technische universität
dortmund

Master's thesis

Evaluation Framework for End-to-End Analysis

Robin Edmaier
September 2024

Supervisors:
Prof. Dr. Jian-Jia Chen
Mario Günzel, M.Sc.

TU Dortmund University
Department of Computer Science
Chair of Computer Engineering and Embedded Systems
Design Automation for Embedded Systems Group
https://daes.cs.tu-dortmund.de

Acknowledgements

At first, I would like to thank my mentor Mario Günzel for his support and his valuable feedback for this master's thesis. Thanks to him and Prof. Dr. Jian-Jia Chen, many implementations for end-to-end analysis were provided to the evaluation framework by different research groups.

In addition, I would like to thank Matthias Becker from KTH Royal Institute of Technology, Prof. Jiankang Ren from Dalian University of Technology, Pourya Gohari from Eindhoven University of Technology, and Paolo Pazzaglia from Robert Bosch GmbH, formerly a researcher at Saarland University. Their valuable contributions of source code for end-to-end analyses, along with their permission to publish them as free software together with the framework, have played a crucial role in this work.

Abstract

For real-time embedded systems, it is important to know how long it takes a system from receiving an input until generating a corresponding output. A self-driving vehicle for example has to react in time after perceiving an obstacle in order to avoid it. Determining this reaction time is crucial for the safety of such systems.

Real-time systems in this context can process the incoming data in multiple steps with intermediate results. Since there could be many of these intermediate processing steps, their analysis is not trivial. There exist multiple analysis methods in the literature that compute upper bounds on different end-to-end latency metrics. These analysis methods can have varying assumptions about the system, leading to different results.

Currently, it is challenging to perform a comparison with multiple analysis methods. Researchers have to do exhaustive literature review, check which analyses are applicable to their model and make several implementations that are compatible with each other. In case the underlying programming languages differ, a re-implementation of analysis methods may also be necessary, which further increases the difficulty for a comparison.

The goal of this thesis is to develop an evaluation framework, that combines the various end-to-end analysis methods available in the literature and reduces the effort of comparing them. Therefore, the framework has to be able to perform the analyses on a set of synthetic sample systems, generated by different benchmark algorithms with a unified model. This ensures, that novel analysis approaches developed in the future can be added with little effort to the framework and are comparable with the existing analyses. At the end, the framework is then able to carry out an evaluation on the analysis results, creating different output diagrams that can be used in scientific works.

For using the framework, a graphical user interface, as well as a command line interface should be available. The graphical user interface allows to use the framework easily, giving the user input fields and configuration controls for all possible options. On the other hand, the command line interface can be used for an automated access of the evaluation framework by a script or another application.

After its implementation, the evaluation framework will be publicly released on GitHub as free software, giving other research groups the opportunity to use the framework for integrating their new analysis methods and comparing them with the already existing methods. Hence, it is essential for the evaluation framework to be easily extensible, allowing changes on the analysis methods, benchmark algorithms and evaluation in the future.

Contents

Notation

Tasks

Notation	Meaning
$\mathbb{T}$	task set
τ	task
$\tau(k)$	k-th job of task τ
T_τ^{min}	minimum inter-arrival time of task τ
T_τ^{max}	maximum inter-arrival time of task τ
T_τ	period of task τ
B_τ	best case execution time of task τ
C_τ	worst case execution time of task τ
D_τ	relative deadline of task τ
R_τ	worst case response time of task τ
ϕ_τ	phase of task τ
$\pi(\tau)$	priority of task τ

Jobs

Notation	Meaning
J	job
$J \in \tau$	a job of task τ
r_J	release time of job J
s_J	first time job J enters the running state
f_J	finishing time of job J
c_J	execution time of job J
d_J	absolute deadline of job J
$\mathrm{re}(J)$	read event of job J
$\mathrm{we}(J)$	write event of job J

Cause-Effect Chains

Notation	Meaning
E	cause-effect chain
$E(i)$	i-th task in the cause-effect chain
$E(i)(k)$	k-th job of the i-th task in the cause-effect chain
$\lvert E \rvert$	number of tasks in the cause-effect chain
$\Phi(E)$	first point in time every task in E has released its first job

Job Chains

Notation	Meaning
$c^{E,\mathcal{S}}$	generic job chain
$\vec{c}^{E,\mathcal{S}}$	immediate forward job chain
$\overleftarrow{c}^{E,\mathcal{S}}$	immediate backward job chain
$ac^{E,\mathcal{S}}$	augmented job chain
$\vec{ac}^{E,\mathcal{S}}$	immediate forward augmented job chain
$\overleftarrow{ac}^{E,\mathcal{S}}$	immediate backward augmented job chain
$\vec{ac}_m^{E,\mathcal{S}}$	m-th immediate forward augmented job chain
$\overleftarrow{ac}_m^{E,\mathcal{S}}$	m-th immediate backward augmented job chain

Miscellaneous

Notation	Meaning
$\mathcal{S}$	concrete schedule

List of Abbreviations

Abbreviation	Meaning
ACET	Average Case Execution Time
BCET	Best Case Execution Time
CLI	Command Line Interface
ECU	Electronic Control Unit
GALS	Globally Asynchronized Locally Synchronized
GUI	Graphical User Interface
LET	Logical Execution Time
MDA	Maximum Data Age
MRDA	Maximum Reduced Data Age
MRT	Maximum Response Time
MRRT	Maximum Reduced Response Time
RTOS	Real-Time Operating System
TDA	Time Demand Analysis
WCET	Worst Case Execution Time
WCRT	Worst Case Response Time

Chapter 1

Introduction

1.1 Motivation

In today's world, there are many critical real-time systems that are used to process data. It is crucial for these system, that they finish their computation in time and generate a corresponding output, which reacts to the given input data. Examples can be found especially in the automotive industries, but also in other fields such as avionic or robotic industries.
Verifying the correct timing behaviors of such systems is important to ensure their safety. To do so, the so-called end-to-end latency of a real-time system is analyzed. An end-to-end analysis returns an upper bound on the duration that the system needs to fully process the input.
In the literature, there exist many different end-to-end analysis methods, which compute upper bounds on the end-to-end latency. The implementations of these analysis methods are spread across multiple GitHub repositories and are even done in different programming languages. Comparing two or more analysis methods with the same underlying system is therefore not trivial. An unified evaluation framework that integrates all of the analysis methods present in the literature is currently missing.
Moreover, there are multiple definitions on the end-to-end latency and differing assumptions about the underlying systems, leading to different results for the latencies. This further increases the difficulty to compare two or more analysis methods.

1.2 Goals

The goal of this master's thesis is to develop an evaluation framework, that can be used to evaluate and compare different end-to-end analysis methods on various systems, generated with the help of different benchmarks. The framework itself should be designed in such a way, that it reduces the difficulty of comparing multiple end-to-end analysis methods.

Therefore, the framework should be primarily usable via a graphical user interface (GUI) and additionally via a command line interface (CLI) for automated usage. After evaluating the different end-to-end analysis methods, the framework should generate an output in the form of diagrams that visualize the results of the conducted evaluation. These diagrams are stored in convenient formats, so that they can be easily included in other scientific work.

Furthermore, the framework should include a large variety of different end-to-end analysis methods that can be used for comparisons. It should also be easily extensible, so that it is possible to add more analysis methods to the framework in the future.

Finally, the source code of the framework will be publicly released on GitHub, allowing other research groups to use the framework and test their newly developed analysis methods against the already existing ones. For this reason, there is a special incentive to document the source code and the usage of the framework as precisely as possible.

1.3 Structure of this Thesis

At first, the theoretical background of this thesis is explained in chapter 2. It includes a short introduction on end-to-end analysis as well as the model definition that is used throughout this thesis and definitions for the metrics under analysis. Furthermore, the chapter also presents a large number of end-to-end analyses, which are available in the literature and integrated in the evaluation framework. This also includes two novel analysis methods, that were developed based on existing analysis methods during the implementation of the framework. Lastly, two different benchmarks for generating task sets and cause-effect chains are described.

After that, chapter 3 covers the actual implementation of the evaluation framework. Therefore, three different GUI frameworks that were possible candidates for the evaluation framework are presented and compared. Afterwards, the basic layout of the framework with its file structure is given, followed by a more in-depth explanation of the different components, analysis methods and implemented features.

Chapter 4 describes how the framework can be used to conduct a comparison between multiple end-to-end analysis methods. This includes all necessary steps to setup the system environment before the evaluation framework can be started successfully. Then follows a theoretical explanation on how the graphical user interface and the command line interface can be used. To give the reader an even better impression on how the framework works, an example evaluation is carried out with both interfaces. Additionally, detailed instructions on how to extend the framework and add another end-to-end analysis method are given at the end of the chapter.

The summary of this thesis results is given in chapter 5. It also includes a future work section, which mentions some possible extensions for this thesis and the evaluation framework.

Chapter 2

Background

This chapter deals with the theoretical background of this master's thesis. To give the reader a better understanding of the topic, at first a short introduction with an example for end-to-end analysis is provided. Next, the end-to-end analysis basics are explained and a corresponding model for the problem is defined. The different metrics used by the analysis methods are then elaborated. Afterwards, multiple end-to-end analyses from the literature as well as two novel analyses are presented and categorized. Finally, two different methods that can be used for creating benchmarks for these end-to-end analyses are examined in more detail.

2.1 End-to-End Analysis Introduction

End-to-end analysis is used to determine, how long it takes a system to completely process an external input event. Processing the input can happen with many intermediate results. When the system is done with its computation, it generates an output for that particular input, which leads to an externally measurable effect. Hence, end-to-end analysis methods try to compute upper bounds for the time interval starting at the cause (input-event) and ending at the effect (output-event). For the analysis, so called cause-effect chains are often used, that describe the system, which processes the input.
Cause-effect chains usually consist of multiple tasks that communicate and process the external input in multiple steps with intermediate results. An example is given in figure 2.1, where the external inputs are the sensor values of a self-driving race car.
A LIDAR sensor measures the distances of the car to its surroundings. As the car reaches a curve, the sampled values of the sensor change and the car has to react in order to not collide with the wall of the track. In this example it is crucial, that the system processes the cause (sensor value, indicating the curve ahead) in time. The effect should be a corresponding output, that controls the actuators of the vehicle in such a way, that the car safely maneuvers the curve without leaving the track or colliding with the walls.

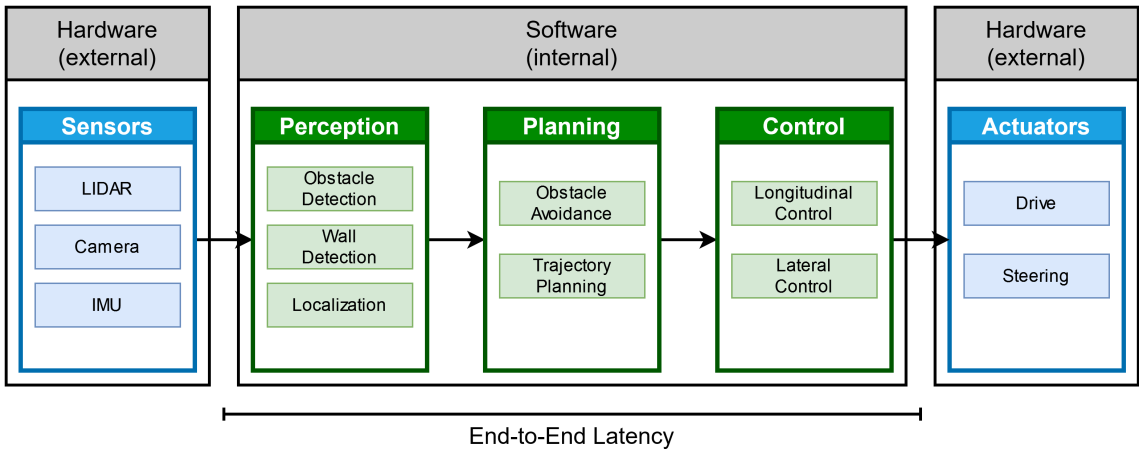


Figure 2.1: Overview of the system components of a self-driving vehicle (similar to figure 1 from [3])

Before this effect can be measured, multiple task will be executed by the system for varying purposes (e.g. for detecting the walls of the track, calculating a trajectory, etc.). Determining an upper bound on the time the system needs to compute the output is therefore heavily dependent on the tasks in this cause-effect chain. A more detailed explanation on cause-effect chains is given section 2.2.2.

There exist many different analysis methods to compute upper bounds for end-to-end latencies. Depending on the analysis method, different assumption about the underlying system are made, which leads to differing results. The most important assumptions about the systems include the communication mechanism between the tasks, the release pattern of the tasks in the underlying task set and whether all tasks are executed on the same or on different electronic control units. Furthermore, there are four different metrics that are used to describe the end-to-end latency of the system. All of these assumptions about the system and the metrics for the analysis will be explained in more detail in the following sections.

2.2 Model Definition

Before going into the details about the theoretical background of the different end-to-end analysis methods, a common system model is introduced. This model will be used for the explanations of the end-to-end analysis methods and extended or adjusted during the description of the individual methods, as they may require additional definitions. Most of the notation is similar to the notation used by Günzel et al. in [19].

2.2.1 Tasks, Jobs and Task Sets

A task τ is a program in execution, which has a worst case execution time (WCET) C_τ and a best case execution time (BCET) B_τ. Each task releases jobs recurrently, with each job being an instance of the task. The i-th instance of a task is therefore the i-th job released by that task. Tasks can follow different release patterns, which determine when they release jobs:

- *Periodic:* Periodic tasks release jobs at regular time intervals defined by their period T_τ. The first job is released at the phase ϕ_τ of the task, and subsequent jobs are released after another period T_τ has passed.

- *Sporadic:* Sporadic tasks do not have a fixed period at which new jobs are released. Instead, a sporadic task can release a job at any time during the interval $[T_\tau^{min}, T_\tau^{max}]$ after the release of a previous job. These boundaries are called minimum and maximum inter-arrival time, as they describe the minimum and maximum time between the release of two subsequent jobs of a sporadic task.

All tasks are assumed to have implicit deadlines, meaning the relative deadline D_τ of a periodic task is equal to its period T_τ. For sporadic tasks, the relative deadline is determined by the minimum inter-arrival time T_τ^{min}.
Tasks are organized into task sets $\mathbb{T}$, which are sets of finitely many tasks $\tau \in \mathbb{T}$. These task sets can consist of either periodic or sporadic tasks, influencing how jobs are released and scheduled. Analysis methods that are capable of analyzing cause-effect chains with sporadic task sets can also be used to analyze cause-effect chains with periodic task sets. Therefore, all end-to-end analysis methods can be used in case of periodic task sets, but only some in case of sporadic task sets.
As already mentioned, jobs are instances of tasks. During its execution, each job of the system follows the same pattern. At first, the job reads an input from a shared memory. Next, the job calculates an output for the given input. After it is done with its computation, the job writes its output back to the shared memory, so that a following job from another task could use this output as an input again. The exact time points of the reads and writes on the shared memory are dependent on the communication mechanism, later presented in section 2.2.3.
The release time of a job $J \in \tau$ is denoted as r_J and the finishing time as f_J. s_J describes the first point in time, job J enters the running state. The actual execution time c_J of the job lies within the interval $[B_\tau, C_\tau]$ and the absolute deadline d_J of a job J is calculated from its release time plus the relative deadline D_τ of its task. Lastly, the worst case response time (WCRT) R_τ of a task τ is defined as the maximum length of the time interval $[r_J, f_J]$ of all $J \in \tau$.

2.2.2 Cause-Effect Chains

A cause-effect chain $E = (\tau_1 \rightarrow ... \rightarrow \tau_n)$ is an ordered sequence of $|E|$ communicating tasks. Each task $E(i)$ takes an input and produces an output, that can be used as input for the following task in the chain $E(i+1)$. The first task of the chain $E(1)$ takes an external input (cause), which is typically assumed to be a sampled sensor value, and the last task of the chain $E(|E|)$ produces the output of the chain (effect), which is simply written to the shared memory again. This last output could be utilized as an input by another external system component, for example to control a motor.
The literature differentiates between two kinds of cause-effect chains, namely time-triggered cause-effect chains and event-triggered cause-effect chains.
In time-triggered cause-effect chains the job releases of a task are triggered by a timer. A new job arrives, according to the period or the minimum/maximum inter-arrival time of its task.
In event-triggered cause-effect chains the release of a job is triggered by the appearance of an event. Such an event can for example occur after the preceding task in the cause-effect chain wrote its output to the shared memory. This way, a new job is only released once the preceding task generated a new output, that can be further processed. Analogously, a job may also be triggered by an outside event, which can occur when new sensor data is available in the shared memory. It is possible to model event-triggered tasks as sporadic tasks, in case the event that triggers the release of a new job can be analytically bounded within a minimum and maximum inter-arrival window.
This thesis only compares different analyses methods that are based on time-triggered cause-effect chains. Event-triggered cause-effect chain analyses are not covered, since their results cannot be directly compared to the result of analyses methods based on time-triggered cause-effect chains, because the underlying task model is too different. For the rest of this thesis the term cause-effect chain always refers to a time-triggered cause-effect chain.
In systems with real-world applications, data dependencies are usually far more complex than a single cause-effect chain and are therefore modeled with data dependency graphs. A cause-effect chain can then be used to analyze one particular path from a source to a sink of this graph. To completely analyze the graph, all possible paths from any source to any sink have to be constructed as cause-effect chains and analyzed separately in order to determine the latency characteristics of the whole system.

2.2.3 Communication Mechanisms

As already mentioned, there can be multiple tasks in a cause-effect chain that produce intermediate results and need a way to communicate these intermediate results to the following task in the chain. To do so, the results are usually stored in a shared memory

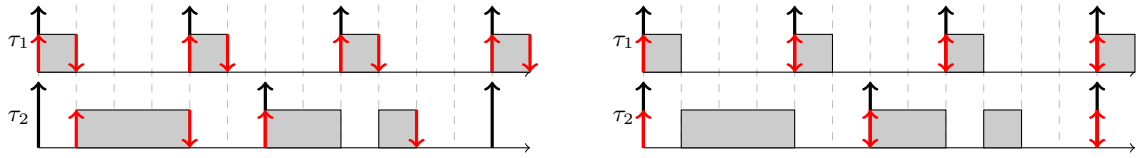


Figure 2.2: Visualization of the different communication mechanisms; The system on the left uses implicit, the system on the right LET communication; Red arrows facing up symbolize a read, red arrows facing down a write event

or shared registers, that can be accessed by both the sending and the receiving task. Therefore, the jobs of a task in the cause-effect chain have to perform reads and writes on the shared memory. At the read event re(J) the job J reads its input from the shared memory and at we(J) the write event of the job takes place. The different communication mechanisms have different assumptions about when these read and write events occur:

- *Implicit Communication:* In the implicit communication model a job J reads input data from the shared memory as it starts with its computation. When it is done with its computation it writes the output back to the shared memory, so that the next job in the chain can use it. Therefore, the read and write events are constrained by re(J) = s_J and we(J) = f_J. A visualization can be seen in the left diagram of figure 2.2.

- *Logical Execution Time (LET)* [21]*:* Additionally there exists the logical execution time (LET). In this communication model the reading process of the job J is done right at the release of the job. The write to the shared memory happens at the deadline of the task. Hence, the events are constrained by re(J) = r_J and we(J) = d_J. An example is shown in the right diagram of figure 2.2.

- *Explicit/Direct Communication:* When the explicit communication model is used, a job can access the shared memory and perform read and write events at any point in time during its execution. The exact timing of these events is therefore dependent on the actual computations of the job, which are very likely unknown when performing the analysis in advance.

The communication mechanisms are sorted according to their popularity in the literature. Most analysis methods for end-to-end latencies assume implicit communication, some assume LET communication and almost none assume the explicit/direct communication. The explicit communication models the tightest bounds for the read and write events, but is dependent on the actual execution of the job during runtime, which is most likely not known in advance. In particular, it is not possible to determine when the read and write events of the job occur, based on a calculated schedule, since a job is allowed to perform reads and writes at any time during its execution. Without the knowledge about the jobs internal processes, an analysis with the explicit communication model is not possible.

Next are the implicit communication read and write events, which are more loosely compared to the explicit communication, but far easier to determine. The read and write events are still dependent on the actual execution of the job during runtime. However, both events can be approximated with a schedule and do not require specific knowledge about the jobs internal processes.
The bounds in the LET communication model are even more pessimistic, than the implicit communication bounds for the memory access events. On the other hand, LET has the benefit that it is independent of the execution behavior of the task. The only necessary assumption is that all jobs finish their execution before their deadline.
Depending on the underlying communication mechanism of the tasks, an analysis of a cause-effect chain can lead to very different results. The comparability of two analysis methods designed for different communication mechanisms is therefore drastically reduced.

2.2.4 Scheduling

A task set is scheduled onto one or multiple Electronic Control Units (ECUs). Most analysis methods assume an partitioned fixed-priority preemptive scheduling, which is hence used in this thesis. This kind of scheduling has the following characteristics:

1. A task is statically assigned to a single ECU. It is not possible for two jobs of the same task to be executed on different ECUs. The ECU of a task is determined by $ECU(\tau)$.

2. The priority of a task (denoted by $\pi(\tau)$) is fixed, meaning it cannot change during runtime. Therefore, all jobs of a task have the same priority.

3. Jobs with lower priority are preempted by higher priority jobs during runtime. In case a job $J_h \in \tau_h$ arrives and a job $J_l \in \tau_l$ is currently executing, J_h preempts J_l if $\pi(\tau_h) > \pi(\tau_l)$. J_l continues with its remaining execution after all higher priority jobs are finished.

A schedule $\mathcal{S}$ is a concrete timetable for a task set $\mathbb{T}$, where all tasks of the task set are scheduled according to their priorities onto the different ECUs, including the restrictions mentioned above. Additionally, each job $J \in \tau$ will receive an individual execution time value $c_J \in [B_\tau, C_\tau]$.

2.2.5 Job Chains

A job chain $c^{E,\mathcal{S}}$ (first described by Dürr et al. in [7]) describes one possible data flow for a cause-effect chain E and a concrete Schedule $\mathcal{S}$. Similar to a cause-effect chain, it is a sequence $(J_1, ..., J_{|E|})$ of $|E|$ communicating jobs, where J_i is a job of $E(i)$. Since the jobs

are communicating with each other, the read event of the following job has to be after the write event of the previous job in the sequence. Therefore $\text{re}(J_i + 1) \geq \text{we}(J_i), \forall i \in [1, |E| - 1]$.

This basic definition of a job chain is extended in the following definitions.

An immediate forward job chain $\vec{c}^{E,\mathcal{S}}$ [7], is a job chain $c^{E,\mathcal{S}}$ with additional restrictions on the choice of the jobs. Specifically, each job in the chain is the first job that is able to read the data generated by the previous job in the chain:

$$J_{i+1} = \arg\min_{J \in E(i+1), \text{re}(J) \geq \text{we}(J_i)} \text{re}(J), \forall i \in [1, |E| - 1]$$

Analog, an immediate backward job chain $\overleftarrow{c}^{E,\mathcal{S}}$ [7], is a job chain $c^{E,\mathcal{S}}$, where each preceding job in the chain is the last job that produced data, that can be read by the following job:

$$J_{i-1} = \arg\max_{J \in E(i-1), \text{we}(J) \leq \text{re}(J_i)} \text{we}(J), \forall i \in [2, |E|]$$

Additionally, there exist augmented job chains $ac^{E,\mathcal{S}} = (z, J_1, ..., J_{|E|}, z')$ (first defined by Günzel et al. in [17]), which are job chains $c^{E,\mathcal{S}}$, that are augmented by a sample event z at the beginning and an actuation event z' at the end, with $z \leq \text{re}(J_1)$ and $z' \geq \text{we}(J_{|E|})$.

In [17], immediate forward augmented job chains $\vec{ac}_m^{E,\mathcal{S}} = (z, J_1, ..., J_{|E|}, z')$ are defined as augmented job chains, with $(J_1, ..., J_{|E|})$ being an immediate forward job chain. Furthermore the event $z = \text{re}(E(1)(m))$ is set to the m-th read-event of the first task of the underlying cause-effect chain E and J_1 is the following job $E(1)(m + 1)$. The actuation event z' is set to the write event of the last task of the cause-effect chain ($z' = \text{we}(J_{|E|})$.

Lastly, the immediate backward augmented job chain $\overleftarrow{ac}_m^{E,\mathcal{S}} = (z, J_1, ..., J_{|E|}, z')$ [17] is an augmented job chain, where $(J_1, ..., J_{|E|})$ is an immediate backward job chain. For the immediate backward augmented job chain, the actuation event z' is at the m-th write event of the last task $E(|E|)$ of the cause-effect chain E. The job $J_{|E|}$ is then the $(m-1)$-th job of $E(|E|)$, ending the immediate backward job chain. The input event z of the immediate backward augmented job chain is set to the read event $\text{re}(J_1)$.

An example is shown in figure 2.3. (J_1^1, J_2^2, J_3^2) is a job chain, but it is not an immediate forward job chain, since the first job after the write event of J_1^1 is not J_2^2, but J_2^1. It is also not an immediate backward job chain, since the last write event of τ_1 before the read event of J_2^2 is not performed by J_1^1, but by J_1^2. (J_1^3, J_2^3, J_3^3) is an immediate forward job chain, but it is not an immediate backward job chain, since the write event of J_1^3 is not the latest write event of τ_1 before the read event of J_2^3. (J_1^5, J_2^4, J_3^4) is an immediate forward job chain and also an immediate backward job chain.

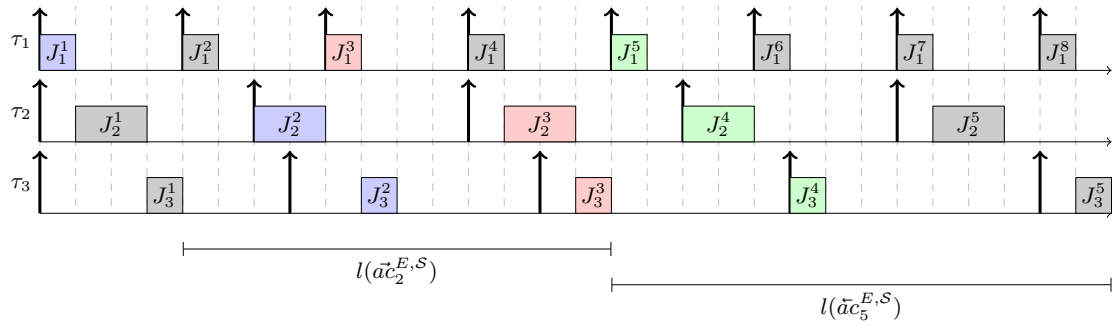


Figure 2.3: Visualization of different analysis metrics on an immediate backward job chain for a task set with implicit communication

An immediate forward augmented job chain can be created using the immediate forward job chain (J_1^3, J_2^3, J_3^3) and adding the sampling event $z = J_1^2$ and actuation event $z' = J_3^3$. The resulting augmented job chain is $\big(\mathrm{re}(J_1^2), J_1^3, J_2^3, J_3^3, \mathrm{we}(J_3^3)\big)$. It is the second immediate forward augmented job chain $(\vec{ac}_2^{E,\mathcal{S}})$ of the cause-effect chain E, as it starts at the read-event of the second job of $E(1)$.
Similarly, $\overleftarrow{ac}_5^{E,\mathcal{S}} = \big(\mathrm{re}(J_1^5), J_1^5, J_2^4, J_3^4, \mathrm{we}(J_3^5)\big)$ is the 5-th immediate backward augmented job chain and can be created from the immediate backward job chain (J_1^5, J_2^4, J_3^4).

The length of a job chain $l(c^{E,\mathcal{S}})$ is defined by [7] as the time between the read event of the first task $\mathrm{re}(J_1)$ and the write event of the last task $\mathrm{we}(J_{|E|})$ of the chain:

$$l(c^{E,\mathcal{S}}) = \mathrm{we}(J_{|E|}) - \mathrm{re}(J_1)$$

In the case of an augmented job chain the length of the augmented job chain $l(ac^{E,\mathcal{S}})$ is defined by [17] as the time between the sampling event z and the actuation event z':

$$l(ac^{E,\mathcal{S}}) = z' - z$$

2.2.6 Inter-ECU Communication

In many real-world scenarios computation processes are separated on different Electronic Control Units (ECUs) that communicate via broadcast busses to exchange data. For example CAN busses are often used in automotive industries to connect multiple ECUs and allow them to interchange data between processes. It is therefore possible to have cause-effect chains that span across multiple ECUs, e.g. the first ECU receives an input signal and communicates via the CAN bus with a second ECU which again communicates with a third ECU that produces an output needed for an actuation.

These cause-effect chains are called interconnected cause-effect chains in this thesis and the term local cause-effect chain is used for cause-effect chains that are restricted to a single ECU. Furthermore, each task runs on one specified ECU. In case there are multiple ECUs available, the ECU of a task is determined by $ECU(\tau)$. Jobs of the same task are always executed on the same ECU.
Analysis methods designed for interconnected cause-effect chains can directly be applied to local cause-effect chains, since a local cause-effect chain is a special interconnected cause-effect chain with only one ECU.
The literature distinguishes two kinds of multi-ECU-systems:

1. Globally synchronized systems
2. Globally asynchronized locally synchronized (GALS) systems [14]

In globally synchronized systems, the local clocks of all ECUs are synchronized. This allows to compare time points among different ECUs to achieve tighter analysis, but increases the complexity of the system itself, as it has to actively synchronize all of the local clocks. GALS systems on the other hand are harder to analyze, but easier to implement, as they do not require a dedicated synchronization mechanism. The local clocks of the different ECUs are not synchronized. Therefore only tasks running on the same ECU have access to the same local clock, making the system locally synchronized.
Moreover it is possible to use analysis methods designed for local cause-effect chains on interconnected cause-effect chains, by inserting an additional communication task τ_c into the cause-effect chain. This task models the communication between two ECUs. When the task starts with its computation it reads the data in the shared memory of the first ECU and at the end of its computation the data has been transferred to the shared memory of the second ECU, so that any task running on the second ECU could access this data without any delay. The communication task is only a model for the communication, therefore it does not consume any processing time of the communicating ECUs.
Another possibility is to assume a globally shared memory, that can be accessed from every ECU. This simplifies the model, as there is no need for introducing an additional communication task into the chain. A task can simply access the output of a task running on another ECU.

2.3 Analysis Metrics

In the literature, cause-effect chains are often used to analyze the end-to-end latency of a given system. However, their analysis results differ, as there are different analysis metrics that are used to determine the lengths of cause-effect chains. In particular the start points (causes) and the end points (effects) of those chain are often differently defined, depending sometimes on the use-case of the analysis.

The starting point can either be defined as that point in time, where a new external input sample is available to the system or as that point in time, when this new external input is read by the first task of the chain. The same applies to the end point of the cause-effect chain. It can either be defined as the first or the latest point in time, the lasts tasks output is available in the shared memory.
Of course, these different definitions have a significant impact on the results of the analysis methods, making them more difficult to compare. Nevertheless, for each metric there are usually multiple analysis methods available for an evaluation.
Most end-to-end analyses methods focus on analyzing the bounds of the Maximum Reaction Time (MRT) or the Maximum Date Age (MDA). These two metrics have differing views on the end-to-end latency, since the MRT considers the latency in forward direction (e.g. how long does it take the system to produce an output after an input signal?), whereas the MDA considers the latency in backward direction (e.g. if the system produces an output, how old were the corresponding inputs?).
There exist multiple definitions for the MDA in the literature. To prevent any ambiguity Günzel et al. [17] introduced the term Maximum Reduced Data Age (MRDA). The precise definitions for these metrics used in this thesis are as follows:

- The *MRT* describes the longest time interval starting from the arrival of an external input and ending at the first output corresponding to that input (maximum button to action delay). It is equivalent to the supremum of the lengths of all immediate forward augmented job chains after a warm-up phase [18].

- The *MRRT* (Maximum Reduced Reaction Time) is a rarely used metric, which was first defined by Günzel et al. in [18]. It is the longest time interval, starting when the external input is first read and ending at the first corresponding output to that input. The MRRT is equivalent to the supremum of the lengths of all immediate forward job chains after a warm-up phase [18].

- The *MDA* (also called worst-case data freshness) is the longest time period between sampling an input signal and an actuation that is based on that particular sample. It is equivalent to the supremum of the lengths of all immediate backward augmented job chains after a warm-up phase [18].

- The *MRDA* is the longest time period between sampling an input signal and producing an output that is based on that particular sample. It is equivalent to the supremum of the lengths of all immediate backward job chains after a warm-up phase [18].

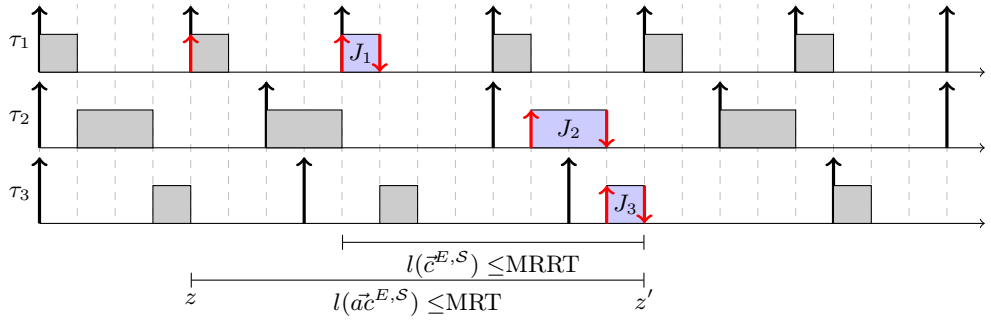


Figure 2.4: Visualization of MRT and MRRT on an immediate forward (augmented) job chain for a task set using implicit communication with the relevant read and write events

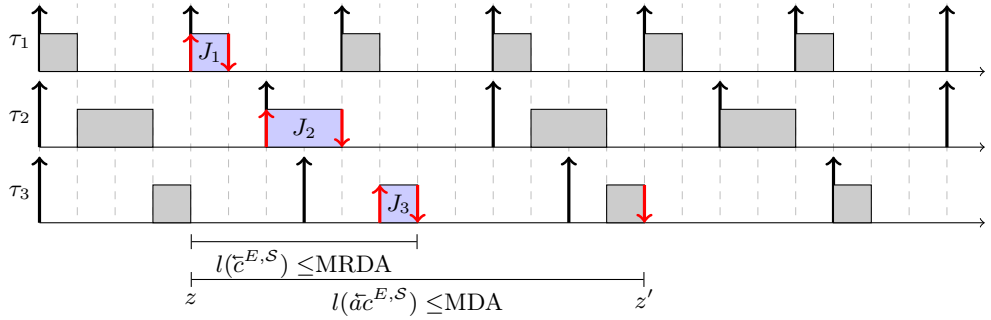


Figure 2.5: Visualization of MDA and MRDA on an immediate backward (augmented) job chain for a task set using implicit communication with the relevant read and write events

Each of the analysis metrics is visualized for one example job chain in figures 2.4 and 2.5. The example uses implicit communication and periodic tasks, but the metrics are also applicable in case of LET communication and sporadic tasks.

Dürr et al. [7] first proved that the MRDA is upper bounded by the MRT. It was then shown by Günzel et al. in [17] and [14] that the MDA is also upper bounded by the MRT. Furthermore, Günzel et al. [18] proved the equivalence of MRT and MDA for a wide range of system models, including systems with implicit/LET communication, periodic/sporadic task sets, over-/undersampling, fixed-/dynamic-priority schedulers and synchronize/asynchronized distributed systems. This means that any analysis method for one of both metrics can also be seen as an analysis for the other metric as well. Therefore, it is possible to directly compare the results of any two analysis methods computing a bound for the MRT or the MDA. A sketch for the prove is given later in section 2.4.10.

2.4 End-to-End Analysis Methods

This section presents the concepts behind different end-to-end analysis methods currently available in the literature and categorizes them according to the different characteristics described in section 2.3. The analysis methods are ordered chronologically according to their publication. A comprehensive list with all analysis methods and their features is given in the appendix section A.2 of this thesis.
The section also includes two novel analysis methods, that were developed during the work on this master's thesis.

2.4.1 Davare 2007

One of the first end-to-end analysis was presented by Davare et al. [6]. The basic idea of this approach is very straight forward. To compute the worst case end-to-end latency, the period and the worst case response time of each task are summed up. This analysis method assumes, that an input signal is only consumed during the following period of the task, as the task of the current period could have already started with its computation. The resulting formula for calculating the MRT is as follows:

$$\mathrm{MRT}(E) \leq \sum_{i=1}^{|E|} \left(T_{E(i)}^{max} + R_{E(i)} \right) \tag{2.1}$$

Since this approach returns a very pessimistic upper bound on the maximum response time of a task chain, it is often used as a baseline approach for comparing the latencies of other end-to-end analysis methods relative to latencies returned by Davare et al. [6].
The analysis is applicable for periodic and sporadic task sets, even though Davare et al. [6] only applied it to periodic task sets. It is not applicable for cause-effect chains using the LET communication paradigm, but only for cause-effect chains with implicit communication. With the cutting theorem presented later in this thesis, the analysis can also be utilized for interconnected cause-effect chains to serve as a baseline as well.

2.4.2 Becker 2016/2017

Becker et al. analyze the end-to-end latencies in [2] and [1] with the help of of a data propagation tree. Every node in the tree represents a job of the cause-effect chain and is connected via a directed edge to all its succeeding jobs. Succeeding jobs are only those jobs, that are able to consume the data produced by the job in the parent node.
The tree is constructed starting with an instance of the first task of the cause-effect chain. Then, all jobs of the second task, that are able to consume the data produced by the first job are connected to the root. This process repeats, until the last task of the cause-effect chain is reached and the corresponding jobs are connected as leaf nodes to the tree. When

	No knowledge	WCRTs known	Exact schedule	LET comm.
$\text{Rmin}(\tau_i(j))$	$\phi_i + (j-1) \cdot T_i$	$\phi_i + (j-1) \cdot T_i$	$s_{\tau_i(j)}$	$(j-1) \cdot T_i$
$\text{Rmax}(\tau_i(j))$	$j \cdot T_i - C_i$	$\text{Rmin}(\tau_i(j)) + R_i - C_i$	$\text{Rmin}(\tau_i(j))$	$\text{Rmin}(\tau_i(j))$
$\text{Dmin}(\tau_i(j))$	$\text{Rmin}(\tau_i(j)) + C_i$	$\text{Rmin}(\tau_i(j)) + C_i$	$f_{\tau_i(j)}$	$j \cdot T_i$
$\text{Dmax}(\tau_i(j))$	$\text{Rmax}(\tau_i(j+1)) + C_i$	$\text{Rmax}(\tau_i(j+1)) + C_i$	$f_{\tau_i(j)}$	$(j+1) \cdot T_i$

Table 2.1: Definitions of $\text{Rmin}(J)$, $\text{Rmax}(J)$, $\text{Dmin}(J)$, $\text{Dmax}(J)$ for different levels of knowledge about the system; The cases "No knowledge", "WCRTs known" and "Exact schedule" assume implicit communication between tasks. Table taken from Becker et al. [1]

a new leaf node gets connected to the tree, the end-to-end latency of this particular path (which is a single job-chain) can be computed with the start (or release) and finishing (or deadline) of the first and last job.

To compute the maximum latency of the given cause-effect chain, all data propagation trees $\text{DPTs}(E)$ starting within the first hyperperiod of the cause-effect chain are constructed. Afterwards, Becker et al. [2] compute an upper bound for the MRDA with the maximum length of any path from a root node to the corresponding leaf nodes over all constructed trees.

$$\text{MRDA}(E) \leq \max_{tree \in \text{DPTs}(E)} \text{MaxTreeAge}(tree)$$

$$\text{MaxTreeAge}(tree) = \max_{\tau_i(j)=\text{root}(tree), \tau_k(l) \in \text{leaves}(tree)} (\text{Rmax}(\tau_k(l)) + C_k - \text{Rmin}(\tau_i(j)))$$

For constructing the tree it is important to determine, which job is able to consume data generated from a previous job in the chain. To analyze this, Becker et al. use read and data intervals in [2]. The read interval $[\text{Rmin}(J), \text{Rmax}(J)]$ describes the interval, in which a job J could read data generated by a predecessor and still meet its own deadline. The data interval $[\text{Dmin}(J), \text{Dmax}(J)]$ of a job J describes the time interval, during which the data produced by a job could be present in the shared memory. In their papers [2] and [1] they describe different ways to compute the bounds of these interval, based on different levels of knowledge about the system. An overview for how they can be calculated is given in table 2.1.

Additionally, to the described analysis method, Becker et al. also describe a method for synthesizing so called job-level dependencies. These are additional constraints at the job-level of a given system to ensure a certain end-to-end latency. Since this step is used to change parts of the system with the idea to reduce the end-to-end latency, it is not considered in the end-to-end evaluation framework. Active modifications to reduce the end-to-end latency are not in the scope of this evaluation framework, but rather the passive examination of cause-effect chains. Hence, only the basic analysis as described above is implemented in the framework.

The analysis methods are applicable for periodic task sets using implicit communication and are restricted to local cause-effect chains.

2.4.3 Hamann 2017

In [20] Hamann et al. explain how to calculate a simple upper bound for end-to-end latencies with cause-effect chains using LET communication. Their analysis method is another very basic approach for computing the end-to-end latency of a cause-effect chain and is thus similar to [6]. They verbally describe the following formula:

$$\mathrm{MRT}(E) \leq \sum_{i=1}^{|E|} \left(T_{E(i)}^{max} + D_{E(i)} \right) \tag{2.2}$$

Analog to [6] this analysis method is very pessimistic and can be seen as a baseline for cause-effect chains with LET communication. It is also applicable for both periodic and sporadic task sets.

2.4.4 Kloda 2018

Kloda et al. present a recursive algorithm in [22] to compute the MRT for a given cause-effect chain. In their analysis, they focus on the latencies between two consecutive jobs in the chain, the producer job J_p and the consumer job J_c respectively. They denote the following property for the release of the consumer job in case both jobs are executed on synchronized ECUs:

$$r_{J_c} = \begin{cases} \left\lceil \frac{f_{J_p}}{T_{\tau_c}} \right\rceil T_{\tau_c} & \text{if } \pi(\tau_p) < \pi(\tau_c) \vee ECU(\tau_p) \neq ECU(\tau_c) \\ \left\lceil \frac{r_{J_p}}{T_{\tau_c}} \right\rceil T_{\tau_c} & \text{otherwise} \end{cases} \tag{2.3}$$

This property is then used to calculate an upper bound on the latency of a single cause-effect chain E for the specific release time $r_{E(1)(i)}$ of the first job of the chain. To determine the worst case response time of the cause-effect chain Kloda et al. therefore repeat the latency calculation for every possible release time of the first job of the chain within the hyperperiod of the underlying task set ($n \cdot T_{E(1)} \in [0, H)$ with $n \in \mathbb{N}$) in algorithm 2.2.

```
1:  function KLODA_RECURSIVE(E, t)          ▷ t is the release of a job from E(1)
2:      τ_p ← E(1)
3:      if |E| = 1 then
4:          return R_{τ_p}
5:      end if
6:      E' ← E without τ_p
7:      τ_c ← E'(1)
8:      compute r_{J_c} using formula 2.3
9:      return r_{J_c} − t+ KLODA_RECURSIVE(E', r_{J_c})
10: end function
```

Algorithm 2.1: Algorithm 1 from Kloda et al. [22]

```
1: function KLODA(E)
2:     H ← hyperperiod of the underlying task set of E
3:     Latency ← 0
4:     for all r_1 ∈ {0, T_E(1), ..., H − T_E(1)} do
5:         Latency ← max(Latency, KLODA_RECURSIVE(E, r_1))
6:     end for
7:     return T_E(1) + Latency
8: end function
```

Algorithm 2.2: Algorithm 2 from Kloda et al. [22]

The analysis method can be applied to synchronous periodic task sets that use implicit communication. It can only be applied to interconnected cause-effect chains with globally synchronized clocks, but not on GALS systems.

2.4.5 Dürr 2019

Dürr et al. [7] provide two separate end-to-end analyses, one for the MRT and one for the MDA. Both analyses are applicable for periodic task sets as well as for sporadic task sets using implicit communication. Additionally, their analyses are designed for interconnected cause-effect chains, allowing to calculate upper bounds for the latencies in distributed GALS systems.

Since their definition for the MDA coincides with the definition of the MRDA used in this thesis, their MDA analysis will be regarded as a MRDA analysis.

The upper bounds of their analyses are achieved by taking a closer look at the arrival times r_i, r_{i+1} of two jobs within an job chain. For the MRT-bound the two consecutive jobs J_i and J_{i+1} of an immediate forward job chain $\vec{c}^{E,\mathcal{S}}$ are considered.

First, in case the following job arrives before the current job has finished its execution ($r_{i+1} < f_i$), the difference $r_{i+1} - r_i$ is bounded by the response time $R_{E(i)}$.

For the case that the following job does not arrive before the current fob has finished its execution ($r_{i+1} \geq f_i$), they derive the bound $T^{max}_{E(i+1)} + R_{E(i)}$ for the difference $r_{i+1} - r_i$. This bound can be further reduced to $T^{max}_{E(i+1)}$ in case the priority of the current task is higher than or equal to the priority of the following task.

They ultimately combine the different cases in the following formula:

$$r_{i+1} - r_i \leq \max\left\{R_{E(i)}, T^{max}_{E(i+1)} + R_{E(i)} \cdot [P]\right\} \tag{2.4}$$

$$[P] = \begin{cases} 1 & \text{if } \pi(E(i)) < \pi(E(i+1)) \vee ECU(E(i)) \neq ECU(E(i+1)) \\ 0 & \text{otherwise} \end{cases} \tag{2.5}$$

At first, the Iverson Bracket $[P]$ is only constructed for the single ECU case. Then, Dürr et al. extend their analysis to the multi-ECU case for interconnected cause-effect chains. Therefore, they adjust the definition of the Iversion Bracket $[P]$ by adding the condition $ECU(E(i)) \neq ECU(E(i+1))$ to the first case. The final formula is given in equation 2.5. Upon equation 2.4, Dürr et al. provide the following equation to bound the maximum response time of a given cause-effect chain:

$$\mathrm{MRT}(E) \leq T^{max}_{E(1)} + R_{E(|E|)} + \sum_{i=1}^{|E|-1} max\left\{R_{E(i)}, T^{max}_{E(i+1)} + R_{E(i)} \cdot [P]\right\} \tag{2.6}$$

A similar approach is used to analyze the arrival times of two consecutive jobs within an immediate backward job chain. Dürr et al. prove, that the MRDA of a cause-effect chain is upper bounded by the following equation:

$$\mathrm{MRDA}(E) \leq R_{E(|E|)} + \sum_{i=1}^{|E|-1} T^{max}_{E(i)} + R_{E(i)} \cdot [P] \tag{2.7}$$

Dürr et al. also show, that their MRT analysis analytically dominates the analysis of Davare et al. [6]. The worst-case of equation 2.6 is achieved by setting the value of $[P]$ to 1 for every iteration of the sum. In this case the resulting equation is equal to equation 2.1.

2.4.6 Günzel 2021/2023 (Inter-ECU)

In [14] Günzel et al. present the cutting theorem, which they use to "cut" interconnected cause-effect chains into local cause-effect chains and analyze them separately. The theorem provided by them is as follows:

2.4.1 Theorem (Cutting Theorem [14]). *Let $E = (\tau_1 \rightarrow ... \rightarrow \tau_{|E|})$ be any cause-effect chain. Furthermore, let $k \in \{1, ..., |E| - 1\}$ be some integer. For the cause-effect chains $E_1 := (\tau_1 \rightarrow ... \rightarrow \tau_k)$ and $E_2 := (\tau_{k+1} \rightarrow ... \rightarrow \tau_{|E|})$ holds that*

$$\mathrm{MRT}(E, \mathcal{S}) \leq \mathrm{MRT}(E_1, \mathcal{S}) + \mathrm{MRT}(E_2, \mathcal{S}), \tag{2.8}$$

$$\mathrm{MDA}(E, \mathcal{S}) \leq \mathrm{MDA}(E_1, \mathcal{S}) + \mathrm{MDA}(E_2, \mathcal{S}), \tag{2.9}$$

for any schedule $\mathcal{S}$.

For the local analysis on one ECU, Günzel et al. [14] estimate lower bounds for the read-events and upper bounds for the write-events of a job using implicit communication. They do this by simulating (scheduling) two different job collections of the underlying task set for two hyperperiods. A job collection jc is the set of all jobs, that are scheduled by a single schedule $\mathcal{S}$.

They introduce two special job collections, namely jc_{min} and jc_{max}. In the job collection jc_{min} all jobs only require the best case execution time B_τ of their underlying task. Analog, in the job collection jc_{max} all jobs always require the worst case execution time C_τ of their underlying task.

A simulation with the job collection jc_{min}, returns the lower bound for the read- and write-events of every job (e.g. the earliest point in time any job J of τ with $c_J \in [B_\tau, C_\tau]$ could have a read- or write-event). The second simulation with jc_{max} is used to get the upper bounds of the read- and write-events of all jobs (e.g. the latest point in time any job J of τ with $c_J \in [B_\tau, C_\tau]$ could have a read- or write-event).

These lower and upper bounds are then used in combination with an abstract integer representation. The abstract integer representation $\vec{I}_i^E = (i_0, ..., i_{|E|+1})$ is a tuple of $|E|+2$ natural numbers, used as an alternative notation for immediate forward augmented job chains $(z, J_1, ..., J_{|E|}, z')$ on a job collection jc.

$$\begin{aligned} z &= E(1)(i_0, jc) \\ J_1 &= E(1)(i_1, jc) \\ &\vdots \\ J_j &= E(j)(i_j, jc) \\ &\vdots \\ J_{|E|} &= E(|E|)(i_{|E|}, jc) \\ z' &= E(|E|)(i_{|E|+1}, jc) \end{aligned}$$

The construction of the abstract integer representation is then based on the previously determined lower and upper bound for the read- and write-events. At the end, the length of the abstract integer representation can be determined by the first value i_0 and the last value $i_{|E|+1}$:

$$l(\vec{I}_i^E) = \text{we}(E(|E|)(i_{|E|+1}, jc_{max})) - \text{re}(E(1)(i_0, jc_{min}))$$

Günzel et al. prove in [14], that the maximum length $l(\vec{I}_i^E)$ of the abstract integer representation is an upper bound for the length of all immediate forward augmented job chains of E for any job collection jc with $\tau(i) \in jc \Rightarrow c_{\tau(i)} \in [B_\tau, C_\tau]$. This means that the maximum length of all abstract integer representations $\vec{I}_i^E$ that start within the first two hyperperiods of the simulation can be used as an upper bound on the MRT of a cause-effect chain E.

Additionally, Günzel et al. provide an analog method in [14], that can be used to construct abstract integer representations $\overleftarrow{I}_i^E$ describing an immediate backward augmented job chain. In a similar way, these abstract integer representations can be used to upper bound the MDA of a given cause-effect chain E.

All of the provided analyses from [14] are applicable for periodic task sets using implicit communication.

2.4.7 Bi 2022

Bi et al. present one analysis method in [4] for computing an upper bound on the MRDA of a cause-effect chain. For this purpose, they examined the time interval between the releases of two consecutive tasks in the chain. These tasks are called producer and consumer. Bi et al. provide a method to compute the upper bound for this time interval for any producer-consumer pair in the chain. Depending on which of the two has higher priority, a different formula is applied (see lines 8 to 14 in algorithm 2.3).

```
1:  function Bi(E)
2:      if |E| = 1 then
3:          return R_{E(1)}
4:      else
5:          MRDA ← R_{E(n)}
6:          for i = 1 to n − 1 do
7:              η_{i,i+1} ← gcd(T_{E(i)}, T_{E(i+1)})
8:              if π(E(i)) < π(E(i + 1)) then
9:                  MRDA ← MRDA +T_{E(i)} − η_{i,i+1}
10:             else
11:                 if mod(R_{E(i)}, η_{i,i+1}) = 0 then
12:                     MRDA ← MRDA +R_{E(i)} + T_{E(i)} − η_{i,i+1}
13:                 else
14:                     MRDA ← MRDA +R_{E(i)} + T_{E(i)} −  mod (R_{E(i)}, η_{i,i+1})
15:                 end if
16:             end if
17:         end for
18:     end if
19:     return MRDA
20: end function
```

Algorithm 2.3: Algorithm 1 from Bi et al. [4] for computing the MRDA of a local cause-effect chain

Additionally, Bi et al. apply the cutting theorem from Günzel et al. presented in [17] and [14]. Therefore, it is also possible to use their analysis method for interconnected cause-effect chains, with local cause-effect chains running on different ECUs.

The biggest benefit of this analysis method is the reduced time complexity when compared to the analysis methods by Günzel et al. in [17] and [14] while still achieving comparably close end-to-end latencies. Bi et al. showed, that their approach for the local analysis has a time complexity of $O(n \log T^{max})$, where n is the number of tasks in the analyzed chain and T^{max} is the maximum period of all tasks in the chain. A single iteration of the algorithm has a time complexity of $O(\log T^{max})$ and is based on the complexity of computing the greatest common divisor using Euclideans algorithm. Since this step is repeated for every producer-consumer combination, Euclideans algorithm is called exactly n times by the analysis method.
The analysis can be applied to cause-effect chains with periodic task sets using implicit communication on GALS systems.

2.4.8 Gohari 2022

In [10] Gohari et al. describe an analysis for cause-effect chains with periodic tasks using implicit communication. The analysis differs from the other presented analysis methods, as it assumes a non-preemptive job-level fixed-priority scheduling. Non-preemptive scheduling implies a run-to-completion semantic, where a job cannot be preempted by any other (higher priority) job until it is finished with its execution. The analysis can therefore lead to different results, compared to the other covered analysis methods.
Moreover, the analysis method is also able to incorporate a release jitter and an execution time variation for a task. The release jitter is a possible delay of the task's release. Execution time variation means, that a job's execution time can vary. It is included in the common model of this thesis, as the execution time of a job lies between the BCET and the WCET of its task. The release jitter is not regarded. However, the analysis method can still be applied by setting the release jitter of every task to 0. Since the analysis is applicable for job-level priorities, it can also be used with task-level priorities, where each job of a task is assigned the same priority.
Gohari et al. perform multiple steps to calculate the upper bound on the MRDA. An overview can be seen in figure 3 of [10]. At first, a data propagation graph is created, where each vertex in the graph represents a task and a directed edge between two vertices represents a data propagation between those two tasks. The directed edge starts at the producing task and ends at the consuming tasks, that reads the data from the producer. This graph is then used together with the set of cause-effect chains (called task chains) to calculate an observation window, which is the time window relevant for their analysis.
The set of all jobs that are released within this time window is created and their start and finishing times are determined. This is done with the help of the schedule-abstraction graph from Ranjha et al. [28], which returns bounds for the start and finishing intervals of each job.

With these intervals Gohari et al. analyze the data dependencies between the jobs, meaning they are searching for all potential source jobs $J_o \in E(1)$ that could produce data for a sink job J_i from the task $E(|E|)$ within the observation window. This way, all potential data flows starting at $E(1)$ and ending at $E(|E|)$ are constructed. To reduce the number of possible paths, pruning rules are additionally applied. When multiple source jobs are found for a single sink job, only the source job with the earliest release is relevant for calculating the MRDA. Therefore, all of the paths starting at another source job can be discarded during analysis.
At the end, the MRDA is determined by the maximum time interval between the latest finishing time of a job of $E(|E|)$ to the earliest start time of a corresponding source job from $E(1)$.
Gohari et al. also note, that their analysis can be adapted for preemptive task models. In case there exists an algorithm that computes the starting and finishing intervals of preemptive jobs, only slight modifications are necessary. Two possible algorithms from Guan et al. [13] and Redell et al. [29] for calculating these intervals are mentioned, but an implementation is not pursued by Gohari et al. in [10].

2.4.9 Günzel 2023 (Heterogeneous)

In [19] Günzel et al. examine end-to-end latencies for heterogeneous systems. In such systems, both the communication policy and the release patterns of a task set are allowed to be heterogeneous. This means, that a task set can simultaneously contain periodic and sporadic tasks, as well as tasks, that are communicating via implicit communication and tasks communicating via LET communication. Analysis methods, that are capable of analyzing heterogeneous systems are also able to analyze homogeneous systems, when only a single release or communication policy is used.
Günzel et al. provide three different analysis methods for heterogeneous systems. They rely on the cutting theorem (2.4.1) presented in [17] and [14], which allows to cut the cause-effect chains into smaller sections that can be analyzed separately. Since these smaller sections are the independent from each other, they can be analyzed using multiple different analysis methods. Depending on their communication policy and their release pattern, it is then possible to find an analysis method specialized for analyzing these smaller cause-effect chains.

The three different analysis methods presented by Günzel et al. [19] are explained in more detail in the following:

1. **Pessimistic Analysis (Baseline)**
   The pessimistic analysis is a straight forward approach to the problem and combines the analysis methods from Davare et al. [6] and Hamann et al. [20]. For each task of the chain, either its maximum inter-arrival time plus its worst case response time (for tasks using implicit communication) or its maximum inter-arrival time plus its deadline (for tasks using LET communication) is calculated and summed up over all tasks in the chain. This analysis method is then later used in their paper as a reference baseline for the other two approaches.

2. **Analysis based on homogeneous bounds**
   The analysis based on homogeneous bounds first cuts the initial cause-effect chains into several smaller cause-effect chains. A cut is performed when the communication policy or the release pattern of a task changes, compared to the previous task in the chain. This way, a number of smaller chains is created, where every chain is homogeneous and can therefore be analyzed with an analysis method designed for homogeneous systems.

   There are four different homogeneous combinations possible, that are analyzed according to the methods listed in table 2.2.

	Periodic release	**Sporadic release**
Implicit comm.	Algorithm 2 from [19]	Dürr et al. [7]
LET comm.	Algorithm 1 from [19]	Hamann et al. [20]

Table 2.2: Analysis methods used for different combinations of release pattern and communication mechanism in the analysis with homogeneous bounds (similar to table 1 from Günzel et al. [19])

function GÜNZEL_PERIODIC_LET(E)
  $lengths \leftarrow \emptyset$
  for $m = 1$ **to** ∞ **do**
    $z \leftarrow \phi_{E(1)} + (m-1) \cdot T_{E(1)}$
    $r_{J_1} \leftarrow z + T_{E(1)}$
    if $r_{J_1} + R_{E(1)} < \Phi(E)$ **then**
      continue
    end if
    if $z > \Phi(E) + H(E) + R(E)$ **then**
      break
    end if
    for $i = 1$ **to** $|E| - 1$ **do**
      $r_{J_{i+1}} \leftarrow \phi_{E(i+1)} + \left\lceil \frac{r_{J_i} + D_{E(i)} - \phi_{E(i+1)}}{T_{E(i+1)}} \right\rceil \cdot T_{E(i+1)}$
    end for
    $z' \leftarrow r_{J_E} + D_{E(|E|)}$
    $lengths \leftarrow lengths \cup (z' - z)$
  end for
  return $max(lengths)$
end function

Algorithm 2.4: Algorithm 1 from Günzel et al. [19] for computing the MRT of a periodic cause-effect chain using LET communication

function GÜNZEL_PERIODIC_IMPLICIT(E)
  $lengths \leftarrow \emptyset$
  for $m = 1$ **to** ∞ **do**
    $z \leftarrow \phi_{E(1)} + (m-1) \cdot T_{E(1)}$
    $r_{J_1} \leftarrow z + T_{E(1)}$
    if $r_{J_1} + R_{E(1)} < \Phi(E)$ **then**
      continue
    end if
    if $z > \Phi(E) + H(E) + R(E)$ **then**
      break
    end if
    for $i = 1$ **to** $|E| - 1$ **do**
      if $\pi(E(i)) > \pi(E(i+1))$ **then**
        $r_{J_{i+1}} \leftarrow \phi_{E(i+1)} + \left\lceil \frac{r_{J_i} - \phi_{E(i+1)}}{T_{E(i+1)}} \right\rceil \cdot T_{E(i+1)}$
      else
        $r_{J_{i+1}} \leftarrow \phi_{E(i+1)} + \left\lceil \frac{r_{J_i} + R_{E(i)} - \phi_{E(i+1)}}{T_{E(i+1)}} \right\rceil \cdot T_{E(i+1)}$
      end if
    end for
    $z' \leftarrow r_{J_E} + R_{E(|E|)}$
    $lengths \leftarrow lengths \cup (z' - z)$
  end for
  return $max(lengths)$
end function

Algorithm 2.5: Algorithm 2 from Günzel et al. [19] for computing the MRT of a periodic cause-effect chain using implicit communication

3. Improved Analysis

In the improved Analysis of [19], Günzel et al. try to reduce some of the downsides of the analysis with homogeneous bounds. Especially, they reduce the pessimism of the previous analysis when the communication policy changes between two consecutive tasks in the chain. To avoid this over approximation, the improved analysis only performs a cut (i.e. only applies the cutting theorem) in case the release pattern of two consecutive task in the chain changes. Otherwise, the cut is prevented, leading to cause-effect chains that are either completely periodic or sporadic.

The completely periodic cause-effect chains are analyzed similar to algorithm 1 and 2 from [19]. The only difference is the case, when a task $E(i+1)$ using LET communication follows a task $E(i)$ using implicit communication. In this case the WCRT of $E(i)$ is directly added to the sum, without comparing the priorities of the two consecutive tasks (see algorithm 3 from [19]).

For the completely sporadic cause-effect chains the analysis is done using the following formulas from [19]:

$$\mathrm{MRT}(E) \leq \sum_{i=1}^{|E|} (T_{E(i)}^{max} + CX_i) \tag{2.10}$$

$$CX_i = \begin{cases} D_{E(i)} & \text{if } comm(E(i)) = \text{LET} \\ max(R_{E(i)} - T_{E(i+1)}^{max}, 0) & \text{if } E(i+1) \text{ exists, } \pi(E(i)) > \pi(E(i+1)) \wedge \\ & comm(E(i+1)) = \text{impl} = comm(E(i)) \\ R_{E(i)} & \text{otherwise} \end{cases} \tag{2.11}$$

At the end, Günzel et al. [19] compare their three approaches against each other. The evaluation indicates, that the analysis with homogeneous bounds and the improved analysis return tighter bounds for the end-to-end latencies than the pessimistic baseline approach. Additionally, the improved analysis returns lower latencies than the analysis with homogeneous bounds, especially if the number of sporadic tasks in the cause-effect chain is less than or equal to 50%.

Günzel et al. point out, that the over approximation of the end-to-end latencies is reduced with less applications of the cutting theorem. This coincides with the results of the evaluation, since the improved analysis performs the fewest cuts and returns the tightest bound, while the baseline analysis performed the most cuts, leading to larger latencies.

2.4.10 Günzel 2023 (Equivalence)

In [18] Günzel et al. show the equivalence of the two evaluation metrics MRT and MDA. They do this by introducing the concept of p-partitioned job-chains. A p-partitioned job-chain pc_m^p is a job chain, which consist of the $(m+1)$-th immediate forward $(\tilde{J}_p, ..., \tilde{J_{|E|}})$ and m-th immediate backward job chain $(J_1, ..., J_p)$. The p-th task of the immediate backward job chain is the same as the first task of the immediate forward job chain, connecting the two chains. Furthermore, $\tilde{J}_p$ is the first job of $E(p)$ after J_p. If J_p is the i-th job of $E(p)$, then $\tilde{J}_p$ is the $(i+1)$-th job of $E(p)$.
Günzel et al. defined the p-partitioned job-chains in such a way, that a 1-partitioned job chain is equivalent to an immediate forward augmented job chain. This is important, since immediate forward augmented job chains can be used to bound the MRT of the underlying cause-effect chain. Additionally, an $|E|$-partitioned job-chain is equivalent to an immediate backward augmented job chain. Immediate backward augmented job chains can be used to upper bound the MDA of the underlying cause-effect chain.
Moreover, Günzel et al. prove, that the maximum length of a p-partitioned job-chain $l(pc_m^p) = \text{we}(\tilde{J}_{|E|}) - \text{re}(J_1)$ is independent of the value p, meaning that any p in the interval $[1, p]$ can be used to calculate the same upper bound. Therefore, the maximum length of a 1-partitioned job chain is always the same as the length of the corresponding p-partitioned job-chain. This directly leads to the equivalence of MRT and MDA, summarized in the equivalence theorem 2.4.2.

2.4.2 Theorem (Equivalence Theorem [18]). *The maximum reaction time and the maximum data age are equivalent, i.e.,*

$$\text{MRT}(E) = \sup\{l(pc_m^p) | m \geq F_p\} = \text{MDA}(E) \tag{2.12}$$

for all $p \in \{1, ..., |E|\}$. F_p denotes the first job of $E(p)$ after the warm-up period.

The equivalence of MRT and MDA implies, that any analysis for one metric can be directly used for the other metric as well. Analysis methods that compute either of both metrics could be directly compared in the evaluation framework, as the two metrics are equivalent.

The analysis presented in [18] is focusing on analyzing cause-effect chains with LET communication. Günzel et al. construct the p-partitioned job-chain, where p is the number of the task with the largest period in the cause-effect chain E. This way the number of job chains that need to be constructed during the analysis is reduced. Afterwards, the MRT and the MDA can be calculated using the Equivalence Theorem 2.4.2. At the end, the MRRT can be computed from the MRT by subtracting the period of the first task in the chain. Similarly, the MRDA is computed by subtracting the period of the last task from the MDA.

For the release patterns of the task in the underlying task set both periodic and sporadic are supported by this analysis method.

2.4.11 Extensions of Günzel 2023 (Equivalence)

While working on this master's thesis, the idea came up to extend the analysis methods presented by Günzel et al. in [18]. In their paper Günzel et al. mention that their analysis method is applicable for cause-effect chains with implicit communication. However, the focus of [18] is to apply the analysis in combination with cause-effect chains that use LET communication (section 8).
For the analysis of Günzel et al. it is necessary to compute immediate forward and immediate backward augmented job chains in order to construct the p-partitioned job-chains. To do this, it is important to know the read and write events of the individual jobs. This allows to determine, which jobs are communicating and therefore part of the job chain.
In [18], Günzel et al. examine cause-effect chains with LET communication. The read and write events for jobs of these chains can directly be calculated, as the read event is equivalent to the release of the job and the write event is equivalent to the deadline of the job (in this case, the release of the consecutive job from the same task). Determining the read and write events for jobs using implicit communication is more challenging, as they do not correspond to the jobs release and deadline. This section extends the presented analysis method from [18] to cause-effect chains with implicit communication in two different ways.

1. **Utilizing the schedule** (similar to [17] and [14]): The underlying task sets of the cause-effect chains are scheduled. This way the precise starting and finishing times of a job can be determined, which are equivalent to the read and write event time points in the implicit communication model. The resulting analysis is therefore a combination of approaches from [17], [14] and the approach from [18].

   At first, a schedule for the cause-effect chains' task set is constructed. The value p is determined by finding the task $E(p)$ with the largest period in the chain. The immediate forward and the immediate backward augmented job chains are constructed with the help of the schedule and saved as their abstract integer representations, similar to [17], [14]. With this abstract integer representation, the length of the partitioned job chain is calculated. The MRT and MDA are then determined by the maximum length of any constructed p-partitioned job chain.

2. **Utilizing the response times**: With the response times of the tasks in the underlying task set, the read and write events can be approximated. The read event is under-approximated as the release of the job and the write event is over-approximated by the response time of the corresponding task. The analysis is then similar to the analysis with the schedule, presented above.

First, the response times for the cause-effect chains' task set are calculated. The value p is determined by finding the task $E(p)$ with the largest period in the chain. The immediate forward and the immediate backward augmented job chains are constructed with the help of the response times and saved as their abstract integer representations. With this abstract integer representation, the length of the partitioned job chain is calculated. The MRT and MDA are then determined by the maximum length of any constructed p-partitioned job chain.

The schedule-based analysis is more precise than the analysis based on the task response times, as it computes a tighter bound on the end-to-end latency of a given cause-effect chain. That is because of the approximation of the read and write events with the response time and the release of a job. However, calculating a schedule and constructing job chains on this schedule is more complex and computationally intensive than calculating response times for a task set. The analysis based on the response times is therefore much faster compared to the schedule-based analysis.

2.5 Benchmarks

In order to compare the performances of the presented end-to-end analyses methods, they have to be applied to different task sets and cause-effect chains. These can either come from an example from a real world application scenario or be synthetically generated by an algorithm. For an automated evaluation framework the use of synthetically generated task sets and cause-effect chains is preferable, since the underlying algorithms can generate an arbitrary amount of different combinations for the evaluation.

There exist two algorithms commonly used for generating task sets. The first is the automotive benchmark by Kramer et al. [24], which is able to create synthetic task sets that meet industry standards of the automotive industries. The second algorithm widely used to generate task sets is the UUnifast algorithm from Bini et al. [5]. This algorithm does not assume any particular use case or industry standard for the task set generation, as it only uses the uniform distribution. It is therefore also called uniform task set generation.

Composing cause-effect chains from the tasks in the task set is usually done using the automotive benchmark by Kramer et al. [24] again. The resulting cause-effect chains are accurate examples for cause-effect chains of systems in the automotive domain. It is currently the only algorithm commonly used in the literature to create cause-effect chains.

An alternative approach to generate cause-effect chains is to uniformly draw a number of tasks from the previously generated task set without replacement. The resulting cause-effect chains are not as close to real-world cause-effect chains compared to the automotive benchmark, but can nevertheless be used to compare the performances of different end-to-end analysis methods.

2.5.1 Automotive Benchmark

In their paper [24] Kramer et al. give an insight on the characteristics of realistic task sets and cause-effect chains from the automotive domain. They describe how such task sets and cause-effect chains can be constructed.
Kramer et al. use a different system model with so called runnables, which are combined to tasks. In their model a task can consist of multiple runnables, where all runnables have the same period. However, runnables are able to release jobs that are able to communicate and therefore similar to tasks from this thesis' system model. To avoid any ambiguity, each runnable is mapped to an individual task and each task only consist of one runnable.

Task Set Generation

For generating a single task with the automotive benchmark [24], the following steps are performed:

1. At first, the period of the task is randomly drawn according to the adjusted probabilities in table 2.3. The probabilities from [24] had to be adjusted, as there are no angle-synchronous tasks in the system model of this thesis. As a result, the remaining probabilities were divided by their sum, so that the sum of all probabilities is equal to 100% again.

Period	Probability (Original)	Probability (Adjusted)
1 ms	3%	3.53%
2 ms	2%	2.35%
5 ms	2%	2.35%
10 ms	25%	29.41%
20 ms	25%	29.41%
50 ms	3%	3.53%
100 ms	20%	23.53%
200 ms	1%	1.18%
1000 ms	4%	4.71%
angle-synchronous	15%	0%

Table 2.3: Distribution of task periods according to Kramer et al. [24], Table III; The adjusted probabilities are computed by setting the probability for angle-synchronous task to 0 and normalizing the remaining values.

Period	Average Execution Times [μs]		
	Min	**Avg**	**Max**
1 ms	0.34	5.00	30.11
2 ms	0.32	4.20	40.69
5 ms	0.36	11.04	83.38
10 ms	0.21	10.09	309.87
20 ms	0.25	8.74	291.42
50 ms	0.29	17.56	92.98
100 ms	0.21	10.53	420.43
200 ms	0.22	2.56	21.95
1000 ms	0.37	0.43	0.46
angle-synchr.	0.45	6.52	88.58
Interrupts	0.18	5.42	12.59

Table 2.4: Average execution times of tasks (runnables) with different periods (from Kramer et al. [24], Table IV)

Period	Best		Worst	
	f_{min}	f_{max}	f_{min}	f_{max}
1 ms	0.19	0.92	1.30	29.11
2 ms	0.12	0.89	1.54	19.04
5 ms	0.17	0.94	1.13	18.44
10 ms	0.05	0.99	1.06	30.03
20 ms	0.11	0.98	1.06	15.61
50 ms	0.32	0.95	1.13	7.76
100 ms	0.09	0.99	1.02	8.88
200 ms	0.45	0.98	1.03	4.90
1000 ms	0.68	0.80	1.84	4.75
angle-synchr.	0.13	0.92	1.20	28.17
Interrupts	0.12	0.94	1.15	4.54

Table 2.5: Scaling factors to compute the BCET and the WCET from the ACET (from Kramer et al. [24], Table V)

2. The average-case execution time (ACET) of the task is approximated with a Weibull distribution according to the average execution times in table 2.4. For each task the generated value of the ACET has to lie within the minimum and maximum ACET value (with respect to its period), otherwise it is discarded. The average ACET over all generated ACETs for one period should then be equivalent to the value in the column "Avg ACET" from table 2.4.

3. To compute the WCET of the task, the previously determined ACET is multiplied with a random factor drawn between f_{min} and f_{max} (see table 2.5). Similarly, the BCET can be computed from the ACET.

Number of Unique Periods	Probability
1	70%
2	20%
3	10%

Table 2.6: Number of unique periods (involved activation patterns) in a task chain and their probabilities (from Kramer et al. [24], Table VI)

Number of Tasks	Probability
2	30%
3	40%
4	20%
5	10%

Table 2.7: Number of tasks (runnables) per unique period and their probabilities (from Kramer et al. [24], Table VII)

These steps are repeated to create a pool of tasks. Afterwards, the subset-sum approximation algorithm can be applied to select a number of tasks from the pool. With this method it is possible to achieve any desired target utilization of the task set.

At the end, it should be checked whether the generated task set is also schedulable. For this purpose the time-demand analysis is used to determine, if there are any tasks in the task set that could miss their deadline. In that case the task set is discarded.

Cause-Effect Chain Generation

Additionally, the automotive benchmark can be used to generate cause-effect chains, necessary to perform an analysis. The parameters for the generation process by Kramer et al. [24] are summarized in tables 2.6 and 2.7.

At first, a number of unique periods for the cause-effect chain is drawn according to the probabilities in table 2.6. After that, the according number of unique periods are drawn from the underlying task set without replacement to ensure that the periods are unique. For each of the drawn periods a number between 2 and 5 tasks are drawn from the set of tasks (without replacement, because a task can only occur once in a cause-effect chain) according to the probabilities in table 2.7.

The cause-effect chains generated with this method have a minimum length of 2, when choosing 2 different tasks for only 1 unique period, and a maximum length of 15, when choosing 5 different tasks for every of the 3 drawn periods.

2.5.2 Uniform Benchmark

The uniform task set generation is based on the UUniFast algorithm from Bini et al. [5]. It is used to uniformly draw $n \in \mathbb{N}$ utilization values from the interval $(0, 1]$ with the constraint that $\sum_{i=1}^{n} U_i = U_{target}$. The target utilization U_{target} and the value n for the number of tasks in the task set can be specified by the user. In a second step the period T_i for each task is determined. These are also uniformly drawn from an interval $[1, T_{max}]$, where T_{max} is a user specified upper bound for the length of the task periods. In the last

step the WCET C_i of a task is calculated by multiplying its utilization with its period ($C_i = U_i \cdot T_i$).

Some analyses in the literature are only applicable, if the hyperperiod of the resulting task set is sufficiently small. During the described generation process n periods are uniformly drawn from the interval $[1, T_{max}]$. The hyperperiods of such task sets can therefore become enormously large, even for relatively small values for n and T_{max}. One method to avoid this problem is presented by Günzel et al. in [14]. They draw the periods of the tasks from the interval $[1, 2000]$ (e.g. setting the value T_{max} to 2000) using a log-uniform distribution and round the drawn periods down to the next smallest period from the automotive benchmark (periods in table 2.3). The resulting task set is semi-harmonic with a hyperperiod of at most 1000.
After a task set has been generated, the uniform benchmark can also be used to create a cause-effect chain. The algorithm selects a specified number of tasks from the task set without replacement and concatenates them to a cause-effect chain. Since this process is straight forward, the generation of cause-effect chains is less time-consuming than the generation process of the automotive benchmark. On the other hand, the cause-effect chains generated by the uniform benchmark do not resemble any real-world use-case. An evaluation on such cause-effect chains is therefore less valuable, as it does not necessarily show the real-world performance of the analysis method.

Chapter 3

Implementation of the Framework

This chapter covers the implementation of the evaluation framework and explains the decisions, that were made during that process. It also gives an overview on the source code of the framework, which is publicly released.

For a better understanding of the implemented analysis methods, all of them are listed in this chapter and mapped to the corresponding theoretical background.

At last, some additional features that are implemented in the framework are elaborated.

3.1 Choice of Programming Language

For implementing the framework, Python was chosen as the main programming language. There are multiple reasons why Python is better suited for the use case of the evaluation framework compared to other programming languages.

A major advantage of Python is its ease of use, especially when compared to languages like C++. This makes the framework itself easier to work on, leading to better extensibility in the future. Additionally, Python code is easier to understand than more complex C++ code, lowering the entry barrier for other researchers who want to use the framework.

Another advantage is that most of the analysis methods mentioned in the theoretical background are already implemented in Python. Therefore, it is often unnecessary to completely reimplement these methods. Instead, one only needs to adjust the underlying models of tasks, task sets, and cause-effect chains to fit the evaluation framework. Choosing Python as the main programming language thus reduces the programming effort.

A disadvantage of Python compared to C++ is its lower performance. Python programs are typically slower because Python is an interpreted language with dynamic typing, which introduces additional overhead during execution. On the other hand, C++ is a compiled language with static typing, allowing for more efficient, machine-level code execution and compiler optimizations, resulting in faster performance.

Since fast performance is not as crucial for the framework compared to its extensibility and ease of use, Python is a reasonable choice for the implementation.

3.2 GUI Frameworks

The evaluation framework should be easily usable via a graphical user interface (GUI). For creating such user interfaces in Python, dedicated GUI frameworks are typically used. This section will give an overview of some of the most popular Python GUI frameworks currently available and determines how well each GUI framework is suited for the use-case of the evaluation framework.

PyQt

PyQt is a powerful GUI framework for Python, widely used to create user interfaces due to its extensive features and capability to build very complex user interfaces.
However, the main disadvantage of this framework is that it is too complex for the use case of a simple GUI. The programming effort required for this graphical user interface framework is higher, leading to reduced extensibility. Adding more options or analysis methods in an evaluation framework using PyQt would therefore be more complex and time-consuming.

Tkinter

Tkinter is the standard GUI framework included with Python, making it highly accessible and easy to use. It is ideal for creating simple and straightforward user interfaces with minimal effort. While it lacks the advanced features and modern aesthetics of PyQt, Tkinter is well suited for small projects and basic applications.
One of the downsides of Tkinter is, that it still requires a lot of code, even for simple user interfaces. To reduce the code necessary for creating the graphical user interface, Tkinter can be used in combination with PySimpleGUI.

PySimpleGUI

PySimpleGUI is a very simple graphical user interface framework for Python. Internally, PySimpleGUI uses another GUI framework (usually Tkinter), so the resulting GUI will look very similar to GUIs created with the underlying framework.

The main difference between PySimpleGUI and frameworks like PyQt and Tkinter is that very little code is necessary to achieve similar-looking graphical user interfaces. This makes it much easier to create or adjust/extend the GUI. Consequently, this reduces the effort needed for any future developments on the framework and leads to easier extensibility.

A disadvantage of PySimpleGUI compared to more powerful GUI frameworks is that the resulting GUI is not as customizable. Very specific changes to the look or behavior of the graphical user interface are not possible, as these are not supported by PySimpleGUI. However, since the main purpose of this GUI is to create a simple, straightforward input mask for the evaluation framework, this disadvantage is not as significant.

After comparing the different options, PySimpleGUI in combination with Tkinter was selected for implementing the graphical user interface. This choice significantly reduces the effort required for creating the GUI, thus enhancing the framework's extensibility for future changes and modifications.

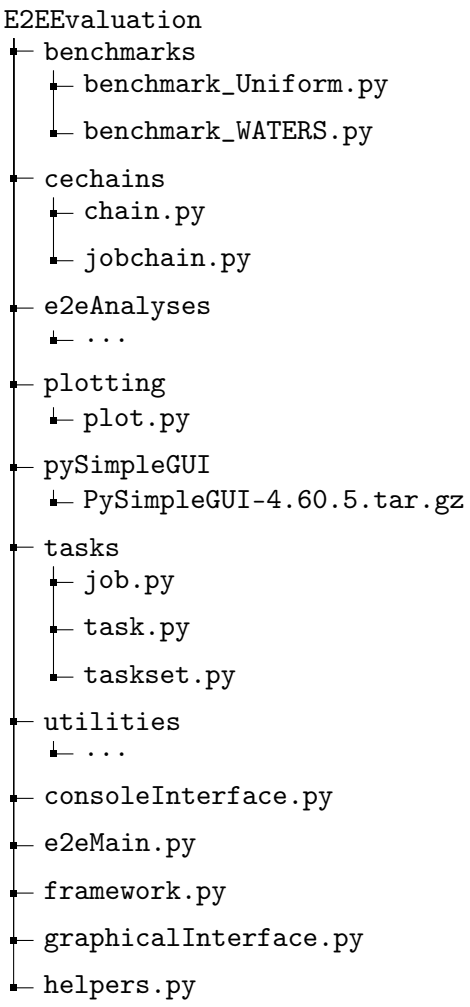
The last version of PySimpleGUI released as free software under the GNU Lesser General Public License is version 4.60.5, published in May 2023. Following versions, starting with PySimpleGUI 5, are released under a proprietary license. Since the idea of this framework is to allow other researchers to easily integrate and compare their analysis with other analysis methods, requiring a proprietary license would contradict this objective. Therefore, the slightly older version 4.60.5 of PySimpleGUI was chosen as the GUI framework for the evaluation framework.

3.3 Layout of the Framework

This section gives an overview about the implementation of the framework and how it is organized. Therefore, an overview about the file structure, an explanation of the implemented model and a list of all implemented analysis methods are presented.

3.3.1 File Structure

The framework consist of multiple python files, organized in different directories. The root directory contains the main file `e2eMain.py`, as well as the Interfaces `consoleInterface.py` and `graphicalInterface.py`. The file `framework.py` contains the main logic of the evaluation framework. `helpers.py` offers some functions for easier file access.



There are seven different subdirectories in the root folder, used for organizing the python source code. Most important is the `e2eAnalyses` directory, which includes all the end-to-end analysis functions. The folders `cechains` and `tasks` are used for the implemented system model, which is described in more detail in the next section. In the directory `benchmarks` are the python files for the different benchmarks for generating task sets and cause-effect chains. The plotting directory contains the python file, relevant for creating the box plot diagrams at the end of the evaluation. The python package for the graphical user interface is located in the directory `pySimpleGUI`. Lastly, the `utilities` directory is used for extra source code files of the end-to-end analyses. This includes for example an event simulator, which is used by some of the analyses for creating a schedule for a task set.

3.3.2 Implemented Model

This section covers the implementation of the system model from section 2.2. The focus of the implementation was to keep the system model as simple as possible, in order to increase the versatility of the framework. A simple implementation allows for faster and easier modifications in the future, increasing the extensibility of the evaluation framework.

Tasks

A task is implemented as a python object with all necessary information about the task stored as attributes of the task object. This is a very basic implementation to reduce the complexity of the system. The necessary information about the task are generated during its creation and stored in the object when it gets initialized. Later during the analysis, each analysis method can access the information of the task it needs to perform the analysis.

Task Sets

The task set is implemented as a list of task, where the position of each task within the list denotes its priority. Meaning, a task with a lower index in the task set list has a higher priority than a task with a higher index.
Additionally, the task set class includes some helper functions to retrieve information about the task set:

- `taskset.prio(task)`: Returns the priority of the task, based on the index of the given task set list.
- `taskset.higher_prio(task1, task2)`: Returns true, if `task1` has a higher priority than `task2`.
- `taskset.utilization()`: Returns the utilization of the task set based on the WCET and minimum inter arrival time of each task.
- `taskset.check_feature(feature)`: Checks if all tasks of the task set have the same feature value for a given feature (e.g. `taskset.check_feature('communication_policy')` returns `'implicit'` if all tasks use implicit communication and `'mixed'` if there are some tasks in the task set using LET communication and some using implicit communication). The available features are defined in the task.
- `taskset.compute_wcrts()`: Computes the WCRTs of all tasks in the task set with time demand analysis (TDA) and stores the result in a dictionary with the task object as key and the corresponding WCRT as its value.

- `taskset.hyperperiod()`: Computes the least common multiple of the periods of all tasks in the task set and returns it.
- `taskset.max_phase()`: Returns the largest phase of all tasks in the task set.
- `taskset.sort_dm()`: Sorts tasks by their deadline and updates their priority.
- `taskset.rate_monotonic_scheduling()`: Sorts tasks according to their period and updates their priority.

Cause-Effect Chains

A cause-effect chain is modeled as an object, inheriting from a task set. It is therefore also a list of tasks, where each task in this list is occurring according to its position in the cause-effect chain.
Additionally, the cause-effect chain object holds a reference to the underlying task set, called `base_ts`. This allows to access the underlying task set later during the analysis in order to retrieve useful information, such as the WCRTs or the hyperperiod of the task set.
Interconnected cause-effect chains are only analyzed by some of the end-to-end latency analysis. They are simply modeled as a list of local cause-effect chains, presented above.

3.3.3 Analysis Methods

This sections includes a list of all analysis methods, which are currently implemented in the framework. Analysis methods that belong to the same paper, are usually grouped into the same python file. All of these python files are located in the folder `e2eAnalyses` in the source code, published with this thesis.

Becker 2016/2017

For the paper [1] by Becker et al. there are two different implementations included in the framework. The first implementation is located in the file Becker2017.py and includes four analysis methods, which were presented in the paper. As these analysis methods overlap with the analysis methods from [2] by Becker et al., there is no separate implementation for the paper [2] and its analysis methods.
The implementation of Becker2017.py is based on an implementation in the programming language Java provided by Matthias Becker, who is the first author of the paper [1]. The implementation is using the object oriented paradigm of Java and will create many objects during the analysis. This will lead to very long analysis time, especially when there are longer chains that need to be analyzed.

Becker2017.py

- `becker17_NO_INFORMATION(local_chain)`
- `becker17_RESPONSE_TIMES(local_chain)`
- `becker17_SCHED_TRACE(local_chain)`
- `becker17_LET(local_chain)`

BeckerFast.py

- `beckerFast_NO_INFORMATION(local_chain)`
- `beckerFast_RESPONSE_TIMES(local_chain)`
- `beckerFast_SCHED_TRACE(local_chain)`
- `beckerFast_LET(local_chain)`

An alternative to the implementation in Becker2017.py is the implementation in BeckerFast.py. This implementation is based on an implementation that was published with the source code of [14] on GitHub [15]. The benefit of this implementation over the implementation in Becker2017.py is, that it does not create as many objects during runtime. Therefore, the results of both analyses is exactly the same, but achieved much faster with the implementations in BeckerFast.py.
Originally the BeckerFast.py file only consisted of the `beckerFast_RESPONSE_TIMES` analysis method. The other three analysis methods of [1] were added to BeckerFast.py similar to the already existing implementation during this work.

Davare 2007

The implementation of the analysis method by Davare et al. in [6] is done in Davare2007.py. It is a straightforward implementation of equation 2.1 for computing the MRT of a cause-effect chain.

Davare2007.py

- `davare07(local_chain)`
- `davare07_inter(*local_chain)`

Furthermore, the method `davare07_inter` is implemented, which is an extension of the original analysis method to also analyze interconnected cause-effect chains. This way the analysis method by Davare et al. can also be used as reference baseline method for the results of other analysis methods capable of analyzing interconnected cause-effect chains.

Kloda 2018

Kloda2018.py

- `kloda18(local_chain)`

The implementation of [22] is done in the method `kloda18` in the file Kloda2018.py. It is based on the implementation of Günzel et al. [16] and was adjusted to fit the underlying system model of the evaluation framework. The implementation is similar to algorithms 2.1 and 2.1.

Dürr 2019

For the implementation of Dürr 2019 from [7] by Dürr et al., the implementation from [16] is used as a basis. There exists the two methods `duerr19_mrt` and `duerr19_mrda` for that can be used to get an upper bound for the two different end-to-end latency metrics. Furthermore, it is possible to use both methods with local and interconnected cause-effect chains.

Duerr2019.py

- `duerr19_mrt(*local_chain)`
- `duerr19_mrda(*local_chain)`

Hamann 2017

The analysis of Hamann et al. [20] is implemented with a single function called `hamann17` in Hamann2017.py. Again, the implementation is very straight forward according to formula 2.2.

Hamann2017.py

- `hamann17(local_chain)`

Günzel 2023 inter

There are a variety of different functions available for the analysis methods from Günzel et al. [17] and Günzel et al. [14]. Since their analysis is capable of analyzing local and interconnected cause-effect chains, three of these functions are used to analyze local cause-effect chains and two are used for interconnected cause-effect chains, which use the local functions in combination with the cutting theorem 2.4.1. The results of `guenzel23_local_mrt`

and `guenzel23_local_mda` will always be the same, but are internally computed in two different ways. `guenzel23_local_mrt` computes the end-to-end latency upper bound by constructing the forward job chains, whereas `guenzel23_local_mda` constructs the backward job chains.

Guenzel2023_inter.py

- `guenzel23_local_mrt(local_chain)`
- `guenzel23_local_mda(local_chain)`
- `guenzel23_local_mrda(local_chain)`
- `guenzel23_inter_mrt(*local_chain)`
- `guenzel23_inter_mrda(*local_chain)`

Günzel 2023 mixed

In [19] Günzel et al. presented three different analysis methods for mixed cause-effect chains. Each of them is implemented in a single python method in the file Guenzel2023_mixed.py. The baseline approach is represented by the method `guenzel23_mix_pessimistic`, the analysis with homogeneous bounds by `guenzel23_mix` and lastly the improved analysis in `guenzel23_mix_improved`.

Guenzel2023_mixed.py

- `guenzel23_mix_pessimistic(local_chain)`
- `guenzel23_mix(local_chain)`
- `guenzel23_mix_improved(local_chain)`

Günzel 2023 equi

There are four different functions available for the Günzel 2023 equivalence analysis approach. The implementation is provided by Günzel et al., which includes the end-to-end analysis with the four metrics MDA, MRT, MRDA and MRRT. However, these functions can only be used to analyze the end-to-end latencies of cause-effect chains that only use LET as their communication policy.

Guenzel2023_equi.py

- `guenzel23_equi_mda(local_chain)`
- `guenzel23_equi_mrt(local_chain)`
- `guenzel23_equi_mrda(local_chain)`
- `guenzel23_equi_mrrt(local_chain)`

Günzel 2023 equi - Extensions

The two extensions of the analysis from Günzel et al. [18] presented in section 2.4.11 are implemented in two separate files. Both implementations are only applicable to local cause-effect chains and return the MRT and the MDA for the given chains.
The first implementation is located in the file Guenzel2023_equi_extension1.py and uses the exact schedule of the underlying task set for the analysis of a cause-effect chain, similar to the implementations in Guenzel2023_inter.py. This way, the exact read and write events of all jobs are known and can be used to precisely approximate an upper bound on the end-to-end latency. The result of this analysis is always equivalent to the result of `guenzel23_local_mrt` and `guenzel23_local_mda`.

Guenzel2023_equi_extension1.py

- `guenzel23_equi_impl_sched(local_chain)`

The second implementation is also based on the equivalence approach from Günzel et al. [18], but uses the response times of the tasks to approximate the implicit communication. This leads to less precise bounds of the communication compared to the first implicit implementation. On the other hand the analysis is performed much faster, as there is no need to generate and analyze the schedule of the underlying task set. Even without the schedule generation, the implementation based on the response times is still at least ten times faster than the implementation using the exact schedule of the task set. Depending on the use-case, it could therefore be preferable to use the analysis method with less precise results, but much faster evaluations.

Guenzel2023_equi_extension2.py

- `guenzel23_equi_impl_rt(local_chain)`

Bi 2022

For the analysis method presented in [4], Bi et al. provided the implementations for their analysis methods. This includes two analyses, one for local cause-effect chains (`bi22`) and one for interconnected cause-effect chains (`bi22_inter`). The analysis method for the interconnected cause-effect chains had to be adjusted to fit the implemented system model, as it does not include the communication task considered by Bi et al.

Bi2022.py

- `bi22(local_chain)`
- `bi22_inter(*local_chain)`

Gohari 2022

An implementation for the analysis of Gohari et al. [10] is given in the repository "Data-age Analysis for Multi-rate Task chains" [9]. The implementation is done in the programming language C++ and extends over multiple files. It was therefore decided to not re-implement the existing implementation in python, but keep the original implementation and integrate it into the framework via a system command.
To do this, the evaluation framework first clones the C++ code from the GitHub repository [9] and compiles it on the users machine. Then it exports the generated cause-effect chains into a YAML file with a special format, that can be read by the implementation from Gohari et al. The C++ binary is executed, reads the YAML file and performs the analysis. When it is finished, the output of the analysis is stored in a CSV file. This CSV file is then read again by the evaluation framework in order to use the results for the evaluation diagrams.
The step of exporting the cause-effect chains, analyzing the cause-effect chains and loading the results back into the evaluation framework is repeated for every underlying task set, since the analysis did not terminate in tests with more than one task set. The step could also be repeated for every cause-effect chain, but would increase the overhead even more, as there are usually more than one cause-effect chain per generated task set.
In the evaluation framework, the method `gohari22` is used to perform all of the steps mentioned above. Since the implementation is different because of the intermediate importing and exporting steps, the method directly takes all of the generated cause-effect chain. A parallelization over the number of cause-effect chains is therefore currently not possible with this analysis method.

Gohari2022.py

- `gohari22(local_chain)`

Furthermore, the analysis method is not always able to return a result for a given set of cause-effect chains. Because the underlying task sets are intended for preemptive scheduling, they may not be schedulable with a non-preemptive scheduling. As a result, the analysis method is not able to analyze all cause-effect chains that can be generated by the evaluation framework. To prevent the program from terminating, the corresponding exception is caught and the value -1 is saved as the result of the analysis, indicating an error.
It should also be noted, that the analysis of a set of cause-effect chains takes relatively long with this analysis method compared to the other analysis methods, even when they are not parallelized. Hence, an evaluation is only possible with a sufficiently small number of cause-effect chains and task sets.

3.3.4 Plotting

After all selected analysis methods are finished with computing an upper bound on the end-to-end latency of the given cause-effect chains, the results can be plotted. The framework allows to plot absolute and normalized box plots. The absolute box plots only use the results of the analysis itself.
For the normalized box plots, normalized values are generated based on the selected normalization analysis method (called baseline). The results are normalized as follows:

$$\text{latency}_{reduction} = \frac{\text{latency}_{base} - \text{latency}_{abs}}{\text{latency}_{base}}$$

The framework creates one (absolute/normalized) diagram for each selected analysis method and an additional (absolute/normalized) diagram that includes all selected analysis methods. If multiple normalization methods are selected, the framework will create plots for each selected normalization method.
Box plots over the normalized data are especially useful to see the latency reduction compared to the reference analysis method. Therefore they are widely used in the evaluations of many end-to-end analysis papers (e.g. in [4], [7], [19]). Absolute plots on the other hand are not as useful for the comparison, as there can be some outliers with extremely large values. These outliers will lead to skewed plots, that will not allow a proper comparison. This cannot happen to the normalized box plots, since outliers are constraint to a much smaller interval and can therefore not skew the resulting plot as much.
Since scientific papers are often created with LaTeX, the file formats offered by the evaluation framework focus on ease of use with LaTeX. The diagrams are saved in the PDF and TeX file formats, which can both be conveniently used to incorporate the diagrams into scientific papers.
The diagrams in the TeX format allow for adjustments later on and are easier modifiable compared to the PDF diagrams. The PDF diagrams on the other hand can also be used outside of LaTeX documents, which makes them more versatile.

3.4 Additional Features

This section goes into the details of some additional features of the evaluation framework.

3.4.1 Multiprocessing

To speed up the performance of the framework, multiprocessing is already implemented in some stages of the analysis. The following processes of the evaluation framework are parallelized in the currently implemented version:

- task set generation, over the number of task sets
- cause-effect chain generation, over all task sets
- schedule generation, over all task sets (only performed if necessary)
- evaluating one analysis method, over all cause-effect chains (except `gohari22`)

During the execution, the framework will then create multiple threads for these steps, depending on the number of threads assigned to the framework (either via the GUI or the CLI).
The multiprocessing is implemented with the help of the python multiprocessing-package [27]. A pool with number of assigned threads is created. With the help of the `map`-method, a specified function is called multiple times with a different input argument. These instances are then executed on the different processes running in parallel. Analogously, there exists the `starmap`-method, which is also part of the multiprocessing-package. This method is similar to the `map`-method, but allows the parallelization of methods that take more than one input argument. The input arguments are passed as a list of tuples, where each tuple is used for one method call of the method that should be parallelized.
Internally, the multiprocessing-package creates a number of duplicate threads that execute the same functions on different copies of the same object. Therefore, it is not feasible to adjust these copies in any of the launched threads, as the changes will disappear after the parallelized section of code is finished.
To prevent this, a universally unique identifier (UUID) can be assigned to an object if necessary. This way, a thread can return the UUID to the thread when it is finished together with the changes that were made to this particular object. This mechanism is implemented in the evaluation framework for the schedule generation, as a reference to the schedule is stored in the corresponding task set.
The parallelization allows to perform larger evaluations with more cause-effect chains in less time, compared to an serialized implementation. The speed up is especially noticeable with a large number of threads assigned to the framework.

3.4.2 Saving and Loading Cause-Effect Chains

The evaluation framework offers the option to store generated cause-effect chains. If the option is selected, the cause-effect chains will be stored in the output directory as a pickle-file. Afterwards, this pickle file can be loaded into the framework instead of generating new cause-effect chains again.
Moreover, the framework also saves the cause-effect chains in a YAML file. The advantage of the YAML file over the pickle file is that the format is directly readable by humans and could also be used in other applications that are able to process YAML files. The YAML

file includes the cause-effect chains, as a list of task ids, and a list of all generated tasks, with all of their attributes. Currently, a YAML file can be generated by the evaluation framework to save the cause-effect chains. However, the corresponding YAML import is not yet implemented.

This feature of saving and loading cause-effect chains allows to redo the analysis multiple times on the same data. It can be especially useful for comparisons, in which the analysis method itself gets modified. Running the analysis on the same cause-effect chains will then return comparable results. It also allows to reuse the cause-effect chains for other evaluations or visualizations, as they can be loaded by another application.

3.4.3 Exporting Analysis Results

After the evaluation is finished, the result can be exported from the framework. Therefore, the option "export raw analysis results (CSV)" has to be selected before running the analysis. If the option was selected the results will be stored in a comma separated format (CSV) in the output folder.

The first line of the file is the header of the table. It stores the short names of the selected analysis methods. In the following rows the results of each analysis method for a single cause-effect chain is stored.

$$\text{analysis}_1.\text{name}, \text{analysis}_2.\text{name}, ..., \text{analysis}_n.\text{name}$$
$$\text{analysis}_1.\text{eval}(E_1), \text{analysis}_2.\text{eval}(E_1), ..., \text{analysis}_n.\text{eval}(E_1)$$
$$\text{analysis}_1.\text{eval}(E_2), \text{analysis}_2.\text{eval}(E_2), ..., \text{analysis}_n.\text{eval}(E_2)$$
$$...$$
$$\text{analysis}_1.\text{eval}(E_m), \text{analysis}_2.\text{eval}(E_m), ..., \text{analysis}_n.\text{eval}(E_m)$$

CSV is a widely used format. The benefit of exporting the analysis results in this format is, that it can easily be used with other software for plotting. An example could be the programming language R, which offers multiple options for loading CSV-data and is also useful for statistical analysis or plotting. Alternatively, a self-written program in another programming language could also use the CSV-data for statistical analysis or plotting of the stored results.

3.4.4 Command Line Interface

Another additional feature of the evaluation framework is the command line interface, which can be used to configure and start an evaluation from a terminal. The implementation is done in the file consoleInterface.py and the available options are similar to the

options from the GUI. To configure the evaluation via the command line interface, the framework has to be called from a shell terminal with a number of arguments. These arguments are parsed with the getopt module available in Python.
At the end, the results will be either saved in a file (equivalent to the evaluation via the GUI) or returned in the shell terminal. This way, another program could directly use the results as input e.g. to do more advanced evaluations, that cannot yet be performed with the evaluation framework.
A more detailed description on how the command line interface can be used to execute the evaluation framework is given in section 4.3.

Chapter 4

Using the Framework

This chapter explains how the evaluation framework can be used via the graphical user interface and via the command line interface. Afterwards, an example comparison is performed, where multiple analysis methods implemented in the framework are evaluated on a specific benchmark.
The chapter also includes information about how to add another analysis method in the future, which can be useful to perform research on new analysis methods.

4.1 Required Software

Before the framework can be started, the following software needs to be installed on the users computer. During development, a machine with the operating system Ubuntu 22.04 was used. Depending on the users operating system, the steps necessary to install the software might differ.

Python 3

Since the evaluation framework is implemented in Python, the Python Interpreter has to be installed in order to start the application. During the development of this framework the interpreter version 3.10 was used, but newer versions will likely work as well. If Python is not already installed on the users system, the following steps can be performed to install the interpreter version 3.10:

```
# adding deadsnakes PPA with older python releases
sudo apt install software-properties-common
sudo add-apt-repository ppa:deadsnakes/ppa
sudo apt update
```

```
# installs python3.10, virtual environment and development tools
sudo apt install python3.10 python3.10-venv python3.10-dev
```

It is also advised to create a virtual environment for the installation of the necessary Python packages. This way possible dependency conflicts between different packages are avoided:

```
# create virtual environment
python3.10 -m venv e2eEval

# activate virtual environment
. e2eEval/bin/activate
```

Tkinter

To use the graphical user interface, Tkinter has to be installed. As already mentioned in section 3.2, PySimpleGUI is used for creating the user interface. Since PySimpleGUI is internally based on Tkinter, the corresponding python package has to be installed in order to launch a PySimpleGUI-based user interface. The following command is used to install Tkinter from a shell terminal:

```
sudo apt-get install python3.10-tk
```

PySimpleGUI

For using PySimpleGUI, the corresponding python package has to be installed on the system. After the authors of PySimpleGUI decided to release the new versions of PySimpleGUI under a proprietary license, they removed all versions that were published as free software from the public Python Package Index in July 2024. Therefore, it is currently not possible to install any version of PySimpleGUI from the Python Package Index without a proprietary license.
In order to avoid the acquisition of a proprietary license for PySimpleGUI, there are still some options available to install the package. First of all, there still exist some forks of PySimpleGUI on github, which were created before the authors introduced their proprietary license. An example is FreeSimpleGUI [8], which was forked from PySimpleGUI and is still actively maintained and available under a LGPL license.
Another option is to use the latest version of PySimpleGUI, which was still published under the LGPL license (PySimpleGUI-4.60.5). Since this version is not available any more on the Python Package Index, it has to be installed from a local archive. The archive is published together with the source code of this thesis on github. It has been extracted

from a local system, that was used for creating the evaluation framework. For installing the local archive, the following command has to be executed in the root directory of the evaluation framework with the activated virtual environment:

```
pip install pySimpleGUI/PySimpleGUI-4.60.5.tar.gz
```

Afterwards, PySimpleGUI is installed on the users system and can be used by the evaluation framework for launching the graphical user interface.

Additional Python Packages

The following python packages have to be installed on the system, in order to launch the evaluation framework correctly:

- **scipy:** The scipy package is used in the implementation of the automotive benchmark. For example it provides a method for the exponential Weibull distribution, which is used in the benchmark.
- **numpy:** The installation of numpy is necessary in order to use scipy, since scipy is based on numpy. Furthermore, numpy provides some methods (e.g. random.randint, random.uniform), which are used in both the automotive and the uniform benchmark.
- **matplotlib:** matplotlib is a library that is used for creating the box plots from the evaluation results.
- **PyYAML:** The PyYAML package is used by the evaluation framework to export the generated cause-effect chains to a YAML file.
- **tikzplotlib:** tikzplotlib [30] is a library that can be used together with matplotlib for generating the output diagrams. Since matplotlib does not directly offer to create vector graphics with PGF/TikZ for LaTeX, tikzplotlib is necessary to generate these graphics. It was decided to use a fork of tikzplotlib [12] instead of tikzplotlib [30], because the current version of tikzplotlib is incompatible with the recent versions of matplotlib.

To install all five libraries on the local system, the following command should be run in a shell terminal with the activated virtual environment from the root directory of the evaluation framework:

```
pip install -r requirements.txt
```

All of the above mentioned packages are listed in the requirements.txt file and installed automatically. In case the user wants to manually install the packages, the following commands can be used:

```
pip install scipy numpy matplotlib PyYAML
pip install git+https://github.com/JasonGross/tikzplotlib.git
```

4.2 Using the GUI

The evaluation framework is started by calling the main function located in e2eMain.py. In case no additional arguments are passed to the main function, the graphical user interface is started automatically. The GUI can then be used to set the desired parameters and start the evaluation. For starting the framework from a shell terminal, the following command can be used in the root directory of the evaluation framework:

```
python3.10 e2eMain.py
```

PySimpleGUI will then launch the graphical user interface. The interface can be separated into 5 blocks, as seen in figure 4.1. Each of these blocks is used to control a different part of the framework. In the following, each block of the GUI is explained in more detail.

Block 1: General Settings

The general settings block first offers the option to either generate new cause-effect chains, or load already existing cause-effect chains from a file. In case the "Generate Cause-Effect Chains"-option is selected, there is also the option to store the generated cause-effect chains to a file. This can be done by checking the box on the right.
In case the user decides to load an already existing set of cause-effect chains from a file, the option to browse the file system becomes available. A file path can be directly entered into the input field. Alternatively, the user can click the "Browse"-button which opens a file browser (see figure 4.2). This file browser can be used to navigate the local file system and find the desired file for performing the analysis. After a file has been selected, the corresponding file path can be seen in the input field for the file name. It will later be automatically opened, when the analysis gets started.
When the user decides to load the cause-effect chains from a local file, the blocks 2 and 3 of the evaluation framework will be disabled. This is due to the fact, that all options of blocks 2 and 3 are only necessary for the use-case of creating new cause-effect chains. As there is no need for generating new cause-effect chains when loading already existing ones, all options of these blocks are disabled.

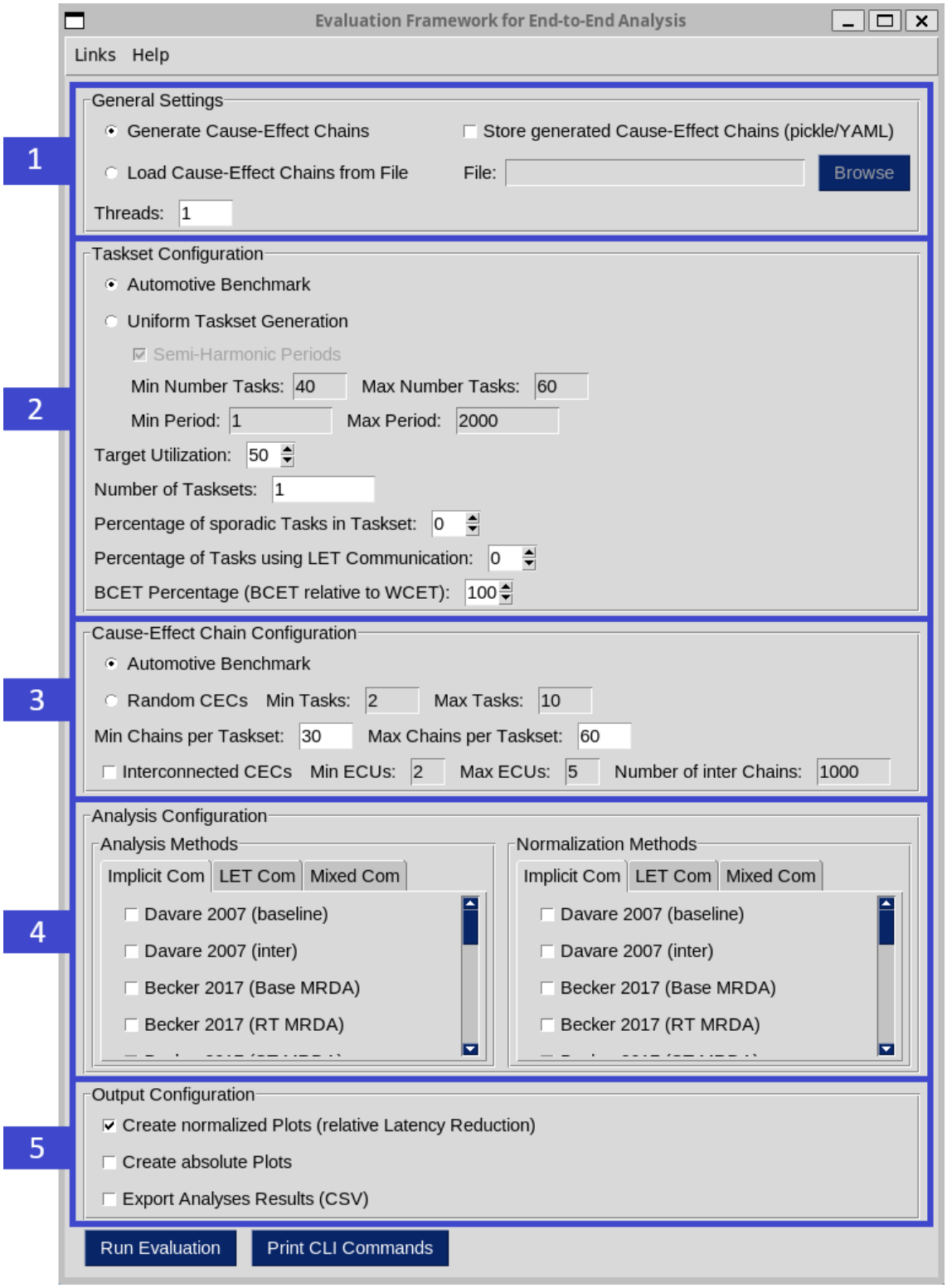


Figure 4.1: Overview of the graphical user interface of the evaluation framework; The GUI can be separated into five blocks, which are used to control different parts of the evaluation.

The last option available in the general settings block is used to control the number of threads. By default, the evaluation framework only uses a single thread. In case there are more threads available on the users machine, the evaluation can be speed up by setting a larger number of threads in the framework. The number of threads should not exceed the number of threads available on the users machine, as this will not increase the performance of the evaluation framework any more. It is advised to assign at most all but one or two of the available threads to the framework. This way the evaluation will be performed relatively fast, while also allowing other processes running on the machine to be executed without too much delay.

Block 2: Task Set Configurations

The configuration options in block 2 are used to control the generation process of the underlying task sets. They are only available, in case the user decided for the "Generate Cause-Effect Chains"-option in the general settings block.

At first, the user can decide to either use the automotive benchmark or the uniform task set generation algorithm for generating the task sets. In case the user decides to select the uniform task set generation algorithm, additional configuration options for this benchmark become available.

The first additional option is the option to use semi-harmonic periods. If this option is checked, the algorithm will only create tasks with periods in $\{1, 2, 5, 10, 20, 50, 100, 200, 1000\}$, as described in section 2.5.2. This assures, that the hyperperiod of the task set will stay relatively small and can be at most 1000. Small hyperperiods are important for some of the end-to-end analysis methods, since their execution time depends heavily on the hyperperiod and can increase exponentially. Therefore, large hyperperiods can lead to very long execution times.

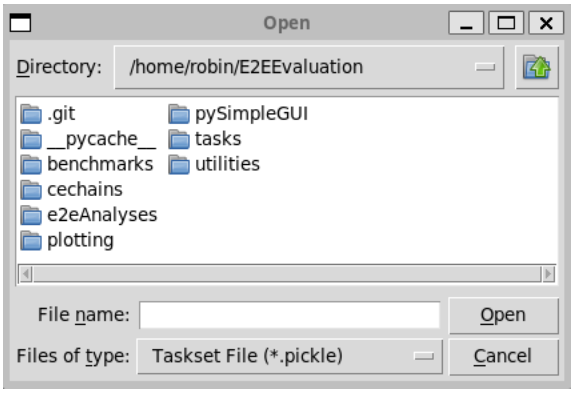


Figure 4.2: File browser, that is launched to select a python pickle file with a set of cause-effect chains

In case the user did not check the semi-harmonic periods option, the algorithm will generate tasks with arbitrary periods within the interval starting at "Min Period" and ending at "Max Period", specified in the input fields below. This will also disable the use of the automotive benchmark in block 3 of the GUI, since this benchmark can only create cause-effect chains when there are multiple tasks with the same period. As this is very unlikely, the automotive benchmark is in most cases not able to produce any cause-effect chains.
There is also the option to decide, how many task should be generated for each task set. With the option "Min Number Tasks" and "Max Number Tasks", the user can specify a range of tasks for all task sets. The algorithm will then create a number of tasks, that lies in this interval for each task set.

In the lower half of block 2 of the GUI are the task set configuration parameter, that are relevant for both benchmarks.
The first option here is the target utilization of the task sets, given in percent. A whole number between 1 and 99 can be entered in this input field, and the selected generation algorithm will then try to approximate this target utilization.
Next is the option to specify, how many task sets should be created. The default value is set to 1, which is the lowest value to run an evaluation. For large-scale evaluations, this value should be increase significantly, resulting in many different cause-effect chains on different task set.
Lastly, there are the three options "Percentage of sporadic Tasks in Task Set", "Percentage of Tasks using LET Communication" and "BCET Percentage". All three options are percentages between 0 and 100. If the "Percentage of sporadic Tasks in Task Set" remains at 0, all tasks in the task set will be periodic tasks. In case the value is higher than 0, each task of a task set has a random chance to be labeled as a sporadic task. This option will only label the task as periodic or sporadic, but will not change their periods and minimum/maximum inter-arrival times.
The same is done for the option "Ratio of Tasks using LET Communication". Again, each task gets a random chance to be labeled as using LET communication. Otherwise, the task will default to using implicit communication.
The BCET percentage is used to calculate a BCET relative to the WCET. As default the value is set to 100%, meaning that all BCETs are equal to the tasks WCETs. In case a lower percentage is assigned, the BCET of a task will be calculated by multiplying that percentage with the WCET of the task.

Block 3: Cause-Effect Chain Configurations

The third block of the GUI is used to set all the configuration parameters for the cause-effect chain generation process.

The first option is again to decide between the automotive benchmark and a random cause-effect chain generation. If the user decides to use the random cause-effect chains there is additionally the option to specify, how many tasks should be in one cause-effect chain. For this, the "Min Tasks" and "Max Tasks" values can be set, to get the desired length for the cause-effect chains. This option is not needed for the automotive benchmark, since the automotive benchmark will automatically select a number of tasks for the cause-effect chain.

After that follows the option to select, how many cause-effect chains should be generated per each task set, created in the previous step. Again, an interval can be specified using the lower bound "Min Chains per Task Set" and the upper bound "Max Chains per Task Set". The framework will then generate a random number of cause-effect chains within the specified interval for each task set.

At the end, there is the option to create interconnected cause-effect chains with a corresponding check box. When the users checks the checkbox, the option to set a number of cause-effect chains becomes available. A minimum number and a maximum number for the number of ECUs can be set. The framework will then select a number of local cause-effect chains within that range, where each local cause-effect chain has a different underlying task set and link them together. Additionally, there is the option to specify how many interconnected cause-effect chains should be created from the local chains by setting the value of "Number of inter Chains" accordingly. This will repeat the process above as often as specified.

If the "Interconnected CECs" checkbox was not selected, the framework will only create local cause-effect chains.

Block 4: Analysis Configurations

In block number 4, the desired analysis methods for the evaluation can be selected. There are two identical tab groups, that are used differently. The tab group on the left side is used to select all analysis methods that should be evaluated. The tab group on the right side is later used for plotting. These analysis methods are also evaluated on the generated cause-effect chains and are afterwards used for the normalization, to get normalized plots. It is therefore possible to evaluate any of the implemented end-to-end analysis methods and normalize it to any other end-to-end analysis method.

The analysis methods are sorted within the tab groups according to the communication mechanisms they can analyze. Analysis methods that can handle tasks with implicit communication are available in the "Implicit Com" tab, methods for tasks with LET communication are in the "LET Com" tab and analysis methods that can handle tasks sets that consist of tasks communicating with implicit and LET communication are sorted into the "Mixed Com" tab.

Block 5: Output Configurations

The output configuration in block 5 offers three different options. By default the option "Create normalized Plots" is selected. This option will create normalized box plots, where each analysis method selected on the left side from the analysis configuration is normalized to each analysis method from the right side of the analysis configuration. Each analysis method is normalized individually, resulting in one box plot per selected analysis method. Additionally, one box plot is created, where all selected analysis methods are compared in a single plot, normalized according to one specific method.

The next option is the "Create absolute Plots" option. This option also creates box plot diagrams, but does not normalize the results before creating the box plots.

The last option is to export the analysis results to CSV. If this option is selected, a CSV file will be created in the output folder, where the latency results of all the selected analysis and normalization methods are stored. This can be useful, in case the results need to be reused, for example for different kinds of diagrams currently not available in the framework.

After setting all of the above configuration parameters as desired, the evaluation can be started by pressing the "Run Evaluation" button on the bottom of the GUI. The GUI will then collect all the input values and pass them to the evaluation framework.

When the evaluation is finished, the GUI will open a pop up window (see figure 4.3) which indicates, that the framework successfully finished with the evaluation. The window will also tell the user the output folder, which was created during the evaluation and where all the outputs of this particular run were stored.

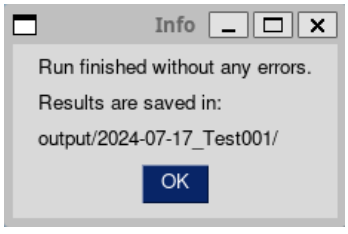


Figure 4.3: Feedback pop up, after successful execution of the evaluation framework

In case the user did not enter correct values into the GUI (e.g. user entered literals instead of numbers), there will be a feedback pop up window which informs about the incorrect values. The user can then correct the input parameters and try to start the evaluation again.
Alternatively, it is also possible to print the corresponding CLI commands py pressing the button "Print CLI Commands" at the bottom of the GUI. This will print the CLI commands to the system console, that are equivalent to the selected options from the GUI. This way it is easier to create the CLI commands with many arguments.

4.3 Using the CLI

This section explains, how the command line interface of the evaluation framework can be used to perform an evaluation.
The evaluation framework is designed in such a way, that the command line interface mode is automatically started, in case the evaluation framework is called from a terminal with one or multiple arguments. There are currently three different ways, the framework can be invoked from the console:

1 **Cause-Effect Chain Generation**
The purpose of the first available option is to generate a set of cause-effect chains and store them in a file. This option can be started with the following command:

```
python3.10 e2eMain.py generate-cecs [options]
```

There are a number of different options available to configure the cause-effect chain generation. A complete list with long and short options is given in the appendix of this work in section A.1.

Depending on the type of the option, the value is set differently. For options of type boolean, the value is `False` per default and can be set to `True` by simply adding it to the argument list:

```
python3.10 e2eMain.py generate-cecs --use_automotive_taskset_generation
```

Options that are of type float, int or string can be set in the following way:

```
python3.10 e2eMain.py generate-cecs --target_util=0.7
```

The above command will call the generate cause-effect chains function of the framework and overwrites the default target utilization of 0.5 with the value 0.7. The default values of each option can also be found in the appendix section A.1.

In case there is a short option available, the value of the variable is passed as another argument behind the short option. The following command starts the evaluation framework from the console with 4 threads:

```
python3.10 e2eMain.py generate-cecs -t 4
```

After the command has successfully terminated, the cause-effect chains will be stored in a file and can be used in another run of the evaluation framework to perform the evaluation on them with the desired analysis methods.

2 Cause-Effect Chain Analysis

Second is the possibility to invoke the framework via the CLI to analyze a cause-effect chain file. The following command is used to start the analysis:

```
python3.10 e2eMain.py analyze-cecs [options]
```

Again, there is a large number of possible options that can be added to the command to configure, which analysis methods should be run on the particular file and what output should be generated. All available options are listed in the appendix section A.1.

The value of the different variables can be assigned in the same way as shown above for the cause-effect chain generation process.

The only difference are the short options `-a` and `-n`, which are used to specify the analysis and normalization methods the evalution framework should use. These two short options can also be repeated, so that multiple analysis methods are evaluated with one run of the evalutation framework. An example is shown in the command below, where the methods `duerr19_mrt` and `bi22` are evaluated and normalized to the results of `davare07`:

```
python3.10 e2eMain.py analyze-cecs -a duerr19_mrt -a bi22 -n davare07
```

In the appendix section A.2 a list with all available analysis methods is given that can be used with the short options `-a` and `-n`.

3 Simple Analysis Mode

The last option to invoke the evaluation framework from a terminal is with the simple analysis mode. This option is very straight forward and has two required and one optional argument:

```
python3.10 e2eMain.py <analysis> <cec_file> [threads]
```

The first required argument is a name of one of the available analysis methods from the evaluation framework. It should match one of the keys of the analysis dictionary of the framework (`analysesDict` in the framework.py file, also see appendix section A.2).

The second required argument is the file path of a file, that stores cause-effect chains in the pickle file format. This file can be generated either with the cause-effect chain generation option from the command line interface or via the graphical user interface. In both cases the format of the file is the same and can be used by the command.

Lastly follows one optional option, to specify the number of threads that should be used by the evaluation framework to perform the analysis. In case no number is passed, the framework will be started with only one available thread.

The difference of this simple analysis mode compared to the cause-effect chain analysis option is, that this option has far less options and will directly output the results of the evaluation to the console. This way, the evaluation framework will not create the output plots and not export the results in the CSV file format, but print the results.

With this option, the framework could be easily called by another application, which can afterward use the output returned directly via the console. The framework is therefore easily integrable into other programs.

4.4 Example Evaluation

This section will present two different example usages of the evaluation framework from start to finish and explains how they can be reproduced. For this purpose, the same evaluation is performed first via the graphical user interface and second via the command line interface. It should give an impression on how the framework can be used to evaluate the performance of one or multiple analysis methods.

4.4.1 Scenario

In this example the three analysis methods "Kloda 2018", "Dürr 2019 (MRT)" and "Günzel 2023 (local MRT)" are compared against each other and normalized according to the results of "Davare 2007". The uniform task set generation is used to generate 100 different task sets. Afterwards, a random number between 30 and 60 cause-effect chains is generated for each of these task sets using the automotive benchmark.

To evaluate the performances of the tree methods, a normalized and absolute plots with the evaluation results should be generated. In order to reuse the results with another plotting tool later on, the computed latencies should also be exported to a CSV file.

4.4.2 Evaluation via the GUI

For performing the evaluation, the graphical user interface is first started with the following command from the root directory of the evaluation framework:

```
python3.10 e2eMain.py
```

After the framework has started, the configuration is entered into the input mask as seen in figure 4.4. In the general configuration block the option "Store generated cause-effect chains" is selected and the number of threads for the framework is set to 10.

In the task set configuration block the generation algorithm is changed to the uniform algorithm with its default parameters. The target utilization is changed to 70% and the number of task sets is set to 100 to get a sufficiently large sample size for the evaluation.

The configuration options in the cause-effect chain block are left at its default values, so there will be 30 to 60 cause-effect chains is generated for each previously generated task set with the automotive benchmark.

In the analysis configuration tab groups, the analysis methods "Kloda 2018", "Dürr 2019 (MRT)" and "Günzel 2023 (local MRT)" are selected in the left tab group. In the right tab group, the analysis "Davare 2007 (baseline)" is selected, so that can be used for the normalized evaluation plots.

Lastly, all three options in the output configuration block are selected, so that normalized and absolute plots as well as a CSV-file with the results of the evaluation are generated. Then the evaluation is started by pressing the "Run"-button at the bottom of the window. Depending on the selected analyses and the number of task sets, the evaluation can take some time to finish.

When the evaluation is finished, the graphical user interface will return with a feedback pop up window, that specifies the output directory. In case the evaluation generated output files, they will be stored in the "output" folder located in the root directory. For every run of the evaluation framework a new directory is created within the output folder, named after the current date and the including a sequential number counting up.

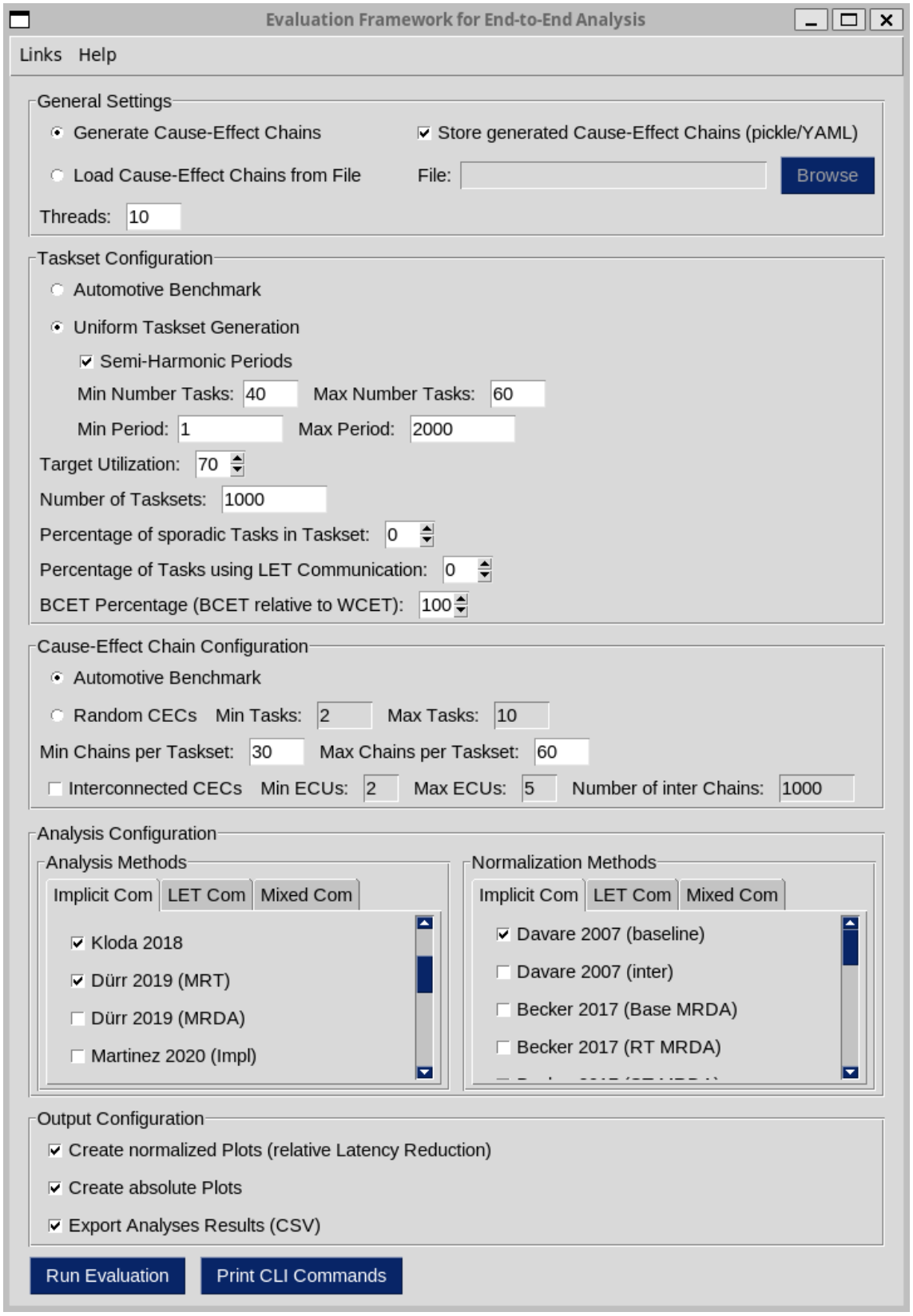


Figure 4.4: Example configuration for an evaluation via the graphical user interface

At the end of the described example evaluation, the folder structure looks as follows:

```
E2EEvaluation
├─ output
│  └─ 2024-08-02_Test001
│     ├─ cause_effect_chains.pickle
│     ├─ cause_effect_chains.yaml
│     ├─ absolute.pdf
│     ├─ absolute.tex
│     ├─ D19(MRT).pdf
│     ├─ D19(MRT).tex
│     ├─ K18.pdf
│     ├─ K18.tex
│     ├─ G23(L-MRT).pdf
│     ├─ G23(L-MRT).tex
│     ├─ normalized_to_D07.pdf
│     ├─ normalized_to_D07.tex
│     ├─ D19(MRT)_normalized_to_D07.pdf
│     ├─ D19(MRT)_normalized_to_D07.tex
│     ├─ K18_normalized_to_D07.pdf
│     ├─ K18_normalized_to_D07.tex
│     ├─ G23(L-MRT)_normalized_to_D07.pdf
│     ├─ G23(L-MRT)_normalized_to_D07.tex
│     └─ results.csv
└─ ...
```

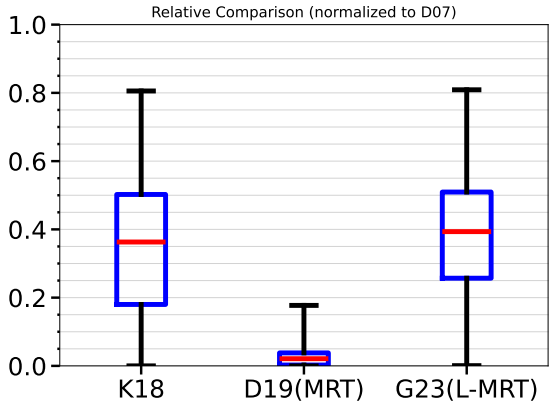


Figure 4.5: Normalized evaluation result for an example run of the framework; Results are normalized according to Davare 2007

The directory includes a pickle and a YAML file with the generated cause-effect chains, as well as the output diagrams and the CSV-file with the results of each analysis method. For each analysis method one absolute plot and one normalized plot were generated in both the PDF and TeX file format. Additionally, one absolute and one normalized plot with all of the analysis methods combined in a single figure were generated. The normalized PDF plot can be seen in figure 4.5, the other output diagrams can be found in the appendix sections A.3 and A.4 of this thesis.
The plots in the PDF file format can be directly included in LaTeX with the `\includegraphics` command. In case the TeX diagrams are used, the following lines should be placed at the beginning of the LaTeX document:

```
\usepackage[utf8]{inputenc}
\usepackage{pgfplots}
\DeclareUnicodeCharacter{2212}{\textminus}
\usepgfplotslibrary{groupplots,dateplot}
\usetikzlibrary{patterns,shapes.arrows}
\pgfplotsset{compat=newest}
```

4.4.3 Evaluation via the CLI

To perform the example evaluation with the described configuration, two separate calls of the evaluation framework are necessary. At first, the task sets and the cause-effect chains are generated and stored to a file with one command:

```
python3.10 e2eMain.py generate-cecs -t 10 --use_uniform_taskset_generation
↪ --use_semi_harmonic_periods --target_util=0.7 --number_of_tasksets=100
↪ --generate_automotive_cecs
```

The options passed via the command line interface are the same as for the graphical user interface. All parameters that were left on their default value are not passed as additional arguments when calling the evaluation framework from a console.
Next is the command to perform the actual analysis of the previously generated cause-effect chains. Therefore, the evaluation framework is invoked again, this time with the `analyze-cecs` option:

```
python3.10 e2eMain.py analyze-cecs -t 10 -a kloda18 -a duerr19_mrt -a
↪ guenzel23_local_mrt -n davare07 -f
↪ output/2024-08-04_Test001/cause_effect_chains.pickle
↪ --normalized_plots --absolute_plots --raw_analysis_results
```

The command includes the path for the cause-effect chain pickle-file, that should be analyzed. All three analyses are specified separately with the `-a` option of the CLI. With the option `-n` the analysis method used for normalization is set to `davare07`.

When the command has successfully finished its execution, the framework generated the same output as if it was started via the graphical user interface. As described above, all of the outputs can then be found in the output folder of the root directory.

Alternatively, the simple analysis mode can be used to analyze the file with the generated cause-effect chains. Since it only allows to perform one analysis method at a time, multiple commands are necessary to run all of the four analyses:

```
        python3.10 e2eMain.py kloda18
↪   output/2024-08-04_Test001/cause_effect_chains.pickle 10

        python3.10 e2eMain.py duerr19_mrt
↪   output/2024-08-04_Test001/cause_effect_chains.pickle 10

        python3.10 e2eMain.py guenzel23_local_mrt
↪   output/2024-08-04_Test001/cause_effect_chains.pickle 10

        python3.10 e2eMain.py davare07
↪   output/2024-08-04_Test001/cause_effect_chains.pickle 10
```

At the end of each command, the computed latencies are simply printed back to the console. It is therefore possible to read this console output with another application to conduct an evaluation. Otherwise, the values can also be copied manually by the user. An automated evaluation with the result is not performed and the evaluation framework will not generate any kind of output files for these commands.

4.5 Adding an Analysis Method to the Framework

One of the main ideas of the evaluation framework is to be easily extensible, in order to add new analysis methods and compare them to the already existing methods. Therefore, this section describes the necessary steps to add one or more analysis methods to the evaluation framework.
Each analysis is currently placed in a dedicated file within the analysis folder. It is therefore advised to create a new file for the new analysis method. The analysis method can then be split up into several methods in this file. In case the analysis method needs even more code, the code should be placed in one or multiple files in the utilities folder.
There should be one method, that is directly called by the framework for the analysis. This method should take one cause-effect chain as an input argument. In case the analysis is able to analyze interconnected cause-effect chain, the method should take a variable number of local cause-effect chains.

```
# analysis for local chains
def local_analysis(chain)
  ...

# analysis for interconnected chains
def inter_analysis(*local_chains)
  ...
```

At the end, the analysis method must return a single value for the (local/interconnected) cause-effect chain, that is the result of the analysis. This value will then later be used for the evaluation and comparison of the different methods.
Of course, the implementation of the analysis method should follow the implemented system model of the evaluation framework, including the implementation of a task, task set and the cause-effect chain itself. A detailed definition of the implemented system model is given in section 3.3.2.
After the source code has been added, it has to be integrated into the framework. First, the analysis function has to be imported into the framework.py at the top of the file. Second, a single line has to be added to the analysis dictionary in the framework.py file. It is located at the start of the file and has to be extended by another entry, describing the new analysis method.
The key of the entry is the name of the analysis method, that is used internally by the framework. This key is also used when calling the evaluation framework from the CLI for specifying a particular end-to-end analysis method. It should ideally be the same name as the name of the analysis method that is about to be added to the framework.
The value of the new entry is an `AnalysisMethod` object. When constructing this object, the previously implemented analysis method is passed as the first argument. After that follows one name for displaying in the GUI of the evaluation framework and one short name, that is used in the output plots.
The last argument is a list of feature values, that should describe the analysis method. For example an analysis method that is capable of analyzing cause-effect chains with implicit communication should have the value `'implicit'` in its feature list. This is necessary, so that the analysis methods can be categorized correctly and are placed in the corresponding tab in the GUI.
If done correctly as described, the analysis method will be displayed in the GUI in the correct tab and can be used in the same way like any other analysis method in the framework.

Chapter 5

Conclusion

This chapter summarizes the results of this thesis and gives an outlook on how this work could be extended in the future.

5.1 Summary

During the work on this master's thesis, an evaluation framework for end-to-end analyses has been developed. This framework combines many state-of-the-art analyses and allows it to compare their evaluations using different benchmarks. In total, there are 29 analysis methods from 12 different papers integrated into the framework. Many of these analyses are based on the original implementations provided by their authors and only slightly adjusted, so that they fit the implemented system model of the framework. In addition to that, two novel analysis methods, that are based on the analyses from [18] and [14] were also implemented and added to framework.

For performing the evaluation, the framework features two different benchmarks: the automotive benchmark, which is widely used because it is based on a real-world use case from the automotive industry, and the uniform benchmark, which is often used because of its simplicity. Both benchmarks can be used interchangeably for the generation of task sets and cause-effect chains.

To use the evaluation framework, two different interfaces are offered. The graphical user interface, created with pySimpleGUI, grants to easily use the evaluation framework. Users can directly see all of the available options, benchmarks and analysis methods and set their parameters by interacting GUI. In case an invalid input was passed, the GUI will give the user a feedback in order to correct the input. In case the evaluation finished successfully, the GUI will print the file path of the output folder, containing the results of the evaluation.

The command line interface on the other hand is as powerful as the graphical user interface, because it allows to perform the same evaluation without limitations. It can either be used by the user directly or even be controlled by another application, that could for example

reuse the results of an evaluation of some analysis. To simplify this process, a command can easily be created using the graphical user interface, which is able to convert its current state to the corresponding commands for the CLI.

At the end of the evaluation, multiple files can be generated by the framework to compare the results of the different analysis methods. Diagrams with absolute and normalized box plots are saved in the output folder in the PDF and the TeX file format, which allow an easy integration in further scientific work. The generated cause-effect chains can be saved in a pickle file, which allows to load the cause-effect chains back into the framework, and in a YAML file, which includes the cause-effect chains in a format that is readable for humans. Additionally, the computed latencies can be saved in a CSV file, that can be used by an external application (e.g. to generate different output diagrams).

The evaluation framework and its source code have been designed in such a way, that they are easily extensible by one or multiple new analysis methods. This is especially useful for researchers and research groups working on new end-to-end analysis methods, as they can quickly integrate their analysis prototype into the evaluation framework and compare their analysis to the already existing analysis methods.

In order to encourage other research groups to use the evaluation framework and test their newly developed analysis methods against the already existing ones, the source code of the framework has been publicly released on GitHub `https://github.com/tu-dortmund-ls12-rt/E2EEvaluation`.

5.2 Future Work

First of all, the evaluation framework could be extended by adding more analysis methods. Examples are the analysis by Kordon et al. [23], the analysis by Martinez et al. [25] and the analysis by Pazzaglia et al. [26]. They could not be added in time to the evaluation framework, as their integration appeared to be more challenging than the other available analysis methods. This is because the source codes for the analyses by Kordon et al. and Martinez et al. were not accessible, requiring a tedious and time intensive re-implementation of their analyses.

The source code of the analysis by Pazzaglia et al. is available on GitHub. However, the implementation is dependent on the IBM ILOG CPLEX Optimizer, which can be used to solve linear, mixed-integer and quadratic programming problems. Integrating this optimizer into the evaluation framework defies the idea of the framework, as the optimizer can only be accessed with an account and a corresponding subscription plan. An implementation of Pazzaglia et al. [26] without the use of this optimizer has to be worked out, in order to add their analysis to the framework.

Another possible opportunity to extend the framework is to add more evaluation benchmarks for the analysis methods. Currently there are two different benchmarks implemented in the framework, which are the most widely used in the literature. But there are also other algorithms available for the creation of task sets, for example the Dirichlet Rescale algorithm (also called DRS algorithm) presented by Griffin et al. in [11]. Integrating this benchmark into the framework would allow for a wider range of different task sets and therefore potentially more valuable results.
Furthermore, it is possible to adapt or expand the system model of the evaluation framework. In the current implementation, only preemptive fixed-priority scheduling is supported, which restricts the integration of analysis methods. By extending the framework to other scheduling schemes (for example non-preemptive scheduling or job-level fixed-priority scheduling) even more analyses could be added to the framework. The model could also be expanded by introducing a jitter for the tasks release, which can be used by some analyses (e.g. the analysis from Gohari et al. in [10]).
A more complex modification of the evaluation framework is the integration of event-triggered or even mixed-triggered cause-effect chains. The framework is currently limited to the analysis of time-triggered cause-effect chains. The extension to cause-effect chains with different triggers again allows to integrate more analysis methods (e.g. the analysis by [31]). On the other hand, it introduces a lot of difficulties, as the task model for event-triggered cause-effect chains has to be adjusted. Jobs in event triggered cause-effect chains are not periodically/sporadically released, but rather released by the occurrence of an event. In case the event occurs according to a periodic or sporadic pattern, the job releases may be approximated with a periodic of sporadic task. Therefore, the analysis of an event-triggered and an (approximated) time-triggered cause-effect chain could be possible simultaneously with analysis methods designed for these different classes of cause-effect chains.
Additionally, the evaluation framework could be made more modular. It offers to export the generated cause-effect chains to a YAML file, but a corresponding import that could load cause-effect chains from such a YAML file is still missing. This function would offer the opportunity to generate cause-effect chains externally by another program, so they can subsequently be loaded and analyzed with the evaluation framework.
Finally, it is possible to make the graphical user interface of the framework even more convenient. In the current implementation, the analysis methods are only sorted according to their supported communication mechanisms. But there are many other criteria, that could be used to sort them. This includes for example the supported release patterns, the analyzed metric (MRT, MDA or MRDA) and whether the analysis can be applied to local or interconnected cause-effect chains.

List of Figures

List of Algorithms

Bibliography

[1] Matthias Becker, Dakshina Dasari, Saad Mubeen, Moris Behnam, and Thomas Nolte. "End-to-End Timing Analysis of Cause-Effect Chains in Automotive Embedded Systems". In: *Journal of Systems Architecture* 80 (Oct. 2017), pp. 104–113. DOI: 10.1016/j.sysarc.2017.09.004.

[2] Matthias Becker, Dakshina Dasari, Saad Mubeen, Moris Behnam, and Thomas Nolte. "Synthesizing Job-Level Dependencies for Automotive Multi-rate Effect Chains". In: *2016 IEEE 22nd International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA)*. 2016, pp. 159–169. DOI: 10.1109/RTCSA.2016.41.

[3] Johannes Betz, Hongrui Zheng, Alexander Liniger, Ugo Rosolia, Phillip Karle, Madhur Behl, Venkat Krovi, and Rahul Mangharam. "Autonomous Vehicles on the Edge: A Survey on Autonomous Vehicle Racing". In: *IEEE Open Journal of Intelligent Transportation Systems* 3 (2022), pp. 458–488. ISSN: 2687-7813. DOI: 10.1109/ojits.2022.3181510. URL: http://dx.doi.org/10.1109/ojits.2022.3181510.

[4] Ran Bi, Xinbin Liu, Jiankang Ren, Pengfei Wang, Huawei Lv, and Guozhen Tan. "Efficient maximum data age analysis for cause-effect chains in automotive systems". In: *Proceedings of the 59th ACM/IEEE Design Automation Conference*. DAC '22. San Francisco, California: Association for Computing Machinery, 2022, pp. 1243–1248. ISBN: 9781450391429. DOI: 10.1145/3489517.3530609. URL: https://doi.org/10.1145/3489517.3530609.

[5] Enrico Bini and Giorgio C. Buttazzo. "Measuring the Performance of Schedulability Tests". In: *Real-Time Systems* 30 (2005), pp. 129–154. URL: https://api.semanticscholar.org/CorpusID:15604771.

[6] Abhijit Davare, Qi Zhu, Marco Di Natale, Claudio Pinello, Sri Kanajan, and Alberto Sangiovanni-Vincentelli. "Period optimization for hard real-time distributed automotive systems". In: *Proceedings of the 44th Annual Design Automation Conference*. DAC '07. San Diego, California: Association for Computing Machinery, 2007, pp. 278–283. ISBN: 9781595936271. DOI: 10.1145/1278480.1278553. URL: https://doi.org/10.1145/1278480.1278553.

[7] Marco Dürr, Georg Von Der Brüggen, Kuan-Hsun Chen, and Jian-Jia Chen. "End-to-End Timing Analysis of Sporadic Cause-Effect Chains in Distributed Systems". In: *ACM Trans. Embed. Comput. Syst.* 18.5s (Oct. 2019). ISSN: 1539-9087. DOI: 10.1145/3358181. URL: https://doi.org/10.1145/3358181.

[8] *FreeSimpleGUI*. URL: https://github.com/spyoungtech/FreeSimpleGUI?tab=LGPL-3.0-1-ov-file.

[9] Pourya Gohari, Mitra Nasri, and Jeroen Voeten. *Data-age analysis for multi-rate task chains*. URL: https://github.com/porya-gohary/np-data-age-analysis.

[10] Pourya Gohari, Mitra Nasri, and Jeroen Voeten. "Data-Age Analysis for Multi-Rate Task Chains under Timing Uncertainty". In: *Proceedings of the 30th International Conference on Real-Time Networks and Systems*. RTNS '22. Paris, France: Association for Computing Machinery, 2022, pp. 24–35. ISBN: 9781450396509. DOI: 10.1145/3534879.3534893. URL: https://doi.org/10.1145/3534879.3534893.

[11] David Griffin, Iain Bate, and Robert Davis. "Generating Utilization Vectors for the Systematic Evaluation of Schedulability Tests". In: Oct. 2020. DOI: 10.1109/RTSS49844.2020.00018.

[12] Jason Gross and Nico Schlömer. *JasonGross/tikzplotlib*. URL: https://github.com/JasonGross/tikzplotlib.

[13] Nan Guan, Martin Stigge, Wang yi, and Ge Yu. "New Response Time Bounds for Fixed Priority Multiprocessor Scheduling". In: Dec. 2009, pp. 387–397. DOI: 10.1109/RTSS.2009.11.

[14] Mario Günzel, Kuan-Hsun Chen, Niklas Ueter, Georg von der Brüggen, Marco Dürr, and Jian-Jia Chen. "Compositional Timing Analysis of Asynchronized Distributed Cause-effect Chains". In: *ACM Trans. Embed. Comput. Syst.* 22.4 (July 2023). ISSN: 1539-9087. DOI: 10.1145/3587036. URL: https://doi.org/10.1145/3587036.

[15] Mario Günzel, Kuan-Hsun Chen, Niklas Ueter, Georg von der Brüggen, Marco Dürr, and Jian-Jia Chen. *End-To-End Timing Analysis*. URL: https://github.com/tu-dortmund-ls12-rt/end-to-end.

[16] Mario Günzel, Kuan-Hsun Chen, Niklas Ueter, Georg von der Brüggen, Marco Dürr, and Jian-Jia Chen. *End-To-End Timing Analysis (inter)*. URL: https://github.com/tu-dortmund-ls12-rt/end-to-end_inter.

[17] Mario Günzel, Kuan-Hsun Chen, Niklas Ueter, Georg von der Brüggen, Marco Dürr, and Jian-Jia Chen. "Timing Analysis of Asynchronized Distributed Cause-Effect Chains". In: *2021 IEEE 27th Real-Time and Embedded Technology and Applications Symposium (RTAS)*. 2021, pp. 40–52. DOI: 10.1109/RTAS52030.2021.00012.

[18] Mario Günzel, Harun Teper, Kuan-Hsun Chen, Georg von der Brüggen, and Jian-Jia Chen. "On the Equivalence of Maximum Reaction Time and Maximum Data Age for Cause-Effect Chains". In: *35th Euromicro Conference on Real-Time Systems (ECRTS 2023)*. Ed. by Alessandro V. Papadopoulos. Vol. 262. Leibniz International Proceedings in Informatics (LIPIcs). Dagstuhl, Germany: Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2023, 10:1–10:22. ISBN: 978-3-95977-280-8. DOI: 10.4230/LIPIcs.ECRTS.2023.10. URL: https://drops-dev.dagstuhl.de/entities/document/10.4230/LIPIcs.ECRTS.2023.10.

[19] Mario Günzel, Niklas Ueter, Kuan-Hsun Chen, and Jian-Jia Chen. "Timing Analysis of Cause-Effect Chains with Heterogeneous Communication Mechanisms". In: *Proceedings of the 31st International Conference on Real-Time Networks and Systems*. RTNS '23. Dortmund, Germany: Association for Computing Machinery, 2023, pp. 224–234. ISBN: 9781450399838. DOI: 10.1145/3575757.3593640. URL: https://doi.org/10.1145/3575757.3593640.

[20] Arne Hamann, Dakshina Dasari, Simon Kramer, Michael Pressler, and Falk Wurst. "Communication Centric Design in Complex Automotive Embedded Systems". In: *29th Euromicro Conference on Real-Time Systems, ECRTS 2017, June 27-30, 2017, Dubrovnik, Croatia*. Ed. by Marko Bertogna. Vol. 76. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2017, 10:1–10:20. DOI: 10.4230/LIPICS.ECRTS.2017.10. URL: https://doi.org/10.4230/LIPIcs.ECRTS.2017.10.

[21] Christoph Kirsch and Ana Sokolova. "The Logical Execution Time Paradigm". In: Oct. 2012, pp. 103–120. ISBN: 978-3-642-24348-6. DOI: 10.1007/978-3-642-24349-3_5.

[22] Tomasz Kloda, Antoine Bertout, and Yves Sorel. "Latency analysis for data chains of real-time periodic tasks". In: *2018 IEEE 23rd International Conference on Emerging Technologies and Factory Automation (ETFA)*. Vol. 1. 2018, pp. 360–367. DOI: 10.1109/ETFA.2018.8502498.

[23] Alix Munier Kordon and Ning Tang. "Evaluation of the Age Latency of a Real-Time Communicating System Using the LET Paradigm". In: *32nd Euromicro Conference on Real-Time Systems (ECRTS 2020)*. Ed. by Marcus Völp. Vol. 165. Leibniz International Proceedings in Informatics (LIPIcs). Dagstuhl, Germany: Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2020, 20:1–20:20. ISBN: 978-3-95977-152-8. DOI: 10.4230/LIPIcs.ECRTS.2020.20. URL: https://drops-dev.dagstuhl.de/entities/document/10.4230/LIPIcs.ECRTS.2020.20.

[24] Simon Kramer, Dirk Ziegenbein, and Arne Hamann. "Real world automotive benchmarks for free". In: *6th International Workshop on Analysis Tools and Methodologies for Embedded and Real-time Systems (WATERS)*. Vol. 130. 2015.

[25] Jorge Martinez, Ignacio Sañudo, and Marko Bertogna. "End-to-end latency characterization of task communication models for automotive systems". In: *Real-Time Systems* 56 (July 2020). DOI: 10.1007/s11241-020-09350-3.

[26] Paolo Pazzaglia and Martina Maggio. "Characterizing the Effect of Deadline Misses on Time-Triggered Task Chains". In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 41.11 (2022), pp. 3957–3968. DOI: 10.1109/TCAD.2022.3199146.

[27] *Python Multiprocessing Library*. URL: https://docs.python.org/3/library/multiprocessing.html.

[28] Sayra Ranjha, Geoffrey Nelissen, and Mitra Nasri. "Partial-Order Reduction for Schedule-Abstraction-based Response-Time Analyses of Non-Preemptive Tasks". In: *2022 IEEE 28th Real-Time and Embedded Technology and Applications Symposium (RTAS)*. 2022, pp. 121–132. DOI: 10.1109/RTAS54340.2022.00018.

[29] O. Redell and M. Torngren. "Calculating exact worst case response times for static priority scheduled tasks with offsets and jitter". In: *Proceedings. Eighth IEEE Real-Time and Embedded Technology and Applications Symposium*. 2002, pp. 164–172. DOI: 10.1109/RTTAS.2002.1137391.

[30] Nico Schlömer. *tikzplotlib*. URL: https://github.com/nschloe/tikzplotlib.

[31] Yue Tang, Nan Guan, Xu Jiang, Zheng Dong, and Wang Yi. "Reaction Time Analysis of Event-Triggered Processing Chains with Data Refreshing". In: *2023 60th ACM/IEEE Design Automation Conference (DAC)*. 2023, pp. 1–6. DOI: 10.1109/DAC56929.2023.10248012.

Appendix A

Appendix Chapter

A.1 CLI configuration options

option	short option	type	default value
`--generate_cecs`		Bool	`False`
`--store_generated_cecs`		Bool	`False`
`--load_cecs_from_file`		Bool	`False`
`--cecs_file_path`	`-f`	String	`''`
`--number_of_threads`	`-t`	Int	`1`
`--debug_output`	`-d`	Bool	`False`

Table A.1: general configuration options of the evaluation framework and their default values

option	short option	type	default value
`--use_automotive_taskset_generation`		Bool	`False`
`--use_uniform_taskset_generation`		Bool	`False`
`--target_util`		Float	`0.5`
`--number_of_tasksets`		Int	`1`
`--sporadic_ratio`		Float	`0.0`
`--let_ratio`		Float	`0.0`
`--use_semi_harmonic_periods`		Bool	`False`
`--min_number_of_tasks`		Int	`40`
`--max_number_of_tasks`		Int	`60`
`--min_period`		Int	`1`
`--max_period`		Int	`2000`

Table A.2: task set configuration options of the evaluation framework and their default values

option	short option	type	default value
`--generate_automotive_cecs`		Bool	`False`
`--generate_random_cecs`		Bool	`False`
`--min_number_of_chains`		Int	`30`
`--max_number_of_chains`		Int	`60`
`--min_number_of_tasks_in_chain`		Int	`2`
`--max_number_of_tasks_in_chain`		Int	`10`
`--generate_interconnected_cecs`		Bool	`False`
`--min_number_ecus`		Int	`2`
`--max_number_ecus`		Int	`5`
`--number_of_inter_cecs`		Int	`1000`

Table A.3: cause-effect chain configuration options of the evaluation framework and their default values

option	short option	type	default value
`--normalized_plots`		Bool	`False`
`--absolute_plots`		Bool	`False`
`--raw_analyses_results`		Bool	`False`
`--output_dir`	`-o`	String	`''`
`--print_to_console`		Bool	`False`

Table A.4: output configuration options of the evaluation framework and their default values

A.2 Analysis Dictionary

Paper	Analysis	Short Name	Periodic	Sporadic	Implicit	LET	local	inter
[6]	davare07	D07	✓	✓	✓		✓	
[6]	davare07_inter	D07-I	✓	✓	✓		✓	✓
[2], [1]	becker17_NO_INFORMATION	B17	✓		✓		✓	
[1]	becker17_RESPONSE_TIMES	B17(RT)	✓		✓		✓	
[1]	becker17_SCHED_TRACE	B17(ST)	✓		✓		✓	
[1]	becker17_LET	B17(LET)	✓			✓	✓	
[20]	hamann17	H17	✓	✓		✓	✓	
[22]	kloda18	K18	✓		✓		✓	
[7]	duerr19_mrt	D19(MRT)	✓	✓	✓		✓	✓
[7]	duerr19_mrda	D19(MRDA)	✓	✓	✓		✓	✓
[4]	bi22	B22	✓		✓		✓	
[4]	bi22_inter	B22(I)	✓		✓		✓	✓
[10]	gohari22	G22	✓		✓		✓	
[17], [14]	guenzel23_local_mrt	G23(L-MRT)	✓		✓		✓	
[17], [14]	guenzel23_local_mda	G23(L-MDA)	✓		✓		✓	
[17], [14]	guenzel23_local_mrda	G23(L-MRDA)	✓		✓		✓	
[17], [14]	guenzel23_inter_mrt	G23(I-MRT)	✓		✓		✓	✓
[17], [14]	guenzel23_inter_mrda	G23(I-MRDA)	✓		✓		✓	✓
[19]	guenzel23_mix_pessimistic	G23(MIX-P)	✓	✓	✓	✓	✓	
[19]	guenzel23_mix	G23(MIX)	✓	✓	✓	✓	✓	
[19]	guenzel23_mix_improved	G23(MIX-I)	✓	✓	✓	✓	✓	
[18]	guenzel23_equi_mda	G23(EQ-MDA)	✓			✓	✓	
[18]	guenzel23_equi_mrt	G23(EQ-MRT)	✓			✓	✓	
[18]	guenzel23_equi_mrda	G23(EQ-MRDA)	✓			✓	✓	
[18]	guenzel23_equi_mrrt	G23(EQ-MRRT)	✓			✓	✓	
[18]	guenzel23_equi_impl_sched	G23(EQ-SCHED)	✓		✓		✓	
[18]	guenzel23_equi_impl_rt	G23(EQ-RT)	✓		✓		✓	
[2], [1]	beckerFast_NO_INFORMATION	BF	✓		✓		✓	
[1]	beckerFast_RESPONSE_TIMES	BF-RT	✓		✓		✓	
[1]	beckerFast_SCHED_TRACE	BF-ST	✓		✓		✓	
[1]	beckerFast_LET	BF-LET	✓			✓	✓	

A.3 Example output diagrams (PDF)

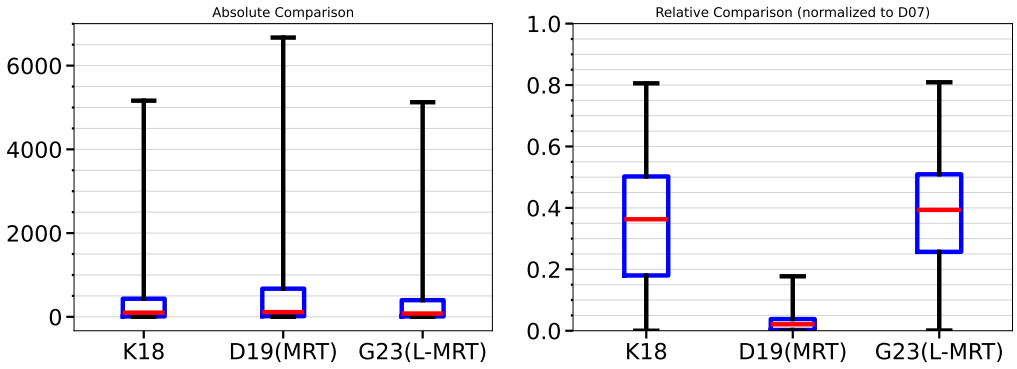


Figure A.1: Absolute and normalized evaluation results for an example run of the framework with three analysis methods normalized to Davare 2007

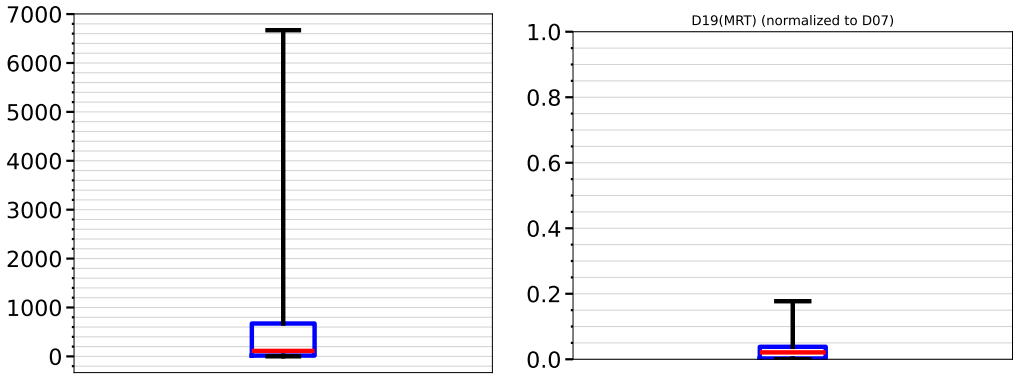


Figure A.2: Absolute and normalized evaluation results for an example run of the framework with Dürr 2019 (MRT)

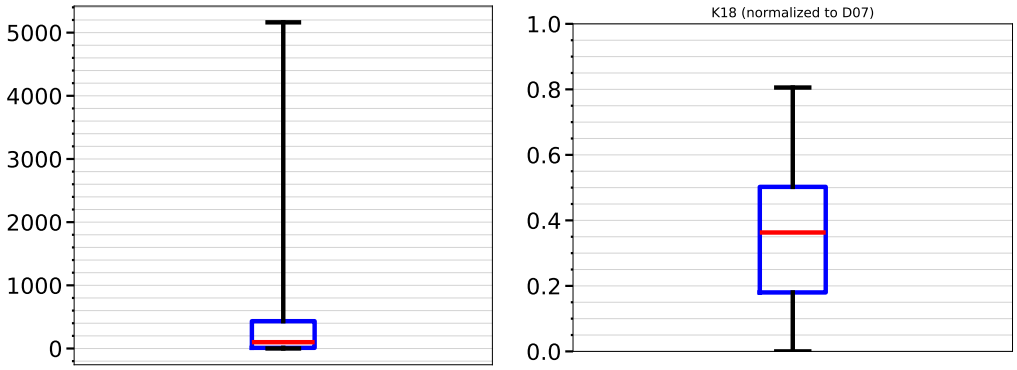


Figure A.3: Absolute and normalized evaluation results for an example run of the framework with Kloda 2018

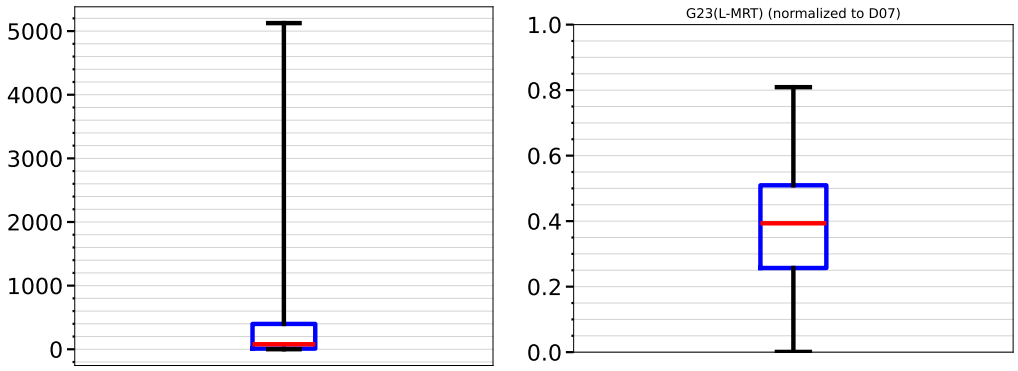


Figure A.4: Absolute and normalized evaluation results for an example run of the framework with Günzel 2023 (local MRT)

A.4 Example output diagrams (TeX)

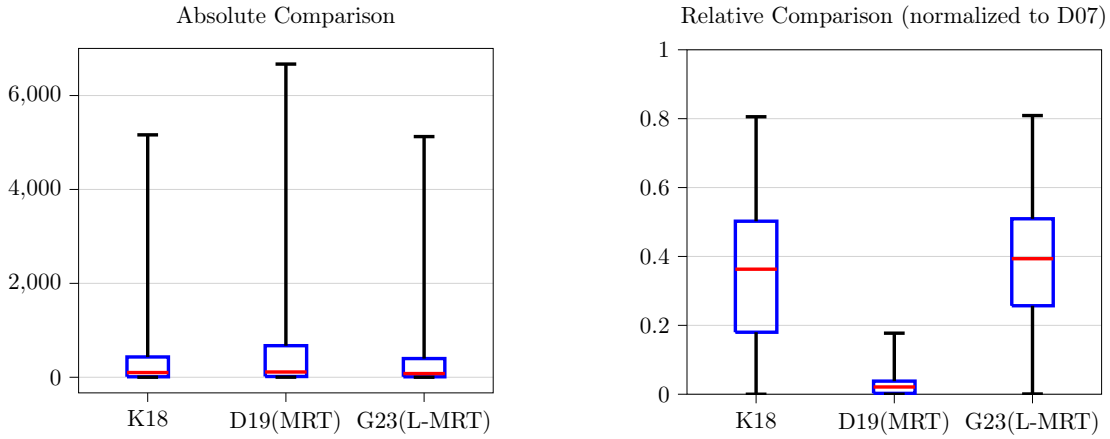


Figure A.5: Absolute and normalized evaluation results for an example run of the framework with three analysis methods normalized to Davare 2007

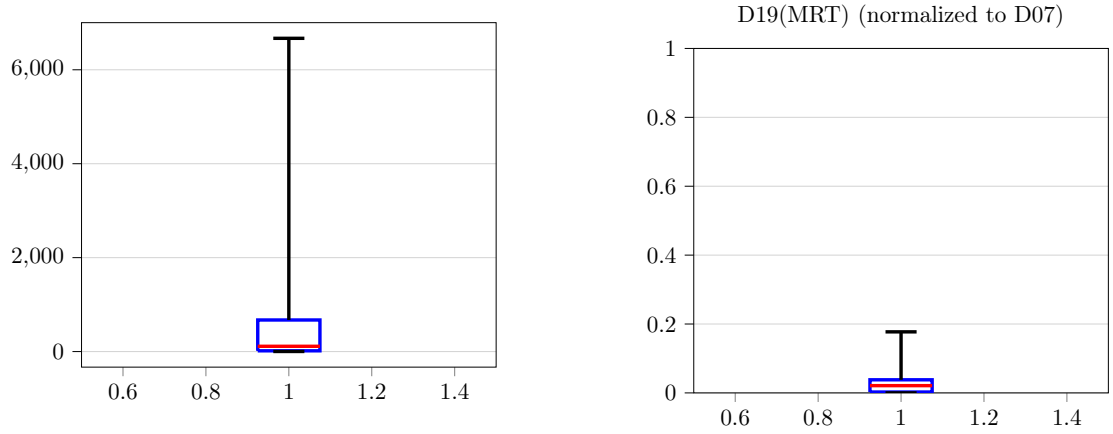


Figure A.6: Absolute and normalized evaluation results for an example run of the framework with Dürr 2019 (MRT)

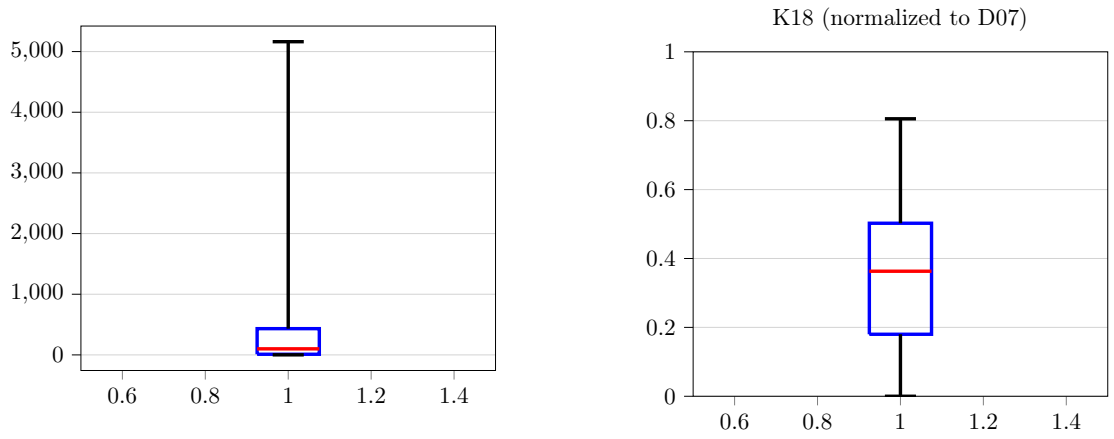


Figure A.7: Absolute and normalized evaluation results for an example run of the framework with Kloda 2018

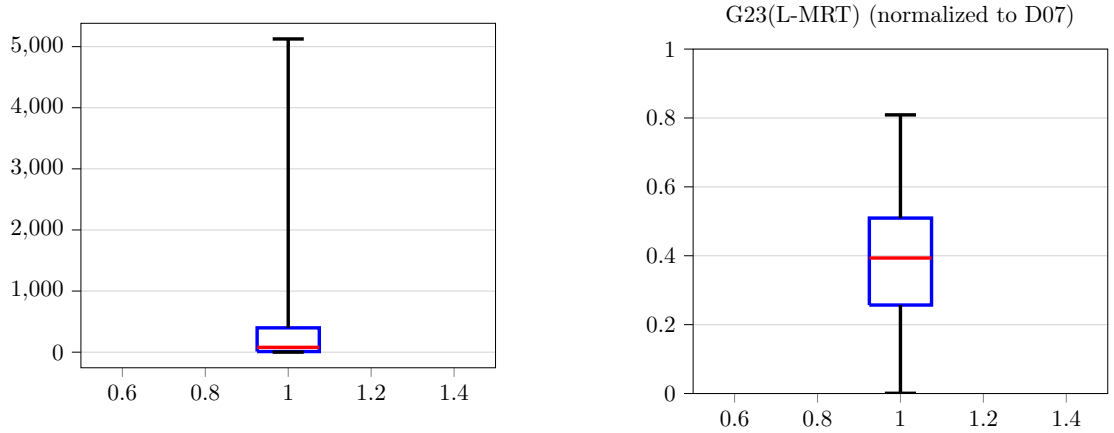


Figure A.8: Absolute and normalized evaluation results for an example run of the framework with Günzel 2023 (local MRT)