Concept and Implementation of a Jitter-Robustness-Analysis of Software for Reactive Embedded Systems

Vlad Popa
Concept and Implementation of a Jitter-Robustness-Analysis of Software for Reactive Embedded Systems

Konzeption und Implementierung einer Jitter-Robustheits-Analyse von Software für reaktive eingebettete Systeme

Author: Vlad Popa
Supervisor: Prof. Dr. Dr. h.c. Manfred Broy
Advisors: Wolfgang Schwitzer, Martin Feilkas
Date: June 15, 2011
Disclaimer

Ich versichere, dass ich diese Bachelorarbeit selbständig verfasst und nur die angegebenen Quellen und Hilfsmittel verwendet habe.

I assure the single handed composition of this bachelor’s thesis only supported by declared resources.

München, den 15. Juni

Vlad Popa
Abstract

Jitter-robustness is a property of distributed embedded systems which communicate through buses. The interaction between the embedded control units can be disturbed by jitter. In this thesis, we do not try to eliminate the jitter sources from a technical point of view. Analysis and improvement techniques are presented in order to make the system robust against jitter. The FOCUS theory together with its delay calculus allow for the formal description of these methods. Furthermore, AutoFOCUS 3 is the main tool in which the jitter-robustness analysis and improvement procedures are implemented and tested. A proof of concept on embedded hardware and the case study of the DENTUM Adaptive Cruise Control systems show the practical results which can be achieved by applying the presented methods.
1 Introduction

1.1 Motivation

Embedded software systems are present in the daily life as part of mobile phones, cars or airplanes. These systems are mostly reactive. There is a permanent interaction between the system and its environment. Sensors read the environment inputs, which are processed and sent as outputs to the actuators. It is very hard to realize the software with only one embedded control unit (ECU). In a car there would be a cabling problem for reaching all sensors and actuators from a single processing unit. This is why the embedded software system is distributed among multiple processors, which are linked through bus systems. Because of different reasons like conflicts on the MAC-layer, the expected transfer time on bus systems can be affected. The deviation from this expected transfer time is called Jitter. If this jitter is not treated properly, it can damage the correctness of the system outputs.

Contribution of this thesis. Metaphorically speaking, the jitter-robustness is like an airbag to the system, which improves the degree of tolerated deviation in transmission time. From the software developers perspective a new design approach is introduced. After designing the new system, the developer has the possibility to analyze its jitter-robustness and finally improve it on demand. All the jitter-robustness analysis and improvement techniques can be done statically before runtime. For example, some software systems in the automotive industry like the ABS (Anti-Blocking-System) should be as jitter-robust as possible. In this case a system failure could have consequences for the driver.

1.2 Problem Statement

In this thesis we purpose an analysis of the jitter-robustness of a system regarding its logical and technical architecture together with their deployment mapping (see figure 1). With transformation methods like the retiming-transformation (see section 2.2.5) we try to improve the systems robustness against jitter.

Considered aspects. The analysis is formulated on the structural level of the logical architecture. This means, in this thesis the behavior of the components is not taken into account.

Figure 1 shows a logical architecture containing three components: $C_1$, $C_2$, $C_3$ communicating through channels: $m_1$, $m_2$ and a technical architecture with two CPU’s which exchange information through a bus. The deployment mapping assigns the components $C_1$, $C_2$ to the first CPU, while the component $C_3$ runs independently on the second CPU. $C_1$ sends inside the first CPU a message to $C_2$ which communicates with $C_3$ through the bus. Channel $m_2$ is mapped on the bus and is responsible for this communication. This is
an example of a possible scenario where jitter can affect the inter-processor communication (IPC) between two software components ($C_2, C_3$).

### 1.3 Goals and Contributions of this Thesis

The main goals are to analyze and improve the jitter-robustness of the software for reactive embedded systems. Furthermore, the presented techniques are evaluated with the case tool AutoFOCUS 3. For evaluation we use the DENTUM Adaptive Cruise Control model from the automotive area as a basis.

*The major contribution of this thesis is the description of a linear programming based jitter-robustness optimization method.*

In this thesis the following topics are covered:

1. Conception of an analysis method, which will determine for each component of a model its robustness to jitter.
   - Deployment of a logical architecture on a technical architecture is known.
   - This method returns the jitter-robustness in logical ticks of the system.

2. Develop a concept for improving the jitter-robustness.
   - This concept can change the causality of the logical architecture but the input/output behavior of the deployed system has to stay the same.
   - A relevant transformation technique, which works on data flow graphs is called retiming-transformation. A detailed description of this can be found in [LS91].
   - The improvement is constrained to the structural level.
3. Prototypical implementation of these concepts in AutoFOCUS 3.
   - AutoFOCUS 3 should support causality with arbitrary order.
   - The retiming-transformation has to be adapted to the logical architecture of AutoFOCUS 3.

4. Evaluation of the concepts with case examples.

Given a logical model and its deployment, the reader should be able to calculate its jitter-robustness in abstract time units (ticks) and should know the possibilities how to improve it.

1.4 Overview

This thesis consists of several sections which are structured as follows. The 2. section provides the background of concepts from related work which is often used. The main development artifact is the model-based development (see section 2.1). This methodology is implemented in the case tool AutoFOCUS 3 which is presented in section 2.3. The jitter-robustness analysis relies on the FOCUS theory with its delay calculus which is briefly described in section 2.2. Section 3 presents some methods used to analyze and improve the jitter-robustness of the system. In order to make this methods in AutoFOCUS 3possible, the following section 4 extends the functionality of this tool. Afterwards, section 5 presents a proof of concept on embedded hardware and the results of the practical case example, the DENTUM Adaptive Cruise Control model, from the automotive industry. Furthermore, a discussion section 6 clarifies some aspects in this thesis and the concluding section 7 describes new paths and approaches on how new methods can be developed in future work.
2 Related Work and Concepts

This section covers the explanation of the concepts used in this thesis. It contains a description of model-based development, followed by the presentation of the FOCUS theory and the iterative data flow graphs. In section 3 we use the FOCUS delay calculus theory as a concept for analyzing the jitter-robustness. Nevertheless, a brief introduction into AutoFOCUS 3 presents the features of the case tool used in this thesis.

2.1 Model-Based Software Development

Models, metamodels and meta-metamodels. Nowadays, there are complex embedded software systems which react on inputs by producing outputs. Most of the times these systems are distributed and can not be generally realized with only one embedded control unit (ECU). The model-based development is adequate for this kind of systems because it aims to reduce the development complexity with the help of abstraction. In this context it is very common to define the meta-metamodel, the metamodel and the model. These three layers of abstraction should be sufficient for describing the system. The model is a instantiation of a metamodel, which in turn is a instantiation of a meta-metamodel (see figure 2). Textual or graphical representation of the system defines a model. For example, if the model is a state diagram, the metamodel represents the artifacts.
used for creating the model and their relationships. In this case the metamodel contains states and transitions. Each state can contain other states and can have any number of ingoing/outgoing transitions. The transitions exist between two states. Therefore, the meta-metamodel includes the associations and metamodel-classes used in the metamodel. More information to this is found on the OMG-Websites at the MOF-standard [MOF].

**Granularity levels and perspectives.** As shown in Figure 3, the amount of detail is reduced and abstraction is achieved by:

- Decomposition of the systems into its sub-systems, which have a lower level of granularity. The sub-systems which do not contain other sub-systems are called basic blocks.

- Consider different development perspectives like: requirements perspective, functional perspective, logical perspective, technical perspective. More details are found here [TRS+10].

![Image of different granularity levels and software development perspectives](image)

Figure 3: Different granularity levels (vertical) and different software development perspectives (horizontal) according to [TRS+10].

The main challenge with model driven development is the mapping of models from different perspectives. Models with the same level of granularity have to be mapped together (*horizontal allocation*) and attached to their sub-systems (*vertical allocation*). A comprehensive modeling theory which can solve this challenges by creating a mathematical
model is called FOCUS theory \cite{RST09}. A case tool which is based on this theory is called AutoFOCUS 3\cite{Aut}.

Figure 3 shows the different granularity levels on the vertical axis and the software perspectives on the horizontal axis. In the requirements perspective timing and performance constraints can be set. These determine the end-to-end latency which influences the jitter-robustness. The functional perspective is used to describe the system behavior. In this thesis, the jitter-robustness is analyzed regarding the logical and technical perspective. The three marked rectangles in the bottom right corner of figure 3 show the granularity levels of the perspectives which are relevant for the jitter-robustness in this thesis.

2.2 Theoretical Foundation

In this section, a fundamental way of describing systems called FOCUS is introduced. With the help of stream-processing functions, this theory efficiently describes systems of components, which process a specific function.

2.2.1 FOCUS and Stream-Processing Functions.

A stream is a finite or infinite ordered sequence of elements. Let $M$ be a set of messages. We use the following notation:

- $M^*$: contains all finite sequences over $M$ including the empty sequence $\langle \rangle$.
- $M^\infty$: contains all infinite sequences over $M$.
- $M^\omega$: contains all sequences over $M$ ($M^\omega = M^* \cup M^\infty$).

For the channel histories it is useful to consider the timed streams: $(M^*)^\infty$ which consist of an infinite stream of finite sequences. For each time $t \in \mathbb{N}$ (in ticks) the element on position $t$ of the timed stream represents the finite sequence that is being sent on the channel in that tick. To operate with timed streams, some functions are being introduced in \cite{BS01} and \cite{Bro10}:

- $x.t$ gets the $t$-th element of the stream $x$.
- $\#x$ gets the number of elements the stream $x$ contains.
- $x \triangleq y$ concatenates the two streams.
- $x \downarrow t$ gets the prefix of length $t$ of $x$. If: $t \geq \#x \Rightarrow x$ is returned.
- $S(t) x$ returns a stream from $x$ by filtering only the elements contained in $S$.
- $x \sqsubseteq y \equiv \exists z : x \triangleq z = y$.
- $\pi$ returns the sequence by concatenating all the elements of the timed stream $x$. 

12
For specifying the functionality of a system, FOCUS uses schemes as presented in table 1. The first row provides a black-box view of the analyzed component. Each input/output channel is associated with a specific data type (or set of messages). The second row is used to describe from a glass-box view the functionality of the component. One can apply here the stream-processing functions presented above.

<table>
<thead>
<tr>
<th>in</th>
<th>&lt;input channels&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>out</td>
<td>&lt;output channels&gt;</td>
</tr>
<tr>
<td></td>
<td>&lt;expression&gt;</td>
</tr>
</tbody>
</table>

Table 1: FOCUS specification of a system.

Figure 4 and the first row of the FOCUS specification table 1 illustrate the syntactical interface of a component. This representation does not give details about the causality of a system. It only defines its structural level. For each component a set of input and output channels express how many messages are being processed and returned. This provides a black-box perspective of the system which is important for the jitter-robustness analysis and improvement in this thesis.

**Example 2.2.1.** Let there be two sets $A$ and $B$ which both contain a third set $C$. In other words, the following holds: $A = C \cup X_1$ and $B = C \cup X_2$, where $X_1$ and $X_2$ are subsets of $C$ and $|X_1|, |X_2| < C$. Furthermore, we want to build a component $F$ with a syntactic interface like the one seen in Figure 4. $F$ is responsible for filtering from both input

![Diagram](attachment:filter_function_diagram.png)

Figure 4: The filtering function which has two inputs from the sets $A$ and $B$ and filters out the messages which belong to the set $C$. 

---

13
channels of type $A$ and $B$ the messages contained in $C$. Using the FOCUS notation, the scheme 2 provides all the semantic and syntactic information needed for the specification of the component $F$.

\[
\begin{array}{c}
\text{in} \\
\hspace{1cm} x : A, y : B \\
\text{out} \\
\hspace{1cm} z : C \\
\end{array}
\]

\[
\exists = C \circ \exists C \circ \exists y
\]

Table 2: FOCUS specification of component $F$.

### 2.2.2 Delay Calculus

**Guaranteed Delays and Delay Profiles** For the delay calculus we use terms such as guaranteed delays or delay profiles. In a network of components there can be many paths from arbitrary inputs that lead to the same output channel. For a output channel $o$ the guaranteed delay is equal to the minimum sum of delays on each of the paths that lead to this channel. We use the following notation for the guaranteed delays: $\text{gardelay}(F, o)$, where $F$ is the component and $o$ is an output channel of $F$. Consequently, let there be $F'$, the same system as $F$, with only the one input channel $i$. The delay profile is defined as a mapping that contains for each output channel $o$ and each input channel $i$ the guaranteed delay of $o$ in the new system $F'$.

A profound introduction and definition of the delay calculus is described by Broy in [Bro10]. This work is also pointing the way for the analysis presented in this thesis.

### 2.2.3 Causality, Delays and Buffers

In this thesis the terms causality, delays and buffers are frequently used. It is essential to understand the context in which these words are used because they all share similarities. The term causality is associated with components, which can be either strongly or weakly causal. Furthermore, all strongly causal components are supposed to delay the inputs by a positive and nonzero number of time units.

**Implementation of delays** Delays are typically implemented by FIFO-buffers. A delay of $x$ time units can be simulated by a first-in-first-out (FIFO) buffer of size $x$. Assuming that the pop- and push-method are being called once in every tick and that the buffer contains exactly $x$ elements, then the popped elements have to be $x$ ticks old. This method for the implementation of the delays has a certain amount of flexibility. If the messages are being received to slow (because of jitter) then the user, perceiving the system from the outside, will not notice a difference on the behavior until one of the buffers is empty. Therefore, such a principle for the delays implementation and consequently for the realization of causality can have a big impact on the robustness of the system.
2.2.4 Iterative Data Flow

According to [Par89, DK82, Ack82, Den80], the data flow representation of an algorithm is appropriate for expressing the available concurrency in a multiprocessor environment suitable for distributed embedded systems. The data flow language is presented in W.B. Ackerman’s article [Ack82]. Furthermore, general concepts of modeling systems with data flow program graphs are found in [DK82]. The iterative data flow model is used for those models which have a global clock. The functionality of these models is executed in each tick. In this thesis, the data flow programs, represented by data flow graphs, are synchronous and nonterminating.

1: for all $t = 1 \rightarrow \infty$ do
2: $z(t+1) \leftarrow x(t) \times 3 + y(t) \times 2$
3: end for

Figure 5: The data flow program and data flow graph.

Data flow programs and graphs. Figure 5 shows how a data flow program (a) is translated into a data flow graph (b). The data flow program computes in each time unit $t$ the expression $x(t) \times 3 + y(t) \times 2$ and assigns it to $z(t)$. The equivalent data flow graph is shown in (b). Data flow graphs are acyclic graphs containing environment inputs/outputs, vertices (tasks), edges (channels) and delays on edges. Environment interface points are represented by the dashed circles. Delay units appear in rectangles with the value ‘D’. They have the role to slow down the transmission by one time unit. Note, that there may be edges which do not contain any delay units. The vertices $A$, $B$ and $C$ of figure 5(b) are tasks executed in each step. These tasks describe functions of arbitrary arity that receive $n \geq 0$ inputs and produce $m \geq 0$ outputs. Hence, inputs $x(t)$ and $y(t)$ are multiplied by 3 respectively 2, added together and sent as the result $z(t+1)$.

Delay operators. Sometimes, outputs need to be delayed by several time units before they are received. Therefore, the data flow graphs allow delay units to be specified for each channel. If a channel from node $A$ to node $B$ contains $i$ delays, $B$ gets the message $A$ sent at time unit $t - i$ (for all $t > i$). Notice that for those ticks in which $t \leq i$, initial values have to be specified.
Homogeneous and heterogeneous data flow graphs. In this thesis, a homogeneous data flow graph is a graph in which all outgoing channels of a vertex have the same number of delays assigned. This definition introduces the term causality for data flow graphs. In a homogeneous graph the vertices which do not have any delays on the outgoing channels are called weakly causal. All other vertices are strongly causal. The heterogeneous data flow graphs are those graphs which accept outgoing channels with different delay numbers.

2.2.5 Retiming-Transformation

The retiming-transformation is a method for changing data flow graphs so that the end-to-end latency remains the same. Therefore, the system behavior remains the same on the outside. Retiming was first introduced by Charles E. Leiserson and James B. Saxe in [LS91] as a transformation method for synchronous circuitry. Initially, this transformation was used to reduce the minimal clock period and to improve the performance of the systems. In this thesis, the retiming-transformation will be adapted to the logical architecture of AutoFOCUS 3 (see section 3.3) in order to improve the jitter-robustness by shifting delays in the network.

The technique uses a data flow graph $G(V, E, \delta)$, where each vertex in $V$ can execute a function and edges in $E$ can contain a non-negative number of delays determined by the function $\delta : E \rightarrow \mathbb{N}$. Therefore, the following must be valid:

$$\forall e \in E, \exists n \in \mathbb{N} : n \geq 0 \land \delta(e) = n$$

This means, each edge gets assigned a positive number of delays. The retiming-transformation can be described with the help of the function $r : V \rightarrow \mathbb{N}$, with the property:

$$\forall v \in V, \forall e = (u, v) \in E : r(v) \leq \delta(e)$$

This means, the retiming-transformation is a number smaller or equal to the number of delays on all incoming edges. The new transformed graph $G_r$ has the same set of vertices $V$ and edges $E$ but a different delay function $\delta_r$. The values of this function are:

$$\forall e = (u, v) \in E : \delta_r(e) = \delta(e) + r(v) - r(u)$$

The retiming-transformation takes for each vertex $v$, $r(v)$ delay units from all inputs and adds them to all the output channels (Analog, retiming can also be done by moving the delays from the output channels to the input channels). Furthermore, the following two lemmas take place (proof can be found in [LS91]):

**Lemma 2.2.1.** For every path $P$ from $u$ to $v$ we have $\delta_r(P) = \delta(P) + r(v) - r(u)$, where $\delta(P)$ represents the number of delays on $P$.

**Lemma 2.2.2.** The number of delays on every cycle will not be modified.
Figure 6: The two possible retiming-transformations that can be executed on a component. Delays can either be moved from inputs to outputs or vice versa.

Retiming-transformation as a linear problem. In the late 80’s C.E. Leiserson and J.B. Saxe used the retiming-transformation to minimize the clock period. The algorithm which specifies how to retime the system is described in [LS91]. It transforms this problem into a linear problem by computing a system of inequalities of the forms:

\[
\begin{align*}
    r(x) - r(y) &\leq W_{xy} - 1 \\
    r(x) - r(y) &\leq \delta(e)
\end{align*}
\]

which then has to be solved. \(r(x)\) and \(r(y)\) stand for the retiming-transformation number of the two nodes \(x\) and \(y\). For inequality 1, \(W_{xy}\) represents the number of delays on the less delayed path between \(x\) and \(y\). If there is a edge \(e\) between \(x\) and \(y\) the inequality 2 comes instead. This is a difference-inequality system which can be solved by linear programming tools like \(lp\_solve\) [Lp] or SMT-solvers like \(Z3\) [Res]. In section 3.4.3 the optimization of the jitter-robustness with help of the retiming-transformation is presented as a linear problem which is efficiently solved by the tool \(lp\_solve\).

2.3 Introduction to AutoFocus 3

The scientific case tool AutoFOCUS 3 implements major parts of the FOCUS theory, which allows model-based development. It has a graphical view to model the system by ‘clicking’ components and channels. Initially, the AutoFOCUS 3 development was motivated by the lack of a case tool which does not concentrate exclusively on aspects like code-generation and user interface, or is too mathematical oriented for the system engineer [HS01]. This
section provides a brief introduction to the main features of AutoFOCUS 3.

**Timing model in AutoFOCUS 3.** In FOCUS theory systems consist of components which communicate with each other through communication channels [BS01]. Besides explaining the causality between the components which determine a specific data flow, this thesis will have to deal with problems caused by the transition from the real time, represented by real numbers, to the discrete time which will only contain discrete intervals. Discrete time advances in ticks, where two events within a tick will not be differentiated. In the context of this thesis, we use the same timing model as presented in [HHR09]. This means, in AutoFOCUS 3 we assume there is a global clock responsible for the synchronicity of time. Also, we assume that the time model is discrete and advances in ticks, so that events within a single tick can not be distinguished. This is a difference to the FOCUS theory, where events are asynchronous.

AutoFOCUS 3 consists of several graphical views which show different perspectives of the system under design, comparable to the perspectives in figure 3.

**System structure specification.** This specification shows a static view of the described system. It can be seen on the left side of the figure 7. According to [BFG+08], the elements of the system structure specification are:

- **Ports** let components communicate with the environment via channels. Each port has a data type and a name and is either an input port or an output port. An input port can only have one ingoing channel. An output port can connect to multiple outgoing channels.

- **Channels** realize the directed communication between components, binding them through their ports. They also have names and data types which have to be compatible with the ones from the involved ports. The logical architecture of AutoFOCUS 3 contains only directed channels. This means that the flow of messages goes in one direction (from the output port to the input port).

- **Logical components** can communicate with each other through channels, forming a network. They are entities containing interface, structural and behavioral information. Furthermore, the system structure specification can also be seen from a hierarchical point of view, where each component may have its own sub-components (observe similarities to model-based development in section 2.1). Each component communicates with the environment through its parent-component. All components have a syntactic interface, determined by their input and output ports, creating a black-box view of their functionality. The semantic interface contains behavior specifications like: stateful automaton specifications or stateless function specifications. This interface can be observed at the level of the atomic components, which can no longer be divided in sub-components. In the current implementation
of AutoFOCUS 3, components are weakly or strongly causal. Weakly causal means the outputs are sent within the same tick to the next component or environment. For strongly causal components the outputs are delayed by exactly one logical tick. An important constraint, is that weakly causal cycles are not allowed.

Automaton specification. This specification is part of the component behavior view and can be seen on the right side of figure 7. It consists of an automaton with control states, data variables, transitions, inputs and outputs. Among all control states there is an initial state, and all data state variables have to have a initial value [HST10]. Every transition is bound to a source and a destination state. It must contain an input pattern, preconditions, an output pattern and optionally postconditions. The input patterns check if the input ports have the desired values. Preconditions define a pattern containing data state variables and input ports which has to be true. The output pattern assigns new values to the output ports. Postconditions assign values to data state variables. Further clarifications to the syntax and semantic of the transition specification can be found in [Aut].

Figure 7: AutoFOCUS 3 structural and automaton view. In the structural specification view the PCS system is a atomic component, whose behavior is shown by the automaton on the right side. The input ports are represented by white circles and the output ports are shown as black circles.
2.3.1 Logical Architecture, Causal Components and Delays

The logical architecture contains information about the system structure and its behavior. The basic principle is to split the system into its suitable sub-systems, which communicate through channels and also form a hierarchical view. On the leaf level, sub-systems contain behavioral information. The main aims of the logical architecture are to:

- structure the system into communicating logical components.
- easily integrate existing components and increase reusability.
- make a seamless transition to the technical architecture.

Delays and causality in AutoFOCUS 3  For AutoFOCUS 3 the term causality is used to express the amount of delays on the outputs of the components. In the current implementation, a strongly causal component delays all outputs by exactly one tick. A weakly causal component does not delay its outputs at all. If a principle such as the retiming-transformation is adapted to the AutoFOCUS 3’s logical architecture, the strongly causal components will have to be able to delay the output by more than one time unit (see section 4.2).

Weakly and strongly causal components. As already described above, there are two component types: weakly causal components and strongly causal components. This section contains mathematical definition of the causality of a component based on FOCUS theory, using the notation from [BS01, Bro10]. Let $F$ be the function defining the I/O-behavior of a given component:

$$F : \overrightarrow{T} \rightarrow \varphi(\overrightarrow{O})$$

This function $F$ is called:

- weakly causal (or properly timed) if the formula: $x \downarrow t = z \downarrow t \Rightarrow (F.x) \downarrow t = (F.z) \downarrow t$ is true;

- strongly causal (or time guarded) if the formula: $x \downarrow t = z \downarrow t \Rightarrow (F.x) \downarrow t + 1 = (F.z) \downarrow t + 1$ is true.

In other words, if the output at a random time $t$ does not depend on input that is received after time $t$, the component is weakly causal. If the output at time $t$ does not depend on an input that is received after time $t - 1$, the component is strongly causal. A possibility to create strongly causal components is to delay all their output channels by exactly one time unit. This is how strongly causal components are implemented in AutoFOCUS 3. Therefore, a method to define the length of a logical tick can be described by assigning it length of the critical path (the longest time consuming chain of weakly causal components). If the tick is shorter, the weakly causal formula will not be valid. Notice that the weakly causal components must not contain any feedback loops (Brock-Ackermann anomaly see [BA81]). Otherwise, the longest chain will be infinitely long and the logical tick will never end.
Delaying messages. For $F$ a I/O-behavior describing function and $n \in \mathbb{N} \cup \{\infty\}$, we define:

$$
\text{delay}(F, n) \equiv [\forall x, z, t : x \downarrow t = z \downarrow t \Rightarrow (F.x) \downarrow t + n = (F.z) \downarrow t + n]
$$

[Bro10]. This binary function suggests the response delaying on all output channels by $n$ ticks. We define $F_s$ and $F_w$ as the I/O-behavior functions that are responsible for a strongly causal component and a weakly causal component. Therefore, delay($F_s, 1$) and delay($F_w, 0$) must hold.

Weakly and strongly causal components to data flow graphs. The concept of causality can also be adapted to iterative data flow graphs. Section 2.2.4 provides the theoretical background for the causality in homogeneous data flow graphs. Figure 8 illustrates how weakly and strongly causal components are translated to data flow graphs. The weakly causal component $F_w$ in figure 8(b) has the input interface $I_w = \{i_1, \ldots, i_n\}$ and the output interface $O_w = \{o_1, \ldots, o_m\}$. There are no delay units on the output channels which means, the result is passed on to the next component within the same logical tick. The values on the output channels are:

$$
o_1(t) = F_w(i_1(t), \ldots, i_n(t)) = o_1(t), \ldots, o_m(t) = F_w(i_1(t), \ldots, i_n(t))
$$

Figure 8(a) sketches the translation of the strongly causal component $F_s$ into a data flow graph. The syntactic interface of component $F_s$ consists of the input channels $I_s = \{i_1, \ldots, i_n\}$ and the output channels $O_s = \{o_1, \ldots, o_m\}$. Each output channel of the data flow graph vertex contains one delay unit. Hence, the output channel values in each time unit is:

$$
o_1(t + 1) = F_w(i_1(t), \ldots, i_n(t)) = o_1(t + 1), \ldots, o_m(t + 1) = F_w(i_1(t), \ldots, i_n(t)).
$$

2.3.2 Technical Architecture

The technical architecture describes the hardware on which the software runs (ECU’s, buses, sensors and actuators). An advantage of using a technical architecture is that the
hardware platform can be changed independently from the logical architecture. The aims of the technical architecture are to:

• present an abstract view of the hardware (ECU’s, buses, sensors and actuators) and its topology.

• describe the sensors/actuators, operating system and drivers.

• ensure that the deployed system fulfills the functional and non-functional requirements.

2.3.3 Deployment

The deployment describes the mapping of the logical components and channels to a specific hardware platform. If two components, which communicate through a channel in the logical architecture, are deployed on different processors, then that channel has to be mapped on a bus that binds the two processing units. Deployment is important to the jitter-robustness. A different deployment mapping can lead to a different robustness against jitter. As mentioned in section 1.2, all the methods presented in this thesis analyze and improve the jitter-robustness for systems with a specific deployment mapping.
3 Jitter-Robustness in Reactive Embedded Systems

After explaining in the previous section 2 the necessary concepts, this section covers the definition, analysis and improvement of the jitter-robustness.

3.1 Defining the Term Jitter-Robustness

For defining the term jitter-robustness, it is necessary to understand the words jitter and robustness. Firstly, the word robustness refers to a software quality aspect. Secondly, jitter expresses the lag in time which can occur while sending a message. Embedded control units (ECU’s) communicate with each other through bus systems. Therefore, some channels in a network of components may be mapped on buses. Sometimes messages sent on bus are being received sooner or later than the expected time. The deviation from this expected time is called jitter (see figure 9). Regarding these definitions, jitter-robustness is the amount of jitter, which does not affect the correctness of the outputs of a system. The outputs are considered correct if they are functionally correct and additionally have the correct timing. The aim of this thesis is to present the jitter-robustness as a metric that is measured in logical ticks.

Figure 9: The definition of jitter and the jitter-deviation which is analyzed and handled in this thesis.

Figure 9 shows in 9(a) two ECU’s which communicate with each other through a bus where jitter takes place. According to figure 9(b) the jitter-robustness is analyzed and improved for those messages that are received to late or with a low rate. If the incoming messages are received sooner than expected, the bus driver can save the data in ring-buffers.

Technical issues that cause jitter. Some embedded systems deliver safety critical outputs. An output is acceptable if it is semantically correct and it is delivered in due time. Especially these safety critical systems have to be as jitter-robust as possible.

This thesis will not try to explain from a technical point of view how to fight the possible reasons for jitter. It will try to cope with them!
The important technical issues that cause jitter are the following:

- **Resource conflicts/concurrency.** In complex systems there are usually more than two partners which exchange information on the same bus. There can be simultaneously many messages on the bus, which delay the sending of a specific message. For example, the CAN-bus (Controller Area Network, [Gmb]) introduces an arbitration principle which always sends the package with the lowest ID. That is why messages with higher ID have to wait all messages with lower ID’s are sent.

- **Electromagnetic compatibility (EMC).** Electromagnetic impulses can change the bits of messages that are sent. When data is received, some bus systems (e.g. CAN-bus) perform a cyclic redundancy check (CRC, see [PFTV]). This method is capable to detect any error caused by the EMC and hence, a time costing resubmission is necessary.

- **Synchronization.** The synchronization of all ECU’s over a longer period of time is not trivial. Especially for those systems which have a high operating frequency, the full synchronization over more units is very difficult. This problem is caused by the transition from the real time which is continuous to the discrete time representation of the machines. There always is a delta of time between the ECU’s that can grow over a longer period of time.

### 3.2 Analyzing Jitter-Robustness

In a top-down approach, the software developer designs the software, analyzes it and improves it if necessary. A main advantage in analyzing the jitter-robustness is that it can be done in an early phase. The purpose of this subsection is to define for a given system if a improvement of the jitter-robustness is needed. In the scope of this thesis the analyzed model has to match some assumptions:

- The technical architecture of the given system contains only one bus.
- The deployment mapping of the bus divides the logical architecture into two disjoint sets of components on two ECU’s.

Let there be $A$ the super-component that contains all components from one set and $B$ the super-component that contains all components from the other set. Furthermore, let there be $I_A$ and $O_A$ the sets of all input and output ports of $A$ and $I_B$ and $O_B$ the sets of all input and output ports of $B$. The super-components $A$ and $B$ can also be seen as functions: $A : \overrightarrow{I_A} \rightarrow \varphi(\overrightarrow{O_A})$ and $B : \overrightarrow{I_B} \rightarrow \varphi(\overrightarrow{O_B})$. For each system the jitter-robustness can be measured in ticks by the function $\rho$. Let there be $I_{BA}$ a subset of $I_A$ with following definition:

$$I_{BA} = I_A \cap \{ o \in O_B | \forall x \in I_B \Rightarrow (B.x).o \in \overrightarrow{I_A} \}$$  (3)
This means $I_{BA}$ contains all channels deployed on the bus from component $B$ to component $A$. If the following holds:

$$\forall o \in \overrightarrow{O_B} \Rightarrow \neg(\exists x \in \overrightarrow{I_B} \land (B.x), o \in \overrightarrow{I_A})$$

meaning that $S$ does not contain feedback channels from $A$ to $B$, then $I_{BA} = \phi$. Analog, let there be $I_{AB}$ a subset of $I_B$ with:

$$I_{AB} = I_B \cap \{ o \in O_A | \forall x \in \overrightarrow{I_A} \Rightarrow (A.x), o \in \overrightarrow{I_B} \}$$

(4)

For the jitter-robustness analysis we need to consider the following components:

$$A' : \overrightarrow{I_{BA}} \rightarrow \mathcal{P}(\overrightarrow{O_A})$$

(5)

$$B' : \overrightarrow{I_{AB}} \rightarrow \mathcal{P}(\overrightarrow{O_B})$$

(6)

Note that these new components have the same functionality as $A$ and $B$ restricted on the new input ports. Therefore, the jitter-robustness it equal to:

$$\rho(S) = \min\{\min\{\text{gardelay}(B', o) | \forall o \in \overrightarrow{O_B}\}, \min\{\text{gardelay}(A', o) | \forall o \in \overrightarrow{O_A}\}\}$$

(7)

This formula states that the jitter-robustness is equal to the number of causalities (delays) on the least delayed path from an input to an output from both components $A'$ and $B'$.

Figure 10: A valid AutoFOCUS 3 model satisfying the necessary constraints for jitter-robustness analysis in this thesis. $C_1$, $C_2$ and $C_3$ are feed-forward channels. Optionally the system may contain additional feedback channels ($C_4$ and $C_5$).
3.3 Retiming-Transformation in a Logical Architecture

As already mentioned in Section 2.2.5, the retiming-transformation is a technique for moving the delays in an data flow graph without changing the end-to-end latency. Retiming is the central technique for shifting delays in section 3.4.3. The algorithm of this transformation takes a constant number of delays from all input channels and adds it to all output channels or vice versa. A problem with logical architectures of AutoFOCUS 3 is that the corresponding data flow graph is always homogeneous. In other words, there is no surjective function which assigns for each data flow graph (homogeneous or heterogeneous) a logical architecture. Hence, the method for adapting the retiming-transformation to a logical architecture is not unique.

Adapting the retiming-transformation to the logical architecture. A logical architecture contains ports, channels and components (see Section 2.3). The retiming-transformation in a logical architecture is performed on the components. The only precon-
dition necessary for the transformation to take place is that all predecessor components are strongly causal. Predecessor components are those components that have at least one channel that leads to the analyzed component. This condition is equal to the one for data flow graphs where every input channel should have at least one delay to offer (see section 2.2.5). Note that a strongly causal component can contain more than one delay unit (for clarifications see section 4). Therefore, the retiming on a specific component can be done with an integer value greater than zero and smaller than the lowest causality order of all predecessor components. To express everything in mathematical terms, let there be \( C \) the set of all components of a system. We define the function \( \delta : C \rightarrow \mathbb{N} \) representing for each component the causality order it has. Hence, \( \delta(X) = 0 \) means component \( X \) is weakly causal. Furthermore, let there be function \( \pi : C \rightarrow \mathcal{P}(C) \) which returns for every component all its predecessors and function \( \mu : C \rightarrow \mathcal{P}(C) \) which returns for every component all its successors. The retiming-transformation is described as a function \( r : C \rightarrow \mathbb{N} \). For a component \( X \) with \( r(X) > 0 \) the transformation will increase the number of delays of this component by \( r(X) \). Let there also be the function \( \delta_r \) defined the same as \( \delta \). It represents the causality order for each component after the retiming-transformation. In order to preserve the end-to-end latency of the system \( r(X) \) causality delays on the predecessors output channels have to be removed. Therefore, the following holds:

\[
\forall P \in C : \delta_r(P) = \delta(P) - \sum_{\mu(P)} r(U) + r(P)
\]  

(8)

However, a problem still persists. For those paths from a component \( P \in \pi(X) \) that lead to an output and do not have \( X \) in it, the number of delays is with \( r(X) \) units smaller. This is because of equation 8. To fix this, an ID component, processing the identity function, has to be created for: \( \forall P' \in \mu(\pi(X)) \setminus X \). The ID component communicates with the predecessor \( P \) of \( X \) and its direct successor \( P' \). Furthermore, ID is strongly causal and has \( \delta_r(ID) = r(X) \).
Example 3.3.1. The following figures 11, 12, 13 and 14 will show step by step how the retiming-transformation operates in a logical architecture.

![Diagram showing the first step of the retiming-transformation](image1)

Figure 11: The first step of the retiming-transformation in a logical architecture.

There are three strongly causal components $A$, $B$ and $C$ with their corresponding delay functions $\delta(A) = y$, $\delta(B) = z$ and $\delta(C) = t$ in figure 11. Component $A$ needs to be retimed because $r(A) = x > 0$. The first step of the retiming-transformation is to ensure that the necessary preconditions are satisfied. Hence, $r(A) = x < \delta(B)$ must hold.

![Diagram showing the second step of the retiming-transformation](image2)

Figure 12: The second step of the retiming-transformation in a logical architecture.

The second step of the retiming-transformation is to add the necessary delays to the retimed components. Therefore, component $A$ gains $x$ causality delay units ($\delta_{r}(A) = x$).

\[\delta_{r}(A) = x + y\]
which causes the path $P_1$ that begins with $B$ and ends with $A$ to change its latency. This is signalized in figure 12 with the red color ($\delta(P_1) = x + y + z$ instead of $\delta(P_1) = z + y$).

In the third step of the retiming-transformation the number of delays on $P_1$ is adjusted. A consequence is that the path $P_2$ between $B$ and $C$ looses $x$ delay units. The number of delays on $P_2$ is $z - x + t$ instead of $z + t$ (see figure 13).

The last step of the retiming transformation is illustrated in figure 14. The end-to-end latency of $P_2$ is adapted to the initial number of delays. This can be done by introducing the
new ID component with $\delta_r(\text{ID}) = x$. As already stated above this solution is not unique. For instance, an ID component can also be inserted for every output of component $C$.

3.4 Improving the Jitter-Robustness

3.4.1 Reasons to Improve the Jitter-Robustness

Sometimes jitter affects a system’s inner state so that outputs are produced in a non-deterministic manner. Reactive embedded systems with buses are exposed to jitter and therefore, need to be analyzed carefully. Especially those components which deliver safety critical outputs should be jitter-robust. For a given system it makes sense to determine whether the jitter can affect the correctness of the system or not. In the section 3.2, a method for determining the jitter-robustness level is presented. The $\rho$ function is responsible for assuring the number of ticks in which jitter can occur without any negative effect on the system. Logical ticks can be translated to real-time numbers for further analysis. For example, a system uses a CAN-Bus for IPC. The number of participants that can simultaneously send messages on the bus as well as the $\rho$ function of the system are computable. In a worst case scenario, the developer can determine if a jitter-robustness improvement is needed by calculating the necessary time, for all the bus participants to submit their messages. The following subsections explain how the jitter-robustness can be improved.

3.4.2 Improving Basic Scenarios

This section reveals how basic models can improve their jitter-robustness. It is shown how a chain (Figure 15), a bifurcation (Figure 16) and a loop (Figure 17) of components can achieve the optimal jitter-robustness.

\[ \text{Figure 15: First basic scenario for improving the jitter-robustness.} \]
Example 3.4.1. Figure 15 sketches the first basic model. In figure 15(a) a chain of three strongly causal components $A, B$ and $C$ with order one is shown. The channel between components $B$ and $C$ is mapped on the bus. Hence, the jitter-robustness of the system is equal to the causality number of $C$ which is one. For the improvement, component $C$ has to receive the causality orders from $A$ and $B$. Figure 15(b) shows the optimal jitter-robust version of the system. It is the result of two retiming-transformations. First, component $B$ is retimed by one unit and then, component $C$ is retimed by two units.

![Diagram](image)

Figure 16: Second basic scenario for improving the jitter-robustness.

Example 3.4.2. The model in figure 16 has two outputs to the environment. The end-to-end latency on both outputs is two units. Figure 16(a) illustrates the model containing the strongly causal components $A, B$ and $C$ with order one. The messages between components $A, B$ and $A, C$ are sent on the bus. Therefore, the jitter-robustness equals one which is the minimal delayed path from $B$ and $C$ to an output. Figure 16(b) shows the optimal jitter-robust model. If we retime component $B$ with one causality unit then a new strongly causal ID-component has to be created between $A$ and $C$ (see section 3.3). After that, the retiming of component $C$ gets the causality of the ID-component. The resulting weakly causal ID-component is deleted and the system with the jitter-robustness of 2 in figure 16(b) emerges.

Example 3.4.3. The model in figure 17 contains a cycle of four components. Figure 17(a) displays the four strongly causal components $A, B, C$ and $D$ with order one. The two channels between $A, B$ and $C, D$ are mapped on the bus. This model has a jitter-robustness of one because the strongly causal components $B$ and $C$ receive bus messages and send them with one delay unit to the environment. Figure 17(b) represents the optimal achievable jitter-robust model from figure 17(a). The cycle contains four delay units which have to be distributed to components $B$ and $C$ (the number of delays on a cycle stays the same, see lemma 2.2.2). Retiming these components with one unit leads to the result seen in figure 17(b).
3.4.3 Optimizing the Jitter-Robustness

After analyzing the jitter-robustness of a model, the developer finds that in some cases the jitter affects the output correctness. In this scenario the developer needs to remodel the system. The assumptions made in the analysis section 3.2 stay the same. System $S$ is divided into two disjoint subsets of components $A$ and $B$ by a single bus. Note that the functionality and the end-to-end latency must not be different after the improvement. The only thing that will change is the components causality order. Knowledge about the behavior of the system is not required for improving the jitter-robustness. For the optimization of the jitter-robustness we employ the retiming-transformation presented in section 3.3. This is the transformation used to shift delays in the given network of components. Regarding the definitions 3 and 4, let us consider the following two sets:

\[
T_1 = \{ C : I_C \rightarrow \wp(O_C) | C \text{ is atomic subcomponent of } A \land \exists i \in I_C : i \in I_{BA} \} \tag{9}
\]

\[
T_2 = \{ C : I_C \rightarrow \wp(O_C) | C \text{ is atomic subcomponent of } B \land \exists i \in I_C : i \in I_{AB} \} \tag{10}
\]

\[
T = T_1 \cup T_2 \tag{11}
\]

In words, the set $T$ contains the components of the system which receive bus messages. If the system $S$ does not contain any feedback loops, we assume that $I_{BA} = \emptyset$.

**Jitter-Robustness optimization as a linear problem.** For maximizing the robustness of the system against jitter, we have to consider a max-min problem. We have to maximize the minimal guaranteed delay in the definition of the $\rho$ function (see definition 7). All components in $T$ can get the causality orders of all their successors by retiming. This is why each guaranteed delay can correspond to the causality order of a component in $T$. We regard the smallest guaranteed delay maximization as the causality order improvement of its specific component in $T$. Therefore, the $\rho_{opt}$ function computes the optimal jitter-
robustness for a system.

\[ \rho_{\text{opt}}(S) = \max_{C} \min \{ \delta(C) \mid \forall C \in T \} \quad (12) \]

For computing the value of \( \rho_{\text{opt}}(S) \) make some observations.

**Lemma 3.4.1.** In a data flow graph the sum of delays on every path from an input to an output of the system stays the same.

**Proof.** Let there be \( P \) a path from \( i \) (input) to \( o \) (output). According to 2.2.1 the number of delays in \( P \) after the retiming is \( \delta_r(P) = \delta(P) + r(o) - r(i) \). Input/Output vertices can not be retimed, that is why \( r(i) = r(o) = 0 \).

\[ \Rightarrow \delta_r(P) = \delta(P) \]

For optimizing the jitter-robustness we regard the data flow graph representation of the system \( S \). This graph \( G = (V, E, \delta) \) contains for each component a vertex in \( V \) and for each channel a edge in \( E \). \( \delta : E \rightarrow \mathbb{N} \) is the function associating each edge the number of delays it contains. The following optimization algorithm is divided in two parts. The first part contains the finding of a path coverage that contains all graph edges and the building of a system of linear expressions. In the second part this system of linear expressions is solved. This can be computed with the help of a linear problem solver. The causality of the system are adapted according to the variable occupancy of the linear expression system result. The algorithm 1, written in pseudocode, computes all paths in the data flow graph which cover all edges. Every path of this coverage contributes with an equation to the linear system.

The method \( \text{findPathFrom()} \), called in row 6, returns via a breadth-first search a path that starts at \( v \) and ends with an output vertex. Analogue, method \( \text{findPathTo()} \), called in row 7, returns a path that starts from a input vertex and ends in \( u \). This can be done by performing a breadth-first search from vertex \( u \) in the transposed graph \( G^T \). Algorithm 1 returns \( IO \), a set of paths from inputs to outputs that cover all edges of a graph. Each path defines a new equation to the linear system. For each \( x_1 \rightarrow x_2 \rightarrow \cdots \rightarrow x_n = P \in IO \), where \( x_1, \ldots, x_n \in V \) and \( x_1, x_n \) are input/output vertices, we determine the following equation:

\[ a_{x_1} + b_{x_1} * a_{x_1x_2} + \cdots + a_{x_{n-1}} + b_{x_{n-1}} * a_{x_{n-1}x_n} + a_{x_n} = \sum_{i=1}^{n} \delta(x_i) \quad (13) \]

In equation 13, \( a_{x_i} \) are variables that represent the number of delays that will go out from vertex \( x_i \). The result of the system of linear expressions assigns to each \( a_{x_i} \) a value which represents the causality order of the corresponding component. The variables \( a_{x_i x_j} \) express the number of delays on the edge from \( x_i \) to \( x_j \). They are important because the optimal jitter-robust graph can be heterogeneous. For translating this graph back into the logical
Algorithm 1 Returns a path coverage that contains all edges of the graph $G = (V, E)$

1: $IO \leftarrow \{\}$
2: $M \leftarrow \{\}$
3: for all $e = (u, v) \in E$ do
4:   if $e = (u, v) \notin M$ then
5:     $M = M \cup \{e\}$
6:     $P \leftarrow \text{findPathFrom}(v)$
7:     $P' \leftarrow \text{findPathTo}(u)$
8:     $IO = IO \cup (P' \times P)$
9:   for all $p \in P \cup P'$ do
10:      for all $e' \in p$ do
11:         $M = M \cup \{e'\}$  \{Add all edges found on path $p$ to $M$.\}
12:      end for
13:   end for
14: end if
15: end for
16: return $IO$

architecture, ID-components are introduced in order to homogenize the graph. A similar technique is used in section 3.3. In addition, equation 13 contains binary constants $b_{x_i}$. They have the following specification:

$$b_{x_i} = \begin{cases} 
1 & \text{if the number of outgoing edges from } x_i \text{ is } > 1; \\
0 & \text{otherwise.} 
\end{cases}$$

If $b_{x_i} = 0$, we do not need to take into account the variables $a_{x_i x_j}$. The graph is already homogeneous in vertex $x_i$.

In the computed system of linear equations we have to find the best solution to maximize the jitter-robustness. The optimal solution is calculated by maximizing the value of the variables $a_{x_i}$ which correspond to graph vertices $x_i$ and represent a component $X_i \in T$ (see definition of the optimal jitter-robustness function, 12).

Example 3.4.4. Given a system containing eight logical components which communicate with each other (see figure 18), the task is to optimize the jitter-robustness without changing the end-to-end latency. Furthermore, we assume that the channels that are deployed on the bus are the ones between $D$ and $E$ and between $H$ and $F$. Regarding the definition of $T$ (see definition 11), we determine the components which are important for the jitter-robustness. These components are: $T = \{E, F\}$. Therefore, we have to retime the system in such a manner that $\min\{\delta_r(E), \delta_r(F)\}$ is maximal. The first step for optimizing the jitter-robustness is to compute those paths which cover all edges of the graph. Executing the algorithm presented in pseudocode 1, two paths, $P_1$ and $P_2$, are
Figure 18: The data flow graph containing the eight strongly causal components of the system. This figure is a modified version of the graph visualized with CADMOS [Cad].

\[ P_1 = \{(A \rightarrow B), (B \rightarrow C), (C \rightarrow D), (D \rightarrow E), (E \rightarrow F)\} \]

\[ P_2 = \{(A \rightarrow B), (B \rightarrow G), (G \rightarrow H), (H \rightarrow F)\} \]

Consequently, the linear system contains the two equations:

\[ a_A + a_B + a_{BC} + a_C + a_D + a_E + a_F = 6 \] (14)

\[ a_A + a_B + a_{BG} + a_G + a_H + a_F = 5 \] (15)

This system of equations has to be solved so that the minimum between \( a_E = \delta(E) \) and \( a_F = \delta(F) \) is maximal. Linear programming solvers are able to compute this max-min problem. For example, the lp.solve program offers a language that can describe this problem (see [Lp]). The following script run in lp.solve produces the desired results:

```plaintext
/* Objective function */
//max part, ojr represents the optimal jitter–robustness
max: ojr;

/* Variable bounds */
a + b + bc + c + d + e + f = 6;
a + b + bg + h + f = 5;

//min part, ojr = min\{e, f\}
ojr < e;
ojr < f;
```
The linear programming solver finds that the optimal solution for $\rho_{opt}$ is 3. The following variable values lead to the optimum:

$$a = 0, b = 0, bc = 0, c = 0, d = 0, e = 3, f = 3, bg = 2, g = 0, h = 0$$

The solver provides a solution which is the result of 6 retiming-transformations done in a specific order.

<table>
<thead>
<tr>
<th>Step</th>
<th>Retiming</th>
<th>Variable Occupancy</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.</td>
<td>$r(B) \leftarrow 1$</td>
<td>$a = 0, b = 2, bc = 0, c = 1, d = 1, e = 1, f = 1, bg = 0, g = 1, h = 1$</td>
</tr>
<tr>
<td>2.</td>
<td>$r(C) \leftarrow 2$</td>
<td>$a = 0, b = 0, bc = 0, c = 3, d = 1, e = 1, f = 1, bg = 2, g = 1, h = 1$</td>
</tr>
<tr>
<td>3.</td>
<td>$r(D) \leftarrow 3$</td>
<td>$a = 0, b = 0, bc = 0, c = 0, d = 4, e = 1, f = 1, bg = 2, g = 1, h = 1$</td>
</tr>
<tr>
<td>4.</td>
<td>$r(E) \leftarrow 4$</td>
<td>$a = 0, b = 0, bc = 0, c = 0, d = 0, e = 5, f = 1, bg = 2, g = 1, h = 1$</td>
</tr>
<tr>
<td>5.</td>
<td>$r(H) \leftarrow 1$</td>
<td>$a = 0, b = 0, bc = 0, c = 0, d = 0, e = 5, f = 1, bg = 2, g = 0, h = 2$</td>
</tr>
<tr>
<td>6.</td>
<td>$r(F) \leftarrow 2$</td>
<td>$a = 0, b = 0, bc = 0, c = 0, d = 0, e = 3, f = 3, bg = 2, g = 0, h = 0$</td>
</tr>
</tbody>
</table>

Table 3: The sequence of retiming-transformations leading to the optimal jitter-robustness.

**Summary** This optimization method delivers the causality values for each component after the retiming-transformation. These are the variable occupancy of the linear expression system result. There can be variable delay values which do not affect the jitter-robustness (in example 3.4.4, $bg = 2$). The developer has the freedom to distribute them in the system by desire via retiming. Furthermore, retiming a system according to the results of a linear problem has been used before. This technique is presented in the document [LS91] for minimizing the clock period by forming a system of inequalities (see section 2.2.5).
4 Implementation of the Analysis and Transformation Methods in AutoFocus 3

While the previous sections provide the theoretical background, this section is concerned with the extension of the case tool AutoFocus 3. The presented analysis and transformation techniques for the jitter-robustness improvement are implemented in this tool. This section describes the extended AutoFocus 3 functionality implemented by the author, which influences the jitter-robustness on structural level.

4.1 Showing the Guaranteed Delays in AutoFocus 3

4.1.1 Motivation

Guaranteed delays are important for the analysis of the jitter-robustness of a system. Section 3.2 presented the \( \rho \) function which depends on the guaranteed delays at some specific outputs. In a complex system with several levels in the hierarchy of the logical architecture, it is hard to determine the guaranteed delay for each output manually. The guaranteed delays displayed at all outputs simplifies the computation of the \( \rho \) function for the given system.

![Figure 19: Each output of a component with its guaranteed delays. The value 'X' is shown for those outputs which do not communicate with any environment input ports.]

4.1.2 Implementation

The recursive algorithm 2 computes the guaranteed delays of a system. It uses a modified depth-first search on the graph with the atomic components and environment ports as vertices and channels as edges of the system. Atomic components of a logical architecture
Algorithm 2 Computes all guaranteed delays given the graph of atomic components.

\[
\text{ANALYZE GUARANTEED DELAYS (graph)}
\]

1: \( \text{visited} \leftarrow \{\} \)
2: \( \text{guard[]} \leftarrow \{\infty, \infty, \ldots, \infty\} \)
3: \( \text{inputs} \leftarrow \text{getInputVertices()} \)
4: \textbf{for all} \( v \in \text{inputs} \) \textbf{do}
5: \( \text{DODFS}(\text{graph, } v, 0, \text{visited, guard}) \)
6: \textbf{end for}
7: \textbf{return guard}

\[
\text{DODFS (graph, node, delays, visited, guard[])}
\]

1: \textbf{if} \( \text{node} \in \text{visited} \land \text{guard}[\text{node}] \leq \text{delays} \) \textbf{then}
2: \( \text{return} \)
3: \textbf{end if}
4: \( \text{guard}[\text{node}] \leftarrow \text{delays} \)
5: \( \text{visited} \leftarrow \text{visited} \cup \{\text{node}\} \)
6: \textbf{for all} \( v \in \text{graph.getOutNodes}(\text{node}) \) \textbf{do}
7: \( \text{DODFS}(\text{graph, } v, \text{delays} + v.\text{getDelays}(), \text{visited, guard}) \)
8: \textbf{end for}

have a behavior specification and do not contain any other subcomponents. The environment ports are those input/output ports which communicate with the environment. Algorithm 2 uses a set \( \text{visited} \), containing vertices which have already been expanded in the depth-first search, and an array \( \text{guard[]} \), storing for each vertex the number of delays on the least delayed path from an input port to it. Therefore, \( \text{guard}[o] \) contains the number of the guaranteed delays, for each output port \( o \).

For the visualization of the guaranteed delays a 'org.eclipse.draw2d.Label' is added to the Figure of the 'ComponentEditPart.java' class located in the 'edu.tum.cs.af3.application' plugin.

**Clarifications.** In each iteration of the depth-first search a vertex is not expanded again if it is visited and the delay on the path from the root of the dfs-tree to it is greater than the value stored in \( \text{guard[]} \) (see line 1). This means, each time a vertex is expanded it improves the guaranteed delays of its successors. Furthermore, the method \( \text{getDelays()} \) at line 7 returns the causality order of the analyzed atomic component. Note that all environment ports have a causality order of 0. Sensors and actuators send the information undelayed to the environment.

**4.1.3 Results**

Near every output port of each component the results of algorithm 2 is seen. The numbers representing the guaranteed delays are displayed in a rectangle box. A 'X' is displayed
instead of a number if there is no path from an input to that output. For instance, this can be the case when submitting debug-information to the environment. Hence, the jitter-robustness for a system like the one in figure 10 is determined by the guaranteed delays. The $\rho$ function of the system is the minimum of all guaranteed delays of the analyzed component. Figure 19 shows how the guaranteed delays are displayed in AutoFOCUS 3.

4.2 Extending the Causality

4.2.1 Motivation

In the current implementation of AutoFOCUS 3 a component is either strongly or weakly causal. A strongly causal component sends on all outputs the result processed in the previous logical tick. The results are delayed by exactly one time unit. This section introduces the causality with arbitrary order for AutoFOCUS 3. If a component is strongly causal, its outputs are delayed by a determined non-negative number of delays. This specification is necessary for the retiming-transformation in a logical architecture. After a component is retimed, its causality can have a higher value greater than one. Moreover, causality with arbitrary order offers the possibility to optimize the jitter-robustness in a network of components. As presented in section 3.4, the result of the linear expression system defines the $\rho_{opt}$ function. It can be adapted to the logical architecture if it supports causality with higher order.

4.2.2 Implementation

The implementation of causality with arbitrary order has to consider the initial value problem. A strongly causal component with order $x > 0$ has $x$ initial values. If a component gets a higher order after a retiming-transformation, the initial values and data state variables have to be chosen carefully. In this case, random initial values can lead to significant deviation from the original functionality. Figure 20 illustrates a strongly causal component of order 4, whose initial values have to be 1, 2, 6, 24 for the behavior to compute the factorial function. Let us assume that $x$ is the stream that is sent to the output. With

![Figure 20: The data flow graph realized with [Cad] of a strongly causal component of order 4.](image)
the help of the data state variables $d_1, d_2, d_3, d_4$ and the initial values $x(1), x(2), x(3), x(4)$ the following program will be executed:

\[
\begin{align*}
&d_1 \leftarrow 2, d_2 \leftarrow 3, d_3 \leftarrow 4, d_4 \leftarrow 5 \\
&x(1) \leftarrow 1, x(2) \leftarrow 2, x(3) \leftarrow 6, x(4) \leftarrow 24 \\
&\text{for } t = 5 \rightarrow \infty \text{ do} \\
&\quad x(t) \leftarrow x(t - 4) \ast d_1 \ast d_2 \ast d_3 \ast d_4 \\
&\quad d_1 \leftarrow d_1 + 1, d_2 \leftarrow d_2 + 1, d_3 \leftarrow d_3 + 1, d_4 \leftarrow d_4 + 1 \\
&\text{end for}
\end{align*}
\]

Therefore, initial values and data state variables have to be computed statically before runtime. Details about how to calculate the initial values are found here [Par89].

4.2.3 Results

Each component in a logical architecture of AutoFOCUS 3 can have a causality of a non-negative order. The component is weakly causal if the order is 0 and strongly causal otherwise. Note that the causality with arbitrary order should be used with caution. It may affect the systems reactivity. The sum of all causality orders on a path from an input to an output defines the response time of the system. Furthermore, a new causality specification is used to describe the new feature. The metamodel containing this new specification contains a new class ‘NewCausalitySpecificationBase’ for the causality with arbitrary order. Figure 21 shows the super-class IComponentSpecification which

![Diagram of the metamodel containing the new specification class which accepts causality with arbitrary order.]

represents all specification types of a component. The old specification contains a boolean, `strongCausal`, which indicates if a component is strongly or weakly causal. In the new specification this variable is a integer revealing the causality order. A component which
has a new causality specification with the attribute \( \text{strongCausal} = 0 \), is a weakly causal component. All other components, with \( \text{strongCausal} > 0 \), are strongly causal.

**Conclusion.** The result of the causality extension is a new metamodel which contains the old and new specification. The created system is backward-compatible allowing the developer to use both specifications.

### 4.3 Implementing the Retiming-Transformation

#### 4.3.1 Motivation

The retiming-transformation is an important technique for changing the causality of a system. Originally invented for retiming synchronous circuitry [LS91], the retiming-transformation can also be adapted for the logical architecture of AutoFOCUS 3. (see section 3.3). Retiming changes the causality of the system without affecting the behavior or the end-to-end latency. This is the case for the jitter-robustness optimizing method presented in section 3.4.3 which produces a result as a sequence of consecutive applied retiming-transformations. The aim of this optimization is to maximize the minimum causality order belonging to some specific components. It is possible that the result contains components which are not relevant for the jitter-robustness and thus have a causality order greater than 0. The remaining causalities can be distributed among the system in AutoFOCUS 3 via retiming-transformation. (e.g. the remaining causalities can be used to increase the parallelism of the system)

#### 4.3.2 Implementation

For the implementation, the steps presented in section 3.3 for preserving the end-to-end latency are considered. The algorithm used to implement the retiming-transformation has to execute the following:

1. Add to the retimed component \( C \) the desired amount \( x \) of causality. A weakly causal component becomes strongly causal.

2. Find a new predecessor component \( P \) which has not been considered yet. If no such component exists, terminate algorithm.

3. Subtract \( x \) from the causality order of \( P \). If \( P \)'s new causality is 0 make it weakly causal.

4. Find a new successor component \( S \) of \( P \) which is not component \( C \). If no such component exists continue with step 2.

5. Delete the channel between \( P \) and \( S \).

6. Create a new strongly causal component \( ID_P \) of order \( x \) which has one input and one output. This component computes the identity function.
7. Add a channel between \( P \) and \( ID_P \) and a channel between \( ID_P \) and \( S \).


In AutoFOCUS 3 each component does not need to have unique names. For each predecessor component \( P \) there may be more than one \( ID_P \) components. Therefore, the logical architecture of the system is easier to understand if the name \( ID_P \) is extended with the name of the incoming channel.

### 4.3.3 Results

In AutoFOCUS 3 any component of the logical architecture can be retimed. Right-clicking the desired component, a dialog box pops up and shows the interval of numbers in which a valid retiming-transformation can be executed. The interval \([0..0]\) states that the component can not be retimed. After a component is retimed, the causality of the system is updated. The end-to-end latency stays the same. Consequently, the retiming-transformation can influence (positive or negative) some properties like the jitter-robustness or logical tick size of a system. A longer chain of weakly causal components as the result of a retiming-transformation increases the logical tick size determined by the critical path. Also, the retiming-transformation can be used with the help of improvement methods like the one presented in section 3.4.3. This leads to an optimal jitter-robustness in the system.

![Figure 22: The logical architecture modification of the ACCSystem whose component PCS is retimed with value 2.](image)
4.4 Adapting the Code Generator

4.4.1 Motivation

This section provides details about how strongly causal components of a higher order delay their outputs in AutoFocus 3. In the current implementation of AutoFocus 3, the code generator produces one delay unit for each strongly causal component. This means, the code generator does not support causality with arbitrary order. Therefore, the new metamodel introduced in figure 21 is adapted to the code generator of AutoFocus 3.

4.4.2 Implementation

For the code generation of a component with arbitrary order, FIFO buffers are used. Each component possesses for each input port a ring-buffer with the size of its causality. These buffers receive the information that is being sent from the outside and always deliver the oldest message stored. In the first logical ticks the initial values are returned. These are the values that are stored in the ring-buffers before runtime.

**Example 4.4.1.** Let us consider a sender-receiver system implemented in AutoFocus 3. The data flow graph of this system is seen in figure 23. The sender is a weakly causal component and the receiver is a strongly causal component of order 3. We assume that the sender has one environment input and one output processing the function \( f_s() \). Moreover, the receiver component has one input and one environment output calculating the function \( f_r() \). The table 4.4.1 shows how the information is stored in the ring-buffer of the receiver component and how the causality of order 3 is preserved.

![Data flow graph](image)

Figure 23: The data flow graph in CADMOS of a simple sender-receiver system (see [Cad]).

<table>
<thead>
<tr>
<th>Tick(t)</th>
<th>( f_s(x.t) )</th>
<th>( f_r(f_s(x.t)) )</th>
<th>Receiver Ring-Buffer</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>( x_4 )</td>
<td>( x_1 )</td>
<td>([x_1, x_2, x_3])</td>
</tr>
<tr>
<td>2</td>
<td>( x_5 )</td>
<td>( x_2 )</td>
<td>([x_2, x_3, x_4])</td>
</tr>
<tr>
<td>3</td>
<td>( x_6 )</td>
<td>( x_3 )</td>
<td>([x_3, x_4, x_5])</td>
</tr>
<tr>
<td>4</td>
<td>( x_7 )</td>
<td>( x_4 )</td>
<td>([x_4, x_5, x_6])</td>
</tr>
<tr>
<td>5</td>
<td>( x_8 )</td>
<td>( x_5 )</td>
<td>([x_6, x_7, x_8])</td>
</tr>
</tbody>
</table>

Table 4: The receiver component, with causality of order 3, delays its outputs with the help of a ring-buffer.
4.4.3 Results

A thorough implementation of all necessary features for complete code generation was not within the context of this thesis. Nevertheless, the prototypical implementation made was employed for the evaluation (see section 5) and has shown to be usable.

4.5 Jitter-Robustness Optimization Implemented in AutoFocus 3

4.5.1 Motivation

This section contains information about how the results of the jitter-robustness optimization method, presented in 3.4.3, are shown in AutoFocus 3. It is important for the developer to know which is the highest achievable jitter-robustness. When changing the systems causality or deployment, the effects on the optimal jitter-robustness can be seen. In this case if the optimum is smaller than the desired robustness, the developer has the option to revert the changes. Furthermore, it is very hard to determine manually the causality of each component in a complex system, in order to achieve the optimal jitter-robustness and preserve the end-to-end latency. Industrial relevant systems often have a complex logical architecture and are very hard to analyze. The method presented in section 3.4.3 assures to compute the optimal achievable jitter-robustness by solving a linear problem. Hence, the developer can adapt the causality of all components on structural level according to the variable occupancy of the solved linear system of equations for the jitter-robustness maximization.

4.5.2 Implementation

Described in section 3.4.3, the optimization of the jitter-robustness has two parts. The first part contains the finding of a path coverage that includes all edges of the graph and the construction of a system of linear expressions. Finding a path coverage in AutoFocus 3, is done by executing the algorithm explained in pseudocode 1. For each path a new equation is added to the system (see equation 13). In the second part, the solving of the linear system is effectuated. Lp_solve (see [Lp]) is executed with the automatically created script of linear expressions. After the solver finishes computing, it offers the possibility to visualize the variable occupancy of the linear system.

4.5.3 Results

The results are seen in AutoFocus 3 for each deployment block. Each deployment category has assigned the logical architecture of a root component and a topology (technical architecture) on which the logical system is mapped. By right-clicking the deployment block and selecting the ‘Optimize Jitter-Robustness ...’ action the lp_solve program starts optimizing the system. A pop-up box, showing each components causality for the optimal jitter-robustness appears as soon as the program finds the optimal solution.
5 Evaluation

This section contains some practical results of the presented concepts. The aim is to show that jitter can have negative effects on a system which is not properly retimed. Using the optimization method from section 3.4.3, the jitter-robustness can improve. The system can be immune to jitter that is smaller than $\rho_{opt}(S)$ time units. Section 5.1 contains a ‘Hello World’ example which demonstrates the optimization effects on the system and section 5.2 reveals how the jitter-robustness can be improved for the Adaptive Cruise Control model in AutoFOCUS 3.

5.1 Proof of Concept on Embedded Hardware

5.1.1 Motivation

The jitter-robustness analysis and optimization concepts are evaluated on embedded hardware. A simple sender-receiver system is supposed to show the importance of jitter-robustness. This example, reveals a new methodological approach. From a top-down perspective the designer creates a logical model, analyzes its jitter-robustness and improves it if necessary. All this can be done statically before runtime by using the retiming-transformation appropriately.

5.1.2 Setup

The logical architecture contains two components communicating through a channel. The first component receiving information from the environment is the sender and the second component sending information to the environment is the receiver. In the first iteration over the system, the sender component has a causality of order 50. This means it sends the information received from the environment with a delay of 50 logical ticks. The setup of the logical architecture is visualized with the help of CADMOS (see [Cad]) in figure 24. Furthermore, the technical architecture contains two Freescale MPC5554EV ECU’s connected with each other through a CAN-Bus. The sender component is deployed on one MPC which receives information from the environment through a potentiometer. The receiver component is deployed on the other MPC which sends information to the environment through a servo motor controlled with pulse-width modulation. Both MPC’s run at a frequency of 50 Hz which corresponds to a logical tick of 20 ms. The aim of this setup is to control remotely the servo motor from the receiving MPC with the

![Figure 24: The data flow graph of the analyzed system.](image-url)
potentiometer from the sending MPC. In order to make jitter even more dangerous, delta values are sent on the bus. This means, the sender returns the difference between the last value and the current value received from the potentiometer. If a message is lost, the servo motor points at a different value than the one read by the sender board.

Creating jitter. Two MPC5554EVB ECU’s connected to the same CAN-Bus are used to create jitter. If both send at a 1 kHz frequency dummy messages with a higher priority on the bus, the sender reduces his sending rate. Therefore, the information comes to slow to the receiver and jitter is produced. Figure 25 represents the technical architecture of the system. MPC’s 1 and 2 are the sender/receiver components communicating through CAN-Bus 5. 6 shows the potentiometer reading environment inputs and 7 is the servo motor actuator. MPC’s 3 and 4 are used to create bus load in order to produce conflicts that delay the sending routine. If these two MPC’s 3 and 4 run simultaneously, the system looses information and the servo motors position will no longer correspond to the potentiometers value. In this example the worst case execution time (WCET) for a component is $\approx 6\mu s$ which is much lower than the logical tick, $20ms$. Therefore, if the jitter is lower than $\approx 20ms - 6\mu s$ the receiver component can still produce an output in time. That is why one MPC is insufficient for the disturbance of the system behavior.

Figure 25: The embedded hardware which represents the technical architecture of the system.
5.1.3 Analysis and Transformation

The first step is to analyze this system $S$ shown in figure 24. Regarding section 3.2, the system has $\rho(S) = 0$. In this case the developer should analyze the bus load carefully. For systems which send delta values, jitter-robustness is essential. It can occur that a message that comes too late has as consequence for the system to run out of sync and produce incorrect outputs. This example deals with a system which has one input and one output with an end-to-end latency of 50 time units. The jitter-robustness is 0 because the causality of the receiver component is 0. However, this model can be improved. For the optimal jitter-robustness the receiver component has to be retimed with the maximal possible value 50. The resulting data flow graph of the transformed system is seen in figure 26. The

![Figure 26: The data flow graph of the transformed, jitter-robust system.](image)

optimal jitter-robustness does not exceed 50 logical ticks which is equal to the latency from the input to the output. In the section 5.1.2, it is stated that the MPC’s, on which the components are deployed, run at a 50 Hz frequency. This means, they produce an output every 20 ms. Therefore, having a jitter-robustness of 50 ticks means that the information can come 1 second later and the produced output is correct.

5.1.4 Results

For this example, the optimal jitter-robust system is created. Jitter smaller than one second does not affect the output. The result is seen when the two MPC’s which create bus load are turned on for less than one second. The system does not change its behavior. Another way to test the jitter-robustness is to unplug the CAN-Bus cable. Removing the cable for less than one second from a MPC causes the receiver buffer to run almost empty but not lose any data. This scenario is described with the help of table 5.1.4. Notice how after the CAN-Bus cable is plugged in, the receiver ring-buffer gets all the elements that have not been transmitted. This is done by the CAN-Driver which stores the messages in an internal buffer, if the sending is unsuccessful. After the cable is plugged in, all the messages stored in the CAN-Driver buffer are sent to the receiver. In this way the receiver buffer becomes full again and the system is ready for new jitter to occur.

5.1.5 Discussion

For this system, the optimal jitter-robustness must not necessarily be achieved. The original system whose jitter-robustness is 0 needs two MPC’s, which submit dummy
frames on the CAN-Bus with a 1 kHz frequency, to affect the output correctness. This is unlikely to occur in a real system. Although, the developer can improve the jitter-robustness by retiming the receiver component with a smaller number in order to exclude the worst case scenario which is determined in the analysis phase.

### 5.2 The Adaptive Cruise Control Case Study

#### 5.2.1 Motivation

In this section a real, industrial relevant system is evaluated. The Adaptive Cruise Control (ACC) model implemented in AutoFOCUS 3 is the result of the DENTUM research projects of the Chair for Software and System Engineering from the Technical University of Munich in cooperation with industrial partners. It is important to test the results of the jitter-robustness optimization on a complex model.

#### 5.2.2 The ACC Functionality and Architecture

The ACC is a system used in the automotive industry for the travel speed control. It maintains a constant speed unless the distance to the next vehicle is too low. Through active breaking the ACC system preserves a minimal distance dependent on the vehicle speed. The system also contains a pre-crash safety (PCS) component. If it detects that a collision is about to happen, it stops the acceleration, tightens the seat belt and breaks actively.

### The Logical Architecture

For the jitter-robustness analysis and transformation methods, the focus is set on the logical architecture. The DENTUM ACC model is implemented

<table>
<thead>
<tr>
<th>Tick(t)</th>
<th>Output</th>
<th>Receiver Ring-Buffer</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>$x_1$</td>
<td>$[x_1, x_2, \ldots, x_{50}]$</td>
</tr>
<tr>
<td>2</td>
<td>$x_2$</td>
<td>$[x_2, x_3, \ldots, x_{51}]$</td>
</tr>
<tr>
<td>3</td>
<td>$x_3$</td>
<td>$[x_3, x_4, \ldots, x_{52}]$</td>
</tr>
<tr>
<td></td>
<td></td>
<td><strong>CAN-Bus Unplugged</strong></td>
</tr>
<tr>
<td>4</td>
<td>$x_4$</td>
<td>$[x_4, x_5, \ldots, x_{52}]$</td>
</tr>
<tr>
<td>5</td>
<td>$x_5$</td>
<td>$[x_5, x_4, \ldots, x_{52}]$</td>
</tr>
<tr>
<td></td>
<td></td>
<td><strong>CAN-Bus Plugged In</strong></td>
</tr>
<tr>
<td>53</td>
<td>$x_{53}$</td>
<td>$[x_{53}, x_{54}, \ldots, x_{102}]$</td>
</tr>
<tr>
<td>54</td>
<td>$x_{54}$</td>
<td>$[x_{54}, x_{55}, \ldots, x_{103}]$</td>
</tr>
</tbody>
</table>

Table 5: The receiver buffer and its elements during the period in which the cable is unplugged.
in AutoFOCUS 3 and contains seven strongly causal atomic components of order one: \textit{PCS, DesiredSpeed, SpeedControl, CoreArbiter, DesiredDistance, DistanceControl} \ and \textit{OnOffArbiter}. We analyze a modified version of the DENTUM ACC model where the new components \textit{ConstInputACC, ConstantSensorEmulation, ConstInputPCS} \ and \textit{SensorProcessing} \ are used to generate inputs. The data flow graph of the model is seen in figure 27. In order to make things clearer, the environment ports have been left out. The logical architecture of the root component and the hierarchical structure are shown in figure 27: The data flow graph of the modified DENTUM ACC model visualized with \textsc{Cadmos}.

\textbf{figure 28. ACCSystem} is the root component of the DENTUM ACC system. It includes two components, \textit{ACC} and \textit{PCS}, which communicate with each other through the channel \textit{pcsSuspend}. The \textit{ACC} system receives from environment inputs regarding the state of the system (\textit{accOnOff}), the gas and brake pedal positions (\textit{gasPedal, brakePedal}) and the request to change distance or to increase/decrease the vehicle speed (\textit{changeDistance, incDecDesiredSpeed}). It calculates the acceleration and sends it through the output port \textit{outputAcc} to the environment. Multicast input ports send information, about the existence and distance to the vehicle ahead and the speed, to both components, \textit{ACC} and \textit{PCS}. Furthermore, \textit{PCC} receives information about the belt and brake status (\textit{beltReady,}


It also has the ability to stop the acceleration of the ACC component by sending the value ‘true’ on the pcsSuspend channel. This can be the case if the distance in meter to the next vehicle is smaller than the half of the vehicle speed. The pre-crash safety system also has the duty to brake and show a warning if a collision is detected.

![Diagram](image)

Figure 28: The hierarchical view of the ACCSystem in figure (a) and the logical architecture of the root component in figure (b).

### 5.2.3 Possible Results of the Jitter-Robustness Optimization

There are many possibilities to deploy the logical architecture on two ECU’s of type MPC5554 connected through a CAN-Bus. This section contains the analysis regarding the jitter-robustness optimization of two different deployment mappings.

**Deployment 1.** All the components which belong to the ACC system are mapped to the first MPC and the PCS component to the second MPC. Hence, the CAN-Bus has to transmit the safety critical information of the channel pcsSuspend and the messages coming from the SensorProcessing component on channels vSpeed, targetExists and targetDistance. This mapping is intuitive and has relative little bus load (see figure 29). The value transmitted on the channel pcsSuspend is mostly ‘false’. The safety critical value ‘true’ is sent only if a collision is detected. For analyzing the jitter-robustness of the system with this deployment, the following observations are taken into account. All components are strongly causal of order one. This means, the jitter-robustness is greater or equal to one time unit. Because the OnOffArbiter component receives information from the bus and sends information to the environment port output with one delay unit, the...
jitter-robustness is exactly one logical tick. Using the optimization method presented in section 3.4 and implemented in AutoFOCUS 3 (see section 4.5), the optimal jitter-robustness is also one time unit. In this case the optimal jitter-robustness is already achieved. No changes need to be done to the model. This deployment mapping can not produce for the system a jitter-robustness greater than one.

<table>
<thead>
<tr>
<th>Name</th>
<th>WCET</th>
</tr>
</thead>
<tbody>
<tr>
<td>ACCSystem, PCS</td>
<td>46.0μs</td>
</tr>
<tr>
<td>ACCSystem, SensorProcessing</td>
<td>21.0μs</td>
</tr>
<tr>
<td>ACCSystem, ConstInputPCS</td>
<td>6.0μs</td>
</tr>
<tr>
<td>ACCSystem, ConstantSensorEmulation</td>
<td>6.0μs</td>
</tr>
<tr>
<td>MPC1</td>
<td>81.0μs</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Name</th>
<th>WCTT</th>
</tr>
</thead>
<tbody>
<tr>
<td>CAN-Bus</td>
<td>8.0 ms</td>
</tr>
<tr>
<td>pcsSuspend</td>
<td>2.0 ms</td>
</tr>
<tr>
<td>vSpeed</td>
<td>2.0 ms</td>
</tr>
<tr>
<td>targetDistance</td>
<td>2.0 ms</td>
</tr>
<tr>
<td>targetExists</td>
<td>2.0 ms</td>
</tr>
<tr>
<td>ACCSystem, ACC</td>
<td>165.0μs</td>
</tr>
<tr>
<td>ACCSystem, ConstInputACC</td>
<td>12.0μs</td>
</tr>
<tr>
<td>MPC2</td>
<td>177.0μs</td>
</tr>
</tbody>
</table>

Figure 29: The first deployment mapping realized in AutoFOCUS 3 of the DENTUM ACC system.

**Deployment 2.** The second deployment mapping gets the OnOffArbiter and the Core-Arbiter on the first MPC and the rest of the ACC system and the PCS component on the second MPC. This mapping is seen in figure 30. The CAN-Bus transmits the inputs generated in the SensorProcessing and ConstInputACC components. Moreover, the bus also sends the messages on the channels constantSpeedAccel, followUpAccel, speedCap and suspend. All this messages are safety critical and need to be jitter-robust. For the jitter-robustness analysis we make the same two observations as with deployment 1 and the result is one time unit. A advantage of this deployment mapping is the possibility to improve the jitter-robustness. With the help of the optimization method implemented in AutoFOCUS 3 (see section 4.5) the developer finds out how to make the system robust against two logical ticks of jitter. The result of the jitter-robustness optimization assigns both components OnOffArbiter and CoreArbiter a causality of order two. Therefore, the optimal robustness against jitter is twice as big as the one in the original system.

### 5.2.4 Discussion

Without the proper algorithm for retiming the DENTUM ACC system, it is very hard to improve the jitter-robustness manually. With the help of the methods from section 3, we
Figure 30: The second deployment mapping realized in AutoFOCUS 3 of the DENTUM ACC system.

analyzed two different deployment mappings and realized that both lead to a different optimal jitter-robustness. The first mapping does not bring any improvements to it, whereas the second one doubles its value. Hence, the deployment mapping influences directly the value of the jitter-robustness and its optimum. The method used in the second mapping was to deploy the components which deliver safety critical outputs on an own MPC. This heuristic causes the robustness optimization algorithm to improve where it is most needed.
6 Discussion

In this section we take a closer look to the presented aspects of this thesis. The previous sections bring a definition to the jitter-robustness (section 3.1), an analysis technique to calculate it (section 3.2) and an optimization method for improving its value (section 3.4.3). An example of a system on embedded hardware shows the validity of these analysis and improvement procedures (section 5.1). Moreover, the case tool AutoFOCUS 3 has been the main tool for testing the jitter-robustness with real models. Some of the analysis and transformation methods have been implemented in AutoFOCUS 3 and used on the DENTUM Adaptive Cruise Control model (section 5.2). By choosing the right deployment the jitter-robustness of the DENTUM ACC model can be improved to be twice as big (see section 5.2.3). However, some questions still need to be clarified.

What are the trade-offs of optimizing the jitter-robustness? When designing a system the developer has to know regarding the inter-processor communication how much bus load there is in a worst case scenario. A strong analysis during the designing phase leads to a conclusion, whether the jitter-robustness of the system is sufficient or not. It is not necessary to change the system to obtain the optimal jitter-robustness if it is not needed. Nevertheless, the optimization method suggests if the constructed system can be robust against jitter. If the optimal jitter-robustness is not sufficient to cover the worst case scenario of jitter, the developer should consider adding more causality to the system or changing the deployment mapping. Furthermore, achieving the optimal jitter-robustness can damage other properties of the system like performance or parallelism. It is the developers task to find a way to preserve a certain amount of all properties.

Is there a connection between the jitter-robustness and the reactivity of a system? The two terms jitter-robustness and reactivity are two properties of a system which are opposite. This means, a system with a higher jitter-robustness value has a lower reactivity level. The other way must not necessarily hold. A lower reactivity level can have both a high and a low jitter-robustness. Regarding section 3.2, the jitter-robustness is calculated by searching the minimal delayed path from a component which receives bus information to an output. If the robustness against jitter is high, the end-to-end latency on that path is also high and this results to a low reactivity. The other way, if the reactivity is low, all the causality delays must not necessarily be on that minimal path to an output.

What happens after the optimization with the causalities that are irrelevant for the jitter-robustness? After the optimization some components can receive causality of a higher order even if they do not improve the jitter-robustness. In this case the developer chooses the distribution of the remaining causality after other criteria. For example, the developer can decide to improve the parallelism or improve the performance by adjusting the minimal clock period time.
Is the jitter-robustness optimization method customizable?  It is described in section 3.4.3 how to achieve the optimal jitter-robustness by changing the logical architecture in a system. However, components which deliver safety critical outputs must be robust against jitter. Hence, the optimizing method can be adjusted to improve the jitter-robustness for specific components. This is done by selecting those components in the inequality part of the linear problem (see section 3.4.3) which deliver safety critical outputs. The entire system can have a lower jitter-robustness value than the optimum but the reliability on the safety critical outputs is higher.

What are the results of this thesis?  The results of this thesis are the jitter-robustness analysis and optimization methods, the retiming-transformation adapted to the logical architecture and the evaluation of the industrial DENTUM ACC model. Jitter-robustness is expressed in logical ticks using the analysis method presented in section 3.2. The formal method for achieving the optimal jitter-robustness is described in section 3.4.3 by creating a linear problem. An advantage of this procedure is its scalability. The optimal result is produced for the industrial relevant Adaptive Cruise Control model implemented in AutoFOCUS 3 in less than 0.1 seconds on a Intel(R) Core(TM)2 Duo CPU with 2.4GHz.
7 Conclusion and Future Work

Jitter-robustness is an important property of embedded multiprocessor systems. It improves the quality insurance. After the analysis method presented in section 3.2, the developer decides if the robustness against jitter needs to be improved. The optimization method in section 3.4.3 indicates the maximal jitter-robustness which can be achieved regarding the logical and technical perspective together with their deployment mapping.

**Improving the jitter-robustness on behavior level as future work.** The focus is set inside the logical tick. Making the duration of a tick longer adds jitter-robustness to the system. However, this affects the performance of the system. Jitter-robustness can also be improved on behavior level by choosing an adequate scheduling algorithm. Finding a heuristic for the task execution order can lead to a higher jitter-robustness (see figure 31).

![Figure 31: Improving jitter-robustness with scheduling.](image)

**Heuristic for finding the optimal deployment concerning the jitter-robustness.** As seen in section 5.2.3, a different deployment mapping can lead to a higher jitter-robustness of the system. A brute-force approach for iterating through all possible deployment mappings can be very time consuming. Finding a good heuristic for choosing a jitter-robust deployment remains an open area for research.

**Testing the optimization algorithm on complex models.** Industry-relevant systems can contain many components and channels. Achieving the optimal jitter-robustness in such a system manually is very hard. The automated algorithm (see section 3.4.3) is used to observe how much more jitter-robustness can be achieved. Testing this method on larger models can make statements about its scalability.
### List of Figures

1. A logical architecture deployed on a technical architecture. .......... 8
2. The model, metamodel and meta-metamodel of a state diagram. .......... 10
3. Different granularity levels (vertical) and different software development perspectives (horizontal) according to [TRS+10]. .............. 11
4. The external interface of the filtering function with two inputs and one output channel. .................................................. 13
5. The data flow program and data flow graph. ............................. 15
6. The two possible retiming-transformations that can be executed on a component. .................................................. 17
7. AutoFOCUS 3 structural and automaton view ............................... 19
8. The translation of the weakly and strongly causal components to homogeneous iterative data flow graphs. [Sch11] .................. 21
9. The definition of jitter and the jitter-deviation which is analyzed and handled in this thesis. ............................................... 23
10. A valid AutoFOCUS 3 model satisfying the necessary constraints for jitter-robustness analysis in this thesis. ......................... 25
11. The first step of the retiming-transformation in a logical architecture. 27
12. The second step of the retiming-transformation in a logical architecture. 27
13. The third step of the retiming-transformation in a logical architecture. 28
14. The forth step of the retiming transformation in a logical architecture. 28
15. First basic scenario for improving the jitter-robustness. ...................... 29
16. Second basic scenario for improving the jitter-robustness. ............... 30
17. Third basic scenario for improving the jitter-robustness. ..................... 31
18. The data flow graph containing the eight strongly causal components of the system. ............................................... 34
19. Each output of a component with its guaranteed delays. The value 'X' is shown for those outputs which do not communicate with any environment input ports. ........................................... 36
20. The data flow graph of a strongly causal component of order 4. ........... 38
21. The metamodel containing the new specification class which accepts causality with arbitrary order. ........................................ 39
22. The logical architecture modification of the ACCSystem whose component PCS is retimed with value 2. ............................ 41
23. The data flow graph in CADMOS of a simple sender-receiver system. ........ 42
24. The data flow graph of the sender-receiver system. ......................... 44
25. The embedded hardware which represents the technical architecture of the system. .................................................. 45
26. The data flow graph of the transformed, jitter-robust sender-receiver system. 46
27. The data flow graph of the modified DENTUM ACC model visualized with CADMOS. .................................................. 48
28 The hierarchical view of the ACCSystem in figure (a) and the logical architecture of the root component in figure (b). ................................................. 49
29 The first deployment mapping realized in AutoFOCUS 3 of the DENTUM ACC system. ........................................................................................................ 50
30 The second deployment mapping realized in AutoFOCUS 3 of the DENTUM ACC system. ........................................................................................................ 51
31 Improving jitter-robustness with scheduling. ..................................................... 54
## List of Tables

1. FOCUS specification of a system. ............................................. 13
2. FOCUS specification of component $F$. ................................. 14
3. The sequence of retiming-transformations leading to the optimal jitter-robustness. .......................................................... 35
4. The receiver component, with causality of order 3, delays its outputs with the help of a ring-buffer. ..................................................... 42
5. The receiver buffer and its elements during the period in which the cable is unplugged. .............................................................. 47
References


[Cad] Cadmos is a tool used to visualize, analyze and transform data-flow graphs developed at the Technical University of Munich, Chair IV for Software and System Engineering. http://code.google.com/a/eclipselabs.org/p/cadmos/.


