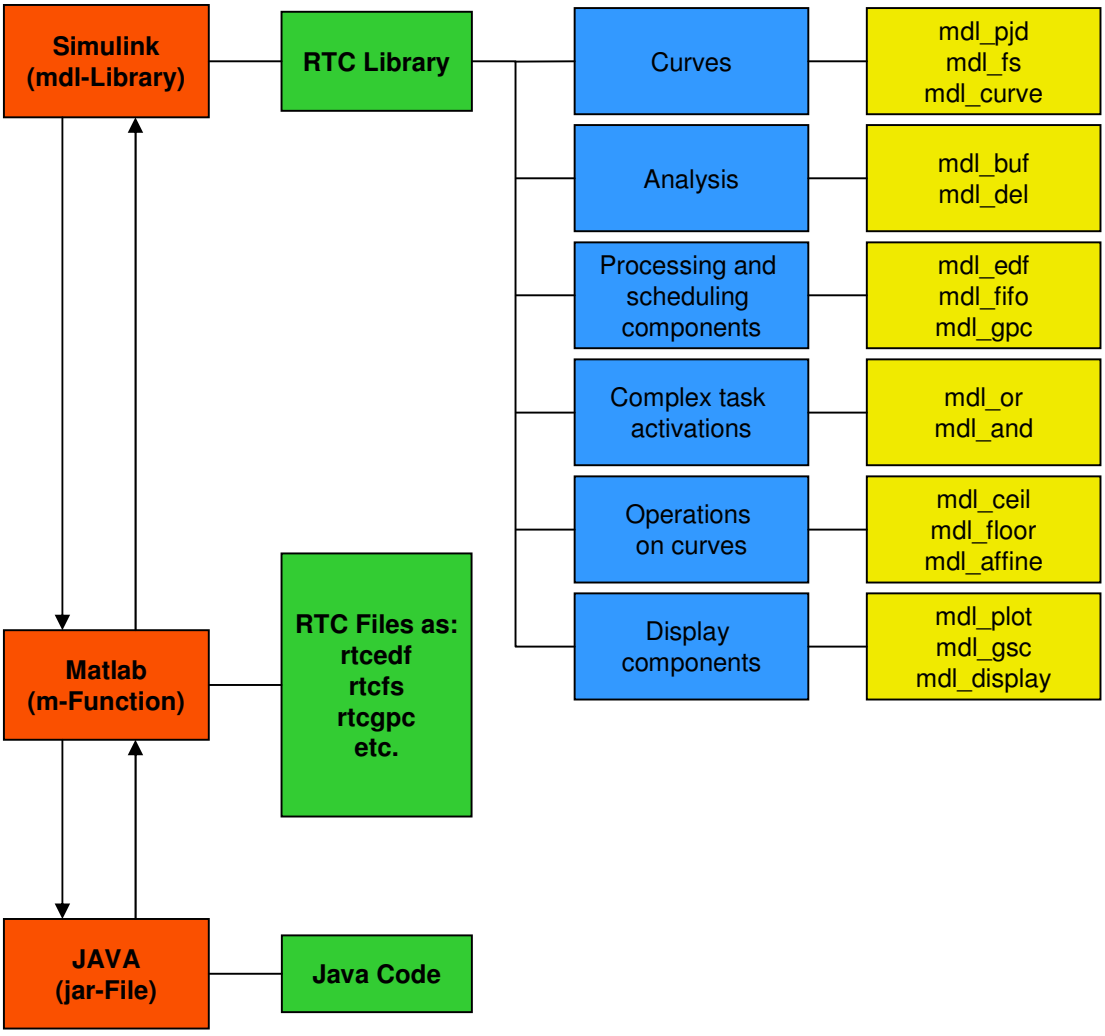


Figure 3.1.: Software Architecture and Dataflow

3.3. Curve Representation in Simulink

The RTC toolbox represents the arrival and service curves as Curve-Objects which is a user defined Java datatype. In Simulink however it is not possible to pass objects but only numerical matrices. This section describes the implications of this property for the implementation of the Simulink library.
The RTC toolbox defines curves as consisting of an aperiodic part and a periodic part. These parts are made of segments where every segment is a three-tuple: x-position, y-position and slope. The end of a segment is defined by the start point of the next segment, so basically a curve is a piecewise linear graph. This structure has to be represented now as a simple matrix to be used in Simulink. A solution to this problem is already provided in the RTC toolbox in form of the functions `importFromSimulink` and `exportToSimulink` which convert a matrix into a curve and vice versa. It is possible to call these functions directly in MATLAB but then the user has to provide the class name, in this case this would be e.g. `MatlabUtil.importFromSimulink` which is not so user-friendly. That is why two MATLAB functions were added which call the Java functions. Their names are `fromSimulink` and `toSimulink`.
The matrix has three columns and the number of rows is defined by a Java variable. In the first three rows additional data about the curve, like the period and the size of the periodic and aperiodic part is stored and then the following rows contain the segment information (x-position, y-position, slope) first of the aperiodic part and then of the periodic part (this means that three segments less than the number of rows can be stored in this matrix). Hence the number of rows is very important for the representation of curves, this length was set to 250 in the RTC toolbox and could not be changed. This was too unflexible, since it permitted to handle accurately only curves up to a fixed size. To overcome this limitation getter and setter methods for this Java variable have been added to the Java-Core (this was the only change to the Java-Core that was done during this work). The corresponding MATLAB functions are called `rtcsetNumInputs` and `rtcgetNumInputs`. `rtcsetNumInputs(num)` sets the number of segments that the user thinks is necessary to store and `rtcgetNumInputs()` returns the number of segments (there is also the function `rtcgetDataLength` which returns the total size of the matrix, i.e. the number of segments plus three). This approach leads to a drawback: At the time where a block is set up in Simulink the size of the output curve is not yet known (because the computation of this is done in a following step). This means that the size cannot be set dynamically to different values for different curves, but there is one value for all curves. This is some sort of waste, many curves will have less seg-

ments than the maximum size and in their matrices there will be many zero values (in MATLAB the Java curve objects require just the memory they need).

Figure 3.2 shows the error message from Simulink if a curve has more segments than

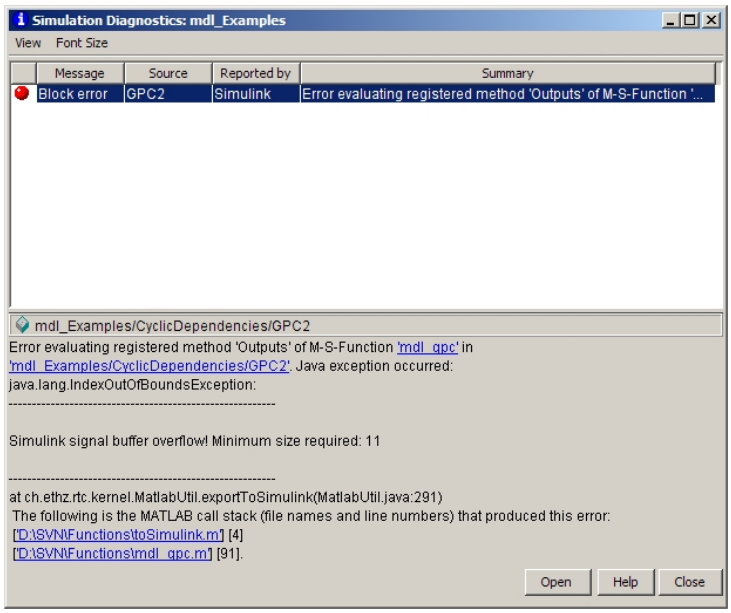


Figure 3.2.: This message means that the number of segments needs to be increased

the matrix can store. This means that the user has to increase the size. Just taking the value provided in the error message might not be such a good idea, because Simulink just stops when it finds the first curve which is too big. This does not mean that the problem is completely solved, it could be that the curves grow even larger while the analysis continues and that the error message shows up again.

The user now has the possibility to adapt the size of the data structure that stores a curve which is an important advantage with respect to the prototype. For the modeling of several sample systems this approach has shown to be useful.

3.4. Function Implementation in Simulink

Simulink is an environment for multidomain simulation and Model-Based Design for dynamic and embedded systems. It provides an interactive graphical environment and a customizable set of block libraries that lets a user design, simulate, implement, and test a variety of time-varying systems, including communications, controls, signal processing, video processing, and image processing. It is integrated with MATLAB, pro-

viding immediate access to an extensive range of tools that lets users develop algorithms, analyze and visualize simulations, create batch processing scripts, customize the modeling environment and define signal, parameter and test data [10]. Simulink provides three possibilities to create custom blocks and libraries:

MATLAB Function Blocks There are three subtypes of these blocks which can use MATLAB commands to implement functionality

Subsystem Blocks These blocks allow to introduce hierarchy: some components can be aggregated into a new single component. This improves conciseness in models, but only predefined blocks can be used

S-Function Blocks They allow to implement custom functionality using M,C or C++ code

Since the RTC toolbox exists in form of M-Files our first approach was to use embedded MATLAB function blocks. This did not work because embedded MATLAB only supports a subset of the MATLAB language and the whole Java-functionality cannot be used. The next approach was to look at the prototype in which some functions were already implemented in form of S-Functions. The most efficient solution seemed to be to do it the same way as in the prototype, hence the blocks of the new library use S-Functions (more specifically they are called Level-2 M-File S-Functions).
Simulink provides a template file for S-Functions. Starting with this template the behaviour of the different blocks had to be implemented and the main part was always done in the „output“ function. The following parts will describe how the curves are implemented, how blocks of different input-output combinations have been constructed and what block masks can do to improve the user interface.

Curves

In the prototype all interconnections between blocks consisted of a combination of lower and upper curves. The other possibility would be to have separate connections for lower and upper curves but this would increase the number of connections and therewith lead to a more cluttered model. That is the reason why the new library also uses the representation with a single connection which consists of a lower and an upper curve. For the code this means the following: The signal has a user-defined number of rows and six columns, the first three are for the lower curve and the last three are for the upper curve (see 3.3 for more explanations). However in the RTC Toolbox all functions which return a combination of a lower and an upper curve the first element is the upper curve e.g.:

```
beta = rtcfs(5);
output = [toSimulink(beta(2)) toSimulink(beta(1))];
```

so element two of beta (the lower curve) will be the first part of the output signal and element one (the upper curve) is the second. Accessing the elements and reconverting them into curves is done using the following code (where it is assumed that the output of the previous code is fed as input to the next code):

```
beta_lower = fromSimulink(input(:,1:3));
beta_upper = fromSimulink(input(:,4:6));
beta = [beta_lower beta_upper];
```

The actual code looks a bit different because of the use of so-called dwork storage elements (more on this in 3.6.2), but these code parts show the general idea.

Blocks with a variable number of inputs and outputs

The most difficult type of blocks to implement were those which had a user definable number of input ports and then produced a variable number of output ports like the EDF and the FIFO block. The first problem there was how to construct the call to the actual RTC-Toolbox function, because it needs a variable argument size e.g. rtcfifo(alpha, beta), rtcfifo(alpha, beta1, beta2) or even worse with different data types. To achieve this the „cell-array“ functionality of MATLAB was used. Cell-arrays can be built dynamically and they can contain different datatypes. The following code shows this: It builds the cell-array args. In every step a curve, a worst-case execution demand and a deadline is added to it. This array can then be passed to the RTC-Toolbox function.

```
for k=1:numInputs
  args = {args{:} beta(k) wced(k) deadline(k)};
end
```

The second problem was that the EDF and FIFO functions return a variable number of outputs, depending on the number of inputs. Unfortunately this could not be solved with a cell-array and another trick was needed. The basic idea here was to use the MATLAB function „eval“ which takes a string and executes it the same way as when it would have been typed on the MATLAB command line. So in a loop the string gets constructed and then the eval-function will execute the statement in the string. To assign the outputs of the function to the ports the eval-function has to be used again. A code example for the EDF-block which does this is given in the following (only the assignment of the last port is shown):

```
% Goal: have a string like [a1,del1,buf1,...]=rtcedf(args{:})
% which will then be EVALuated
```

```
strbegin = '[';
ret = 'a1,del1,buf1';
for k=2:(block.NumInputPorts-1)
    ret = strcat(ret, ',a', mat2str(k), ',del',mat2str(k),',
          buf',mat2str(k));
end
strend = ',b]=rtcedf(args{:});';
func = strcat(strbegin, ret, strend);
eval(func);
% Assign the outputs of the function to the output ports
be=eval('b');
block.OutputPort(block.NumOutputPorts).Data=[toSimulink(be(2))
                                              toSimulink(be(1))];
```

Block Masks

Another limitation of the prototype was that the block masks could not handle parameterized number of input ports. If a block needed more inputs than the predefined number the ports did not get a name. This can be solved with dynamic masks, so the mask uses a so-called callback-function to update its appearance. For some blocks like the DEL WCED or FIFO the block mask has an additional functionality. These blocks need parameters for all inputs. The user cannot enter the number of input ports that he wants to have like in other blocks but only a vector with the parameters. The block mask then determines the number of input ports from the size of this vector.

3.5. Analysis of non-cyclic systems

For systems which do not include cyclic dependencies the analysis is straightforward i.e. the output of a block becomes the input of the next block. But the fact that the same blocks are used for cyclic and for non-cyclic systems has an impact on the efficiency of non-cyclic systems: They need to be calculated twice, the first time only the initial curves of the blocks are taken into account. For the second time the real inputs are used and the result will be right. This means that the analysis time is twice as high as in the optimal case. The explanation for the use of these initial curves is given in the next section.

3.6. Analysis of cyclic systems

3.6.1. Initial Curves

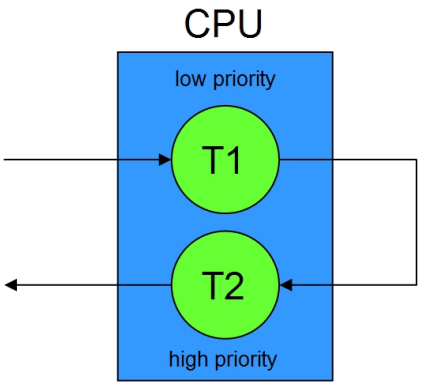
After implementing the first version of the blocks one big problem occurred: For systems with cyclic dependencies the blocks need some initial curves for the first execution. An example of a system with cyclic dependencies is given in Figure 3.3. Task two has a higher priority than task one. This means that task two gets a full service resource and task one gets what is left-over from task two. But task one needs to process the event stream first. So the situation is the following: Task one has events to process but does not get processing time and task two has a full service resource available but no events to process. The first try to solve this problem was to implement an initial block. This block has two inputs: for the first time the input from port one (the initial curve) is forwarded to the output and thereafter the input of port two (the actual curve) gets forwarded (Figure 3.4). The usage of this block did not solve the problem. Simulink has rules to determine the execution order of blocks and this rule does not choose the initial block as first block. Hence it will execute a block which does not have all inputs available and the problem persists. Another solution which works is presented in the next section.

3.6.2. Storage elements

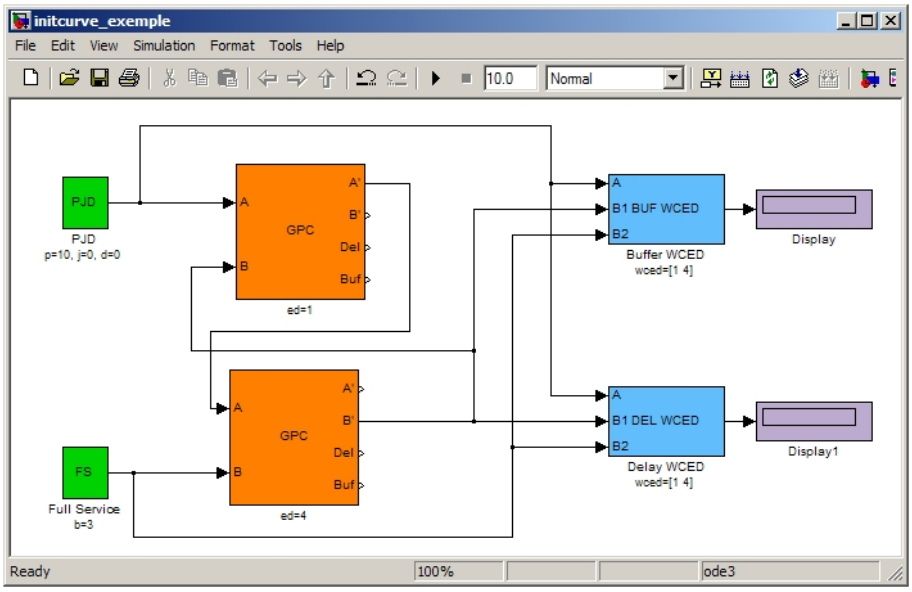
In Simulink a block can have storage elements. These are called „dwork“ and are vectors (curves are represented as matrices so there will be a conversion from a vector to a matrix and vice versa). The main idea is that the functions in the blocks do not directly access the input data but the storage element (as depicted in Figure 3.5) and this element will be initialized before the first access. After the first timestep the values of the storage element will be overwritten with the values from the input ports. For the previous example (Figure 3.3) this means that task one can be executed, it will use the curves in the storage element and the same holds for task two. After this step both blocks update their storage elements with the actual input.

For the introduction of the dwork storage element a property in the block setup function needs to be changed and some supplementary functions need to be implemented:

`Setup` Hence the blocks do not access directly the input ports to calculate the output data, the DirectFeedthrough option of the input ports has to be declared to be false.



(a) Model



(b) Simulink Representation

Figure 3.3.: Example system with a cyclic dependency

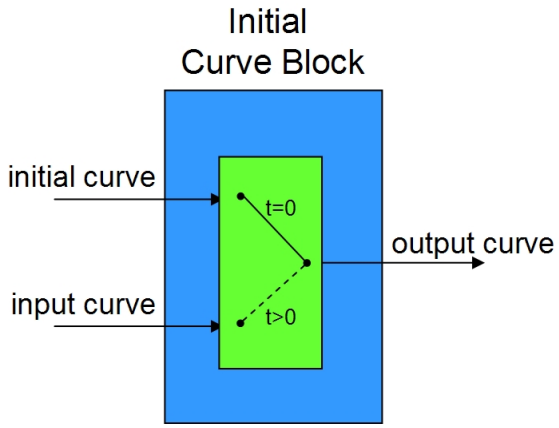


Figure 3.4.: Initial Curve Block

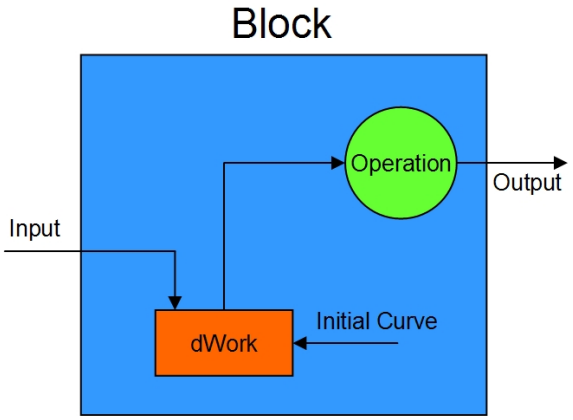


Figure 3.5.: The construction of a block with dwork

`PostPropagationSetup` In this function the dwork elements are declared.

`Start` The Start-Function is executed only once during an analysis. The initial curves are defined and stored in the dwork elements.

`Output` This function is in all blocks, independent whether they have dwork elements included or not. This function is the heart of the block because here it is implemented what the block is designed for and this means for the dwork elements, that they will first be read out and the vector is stored in a matrix. The matrix will then be converted into curve-objects. After this procedure the function can do its job.

`Update` At the time the block is executed, it uses the curves from the dwork elements and the current input curves need to be stored after the execution of the Output-Function. This is exactly what the Update-Function does: The data from the input ports are stored in the dwork elements.

An example of the dwork implementation is in the appendix A.4.

The initial curves solve the problem of cyclic dependencies but a new problem can occur. Figure 3.3 shows an example where this problem arises. On a CPU two tasks are executed. The second task has higher priority than the first one. The scheduling determines that task two gets full service while task one gets only time to execute when task two is idle whereas task two can only execute when task one already has done some executions. For the first iteration each block will be executed with initial curves. The outputs of the tasks depend on the initial curves which means that by changing the initial curves the output change and therefore the inputs of the two blocks for the second iteration. This difference at the beginning of the analysis can have consequences for the whole analysis.
The parameters for the initial curves are shown in Table 3.1 and the Simulink representation is in Figure 3.3(b) It is difficult to find some useful initial curves because there

Input	**Initial Curve in GPC**	**Delay & Buffer**
β = FS(1) α = PJD(10,0,0)	β=FS(1) α=PJD(10,50,1)	Delay = 17 Buffer = 2
β = FS(1) α = PJD(10,0,0)	β=FS(1) α=PJD(1,0,0)	Delay = 9 Buffer = 1

Table 3.1.: Initial Curve Problem

always exists a model where the result is not correct. All blocks of the new Simulink library have the initial curves shown in Table 3.2.

Curve	**Values**	**Description**
α	FS(0)	no events in the event stream
β	FS(1)	full service curve with bandwidth = 1

Table 3.2.: Initial Curves

The advantage of these initial curves is the following: The first time a block (e.g. GPC) is executed it uses the initial curves, which are for α: full service with bandwidth zero (nothing arrives, a service curve generator has been taken because it is the only one

which produces an all-zero curve - with the PJD-Block this does not work) and for β: full service with bandwidth one. The clue is that after the execution the output α- and β-curve are the same as the input ones. For the example in Figure 3.3 this means that the first GPC gets the full service because the second GPC has nothing to do and the calculation does not change the α- and β-curve.
From these considerations it can be seen that the initialization values for the curves play an important role. In the library they are hard-coded into the M-Files which is not a very flexible approach. It would be much better if the user would be able to set them himself. A first idea to achieve this was to define special functions in the MATLAB workspace and then use them as initialization curves. However it was not possible in Simulink to access these functions. Another possibility would be to have the desired initial curves as a parameter in every block. So the user could double-click on a block and then select which curves he wants to have as a starting point. This could be done using a text field where the user enters the function name and this string would then be evaluated and assigned to the dwork. There might be a problem for blocks with a variable number of input ports, the user may be forced to define many curves. This implementation could not be done because it exceeded the time of this semester thesis.

3.7. Testing

One of the most important requirements for the graphical user interface was that it should compute the same results as the RTC toolbox. For the testing this means that a model built with Simulink blocks should produce the same results for all possible analysis scenarios as the model of the same system built with MATLAB code.
In the RTC toolbox there are some demo systems provided which show how to use the functions. For testing purposes the cyclic dependency demo is especially interesting because it shows the two major problems of this work, namely curve representation in Simulink and cyclic data dependencies. In the example system there are two scenarios, one which has cyclic dependencies, the other not. With the non-cyclic version the implemented blocks worked fine and the outputs from the model and from the MATLAB code were identical. But if the model was changed to include cyclic dependencies, the problem of algebraic loops occurred (see Figure 3.6 for the error message). This problem was then solved by the help of data storage vectors that contain initial curves as described in chapter 3.6.2. With further testing another problem was found out: The initial curves for the data storage vector can influence the whole calculation (see chapter 3.6.2).

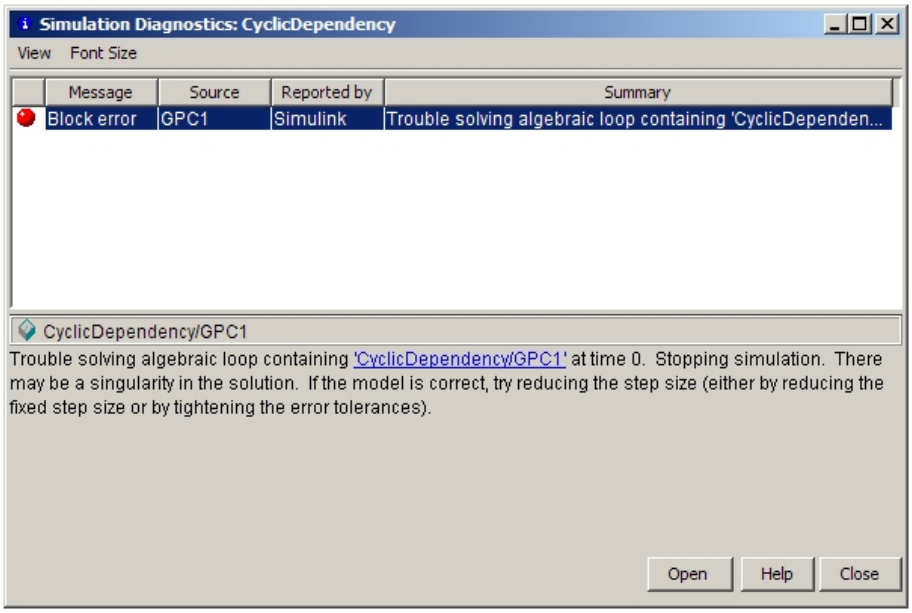


Figure 3.6.: Error message caused by algebraic loop

The result of an analysis can be wrong because of different reasons as pointed out before, therefore are here some points to test if the result is correct.

- Change the values for the initial curves. This has to be done in the Start-Method of the corresponding function, so if you want e.g. to change the initial curves for the GPC-Block you have to edit the file mdl_gpc.m. If it returns the same values for buffer and delays the results should be correct.
- To ensure that the calculation has reached a fix-point the values for the buffer and the delay should no longer change during the last steps of the analysis, otherwise the number of iterations has to be increased. In Simulink this can be done by increasing the stop time because every timestep corresponds to an iteration.

4. User Guide

4.1. Tutorial

This step by step tutorial shows how to build a simple performance model with an example. Figure 4.1 shows the system. It consists of 3 tasks, which are run on two different CPUs, whereas each one can execute one event per time-unit. The first and third task share a CPU, the scheduling strategy is fixed-priority with task three having a higher priority than task one. For task one the Best Case Execution Demand (BCED) and Worst Case Execution Demand (WCED) are equal to one. Task two and three are identical and have equal WCED and BCED. This leads to a cyclic dependency between these two tasks. Task one has lower priority than task three therefore it can only execute when task three is idle while task three can always execute as long as it has events to execute but the events first have to be executed by task one and two. If the priority of task one and three is swapped, the model will no longer have cyclic dependencies because task one then has higher priority than task three and executes all the arriving events while task three has to wait until task one is idle afterwards it can execute all his events. The number of needed blocks is independent whether the system has cyclic dependencies or not. The differences are just in connecting the blocks.
The event stream or arrival curve is a PJD-curve. The events arrive with a certain period (e.g. 10 time units), can have jitter (e.g. 50 time units) and must have a minimum event inter-arrival time (minimum time difference between two events, e.g. 1). The parameters for the example model are summarized in Table 4.1

4.1.1. How to create a Model

1. Enter in MATLAB: `rtcNewModel('Testmodel')`
   This function creates a new model with name „Testmodel“ in the current folder. It initializes the fixed step size to 1, the solver to discrete values, Start Time to 1 and Stop Time to 10. Finally the warnings for unterminated output ports are avoided. Such a model has a predefined number of segments for curves of 250. If a curve grows larger, an error message will appear. The number of segments can

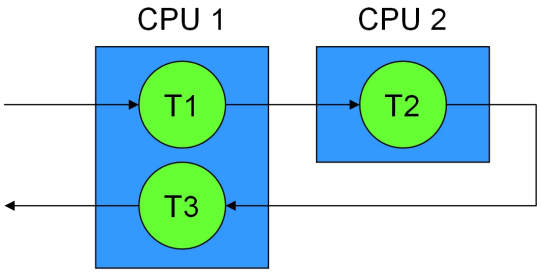


Figure 4.1.: Example Model

arrival curve	period = 10 jitter = 50 minimum event inter-arrival time = 1
service curve	bandwidth = 1
Task 1	WCED = 1 BCED = 1 priority = low
Task 2	WCED = 4 BCED = 4 priority = high
Task 3	WCED = 4 BCED = 4 priority = high

Table 4.1.: Specification of the model

be increased using the function `rtcsetNumSegments(len)` on the MATLAB prompt (see 3.3 for more explanations).

2. Start Simulink and look for the Real Time Calculus Toolbox. Here are all needed blocks for the model.

3. Take the boxes by „drag and drop“ in your model. For the example 3 GPCs (which will model the tasks), 1 DEL WCED, 1 BUF WCED, 2 FS, 1 PJD and 2 Displays (to show the size of the buffer and the delay) are needed, as seen in Figure 4.2. Double click on the box and enter the parameters given in Table 4.1 for the selected box (e.g. b=1 for FS)

4. The last step is to connect these blocks properly. Here it will be described for the case that task one has higher priority than task three (no cyclic dependencies).
   - The best way is to start at the PJD curve generator. The event stream (α) first

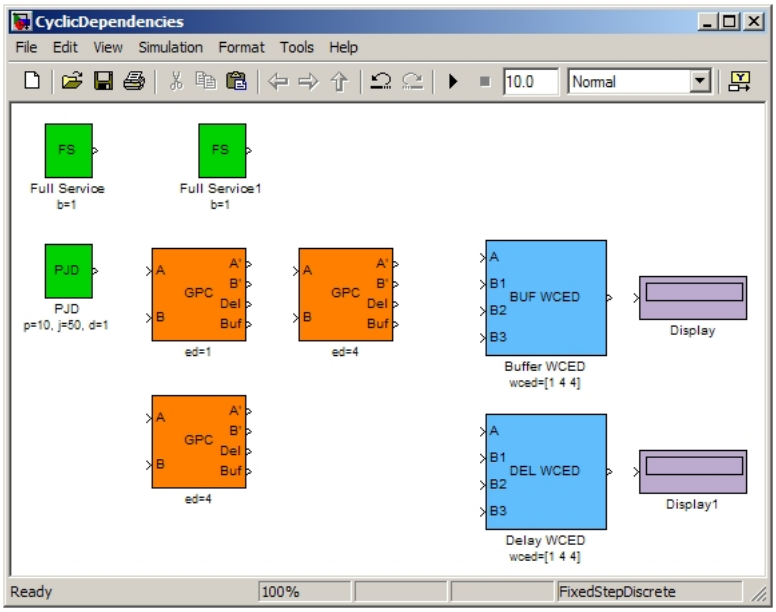


Figure 4.2.: Blocks for the Tutorial Model

goes to the first GPC (input port A) and leaves it at output port A′. Then enters to the second GPC and finally to the third one. The connections are made by clicking with the mouse-pointer on an output port and then holding and dragging it to the input port of the next block.

- The next step is to connect the service curves. Task one has higher priority hence it gets the full service (connect the FS block to the input B of the first GPC). The output port B′ is the service curve for the third GPC (task three), so B′ of the first GPC can be connected to the input port B of the third GPC.
- The second CPU has just one task, therefore the second FS block is connected to the input port B of the second GPC. Now the system is modeled, but the performance analysis part is missing. This will be done in the next steps.
- To calculate the delay and the buffer requirement there are two ways: the first one is the modular approach which sums up all the outputs DEL and BUF from the GPCs and the second one is a holistic approach which gives a tighter bound (for more details have a look at Subsection 2.1.4). This example will use the second method with the blocks DEL WCED and BUF WCED. The blocks get the α curve from the PJD block and the β curves from the input of the three GPCs.

- Finally connect the output of the BUF WCED and DEL WCED block to the displays.

After this guidance your model should look the same as Figure 4.3(a). The Figure 4.3(b) shows the model when task 3 has higher priority than task 1.

4.2. Errors and Solutions

There are two main errors which can occur therefore here are some solutions to get rid of them.

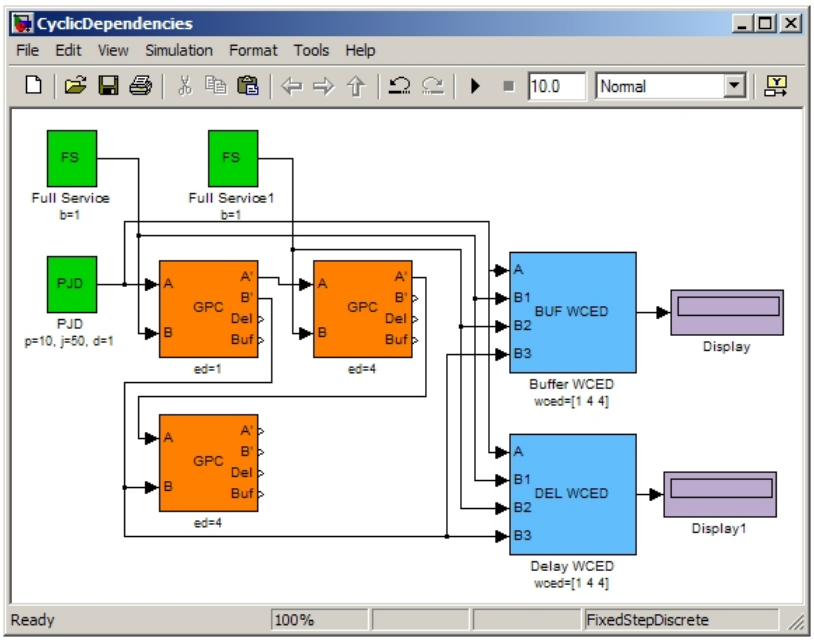
4.2.1. Number of Curve Segments

Simulink cannot deal with Java-objects like the curve-object used for the curve representation therefore the curve-object needs to be converted into a matrix of finite dimension. The rows of the matrix represent segments which are defined as x-position, y-position and slope. By default the matrix has 250 rows for the segment entries. If a curve has more than 250 segments after a calculation Simulink returns an error. In Figure 4.4 is the error message for the cyclic dependency example in chapter 4.1.1, which occurs when the maximal number of segments is ten. The error message says that the minimum required size for the curve is eleven. After resizing the minimum number of segments to eleven the error message occurs again, this time the size should be 16. The error message will appear several times until the segment size is 27 therefore it is better to enlarge it much more than the error message indicates otherwise the error occurs again and again. To be sure that this error nearly never occurs, the segment size can be set to 10^8 but before testing this new size be aware of the fact that the execution takes much more time than before or that the computer may not have enough memory. It is important to increase the segment size in a intelligent way, not in small steps to avoid a lot of executions for nothing and not too big to avoid memory overflow.

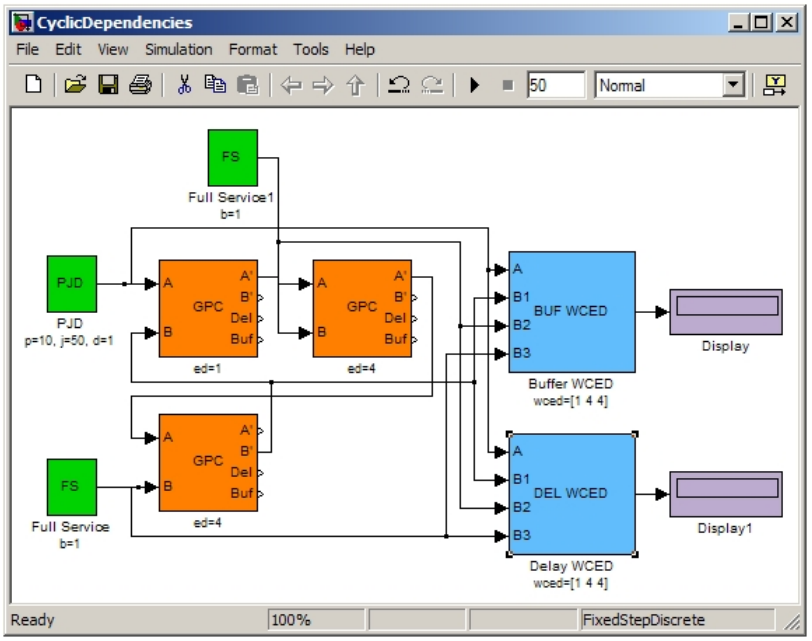
The number of segments can be set and got at the Matlab command prompt by invoking these functions:
`rtcsetNumSegments(len)` and `rtcgetNumSegments()`.

4.2.2. Initial Curve Problem

All models can be divided into two main groups of systems: non-cyclic dependency and cyclic dependency systems. For the second kind of system an error can appear. For



(a) Simulink Representation when task 1 has higher priority than task 3



(b) Simulink Representation when task 3 has higher priority than task 1

Figure 4.3.: Simulink Representation of the Tutorial Example

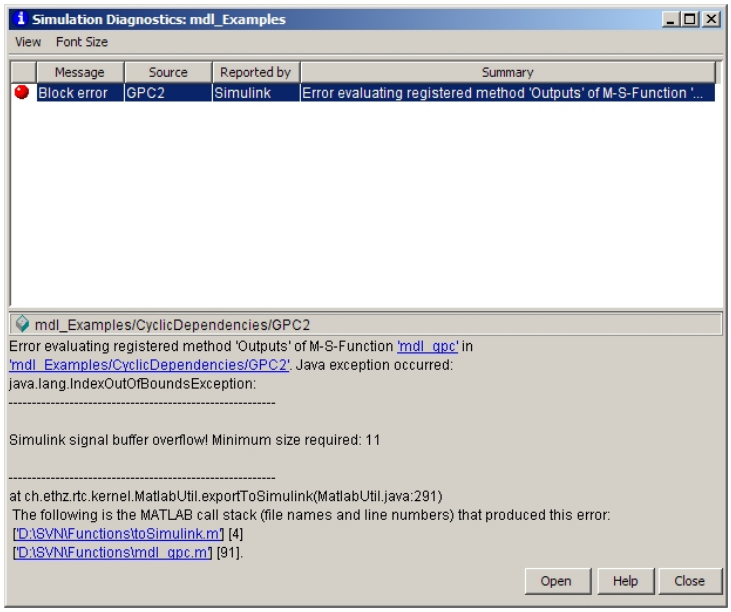
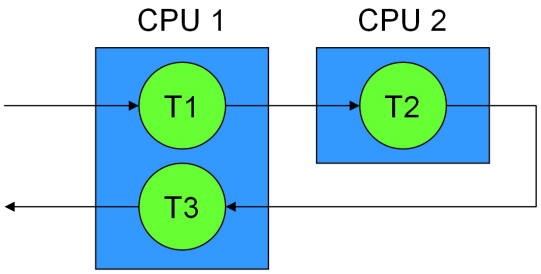


Figure 4.4.: Error Message if the number of segments needs to be increased

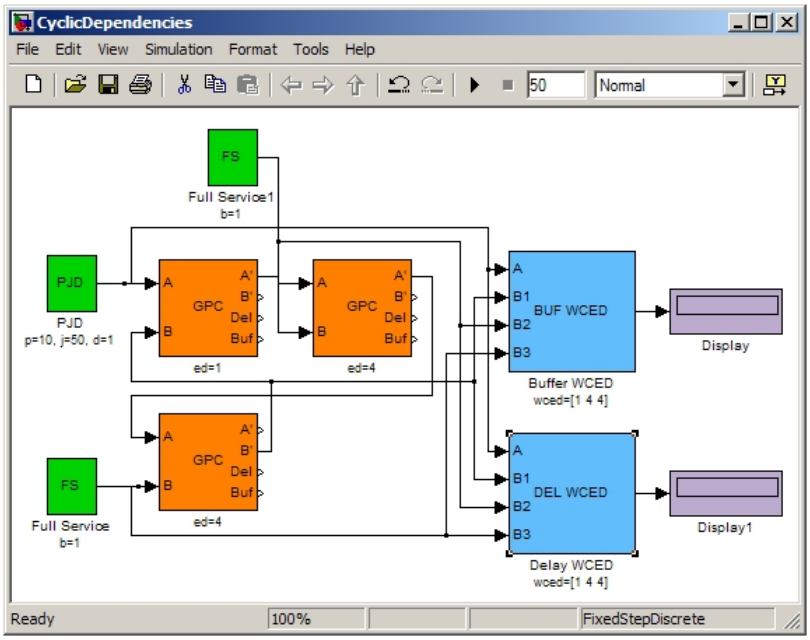
example: There are two tasks running on the same CPU. For the scheduling task two has higher priority than task one. The event stream goes from task one to task two. Task one has events to execute but can only run while task two is idle and task two can only run when task one already executed some events. In Simulink a problem emerges because of the execution order of the blocks. For example: Simulink wants to run the first task but this one has no service curve as input because it depends on task two while task two can not execute either because there is no the event curve at the input. None of the tasks can be run. The initial curves solve this problem because each block can execute its first time but causes another problem.

The initial curves can influence the whole analysis. As an example the tasks have as initial service curve a full service curve with bandwidth one (CPU can execute one task at the time) and an event curve which is non zero (e.g. pjd(10,50,1)). Let us assume that the execution order of Simulink first executes task two afterwards task one. The first execution of task one runs with the initial curves. The service curve at the output is not anymore a full service curve of bandwidth one because some events of the initial arrival curves are executed by task two but in the real model there cannot be any events before task one has executed some events before. Therefore task one should get a full service curve of bandwidth one after the execution of task two with the initial curve values. This small difference can change the values for buffer and delay. In table 4.2 are the

values for an example which is shown in Figure 4.5(a) and the Simulink representation in Figure 4.5(b).



(a) Model for initial curve problem



(b) Simulink Representation

Figure 4.5.: Simulink Representation for the initial curve problem

Input	Initial Curve in GPC	Delay & Buffer
β = FS(1) α = PJD(10,0,0)	β=FS(1) α=PJD(10,50,1)	Delay = 21 Buffer = 3
β = FS(1) α = PJD(10,0,0)	β=FS(1) α=PJD(1,0,0)	Delay = 13 Buffer = 2

Table 4.2.: Initial Curve Problem

4.3. List of Components

All components are listed and shortly described in this chapter. Here it will be distinguished between a curve and a curve set. This means: an α (arriving) or β (service) curve set consists of an upper curve and a lower curve. Most blocks in Simulink have curve sets as inputs or outputs.
The functions are sorted the same way as in the RTC Toolbox: curves, analysis, processing and scheduling components, complex task activations, operations and display components. If a block needs a parameter it can be entered by double clicking the block. In the block mask is a guide how to enter the values. If an error occurs during the execution, call the help function which can be found in the column „Help-Function“ of the required block. This help function can be used on the Matlab command prompt (e.g. `help rtcfs`). In the following tables BN means block name and HF stands for Help-Function. ED is called execution demand and is the time an event needs to get executed. In some cases the best case execution demand (BCED) and the worst case execution demand (WCED) are different.

4.3.1. Curves

These functions generate an arrival or service curve.

BN	Description	Parameter	Outputs	HF
BD	Bounded Delay	delay, bandwidth	β-curve set	rtcbd
Curve	any curve	see the mask or help	an upper or lower curve	rtccurve
FS	Full Service	bandwidth	β-curve set	rtcfs
TDMA	Time Division Multiple Access	slot length, cycle length, bandwidth	β-curve set	rtctdma

PJD	(P,J,D) event model	period, jitter, minimum event interarrival time	β-curve set	rtcpjd
PS	Periodic Service	share, period, bandwidth	α-curve set	rtcps

Table 4.3.: Curve Components

4.3.2. Analysis

These blocks calculate the occurred delay and required buffer size.

BN	**Description**	**Parameter**	**Inputs**	**Outputs**	**HF**
BUF	Buffer		1 α-curve set, n β-curve sets	required buffer	rtcbuf
BUF WCED	Buffer (WCED needed for calculation)	WCED	1 α-curve set, n β-curve sets	required buffer	rtcbuf
DEL	Delay		1 α-curve set, n β-curve sets	occured delay	rtcdel
DEL WCED	Delay (WCED needed for calculation)	WCED	1 α-curve set, n β-curve sets	occurred delay	rtcdel
Curve to BD	Buffer and Delay of a Bounded Delay Curve		β-curve set	occurred delay, required buffer	rtcbd

Table 4.4.: Analysis Components

4.3.3. Processing and Scheduling Components

These blocks model tasks and scheduling strategies

BN	Description	Parameter	Inputs	Outputs	HF
EDF	Earliest Deadline First	deadline and WCBC	n α-curve sets, 1 β-curve set	n α-curve sets, 1 β-curve set, occurred delay and required buffer	rtcedf
FIFO	First In First Out	WCBC	n α-curve sets, 1 β-curve set	n α-curve set, 1 β-curve set, occurred delay and required buffer	rtcfifo
GPC	Greedy Processing Component	ED	1 α-curve set, 1 β-curve set	1 α-curve set, 1 β-curve set, occurred delay and required buffer	rtcgpc
GSC	Greedy Shaper Component		α-curve set, a shaping-curve for S	α-curve set, occurred delay and required buffer	rtcgsc

Table 4.5.: Processing and Scheduling Components

4.3.4. Complex Task Activations

BN	Description	Input	Outputs	HF
AND	compute the input to a component with AND activation	2 α-curve sets	one α-curve set	rtcand
OR	compute the input to a component with OR activation	2 α-curve sets	one α-curve set	rtcor

Table 4.6.: Complex Task Activations Components

4.3.5. Operations

Operations on Curve Sets

The blocks here are operations, which can be done on curve sets.

BN	Description	Input	Outputs	HF
Ceil	computes the ceil of curve set	α or β-curve set	α or β-curve set	rtcceil
Floor	computes the floor of curve set	α or β-curve set	α or β-curve set	rtcfloor
Steps	computes the floor of the lower curve and the ceil of the upper curve	α or β-curve set	α or β-curve set	rtcsteps
Affine	an affine transformation of curve set	α or β-curve set, 2 values	α or β-curve set	rtcaffine
Hdist	computes the max horizontal distance between 2 curve sets	2 α or β-curve sets	α or β-curve set	rtch
Vdist	compute the max vertical distance between 2 curve sets	2 α or β-curve sets	α or β-curve set	rtcv

Max	maximum of 2 curve sets	2 α or β-curve sets	α or β-curve set	rtcmax
Min	minimum of 2 curve sets	2 α or β-curve sets	α or β-curve set	rtcmin
Minus	substraction of 2 curve sets	2 α or β-curve sets or a scalar	α or β-curve set	rtcminus
Plus	addition of 2 curve sets	2 α or β-curve sets or a scalar	α or β-curve set	rtcplus
Equal	equality check of 2 curve sets	2 α or β-curve sets	α or β-curve set	rtceq
Uminus	unary minus of curve set	α or β-curve set	α or β-curve set	rtcuminus
Split	splits a curve set into curves	α or β-curve set	α or β the upper and lower curve	
Divide	division of a curve set with a scalar	α or β-curve set and a scalar	α or β-curve set	rtcrdivide
Multiply	multiplication of a curve set with a scalar	α or β-curve set and a scalar	α or β-curve set	rtctimes

Table 4.7.: Operation on Curve Set Components

Operations on Curves

The blocks here are operations, which can be done on curves.

BN	**Description**	**Input**	**Outputs**	**HF**
Join	join the lower and upper curve to a curve set	α or β the upper and lower curve	α or β-curve set	
MaxConv	max-plus convolution of curves	α or β-curve sets or scalars	α or β-curve set	rtcmax-conv
MinConv	min-plus convolution of curves	α or β-curve sets or scalars	α or β-curve set	rtcmin-conv

Max-Deconv	max-plus deconvolution of curves	α or β-curve sets or scalars	α or β-curve set	rtcmax-deconv
Min-Deconv	min-plus deconvolution of curves	α or β-curve sets or scalars	α or β-curve set	rtcmin-deconv

Table 4.8.: Operation on Curve Components

4.3.6. Display Components

These blocks can plot curves or display numbers like delay or buffer size. PlotGPC and PlotGSC are the same as GPC and GSC but with an added plot function.

BN	**Description**	**Parameter**	**Inputs**	**Outputs**	**HF**
PlotGPC	compute and plot the output of a greedy processing component	ed, X-Values	1 α-curve set, 1 β-curve set	α and β-curve sets, occured delay and required buffer	rtcplotgpc
PlotGSC	compute and plot the output of a greedy shaper component	X-Value	α-curve set, a shaping-curve for S	α-curve set, occured delay and required buffer	rtcplotgpc
Plot	plot a curve or curve set	parameter for the appearance of the curve	n α and β-curves or curve sets	plot of the curves or curve sets	rtcplot
Display			numbers (no vectors)	shows the number	

Table 4.9.: Display Components

5. Conclusions and Outlook

5.1. Conclusions

This work described how a graphical user interface for the analysis of real-time embedded systems was implemented. It was done in form of a Simulink library which uses the MATLAB Real-time calculus toolbox. A protoype of such a library was already given but it had some limitations which could be overcome.
Basically the prototype had two major problems, the first concerned the representation of variability characterization curves (VCC) in Simulink and the second problem was that the blocks of the prototype were unable to deal with cyclic systems.
A solution to the first problem was to change the Java-Core of the RTC toolbox. Simulink cannot handle objects but only matrices, so the VCCs needed to be converted into matrices in an efficient way. Functions which do exactly this were already implemented in the Java-Core but they used a fixed number of segments to represent curves. That was an unflexible approach because curves might grow larger than the predefined number of segments and then the accuracy would be lost. So two functions were added to the Java-core which allow to get and set the number of curve segments that the library should use. If a curve grows larger than this value, Simulink will issue an error message and the user can define a higher number.
For the second problem it was explained how cyclic dependencies can lead to a situation where some blocks do not get any data to process. In such a case Simulink will issue an error message and stops the analysis. In the first implementation the blocks took the inputs, processed them and assigned the results to the outputs but with cyclic dependencies this approach does not work. To solve this problem, the implementation of the blocks had to be changed to include data storage elements. These elements were then initialized with predefined curves and then in a second step overwritten with the actual inputs of the block. Now the blocks took the values from the data storage elements and computed the outputs with them. This means that every block has always some data to process, Simulink will no longer complain about algebraic loops. The drawback of this solution is that for non-cyclic systems the analysis has to run for two

timesteps, since at the first timestep only the initial values are taken into account for the analysis. After this step the storage elements are updated with the actual input data and the right output is produced. The same blocks can be used to analyze either cyclic or non-cyclic systems but for non-cyclic systems the analysis time is twice as high as an optimal solution.
This new graphical user interface presents important improvements for the analysis of distributed embedded systems: Whereas the modeling of systems with only the RTC toolbox implies the writing of MATLAB-code and this is time-consuming and error-prone, now with the Simulink library it should speed up the development time and be much more user-friendly.

5.2. Outlook and Future Work

Although the library should be very useful already in the current state there are some points which could further improve it:

- The initial values for the curves are hard-coded into the library blocks. As there exist examples which may show different results (i.e. in terms of delay or buffer requirement) with different initial values it would be better if the user can define these values by himself. A first try to implement such a functionality was not successful. The idea was that a user can define initial curves with special names in the MATLAB workspace but somehow the Simulink model was not able to find and use these curves. In the work another possibility was presented but not implemented: It would require to change all blocks to have the initial curves as additional parameters.

- In the MATLAB code it is possible to stop the iterations when a fixpoint is reached and the used number of computation steps can easily be accessed. However achieving this functionality in Simulink was not possible during this work. It was found out that it does not suffice to look for a curve which does no longer change because the curves stay the same for some fixed number of timesteps and then change. Only when the fixpoint is reached the curves stay the same for more timesteps than the fixed number. If one analyzes this behaviour more in depth (i.e. what determines the fixed number of timesteps) it might be possible to build a „stop“-block.

- In the current implementation there exists a one to one relationship between blocks and functions from the RTC toolbox. For future usage it would be good to increase the abstraction level of the user interface such that the user requires only very few knowledge about the underlying RTC toolbox. This means that the GUI should become more 'intelligent' to make the task of the user easier. For example the modeling of the fixed priority scheduling could be changed. Instead of connecting service curves appropriately, it would be much easier if the user could just select a scheduling strategy for some tasks and set the priorities and the rest would be done in the background. Another intelligent behavior would be if the model can detect a chain of tasks and therefore automatically choose the convolution variant for the computation of delays to get a tighter bound.

- Quite another approach to redesign the library would be to completely detach it from the RTC toolbox. The GUI would then generate a tool-independent XML description of the distributed embedded system which could then be used through some converters as input for several different analysis methods (like SymTA/S, MPA or PESIMDES).

A. Appendix

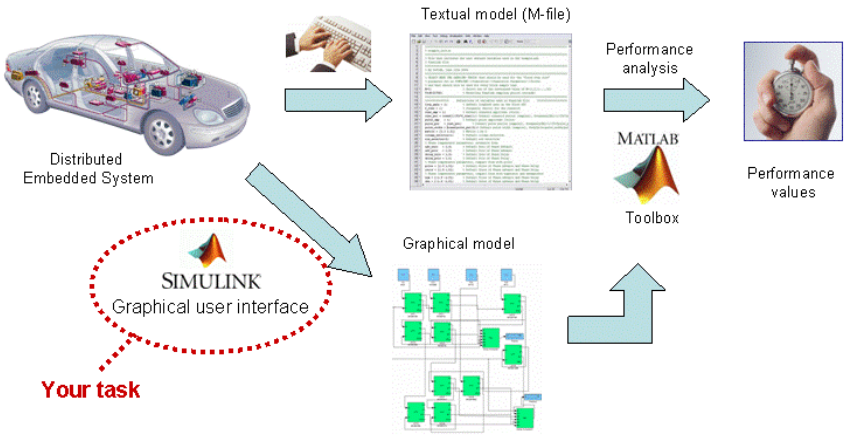
A.1. Covered Functions

All functions from the folders Curve and MPA have been implemented. From the folder Operations three functions are missing, namely rtcplotbounds, rtcplotv, rtcploth.
The plot function of the library was copied from the prototype.

A.2. Original Task Description (german)

Entwicklung einer Simulink Toolbox für die Analyse von Real-Time Embedded Systems

Am TIK wurden in den letzten Jahren erfolgreich Modelle und Methoden entwickelt, um die Performance von verteilten eingebetteten Systemen in einer frühen Designphase zu analysieren. Dabei werden beispielsweise Latenzen von Ereignissen oder Bufferlevels auf Systemebene untersucht. Die entwickelten Modelle und Methoden wurden in Form einer Matlab-Toolbox implementiert, um die Forschungsresultate auf grosse, reale Systeme anwenden zu können. Im Rahmen dieser Arbeit soll basierend auf der bestehenden Matlab-Toolbox eine Simulink-Toolbox entwickelt werden.
Während man in der Matlab Toolbox die zu analysierenden Systeme von Hand programmieren muss, würde die Simulink-Toolbox als graphisches User Interface dienen, um die Systeme als Blockdiagramme zu entwerfen. Dieses Interface würde die zugrunde liegende Matlab-Toolbox einem breiteren Publikum zugänglich machen und würde die Entwurfszeit erheblich verkürzen.



A.3. Real-Time Calculus

The min-plus convolution $\otimes$, the min-plus deconvolution $\oslash$, the max-plus convolution $\overline{\otimes}$ and the max-plus deconvolution $\overline{\oslash}$ of two functions f and g are defined as:

$$\begin{aligned}
(f \otimes g)(\Delta) &= \inf_{0 \le \lambda \le \Delta} \{f(\Delta - \lambda) + g(\lambda)\} \\
(f \oslash g)(\Delta) &= \sup_{\lambda \ge 0} \{f(\Delta + \lambda) - g(\lambda)\} \\
(f \overline{\otimes} g)(\Delta) &= \sup_{0 \le \lambda \le \Delta} \{f(\Delta - \lambda) + g(\lambda)\} \\
(f \overline{\oslash} g)(\Delta) &= \inf_{\lambda \ge 0} \{f(\Delta + \lambda) - g(\lambda)\}
\end{aligned}$$

A.4. Implementation of the dwork

The printed source code is from the GPC block and it shows the implementation of the dwork.

Function DoPostPropSetup

```
block.NumDworks = 2;                          % number of needed dworks
l = rtcgetDataLength();                       % length of the curves
for c = 1:2
  block.Dwork(c).Name=strcat('curve',mat2str(c)); % dwork's name
  block.Dwork(c).Dimensions = 6*l;                % array size
```

```
  block.Dwork(c).DatatypeID = 0;                  % 'double' = 0
  block.Dwork(c).Complexity = 'Real';             % real values
end
```

Function Start

```
alpha = rtcfs(0); % initial alpha curve
beta  = rtcfs(1); % initial beta curve

% convert curve set into a matrix
alpha1= [ toSimulink(alpha(2)) toSimulink(alpha(1)) ];
beta1 = [ toSimulink(beta(2))  toSimulink(beta(1))  ];

block.Dwork(1).Data=alpha1(:); % stored in the dwork
block.Dwork(2).Data=beta1(:);
```

Function Output

```
l = rtcgetDataLength();         % length of the curves
ip=zeros(l,6);
ip(:)=block.Dwork(1).Data(:);   % dwork stored in a new variable
ipl = fromSimulink(ip(:,1:3));  % lower alpha curve
ipu = fromSimulink(ip(:,4:6));  % upper alpha curve
alpha = [ipu ipl];              % convert from matrix to curve set object
ip(:)=block.Dwork(2).Data(:);   % dwork stored
ipl = fromSimulink(ip(:,1:3));  % lower beta curve
ipu = fromSimulink(ip(:,4:6));  % upper beta curve
beta = [ipu ipl];               % convert from matrix to curve set object
...
```

Function Update

```
for k=1:block.NumInputPorts
    data = block.InputPort(k).Data; % input curve stored in a matrix
    block.Dwork(k).Data=data(:);    % matrix stored in an array of dwork
end
```

Bibliography

[1] S. Chakraborty, S. Künzli, L. Thiele, 2003, *A general framework for analysing system properties in platform-based embedded system design*, In Proc. 6th Design, Automation and Test in Europe (DATE), pages 190195

[2] Ernesto Wandeler, 2006, *Modular Performance Analysis and Interface-Based Design for Embedded Real-Time Systems*, PhD Thesis ETH Zrich

[3] Jean-Yves Le Boudec, Patrick Thiran, 2001, *Network calculus: a theory of deterministic queuing systems for the internet*, Springer-Verlag New York

[4] SymTA/S Homepage, `www.symtavision.com`, accessed 18.12.2007

[5] Simon Perathoner, Ernesto Wandeler, Lothar Thiele, 2006, *Evaluation and Comparison of Performance Analysis Methods for Distributed Embedded Systems*, TIK Report Nr. 276

[6] Lothar Thiele, Ernesto Wandeler, 2005, *Performance analysis of distributed embedded systems*, Embedded Systems Handbook, CRC Press

[7] Lecture Notes for Hardware/Software Codesign AS2007, Prof. Lothar Thiele, `http://www.tik.ee.ethz.ch/tik/education/lectures/hswcd/slides/10_ModularPerformanceAnalysis.pdf`, accessed: 05.12.2007

[8] Exercise 11 of Hardware/Software Codesign Lecture. Prof. Lothar Thiele, `http://www.tik.ee.ethz.ch/tik/education/lectures/hswcd/exercises/exercise11_solution.pdf`, accessed: 30.12.2007

[9] MPA Homepage, `http://www.mpa.ethz.ch/`, accessed: 04.12.07

[10] `http://www.mathworks.com/products/simulink/description1.html`, accessed: 23.10.2007

[11] `http://www.mathworks.com/access/helpdesk/help/toolbox/simulink/`, accessed: 23.10.2007

List of Figures

List of Tables