MASTER THESIS

Designing a Scheduling Tool for Distributed Real-Time Communication

submitted in partial fulfillment of the requirements to obtain the academic degree

Master of Science in Electronic Engineering

Dipartamento di Elettronica e Telecomunicazione

Author	Ayman HATOUM
Student ID	s263893
Supervisor	Prof. Edgar Ernesto SANCHEZ
University	Politecnico di Torino
Institute	Dipartamento di Automatica e Informatica
Supervisor	DI Sascha EINSPIELER
Company	Infineon Technologies AG
Subsidiary	Kompetenzzentrum Automobil- und Industrieelektronik GmbH

Italy, Oct 2020

Abstract

Real-time paradigms play a crucial role in our society since an increasing number of complex systems rely on dependable computing. From those complex systems, mixed-criticality and high availability distributed systems strongly rely on deterministic time-triggered communication to ensure real-time behavior. In time-triggered distributed systems, a common approach utilizes a periodically repeating communication schedule. Finding such a schedule needs an intuitive approach due to the problem's complexity class.

This thesis focuses on the time-triggered scheduling problem emerging within the distributed modular power stress (MoPS) system. Here, the production as well as consumption of messages is carried out within software tasks running on distributed modular targets. A customized approach is presented for modeling the project specific requirements into a concrete less complex optimization problem. The proposed approach aims in minimizing the end-to-end latency among the distributed targets, respecting the precedence constraints and the system parameters. As a consequence, the optimal schedule time-line is obtained where the instants, at which information is delivered or received, are computed in such a way that all the modeled constraints are satisfied.

In the course of this thesis the design framework was elaborated from scratch, by which the precedence relations between the real-time tasks can be designed. Such a framework ensures, that the flow of the design procedure is error resistant as much as possible. Therefore, a complete tool along with its graphical user interface (GUI) has been implemented, where all the system constraints are considered and several verification paradigms are integrated.

Acknowledgements

First and foremost, I would like to express my gratitude to my supervisor at KAI GmbH, DI Sascha Einspieler for giving me the opportunity to work in this futuristic project. His support and guidance has been instrumental in this work. I would also like to thank Prof. Edgar Ernesto Sanchez for assessing my work continuously and providing me with his invaluable feedback. I am also thankful to Dr. Benjamin Steinwender for his suggestions regarding my work and his vivid explanations of different topics.

Last but not the least, I would like to thank my colleagues at Kompetenzzentrum Automobil- und Industrie-Elektronik (KAI), my friends and my family for supporting me throughout this project and for keeping me motivated.

Contents

List of Figures

List of Tables

CHAPTER 1 Introduction

Today's market challenges the semiconductors production in several demands, like decreasing time-to-market, increasing manufacturing robustness and quality requirements [1]. On the other hand, the design complexity of the produced System on Chips (SoCs) has increased. Therefore assuring dependable systems, requires improvements in the test paradigms already available. One of those paradigms, is the so-known as *Burn-in* or *Stress Test*. The latter is the process by which the device under test (DUT) is exercised before launched into market. Thus trying to force certain failures to occur, under supervised stress conditions. Burn-in aims at accelerating detection of so-called *infant mortalities* [2], as well as formulating statistical performance sheets. Such a system, called MoPS test system, is being developed at KAI. The MoPS architecture works completely in a modularized approach from both SW and hardware (HW) point of views [3]. Figure 1.1 shows the architecture of the MoPS system. A basic review, of the architecture of the MoPS system, is elaborated in Section 2.3.

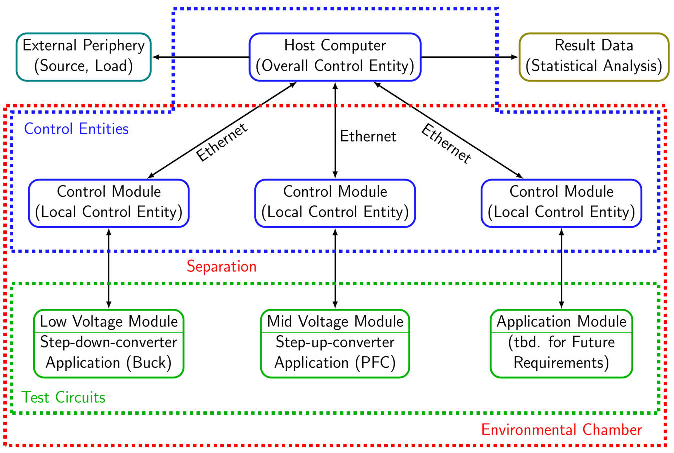

External Periphery (Source, Load)
Host Computer (Overall Control Entity)
Result Data (Statistical Analysis)
Control Entities
Ethernet
Ethernet
Ethernet
Control Module (Local Control Entity)
Control Module (Local Control Entity)
Control Module (Local Control Entity)
Separation
Low Voltage Module Step-down-converter Application (Buck)
Mid Voltage Module Step-up-converter Application (PFC)
Application Module (tbd. for Future Requirements)
Test Circuits
Environmental Chamber

Figure 1.1: MoPS architecture.

The MoPS holds a distributed communication network, which makes it superior in terms of flexibility to a centralized system [3]. For further extending this flexibility, real-time behavior has to be an option for building the test plan of the MoPS system. There are two major design paradigms for implementing real-time behaviors, the event-triggered and the time-triggered approach [4]. In a time-triggered distributed system, communication takes place according to a common periodic communication schedule. This thesis is concerned in designing a project specific tool, for generating the schedule of a distributed real-time communication. The name given for the tool is Time-Driven Communication Schedule Planner (TDCS-Planner). Therefore, TDCS-Planner refers to the developed tool, within the course of this thesis.

1.1 Motivation

The non real-time behavior of a test procedure, in the MoPS system, resembles the finite-state machine (FSM) model [3]. The design of the test plan is done by the test plan builder (TP-Builder), already developed [5], which aims to ease drawing of the interactive FSM diagrams for every layer of the MoPS architecture. Until this point, the test procedure happening on multiple targets has a non real-time behavior, and is composed of event-triggered tasks. On the other hand a real-time extension of the MoPS system, is being researched within a PhD thesis topic. The PhD thesis focuses on a mixed-triggered communication approach, which provides real-time behavior under normal conditions while it flexibly transitions to a fallback mode when temporal boundaries are not met [6]. Therefore, there is a need for a SW containing a test plan builder for real-time tasks, and also a scheduler for managing the distributed real-time communication.

Scheduling, in a distributed real-time system, is affected significantly by inter-task communication and hence, a pre-runtime task allocation algorithm is needed [7], which takes into consideration the real-time constraints. Moreover, the generated test plan should be as error resistant as possible. This results in the implementation of several sanity checks during the whole building procedure, and even over the generated schedule script which describes the test plan.

1.2 Goals

The objective of this thesis can be summarized into two key goals. First, to simplify the creation of a test plan occurring between distributed targets, and generate the optimal schedule time-line. Second, to verify the sanity requirements of such a designed test plan. Thus, an intuitive application along with its GUI has to be provided. The latter should be accountable of the following:

- Fetching the data files of the available type of targets, for the MoPS system, and represent them in a user friendly manner.
- Providing a design interface for the inter-task communications, in the form of simple data dependency graphs.
- Verifying the correctness of the built design, and formulating it into a well structured model.
- Generating the optimal schedule, while considering all the transmission delays which are network dependable.
- Visualizing the generated schedule time-line in a smooth representation.
- Outputting both the test plan description and the schedule script in JavaScript object notation (JSON) format, respecting their defined schemas. Also, providing complete correctness check for the last-mentioned files.

The application is completely developed in *Python 3* language. Thus, it is benefitted from all the relevant available Python modules. For the GUI representations, *PyQt5* libraries and modules, are used for developing the tool. Section 2.7 presents a review of the available programming languages and alternatives that could have been chosen.

1.3 Problem Statement

This thesis emerges from the need of a real-time communication scheduling tool, for the time-triggered extension of the MoPS test plan. Therefore, the problem statement is divided into three areas:

- How to represent the creation process of the test plan, to provide an intuitive user friendly interface?
  - How to visualize the type of targets, with all its description?
  - How to construct the dependency graph of the inter-task communications?
  - What schema to follow for better non-redundant description of the test plan file?

- What scheduling algorithm should be used, for fulfilling the project specific requirements?
  - How to integrate the communication delays in the scheduling problem?
  - How to benefit of maximum Central Processing Unit (CPU) utility?
  - How to visualize the resulted schedule time-line, in an intelligible way?
  - What schema to follow for better non-redundant description of the schedule script file?
- How to make sure that the test plan is as error proof as possible?
  - How to prevent errors during the building process of the test plan?
  - How to re-check for errors in a test plan description?
  - How to re-check for errors in a schedule script file?

1.4 Thesis Outline

This section gives some reading advice, for easing the understanding of this thesis work. The introduction is exhibited in Chapter 1, where the motivation, goals, and problems statement are elaborated. It is highly recommended for getting a basic overview of the direction of this thesis. A literature review is presented in Chapter 2, for emphasizing on some concepts that are used during the course of this thesis. The latter explains:

- The different scheduling paradigm concepts, which are necessary for understanding the need of implementing a project specific scheduling algorithm.
- The overview of the parent project MoPS, from which this thesis emerges. This part is a must for getting a complete picture of the system model, later explained in Chapter 3.
- SW and mathematical basic concepts, used within the modeling of the problem statement researched by this thesis. These concepts can be skipped, if there is already a relative background.

Chapter 3 explains the system model, and the paradigm followed behind the intuitive scheduling algorithm. Chapter 4 exhibits the full creation of the application, from both core and GUI aspects. Finally, an evaluation of the implementation is done in Chapter 5 and the conclusion with the outlook, follows in Chapter 6.

CHAPTER
2 Literature Review

2.1 JavaScript Object Notation

JSON is a lightweight data-interchange format. It uses human-readable text to store and transmit data objects consisting of attribute–value pairs and array data types[1]. JSON is becoming a clear choice for mainstream data applications [8]. All the data files of the TDCS-Planner are represented in JSON format, because of the last-mentioned advantages. Therefore, a schema for each of the used files is written, following the JSON-schema draft *Draft 2019-09*[2]. Listing 2.1 shows an example of JSON format representation. Moreover, Python includes predefined modules which allows to generate and parse JSON-format data, this eases the coding developing.

2.2 CAN Bus

CAN is a multiplexed serial communication channel, used for data transfer among distributed electronic modules. It emerged as the standard in-vehicle network. CAN has several different physical layers. These physical layers classify certain aspects of the CAN protocol, such as electrical levels, signaling schemes, cable impedance, maximum baud rates, and more[3]. This section exhibits a brief explanation of the high-speed physical layer.

The distributed communication of the targets layer, in the MoPS system, is of CAN type. Specifically, high-speed CAN is being used. Nevertheless, for better maintainability of TDCS-Planner, the system model deals with the communication network in an abstract methodology, as described in Chapter 3.

[1]JSON data interchange: http://json.org
[2]Specification page: http://json-schema.org
[3]White papers: Controller Area Network (CAN) Overview, http://ni.com

High-speed CAN networks are implemented with two wires and allow communication at transfer rates up to *1 Mbit/s*. These two wires resemble two signals, *CANL* and *CANH*, either driven to a *dominant* state with *CANH (5V)* > *CANL (0V)*, or not driven and pulled by passive resistors to a *recessive* state with $CANH \leq CANL$. A 0 data bit encodes a dominant state, while a 1 data bit encodes a recessive state, supporting a wired-AND convention, which gives nodes with lower ID numbers priority on the bus.

Each node, of the network, requires: a CPU, CAN controller, and transceiver. The latter converts the data stream from CAN bus levels to levels that the CAN controller uses. Any node is able to send and receive messages, but not simultaneously.

A message or Frame consists primarily of the ID (identifier), which represents the priority of the message, and up to eight data bytes. Other overhead are also part of the message. Two different message formats can be used: using standard frames with 11bit identifiers (known as CAN 2.0 A) and extended frames with 29bit identifiers (known as CAN 2.0 B). Except for the different length of the identifiers, the messages are built up equally. The MoPS system uses the standard frames. Figure 2.1 illustrates the CAN data frame sections.

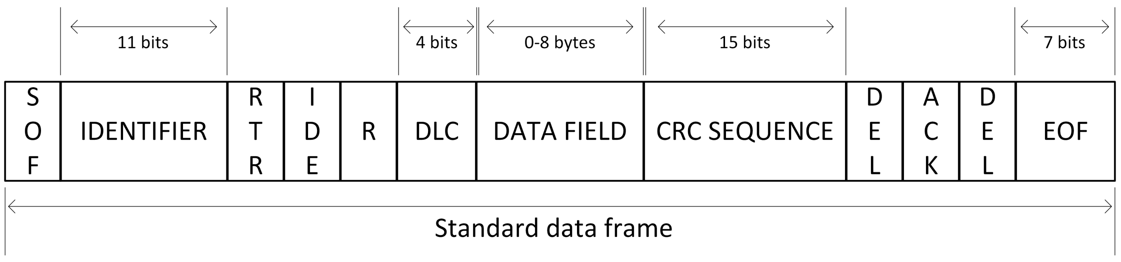


Figure 2.1: Standard CAN data frame.

SOF Start Of Frame - 1 bit.

IDENTIFIER A message identifier sets the priority of the data frame - 11 bits.

RTR Remote Transmission Request, defines the frame type (remote or data frame) - 1 bit.

IDE Identifier Extension - 1 bit.

R Reserved bit - 1 bit.

DLC Data Length Code, number of bytes of data - 4 bits.

DATA FIELD Data to be transmitted - 0-8 bytes.

CRC SEQUENCE Cyclic Redundancy Check - 15 bits.

DEL CRC delimiter - 1 bit.

ACK Acknowledgement - 1 bit.

DEL ACK delimiter - 1 bit.

EOF End Of Frame - 7 bits.

2.3 MoPS

The MoPS test system is a distributed test system. The system architecture is shown in Figure 1.1. MoPS design aims in providing a flexible infrastructure for customizable stress test applications. It is capable of running different test applications with a common base framework. The test engineers and test operators have to configure a simple comprehensive system, instead of having to manage different test systems [9].

To handle the complex requirements, MoPS is split up into hierarchical entities. The overall control entity, the local control entity, and the application entity.

Overall Control Entity Contains the host computer, which controls the overall test flow and communicates with the control modules. It also manages the external periphery and stores the measured data into the file system.

Local Control Entity Contains control nodes, which may be many (typically 8 to 24 for one test system) and are connected to the host computer via Ethernet.

Application Entity Contains application modules, which are referred as targets within the scope of this thesis. Each target is connected to one control module, from the local control entity. The target executes the test, drives and monitors a DUT. All the targets share a CAN communication.

Both the control nodes and targets are typically placed within an environmental chamber. Only the host and the external periphery are placed outside. The essential advantage of this test system architecture is the separation of the control and data acquisition parts from the actual test circuit. Therefore, only the application entity has to be redesigned, when changing the type of test performed. This saves development effort, design time and provides a unified data acquisition and control methodology.

2.3.1 Targets Configuration

The targets are tailored to individual types of tests. Thus, every target is described within a configuration file, called *Targets Configuration*. Listing 2.1 shows a configuration of an example target. The *Targets Configuration* states the following:

- Name of the target description.
- Both SW and HW versions of the target.
- List of the available tasks, which the target can execute.

Such files are described in JSON format. Therefore a schema respecting the description, is developed for checking the correctness of the file.

2.4 Scheduling Paradigms

For a given set of tasks, the general scheduling problem can be understood as the problem of finding an order, according to which the tasks are to be executed such that various constraints are satisfied. Typically, a task is characterized by its execution time, ready time, deadline, and resource requirements. Tasks can have a variety of interpretations, depending on the field of study. For the scope of this thesis, task resembles a computation that is executed by a CPU in a sequential fashion, with an access to a communication network.

There exists an extensive literature on the topic of scheduling theory. This section presents a small part of that extensive literature. Only the methodologies, which are relevant to the research questions of this thesis, are discussed. The scheduling problem variants can be summarized as follows:

Machine Environment Tasks can be scheduled using a *mono-processor, multi-processor* or *distributed* approach. The tasks of the MoPS system are each linked with a single target and the targets share a distributed communication network, therefore, the scheduling problem contains both *mono-processor* and *distributed* approaches.

Release Time In the static case, all the tasks are given and ready to run. In the dynamic case, new tasks may arrive at any time during the execution. The test plan, of the MoPS system, is completely designed and planned, by the test engineers, before runtime. Therefore, the task set of the scheduling problem is considered static.

Listing 2.1: µMoPS_v5 configuration file snippet.

```
{
    "name":"uMoPS_v5",
    "hw":"MOPS-2h3j4h5j",
    "sw":"CORE-joh345jk",
    "tasks": [ {
        "name":"setPwmFreq",
        "wcet":12,
        "in":[ {
            "label":"freq_hz",
            "type":"int",
            "default":0,
            "min":0,
            "max":1e6
        } ],
        "out":[]
    },{
        "name":"getPwmFreq",
        "wcet":9,
        "in":[],
        "out":[ {
            "label":"freq_hz",
            "type":"int"
        } ]
    },{
        "name":"add",
        "wcet":7,
        "in":[ {
            "label":"a",
            "type":"int",
            "default":0
        },{
            "label":"b",
            "type":"int",
            "default":0
        } ],
        "out":[ {
            "label":"sum",
            "type":"int"
        } ]
    } ]
}
```

Execution Time When dealing with real-time systems, execution times of tasks have to be known a priori, or a certain estimation has to be computed. For the scope of this thesis, Worst Case Execution Time (WCET) is supposed as a given parameter, fetched from the *Targets Configuration* files. Another project is emphasized in such execution time estimations, for the real-time extension of MoPS [10].

Preemption The execution time, of a task, might or might not be interrupted, thus, the terms preemptive and non-preemptive scheduling. For the MoPS system, the tasks are not preemptive, therefore, the scheduling problem is non-preemptive.

Dependecies Over the set of tasks, there might be precedence relation which constraints the order of execution. This is the case for the task set of the MoPS system.

Periodicity The tasks can be periodic, and therefore it is better described as a job. Otherwise, the tasks can be aperiodic. In the MoPS system, the used tasks are considered aperiodic.

Resources Allocation of resources over time, to perform the task set. In the MoPS system, the shared resources are: the CPU between the tasks happening in the same target, and the communication network between the distinct targets.

Online/Offline If release time is dynamic, then an online scheduling is required. Otherwise, if release time is static, then an offline scheduling can be done. In the MoPS system, all the information of the test plan is known a priori, before runtime. Thus, the schedule is generated offline.

2.4.1 Scheduling Problem

The key parameter of any scheduling problem is *time*. A task should be completed before its deadline, which in general is known a priori. Moreover, over the set of tasks, there is precedence relation which constraints the order of execution. Real-time scheduling refers to the case in which each task has its individual offset time (release time) and end time (deadline time) [11].

In general, to form a scheduling problem, three sets are needed: a set of n tasks $\Gamma = \{\tau_1, \tau_2, ..., \tau_n\}$, a set of m processors $P = \{p_1, p_2, ..., p_n\}$, and a set of k types of resources $R = \{r_1, r_2, ..., r_n\}$. In addition, precedence relations among the tasks can be defined through a Directed Acyclic Graph (DAG), and the timing parameters are associated with each task. Scheduling, thus, means assigning processors from P and resources from R to tasks from Γ, such that all tasks are completed respecting the imposed constraints [12]. This problem, in its general form, has been shown to be *NP-complete*, and hence computationally intractable [13].

2.4.2 Scheduling Algorithms

The scheduling problem's variants are typically used to classify the various scheduling algorithms. From those variants, the periodicity is the main classification, which encircles all the other variants. Therefore, only the aperiodic scheduling algorithms are discussed. In this context, the algorithm, fulfilling the requirements of this thesis, should consider:

- Non-preemptive
- Static
- Precedence relations
- Offline
- Optimal

A description, which serves as a basis for the classification scheme, is used from [13]. Such a notation uses three fields ($\alpha \mid \beta \mid \gamma$). Where α states the machine environment *(mono-processor, multi-processor, distributed)*, β describes tasks and resource characteristics *(preemptive vs. non-preemptive, independent vs. precedence constraints,...)*, and γ indicates the criterion of the objective function.

Earliest Due Date

The scheme this algorithm considers is $1 \mid SYNC \mid L_{max}$. That is, the set of tasks have a synchronous release time, and have to be scheduled on a single processor, minimizing the maximum latency. This algorithm was found by Jackson in 1955, which can be summarized as: execute the tasks in order of non-decreasing deadlines. Earliest Due Date (EDD) is proven to be optimal with respect to minimizing the maximum latency [13]. The complexity of EDD is $O(n \log n)$. No other constraints are considered, by EDD, hence tasks cannot have precedence relations and cannot share resources.

Earliest Deadline First

The problem this algorithm considers is $1 \mid PREEM \mid L_{max}$. Tasks in this case, are not synchronous, and can have dynamic arrival times and preemption is allowed. Earliest Deadline First (EDF) was found by Horn in 1974, it can be summarized as: at any instant, execute the task with the earliest absolute deadline among the ready tasks. EDF is proven to be optimal with respect to minimizing the maximum latency [13]. The complexity of EDF, in this case per task, is $O(n \log n)$, if ready queue is a heap, or $O(n)$, if ready queue is a list. No other constraints are considered, by EDF in this case, hence tasks cannot have precedence relations and cannot share resources.

Bratley's

This algorithm considered the scheme $1 \mid NO\_PREEM \mid feasible$. Which is to say, a set of non-preemptive tasks with arbitrary arrivals times, have to be scheduled on a single processor. Bratley proposed this algorithm in 1971. Instead of the exhaustive search, Bratley's uses a pruning technique to determine when a current search can be reasonably abandoned. The worst case complexity of Bratley's is $O(n.n!)$. It can only be run off-line, which is the case for time-trigger systems but it is not optimal with respect to minimizing maximum latency.

Spring

This algorithm aims in finding a feasible schedule when tasks have different types of constraints, such as precedence relations, resource constraints, arbitrary arrivals, non-preemptive properties, and importance levels. It was designed by Stankovic and Ramamritham and is used in distributed computer architectures. Therefore, Spring deals with *NP-hard* problems [13]. Spring is not optimal, then if there is a feasible schedule, Spring may not find it.

Latest Deadline First

The scheme this algorithm considers is $1 \mid PREC, SYNC \mid L_{max}$. That is, minimizing maximum latency of a set of tasks, with precedence relations and synchronous arrivals, scheduled over a single processor. Lawler proposed Latest Deadline First (LDF) in 1973. LDF builds the scheduling queue from tail to head, by picking the latest deadline leaf of the DAG, to be scheduled last. In this context, the first task inserted in the queue is executed last. The complexity of LDF is $O(n^2)$, and it is proven optimal with respect to minimizing maximum latency [13].

EDF Precedence Constraints

This algorithm considers the problem $1 \mid PREC, PREEM \mid L_{max}$. Thus, scheduling a set of tasks with precedence relations and dynamic activations. Modified EDF was proposed by Chetto, Silly, and Bouchentouf in 1990. The basic idea is to transform the set of dependant tasks into another set of independent tasks, by an adequate modifications of the timing parameters of each task *(release times and deadlines)*. The transformation algorithm ensures optimality with respect to minimizing maximum latency [13].

Scheduling Time-Triggered Communication

In time-triggered distributed systems, communication takes place according to a common periodic communication schedule [4]. Therefore, discrete time slots are assigned for each node, where it is allowed to send and receive messages. This provides an isolation between different functionalities and guarantees a deterministic latency, but makes the scheduling problem the bottleneck of the system. The MoPS system, containing a time-triggered distributed communication, compels a project-specific scheduling algorithm. Such an algorithm is developed within this thesis, benefiting of the concepts offered by others algorithm, which do not completely consider the requirements.

2.5 Computational Complexity Theory

In computational complexity theory, decision problems are classified based on the hardness of the problem. This section gives a brief explanation of the fundamental complexity classes.

P Problems

This kind of problems refer to all the decision problems that can be solved using a polynomial amount of computation time. Therefore, they are called *Polynomial Time Problems*. P class problems are efficiently solvable by a deterministic Turing machine.

NP Problems

This kind of problems cannot be solved in polynomial time. However, they can be verified in polynomial time. That is to say, NP class problems are solvable in polynomial time by a non-deterministic Turing machine [14]. P class problems are always also NP class.

NP-Hard Problems

A decision problem is said to be NP-hard, if and only if every problem in NP is reducible to it, in polynomial time.

NP-Complete Problems

NP-complete problems are the hardest between the NP problem class. A problem is classified as NP-complete, if it is both NP and NP-hard.

The real-time scheduling problem that this thesis is dealing with is of type NP-complete. As a consequence, one of the research objectives is to identify a simpler and practical approach for finding a solution for such a problem.

2.6 Time Utility Function

Time Utility Function (TUF) are created to quantify the utility of completing each task at a given time. This concept aims in extending the pure deadline based real-time specification by a generic function. The utility functions used in the scope of this thesis, are considered to be monotonically decreasing. In this context, a TUF returns a high value if the task is scheduled with minimum latency, and a low value if a task is scheduled with maximum latency. TUFs help in quantifying the time parameters assigned for every task, both the offset (release time) and end time (deadline time). Therefore, TUFs are essential in modeling the objective function of the scheduling problem of this thesis, which is fully described in Chapter 3.

2.7 Programming Languages

When developing software, there are multiple factors that have to be considered for choosing the convenient programming language. This section highlights some of those factors for different programming languages.

Java

Java is an object-oriented programming language that can be written on any device. It is platform independent in both binary and source level. Java is well known for its feature of safety which can disrupt corruptions or errors. Java requires high storage capacity and uses more memory, thus, it becomes slower in performance compared to other languages.

Ruby

Ruby is a pure object-oriented programming language, everything appears to Ruby as an object. It is simple since it consists of easy and understandable syntax. Ruby on Rail is quite popular for web development. However, for developing desktop applications, Ruby is not so popular in the community. For big applications, Ruby is said to have a slow runtime speed.

C/C++

C is one of the most difficult programming languages for software development. It is known to be the building block of many other languages used today. C++ is an object-oriented programming language, it offers the feature of platform independence. Also, since C++ is closely associated to C, then it allows low-level manipulation. However, C++ is still considered harder than other languages, it lacks the feature of garbage collector to automatically filter out unnecessary data.

Python

Python is an interpreted programming language used for general-purpose programming. With a simple syntax, Python has automatic memory management and dynamic features that make it suitable to be used in a variety of applications. It offers a lot of third-party modules, which can ease in time the software development process. For Python everything is an object. Python is easy to read and learn, and it is one of the world's most popular programming languages. Therefore, there is a big community of Python developers, which is always of benefit for any software development.

Python language is chosen for developing the TDCS-Planner. The version of Python used is *Python 3.8.2*. TDCS-Planner is part of a bigger project, therefore future enhancements and integrations are probable. The maintainability and the increasing community of Python, makes such integrations easier.

2.7.1 Graphical User Interfaces

Python offers multiple options for developing a GUI. This section exhibits the top GUI tool-kits with a brief overview on each.

TKinter

TKinter is an open source and standard GUI toolkit for Python. TKinter is a wrapper around tcl/TK graphical interface. It is popular because of its simplicity and having an old active community. TKinter is portable for Macintosh, Windows and Linux platforms. However, it is mostly preferred for small scale GUI applications.

PyQt

PyQt toolkit is a wrapper around QT framework. PyQt is one of most popular cross platform Python binding over C++ implementing QT-library for QT framework. PyQT can be used for large scaled GUI application. The design can be done with the use of QT designer and convert the *.ui* files to Python code. However, writing code manually ensures better control of the code structure and representation.

PySide

Just like PyQt, PySide is also a Python binding of the cross-platform GUI toolkit Qt. The two interfaces were comparable at first but PySide ultimately development lagged behind PyQt.

For developing the GUI of the TDCS-Planner, PyQt is chosen. The version of PyQt used is *PyQt5.15.0*.

CHAPTER 3

System Model

This chapter proposes a system model for the scheduling problem, which is invoked by the MoPS system. First, an overview of the *general process* is presented. Then, the modeling of the *raw data* structure is exhibited, along with the basic concepts of the *tasks* classifications. Afterwards, the *system complications* are fully described by analyzing a case study, before explaining the procedure of *data analysis*. Later on, the *time-line model* is shown, containing all the timing representations. In this context, the *task set transformation* comes as a consequence of the last-mentioned sections. Finally, the complete formulation of *optimization problem* is presented.

3.1 General Process

This thesis presents an approach to minimize the end-to-end latency of tasks, occurring between nodes of the MoPS system. The approach respects the precedence constraints as well as the communication network dependencies. The general process is based upon what is presented in [15], but with major modification to fit the MoPS system specifications. The main key is in the task set transformation from which an Integer Linear Programming (ILP) optimization problem is formulated to find the feasible task set. The optimality criteria is based on the TUF assigned for each task, specifying the task tolerance towards latency. As a result, the optimal task set is found, rather than searching for a final optimal schedule among all the possible search space.

The general process forming the whole approach is illustrated in Figure 3.1. The test plan is built within the *Builder View*, described in Section 4.2.2. The *Builder Core*, therefore, outputs a *description file*. This latter undergoes a complete *sanity check*, described in Section 4.3.4. If the sanity check is passed, then the *raw data* is extracted and structured into a parametrized *task set*, which is referred to as Γ. This task set is then reformulated into another *task set* dictionary, that is called Γ_{form}. The reformulation process is done by running a full *data analysis*, covering every aspect of the constraints forced over the MoPS system. Hereafter, the *optimization problem* is constructed by modeling Γ_{form} into a concrete objective function with all its corresponding inequality constraints. The *optimization problem* is thus given to an *Optimizer*, which solves ILP problems. In this context, the search space is narrowed due to the restricted input domains of offsets and deadlines, thus, finding the optimal values for the *decision variables* is simpler. The *decision variables* are then processed into a static schedule time-line, which is optimal with respect to minimizing maximum tasks' latency.

This chapter explains the logic behind all the modules which appear after the *sanity checks* block, in Figure 3.1. The blocks, appearing before *sanity checks*, are covered in Chapter 4.

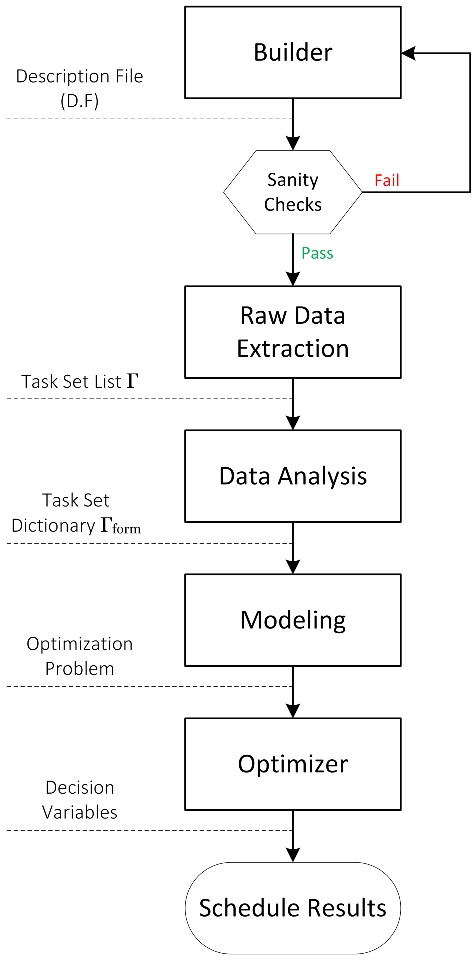


Figure 3.1: General process flow chart.

3.2 Task Model

3.2.1 Raw Data Representation

The task set Γ is extracted from the description file, if the latter respects the builder sanity requirements described in Section 4.3.4. Every task τ_i in Γ is represented by the following tuple:

$$\tau_i(\ ID_{target},\ ID_{task},\ WCET,\ inputs,\ outputs\ ) \tag{3.1}$$

Where ID_{target} states the parent target of the task. Since the CAN bus ID is an unique integer for every target, then it is used as a target identifier. ID_{task} states the identifier of the task for distinguishing it between its twin siblings (tasks from same parent target). This integer is assigned by the *Builder Core*, and serves as a unique identification. $WCET$ states the worst case execution time of the task. This value is fetched from the *Targets Configuration* files. *inputs* is a list of tuples representing every input of the task as follows:

$$in(\ type,\ ID_{task},\ name\ ) \tag{3.2}$$

Where *type* states the type of the input, which can be either *int, float* or *bool.* ID_{task} here states the identifier of the task that is feeding this input, notice that it is only *one* integer since an input can be fed only from one source. *name* states the name of the output port (or variable) from which this input is getting its value.

outputs from (3.1) is another list of tuples representing every output of the task as follows:

$$out(\ type,\ IDs_{task}\ ) \tag{3.3}$$

Where *type* states the type of the output, which again can be either *int, float* or *bool.* IDs_{task} is a list of integers representing the tasks this output is feeding. Notice that they can be many tasks since an output can feed several inputs at once.

Γ is called the *raw data set* since it has no transformations and no data analysis is applied on it. Γ solely describes the complete built scenario of interconnected tasks, required in the test plan. The main concern, while formulating Γ, is to have the enough information for proceeding with the data analysis. It is why, *inputs* and *outputs*, in (3.1), are represented differently though they both describe connection ports (or variables). Redundancy, in this case, is completely avoided for reducing data structures. Subsequently, it is explained how every argument, of the data tuples, serves a purpose when it comes to data analysis.

3.2.2 Task Types

Before describing the data analysis, it is crucial to understand the fundamental classification of tasks. Differentiation is made between three fundamental task types: *free tasks, consumer tasks* and *producer tasks.*

Free Tasks

These type of tasks are the tasks which are completely independent from the communication network. However, these tasks can have a generic number of data dependencies, either waiting some input arguments for execution, or producing some output arguments for other task to execute. The key of this type is in having the data dependencies coming from sibling tasks (from same parent target), thus, not passing through the communication network.

Consumer Tasks

These type of tasks are the tasks which have input data dependencies coming from the communication network. Thus, *consumer tasks* have to wait the desired input value, for the complete transmission to finish. Only after transmission is received, *consumer tasks* can start execution of their function.

Producer Tasks

These type of tasks are the most critical for the scheduling problem. Subsequently, it is explained how *producer tasks* define the *time slots* of the schedule time-line. After finishing execution, *producer tasks* produce an output which is transmitted through the communication network. This output is consumed by the above explained *consumer asks.* Therefore, both the execution time plus the transmission time formulate an output dependency for the *producer tasks.*

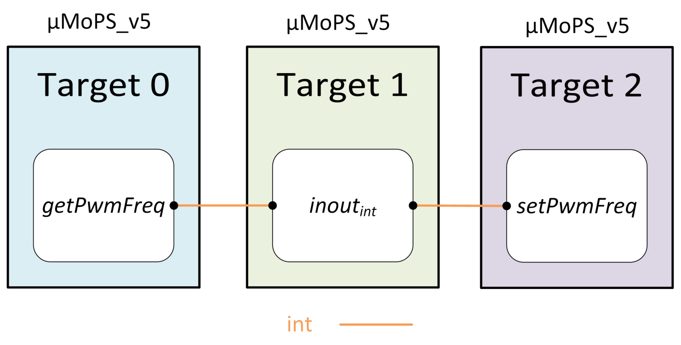


Figure 3.2: Producer (*left one*), consumer & producer (*middle one*) and consumer (*right one*) tasks.

Note that the last two types of task, described above, can happen together, i.e. a task can be both *consumer* & *producer*. Thus, the task is said to have both input and output dependencies with the communication network. Figure 3.2 illustrates three types of tasks. Going from left to right: *producer task, consumer* & *producer task* and *consumer task.* Notice the difference between the parent targets, which means any connection between tasks passes through the communication network first.

Moreover, if a task is *consumer* & *producer*, then it is necessary for it to consume first, and then produce. Otherwise, it will introduce a contradiction in the model, when scheduling the task for minimum delays. In case it is desired for a task to produce, and then consume, hence, it must be split into two separate tasks. One *producer* and another *consumer*, as it is in the case of the MoPS system.

3.3 System Complication

The scheduling problem, studied in this thesis, arises within the MoPS system. In such a system, a CAN bus is shared between all nodes of the system, these nodes are referred to as targets. Therefore, there is a need of managing the access to the communication network. Each target, in the distributed system, is a micro controller (µC) directly controlling the DUT.

Figure 3.3 illustrates the CAN bus shared between the targets of the system. Notice that targets can be different from each others, the labels over the µCs, state the type of the target. Every type of target has its own set of tasks that can be executed. The functionality, of each task, varies along with its execution time WCET. The number of inputs and outputs, of each task, vary as well. As an example, the task set of the target type *µMoPS_ v5*, is summarized in Table 3.1. This task set is taken from the *Targets Configuration* file, described in Section 2.3.1.

The name, of each task, reflects its functionality. This functionality is defined depending on the design of the *application entity*, described in Section 2.3. Therefore the number of arguments of the task is generic. On the other hand, the origin source of the arguments value is crucial for the model. Some tasks can be completely independent from the network, for instance *getTemp*, can be used for monitoring purposes, and thus, no need to neither read nor write on the communication network. However, some tasks can be dependent on the network, either by reading, or writing, or even both. For instance *inout*, can be used for controlling a sequence of states between all targets involved in the test plan. Suppose a specific target has to wait a signal, from other target, before starting

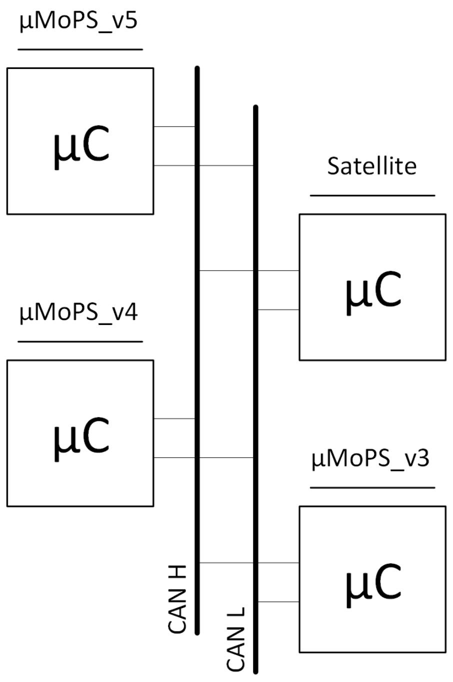


Figure 3.3: CAN bus communication network between targets.

execution, and also has to trigger the successive target whenever it finishes. When these sequences increase in number, the problem becomes very complicated, and in its general form has been shown to be NP-complete [16].

Moreover, any argument received or sent on the network needs a transmission time interval. Computing this time, mainly depends on the type and number of arguments, which are generic as previously stated. Therefore, the transmission time is generic as well and has to be considered by the model. Also, baud rate of the network and overhead of the protocol are considered, but outside the scope of the model, since they are abstract parameters.

Determinism is the key concept of real-time behavior, this idea is exploited by defining discrete *time slots* where tasks are allowed to execute. Another evident aspect is the CPU utility, which adheres to scheduling within the *time slot* itself.

For summing it all together, a simple scenario, formed by two targets, is demonstrated for highlighting the complication. *Target 1* has four tasks to execute, two of them are *free tasks* and the other two are *producers*. *Target 2* has two tasks to execute, both being *consumers*. Figure 3.4 illustrates the latter explained in a graphical way.

For simplicity, the following suppositions are made:

Table 3.1: Task set of µMoPS_v5.

Tasks	**Inputs**	**Outputs**
$setPwmFreq$	int	–
$getPwmFreq$	–	int
$setAO$	$float$	–
$getTemp$	–	$float$
$gpioWrite$	$bool$	–
add	$int\&int$	int
in_{int}	int	–
out_{int}	–	int
$inout_{int}$	int	int
in_{bool}	$bool$	–
out_{bool}	–	$bool$
$inout_{bool}$	$bool$	$bool$
and_{bool}	$bool\&bool$	$bool$
in_{float}	$float$	–
out_{float}	–	$float$
$inout_{float}$	$float$	$float$

- Both targets are of the same type *µMoPS_v5*, thus, the used tasks are from the task set shown in Table 3.1.

- All WCETs of the used tasks are equal.

- Transmission time of an *int* payload is equal to WCETs of the used tasks.

The case study gives better understanding of data dependencies between *free tasks* and network dependencies between tasks of different targets. Figure 3.5 shows a timeline demonstrating the optimal schedule solution, with respect to minimum latency period and maximum utility of CPU.

As mentioned previously, transmission time is completely generic but in this case, it is supposed equal to the WCET. In this context, this time interval can be used for executing another task, knowing that transmission is done by a peripheral module and not the CPU itself. The latter pattern is also considered by the model. Moreover, notice that in_{int} is pushed after out_{int} on *Target 1*, though the data dependency of in_{int} is already fulfilled. This is due to the sequence of tasks that a *producer* carries ahead of it, which makes them crucial for the model.

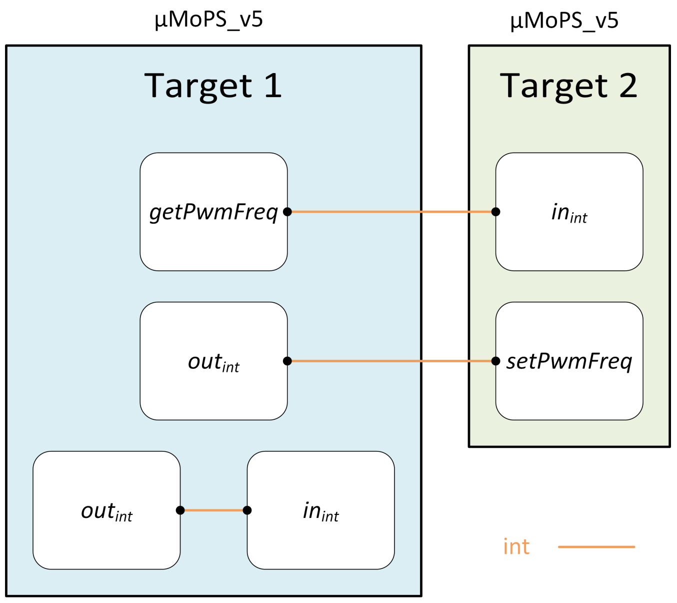


Figure 3.4: Case study between two targets.

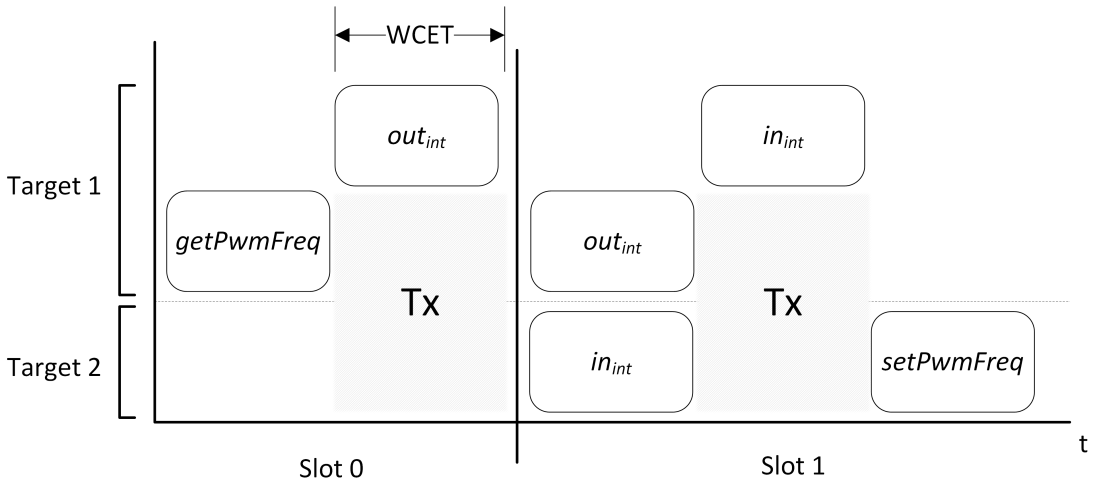


Figure 3.5: Case study schedule time-line.

Finally, the declaration of *time slots* adheres to the *producer tasks*. Note that, reading from the network is always ongoing by all targets distributed on the network. In this context, *time slots* are defined, as the time windows where tasks are allowed to write over the network. Consequently, determinism is exploited, and thus, tasks are executed in well defined discrete time windows.

3.4 Data Analysis

The task set Γ, described in Section 3.2.1, is analyzed in order to get a formulated task set Γ_{form}. First, the tasks coming from the same targets are grouped. Thus a dictionary is obtained having the ID_{target}'s as key entries. The value entries are, thus, lists of tasks contained by the respective ID_{target}, as follows:

$$\begin{gathered} \{\, ID_{target_i} \;:\; [\, \tau_j(\, ID_{task},\; WCET,\; inputs,\; outputs\,), \ldots], \ldots \} \\ 1 \;\leq\; i \;\leq\; n \\ 1 \;\leq\; j \;\leq\; m_i \end{gathered} \tag{3.4}$$

Where n is the number of used targets, and m_i is the number of children tasks of the i_{th} target. The above obtained dictionary, helps is distinguishing between the data dependencies. Weather, the dependency, comes from the communication network or internally from another sibling task (from same parent target).

3.4.1 Internal Dependencies

Internal dependencies are the data dependencies originating within a target itself, that is to say, if input arguments, of a task, come from another sibling task belonging to the same target. For instance, in Figure 3.4, out_{int} and in_{int} have an internal data dependency within *Target 1*.

For avoiding redundancy, internal dependencies are defined relying only on the *inputs* of the task. Since it is verified, that every *input*, of a task, is connected to another *output*, of another task, as per the sanity requirements of Section 4.3.4. Hence, the internal dependencies of *outputs* are intrinsically considered as well. For every task an *interDependencies* list is defined. *interDependencies* is built going through all inputs, of a task, and check if the ID_{task} (origin source of the input) belongs to the same tasks' group. Algorithm 1 expresses what is formerly described. In other words, the *interDependencies*, of a task, states the ID_{task}'s, for which this task has to wait before being able to execute.

3.4.2 Network Dependencies

Network dependencies are data dependencies originating within distinct targets, that is to say, if input arguments, of a task, come from another task which belong to another

```
Data: $Dictionary$ : targets' groups
Result: $interDependencies \leftarrow [\,]$

foreach $(ID_{target}, taskSet) \in Dictionary$ do
    foreach $task \in taskSet$ do
        foreach $input \in task \rightarrow inputs$ do
            if $input \rightarrow ID_{task} \in taskSet$ then
                $interDependencies \leftarrow interDependencies + input \rightarrow ID_{task}$
            end
        end
        $task \rightarrow interDependencies \leftarrow interDependencies$
        $interDependencies \leftarrow [\,]$
    end
end
```

Algorithm 1: Internal dependencies algorithm.

target. Thus, for this dependency to propagate from one target to another, it has to pass through the communication network. For instance, in Figure 3.4 *getPwmFreq* and in_{int} have a network dependency that has to propagate between *Target 1* and *Target 2*.

There are two types of network dependencies, *consumer* and *producer*. The main difference is that, in the *producer*, transmission time is considered while, in the *consumer*, transmission time is not considered.

Consumer Dependencies

Consumer dependencies mean that, a task, is waiting for an argument to be produced on the network in ordered to be consumed, before being able to execute. Similarly to the way of finding *interDependencies*, it is relied only on the *inputs* of a task. For every task a *consumerDependencies* list is defined. While checking for *interDependencies*, an *else* clause, is added within Algorithm 1, and thus, obtaining the *consumerDependencies*. In other words if the ID_{task} (origin source of the input), does not belong to the same tasks' group, thus, it is a *consumer* dependency. And therefore, a task has to wait, for the ID_{task}'s listed by *consumerDependencies*, before being able to start execution. The modified Algorithm 2 expresses what is formerly described.

```
Data: $Dictionary$ : targets' groups
Result: $interDependencies \leftarrow [\,]$, $consumerDependencies \leftarrow [\,]$

foreach $(ID_{target},\ taskSet) \in Dictionary$ do
    foreach $task \in taskSet$ do
        foreach $input \in task \rightarrow inputs$ do
            if $input \rightarrow ID_{task} \in taskSet$ then
                $interDependencies \leftarrow interDependencies + input \rightarrow ID_{task}$
            end
            else
                $consumerDependencies \leftarrow consumerDependencies + input \rightarrow ID_{task}$
            end
        end
        $task \rightarrow interDependencies \leftarrow interDependencies$
        $task \rightarrow consumerDependencies \leftarrow consumerDependencies$
        $interDependencies \leftarrow [\,]$
        $consumerDependencies \leftarrow [\,]$
    end
end
```

Algorithm 2: Internal & consumer dependencies algorithm.

Producer Dependencies

Producer dependencies arise from data dependencies between two tasks coming from distinct targets as mentioned before. This dependency has two concerns, the network physical dependency which includes transmission time, and the data flow sequence. The data flow dependency is intrinsically included within *consumerDependencies*. While the network physical dependency is considered here and is called *producer* dependency. In other words, when a task has an output argument to be written over the network, then, the task should wait to get exclusive access over the communication network.

In this context, it is relied on the *outputs* of a task, specially because an output can be connected to several inputs. But from a network point of view, it is still only one message while being read from different targets. This is taken into account while computing transmission payload, which is described forthcoming.

For every task a *producerDependencies* list is defined. *producerDependencies* is built going through the *outputs*, of a task, and check the ID_{task}'s for each output. Those

ID_{task}'s represent the tasks, which this output is connected to. If an ID_{task} does not belong to the same tasks' group, thus, it is a *producer* dependency. Algorithm 3 expresses what is formerly described. In other words, *producerDependencies* states the ID_{task}'s which have to wait for this task to finish (execution + transmission time), before start executing.

As a consequence, the data flow sequence is considered twice, in both *consumerDependencies* & *producerDependencies*. It is why only one of them is enough for formulating the constraints of the optimization problem. The use of the first, *consumerDependencies*, is solely for the purpose of recognizing the *consumer task*. More about this issue is described in Section 3.7.

```
Data: $Dictionary$ : targets' groups
Result: $producerDependencies \leftarrow [\,]$

foreach $(ID_{target},\ taskSet) \in Dictionary$ do
    foreach $task \in taskSet$ do
        foreach $output \in task \rightarrow outputs$ do
            foreach $ID_{task} \in output \rightarrow IDs_{task}$ do
                if $ID_{task} \notin taskSet$ then
                    $producerDependencies \leftarrow producerDependencies + ID_{task}$
                end
            end
        end
        $task \rightarrow producerDependencies \leftarrow producerDependencies$
        $producerDependencies \leftarrow [\,]$
    end
end
```

Algorithm 3: Producer dependencies algorithm.

Transmission Time Computation

As previously mentioned, the *producer tasks* have to produce output arguments over the communication network. These arguments, referred to as *payload*, need certain time to be transmitted from one target to another through the communication network. This time is called *transmission time* or *Tx time*. *Tx time* is completely dependent on the type of the used communication network. As a consequence, the computation approach is split in two. First, recognizing the type of *payload*, which is a requisite for the model. Second, running the time computation function, which is network specific. In this way, the model's tolerance is enhanced for such design changes later on if desired.

Recognizing the type of payload is done while looking for the *producerDependencies*. The type of the *output*, which is considered to be invoking a producer dependency, is appended to the *payload*. However, the payload is not duplicated, in case the *output* is connected to several input arguments. Because, from the network point of view the message is still one, even if several targets are reading it. The modified Algorithm 4 expresses what is formerly described.

```
Data: $Dictionary$ : targets' groups
Result: $producerDependencies \leftarrow [\,],\ payload \leftarrow [\,]$

foreach $(ID_{target},\ taskSet) \in Dictionary$ do
    foreach $task \in taskSet$ do
        $tempIsPayload \leftarrow False$
        foreach $output \in task \rightarrow outputs$ do
            foreach $ID_{task} \in output \rightarrow IDs_{task}$ do
                if $ID_{task} \notin taskSet$ then
                    $tempIsPayload \leftarrow True$
                    $producerDependencies \leftarrow producerDependencies + ID_{task}$
                end
            end
            if $tempIsPayload$ then
                $tempIsPayload \leftarrow False$
                $payload \leftarrow payload + output \rightarrow type$
            end
        end
        $task \rightarrow producerDependencies \leftarrow producerDependencies$
        $task \rightarrow payload \leftarrow payload$
        $producerDependencies \leftarrow [\,]$
        $payload \leftarrow [\,]$
    end
end
```

Algorithm 4: Producer dependencies & payload recognizing algorithm.

The MoPS system uses high-speed CAN bus communication network, which is described in Section 2.2. Therefore, the proposed time computation function is specific for CAN

Type	**Representation**
int	4 *bytes*
float	4 *bytes*
bool	1 *byte*

Table 3.2: Variables' representation in *bytes*.

protocol. Recall that the data frame of a CAN transmission carries a significant amount of overhead and bit stuffing. Considering the last-mentioned, and for being in the safe side it is assumed the worst case of bit stuffing, which is all bits being *1's*. The below formula helps in computing the size of a complete frame in *bits*:

$$\begin{gathered} 8n + 44 + \left\lceil \frac{34 + 8n - 1}{4} \right\rceil \\ 1 \leq n \leq 8 \end{gathered} \tag{3.5}$$

Where n is the number of bytes in the data field of the frame. One data frame can carry up to *8 bytes* of data maximum. Thus $8n + 44$ represents the size of the transmission frame before stuffing, and the remaining of the formula, estimates the maximum bit stuffing.

The type of arguments (or variables), of a task, can be either *int, float* or *bool*. Table 3.2 shows the *variables* representation size in bytes. The number of data frames needed, to fit a required transmission is computed, by simple calculations using the *payload*. Thus, the total number of bits that has to be transmitted per *producer task* is obtained. Finally, to get the estimated transmission time (*Tx time*), the obtained number of bits is divided by the speed (or *baud rate*) of the high-speed CAN bus, which is *1 Mbps*.

3.5 Time-Line Model

One of the objectives, of this thesis, is to find a time schedule where all the dependencies, previously described, are respected. In such a way, that the maximum CPU utility is exploited, for obtaining the minimum latency for each and every task, and consequently the minimum period. In other words, it is desired to find the optimal *time-line*. Before understanding how this is included in the model, some concepts are first explained: *time slots* and its corresponding *buffer*.

3.5.1 Time Slots

In systems with real-time behavior, determinism always comes first. In order to insure the latter, the concept of *time slots* is introduced. *Time slots* are discrete time windows, where writing over the communication network is granted for one and only one target. As described in Section 3.4.2, the transmission frame size is generic, and the task WCET is generic as well. Therefore, *time slots* cannot be predefined, specially that the tasks, used within the MoPS system, are non-preemptive tasks. As a consequence, *producer task* are responsible for determining the *time slots*. This is mainly the reason why the *producer tasks* are crucial in the proposed model.

Regarding the two other type of tasks, *free* and *consumer tasks*, eventually fall in one of the time slots, respecting their precedence constraints. However, tasks have to obey the bounds of the *time slots*, this issue is discussed in more details in Section 3.7.

Slot Buffer

The parameters of a task which describe time are: *WCET* and *Tx time*. Both of them are estimated values, WCET is estimated outside the scope of this thesis. While *Tx time*, is estimated within the model, as shown in Section 3.4.2. Because of this estimations and other uncertainties, it is desired to have some marginal time within the *time slots* for staying in the safe side, and for respecting the desired fault tolerance of the design. This concept is introduced into the proposed model by the name of *slot buffer*. *Slot buffer* is a scheduling parameter defined uniformly to all slots by the test engineer. It is assigned to the *producer task*, since they are the ones defining *time slots*.

3.5.2 Task Delays

For every task τ_i, two time-line instants are defined: ϕ_i & D_i. ϕ_i being the offset of the task in the time-line, i.e. where the task is supposed to start execution at earliest. D_i being the deadline of the task in the time-line, i.e. where the task is supposed to end execution by latest. These two instants are used as the decision variables of the *optimization problem*. Thus the final objective, is to find the tuple $(\phi_i,\ D_i)$ for each task τ_i, respecting the precedence constraints and the scheduled communication. In order to do so, the concept of TUF, described in Section 2.6, is employed. For each of ϕ_i & D_i, a distinct TUF is defined as follows:

$$TUF_{\phi_i} \ : \ [\,0,\ T - WCET_i\,] \rightarrow [\,0,\ 1\,] \tag{3.6}$$

$$TUF_{D_i} \ : \ [\,WCET_i,\ T\,] \rightarrow [\,0,\ 1\,] \tag{3.7}$$

Where T is the estimated upper bound for the schedule period. The estimation of T is explained in Section 3.7. The input domain intervals are defined for the ideal case (i.e. no precedence constraints) Ideally, any task should be able to start between the beginning of the time-line (zero instant) and the end of the time-line subtracting the task's execution time (period T - WCET). Also, any task, ideally, should be able to end between the WCET instant, of the task, and the end of the time-line (period T).

Both TUF_{ϕ_i} & TUF_{D_i} are chosen to be decreasing straight lines going from a maximum of 1 into a limit of 0. If the latter is attained for any of ϕ_i or D_i, it denotes an invalid value for the respective instant parameter. Therefore, an invalid schedule. Whereas, if the maximum is attained, it means minimum latency obtained for the respective parameter. Minimum latency is the main goal, so in other words, a value of 1 is sought for all the defined TUFs.

3.6 Task Set Transformation

Before drawing up the optimization problem's constraints, the raw task set Γ has to be transformed into a formulated task set Γ_{form}. That is to say, all the algorithms, explained in Section 3.4, have to be run over Γ. Moreover, the *slot buffer* attribute is appended in correspondence with what is stated in Section 3.5.1. Hence, the output, of such data transformation, is a dictionary of task sets grouped with respect to their parent target:

$$\begin{gathered}\{\ ID_{target_i}\ :\ [\ \tau_{form_j}(\dots),\dots\ ],\dots\ \}\\ 1\ \leq\ i\ \leq\ n\\ 1\ \leq\ j\ \leq\ m_i\end{gathered}\tag{3.8}$$

Where n is the number of used targets, and m_i is the number of children tasks of the i_{th} target. Every task τ_j of Γ is transformed into τ_{form_j}, as follows:

$$\begin{multlined}\tau_{form_j}(\ ID_{task},\ WCET,\ Tx,\ slotBuff,\ interDependencies,\\ consumerDependencies,\ producerDependencies\ )\end{multlined}\tag{3.9}$$

3.7 Optimization Problem

The objective is to transform Γ_{form} into a complete structured ILP problem. For easing the understanding, the ILP problem is uncovered step by step, explaining each by its own.

Any optimization problem starts by recognizing the decision variables. For the proposed model, the *decision variables* are the mentioned task delay in Section 3.5.2. Therefore, for every task τ_{form_j}, two decision variables are defined, ϕ_j & D_j. Ideally, the relation between the decision variables is as follows:

$$D_j \geq \phi_j + WCET_j \tag{3.10}$$

But since it is desired to find the optimal time-line (i.e. minimum latency), as described in Section 3.5, the following equality is forced for every task τ_{form_j}:

$$D_j = \phi_j + WCET_j \tag{3.11}$$

In this context, the idea of having two decision variables can be dropped. But it is decided to keep them both, for easing the formulation of the constraints, and for keeping the option of returning the inequality of (3.10) at any time.

3.7.1 Period Bound

The *decision variables* chosen represent time instants, thus, they can take any value between $[\ 0,\ +\infty\ ]$. But for giving an upper limit to the optimizer, an estimation, of the upper bound for the schedule period T, is computed. Specifically for the case of an unfeasible schedule. The upper bound of T is computed by accumulating the $WCET$ plus the Tx (if any), plus the $slotBuff$ (if any) of all the tasks. The latter three parameters are attributes of the formulated task τ_{form_j} described in (3.9). Algorithm 5 expresses what is formerly described. In other words, the upper bound T supposes as if all the tasks are happening consecutively over one target only.

```
Data: $\Gamma_{form}$ : formulated task set dictionary
Result: $periodBound \leftarrow 0$

foreach $(ID_{target},\ taskSet) \in \Gamma_{form}$ do
    foreach $task \in taskSet$ do
        $periodBound \leftarrow periodBound + task \rightarrow WCET + task \rightarrow Tx + task \rightarrow$
          $slotBuff$
    end
end
```

Algorithm 5: Period T upper bound algorithm.

3.7.2 Objective Function

As described in Section 3.5.2, the concept of TUF is employed. The *objective function* is defined as the accrual of all the TUFs designated for both ϕ_j & D_j. Recall that each TUF can attain values between [0, 1]. 0 representing an invalid value for the respective instant parameter. Whereas 1 denotes that the minimum latency is obtained for the respective instants parameter. Since it is desired to find the optimal time-line (i.e. minimum latency). Therefore, the goal is to *maximize* the objective function:

$$max \sum_j (TUF_{\phi_j} + TUF_{D_j}) \tag{3.12}$$

3.7.3 Inequality Constraints

The first constraints introduced are for ensuring exclusivity of CPU on each target. Every target can execute one task a time at most, no matter the type of the task. Hence two tasks cannot be scheduled at the same time on the same target's time-line. Figure 3.6 helps in understanding this concept. Suppose having two task, A & B, which have to be scheduled on the same target. Supposing *Task A* is happening where it is shown on the time-line. Then *Task B* can either end before *Task A* starts, OR *Task B* can start after *Task A* has already ended. The same stands for scheduling *Task A* with

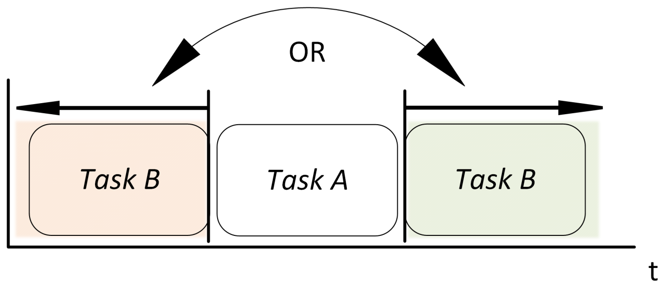


Figure 3.6: CPU exclusivity.

respect to *Task B*. The complication in this case, is the presence of the OR logic. This is problematical because it cannot be translated into inequality constraints, unless auxiliary binary variables are defined. Therefore, the constraints are constructed as follows:

$$D_i \leq \phi_j + T * \delta(i,j) \tag{3.13}$$
$$D_j \leq \phi_i + T * (1 - \delta(i,j)) \tag{3.14}$$

Where δ is the auxiliary binary variable, defined for every (i, j) combination of tasks, coming from the same parent target. Such that $i \neq j$. Notice the presence of the *factor* added on the right side of the inequalities. Depending on the value of the binary variable, this *factor* is present or not, to ensure that the two inequalities do not contradict each others. Thus, the *OR* logic is modeled. In principle, this *factor* has to be a big number, as a consequence, the upper bound of the time-line period T is chosen to be used.

The exclusivity of the communication network is formulated as inequality constraints as well. For defining such constraints, the *producer tasks* are the intended parameters, thus, the *slots* shaping. Following the same logic for constructing (3.13) & (3.14). The constraints are defined as follows:

$$D_i + Tx_i + slotBuff_i \leq \phi_j + T * \alpha(i, j) \tag{3.15}$$

$$D_j + Tx_j + slotBuff_j \leq \phi_i + T * (1 - \alpha(i, j)) \tag{3.16}$$

Where α is another auxiliary binary variable, defined for every (i, j) combination of *producer tasks*. Such that $i \neq j$. Notice the presence of Tx time and the $slotBuff$, which have values solely for *producer tasks*.

It is necessary to make sure that both *consumer* & *free tasks*, fall completely under the margins of a certain *slot*. In other words, any task cannot be surpassing a *slot* boundary. Therefore, following the same above logic once more, the constraints are constructed as follows:

$$D_i \leq D_j + Tx_j + slotBuff_j + T * \beta(i, j) \tag{3.17}$$

$$D_j + Tx_j + slotBuff_j \leq \phi_i + T * (1 - \beta(i, j)) \tag{3.18}$$

Where β is the last auxiliary binary variable used. Defined for every (i, j) product between a *producer task* j and a *non-producer task* i (*consumer* or *free task*). Notice that, only the end boundary of the *slot* is considered, because intrinsically the start boundary of the consecutive one will be considered and so on.

Finally, constructing the data dependencies constraints is simpler due to their suitable modeling described in Section 3.4. The *interDependencies* attribute, of every task τ_{form_j}, aids in defining such constraints. As previously stated, *interDependencies* lists all the ID_{task}'s for which τ_{form_j} has to wait before starting execution. Hence the constraints are formulated as follows:

$$\begin{gathered} D_k \leq \phi_j \\ \forall\ k \in interDependencies_j \end{gathered} \tag{3.19}$$

For the other data dependencies between *producers* & *consumers*, the constraints are formulated by using the corresponding defined attributes. The *producerDependencies*, of every task τ_{form_j}, is only used instead of using both defined attributes, since in this context, the *consumer* dependencies are inherently considered. Recall that *producerDependencies* lists the ID_{task}'s which have to wait for τ_{form_j} to finish execution plus its transmission time, before being able to start. The constraints are thus as follows:

$$\begin{gathered} D_j + Tx_j + slotBuff_j \leq \phi_k \\ \forall\ k \in producerDependencies_j \end{gathered} \tag{3.20}$$

CHAPTER 4

Implementation

This chapter exhibits a full explanation of the complete SW implementation of the TDCS-Planner proposed by this thesis. The *system model* integration, discussed in Chapter 3, is included as well. Also, the *builder* and all its complementarities are presented. Moreover, all the *data base* interfaces, of the TDCS-Planner, are described within this chapter. The whole process of generating a schedule plan is error-prone, starting from the data fetching going to the targets' instantiation plus defining their tasks' interconnections. Specifically, since at the last-mentioned steps, human interaction is required to build the test plan. Hence, this chapter shows how several approaches are taken, from within the design phase, to avoid human mistakes as much as possible. Several *sanity checks* are implemented as well, to make sure the output of the TDCS-Planner is completely infallible.

The flow of this chapter goes by describing roughly the *SW architecture* from a higher abstract level point of view. Then, the details of all the modules are exhibited going from a high to low level of abstraction. Thus, beginning from the *user interfaces*, which are listed in a hierarchical order. Moving on to a deeper abstract level, where the *core modules* are presented. The *core modules* emphasizes the approaches which this thesis brings up to research. Finally the complete *data base* of the TDCS-Planner is demonstrated, from every aspect of data interface that the tool brings in need.

4.1 Software Architecture

The SW architecture describes the paradigm followed in order to achieve the required functionality. Thus, the architecture states what modules are created, how are they connected together, what hierarchies are they following. These questions are answered within this section. Recall that the programming language used, for developing the TDCS-Planner, is *Python 3* language. The choice of using the latter language is argued in Section 2.7. For developing the graphical user interface aspect of the TDCS-Planner, *PyQt5* libraries are used, this is also argued in Section 2.7.1.

Figure 4.1 shows a hierarchical chart of the modules composing the SW architecture. The *Main Window* module represents the highest hierarchical level of the TDCS-Planner. It is the *parent* class for all the other classes defining the rest of the modules. The *Menu Bar, Tool Bar* & *Status Bar* represent the highest level graphical user interface, by which the user can directly interact with the *Main Window* module. Either in an input direction, by triggering desired actions through both the *Menu Bar* & *Tool Bar*, or in an output direction, by getting the messages prompted over the *Status Bar*. The *Main Actions* portrays, solely, a core aspect, where all the main actions, of the TDCS-Planner, are defined. Except the specific actions, of the children modules, which are defined within the respective module itself. Having the *Main Actions* module defined over the highest hierarchical level, gives a flexibility advantage. For instance, the latter module can be accessed from any other child module, thus, dealing with *dependency injections* is not necessary.

The *Targets* module describes the targets instantiations of the desired targets, used while building the test plan. This module is, also, responsible for fetching the *configuration* files of the predefined targets. Hence, both of the core and user interface aspects are necessary for the *Targets* module. The *Builder* module is where the test plan is designed, with the desired requirements. The *Builder* provides a very interactive and friendly user interface, where all the precedence relations are drawn up by means of connections between the tasks. Simultaneously, this module stores the test plan data in the backend, and is able to run the *sanity checks* over the data, before passing it over, for the *Scheduler* when needed. This last-mentioned actions stand as the core aspect of the *Builder* module. The *Scheduler* module runs the scheduling algorithm based on what is described in Chapter 3. In addition, it allows the illustration of the resulted schedule time-line, in a very comprehensive way, emerging a smooth user interface experience. Finally, the *Prompt* module shows one of the logging streams. The main logging stream, where the user can get all sorts of feedback depending on the interactions performed.

In the next sections, each module is explained comprehensively. Talking first about the user interface aspect, of the modules, and what features are chosen to be included for better enhancing the user experience. Then, the core aspect is described, stating what methods are followed in order to prevent errors as much as possible. Also, all the sanity checks developments are elaborated, where data cross-checks are issued. The explanation is thus concluded by exhibiting all the data interfaces of the TDCS-Planner with the memory *data base*.

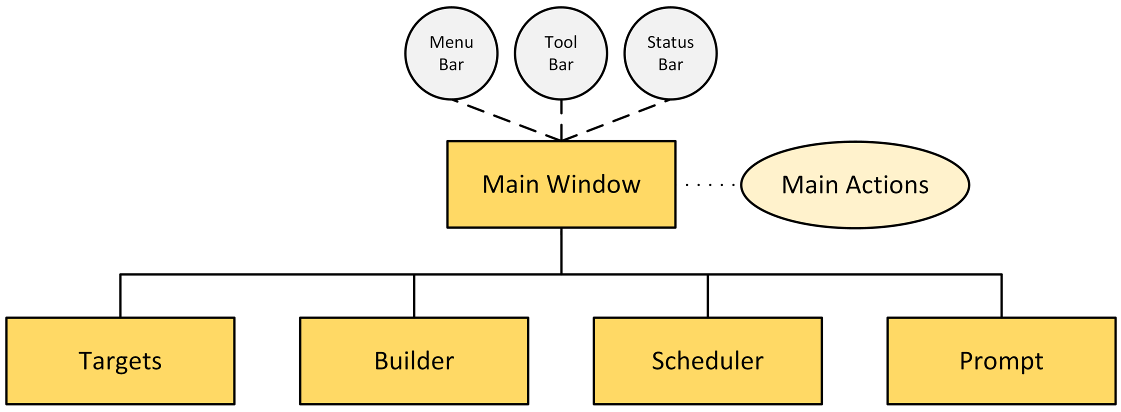


Figure 4.1: SW architecture modules.

4.2 User Interface

When developing graphical user interfaces, the main aim comes in achieving the best user experience possible. Before describing how the last-mentioned aim is attained, the requirements from the TDCS-Planner are recalled. From a solely interface point of view, the TDCS-Planner should allow the user to do the following main tasks:

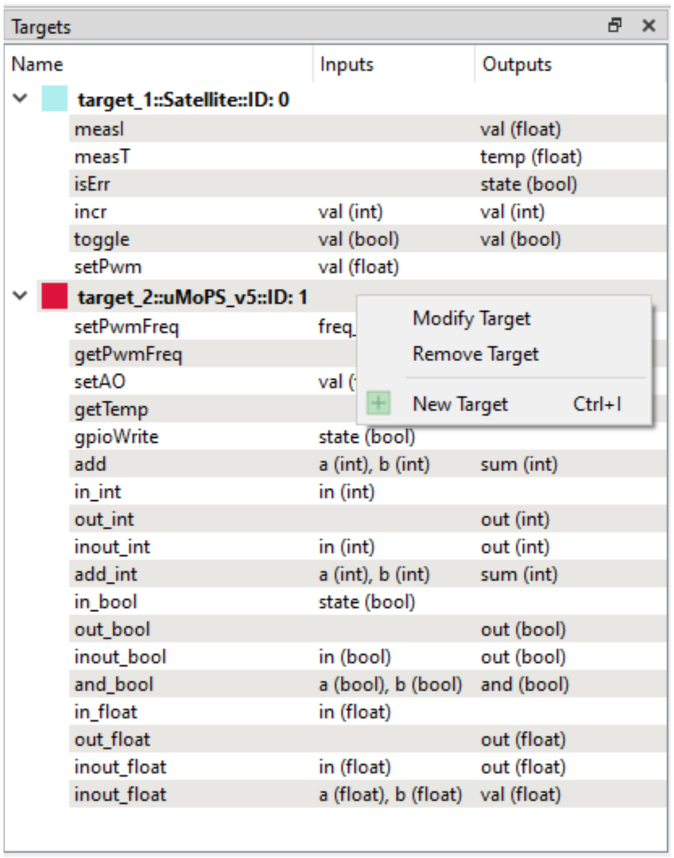
- Create target instances based on the available configuration files, assigning for each their specific parameters ($Label_{target}$, *CAN ID,...*).
- Create task instances with reference to the parent target, assigning as well their corresponding parameters (*Default Inputs*, $Label_{task}$,...).
- Create the desired precedence relations between the instantiated tasks.
- Generate the optimal schedule time-line for the designed test plan.

Having these points clear, the user experience can be further argued. Forthcoming, the user interface, of every module, is discussed by its own.

4.2.1 Targets Tree

This part of the TDCS-Planner is where the user is able to create target instances. Recall that targets refer to specific µC. There are certain parameters that configure a target, forming a *targets configuration*, as described in Section 2.3.1. In this section, the focus is only on the parameters of interest for the user interface. The rest of the configuration parameters are dealt with in Section 4.4.2. From the *targets configuration*, for instance the one shown in Listing 2.1, the only two entries which are relevant for the user are the *name* and the *tasks*. This is why it is chosen to represent the information, given by the last-mentioned entries, in the form of a hierarchical tree. Specially that it is convenient to emphasize the parent-child relation between every used task with its corresponding target. These children tasks are then used as dragging items for visualizing tasks over the Builder View.

Figure 4.2 illustrates the *targets tree* window of the TDCS-Planner. As depicted, three columns are used for visualizing the data of every target instance. The first column lists the *Name* of each target and task. The other two columns list the *Inputs* and *Outputs* for each task. Moreover, for every entry in either of the last-mentioned columns, the *name* and the *type* of the argument (or variable) are both stated. Notice as well, that beside the name of the instantiated target, there are two other parameters. The $Label_{target}$ and



Name	Inputs	Outputs
🟦 **target_1::Satellite::ID: 0**		
measI		val (float)
measT		temp (float)
isErr		state (bool)
incr	val (int)	val (int)
toggle	val (bool)	val (bool)
setPwm	val (float)	
🟥 **target_2::uMoPS_v5::ID: 1**		
setPwmFreq	freq_	
getPwmFreq		
setAO	val (	
getTemp		
gpioWrite	state (bool)	
add	a (int), b (int)	sum (int)
in_int	in (int)	
out_int		out (int)
inout_int	in (int)	out (int)
add_int	a (int), b (int)	sum (int)
in_bool	state (bool)	
out_bool		out (bool)
inout_bool	in (bool)	out (bool)
and_bool	a (bool), b (bool)	and (bool)
in_float	in (float)	
out_float		out (float)
inout_float	in (float)	out (float)
inout_float	a (float), b (float)	val (float)

Figure 4.2: Targets tree window.

the assigned *CAN ID*. Both of the last-mentioned parameters are determined by the user. One last thing, is the *color* box shown beside each target. This is automatically assigned by the *targets core*, as described in Section 4.3.1. The *color* helps the user distinguish between the parent target of each used task, while building the test plan. Thus, enhancing the parent-child relation in a user friendly manner. This amount of information about every target instance is enough for the user to be able to proceed building the desired test plan. If a target is already instantiated, the user has the option to remove it at any time. To perform that, the user has to hoover over the desired target instance, and trigger the *Remove Target* action from its corresponding *context menu*. Hereafter, the parameters reserved by the removed target instance are freed by the *targets core*, and can be used again, as described in Section 4.3.1. Once a target instance is removed, all its linked tasks, used in the test plan, are automatically removed as well.

New Target Dialog

This dialog is prompted whenever the *New Target* action is triggered. The latter action is defined within the *Main Actions* module, and thus, can be triggered in a global scope, as described in Section 4.3.5. The dialog is chosen to be of *modal* type, considering the importance of the action. Also, since there is no logic operation that can happen simultaneously with creating a new target. Therefore, the dialog blocks the user interface, until the dialog is either accepted or rejected.

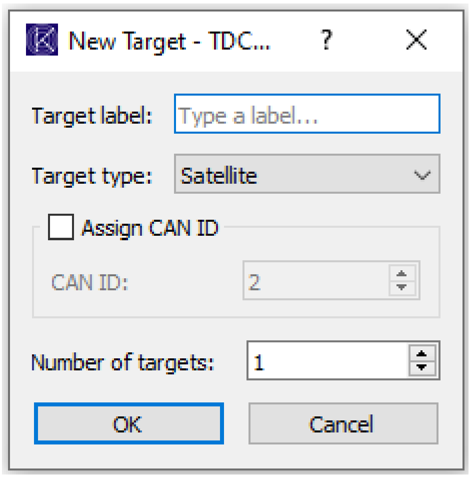
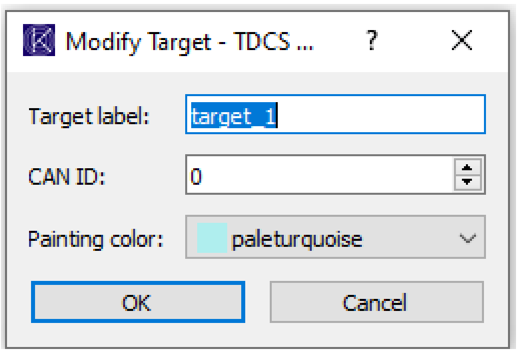


Figure 4.3: New target dialog.

Figure 4.3 demonstrates the dialog. The entries of the dialog determine all the necessary parameters which are assigned by the user for creating a target instance. *Target label* should be a non-empty unique string. *Target type* gives the possibility to choose a target type from the combo box, which are dependent on the fetched *targets configuration* files. *CAN ID* is either automatically assigned by the TDCS-Planner, or by the user, it should be an unique positive integer. The last entry, is the *Number of targets* which determines how many instances are desired. All of the entries' data undergo certain logic for either checking the correctness of the inputs, or for prompting the correct options, in the case of the *Target Type* for instance. These last-mentioned data checks are performed by the *targets core*, which is addressed in Section 4.3.1.

Modify Target Dialog

This dialog is directly linked with a certain target instance, already created by the user in the targets tree. In this context, the action that triggers the dialog is not of global scope, thus it is not part of the *Main Actions* module. In order to trigger the action, the user has to hoover over an existing target instance, and trigger the *Modify Target* action from its corresponding *context menu*. Hereafter, the dialog prompts as demonstrated by Figure 4.4. The dialog is chosen to be of *modal* type. Hence, it blocks the user interface until the dialog is either accepted or rejected.



Target label: target_1
CAN ID: 0
Painting color: paleturquoise

Figure 4.4: Modify target dialog.

The aim of this dialog is to give the user the possibility to modify the basic target parameters, after the instance has already been created. *Target label, CAN ID* and *Painting color* are the modifiable parameters. The *Painting color* is just in case the user desires to visualize the tasks of this specific target in a different color. The user has to choose between a list of available predefined colors. First, for ensuring the uniqueness of colors used and second, since some colors are saved for specific use by the TDCS-Planner. For instance, the colors of the connections, described in Section 4.2.2. Moreover, data entries are re-checked to be consistent with the logic of TDCS-Planner, and ensure correctness, as exhibited in Section 4.3.1.

4.2.2 Builder View

This is the most sophisticated part of the the graphical interface of the TDCS-Planner, since it is where the whole test plan gets designed. Therefore, the user spends most of the time, while using the TDCS-Planner, within the *Builder View*. Thus, the aim is to make the *Builder View* as user friendly as possible to enhance the user experience. A graphical scene is used, as the main component of the builder, where the user is able to graphically visualize each task with all the precedence relations. Figure 4.5 illustrates the *Builder View*. Notice the *task items* along with their *connection items* determining the data dependencies (precedence relations). Forthcoming, each of the last-mentioned items is discussed. Moreover, for better enhancing the user experience certain features are offered by the *Builder View*. For instance, *pan* & *select* tools are available for panning and selecting through the *tasks items* and *connection items*. The *arrow keys* can also be used for navigating, or moving items through the *Builder View*. *Zooming* feature is also available by either using the *mouse wheel*, or triggering the actions defined in the *Main Actions* module, as described in Section 4.3.5.

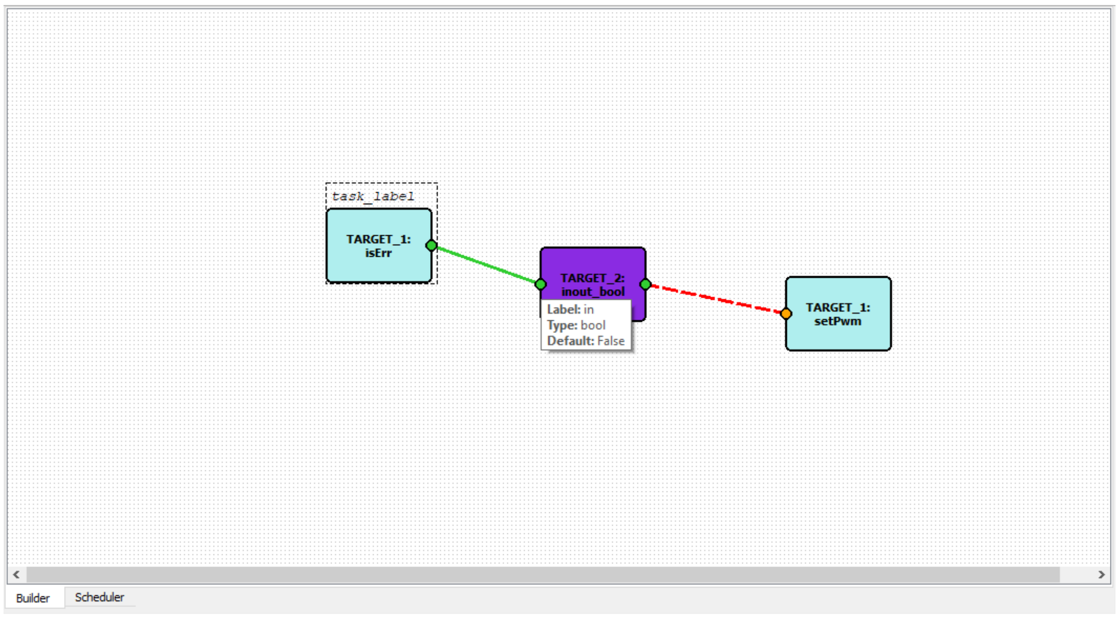


Figure 4.5: Builder view window.

Task Item

First thing that has to be considered, is the direction of the data flow. It is chosen to take the general convention of data flow, which is from left to right. Thus, any *task* must take its inputs at the left side and provide its outputs at the right side. The inputs and outputs are represented in terms of *ports*. The number of *ports*, of every *task item*, reflects the number of inputs/outputs defined in the *Targets Configuration* file, as described in Section 2.3.1. The *Builder* module, automatically, computes the best size of representation of the *task item*, for giving a symmetric aligned shape containing all the necessary predefined *ports*. Each *port* has a predefined $label_{port}$ and *type* as it is seen in the inputs/outputs configuration. The input *port* configuration contains the *default value* as an entry, and in some cases *MIN* and *MAX* are included entries as well. *MIN* and *MAX* are shown for the user while accessing the *modify parameters dialog*, described in Section 4.2.2. On the other hand, $label_{port}$, *type*, and *default value* are visualized, for the user, as a tool tip whenever a *port* gets hoovered over, as it can be seen in Figure 4.5.

The referencing of a *task* with its parent target is depicted by two things. First, the assigned *color* of the target, which is used for painting the body of the *task item*. Second, the $Label_{target}$ of the target is written above the *name* of the task. The second link helps making the TDCS-Planner user friendly for color-blindness cases as well. Finally, the user can also insert a $Label_{task}$ for a specific *task*. The $Label_{task}$, by default, is none but it can be modified by the user, as it is shown in Section 4.2.2. As a consequence, it is visualized above the *task item*, as also can be seen in Figure 4.5.

Modify Parameters Dialog

Similarly to the *modify target dialog*, described in Section 4.2.1, this dialog is also linked to a specific *task item* already instantiated in the *Builder View*. Thus, the action to prompt the dialog is not part of the *Main Actions* module. To prompt the dialog, the user has to either *double click* the desired *task item*, or use its respective *context menu* to trigger the corresponding action. The dialog is chosen to be of *modal* type. Hence, it blocks the user interface until the dialog is either accepted or rejected. Figure 4.6 illustrates the parameters dialog corresponding to the *task item* defined by add_{int} (from the task set of $\mu MoPS\_v5$). *Task label* is an absolute entry of this dialog, i.e., it is shown for all parameters dialogs of any *task item*. This latter entry has no restrictions, it can be also left empty. When a non-empty string is entered, it is visualized above the *task item*. The group box *Inputs Default Values* is relative to every *task item*. It shows the available input arguments of the corresponding task, allowing the user to choose the desired default values for each, depending on their types. If the *port* type is either *int* or *float*, then, a tool tip is shown when hoovering over the argument's entry, demonstrating the *MIN* and *MAX* limits, as it can be seen in Figure 4.6. The *MIN* and *MAX* attributes are not always predefined for each task in the *Targets Configuration*. If they are not predefined, then, the TDCS-Planner forces the default *MIN* and *MAX* limits for the respective types.

Attaining these limits is ensured by the entry spin boxes, thus, this whole procedure is completely error-proof.

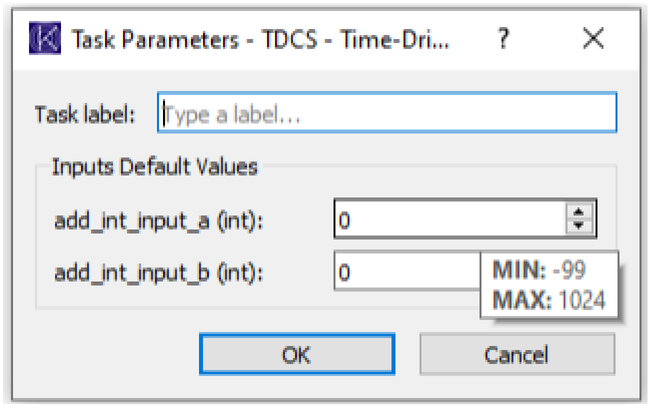


Figure 4.6: Modify parameters dialog.

Connection Item

This item allows the user to establish the desired *data dependencies* (precedence relations), by simply drawing it between two *ports*. To draw a *Connection*, the user has to *click* the desired port and then latch it, by clicking again the second port. For respecting the logic of variables, the *input port* is allowed to have only one connection, while the *output* port can have many. Moreover, the color of the connection reflects the type of the variables, where this connection is being established, in the case of matching types. In this context, *blue, orange*, and *green* are respectively used for *int, float*, and *bool*. These last-mentioned colors are saved for the use of the TDCS-Planner only and cannot be chosen as a target's *color*. However, if the connection is established between two distinct variable types, then it is considered to be a broken connection. This latter connection is visualized as a *red dashed* line to emphasize its invalidity.

4.2.3 Scheduler View

The aim of this view is to allow the user visualize the results of the generated schedule. The *Scheduler View* contains two main windows: *schedule table* & *schedule time-line*. The *schedule table* allows the user to read the values of all the fundamental parameters, computed by the scheduler, for each task. The *schedule time-line*, provides the user a very smooth expressive time-line carrying all the time instants of the tasks. In this way, the user can get a full picture of the generated schedule in both quantitative and graphical perspectives. Figure 4.7 demonstrates the default *Scheduler View*, i.e., before any schedule has been generated.

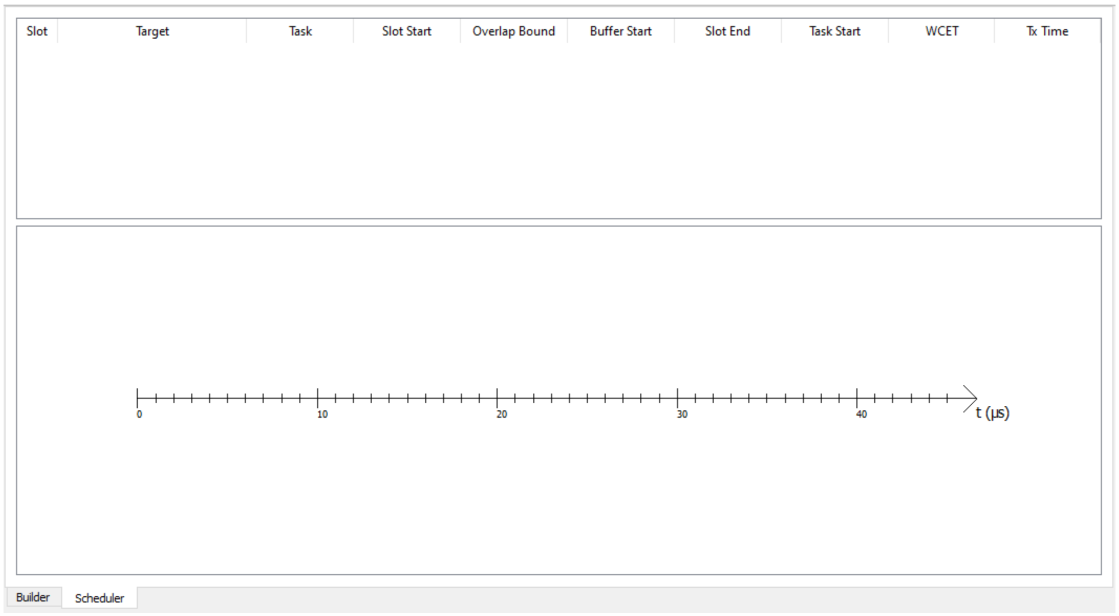


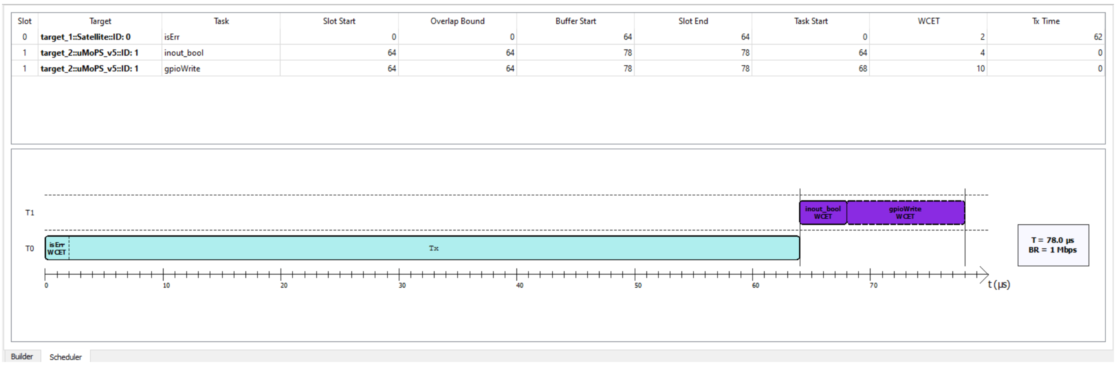
Figure 4.7: Schedule default view window.

Schedule Table

The *schedule table* represents the values of the fundamental parameters determined by the *Scheduler*. Each row of the table characterizes a *task item*, used in the designed test plan. The columns display the data entries of each item. The *Slot* column represents the slot ID, where this *task item* is scheduled, showing positive integers. The *Target* column lists the name of the parent target of the respective *task item*. The complete name is shown, i.e., $Label_{target}$, *type* and the assigned *CAN ID*. The *Task* column represents the name of the *task item*. The rest of the entries represent all the absolute time-line instants. Thus, the columns' unit follows the *time unit* determined in the *preference settings*, which is described in Section 4.2.5. The *Slot Start* represents the time instant where that task's slot is starting. The *Overlap Bound* states the instant until which the respective slot provides overlapping. The *Buffer Start* lists the starting instant of the respective slot buffer. The *Slot End* represents the ending instant of the respective task's slot. The *Task Start* portrays the instant where the task is supposed to start. The *WCET* lists the worst case execution time of each task. Finally, the *Tx Time* states the transmission time required by the task. This has non-zero values only if the task is a *producer*, as described in Section 3.2.2.

Schedule Time-line

After a schedule is generated, it is directly visualized in the graphical time-line. The time axis resolution can be chosen by the user within the *Display Settings Dialog*, described in Section 4.2.5. This view also provides the *zooming* feature which can be accessed either by the *mouse wheel* or through the respective actions defined in the *Main Actions* module. Figure 4.8 illustrates an example of the *schedule time-line* after a schedule has been generated. The y-axis, of the time-line, represents the different targets, used in the designed test plan. Each target, named $T < ID_{target} >$, can have several rows for its illustration. The first row portrays two type of tasks. The *free tasks* which are distinguished by being drawn with a dashed-line contour, and the *consumer tasks* with a solid-line contour. These task types have no transmission time, as described in Section 3.2. Thus, their representation on the time-line is solely based on their corresponding WCETs. Every *producer task* occupies a distinct unique row for itself. This allows the clean representation of the overlapped time intervals between distinct tasks. The *producer task* has two time intervals to be depicted over the time-line. The WCET and *Tx Time* are separately shown over the time-line as portrayed by Figure 4.8. Moreover, the *colors* of the targets, determined during the building phase, are preserved for emphasizing the relation between the *Builder View* and the *Scheduler View*. Finally, the *legend* of the time-line shows the resulted *period* along with the *Baud Rate (BR)*.

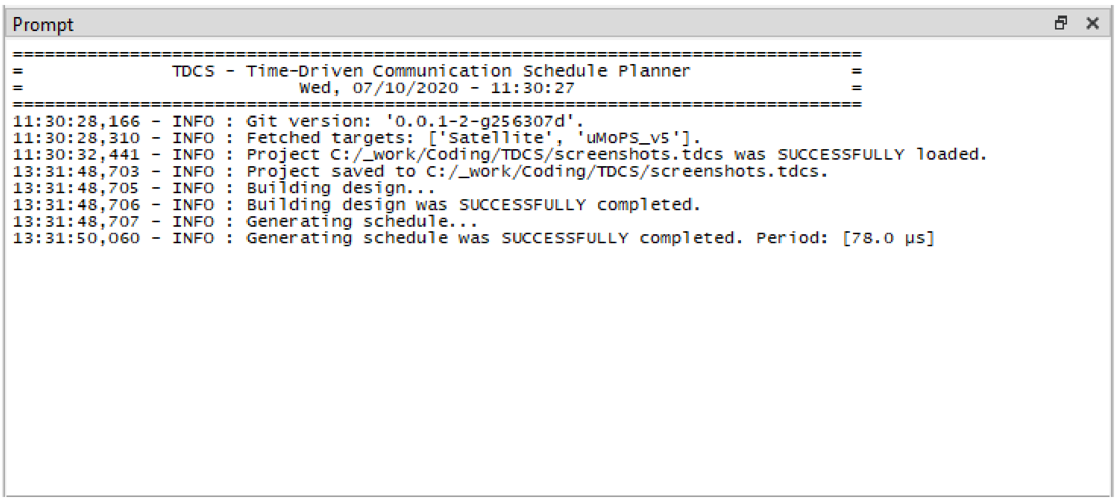


Slot	Target	Task	Slot Start	Overlap Bound	Buffer Start	Slot End	Task Start	WCET	Tx Time
0	**target_1::Satellite::ID: 0**	isErr	0	0	64	64	0	2	62
1	**target_2::uMoPS_v5::ID: 1**	inout_bool	64	64	78	78	64	4	0
1	**target_2::uMoPS_v5::ID: 1**	gpioWrite	64	64	78	78	68	10	0

Figure 4.8: Schedule view window.

4.2.4 Prompt View

The *Prompt View* is where the user is able to see the logging information of the TDCS-Planner. This view shows one of the logging streams of the TDCS-Planner, which are described in Section 4.4.4. Figure 4.9 illustrated the *Prompt View* of the tool.



```
================================================================================
=                  TDCS - Time-Driven Communication Schedule Planner              =
=                              Wed, 07/10/2020 - 11:30:27                         =
================================================================================
11:30:28,166 - INFO : Git version: '0.0.1-2-g256307d'.
11:30:28,310 - INFO : Fetched targets: ['Satellite', 'uMoPS_v5'].
11:30:32,441 - INFO : Project C:/_work/Coding/TDCS/screenshots.tdcs was SUCCESSFULLY loaded.
13:31:48,703 - INFO : Project saved to C:/_work/Coding/TDCS/screenshots.tdcs.
13:31:48,705 - INFO : Building design...
13:31:48,706 - INFO : Building design was SUCCESSFULLY completed.
13:31:48,707 - INFO : Generating schedule...
13:31:50,060 - INFO : Generating schedule was SUCCESSFULLY completed. Period: [78.0 µs]
```

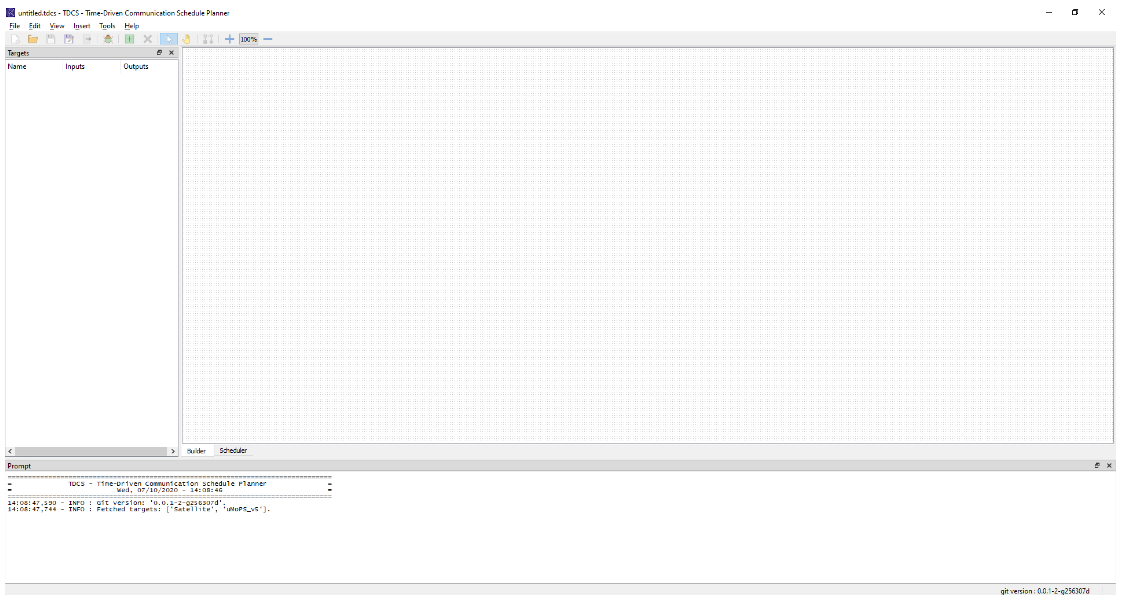
Figure 4.9: Prompt view window.

4.2.5 Main Window

The *Main Window* user interface is the collection of the above described user interfaces. In addition to them, the *Menu Bar, Tool Bar*, and *Status Bar* are, also, included. The former two mentioned is where the user can access the actions defined in the *Main Actions* module, while the latter is just an output interface for the user. Figure 4.10 illustrated the main window of the tool.

The *Targets Tree* and the *Prompt View* are implemented as docked widgets. This gives the user the option to float these windows, which enhances the user experience. The central window, of the *Main Window*, displays a tabbified widget, which contains both the *Builder View* and *Scheduler View*. Having both views in form of tabs grants better presentation of the TDCS-Planner, and endorses the logic of the applications as well. Since, the *Builder View* and the *Scheduler View* are not meant to be simultaneously used.

The *Menu Bar* represents all the actions defined by the *Main Actions* module. Alternatively, the *Tool Bar* shows a group of the *Main Actions*, which the user uses more


untitled.tdcs - TDCS - Time-Driven Communication Schedule Planner
File
Edit
View
Insert
Tools
Help
100%
Targets
Name
Inputs
Outputs
Builder
Scheduler
Prompt
git version : 0.0.1-2-g256307d

```
=================================================================================
=                TDCS - Time-Driven Communication Schedule Planner              =
=                          Wed, 07/10/2020 - 14:08:46                           =
=================================================================================
14:08:47,590 - INFO : Git version: '0.0.1-2-g256307d'.
14:08:47,744 - INFO : Fetched targets: ['Satellite', 'uMoPS_v5'].
```

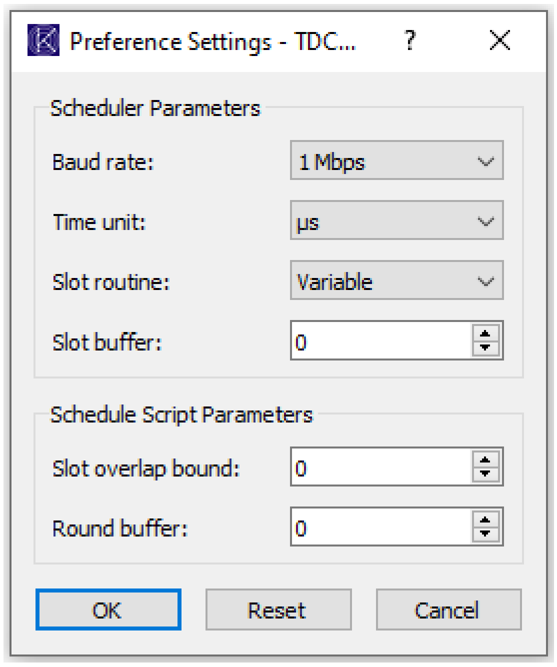
Figure 4.10: Main window.

frequently. Moreover, the user is able to manipulate the zooming feature, within the *Tool Bar*, for both the *Builder View* and *Scheduler View*, while getting a percentage representation of the respective zooming scale. Forthcoming, all the actions defined in *Main Actions* module are described in Section 4.3.5. Ultimately, the *Status Bar* shows all the temporary messages with respect to the user interactions. Also, over the right side, the latter bar displays permanently the version of the TDCS-Planner.

Preference Settings Dialog

This dialog helps the user determine the preference settings for the application instance. It is prompt whenever the *Preference Settings* action is triggered. The latter action is defined in the *Main Actions* module, and thus, can be triggered in a global scope as described in Section 4.3.5. The dialog is chosen to be of *modal* type, considering the importance of the action. Hence, it blocks the user interface until the dialog is either accepted or rejected. Moreover, this dialog offers the reset option, which permits the user to get back to the default *preference settings*, at any time during the application instance.

Figure 4.11 demonstrates the dialog. The entries of the dialog list the necessary settings that the user has to specify for the work flow of the TDCS-Planner instance. The *Scheduler Parameters* group box lists the data entries which are required by the *Scheduler*,


Preference Settings - TDC...

Scheduler Parameters

Baud rate: 1 Mbps

Time unit: µs

Slot routine: Variable

Slot buffer: 0

Schedule Script Parameters

Slot overlap bound: 0

Round buffer: 0

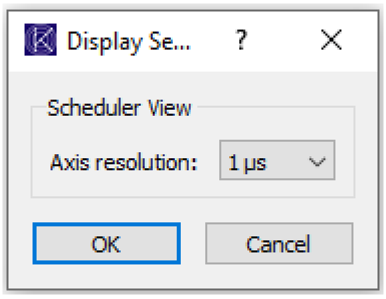
Figure 4.11: Preference settings dialog.

i.e., they take part in the schedule computation. *Baud rate* determines the BR of the used communication bus. Recall that the MoPS system uses CAN bus. Thus, the listed options for the *Baud rate* entry are the default CAN bus speeds. The *Time unit* specifies the time unit for all the scheduler computations, including the fetched time parameters, from the *Targets Configuration*. The *Slot routine* is a feature offered by the *Scheduler* which gives the user the choice between *Fixed* and *Variable* slot width. More about this scheduling feature is presented in Section 4.3.3. The *Slot buffer* determines the amount of buffer desired for the scheduling slots, which is uniquely considered for all slots.

The *Schedule Script Parameters* group box lists the data entries which are part of the *Schedule Script*, as described in Section 4.4.6. The *Slot overlap bound* determines the allowed overlap limit in each slot, which is uniquely considered for all slots. The *Round buffer* specifies the desired buffer for the complete period or round. All of the above mentioned data entries have a data consistency with the allowed data type, for each respectively. Therefore, the procedure of inputting these entries is completely error-proof.

Display Settings Dialog

This dialog helps the user determine the display settings for the application instance. It is prompt whenever the *Display Settings* action is triggered. The latter is defined in the *Main Actions* module, and thus can be triggered in a global scope as described in Section 4.3.5. The dialog is chosen to be of *modal* type. Hence, it blocks the user interface until the dialog is either accepted or rejected.



Axis resolution: 1 µs

Figure 4.12: Display settings dialog.

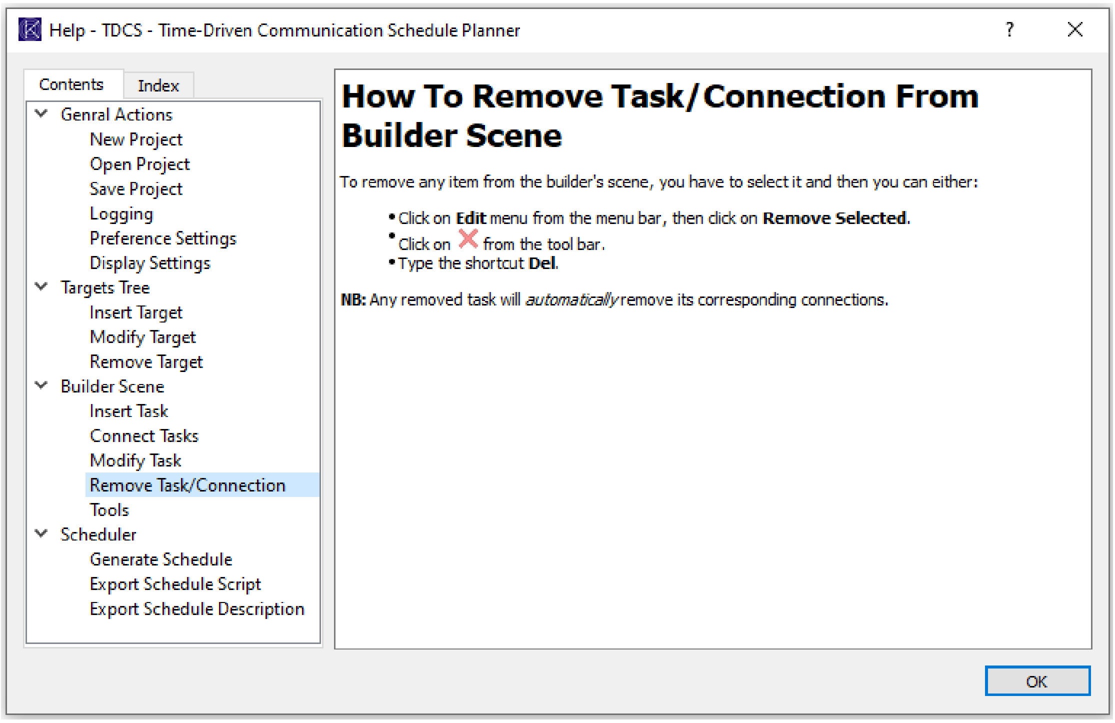
Figure 4.12 demonstrates the dialog. It contains only one entry, which is *Axis resolution*. The user can select between the list of available resolutions, which depends on the chosen *time unit*. By default, the axis resolution is a unit vector. Once a schedule time-line is generated, the user can switch between the resolutions, which redraws the time-line instantaneously.

Help Browser

Help Browser is the only non-modal dialog used in the TDCS-Planner. It allows the user to navigate through the help pages while using the tool simultaneously. Figure 4.13 demonstrates the help browser dialog. On the left side, there is a tabbified window from which the user can switch between *Contents* and *Index*. The former displays the full help tree of the content. The latter lists certain keywords from the most probable questions. Using these last-mentioned tab window, the user can choose what to display on the right window of the dialog, which is a web browser. All the entries of the tree of content have a linked Hyper Text Markup Language (HTML) file which describes verbosely what to do in that respective case.

4.3 Core Modules

The SW architecture of the TDCS-Planner, described in Section 4.1, is composed of several modules. In the previous section, the *user interface* aspect is fully presented.


Help - TDCS - Time-Driven Communication Schedule Planner

How To Remove Task/Connection From Builder Scene

To remove any item from the builder's scene, you have to select it and then you can either:

- Click on **Edit** menu from the menu bar, then click on **Remove Selected**.
- Click on ✗ from the tool bar.
- Type the shortcut **Del**.

NB: Any removed task will *automatically* remove its corresponding connections.

Figure 4.13: Help browser dialog.

In this section, the *core* developed functionalities of each module, composing the SW architecture, is exhibited. Moreover, the core functionalities which are managed by the *Main Window* module, are presented separately. The latter module stands at the highest hierarchical level of the SW architecture, and hence, its provided functionalities carry a big part of what this thesis brings to research. For instance, the *sanity checks* which are run to ensure infallibility during the scheduling process. Also, the actions offered by the *Main Actions* module, which are described within a separate sub-section.

4.3.1 Targets Core

The *Targets Core* is where the *Targets Configuration* files are fetched. The fetching process is described in Section 4.4.2. For this section it has to be noted, that the fetched data is stored as an attribute of the *Targets Core* module. Whenever the *new target dialog*, described in Section 4.2.1, is requested, the fetched data is used to display the available target types within the dialog. In this context, it is ensured that the target options displayed have already passed the fetching sanity checks, described in Section 4.4.2.

Any target instance has both representable and non-representable parameters. The latter is referred to as $Target_{data}$. The $Label_{target}$ of any target instance has to be a unique non-empty string. Therefore, before acknowledging the data input of the user, the *Targets Core* checks the correctness of the latter mentioned entry. In other words, the core first ensures that the *Target label* entry is not an empty string. In addition, it compares the user entry with the already reserved $Label_{target}$'s (labels of other created target instances). If the last-mentioned requirement is not attained, the *Targets Core* sends a customized warning message to the *Main Window* module, which is prompted over the *Status Bar*.

The *CAN ID* of any target instance must be a positive unique integer. The entry spin box of the latter already assures the correctness of the entry's type. In addition, the uniqueness of the *CAN ID* is checked by the *Targets Core*, similarly to the paradigm stated above. If the check is not successful, a customized warning message is sent to the *Main Window* as well. For this entry, the TDCS-Planner offers the option of an automatic *CAN ID* assignment. If this option is chosen by the user, the *Targets Core* assigns the smallest *CAN ID* available for the respective target instance.

The *Number of targets* entry is not checked itself, but if the user inputs it greater than one, then the $Label_{target}$ gets automatically indexed by the respective number. Also, the *CAN ID* of each target instance, gets automatically assigned the consecutive integer. Therefore, the checking procedure mentioned above are again run by the *Targets Core*. Hereafter, the data input by the user is acknowledged by *Targets Core*, which automatically assigns a *color* for the target instance. The *color* is randomly picked from a list of predefined colors, such that every pick gets exclusively reserved for the target instance. In this way, the *Targets Core* ensures that no color is used twice for distinct instances.

A target instance is characterized by its $Label_{target}$, *CAN ID, Color*, and the non-representable $Target_{data}$. Recall that a target instance is illustrated into a tree item, as described in Section 4.2.1. The children of the tree item, are the target's tasks. The *Targets Core* grants the possibility to originate a *Multipurpose Internet Mail Extensions (MIME)* data type out of the latter mentioned children items. Consequently, a child task item can be dragged and dropped, over the *Builder View*, while indexed to its respective target instance, through the originated MIME data type.

The *Targets* module provides the possibility to modify or remove a target instance at any time. Whenever a target instance gets modified, the *Targets Core* runs the same checking procedure, described above, over the data entries. Moreover, if a target instance gets removed by the user, then all its reserved parameters get freed, and can be again used by new target instances. Also, the *Targets Core* notifies the *Builder Core* to remove all the respective tasks of the removed target instance.

4.3.2 Builder Core

The *Builder Core* is responsible for structuring the data of the test plan, which is built by the user. The *Builder View* accepts drop items resembled by a MIME data type, originated from the *Targets Core*. Whenever one of the latter gets dropped over the *Builder View*, the *Builder Core* automatically unpacks the data and transforms it into a task instance. This task instance is what the *Builder View* illustrates as a graphical *task item*, as described in Section 4.2.2. The *Builder Core* keeps track of all the instantiated tasks, and stores them in the backend along with their respective links to the *Targets Tree*. Therefore, any modification performed on a target instance is directly reflected on the respective task instance.

Apart from the target's instance reference, a task instance is characterized by its *Name*, $Label_{task}$, ID_{task}, *Inputs, and Outputs*. The *Name* of the task is fetched from the task data passed through the MIME object. ID_{task} is a positive unique integer assigned automatically by the *Builder Core*. The $Label_{task}$ parameter is assigned by the user and has no restrictions, the *Builder Core* considers it none by default. The *Builder Core* is responsible of creating port instances for each *Input* and *Output* of the task instance. Hence, the task instance attributes, *Inputs* and *Outputs*, are lists of the latter port instances. Every port instance is characterized by its $Label_{port}$, *Type, Default, MIN, MAX* and *Direction*. The latter parameters are also fetched from the task data passed through the MIME object. The *Default* attribute is modifiable by the user through the *modify parameters dialog*, as described in Section 4.2.2.

Builder Core offers two modes of functioning: $Mode_{default}$ and $Mode_{connecting}$. The former is where the user creates task instances, while the latter is where the user establishes the data dependencies through connection instances. A connection instance is characterized by its *Out, In* parameters, and a *Valid* flag. The former two parameters, represent the port instances where this connection is established, while the latter flag asserts the validity of the connection. It is to say, if the connection instance is established between two distinct types of port instances, then the connection is set invalid by the *Builder Core*, and vice versa. Moreover, the *Builder Core* monitors all the connection instances and stores them in the backend as well. Both, the connection and task instances are selectable, thus, they can be removed by triggering the *remove selected* action, described in Section 4.3.5. If a task instance is removed, the *Builder Core* automatically removes all its respective established connection instances as well.

Finally, *Builder Core* provides a data check method, which is accessible by the *Main Window* module when running the *sanity checks*, as described in Section 4.3.4. In addition, another data extraction method is provided, from which the raw data set Γ originates.

This latter set is described verbosely in Section 3.2.1. The extraction method is invoked by the *Main Window* module within the scheduling procedure. The whole procedure is presented in the *Generate Schedule* action described in Section 4.3.5.

4.3.3 Scheduler Core

The modeling process, described in Chapter 3, occurs within the *Scheduler Core*. The *preference settings* parameters are stored in the backend of the *Scheduler Core*. These parameters aid the *Scheduler Core* in modeling the ILP optimization problem. The *PuLP*[1] modeler is used to formulate the optimization problem. The problem is thus passed to a COIN-OR Branch and Cut (CBC)[2] solver. Whenever a scheduling process is requested by the user, the *Scheduler Core* receives the raw data set from the *Main Window* module. The raw data set is only delivered if the *sanity checks* have been successfully completed.

The modeling and solving process for the optimization problem are time consuming. Therefore, if the latter processes are executed within the main thread of the TDCS-Planner instance, the user interface freezes for a significant time. To overcome this issue, the *Scheduler Core* creates another exclusive thread to execute the modeling and solving processes of the ILP problem.

To ensure thread safety, all the attributes necessary for the scheduling procedure, are passed as arguments to the thread constructor. Once the solver is finished, the results are communicated back to the main thread for illustration, using the signal-slot mechanism. The signal-slot mechanism provided by *PyQt*, ensures thread safety as well. The *Scheduler Core* reformulates the last-mentioned results, and thus delivers them to the *Scheduler View* for the demonstration of both the *schedule table* and *scheduler time-line*.

Moreover, the *Scheduler Core* extracts the *Schedule Description* and *Schedule Script*, whenever a successful schedule is generated. These are both stored in the backend of the core and are accessible for the *Main Window* module for exporting, as exhibited in Section 4.4.

[1]PuLP is an LP modeler written in Python: https://pypi.org/project/PuLP/

[2]CBC is an open-source mixed integer linear programming solver written in C++: https://github.com/coin-or/Cbc/blob/master/

4.3.4 Sanity Checks

The test plan is designed by the testing engineer, therefore it is very error-prone. The TDCS-Planner provides intermediate verifications all along the generation process of the *test plan*. Whenever the user requests the generation of the test plan schedule, the stored structured data within the *Builder Core* undergoes an all-inclusive sanity check. This check, makes ensures that:

- All the port instances, of all the task instances, have at least an established connection. Thus, there is no produced argument which is not consumed by a task, and vice versa.
- All the established connections are valid. Therefore, all the data dependencies have identical argument types.

Once the above points are fulfilled, the *Builder Core* can proceed extracting the raw data set.

The TDCS-Planner also provides a complete sanity check for the *project description file*. Though the latter file is originated by the TDCS-Planner, but it is intended to be used by a future tool instance and not specifically the instance of origin. The fetched data, from the *Targets Configuration* files, changes from one instance to another. Therefore, the correctness of the description has to be reviewed from every aspect:

- The used target instances are checked for: matching the configuration parameters with the fetched files, and uniqueness of target parameters (*Label, Color, and CAN ID*).
- The used task instances are checked for: correct referencing with their parent target, matching the configuration parameters including the default values assignment, and uniqueness of the ID_{task}.
- The used connection instances are checked for: correct referencing with their respective tasks, matching of the input/output ports, and uniqueness of connection in case of an input port.

Only when all *sanity checks* successfully pass, the TDCS-Planner proceeds by loading the description file.

Moreover, the TDCS-Planner ensures a complete sanity check for the *schedule script*. The latter script is originated by the tool itself, and it is provided to the MoPS system. Therefore, it is necessary to re-check again the generated script from every aspect:

- The used target instances are checked for: matching the configuration parameters with the fetched files, and uniqueness of targets *CAN IDs*.
- The used variables are checked for: correct referencing in the task instances, and matching of types.
- The task instances are checked for: correct referencing with the targets' instances, configuration parameters, and correct referencing of variables.
- The slots are checked for: correct assignment of target/task instances, and for matching round period with last slot end.

The TDCS-Planner provides the command line execution of the last-mentioned sanity check. The command that has to be run is as follows:

```
tdcs --complete-check <path of script>
```

When running this command, the tool fetches the *Targets Configuration* files, and runs both, the schema and sanity check over the passed script.

4.3.5 Main Actions

Main Actions module is defined over the highest hierarchical level. Its interface is managed by the *Main Window* module, which ensures a global scope access for all the other modules. For the user to access the *Main Actions*, the *Main Window* represents them in the *Menu Bar* and *Tool Bar*. The former is split with respect to the actions categories and represents all the available actions, while the latter only shows the most used actions for easing the user experience. Forthcoming, all the action categories are presented, exhibiting all the actions in each category. Figure 4.14 illustrates the category menus represented by the *Menu Bar*.

File Menu

New Project action creates a new blank project. If there is a project going on within the tool instance, the user is asked to save/discard it before proceeding. Whenever a new project is created, the tool re-fetches the *Targets Configuration* files. This action can be accessed from both, *Menu* and *Tool* bars, along with the action shortcut *Ctrl+N*.

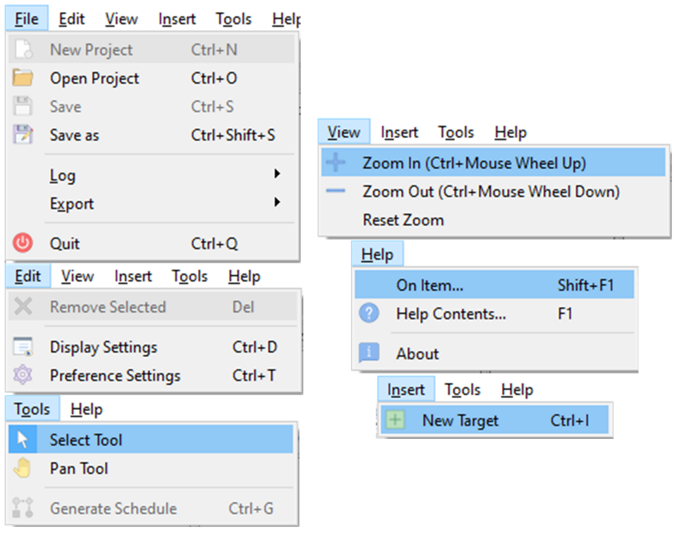


Figure 4.14: Main actions distributed over the category menus.

Open Project action prompts the file explorer dialog, where the user can choose an existing project to load into the tool instance. If there is an open project within the tool instance, the user is asked to save/discard it before proceeding. The *project description file* chosen undergoes a schema verification and the sanity check, described in Section 4.3.4. This action can be accessed from both, *Menu* and *Tool* bars, along with the action shortcut *Ctrl+O*.

Save and *Save As* actions save the *project description file* of the ongoing project, respecting the corresponding schema. *Save* action can be accessed from both, *Menu* and *Tool* bars, along with the action shortcut *Ctrl+S*. *Save As* action can be accessed from both, *Menu* and *Tool* bars, along with the action shortcut as *Ctrl+Shift+S*.

Log/Debugging is a checkable action, which enables the DEBUG level of logging within the file stream. More about the logging streams is shown in Section 4.4.4. This action can be accessed from both, *Menu* and *Tool* bars. *Export/Schedule Description* and *Export/Schedule Script* actions, export the *Schedule Description* and *Schedule Script* respecting the schemas accordingly. Both of the latter actions are only activated, if a schedule has been generated by the user. *Export/Schedule Script* action can be accessed from both, *Menu* and *Tool* bars, along with the action shortcut *Ctrl+E*. *Export/Schedule Description* action can be accessed from the *Menu* bar, and the action shortcut, *Ctrl+Shift+E*.

Quit action invokes the close method of the application. If there is an open project within the tool instance, the user is asked to save/discard it before proceeding. *Quit* action can be accessed from the *Menu* bar, and the action shortcut, *Ctrl+Q*.

Edit Menu

Remove Selected action deletes all the selected items from the *Builder View*. This action is only activated, if the user selects at least one item. The item can be either of both *connection* or *task item*. Any removed *Task Item*, automatically removes its corresponding *Connection Items*. *Remove Selected* action can be accessed from both, *Menu* and *Tool* bars, the *Builder View*'s context menu, and the action shortcut *Del*.

Display Settings and *Preference Settings* actions prompt the *display settings dialog* and *preference settings dialog*, described in Section 4.2.5. *Display Settings* action can be accessed from the *Menu* bar, and the action shortcut, *Ctrl+D*. *Preference Settings* action can be accessed from the *Menu* bar, and the action shortcut, *Ctrl+T*.

View Menu

Zoom In, Zoom Out and *Reset Zoom* actions work on the active view between either of the *Builder View* or the *Scheduler View*. The active view is the selected tab of the tabbified widget in the central window. *Zoom In* and *Zoom Out* actions can be accessed from both, *Menu* & *Tool* bars, along with the scroll wheel of the mouse, scroll up and down respectively. *Reset Zoom* action can be accessed from both, *Menu* and *Tool* bars (by pressing over the scale ratio percentage).

Insert Menu

New Target action prompts the *new target dialog*, described in Section 4.2.1. *New Target* action can be accessed from both, *Menu* and *Tool* bars, the *Targets Tree*'s context menu, and the action shortcut *Ctrl+I*.

Help Menu

On Item action activates the *Whats This* mode of the application. The latter is when the user can click on any desired component of the tool, and gets a customized help message. *On Item* action can be accessed from the *Menu* bar, and the action shortcut *Shift+F1*. *Help Contents* prompts the *help browser*, described in Section 4.2.5. *Help Contents* action can be accessed from the *Menu* bar, and the action shortcut *F1*. *About* action prompt an *about dialog*, describing the application licenses along with version of the tool. *Help Contents* action can be accessed from the *Menu* bar.

4.4 Data Base

The data interface of the tool contains both input and output channels. The first is through which the tool fetches the *Targets Configuration* files, the *Configuration Files* for the tool, and the pictures used within the GUI. The input channels are necessary for an application instance to be functional. The output channels are features provided by the TDCS-Planner, categorized by two formats, one for the logging messages and another for the descriptive files. All the descriptive data interfaces of the tool are chosen to be JSON format. Section 2.1 argues the choice of JSON format.

4.4.1 Configuration Files

On the startup of a TDCS-Planner instance, the application searches within the predefined configuration directory for three folders containing:

- The logging configuration file, which configures the application logger.
- The schema files, for the descriptive data interfaces which are JSON format. There is a schema for the *Targets Configuration, the Project Description File, the Schedule Description* and *the Schedule Script.*
- The compressed HTML files describing the help contents, in a binary Qt Help Collection (QHC) format. This allows shrinking the data size requirement of the tool, instead of having all the HTML files.

4.4.2 Target Description

Whenever a new project is started within an application instance, the tool fetches the *Targets Configuration* files, described in Section 2.3.1. The TDCS-Planner searches for the last-mentioned files in a predefined directory, and filters them to read only the JSON files among the available ones. The obtained files undergo a schema check to verify the correctness of the expected data structure. Once this check is passed, the tool proceeds by fetching the data from the *Targets Configuration.* The latter data is utilized, by the TDCS-Planner, for describing any target instantiated within the ongoing project.

4.4.3 Project Description File

The *Project Description File* is originated by the tool, it describes completely the project while avoiding redundancy. The project description is generated respecting the predefined JSON schema. Thus the new file extension is created (.tdcs), of format Time-Driven Communication Schedule (TDCS). This last-mentioned format contains:

- The preference settings chosen for the project.
- All the targets' instantiated in the *Targets Tree*, along with all the compiled parameters, and the state of the tree items (expanded/collapsed).
- All the items instantiated in the *Builder View*, along with all the necessary interconnections between the items.

This file is generated by the TDCS-Planner, whenever the user saves an ongoing project.

4.4.4 Logging

The application defines one logger, which contains two handlers. The first is a file handler, directed to *log.tdcs.log*. The location of the latter is the working directory of the tool instance. The detailed formatter of the file handler is as follows:

Date & Time – Module Name (line #) – Function Name – Level Name – Messages

The second is a stream handler, directed to the *Prompt View*. The formatter of the second handler, is a simple formatter as follows:

Date & Time – Level Name – Messages

By default, the log level is set to *INFO* level, which means that the *DEBUG* logs are not handled by any handler. Once the *DEBUG* level is checked by the user, the file handler starts handling the *DEBUG* logs.

4.4.5 Schedule Description

Once a successful schedule is generated, the *Schedule Description* is formulated. The latter describes all the schedule parameters, along with the timing assignments. The *Schedule Description* is generated respecting the predefined JSON schema. Thus the new file extension is created (.sd), of format SD (Schedule Description). The description exhibits the following:

- The scheduler parameters, declared by the user within the *preference settings*, for producing the schedule.
- The computed period of the schedule.
- The slots timing description: *start, overlap bound, buffer and end.*
- The tasks assigned within each slot, stating all its attributes: *name, parent target, WCET, type and default inputs (if any).* Also the computed *start time-instant, and the transmission time (if any).*

4.4.6 Schedule Script

The *Schedule Script* is a JSON format file, which serves as an input for the MoPS system. Therefore, it does not explain all the schedule parameters neither all the tasks used within the project generating the schedule. The script contains all the necessary information for managing the communication network Thus, *Free Tasks* are not mentioned in the script, even if they are part of the project of origin. The *Schedule Script* is generated respecting the predefined JSON schema. Thus, the new file extension is created (.ss), of format SS (Schedule Script). The script exhibits the following:

- From the used targets, only the parents of *producer* or *consumer* tasks used within the project.
- The communication network links, which are the variables of the *producer* or *consumer* tasks used within the project.
- The *producer* or *consumer* tasks used within the project.
- The slots time-instants, describing the time-line assignment of the targets and tasks, above mentioned.

CHAPTER

5 Evaluations

This chapter covers the evaluation of the *implementation* of the TDCS-Planner. First, a complete *test plan* is demonstrated in the *Results* section. In addition, a discussion is made on the efficiency of the test plan design along with the scheduling approach. Finally, the proposed *improvements* are presented, which are integrated into the TDCS-Planner.

5.1 Results

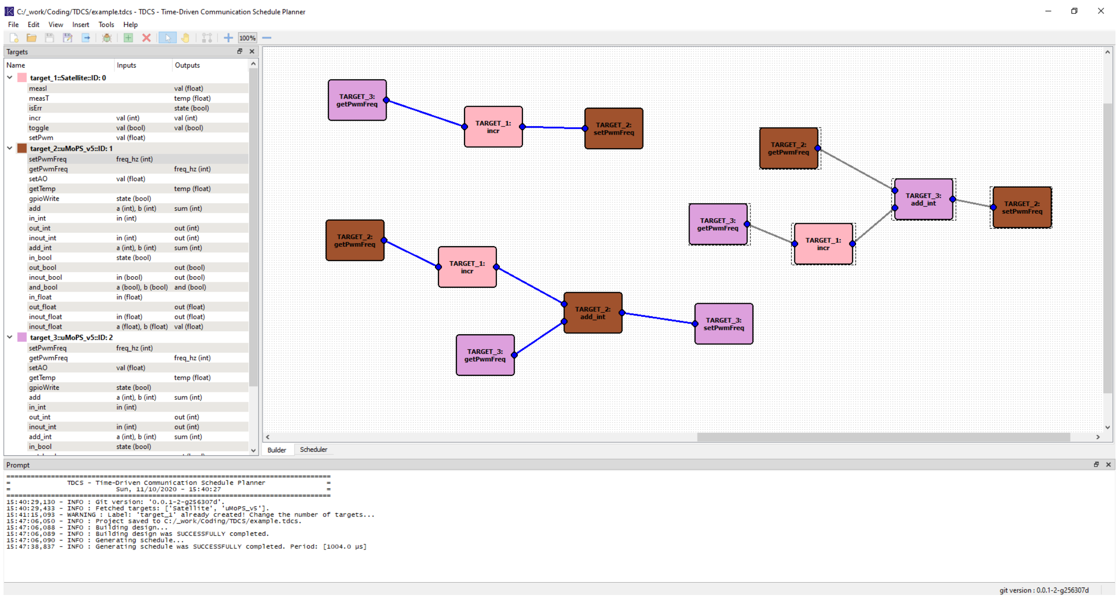
The problem statement of this thesis is divided into three areas, as described in Section 1.3. This section presents how the research questions are answered. The answers are exhibited in coherence with the division of the problem statement.

5.1.1 Test Plan Builder

The implementation of the *test plan* builder, resulted with an intuitive user interface. Figure 5.1 shows the GUI of the builder. On the left side, the targets window is depicted. Within this window, the user is able to visualize all the instantiated targets, in the form of a hierarchical tree. In this way, the user can easily understand the parent-child relation between the *tasks* and the *targets*.

On the left side, there is a wide scene area, for designing the precedence relations, required for the *test plan*. Therefore, the user can easily drag and drop the tasks over the scene, and define the DAG by simple clicks between the ports of the tasks. The test plan design, can become very complex, thus, the more space, the better it is for the user. The builder scene can be zoomed in/out, for easing the design process. Moreover, the builder window can be made the only window of the TDCS-Planner, i.e., all the other windows can be hidden, as shown in Figure 5.2. Hence, more space is offered for drawing the DAG of the test plan.

On the bottom side, the prompt window is depicted. Within this window the user can see the log messages, while progressing with the design procedure.



Name	Inputs	Outputs
target_1::Satellite::ID: 0		
measI		val (float)
measT		temp (float)
isErr		state (bool)
incr	val (int)	val (int)
toggle	val (bool)	val (bool)
setPwm	val (float)	
target_2::uMoPS_v5::ID: 1		
setPwmFreq	freq_hz (int)	
getPwmFreq		freq_hz (int)
setAO	val (float)	
getTemp		temp (float)
gpioWrite	state (bool)	
add	a (int), b (int)	sum (int)
in_int	in (int)	
out_int		out (int)
inout_int	in (int)	out (int)
add_int	a (int), b (int)	sum (int)
in_bool	state (bool)	
out_bool		out (bool)
inout_bool	in (bool)	out (bool)
and_bool	a (bool), b (bool)	and (bool)
in_float	in (float)	
out_float		out (float)
inout_float	in (float)	out (float)
inout_float	a (float), b (float)	val (float)
target_3::uMoPS_v5::ID: 2		
setPwmFreq	freq_hz (int)	
getPwmFreq		freq_hz (int)
setAO	val (float)	
getTemp		temp (float)
gpioWrite	state (bool)	
add	a (int), b (int)	sum (int)
in_int	in (int)	
out_int		out (int)
inout_int	in (int)	out (int)
add_int	a (int), b (int)	sum (int)
in_bool	state (bool)	

```
==========================================================================
=               TDCS - Time-Driven Communication Schedule Planner        =
=                          Sun, 11/10/2020 - 15:40:27                    =
==========================================================================
15:40:29,130 - INFO : Git version: '0.0.1-2-g256307d'.
15:40:29,433 - INFO : Fetched targets: ['Satellite', 'uMoPS_v5'].
15:41:15,093 - WARNING : Label: 'target_1' already created! Change the number of targets...
15:47:06,050 - INFO : Project saved to C:/_work/Coding/TDCS/example.tdcs.
15:47:06,088 - INFO : Building design...
15:47:06,089 - INFO : Building design was SUCCESSFULLY completed.
15:47:06,090 - INFO : Generating schedule...
15:47:38,837 - INFO : Generating schedule was SUCCESSFULLY completed. Period: [1004.0 µs]
```

Figure 5.1: Test plan builder.

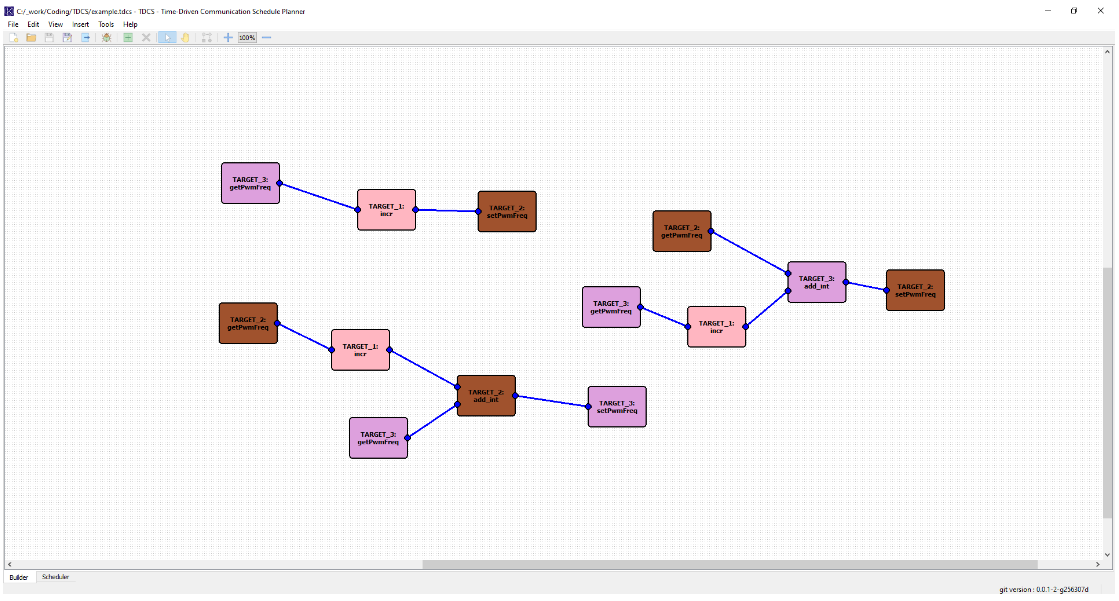
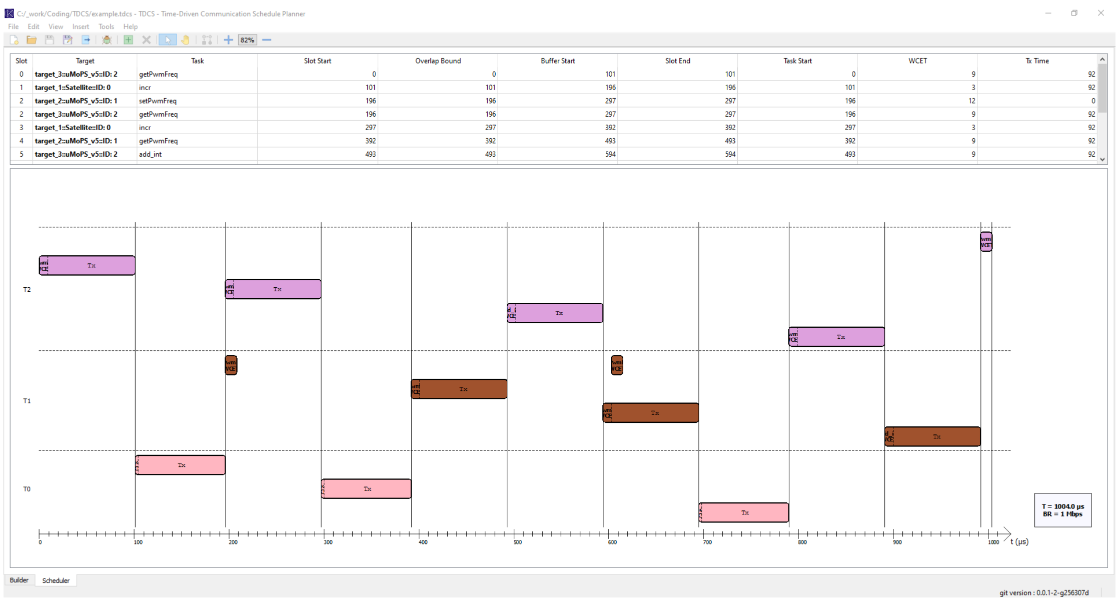


Figure 5.2: Extended builder scene.

5.1.2 Test Plan Schedule

The scheduling approach elaborated within this thesis, provided a convenient model for the real-time *test plan*. The optimal schedule time-line is achievable within negligible execution time, for the simple *test plan* designs. Figure 5.3 demonstrates the output schedule time-line, for the *test plan* shown in Figure 5.1. The displayed schedule time-line is optimal with respect to minimizing maximum latency. Notice that the resolution is narrowed for showing the complete time-line at once without the need to scroll.



Slot	Target	Task	Slot Start	Overlap Bound	Buffer Start	Slot End	Task Start	WCET	Tx Time
0	**target_3::uMoPS_v5::ID: 2**	getPwmFreq	0	0	101	101	0	9	92
1	**target_1::Satellite::ID: 0**	incr	101	101	196	196	101	3	92
2	**target_2::uMoPS_v5::ID: 1**	setPwmFreq	196	196	297	297	196	12	0
2	**target_3::uMoPS_v5::ID: 2**	getPwmFreq	196	196	297	297	196	9	92
3	**target_1::Satellite::ID: 0**	incr	297	297	392	392	297	3	92
4	**target_2::uMoPS_v5::ID: 1**	getPwmFreq	392	392	493	493	392	9	92
5	**target_3::uMoPS_v5::ID: 2**	add_int	493	493	594	594	493	9	92

T = 1004.0 μs
BR = 1 Mbps

git version : 0.0.1-2-g256307d

Figure 5.3: Test plan schedule.

After generating a successful schedule, the scheduler window replaces the builder scene in the main window automatically. In the scheduler window, the schedule is described, both quantitatively and graphically. In this context, the user gets a full understanding of the generated schedule. The schedule time-line is visualized, in the schedule scene, in an intuitive representation. Within the schedule scene, the user can navigate, and zoom in/out through the time-line. Also, the user can distinguish the type of tasks, from their respective drawing on the time-line, as stated in Section 4.2.3. Moreover, the scheduler window can be made the only window of the TDCS-Planner, similarly to the builder window. As a consequence, the approach for representing the schedule time-line resulted in a very smooth user experience.

The main complication of the research question relative to this section, is how to solve an NP-complete problem. This thesis was able to provide a very convenient model for formulating the project specific problem. The methodology followed narrows the search space as much as possible, while respecting the precedence relations. However, for complex test plans, the resulted optimization problem is still time consuming. Therefore, more improvements have to be done within the solver scope, which is not part of this thesis work. In this thesis, the solver used is the CBC solver, since it is the only free solver available.

5.1.3 Error Proof

The *test plan* design process, proposed by this thesis, is completely error-proof. The possible errors are avoided, all along the building process, by directing the user during the *test plan* procedure. The *sanity checks* performed over the DAG, designed within the builder, ensure design correctness before generating any schedule.

The implementation of the TDCS-Planner provides complete *sanity checks* for: *targets configuration*, *description file*, and *schedule script*. Therefore, the data files can be re-checked again, and detect any possible malware. Listing 5.1 shows a simple example of the output message from the TDCS-Planner, in the erroneous case. Notice how the user is directed for easily fixing the error found, within the checked file.

Listing 5.1: Erroneous case example.

```
14:45:25,163 - WARNING : Target config C:\_work\Coding\TDCS\
    data\targets\uMoPS_v5.json doesn't meet schema
    requirements!

'intt' is not one of ['int', 'float', 'bool'].

On instance['tasks'][0]['in'][0]['type']:
"intt"

14:45:25,165 - ERROR : Could NOT fetch target's config file C
    :\_work\Coding\TDCS\data\targets\uMoPS_v5.json!
```

5.2 Improvements

During developing, the implementation of TDCS-Planner got very intuitive after several reviews and productive usages from a small user group. The improvements done can be summarized as follows:

- The objective function, of the *optimization problem*, is defined as the accrual of the TUFs, of each task. The TUFs of the *producer tasks* is multiplied by a weight factor, for emphasizing the importance of the *producer tasks*. Therefore, the latency of any *producer task* is more significant with respect to the objective function. This helps, in getting the optimal scheduler faster.

- The automatic *CAN ID* assignment for a target instance, eases the creation of a target instance, and directs the user towards occupying the smallest integers first.

- Zooming feature was added for both of the views of the TDCS-Planner, in a sense that the user can visualize the percentage zoom ratio.

- Keyboard shortcuts were added to almost all actions of the TDCS-Planner, also, the key arrows can be used for navigating through the builder view.

- Resolution feature was added for the schedule time-line, to better visualize the tricky time-lines (too small/long).

CHAPTER 6 Conclusion & Outlook

This chapter presents the work summary of this thesis, by which the problem statement questions are answered. In addition, an outlook is exhibited where some possible improvements and additional implementation ideas are covered.

6.1 Thesis Contributions

A desktop application has been developed within this thesis. The application covers the three division areas which emerged from the problem statement (see Section 1.3). The divisions can be recalled roughly: the design framework (*builder*), the scheduling technique (*scheduler*), and the error resistance and verifications. The first two, *builder and scheduler*, are the main structures of the developed tool. While the last one, *error resistance*, enfolds both of the latter structures, thus, all the logical operations of the application have been manipulated to be error resistant as much as possible.

The implemented *builder* solves the design complications, it provides a very user friendly methodology for drawing the DAG defining the precedence relations between the tasks. Moreover, the designed framework translates the distributed system architecture into a hierarchical tree with colorful representations of the tasks. In this context, the design of the *test plan* gets very simple (drag and drop, and connect), even if the architecture of a distributed system is known to be very complicated.

The developed scheduling approach aimed in modeling the scheduling problem into a less complex optimization problem. The proposed system model succeeded in narrowing the search space which eases the optimization process. The model exploited the *time utility* concept within the objective function of the problem, therefore, the obtained schedule is optimal with respect to minimizing maximum latency. In addition, the *time slots* are formulated within the developed scheduling approach, which ensures the exclusivity of the communication network while respecting the precedence constraints. The formulated *time slots* can be both, *fixed* or *flexible*. The implementation provides, also, the concept of *buffering* which serves in obtaining a more dependable schedule time-line. Thus, it can be concluded that the generated schedule time-line succeeded in exploiting determinism and dependability. However, the execution time required for obtaining such a schedule, for a complex test plan, is still not negligible, which opens a window for improvements.

Ultimately, the developed application provides several *sanity checks* all along the test plan generation procedure. The *sanity checks* are made accessible from the console, which makes the application accessible to the SW application programming interface (API) of the parent project MoPS system. As a consequence, the verification and validation, of the generated test plan description and schedule, is achieved within this thesis.

6.2 Future Works

The implementation of the proposed tool works pretty well, however there is always place for improvements. The main improvements concern the speeding of execution of the optimization problem. Within this thesis, the CBC solver is used, since it is the only free solver available. The CBC solver spends too much time finding the best solution when the problem is too symmetrical. On the other hand, there are several commercial solvers that can be used which already surpass the symmetry issue. Recall that for modeling the optimization problem, the *PuLP* modeler is used. The *PuLP* modeler has an already available API for several commercial solvers and not only for CBC. Some of the commercial solvers which are worth trying are stated below:

Gurobi

The Gurobi Optimizer is a commercial optimization solver for several type of problems. Gurobi offers an academical license, which can be obtained compiling the license form.

CPLEX

The CPLEX Optimizer is provided by IBM. It is a commercial solver for several mathematical problems. The academic license can be obtained as well, through the Academic Initiative program offered by IBM.

Moreover, improvements can be done regarding the user interface. A command line is quasi-developed already within the course of this thesis, but its integration into the TDCS-Planner is yet to be done. This command line can be easily integrated within the *prompt* window of the application.

Acronyms

µC	micro controller
API	Application Programming Interface
BR	Baud Rate
CAN	Controller Area Network
CBC	COIN-OR Branch and Cut
CPU	Central Processing Unit
DAG	Directed Acyclic Graph
DUT	device under test
EDD	Earliest Due Date
EDF	Earliest Deadline First
FSM	finite-state machine
GUI	graphical user interface
HTML	Hyper Text Markup Language
HW	hardware

ILP	Integer Linear Programming
JSON	JavaScript object notation
KAI	Kompetenzzentrum Automobil- und Industrie-Elektronik
LDF	Latest Deadline First
MIME	Multipurpose Internet Mail Extensions
MoPS	modular power stress
QHC	Qt Help Collection
SoC	System on Chip
SW	Software
TDCS	Time-Driven Communication Schedule
TDCS-Planner	Time-Driven Communication Schedule Planner
TP-Builder	test plan builder
TUF	Time Utility Function
WCET	Worst Case Execution Time

CHAPTER 6

Bibliography

[1] S. R. Vock, O. Escalona, C. Turner, and F. Owens, “Challenges for semiconductor test engineering: A review paper”, *Journal of Electronic Testing*, vol. 28, pp. 365–374, 2012.

[2] Y. H. Ng, Y. H. Low, and S. Demidenko, “Improving efficiency of ic burn-in testing”, in *2008 IEEE Instrumentation and Measurement Technology Conference*, IEEE, 2008, pp. 1685–1689.

[3] B. Steinwender, “A Distributed Controller Network for Modular Power Stress Tests”, PhD thesis, Alpen-Adria-Universität Klagenfurt, Jun. 2016.

[4] W. Elmenreich, G. Bauer, and H. Kopetz, “The time-triggered paradigm”, in *Workshop on Time-Triggered and Real-Time Communication Systems*, 2003. [Online]. Available: https://mobile.aau.at/~welmenre/papers/2003/rr-54-2003.pdf (visited on 2017-11-02).

[5] K. Plankensteiner, “Test Plan Generation and Verification for a Modular Power Stress Test System”, Master’s thesis, Graz University of Technology, 2015.

[6] S. Einspieler, B. Steinwender, and W. Elmenreich, “Flexible real-time control and diagnostic system for application related stress testing”, Unpublished thesis, PhD thesis, Alpen-Adria-Universität Klagenfurt, 2021.

[7] D.-T. Peng, K. G. Shin, and T. F. Abdelzaher, “Assignment and scheduling communicating periodic tasks in distributed real-time systems”, *IEEE Transactions on Software Engineering*, vol. 23, no. 12, pp. 745–758, 1997.

[8] C. Severance, “Discovering javascript object notation”, *Computer*, vol. 45, no. 4, pp. 6–8, 2012.

[9] B. Steinwender, S. Einspieler, M. Glavanovics, and W. Elmenreich, “Distributed power semiconductor stress test & measurement architecture”, English, in *Proceedings of the 11th IEEE International Conference on Industrial Informatics*, Bochum, Germany, Jul. 2013, pp. 129–134. DOI: 10.1109/INDIN.2013.6622870.

[10] A. Prabhakara, B. Steinwender, and W. Elmenreich, "Fail-silent strategy to counteract message timing violation and error causing nodes through execution timing analysis of a distributed hard real-time system", Unpublished thesis, PhD thesis, Alpen-Adria-Universität Klagenfurt, 2021.

[11] C. Liu, "Fundamentals of real-time scheduling", in *Real Time Computing*, Springer, 1994, pp. 1–7.

[12] K. Ecker, "Algorithmic methods for real-time scheduling", in *Real Time Computing*, Springer, 1994, pp. 9–29.

[13] G. C. Buttazzo, *Hard Real-Time Computing Systems: Predictable Scheduling Algorithms and Applications*. Springer Science & Business Media, 2011, ISBN: 978-1-4614-0675-4. DOI: 10.1007/978-1-4614-0676-1.

[14] E. W. Weisstein, *NP-Problem*, From MathWorld - A Wolfram Web Resource, Sep. 2020. [Online]. Available: https://mathworld.wolfram.com/NP-Problem.html (visited on 2020-09-22).

[15] S. S. Craciunas, R. S. Oliver, and V. Ecker, "Optimal static scheduling of real-time tasks on distributed time-triggered networked systems", in *Proceedings of the 2014 IEEE Emerging Technology and Factory Automation (ETFA)*, IEEE, 2014, pp. 1–8.

[16] S. S. Craciunas and R. S. Oliver, "Combined task-and network-level scheduling for distributed time-triggered systems", *Real-Time Systems*, vol. 52, no. 2, pp. 161–200, 2016.