Scheduling Strategies for Event-Triggered Control Using Timed Game Automata Over CAN Networks

Aniket Ashwin Samant

Master of Science Thesis

TU Delft Delft University of Technology

Delft Center for Systems and Control

Scheduling Strategies for Event-Triggered Control Using Timed Game Automata Over CAN Networks

Master of Science Thesis

For the degree of Master of Science in Embedded Systems at Delft University of Technology

Aniket Ashwin Samant

August 20, 2020

Faculty of Electrical Engineering, Mathematics, and Computer Science (EEMCS) · Delft University of Technology

Abstract

Modern times have seen an increasing use of networked control systems, in which plants and controllers may not necessarily have a direct link but instead be connected through a network, thereby closing control loops over multiple nodes. The system may also be spread out spatially over a large area, and thus the associated network delays could greatly hamper control performance, potentially affecting the closed-loop stability of the system.

In such scenarios, event-triggered control approaches could greatly reduce network congestion by allowing a means for the controllers to send control loop computation packets over the network only when required, in an event-driven manner, rather than through periodic transmissions. However, in practice, the number of parallel channels is limited compared to the number of controllers and hence the transmission of packets needs to be scheduled carefully to avoid network conflicts.

This thesis explores using a network of timed (game) automata composed of models representing a networked control system's control loops and its communication network. This reduces the scheduling problem of transmission of control loop computations to one of creating strategies using known algorithms, with the objective being to avoid network conflicts brought about by simultaneous transmissions. Furthermore, the proposed automata models also aim to reduce the conservatism of generated scheduling strategies by allowing the control loops a bounded number of retransmission attempts to send packets over the network in case it is already occupied. The concept is finally demonstrated in practice using simulated plants and controllers distributed over multiple machines connected via a physical CAN network.

Table of Contents

List of Figures

List of Tables

Acknowledgements

I would like to thank my supervisor Dr. Manuel Mazo Jr. for his supervision during the writing of this thesis. This period has seen some exceptionally testing times and it would have been really difficult to make progress without his strong support and guidance. I would like to thank his entire research group as well for their support and some really interesting discussions on technical as well as non-technical topics. In particular, I would like to thank ir. Gabriel Gleizer; though he wasn't my official supervisor, I received a lot of timely help and feedback from him and I really appreciate it.

I would like to thank all my friends and family for their continued support throughout my study programme, and in particular during my thesis. It was a period of intense ups and downs, but perhaps that is how it should be. The love and support offered by everyone was much required for strength and it would have been very difficult to get through without their involvement.

I would like to thank Dr. Peyman Mohajerin Esfahani and Dr. Mitra Nasri for being part of my thesis committee and assessing my technical contributions. Critical feedback is essential in any kind of research and I highly value their input.

Delft, University of Technology
August 20, 2020

Aniket Ashwin Samant

"In the future, airplanes will be flown by a dog and a pilot. And the dog's job will be to make sure that if the pilot tries to touch any of the buttons, the dog bites him."

— *Scott Adams*

Chapter 1

Introduction

The field of control engineering has evolved greatly over the past few decades, having seen a gradual transition from the usage of classical control approaches involving frequency-domain analyses, towards modern control theory employing state-space methods for system analysis and control.[10]

Around the same time, a steep increase in computational processing power brought about by the digital revolution was followed by the concept of a Networked Control System (NCS) being studied and developed - a system involving a distributed arrangement of sensors, actuators, and controllers over a network. As illustrated in Figure 1-1, NCSs essentially involve closing control loops over a communication network between the plants and the controllers; control actions computed by the controllers are transmitted to actuators via the network.[1]

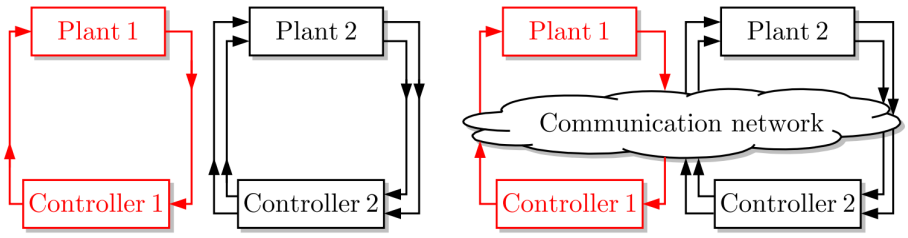

Figure 1-1: An illustration showing a comparison between a traditional control system (left) and a networked control system (right) - in the latter, the control loops are closed over the communication network [1]

Intuitively, having an additional component between a plant and a controller adds to delays in applying control actions, brought about by the latency of the network. Moreover, in most applications, it is very expensive to have a dedicated channel between each plant and controller (which may be distributed over a large area), and thus we resort to *a number of bandwidth-limited common digital communication channels shared by the entire network.* Owing to varying network loads, information transmission between nodes in such a setup is

non-deterministic; in a control context, this could even lead to instability [11]. This motivates the need to

- use a control law that requires a reduced number of computations as compared with the typical, periodic type of control, and
- schedule transmission of information packets over the network appropriately, to ensure that control actions are applied to plants without being greatly affected by delays due to network congestion.

In such a case, Event-Triggered Control (ETC) is greatly beneficial. It is a control approach in which a *triggering criterion (or criteria)* based on sensor measurements decides when a control computation needs to be performed for a certain actuator. Accordingly, the actuator output gets updated and is maintained in a zero-order hold (ZOH) manner until the next update. The control loops thus need not be updated periodically as in the traditional case but based on the occurrence of the triggering events, and hence this scheme saves essential bandwidth as a result of fewer computations.

Since the number of channels between the components is limited, we need to schedule these computations based on various criteria - that is to say, we need to have a scheduler that decides which control loop to update at a certain time instant based on its knowledge of events received from various sensors. Figure 1-2 shows an NCS scheme with a scheduler involved in the system.

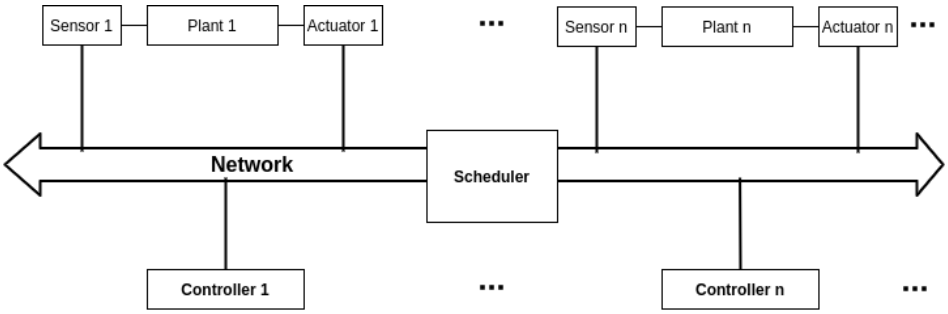

Figure 1-2: An NCS arrangement involving a scheduler in the network loop. The scheduler needs to have all the sensor data necessary for scheduling control actions [2]

Several scheduling policies have been developed over time in the context of embedded systems (such as FP, EDF, RR, etc.) that suit various applications and the interested reader can refer to the literature for further understanding [12, 13]. In this thesis, we deal with a scheduler that can be implemented on a distributed control system in which the components are connected over a single CAN network for communication.

The scheduler is designed by making use of *abstractions* (formal logical representations) of the control loops and the communication network created using the concept of timed automata [14], as will be introduced in the forthcoming chapters. The scheduling policy needs to ensure that control actions are transmitted such that network conflicts are avoided and the stability of the individual control loops is maintained. It is hence very important that the physical control loops and the network are modelled correctly in this approach.

In [2], abstraction models representing the control loops and the network are created and scheduling strategies are generated accordingly. However, with these models, there is no possibility of a case in which a control loop attempts to access a busy network and waits till it becomes free for use - that is to say, a control loop does not attempt to *retransmit* its data over the network once it is free. The models treat simultaneous requests to access the network as undesirable and are hence *conservative*; a modification of the models can be made to address this concern. Additionally, [2] also raises a point about the suitability of such models for an implementation on physical networks.

Based on this understanding, we introduce the two main objectives of this thesis - one theoretical and one practical:

- to create timed automata models that incorporate the concept of limited *retransmission* of control actions over the communication channel in case it is already occupied; this is a theoretical extension of models introduced in [2], and,
- to demonstrate that the scheduler functions as expected in an actual NCS environment - one involving simulated control loops running on machines connected over a physical network in the form of a common CAN network.

Through the chapters, we cover how the simple models introduced in [2] are built upon to have new models that take into consideration retransmission of control actions for generating scheduling strategies. We also implement these models on separate machines connected over a common CAN network and perform scheduler synthesis [9], showing results obtained by applying strategies generated using the new models.

1-1 Notation

$\mathbb{R}^n$ refers to the n-dimensional Euclidean space. $\mathbb{N}$ denotes the set of all positive integers, and $\mathbb{N}_0$ refers to the set of natural numbers including 0 (i.e. positive integers and 0). $\mathbb{R}^+$ denotes positive reals. $|A|$ refers to the 2-norm of a vector $A \in \mathbb{R}^n$. An empty set is represented by $\emptyset$. The set of all closed intervals over $\mathbb{R}^+$ is represented by $\mathbb{IR}^+$. $\xi_x : \mathbb{R}_0^+ \to \mathbb{R}^n$ is the solution to the initial value problem with $\xi_x(0) = x$, given an ODE of the form $\dot{\xi}(t) = f(\xi(t))$. The *power set* of a set A is the set of all its subsets, denoted by 2^A. The set of all real-valued $n \times p$ matrices is given by $\mathcal{M}_{n\times p}$, and of all $n \times n$ real-valued symmetric matrices by $\mathcal{M}_n$.

1-2 Organization

There are several concepts involved in achieving the thesis's objectives, and they will be introduced in a logical sequence over the following chapters:

- Chapter 2 covers the preliminary knowledge necessary for understanding relevent concepts - ETC, abstractions, timed automata, and the Controller Area Network (CAN) protocol,

- Chapter 3 covers modelling of the network and the control loops as Timed Game Automata (TGA) and their parallel composition as a Network of Timed Game Automata (NTGA), and how they are used for scheduling,
- Chapter 4 explains the software and hardware involved in the implementation of the theoretical ideas, and the simulation environment in which the scheduler is tested,
- Chapter 5 shows some results obtained from running the NCS simulations and discusses them, and also some implications of increasing the number of states involved in the system, and,
- Chapter 6 provides conclusions and directions for future work based on this thesis.

Figure 1-3 shows an overview of the main parts involved in this thesis. The main contributions, in line with the objectives, are highlighted in it.

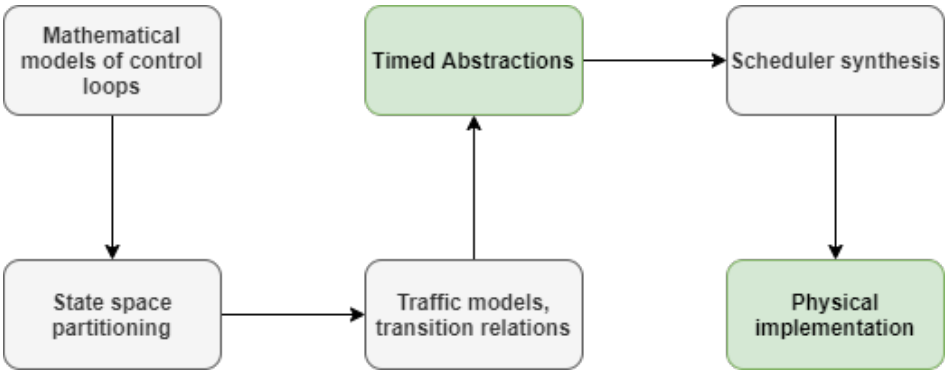

Figure 1-3: A top-level diagram of the flow of this thesis. The main contributions are represented by green blocks.

Chapter 2

Theoretical Preliminaries

This chapter covers the foundational concepts required for understanding the main objective of this thesis. The preliminaries are broadly categorized into the following topics:

- (Periodic) event-triggered control (ETC),
- Traffic abstractions,
- Timed automata, and,
- Controller Area Network (CAN) bus.

Each of these topics will be covered in the sections that follow, and the subsequent chapters will build upon the concepts towards the main thesis objectives.

2-1 Event-Triggered Control

Event-triggered control can generally be applicable to any control system in principle, but in this thesis we consider only Linear Time-Invariant (LTI) systems. In line with [2] we use the following state space model for describing a typical LTI system:

$$\dot{\xi}(t) = A\xi(t) + Bv(t), \quad \xi(t) \in \mathbb{R}^n,\ v(t) \in \mathbb{R}^m. \tag{2-1}$$

Here, the A and B matrices have the appropriate dimensions determined by the dimensions of the state vector ξ and the input vector v.

We consider a linear state-feedback law:

$$v(t) = K\xi(t). \tag{2-2}$$

The gain, K, is calculated (using familiar methods like the LQR algorithm) to make the overall closed-loop system asymptotically stable; K's dimensions are determined as per A's

and B's. This feedback law is implemented in a zero-order-hold manner, and accordingly, in terms of discrete time samples, it is given by

$$v(t) = K\xi(t_k), \quad t \in [t_k, t_{k+1}),\ k \in \mathbb{N}_0, \tag{2-3}$$

such that $t_0, t_1, ...$ is a divergent sequence of update times. Practically, there may be delays involved due to various factors (for instance, delays in sensor readout), but we assume no delays at this point. Interested readers may consult [15] for further reading.

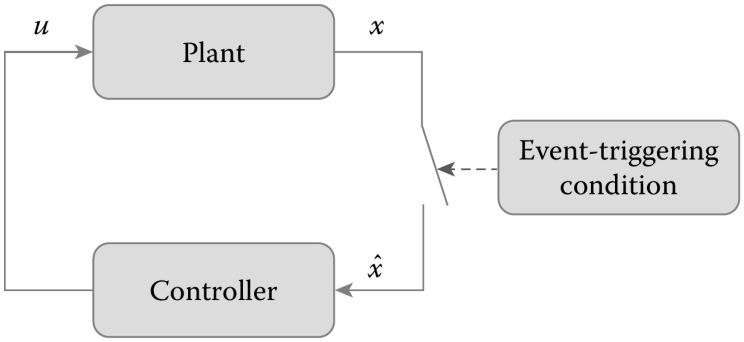

Figure 2-1: An ETC scheme - the held value of the control input u(t) is updated to $Kx(t)$ from $K\hat{x}(t)$ if the event-triggering condition is satisfied [3]

ETC involves making use of a control law for determining the next update instant (t_{k+1}) online, based on the current state of the plant. Knowing that the plant follows the dynamics represented by Eq. (2-1), taking the current state to be $\xi(t)$ and the last sampled state to be $\xi(t_k)$, we define an auxiliary variable:

$$e(t) := \xi(t) - \xi(t_k), \quad t \in [t_k, t_{k+1}),\ k \in \mathbb{N}_0. \tag{2-4}$$

Intuitively, this variable represents how the state evolves from its last recorded sample. Based on the event-triggering approach proposed in [15], it is used to calculate when the next triggering should occur, as given by the following sampling triggering law:

$$t_{k+1} = min\{\, t \,|\, t > t_k \text{ and } |e(t)|^2 \geq \alpha|\xi(t)|^2\}, \tag{2-5}$$

where $\alpha \in (0, 1)$ is an appropriately chosen *triggering coefficient* used to establish a tradeoff between convergence rate and the rate of triggering [15].

Based on these equations and assumptions, we can formulate $\tau_\alpha(x)$, the *inter-sample time* of a sampled state x as follows:

$$\tau_\alpha(x) = min\{\, t \,|\, |e(t)|^2 \geq \alpha|\xi(t)|^2 \ and\ \xi(0) = x\} \tag{2-6}$$

In [15], it is shown that for linear plants with state-feedback controllers, a minimum non-zero inter-event time is always guaranteed to exist.

Following [16], we take $\xi(t)$ as the solution of an LTI system in the sampling interval $[t_k, t_k+\sigma]$ with x as the initial condition (that is, $x = \xi(t_k)$), and $e(t)$ from Eq. (2-4). Thus, we have

$$\xi(t_k + \sigma) = \Lambda(\sigma)x, \tag{2-7}$$

$$e(t_k + \sigma) = [I - \Lambda(\sigma)]x, \tag{2-8}$$

where

$$\Lambda(\sigma) := [I + \int_0^\sigma e^{Ar} dr(A + BK)]. \tag{2-9}$$

Based on these formulations, we can express the inter-sample time introduced in Eq. (2-6) as:

$$\tau_\alpha(x) = min\{\, \sigma > 0 \,|\, x^T \Phi(\sigma) x = 0 \,\}, \tag{2-10}$$

where

$$\Phi(\sigma) := [I - \Lambda^T(\sigma)][I - \Lambda(\sigma)] - \alpha \Lambda^T(\sigma)\Lambda(\sigma). \tag{2-11}$$

We can thus see that given an LTI plant's current state, its inter-event time can be calculated using a series of matrix operations formulated in Eq. (2-10). This information is essential for partitioning the state-space of the plant, as will be covered in Section 2-2.

The method presented so far is one of several approaches to introduce the basic mathematical concepts involved in formulating event-triggered control problems. Interested readers may refer to [17, 18, 19, 20] for further reading on more ETC approaches in various scenarios, and to also understand a similar concept called Self-Triggered Control.

In this thesis, we implement a special case of ETC, called PETC; it is introduced briefly in the next section.

2-1-1 Periodic Event-Triggered Control

The ETC approach introduced thus far deals with monitoring a condition in continuous time to determine whether triggering needs to take place, and is also referred to as continuous ETC (CETC). In [3], a class of ETC called Periodic Event-Triggered Control (PETC) is introduced - the basic principle being to sample the state measurements *periodically*, that is, at $t_k = kh$, $k \in \mathbb{N}$, where $h > 0$ is a carefully chosen sampling interval.

In LTI systems, the control law represented by Eq. (2-3) is applicable to PETC systems as well, but with a slightly different representation for the control input,

$$\hat{v}(t) = K\hat{x}(t),\ t \in \mathbb{R}^+ \tag{2-12}$$

where $\hat{x}$ is a left-continuous signal given by

$$\hat{x}(t) = \begin{cases} x(t_k), & for \quad \mathcal{C}(x(t_k), \hat{x}(t_k)) > 0, \\ \hat{x}(t_k), & for \quad \mathcal{C}(x(t_k), \hat{x}(t_k)) \leq 0 \end{cases} \tag{2-13}$$

Here, $t \in (t_k, t_{k+1}]$, $k \in \mathbb{N}$, and $\mathcal{C}(x(t_k), \hat{x}(t_k))$ is a mathematical expression representing the triggering condition based on the sampled state $x(t_k)$. Accordingly, Figure 2-1 is also applicable to PETC systems.

The physical manifestation of a PETC scheme in an NCS can be interpreted as plants transmitting their states over the network at a periodic rate (dictated by the sampling interval, h), and the controllers updating their input to the plants based on the triggering condition represented by Eq. (2-13). Thus, *a PETC scheme is suitable for demonstration in a practical implementation of an NCS.*

2-2 Traffic Models

We formally define some basic concepts first. These definitions are used in the forthcoming sections to model control systems.

Definition 2-2.1 (System [21]). *A system is a sextuple* $(X, X_0, U, \rightarrow, Y, H)$ *consisting of*

- *a set of states* X*;*
- *a set of initial states* $X_0 \subseteq X$*;*
- *a set of inputs* U*;*
- *a transition relation* $\longrightarrow \subseteq X \times U \times X$*;*
- *a set of outputs* Y*;*
- *an output map* $H : X \longrightarrow Y$*.*

Definition 2-2.2 (Power Quotient System [22]). *Let* $\mathcal{S} = (X, X_0, U, \rightarrow, Y, H)$ *be a system and* R *be an equivalence relation on* X*. The power quotient of* S *by* R*, denoted by* $\mathcal{S}_{/R}$*, is the system* $(X_{/R}, X_{/R0}, U_{/R}, \underset{/R}{\longrightarrow}, Y_{/R}, H_{/R})$ *where*

- $X_{/R} = X/R$*;*
- $X_{/R0} = \{x_{/R} \in X_{/R} \,|\, X_0 \cap x_{/R} \neq \emptyset\}$*;*
- $U_{/R} = U$*;*
- $(x_{/R}, u, x'_{/R}) \in \underset{/R}{\longrightarrow}$ *if* $\exists (x, u, x') \in \rightarrow$ *in* $\mathcal{S}$ *with* $x \in x_{/R}$*,* $x' \in x'_{/R}$*;*
- $Y_{/R} \subset 2^Y$*;*
- $H_{/R}(x_{/R}) = \bigcup_{x \in x_{/R}} H(x)$*.*

The interested reader may consult [22] for more mathematical details on the relationship between a system and its power quotient systems.

2-2-1 State Space Partitioning

We have seen that an ETC loop is mainly characterized by the triggering time ($\tau(x)$) for the next event based on the current state (x) of the plant. This can be represented using a system [16] from Definition 2-2.1:

$$\mathcal{S} = (X, X_0, U, \rightarrow, Y, H) \tag{2-14}$$

where

- $X = \mathbb{R}^n$;
- $X_0 \subseteq \mathbb{R}^n$;
- $U = \emptyset$, since there are no external inputs to the system;
- $\rightarrow \in X \times U \times X$ such that $\forall\, x, x' \in X : (x, x') \in \rightarrow$ iff $\xi_X(\tau(x)) = x'$;
- $Y \subset \mathbb{R}^+$;
- $H : \mathbb{R}^n \rightarrow R^+$ where $H(x) = \tau(x)$.

Given that there are infinitely many states in the system $\mathcal{S}$, it is not possible to explicitly define the inter-event time for each state. Instead, we follow the formulation proposed in [16] - to construct a power quotient system $S_{/R}$ (cf. Definition 2-2.2) of the original system such that *all possible sequences of inter-event times* are captured:

$$\mathcal{S}_{/R} = (X_{/R}, X_{/R0}, U_{/R}, \underset{/R}{\longrightarrow}, Y_{/R}, H_{/R}) \tag{2-15}$$

where

- $X_{/R} = \mathbb{R}^n_{/R} := \{R_1, \ldots, R_q\}$;
- $X_{/R0} = \{R_i \,|\, X_0 \cap R_i \neq \emptyset\}$;
- $U_{/R} = \emptyset$, since there are no external inputs to the system;
- $(x_{/R}, x'_{/R}) \in \underset{/R}{\longrightarrow}$ if $\exists x \in x_{/R}, \exists x' \in x'_{/R}$ such that $\xi_X(H(x)) = x'$;
- $Y_{/R} \subset 2^Y \subset \mathbb{IR}^+$;
- $H_{/R}(x_{/R}) = \left[\inf_{x \in x_{/R}} H(x), \sup_{x \in x_{/R}} H(x)\right] := \left[\underline{\tau}_{x_{/R}} \overline{\tau}_{x_{/R}}\right]$, where the bounds of the set indicate lower and upper bounds on inter-event times for a given state.

Conceptually, the system $\mathcal{S}_{/R}$ approximates the original system $\mathcal{S}$ by dividing the state-space into q regions and deriving relations between them, in terms of transitions that could lead from one region to another; each state of the system is necessarily a member of one region. Due to approximations (i.e. having a finite number of regions to represent the entire state space), each region has its own *range* of inter-event times rather than an exact value, which is the case for the exact system $\mathcal{S}$.

Now we briefly cover a couple of approaches to divide the state-space into a number of regions and construct power quotient systems with their equivalence relations.

Isotropic Partitioning

In [4], the authors propose a conic covering of the state space using generalized spherical coordinates of the state in $\mathbb{R}^n : (r, \theta_1, \ldots, \theta_{n-1})$, where n is the dimension of the state space.

There are $n-1$ angular coordinates based on this covering, and each of these coordinates is divided into m equal sectors (i.e. there are m^{n-1} conic regions). An example of this process for a two-dimensional state space is shown in Figure 2-2.

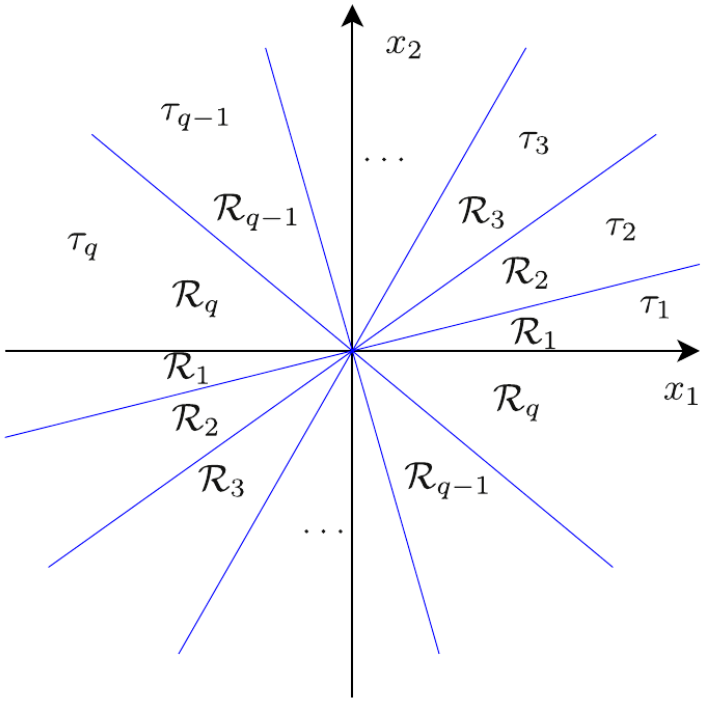

Figure 2-2: Partitioning of a 2-D state space [4]

Before proceeding to describe how this concept can be used to make a power quotient system, an important proposition needs to be introduced.

Proposition 2-2.1 (from [4])**.** *States lying on the same ray crossing the origin have the same inter-sample time, i.e.,* $\tau(x) = \tau(\lambda x)$, $\forall \lambda \neq 0$, $x \neq 0$.

Based on this idea, a union of an infinite numbers of these rays is represented using convex polyhedral cones pointed at the origin, and a finite number of these cones together can

constitute the entire state space; the limits on the angular coordinates of each cone being determined by the number of cones, using the concept of isotropic covering. Furthermore, the cones can be represented using a set of matrices [22]:

$$\begin{aligned} \mathcal{R}_s &= \{x \in \mathbb{R}^2 \,|\, x^T \mathcal{Q}_s x \geq 0\}, \qquad if\ n = 2 \\ \mathcal{R}_s &= \{x \in \mathbb{R}^n \,|\, E_s x \geq 0\}, \qquad if\ n \geq 3 \end{aligned} \tag{2-16}$$

for $s \in \{1, \ldots, q\}$ and appropriately designed matrices $\mathcal{Q}_s = \mathcal{Q}_s^T \in \mathcal{M}_2(\mathbb{R})$ or $E_s \in \mathcal{M}_{n \times p}(\mathbb{R})$ with $p \leq 2n - 2$.

Thus, through this process we get a set of regions $\{R_1, \ldots, R_q\}$ such that every state of the system necessarily belongs to one region. More details and mathematical proofs can be found in [4, 22].

Time-Based Partitioning

In [23], the authors introduce a *time-based* partitioning approach of the state space for PETC systems, making use of the fact that triggering takes place only at certain instants of time (i.e. kh, as stated in Section 2-1-1) without any timing uncertainty.

Using the standard formulation of a linear PETC system with quadratic event-triggering conditions [3], the set of states that will certainly have triggered by time k is given by:

$$\mathcal{K}_k = \begin{cases} \{x \in \mathbb{R}^n \,|\, x^T N(k) x > 0\}, & k < \overline{k}, \\ \mathbb{R}^n, & k = \overline{k} \end{cases} \tag{2-17}$$

where

$$N(k) = \begin{bmatrix} M(k) \\ I \end{bmatrix}^T Q \begin{bmatrix} M(k) \\ I \end{bmatrix}, \quad M(k) = e^{Akh} + \int_0^{kh} e^{A\tau} d\tau BK \tag{2-18}$$

I denotes the identity matrix, and Q (a symmetric matrix) is the *triggering matrix*. The reader is referred to [3] for further reading; in short, $N(k)$ represents a triggering condition corresponding to k time steps, and it can be checked for any state except the origin.

Now, the minimum value of k that causes triggering can be calculated by removing the states that would have triggered before:

$$\mathcal{R}_k = \begin{cases} \mathcal{K}_k \backslash \bigcup_{j=1}^{k-1} \mathcal{R}_j, & k > 1, \\ \mathcal{K}_k, & k = 1 \end{cases} \tag{2-19}$$

Thus, using this recursive relation, the set $\{\mathcal{R}_1, \mathcal{R}_2, \ldots\}$ can be computed, and it also happens to be a partition of the state space $\mathbb{R}^n$.

2-2-2 Inter-Event Time Bounds Construction

We have thus seen how the state space can be partitioned into regions based on certain methods, and how any arbitrary state's membership of a particular region can be computed.

The next step for our approximate system is to construct the output map $H_{/R}(x_{/R})$ which provides a set of bounds on the inter-event times for each region.

- In the case of isotropic partitioning, based on Eq. (2-11) for determining state evolution, a set of LMIs can be formulated [22]. Solving those, a set of bounds $[\underline{\tau}, \overline{\tau}]$ can be computed for every region of the partitioned state space.
- Likewise, in the case of a PETC system with isotropic partitioning, the discrete analogue of the regional bounds $[\underline{k}, \overline{k}]$ can be computed [24].
- However, in the case of time-based partitioning, the regions are determined by when triggering needs to necessarily take place for a state, and hence, inherently, the inter-event times are exact, i.e. the lower and upper bounds are the same. By construction, $H(x) = k, \forall x \in \mathcal{R}_k$ (cf. Eq. (2-19)).

For brevity's sake, mathematical details are not covered here, and the interested reader is referred to [22] for bounds calculation in the isotropic partitioning case ([24] for the PETC case) and [23] for time-based partitioning. Relevant proofs can be found in the same references.

2-2-3 Transition Relations

Once we have the bounds on inter-event times and the set of regions into which the state space is divided, we need to compute the transition relations between the regions, i.e., how triggering in a given region could lead to the state evolving as per its dynamics into another region; on the abstraction, this may lead to nondeterminism, since the regional partitions yield just an approximation of the original system.

Definition 2-2-3 (Flow pipe [25])**.** *The set of reachable states, or flow pipe, from X_0 in the time interval $[t_1, t_2]$ is defined as:*

$$\mathcal{R}_{[t_1,t_2]}(X_0) = \bigcup_{t \in [t_1,t_2]} \mathcal{R}_t(X_0).$$

Accordingly, in the case of state space partitioning with inter-event timing bounds, the flow pipe that needs to be computed is (from [16]):

$$\mathcal{X}_{[\underline{\tau}_s,\overline{\tau}_s]}(X_{0,s}) = \bigcup_{t \in [\underline{\tau}_s,\overline{\tau}_s]} \{\xi_{x_0}(t) \,|\, x_0 \in X_{0,s}\}.$$

Computing the set of reachable states from any region thus yields transition relations between regions for the isotropic partitioning case. This can be done first by deriving a polytopic outer

approximation to the flow pipe by dividing the time interval $[\underline{\tau}_s, \overline{\tau}_s]$ into $\bar{l}$ subintervals, so as to get

$$\hat{\mathcal{X}}_{[\underline{\tau}_s,\overline{\tau}_s]}(X_{0,s}) = \bigcup_{l} \hat{\mathcal{X}}_{[t_l,t_{l+1}]}(X_{0,s}) \quad \forall f \in \{1,\ldots,\bar{l}\}$$

with $t_1 = \underline{\tau}_s$ and $t_{l+1} = \overline{\tau}_s$. Following this, by solving feasibility problems for each pair of conic regions by computing the outer approximation of each flow pipe segment from an *initial conic region*, the transitions from that region can be derived [22].

When time-based partitioning is involved, such transition relations can be computed by solving a non-convex quadratic constraint satisfaction problem [23]. Given two regions $\mathcal{R}_i$ and $\mathcal{R}_j$, and the state $x \in \mathcal{R}_i$, we have the following problem formulation:

$$\begin{aligned} \exists \quad & x \in \mathbb{R}^{n_x} \\ s.t. \quad & x^T N(i) x > 0, \\ & x^T N(i') x \leq 0, \forall i' \in \{1,\ldots,i-1\}, \\ & x^T M(j)^T N(i) M(j) x > 0, \\ & x^T M(j)^T N(i') M(j) x \leq 0, \forall i' \in \{1,\ldots,i-1\}. \end{aligned} \tag{2-20}$$

The satisfaction of these conditions implies that there exists a transition relation between the two regions. Furthermore, [23] further proposes a semi-definite relaxation on the problem to fit the semi-definite programming formulation, and the interested reader is referred to the paper for details.

2-3 Timed Automata for Constructing Abstractions

Some concepts from automata theory are employed in the modelling of control loops and the communication network, and those need to be introduced for a better understanding of this thesis's objectives.

2-3-1 An Introduction to Timed Automata

The notion of a *finite automaton (state machine)* needs to be introduced first - it is essentially a directed graph consisting of finitely many nodes (states) connected with finitely many labelled edges. An example is used to illustrate the concept in Figure 2-3 - a finite automaton representing a turnstile whose behaviour is determined by the user's action of inserting a coin and pushing a button.

On extending a finite automaton with certain real-valued variables (referred to as *clocks*), we get a *timed automaton*, first introduced in [14]. As the name suggests, these *clock* variables are used to model logical clocks, and are initialized to zero at the start of the system, increasing synchronously at the same rate thereafter. Timed automata are used for capturing a system's behaviour taking into consideration the passage of time. Figure 2-4 shows a simple example [6] involving a model in which a lamp changes its brightness depending on the timing of a user's

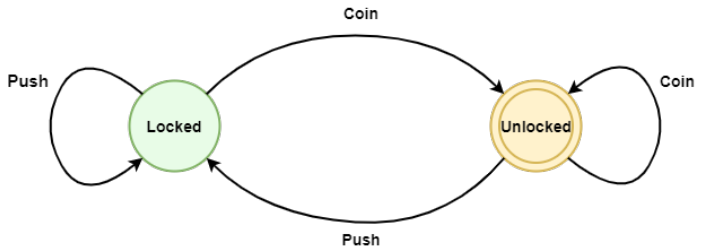

Figure 2-3: A simple state machine representing the behaviour of a *turnstile* - governed by the actions of *inserting a coin* and *pushing* to transition between the *Locked* and *Unlocked* states. An initial state needs to be specified in a finite automaton; in this example, it is the *Unlocked* state [5]

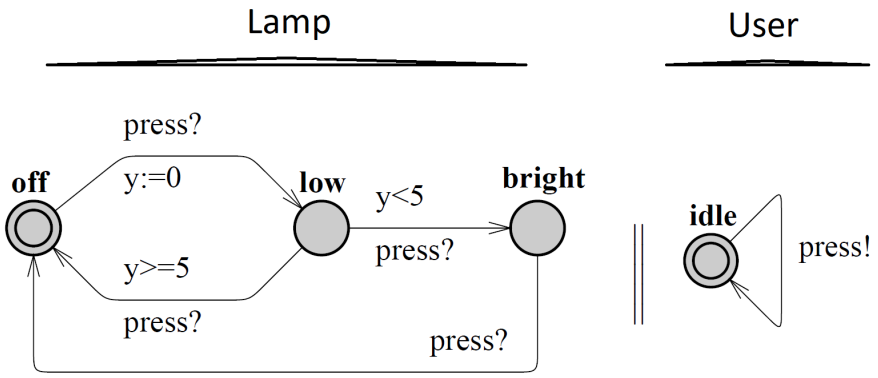

Figure 2-4: A timed automaton modelling a lamp (left) and a user (right). Based on the user's *press* action, the lamp responds synchronously depending on its timing; y represents the clock variable [6]

button press actions - when the lamp is off, if the user presses the button, the lamp turns on and glows dimly. If the user presses the button again within a short time interval, the lamp glows brightly, but if the interval between the two press actions is long, the lamp turns off. Thus, the behaviour of the lamp differs because of timing even though the physical actions performed are the same.

Now, before formally defining a timed automaton, we need to introduce the notion of clock constraints. Defining $\mathcal{C}$ as the set of finitely many clocks, a *clock constraint* is a conjunctive formula of atomic constraints, formed as $x \bowtie n$ or $x - y \bowtie n$ where $x, y \in \mathcal{C}$ and $\bowtie \in \{\leq, < , =, >, \geq\}$ and $n \in \mathbb{N}_0$. Clock constraints are denoted by $\mathcal{B}(\mathcal{C})$.

Definition 2-3.1 (Timed (Safety) Automaton [2])**.** *A Timed (Safety) Automaton is a sextuple* (L, l_0, Act, C, E, Inv) *where*

- L *is a set of finitely many locations (also called nodes);*
- $l_0 \in L$ *is an initial location;*
- Act *is a set of finitely many actions;*
- C *is a set of finitely many real-valued clocks;*

- $E \subseteq L \times \mathcal{B}(C) \times Act \times 2^C \times L$ *is a set of edges;*
- $Inv : L \to \mathcal{B}(C)$ *assigns invariants to locations.*

Location invariants here are restricted to downwards-closed constraints of the form: $c \leq n$ *or* $c < n$ *where* c *is a clock and* $n \in \mathbb{N}_0$.

Edge transitions may also be represented by $l \xrightarrow{g,a,r} l'$ for $(l, g, a, r, l') \in E$ where $l, l' \in \mathtt{L}$, g represents the guards over clocks enabling the transition, $a \in Act$, and $r \subseteq C$ is the set of clocks to be reset on the transition. We denote transitions with arbitrary labels as $l \to l'$.

Remark 2-3.1 (from [2])**.** *When first introduced in [14], Büchi and Muller accepting conditions were used to enforce progress properties in timed automata. However, in this thesis, we focus on a simplified version called Timed Safety Automata (TSA) [26], which uses local invariant conditions to specify progress properties. Along the lines of [2], we refer to TSA as simply TA.*

Based on Definition 2-3.1, the following notions can be used to understand behaviour modelling using timed automata:

- Since timed automata capture the dimension of time (through logical clocks, in addition to the *states* of finite automata), they are used in modelling real-time systems, which involve monitoring the elapsing of time [14].
- *States* and *Locations* hold different meanings in the context of timed automata - a state is defined by the current node a timed automaton is in, and the current clock values; nodes (called *states* in finite automata) are referred to as locations in TA.
- A timed automaton's timing behaviour is restricted by means of clock constraints.
- A timed automaton's edge may be subject to *guard* conditions on the values of clocks that determine whether that edge can be taken or not (i.e. if the conditions are satisfied, the edge is said to be *enabled*); an edge transition can be taken only if the edge is enabled.
- Clocks can be individually reset to zero when an edge is taken.

In our TA models, clock constraints are specified in the following manner:

- While in a location, clocks may increase as long as the clock constraints on that location hold, violating which it must be left via an edge. These constraints are referred to as *invariants*.
- Only *enabled* edge transitions can be taken; constraints on edges determining whether they are enabled or not are called *guards*.

The semantics of a TA are defined as a transition system where a state has the *current location* and *current value of clocks*. In this system, two types of transitions are possible between states:

1. a delay transition, wherein the automaton simply delays for some time, and,
2. a discrete transition, wherein the automaton takes an enabled edge.

To formalize this, we need some more notations. Clock assignments are functions that map clocks to their current values, i.e. $u : C \to \mathbb{R}_{\geq 0}$. The notation $u \vDash g$ indicates that the clock values of u satisfy the guard g. In order to formally define *operational semantics*, we use these notations as follows:

- Given $d \in \mathbb{R}_{\geq 0}$, $u + d$ denotes the clock assignment mapping all $c \in C$ to $u(c) + d$,
- Given a set of clocks $c \subseteq C$, $u[c]$ denotes the clock assignment mapping all clocks in c to 0 and agreeing with u for the rest of the clocks, i.e. $C \setminus$ c.

Definition 2-3.2 (Operational Semantics [2])**.** *The semantics of a timed automaton is a (timed) transition system wherein states are pairs of location (l) and clock assignment (u), and transitions are defined by the rules:*

- *Delay transition:* $(l, u) \xrightarrow{d} (l, u + d)$ *if* $u \vDash \mathit{Inv}(l)$ *and* $(u + d) \vDash \mathit{Inv}(l)$ *for a non-negative real number* $d \in \mathbb{R}_{\geq 0}$
- *Discrete transition:* $(l, u) \xrightarrow{a} (l', u')$ *if* $l \xrightarrow{g,a,r} l'$, $u \vDash g$, $u' = u[r]$, *and* $u' \vDash \mathit{Inv}(l')$

Additionally, a run of a timed automaton is defined as a sequence of alternating delay and discrete transitions in the transition system.

An action (denoted by `Act` in Definition 2-3.1) could be a *synchronizing action* (a pair of input (**act?**) and output (**act!**) actions) or an internal action ("tau" or $*$). Sychronizing actions are used as a means of handshake synchronization between two or more TA, and internal actions are used for transitions within the same TA. Each transition (edge) may have only one action associated with it.

2-3-2 Parallel Composition

Several TA representing individual (sub-)systems may be composed in a parallel fashion to model a system involving all those subsystems running concurrently; in such a scenario, synchronizing actions are used for modelling synchronous communication between two or more subsystems. In the example from Figure 2-4, the lamp and the user are two TA, and their parallel composition (represented using the "||" symbol) is also a TA.

Such a TA, involving a parallel composition of multiple TA, is called a *network of timed automata (NTA)*. This concept is particularly important for this thesis because models in the forthcoming sections involve parallel compositions of TA representing various parts of the NCS. Interested readers are referred to [27] for a more in-depth understanding of this topic.

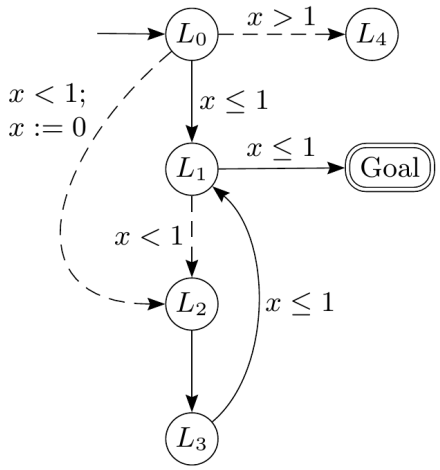

Figure 2-5: A timed game automaton - dashed arrows represent uncontrollable edges and solid arrows represent controllable edges [7]

2-3-3 Timed Game Automata

In a physical system, a controller may be able to precisely control some variables (say, flow rate) but not others (for instance, a temperature range may be known but a desired temperature may not be precisely attainable) - and this uncertainty can be taken into consideration while modelling TA.

A TA can be further extended by partitioning its set of actions into sets of *controllable* and *uncontrollable* actions, such that the controller (player) can trigger only the former, and the environment (opponent) can trigger only the latter. Figure 2-5 shows an example of a TGA.

Definition 2-3.3 (Timed Game Automaton [2])**.** *A Timed Game Automaton is a septuple* $(L, l_0, Act_c, Act_u, C, E, Inv)$ *where*

- $(L, l_0, Act_c \cup Act_u, C, E, Inv)$ *is a timed automaton;*
- Act_c *is the set of controllable actions;*
- Act_u *is the set of uncontrollable actions;*
- $Act_c \cap Act_u = \emptyset$*.*

As with TA, a parallel composition of TGA is also possible, referred to as a Network of Timed Game Automata (NTGA).

Definition 2-3.4 (Network of Timed Game Automata [2])**.** *Let* $TGA^i = (L^i, l_0^i, Act_c^i, Act_u^i, C^i, E^i, Inv^i)$ *be a TGA for* $i \in \{1, \ldots, n\}$*. The parallel composition of* $TGA_1, \ldots, TGA_n$*, denoted by* $TGA_1 | \ldots | TGA_n$ *is a timed game automaton* $TGA = (L, l_0, Act_c, Act_u, C, E, Inv)$ *where*

- $L = L^1 \times \ldots \times L^n$*;*

- $l_0 = (l_0^1, \ldots, l_0^n)$;
- $Act_c = \{*\} \cup \bigcup_{i=1}^{n} \{a \in Act_c^i \mid a \text{ is an internal action}\}$;
- $Act_u = \{\circledast\} \cup \bigcup_{i=1}^{n} \{a \in Act_u^i \mid a \text{ is an internal action}\}$;
- $C = C^1 \cup \ldots \cup C^n$;
- *E is defined as per the following couple of rules:*
  - *a TA makes a move on its own via its internal action: an edge is controllable iff the internal action is controllable;*
  - *two TAs move simultaneously via a synchronizing action: an edge is controllable iff both input and output actions are controllable (that is to say, the environment has priority over the controller);*
- $Inv((l_1, \ldots, l_n)) = Inv^1(l_1) \wedge \ldots \wedge Inv^n(l_n)$.

This timed game automaton, composed of a parallel composition of multiple timed game automata, is called a Network of Timed Game Automata (NTGA).

A point to note here is that in a parallel composition of TGA, a pair of input and output actions is denoted by a single synchronizing action (i.e. semantically speaking, since the input and output actions should occur simultaneously, it is as though a single action is performed). In Definition 2-3.4, $*$ refers to controllable internal actions, and $\circledast$ to uncontrollable internal actions.

2-3-4 Runs and Strategies

Following Definition 2-3.2, we define `Runs(TA)` as the set of runs of a TA starting from its initial state (l_0, u_0), l_0 being the set of initial locations and u_0 being a clock assignment mapping all clocks to 0. $last(\rho)$ is the last state of a run ρ, if the run is finite.

In general, a strategy is a function that defines what action a controller should perform to win (or avoid losing) a game. Let us denote a *delay* action by λ (which semantically means that the controller should just wait) and use it to formally define a strategy in the context of TGA [2].

Definition 2-3.5 (Strategy [2])**.** *Let $TGA = (L, l_0, Act_c, Act_u, C, E, Inv)$ be a timed game automaton. We define $TA = (L, l_0, Act_c \cup Act_u, C, E, Inv)$ as its derived timed automaton. A strategy f over TGA is a partial function from $Runs(TA)$ to $Act_c \cup \{\lambda\}$ such that for every finite run ρ, if $f(\rho) \in Act_c$ then last $(\rho) \xrightarrow{f(\rho)} (l', u')$ for some (l', u').*

In this thesis, having a strategy that *avoids losing* (i.e., avoids having runs ending in a certain set of states) is particularly important, as will be presented in the forthcoming sections.

2-4 Controller Area Network (CAN) Bus

The Controller Area Network (CAN) standard is a *vehicle bus* standard (i.e. for a communication network used in connecting vehicular components like ECUs) developed by Bosch in the 1980s. With increased usage, it was later standardized internationally as ISO 11898 [8].

There are several electronic devices involved in automotive applications, and having dedicated signal lines between each device is expensive and difficult to maintain. CAN was developed as a means to reduce the complexity involved in such applications by connecting various devices over a common serial bus (CAN bus). The connection is such that any device on the CAN network can communicate with any other device on the same network.

2-4-1 Relevant Terminology

There are several technical terms associated with a CAN bus, but only those relevant to the scope of this thesis are covered here.

- **CAN frame (message)**: A chunk of data in a predefined format (defined in the Bosch specification[1]) transmitted by a CAN device is called a CAN frame, or a *message*. Depending on the purpose, there are multiple types of frames, but in this thesis we consider only data frames.
- **CAN identifier**: Each device on the common bus has a unique ID (also called the *arbitration* ID). When a device transmits a message, all devices on the bus receive it - the CAN ID precedes the data and is used by the receiving device to determine if the received data is relevant to it (if not, it is ignored).
- **Message priority**: If two or more devices transmit messages simultaneously, the CAN identifier is used for deciding the priority - lower numerical value implies higher priority, and the device with the highest priority gets to transmit its message on the network.
- **Collision**: A situation in which two or more devices attempt to send a message. simultaneously is called a *collision*
- **Data bytes**: In CAN data frames, the data may be 0 to 8 bytes in length.

2-4-2 CAN Operation

CAN devices on a common network are loaded with a CAN *database* which consists of information about signals and arbitration IDs. This is used decide if certain frames received by a particular device are relevant to it, or are to be ignored (for instance, a vehicle's RPM values bear no meaning for its electronic mirror controller). Since a CAN bus is common across the network, any outgoing message from a device is received by all devices on it; the receiving device decides if the message is relevant or not [8].

If two or more devices attempt to use the bus to transmit data simultaneously, it leads to a collision. In such a case, the device arbitration IDs are used to resolve the collision; the

[1] Robert Bosch GmbH, "CAN Specification 2.0", 1991.

device with the highest priority continues transmitting, while the lower priority devices stop. Figure 2-6 shows an example for two devices.

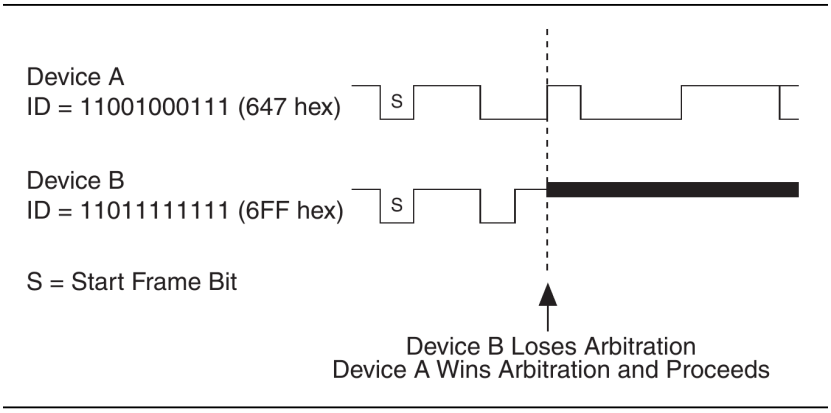

Figure 2-6: An example of arbitration in two CAN devices [8]

Additionally, the CAN protocol provides error detection and confinement mechanisms. A receiving device transmits an *error flag* if it detects an error in the received frame, and the transmitting device can accordingly retransmit the same frame on the network to correct the error. Based on the number of errors generated, certain devices may be labelled as malfunctioning and accordingly they may be made *passive* or completely disabled depending on the error counter value. This *error confinement* mechanism is to handle malfunctioning devices so as to not cause a disturbance in network traffic.

Chapter 3

Modelling through Abstractions and Scheduling Strategies

The fundamentals of event-triggered control and timed automata theory have been introduced in Chapter 2; now we link those concepts to make *abstractions* of control loops and the network.

In a nutshell, each component of the system (comprising the control loops and the connecting network) is modelled as a timed game automaton, and an NTGA of these components is an abstract representation of the overall system - it is thus used for appropriate scheduling of actions as required by the physical system.

3-1 Abstraction of Control Systems

The outcome of partitioning the state space of a system into regions based on inter-event times, and constructing transition relations between them based on a reachability analysis, is semantically equivalent to a TA - with locations corresponding to regions, and edges corresponding to their transition relations.

Based on this notion, considering a control system given by Eq. (2-1), partitioning its state space and constructing its power quotient system (cf. Eq. (2-15)), we can represent its control loop with a TA along the lines of [22] and [2].

$TA_{CL} = (L, l_0, Act, C, E, Inv)$ where:

- $L = X_{/R} = \{R_1, \ldots, R_q\}$,
- $l_0 = R_s$ such that $\xi(0) \in R_s$,
- $Act = \{*\}$, a set of internal actions for labelling edge transitions,
- $C = \{c\}$, a set containing a single clock,

- $(R_s, \underline{\tau}_s \leq c \leq \overline{\tau}_s, *, \{c\}, R_t) \in E$ such that the flow pipe, $\mathcal{X}_{[\underline{\tau}_s, \overline{\tau}_s]}(R_s) \cap R_t \neq \emptyset$, and,
- $Inv(R_s) = \{0 \leq c \leq \overline{\tau}_s\}, \forall s \in \{1, \ldots, q\}$.

This is a simple representation to link the two concepts of ETC and TA - the main idea is that all possible states of a control system can be *abstracted* into a finite number of locations; each location R_s having its inter-event time bounds $\underline{\tau}_s$ and $\overline{\tau}_s$. The clock variable is used to keep track of time elapsed since the last transition to a location, and thus to check if the corresponding inter-event time constraints are satisfied - based on which the next transition occurs. Invariants on locations guarantee that they are left as soon as the clock value exceeds the upper bound. The clock is reset on a transition to a new location to track elapsed time specific to that location, and so on and so forth. Figure 3-1 shows an example of a control loop's TA model.

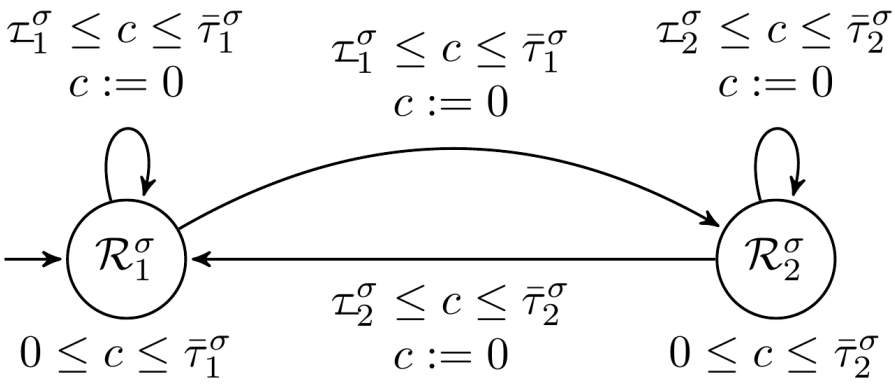

Figure 3-1: A TA model of a ETC loop with two state space regions [2]

Triggering in this context means that the control input to the plant is updated based on the state the plant is currently in; in terms of TA, when the plant is in region $\mathtt{R_s}$ (corresponding to its state $\xi(t) = x$) and the clock value is between $\underline{\tau}_s$ and $\overline{\tau}_s$, the input is updated. Subsequently, the region the plant enters ($\mathtt{R_t}$) is determined by the state $\xi(t + \tau(x))$, where $\tau(x) \in [\underline{\tau}_s, \overline{\tau}_s]$ is the time at which triggering occurs. A flow pipe for $\mathtt{R_s}$ is accordingly constructed to determine the set of all possible *target* regions [2].

3-2 Modelling Using ETC Abstractions

In [28], some examples of using timed automata for scheduling tasks are provided. Following this concept, we make use of TGA models for scheduling control tasks.

In this thesis, we consider a NCS in which control loops are distributed over a common communication channel. Each control loop involves a plant with a sensor, a feedback controller (cf. Eq. (2-2)), and an actuator, distributed over the network, which is physically manifested in the form of a CAN network connecting the loops, as will be explained in the next chapter.

Following [2], the network is such that it can be used by at most one control loop at a time, otherwise a conflict arises - in physical terms, *network occupancy* translates the period of time used for data transmission between a control loop's controller and actuator over the network.

Given these constraints, we can see that the goal translates to scheduling of triggering actions in an appropriate manner to avoid network conflicts. Accordingly, we synthesize a scheduler for this problem with certain objectives in mind.

In this context, *scheduling* from the scheduler's perspective refers to actions it can perform - it can (i) force a control loop to trigger by transmitting its control action data before the control loop enters a state ready for triggering (i.e., the control loop is in a region for less than $\underline{\tau}_s$ units of time, $\underline{\tau}_s$ being the lower IET bound for that region), or (ii) simply *wait* for a control loop to enter a state in which it triggers automatically (i.e., wherein the control loop is in a region for an amount of time within the IET bounds for that region).

Based on this idea of scheduling, the scheduler's objectives can be stated more specifically:

- to force the control loops to trigger early or let them trigger automatically, in such a fashion that only one control loop occupies the network at a given time,
- in line with the thesis's objectives to reduce conservatism, to allow each control loop a finite number of *retransmission* attempts on using the network - that is to say, if a control loop has to trigger, it needs to request access to the network; if it is already in use, the control loop has to wait to trigger till it becomes available again, and,
- to prevent the number of retransmission attempts for any loop from exceeding a predefined, loop-specific upper bound - this is to ensure that the reduced conservatism does not lead to instability of the control loops.

These objectives can be achieved by:

1. modelling TGA abstractions of the control loops and the networks,
2. composing them in parallel to yield an NTGA,
3. defining objectives for this NTGA that translate semantically to the aforementioned scheduling objectives. Though such objectives can generally be of multiple types, based on our desired goals, we define a safety objective:
    - Safety objective: As a minimum objective, none of the control loops should enter a *Bad* location as a result of exceeding their limit on retransmission attempts while in the same region, i.e., triggering must necessarily occur within a certain number of attempts,
4. generating *strategies* to be followed for achieving the NTGA's objectives.

Based on [2] with some modifications suited for use on a CAN network, the generated strategies can use the following *actions* to be taken by the scheduler (the game player in a TGA context):

- while the NTGA is currently in a location corresponding to a region, not doing anything and letting the control loop trigger by itself,

- forcing an *early trigger* (i.e., forcing a control loop to trigger before the clock value crosses the lower IET bound for that region),
- sending an acknowledgement (ACK) synchronizing action from the network to a control loop if the latter requests the former for use while free, and,
- sending a *negative acknowledgement* (NACK) to a control loop if the network is occupied when the control loop makes a request to it for occupying it.

In this context, controllable edges represent actions that are deterministic from the scheduler's perspective (i.e. the scheduler can control precisely when to take such edges) and uncontrollable edges represent actions that are taken based on *environmental* conditions that enable them (such that the scheduler cannot precisely control them).

Considering this information, we proceed with the formal modelling of TGA for the network and the control loops conforming to our requirements.

Legend: In the following models, the convention used is as follows:

- - - ->: Dashed arrows represent uncontrollable edges;
- ——>: Solid arrows represent controllable edges;
- guards: Text in red is for edge guards;
- resets: Text in blue is for "resets" (clock and other variable assignments on taking the corresponding edge transitions);
- invariants: Text in magenta is for invariants on locations.

We also define *urgent locations* as these serve an important purpose in the TA models.

Definition 3-2.1 (Urgent locations [6])**.** *Urgent locations are locations in which time cannot elapse, i.e. such locations have to be left via an outgoing edge transition as soon as they are entered. Semantically speaking, there is an additional clock variable* `x` *which gets reset on every incoming edge with an implicit invariant of* $\texttt{x} \leq \texttt{0}$.

In the models that follow, urgent locations are coloured orange, absorbing (`Bad`) locations are red, and usual locations are blue.

3-2-1 Network TGA Model

The network's TGA model is modelled such that the network can be used by only one control loop at a time, and other control loops need to wait until it becomes *idle*. This reflects network protocols in which one common bus connects all devices, such as in the CAN protocol.

Note: The `ack` *and* `nack` *actions introduced henceforth do not refer to protocol-specific signals of the same names; the actions have been named such owing to similar roles of acknowledgement in the TGA models.*

When a control loop requests access (via an `up?` synchronizing action), the network performs an `ack!` synchronizing action to indicate that it can indeed be occupied by the *client* control loop, if it is idle. But in case the network is already in use by a control loop (i.e., it is in the *InUse* location) and another control loop requests access, it performs a `nack!` action to indicate to that control loop that it needs to wait and try accessing the network again later. Each of these synchronizing actions is performed via an outgoing edge from urgent locations `ACK_loc` and `NACK_loc` as seen in Figure 3-2. Additionally, the control loop using the network signals that it is done transmitting relevant data via a `down!` synchronizing action, after which the network returns to the `Idle` location.

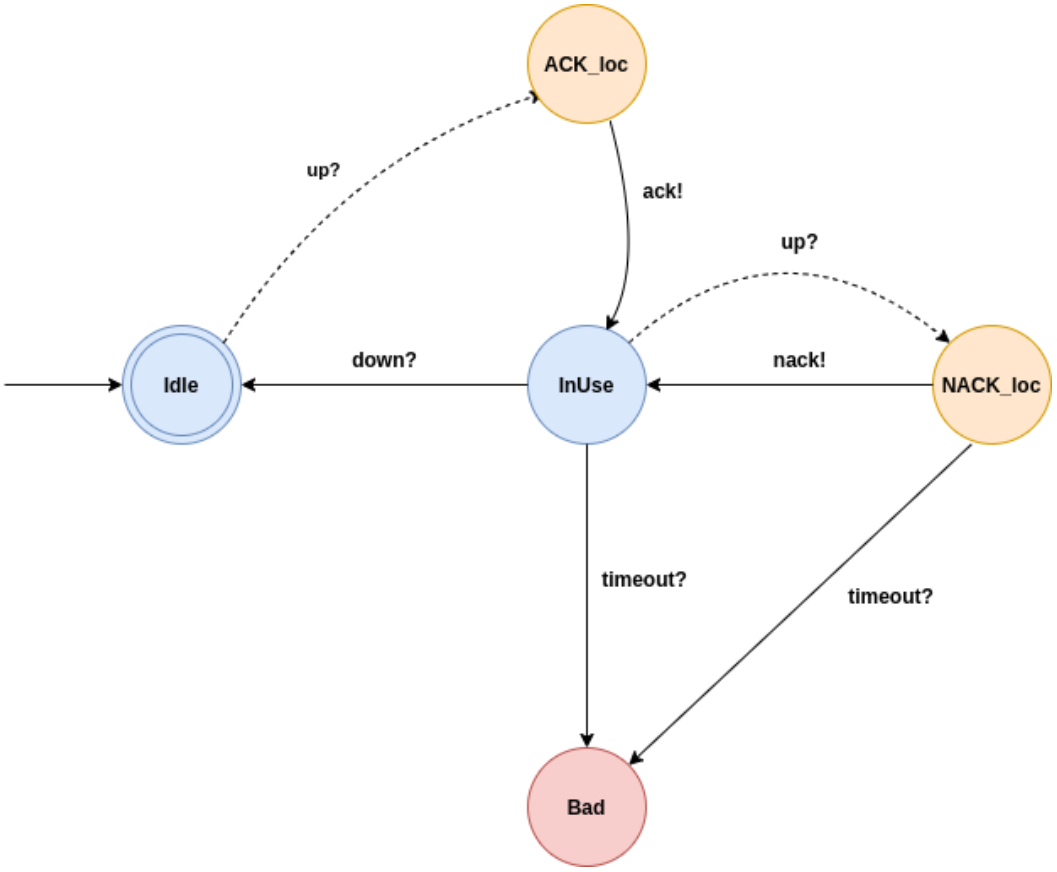

Figure 3-2: TGA model for the communication network

If any of the control loops enters an undesirable location (because of its exceeding its number of retransmission attempts), the network is accordingly informed via a *timeout* synchronizing action by that control loop (this *timeout* action will be introduced in the control loop TGA model) - this leads to the network entering the *Bad* location. This is a scenario to be avoided, and if it occurs, it indicates that the strategy followed by the scheduler led to this *loss* and hence is not to be taken. The *Bad* location is an absorbing location, which means that it cannot be left via any action once entered.

Definition 3-2.2 (Network TGA Model)**.** *The TGA model for the network is represented by* $TGA^{net} = (L^{net}, l_0^{net}, Act_c^{net}, Act_u^{net}, C^{net}, E^{net}, Inv^{net})$ *where:*

- $L^{net} = \{Idle, InUse, ACK\_loc, NACK\_loc, Bad\}$;

- $l_o^{net} = Idle;$
- $Act_c^{net} = \{ack!, nack!, down?, timeout?\};$
- $Act_u^{net} = \{up?\};$
- $C^{net} = \emptyset;$
- $E^{net} =$ *a set comprising the following edges:*
  - $(Idle, true, up?, \emptyset, ACK\_loc)$,
  - $(ACK\_loc, true, ack!, \emptyset, InUse)$,
  - $(InUse, true, up?, \emptyset, NACK\_loc)$,
  - $(NACK\_loc, true, nack!, \emptyset, InUse)$,
  - $(NACK\_loc, true, timeout?, \emptyset, Bad)$,
  - $(InUse, true, timeout?, \emptyset, Bad);$
- $Inv^{net} = \emptyset$ *for all locations, since there are no clocks.*

A point to note is that this network model does not make use of any clocks. Additionally, a guard of `true` on an edge indicates that there are no constraints on taking that edge.

3-2-2 Control Loop TGA Model

In the control loop model, each state space region has an eponymous location (i.e., for region $\mathcal{R}_1$ there is a location R_1 and so on) from which to trigger *naturally*, and an additional location for *early triggering* while in that region (for instance, Ear_1). That is to say, in each control loop:

- triggering can occur *naturally*, i.e. non-deterministically when the clock value is between the lower and upper bounds for a region, or,
- triggering can be forced to be earlier than that, i.e. when the clock value is lower than the calculated lower bound for a region.

This distinction of triggering approaches is to provide a certain sense of flexibility in terms of what the scheduler can control, and hence to avoid network conflicts.

For keeping track of transitions to be taken (based on the control loop's reachability analysis) and the number of *retransmissions* performed by the control loop, we make use of the following *state variables* (all being integer values):

- `to_region`: index number of the target region from a given region (i.e. if the loop is to enter $\mathcal{R}_2$, `to_region = 2`);
- `from_region`: index number of the source region;
- `rt_count`: number of retransmission attempts made by the loop without getting access to the network.

We also define some integer constants:

- `RT_MAX`: the limit on the number of consecutive retransmission attempts by the control loop from the same region;
- $\texttt{N}_{\texttt{max}}$: the maximum number of clock ticks allowed as a relaxation on the regional upper bound to account for retransmissions;
- Δ: The channel occupancy time in terms of clock ticks.

The model is illustrated for a control loop with two regions in Figure 3-3. The flow of logic is explained with reference to this figure, as follows:

- We begin with the assumption that the loop is currently in region $\mathcal{R}_1$ and the variable `from_region` is set to the index number of that region (i.e., `from_region` = 1).
- While in region $\mathcal{R}_1$ the loop can either stay there or move to its corresponding early location Ear_1 depending on the clock value.
- In case the loop does not take an edge to the early location, triggering occurs when the clock value is between the triggering bounds. Network access is requested by means of an `up!` synchronizing action.
- All early locations are urgent - so when an edge is taken to an early location, access to the network needs to be requested immediately for triggering, via an `up!` action.
- On requesting network access, the loop enters an urgent location `Transition_loc` via one of the enabled edges with an `up!` action; as part of this incoming edge's *reset* assignment, the variable `to_region` is set to the index number of the trigger's *target* region. The target region is determined beforehand by reachability analysis using flow pipes originating from the corresponding source regions and timing bounds (i.e. $[\underline{\tau}, \overline{\tau}]$, or $[\underline{d}, \overline{d}]$ in *early* locations).
- While in `Transition_loc`, time cannot elapse; the location is merely an intermediate one to instantly get information about the network's availability. The network TA performs either an `ack!` action to indicate that it is idle and can be used by the loop, or a `nack!` action to indicate that it is already in use and the loop cannot have access yet. The corresponding input actions for the control loop are `ack?` and `nack?` respectively:
  - `nack?` edge: If the network is already in use, the control loop has to wait until it becomes free; the `from_region` variable determines the `nack?` edge that is enabled in this case (i.e. if the loop was in region $\mathcal{R}_2$ or Ear_2 prior to the `up?` transition to `Transition_loc`, only the `nack?` edge with the guard `from_region` == 2 is enabled). Additionally, a guard of `rt_count` < `RT_MAX` ensures that the loop can have at most `RT_MAX` consecutive attempts at retransmitting data via the network. `rt_count` := `rt_count` + 1 on the `nack?` edge's reset assignment is to keep track of the number of consecutive such attempts;

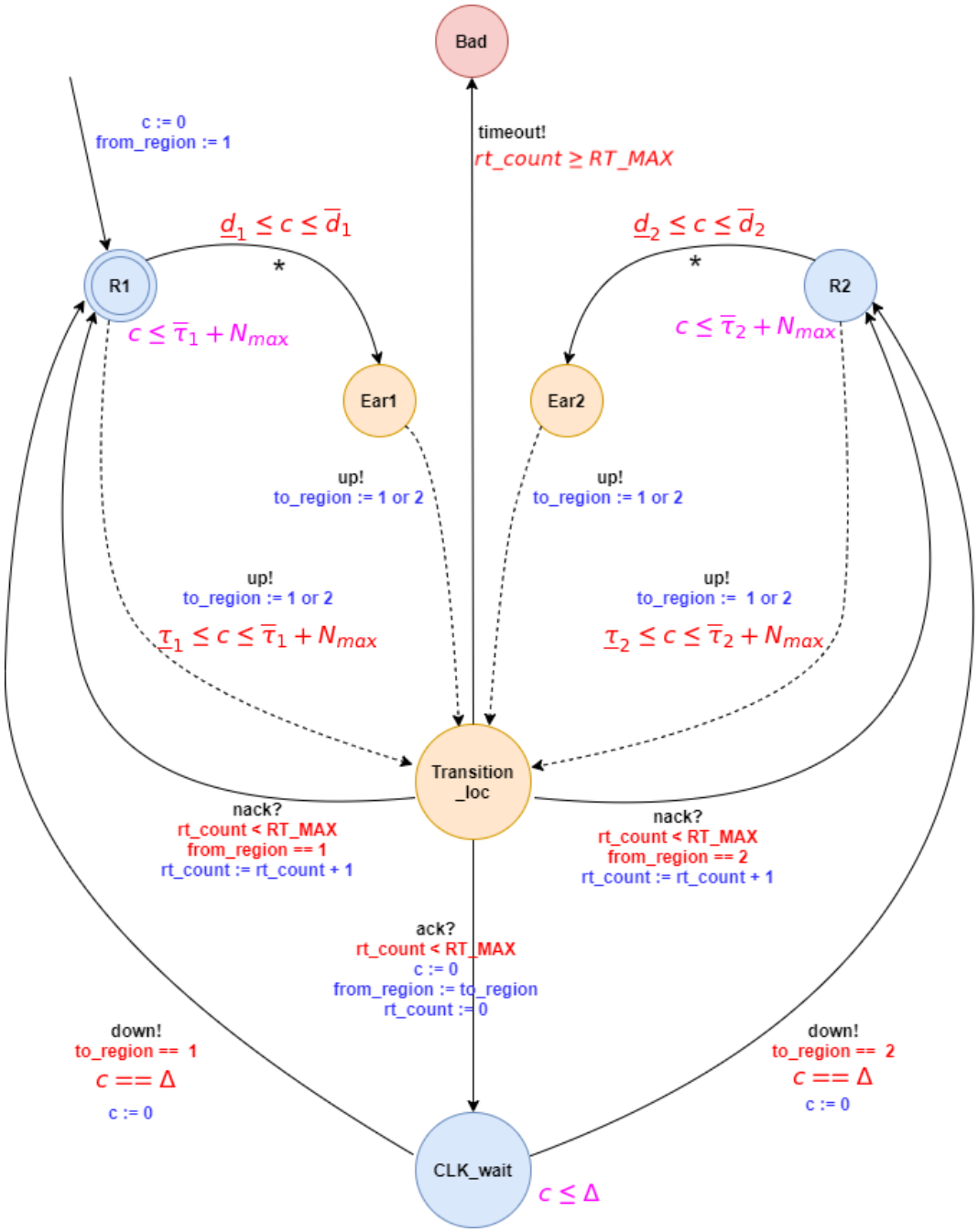

Figure 3-3: TGA model for a control loop with two regions. R1 is assumed to be the initial location here, and uncontrollable edges for `up!` actions are represented by one edge for simplicity of the diagram (but in the actual model, each `to_region` assignment corresponds to its own edge)

– `ack?` edge: Taking this enabled edge implies that the network has provided access to the control loop immediately for triggering, and hence the control loop TA enters a location `CLK_wait`; this being a successful triggering action, the reset assignment has `rt_count` := 0 to indicate that the retranmission attempts were eventually

successful. Additionally, the clock is reset to 0 as it is needed in the `CLK_wait` location for keeping track of time. Finally, to indicate that the *target* region of the triggering transition is the new *source* region for future transitions after successful triggering, `from_region` := `to_region` is also included in the reset assignment on the `ack?` edge.

- The `CLK_wait` location is to indicate that the control loop is currently using the network for transmitting relevant data. We make use of a predefined amount of time (defined in the TA in terms of clock units, Δ) as the *channel time*, i.e., a fixed period of time within which data transmission is expected to complete. Thus, the guards on the outgoing edges (with the synchronizing action `down!`) include $\mathtt{c} == \Delta$ to enable them after the channel time has elapsed, i.e., all relevant data has been transmitted. Additionally, an invariant $\mathtt{c} \leq \Delta$ ensures that the location is left as soon as Δ clock units elapse. The updated `from_region` variable (whose value is assigned on the `ack?` edge's reset assignment) determines the *region* location to enter after leaving `CLK_wait`.
- From `Transition_loc`, an outgoing edge labelled with the action `timeout!` is enabled when $\mathtt{rt\_count} \geq \mathtt{RT\_MAX}$. This is the only enabled edge when the number of retransmission attempts exceeds the predefined limit, and it leads to a location *Bad*, which is an absorbing location; this scenario needs to be avoided as a safety objective, as it indicates a *loss* on part of the scheduler in the game involving following a strategy that avoids this location.

Based on this understanding, the TGA model for the control loop is formally defined.

Definition 3-2.3 (Control Loop TGA Model)**.** *Based on the general TA representation of a control loop* $TA_{CL} = (L, l_0, Act, C, E, Inv)$ *(cf. Section 3-1), and considering a set of early triggering time parameters* $\{\underline{d}_1, \overline{d}_1, \ldots, \underline{d}_q, \overline{d}_q\}$ *such that*

$$\forall s \in \{1, \ldots, q\} : \overline{d}_s \leq \underline{\tau}_s,$$

integer variables

to_region, from_region, and rt_count,

and integer constants

N_{max}*, RT_MAX, and,* Δ*,*

the TGA model for a control loop, TGA^{CL}*, is represented by:*
$TGA^{CL} = (L^{CL}, l_0^{CL}, Act_c^{CL}, Act_u^{CL}, C^{CL}, E^{CL}, Inv^{CL})$ *where:*

- $L^{CL} = \{Transition\_loc, CLK\_wait, Bad\} \cup \bigcup_{i=1}^{q} \{R_i, Ear_i\}$;
- $l_o^{CL} = R_i$ *such that* $\xi(0) \in R_i$;
- $Act_c^{CL} = \{ack?, nack?, down!, timeout!\} \cup \{*\}$;

- $Act_u^{CL} = \{up!\}$;
- $C^{CL} = \{c\}$;
- E^{CL} = *a set comprising the following edges:*
  - $\bigcup_{i=1}^{q}(R_i,\ \underline{d}_i \leq c \leq \overline{d}_i,\ *,\ \emptyset,\ Ear_i)$,
  - $\bigcup_{i=1}^{q}\bigcup_{t\in\varepsilon_i}(Ear_i,\ true,\ up!,\ to\_region := t,\ Transition\_loc)$,
  - $\bigcup_{i=1}^{q}\bigcup_{\{t|(\mathcal{R}_i\rightarrow\mathcal{R}_t)\in E\}}(R_i,\ \underline{\tau}_i \leq c \leq \overline{\tau}_i + N_{max},\ up!,$ $to\_region := t,\ Transition\_loc)$,
  - $\bigcup_{i=1}^{q}(Transition\_loc,\ \{rt\_count < RT\_MAX,\ from\_region == i\},$ $nack?,\ rt\_count := rt\_count + 1,\ R_i)$,
  - $(Transition\_loc,\ true,\ ack?,$ $\{\ c := 0,\ rt\_count := 0,\ from\_region := to\_region\},\ CLK\_wait)$,
  - $\bigcup_{i=1}^{q}(CLK\_wait,\ \{\ to\_region == i, c == \Delta\ \},\ down!,\ c := 0,\ R_i)$,
  - $(Transition\_loc,\ rt\_count \geq RT\_MAX,\ timeout!,\ \emptyset,\ Bad)$;
- $Inv^{CL}(R_i) = \{\ c \mid c \leq \overline{\tau}_i + N\_max\}$, $Inv^{CL}(CLK\_wait) = \{\ c \mid c \leq \Delta\}$ *(additionally, implicit invariants apply on all urgent locations (cf. Definition 3-2.1)).*

In the definition, ε_i refers to the set of reachable regions from a location on early triggering based on the flow pipe, i.e., such that $\{t \mid \mathcal{X}_{[\underline{d}_i,\overline{d}_i]}(R_i) \cap R_t \neq \emptyset\}$. E is the set of edges as defined in the general control loop TA definition in Section 3-1.

Along the lines of [2], we define an additional *initial* location $\mathcal{R}_0$ from which there are outgoing uncontrollable edges to each *region* location (i.e. $\{\mathcal{R}_1, \ldots, \mathcal{R}_q\}$) and an invariant $\mathtt{c} = \mathtt{0}$ so that the location is left immediately and the control loop is in one of the actual region locations. This is done because the physical control loop could be in any region based on its initial conditions and it is desirable to generate strategies (as will be shown later) for any case; the edges are uncontrollable to signify that the scheduler cannot control the loop's initial state's region. Figure 3-4 shows the relevant additions to be made to the control loop TGA model from Figure 3-3 to include this concept.

3-2-3 Modelling as NTGAs

The communication network and control loops operate in parallel, i.e., they are entities that perform their functions individually at the same time, but with occasional interactions (like requesting network access). Thus, from a TA perspective, we model the system as a parallel composition of the network TA and the control loop TAs, giving us an NTGA:

$$\mathtt{Sys\_NTGA} = \mathtt{Network} \,|\, \mathtt{Control\_Loop\_1} \,|\, \ldots \,|\, \mathtt{Control\_Loop\_n}. \tag{3-1}$$

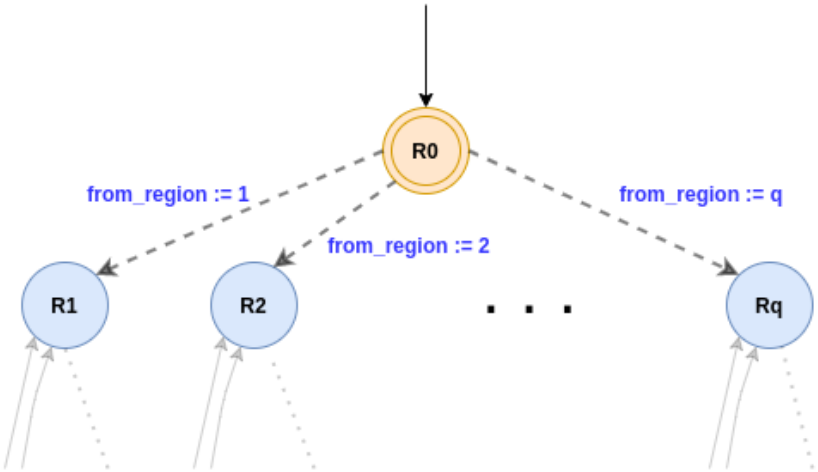

Figure 3-4: The control loop TGA model with an additional initial location R_0

3-2-4 Caveats

There are some caveats to be made with regards to the TGA models introduced. They include the following:

- For simplicity, a single triggering coefficient (α) is assumed in the control loop TGA model. The model can be extended to have an controllable action for selecting a triggering coefficient along the lines of [2];
- All synchronizing actions are assumed to be *binary* synchronizations, i.e., an input action corresponds to exactly one output action - for instance, if multiple edges with an `up?` action are enabled, and an edge with a `up!` action is also enabled, only *one* edge from those with `up?` can be taken along with the `up!` edge. Moreover, an edge with `up!` necessarily needs a corresponding edge with `up?` to be enabled;
- The channel occupation time, Δ is assumed to be small enough that it does not affect the transition relations, i.e., the locations resulting from an `ack?` action aren't expected to change based on the additional time spent in the `CLK_wait` location.
- The concept of *retransmission* of data is only used in the *modelling* of the system, and may not necessarily have a physical manifestation (i.e., as will be seen in the forthcoming sections, data is not transmitted over the CAN network when a `nack?` action is performed; based on the TGA model, such actions only *inform* the scheduler that a retransmission is needed, but eventually data transmission occurs only when an `ack?` action is performed).

3-3 Scheduling Strategies

In this section we cover how the TGA models from the previous section are used for generating scheduling strategies by defining the formal objectives of the system.

3-3-1 Safety Objective

We propose a *safety* objective - the strategy followed by the scheduler should be such that none of the control loops ends up reaching its *Bad* location as a result of any *run* (cf. Section 2-3-4) of the NTGA. Since our TGA model definitions are such that any control loop ending up in the *Bad* location also causes the network's TA to end up in its *Bad* location, a sufficient safety objective is defined as avoiding a set of states (say, `Avoid`) such that the network TA is in the *Bad* location, akin to [2]:

$$\texttt{Avoid} = \{l_{net}, l_1, \ldots, l_n, u_{net}, u_1, \ldots, u_n \,|\, l_{net} = Bad\}. \tag{3-2}$$

Based on the dynamics of the control loops and their ETC implementations, satisfying this objective should guarantee stability of the overall system. However, the scheduler may end up triggering early too often as a result of following this strategy "as is", thereby increasing network usage (in other terms, behaving *conservatively*).

To reduce the number of early triggers, we make slight modifications to the existing NTGA by defining additional *global* variables to be used for the purpose.

3-3-2 Limiting Early Triggers

We follow the modification proposed in [2] for limiting the number of consecutive early triggers, applying it to the system NTGA model defined in Section 3-2-3. Since *early triggering* is in the context of control loops, we create a modified version of the control loop TGA model. The following changes are made to the TGA's set of *edges* from Definition 3-2.3:

- Edges going to *early* locations have an additional guard and a reset assignment:
  $\bigcup_{i=1}^{q}(R_i,\ (\underline{d}_i \leq c \leq \overline{d}_i) \wedge (earlyCount < earlyMax),\ *,$
  $earlyCount := earlyCount + 1,\ Ear_i)$

- Outgoing edges from all R_i locations have an additional reset assignment:
  $\bigcup_{i=1}^{q} \bigcup_{\{t|(\mathcal{R}_i \rightarrow \mathcal{R}_t) \in E\}}(R_i,\ \underline{\tau}_i \leq c \leq \overline{\tau}_i + N_{max},\ up!,$
  $(to\_region := t) \wedge (earlyCount := 0),\ Transition\_loc)$

In semantic terms, we impose a limit on the number of consecutive early triggers, `earlyMax`, and maintain a *global* count variable `earlyCount` to keep track of the current number of consecutive early triggers. When an early transition is taken, `earlyCount` is incremented by 1, and when a natural trigger occurs, `earlyCount` := `0` to indicate that the scheduler allowed the ETC mechanism to trigger on its own without forcing it.

By adding a guard on an early edge as `earlyCount < earlyMax`, we impose a condition that *the scheduler should not force the loops to trigger beyond a certain number, consecutively; it should let them trigger naturally.* This effectively reduces the number of early, forced triggers, thereby lowering the usage of the network. However, this could also leave the NTGA more prone to not finding a strategy that can avoid the *Bad* location; it is a matter of adjusting the `earlyMax` value appropriately to see if finding a satisfying strategy is possible or not.

Chapter 4

Hardware Setup and Software Toolchain

After introducing the preliminaries and building upon them to have theoretical abstraction models defined in the preceding chapters, now we move towards their implementation in two senses - first, translating the models into software to generate scheduling strategies, and then using them for scheduling control actions in a physical setup with simulated plants and state-feedback controllers connected to the scheduler via a CAN bus.

In this chapter, we first describe the hardware setup hosting the plants, controllers, scheduler, and the network, and subsequently introduce the software tools involved in the process. We finally proceed towards a detailed description of all modules, drawing a link between the software and hardware involved in achieving the objectives of this thesis.

4-1 Hardware Setup and System Overview

We first describe the hardware components involved, and then briefly state how the overall system is composed with an arrangement of these components.

4-1-1 Hardware Components

The theoretical abstraction models from the previous chapters are physically modelled and implemented as mathematical representations on hardware intended for the purpose; the hardware is briefly described in this section.

Note: A *controller* in this context refers to a PXI I/O controller module[1] from National Instruments, and not a controller from the control loop. Moreover, we refer to the devices by their product numbers for simplicity.

The following hardware is involved in implementing the theoretical models:

[1] *National Instruments*, "PXI Express: NI PXI 8880-e User Manual", March 2015.

- NI PXIe-8880: It is an embedded controller module running the Phar Lap ETS RTOS. All compiled code representing the plants, controllers, and the scheduler resides in these controller modules.
- NI-PXI 8512: It is a CAN interface module used for transferring data between the 8880 controllers over a CAN bus using the CAN protocol.
- NI PXIe-1071: It is a PXI chassis from National Instruments with slots for hosting one controller module, and multiple other modules; in this case, for hosting a 8880 controller and an 8512 module. The modules can communicate via an internal bus present in the PXI chassis.
- NI No-termination CAN cables: Cables for connecting CAN devices; in this case, for connecting 8512 modules.

4-1-2 System Overview

The physical system involves three PXI chassis, each hosting an 8880 controller and an 8512 module. A set of CAN cables is used to connect the 8512 modules such that one 8512 module is connected to each of the other two 8512 modules (this setup is a physical implementation of a controller machine connected to two plants via a CAN network).

Figure 4-1 shows a block diagram of this setup, and what the individual components and interfaces represent at the system level. Figure 4-2 shows the physical setup from a hardware perspective.

4-2 Software Toolchain

Several software programs are used to implement the various modules involved in the processes of modelling, strategy generation, plant and controller simulation, and inter-module communication. Broadly, the toolchain can be divided into two categories - (i) software for creating abstraction models and generating strategies, and (ii) software for creating a simulation environment in which to apply and test the generated strategies.

4-2-1 Generating Abstractions and Strategies

The tools in this section are used solely for defining TGA models and generating strategies with a safety objective:

- **UPPAAL**: It is an *integrated tool environment* used for modelling, verification, and validation of systems modelled as NTAs. In this thesis, an extension of UPPAAL, UPPAAL Stratego, is used for generating strategies from the system NTGA, and also for debugging the generated strategies using the UPPAAL built-in simulator. The NTGA itself is generated using Python, as is explained in the forthcoming points;

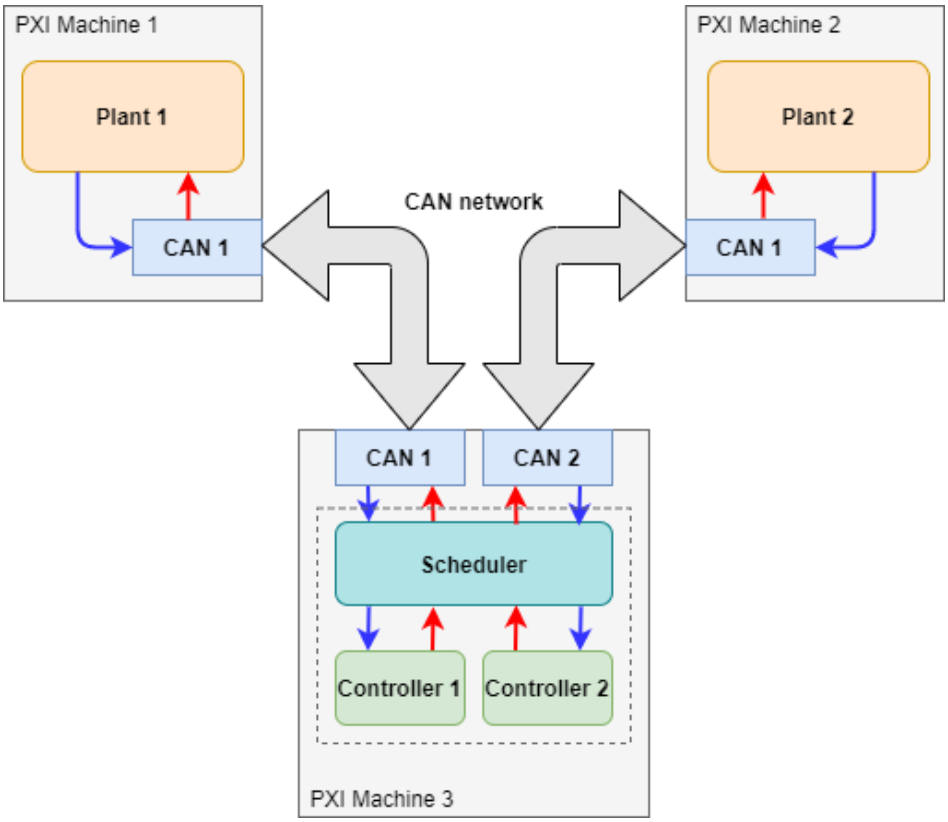

Figure 4-1: The system setup. The plants, scheduler, and controllers are hosted on 8880 modules, and CAN interfaces on 8512 modules. Note that the scheduler and two controllers are on the same 8880 (represented by a dotted rectangle). Arrows in red indicate flow of control input and arrows in blue indicate flow of state information

- **Python**: The programming language Python is used for defining the dynamics of the LTI plants, reachability analysis, and generating TGA models. Specifically, in addition to the Python standard library, the following *packages* are used:
  - *Python Control Systems Library*: For defining LTI plants with state-feedback controllers for which to generate TGA models;
  - *SciPy*: For linear algebraic operations (mainly matrix operations) and appropriate data structures (for instance, *NumPy arrays*);
  - *PyUPPAAL*: For generating TGA models that can be conveniently defined in Python and parsed by UPPAAL. A key point to note is that UPPAAL model files are XML files with a specific format, and hence the process of defining the NTGA models can be performed using string-based data structures in Python for convenience. The last step of generating UPPAAL-parseable XML files is delegated to PyUPPAAL;
  - *Pickle*: For saving and loading Python objects; a lot of the code being modular and spread across scripts, this package makes it convenient to transfer information between different Python scripts.

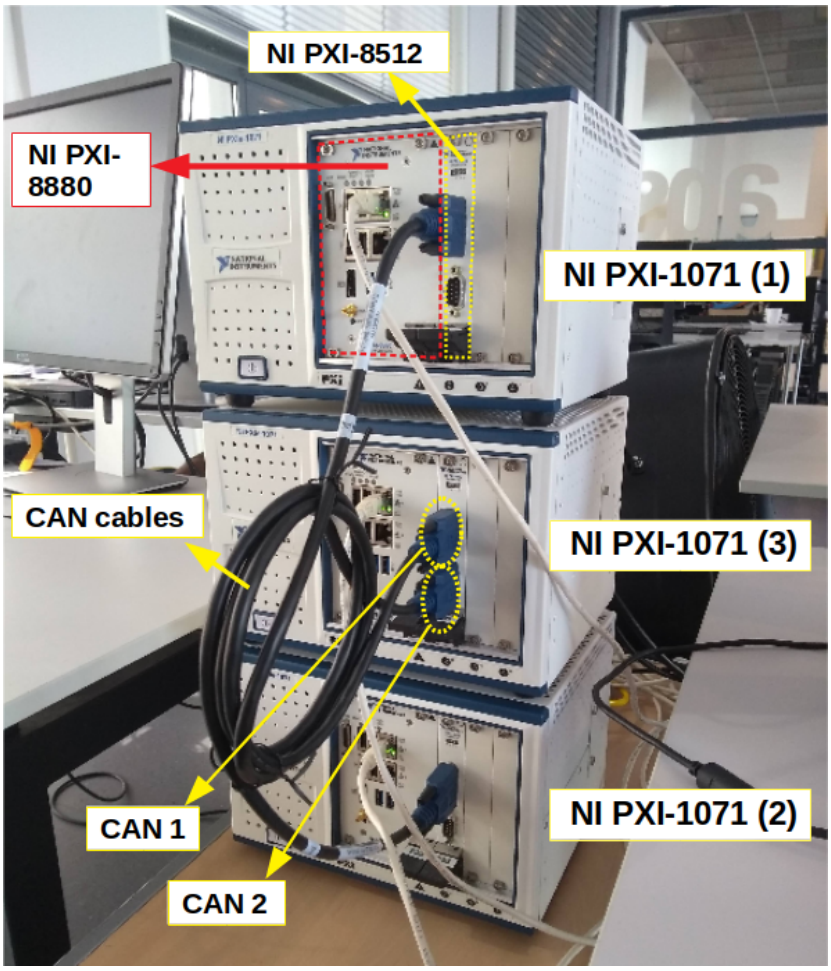

Figure 4-2: The physical setup, with all relevant components labelled

4-2-2 Simulation Environment and Strategy Parsing

The tools introduced hence are for creating a simulation environment in which to test the UPPAAL Stratego-generated strategies. The strategies themselves are in a format that needs to be parsed to obtain the necessary information for scheduling.

- **Mathworks Simulink**: It is a graphical programming environment for model-based design and simulation. In this thesis, it is used for defining LTI plant models with ports of appropriate dimensions for inputs and outputs, i.e. the control input vector $u(t)$ and the state output vector $x(t)$;
- **Simulink Coder**: It is a plugin for Simulink used for generating C/C++ code and compiled modules (DLLs) representing the models created in Simulink. We use it for generating C++ code representing the LTI plants;
- **NI LabVIEW**: It is a graphical programming environment in which code is represented using graphical elements; it is mainly used for applications involving multiple parallel modules. Given it is the case in this thesis, the bulk of the implementation is in

LabVIEW, as will be explained in the forthcoming sections. The version used for this thesis is NI LabVIEW 2016;

- **NI Veristand Model Framework**: It is a group of files (a *framework*) used with Simulink Coder to generate C/C++ code and DLLs that can be loaded as modules in LabVIEW;
- **NI Model Interface Toolkit**: It is a plugin to load DLLs generated using NI Veristand Model Framework in the LabVIEW environment; *Inports* and *Outports* defined in the source Simulink model are treated as input and output arrays in LabVIEW;
- **NI-XNET**: It is an instrument driver that provides APIs for interfacing with CAN, LIN, and FlexRay network hardware from National Instruments. We use the X-NET APIs for communication over the CAN network in this thesis;
- **JSON**: It is a general data-interchange format; it is used for loading Python-generated data in the LabVIEW environment in the scope of this thesis. The *data* includes certain matrices for region determination, and also strings with a defined format for interpreting UPPAAL-generated strategies in LabVIEW.

4-3 Offline Software Flow

Certain software routines need to be run before involving the control loop simulations over the physical CAN network, i.e., some computations need to be offline. Broadly, this offline software flow can be divided into multiple *phases*:

- Generation of control loops' region maps and transition relations as Python data structures;
- Generation of abstraction models in Python and their translation to an UPPAAL-parseable XML;
- Generation of strategies in UPPAAL and their parsing into JSON strings to be interpreted in LabVIEW.

The top-level flow of data is represented in Figure 4-3. We describe these phases in detail in their specific sections.

4-3-1 Region Maps and Transition Relations

Note: *The code used for creating region maps and transition relations described in this section is taken from the authors of [23].*

In this thesis, a PETC scheme involving time-based state space partitioning (cf. Section 2-2-1) is followed and accordingly the state space of each LTI plant is partitioned into regions. Thus, transition relations are in terms of *numbers of steps*, each step being separated by the sampling interval h.

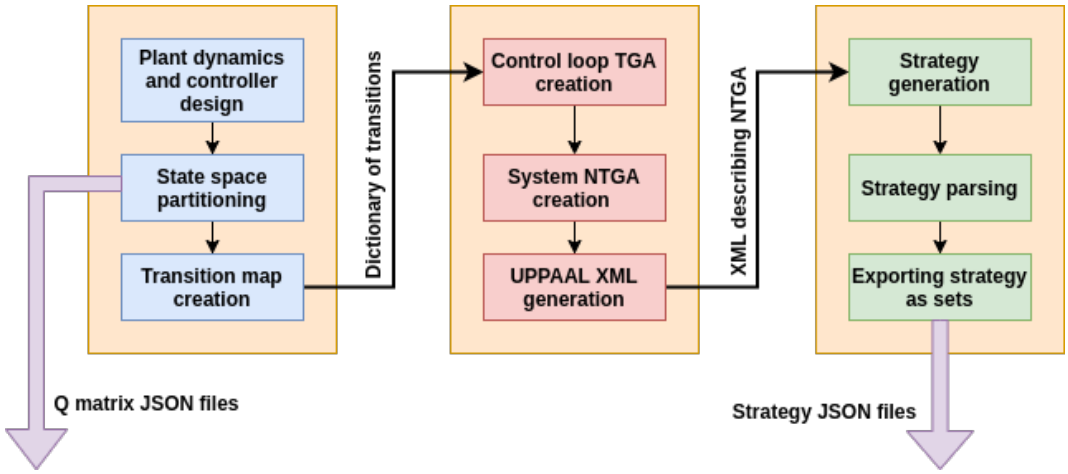

Figure 4-3: Flow of data in the offline computations part. The purple arrows indicate the data that is eventually required for online simulation.

The output of this phase is a Python *dictionary* in which the key is a *set of* `(source region, k)` pairs, and the corresponding value is a *set of* `(target region)` depending on which target regions are possibly reached from the source region on triggering after `k` steps.

Specifically, the following steps are performed:

1. LTI plants are defined and appropriate feedback controllers are designed (say, following the LQR algorithm) using the Python Control Systems package.
2. Following [23], *Q matrices* for each region are computed. In a nutshell, for a list of regions sorted in increasing order of their index numbers, for each region $\mathcal{R}_k$, a list of matrices is computed such that: for a list of Q matrices, $\forall Q$ in the list, if $x^T.Q.x \leq 0$, $x \in \mathcal{R}_k$. This information is essential for region determination, as will be seen during the section on simulation.
3. Based on the dynamics of the plants and triggering times, a transition map is computed, i.e., for each region, the possible target regions resulting due to triggering after k steps are recorded in a dictionary.
4. A dictionary is created for each control loop (its traffic model), and a list of these dictionaries is saved in a Pickle file for use in the abstraction models.

4-3-2 Abstraction Models

Note: *The code for implementing abstraction modelling described in this section is based on publicly available code*[2] *used in [9]. The TGA models used for reference in the original source code are from [2]; the code is appropriately modified to suit the TGA models introduced in Chapter 3 of this thesis.*

[2]Schalkwijk, P (2019) Python2Uppaal [Source code].
https://github.com/pschalkwijk/Python2Uppaal.

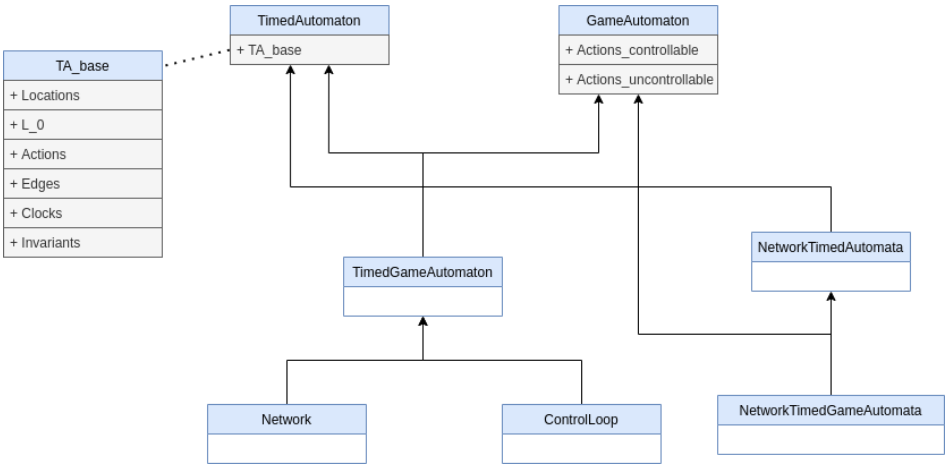

Figure 4-4: A simplified UML diagram representing the structure of the classes. The class `TimedAutomaton` consists of a reference to PyUPPAAL and is used for converting to an XML representation (based on [9])

An object-oriented approach is followed for defining classes of timed automata and timed game automata in Python, along the lines of [9]. The following steps are performed to eventually get an XML that can be correctly interpreted as an NTGA model in UPPAAL:

1. Python classes are defined for the network TA and the control loop TA. These classes consist of data structures containing all the necessary information as per the TA definitions (cf. Definitions 3-2.2 and 3-2.3) like regions, invariants, etc.

2. The network TA is defined with locations and guards as in Figure 3-2. It does not require any additional information.

3. The traffic model objects created in Section 4-3-1 are loaded using Pickle, and the data required for the control loop TA is collected, i.e., information about all regions and their transition relations.

4. In the control loop TA, locations for the regions (including *early* locations) are created, and the inter-region transition relations are translated to their corresponding edges along with the guards (represented using strings).

5. The remaining locations are added to the loop TA class (for instance, `Transition_loc`) and invariants are defined for all locations as applicable.

6. An `NTGA` class is defined for parallel compositions. An object of this class is instantiated with the network and control loops' models' objects used for the parallel composition, and this defines the system NTGA.

7. `PyUPPAAL` is used for translating the system NTGA object to an XML file to be interpreted in UPPAAL.

```
State: ( cl1PHDvPH.R27 cl2ExQmKf.Trans_loc NetworkRDmWHa.InUse_ack )
EarNum=1 EarMax=4 cl1PHDvPH.to_region=27 cl1PHDvPH.from_region=27
cl1PHDvPH.count=0 cl2ExQmKf.to_region=27 cl2ExQmKf.from_region=10
cl2ExQmKf.count=1

When you are in (cl1PHDvPH.c-cl2ExQmKf.c<=-8 && cl2ExQmKf.c<10), take
transition NetworkRDmWHa.InUse_ack->NetworkRDmWHa.InUse { true, ack!, 1 }
cl2ExQmKf.Trans_loc->cl2ExQmKf.Clk_wait { true, ack?, c := 0,
from_region := to_region, count := 0 }
```

Figure 4-5: A snippet from a generated strategy

4-3-3 Strategy Generation

The NTGA XML generated from the previous phase can be verified for its syntax using UPPAAL. It is then used for generating a scheduling strategy by providing a formal specification for the safety objective. The steps followed are:

1. The scheduling objective is stated formally in UPPAAL as:

   ```
   control:  A[] not Network.Bad
   ```

   which semantically means that the scheduling objective is a pure safety objective such that it should avoid reaching the network's *Bad* location (`Network.Bad`) [7].

2. Using this objective and UPPAAL's command-line interface with the appropriate input arguments, a *strategy* file is generated which consists of several *if-then* conditions involving clock value comparisons, and the actions (edges) that the game player (scheduler) needs to take such that `Network.Bad` is always avoided.

3. Since the output strategy has a definite structure (as seen in Figure 4-5), it can be translated into a more useful format for extracting the information required for scheduling triggering actions. Regular expressions (*regexp*s) are defined based on this structure, and the following information is extracted *only* for cases in which successful triggering occurs (i.e. text blocks in the strategy file in which the `ack?` action is present) using Python:

   - Control loops' source regions;
   - Clock conditions;
   - Control loop to trigger.

   The remaining text blocks are not relevant to the scheduler (i.e., they consist of actions in which the scheduler has to *wait*) and are hence not considered as useful information; no data is extracted from these blocks. The regexp and exact data structure generated can be found in Appendix A.

4. The extracted information is converted to a JSON string and exported to a file, which is to be loaded and used in the simulation environment by the scheduler, as will be explained in the next section.

4-4 Simulation System

We cover various aspects involved in simulating the NCS and how the strategy resulting from the offline computation phase is used for scheduling control actions.

4-4-1 Plant and Controller Models

- Plants are defined as LTI systems using the appropriate blocks in Simulink, with inports representing the control input vector *u(t)* and outports representing the state vector *x(t)*. Figure 4-6 shows a plant model created in Simulink.

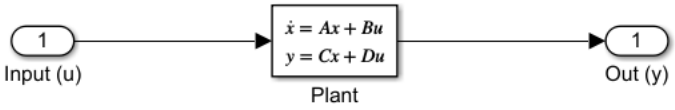

Figure 4-6: Simulink LTI plant model

- Using Simulink Coder and NI VeriStand Model Framework, DLLs corresponding to these plant models are generated. These DLLs have entry points that are recognized by the NI Model Interface Toolkit, and using the appropriate LabVIEW APIs, they can be loaded as blocks and *ticked* (that is, the model can be run for one time-step as defined in Simulink) - the input array to the block being the input to the *inport* of the Simulink model, and the output array from the block being the output from the Simulink model's *outport*. By ticking the loaded model continuously in a loop with the appropriate time-step, the dynamics can be simulated in LabVIEW.
- State feedback controllers are used for closing the control loop and stabilizing the plants; feedback gain is calculated using a known algorithm (such as LQR). The control loop, however, is closed over the network - the controller block is not present on the same machine as the corresponding plant.
- The control input value to the plant is provided in a zero-order hold fashion, and updated only when the control loop is triggered. The zero-order hold input is provided via an input array to the plant block, and the current state of the plant is obtained from the output array of the plant block.

4-4-2 CAN Network Communication

NI-XNET uses a FIBEX database [29] to describe a cluster connected over a CAN bus. Accordingly, we create one to describe the various communicating interfaces involved in this NCS. *We assume for the scope of this thesis that the state is two-dimensional and the control input is a scalar.*

Each control loop has the following CAN frames:

- *Control input frame*: This frame consists of a signal representing the control input as an IEEE floating point number;
- *State vector frame*: Since CAN frames can hold a maximum of 8 bytes of data, we use 4 bytes for each dimension of the state to be represented as an IEEE floating number, i.e. the state vector $[X_1\ X_2] = [F_{0,31}\ F_{32,63}]$ where $F_{x,y}$ denotes the decimal value represented by the bits between positions x and y of the CAN frame data.

The NI-XNET Read API is such that it buffers the last frame value transmitted over the CAN bus, and returns the same value unless a new value for that frame is written to the bus using the NI-XNET Write API [8]. Hence, this allows an inherent ZOH implementation of the control input, which is obtained from the Read API and provided to the plant model. Accordingly, the updated control input value is written to the network via the Write API only when triggering needs to take place, as dictated by the scheduler.

Remark 4-4.1. *After applying the ETC mechanism, we can see that some plants require triggering more frequently (based on the IET bounds) and hence need to be prioritized for triggering. Accordingly, each plant is assigned an arbitration ID - plants that need to be prioritized are assigned lower values to indicate that they get a higher priority in the CAN network for data transmission. However, this is not a strict relation between CAN priorities and plant dynamics; appropriate abstraction modelling and strategy generation ensure that communication conflicts do not arise.*

4-4-3 State Space Region Determination

Earlier, we introduced the notion of *Q matrices* (cf. Section 4-3-1) for determining which region a given state belongs to. Now we describe how it is used and translated to the appropriate LabVIEW data structures for determining the current region of the control loop during the NCS simulation.

All of the Q matrices can be computed offline prior to starting the NCS simulation, but their values need to be stored since they are used during the simulation for region determination. Each Q matrix is in the form of an $n \times n$ NumPy array, where n is the dimension of the plant's state vector. Through Python, a JSON file consisting of key-value pairs with the following format is created:

```
"region_index":  (list of Q matrices for that region)
```

for each region in the partitioned state space. The keys are sorted in increasing order for speed optimization; additionally, each Q matrix is stored as a list of its constituent rows for JSON compatibility. The JSON file is loaded in LabVIEW to get the values of all Q matrices computed for all regions.

Thus, the goal of this section is to have a single `r` $\times$ `Q_max` $\times$ `n` $\times$ `n` matrix *per control loop* loaded in LabVIEW, such that:

- `r` = number of regions;
- `Q_max` = maximum length among all lists of Q matrices, across all regions;
- `n` = dimension of state space.

This 4-D matrix can then be used to get the appropriate lists of Q matrices and determine the region online during the NCS simulation.

However, due to limitations of the LabVIEW JSON API and lack of native support for *dictionaries* in LabVIEW 2016, the JSON file with Q matrices cannot be used "as is". Thus, a *supporting* JSON file with the following fields is created:

```
"regions":  <list of regions into which state space is partitioned>
"region1":  k1
"region2":  k2
⋮
"regionN":  kN
```

The `regions` field consists of a list of regions into which the state space is partitioned, say `[15, 16, 17, 18]`. Accordingly, for each region, its corresponding field consists of the *number of Q matrices* for that region. In this example, the supporting JSON file would look like

```
"regions":  [15, 16, 17, 18], "15":  20, ..., "18":  30
```

where region 15 has 20 Q matrices, and region 18 has 30. (Note that this is only a generic example to demonstrate the concept).

The LabVIEW JSON API provides a means to access a JSON file's data by means of *paths* - that is to say, for nested JSON data[3], index numbers are used to access specific elements. In this case, since the number of Q matrices is not the same for all regions, we make use of the supporting JSON file to get the range of indexes per region (i.e. if there are 20 Q matrices for a region, the index range is [0, 19]).

Additionally, the regions may not be in a fixed sequence (for instance, the regions could be 18, 20, 27, etc.) and hence we need an additional *index array* which has entries corresponding to the index of the region in the LabVIEW matrix. (i.e., if region 20's Q matrices are at index 4 in the single matrix in LabVIEW, the index array will have the value "4" at index 20. It will have 0's at indexes not corresponding to a region).

The algorithm for generating the single matrix in LabVIEW is presented formally in Algo 1. The output is the matrix itself and the corresponding index array.

Once this single Q matrix and index array are loaded before the simulation (offline), they are used during the simulation (online) to detect the region by following Alg. 2.

[3] An example of nested JSON data: `{"xy": "z", "ab": {"cd" : 2, "ef" : 3}, "gh" : 4}`. The path `"ab/cd"` yields the value 2, and `"ab/ef"` yields 3.

Algorithm 1: Generating single 4-D Q matrix and index array in LabVIEW

Data:
`Q_data` = Q matrices data loaded from JSON file using LabVIEW JSON API;
`Q_info` = Supporting information loaded from JSON file using LabVIEW JSON API;

Result: LV 4-D Q matrix, Index array (integer array)
begin
    `Q_max` = Maximum length of list of Q matrices among regions;
    `n` = Dimension of state space;
    `r` = Number of partitions;
    `region_max` = Highest index among regions;
    Initialize an array `idx_arr` = `int`[`region_max`];
    Initialize a matrix `Q_CL` = `r` × `Q_max` × `n` × `n`;
    for $i \leftarrow 0$ **to** r **do**
        `region_current` = `Q_info`["regions"][`i`];
        `num_Q_mats` = `Q_info`[`region_current`];
        `idx_arr`[`region_current`] = `i`;
        for $j \leftarrow 0$ **to** num_Q_mats **do**
            for $k \leftarrow 0$ **to** n **do**
                `row_path` = "`region_current`/`j`/`k`";
                `Q_CL`[`i`, `j`, `k`, :] = `Q_data`[`row_path`]
            end
        end
    end
end

4-4-4 Scheduler

In Section 4-3-3, the relevant information to be gleaned from the generated UPPAAL strategy was introduced. Now we describe how this information is used to translate the strategy to be interpreted as LabVIEW code, i.e., as a scheduler block.

An approach similar to the one for determining the current region is followed for loading data structures from Python to LabVIEW. The main difference in the case of loading the scheduler block is that instead of a matrix, an array of clusters is used. A description of this data structure is included in Appendix A.

The scheduler is run once *per iteration of the simulation loop*, and maintains the following information:

- For each control loop:
  - Current clock value
  - Current region
- Whether triggering needs to take place in the current iteration
- The control loop that needs to be triggered

Algorithm 2: Determining region of a given state

Data:
`Q_CL` = 4-D Q matrix containing all the information about a control loop's Q *matrices*;
`idx_arr` = Index array;
`regions` = List of regions;
`x` = current state vector

Result: Current region (an integer)

```
begin
    r = length (regions);
    Q_dim2 = Number of columns in Q_CL (Q_max from Alg 1);
    for i ← 0 to r do
        region_test = regions[i];
        region_idx = idx_arr[region_test];
        flag = true;
        for j ← 0 to Q_dim2 do
            Q = Q_CL[region_idx, j, :, :];
            if x^T.Q.x ≤ 0 then
                continue;
            end
            flag = false;
        end
        if flag then
            break;
        end
    end
    return region_test
end
```

Using regular expressions, the strategy is parsed such that only the *if-then* condition blocks which result in an `ack?` action are considered for triggering, i.e., `ack?` indicates successful network access, and hence a successful triggering attempt.

Thus, for each *control loop*,

- for each *region*, the following data is parsed:
  - the clock value range which satisfies the triggering condition;
  - the control loop to trigger.

When the control loops' current clock and region conditions satisfy the strategy's conditions for triggering, the scheduler accordingly sets the *triggering* flag to true, and the index of the control loop that needs to be triggered.

4-4-5 Online Simulation Loop

For simulating the NCS, the system is arranged as stated in Section 4-1-2. In this thesis, two LTI plants are simulated and connected to their feedback controllers and the scheduler via a CAN bus. Figure 4-7 shows an overview of the NCS simulation setup.

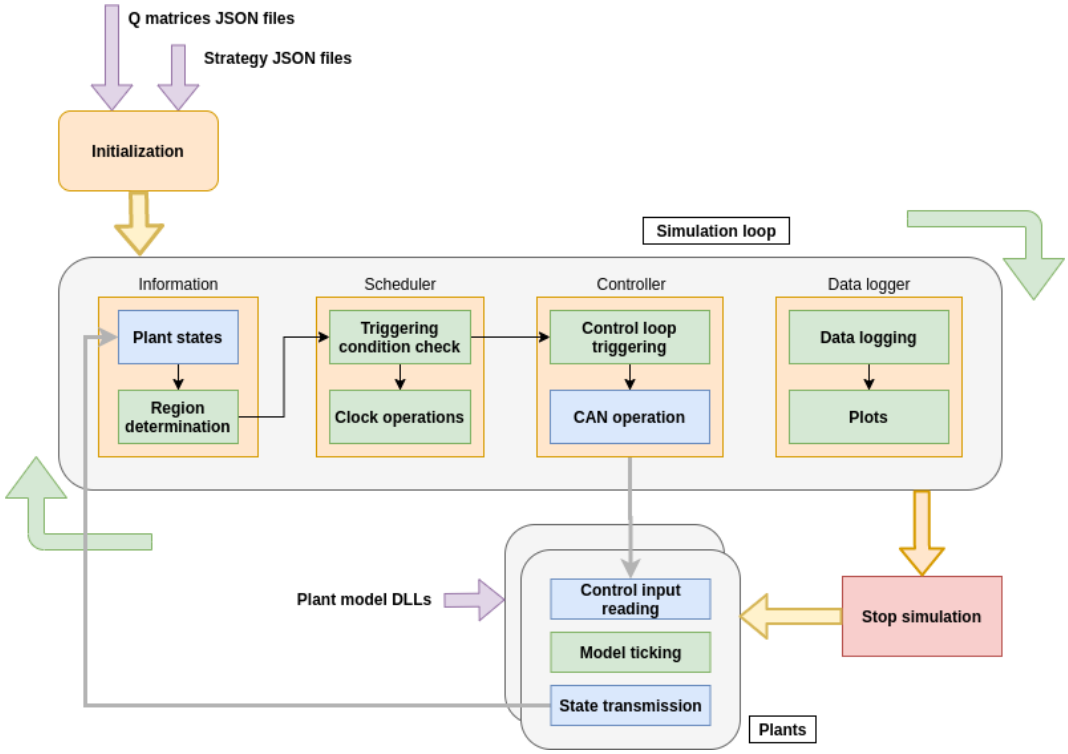

Figure 4-7: The NCS simulation setup. Arrows in purple represent data from offline computations, and arrows in grey represent data over the CAN network.

The plants run independently of each other and the network, and the control input to them is provided via the CAN bus. As stated previously in Section 4-4-2, the input read by the plant is the ZOH value for that control loop's input written to the CAN bus by the scheduler; it is updated appropriately when that control loop is triggered. On every iteration of the plant model *tick* (cf. Section 4-4-1), the current states of the plants are transmitted through the CAN bus in the corresponding CAN frames.

Prior to initiating the continuously running simulation loop, the following operations are performed:

- Loading of the Q matrices from JSON files into a single matrix;
- Loading of the UPPAAL strategy as described in Section 4-4-4;
- Sending signals to the PXI controllers hosting the plant simulation models to initialize them and begin running;
- Initializing data structures for scheduling (clock variables) and logging relevant data.

Once these operations are performed, the scheduler simulation loop is initiated - a fixed series of operations being performed on every iteration in one machine. With the plant models running independently in different machines, and being connected to their controllers via a network, the setup simulates an NCS.

Each iteration of the simulation loop consists of the following steps, performed successively (refer to Figure 4-7 for a graphical overview):

1. **Reading current states of the plants**: The CAN frames corresponding to the plant states are read to get the current output (states) of the plants.
2. **Region determination**: Based on the Q matrices computed and loaded prior to running the simulation (cf. Section 4-4-3), and after reading the states of the plants, the current region of each control loop is determined.
3. **Scheduler action**: Knowing the regions of the control loops and the current clock values, the scheduler can either determine that triggering needs to take place or not;
    - If triggering needs to take place as per the strategy, the control loop that needs to be triggered is recorded, and the corresponding clock variable is reset.
    - If triggering does not need to occur, i.e., the scheduler has to do nothing, the clock variables are incremented by 1 to indicate passage of time.
4. **Triggering**: If triggering has to occur, the feedback input is calculated based on the current state of the plant to be triggered, and the value is written using the appropriate CAN frame to the CAN bus.
5. **Data logging**: Relevant information (such as the current states of the plants and the scheduler) is written to appropriate variables.

On terminating the simulation loop, the accumulated data is logged to a file. Terminating the simulation loop also triggers a remote routine in the PXI controllers hosting the plant models that terminates their loops, thus bringing the overall system to an idle state.

Chapter 5

Experimental Results and Discussions

In this chapter we cover results obtained from simulating an NCS involving two plants and applying scheduling strategies generated using UPPAAL. We also show results obtained using periodic sampling instead of event triggering as a basis for comparison between the two approaches.

Additionally, to demonstrate the limitations of using TGA models for this application, we present the results obtained on increasing the number of partitions, in terms of increased memory footprint.

5-1 Plant Models

We use the LTI plant models and controllers defined in [2] in the NCS simulation.

Control loop 1:

$$\begin{aligned} \dot{x} &= \begin{bmatrix} 0 & 1 \\ -2 & 3 \end{bmatrix} x + \begin{bmatrix} 0 \\ 1 \end{bmatrix} v, \\ v &= \begin{bmatrix} 1 & -4 \end{bmatrix} x \end{aligned} \tag{5-1}$$

Control loop 2:

$$\begin{aligned} \dot{x} &= \begin{bmatrix} -0.5 & 0 \\ 0 & 3.5 \end{bmatrix} x + \begin{bmatrix} 1 \\ 1 \end{bmatrix} v, \\ v &= \begin{bmatrix} 1.02 & -5.62 \end{bmatrix} x \end{aligned} \tag{5-2}$$

The control loop in both cases is closed over the network, and the input v is provided in a ZOH fashion using the PETC mechanism. The state space is partitioned using the time-based approach, and the triggering condition is determined by a continuous-time Lyapunov equation as in [23], setting the design parameter $\sigma = 0.08$.

Table 5-1: Control loop information for the experiments

Parameter	Control Loop 1	Control Loop 2
Number of partitions	50	40
Sampling time, seconds (h)	0.01	0.01
Feedback gain	[1 -4]	[1.02 -5.62]
Arbitration ID	2	1

Table 5-1 shows the necessary information related to the control loops used in the experiments.

5-2 Simulation Results and Discussions

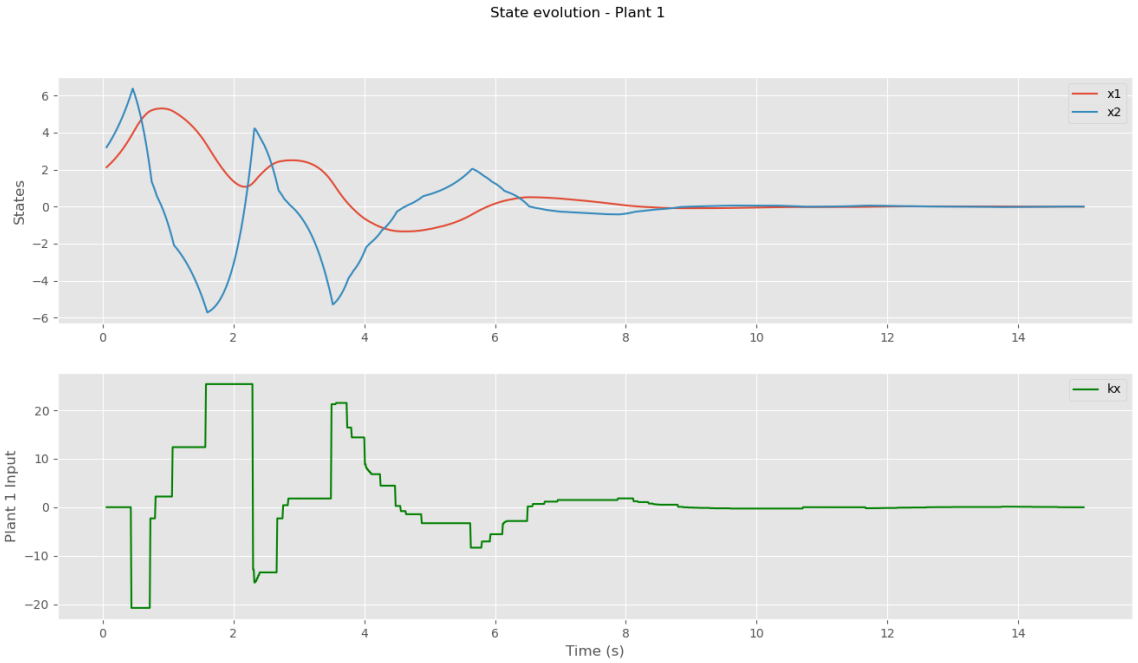

Figure 5-1: State evolution and control input for plant 1 with a scheduling strategy involving 50 state space partitions

The NCS simulation is run till the norm of the state vector is below a threshold $\epsilon = 10^{-7}$ for both plants. Figure 5-1 and Figure 5-2 show: (i) the evolution of the states for the plants, and (ii) the ZOH input provided to the plants over time. The control input plots show varying time intervals between triggers (noticeable through the *held* values) indicating an event-driven approach.

Figure 5-3 shows the *region map* of the plants, i.e., the computed state space regions over time. We can clearly see how plant dynamics affect the rate at which the plants enter different regions.

Additionally, Figure 5-4 and Figure 5-5 show the state evolution and control inputs for the

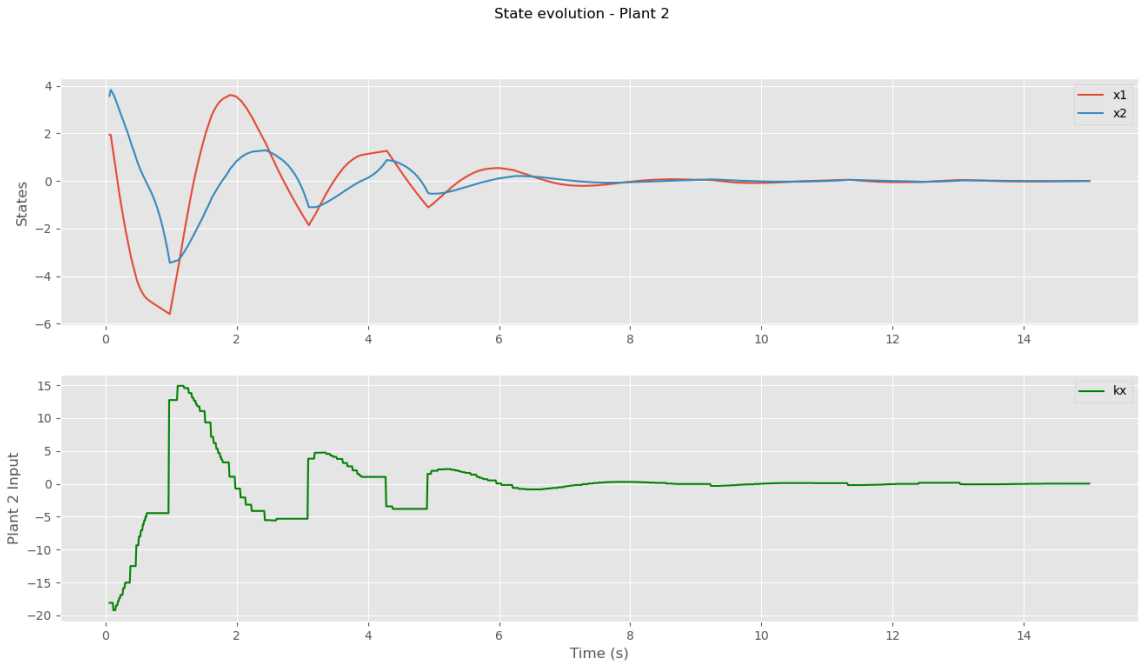

Figure 5-2: State evolution and control input for plant 2 with a scheduling strategy involving 40 state space partitions

plants in the case of periodic sampling - plant 1 being sampled once every 200 ms and plant 2 every 100 ms. Control actions are thus transmitted at a periodic rate and not scheduled as in the previous case.

We can further discuss the results of these experiments by observing the gathered data:

- On comparing the temporal evolution of the states between the ETC and periodic control approaches qualitatively, we can clearly see that the trajectories are much smoother in the latter, with faster settling times and fewer oscillations. However, this comes at the cost of additional network usage, as seen in Figure 5-6; in applications where performance is not critical and bandwidth is constrained, ETC can be useful. The tradeoff between network usage and performance is a design choice in such applications;

- Depending on the dynamics of the plants, some plants need a lot more triggering and may require more state space partitions than others to account for more accurate control. Figure 5-6 shows that in an ETC scheme, the second plant needs more triggers than the first one;

- The CAN protocol allows a maximum of 8 bytes of data per frame, and hence for plants involving more than two states (with floating point representations), a single frame may not provide enough resolution for transferring state information accurately. Multiple frames are required, which would increase network usage, and therefore ETC becomes even more relevant in such cases.

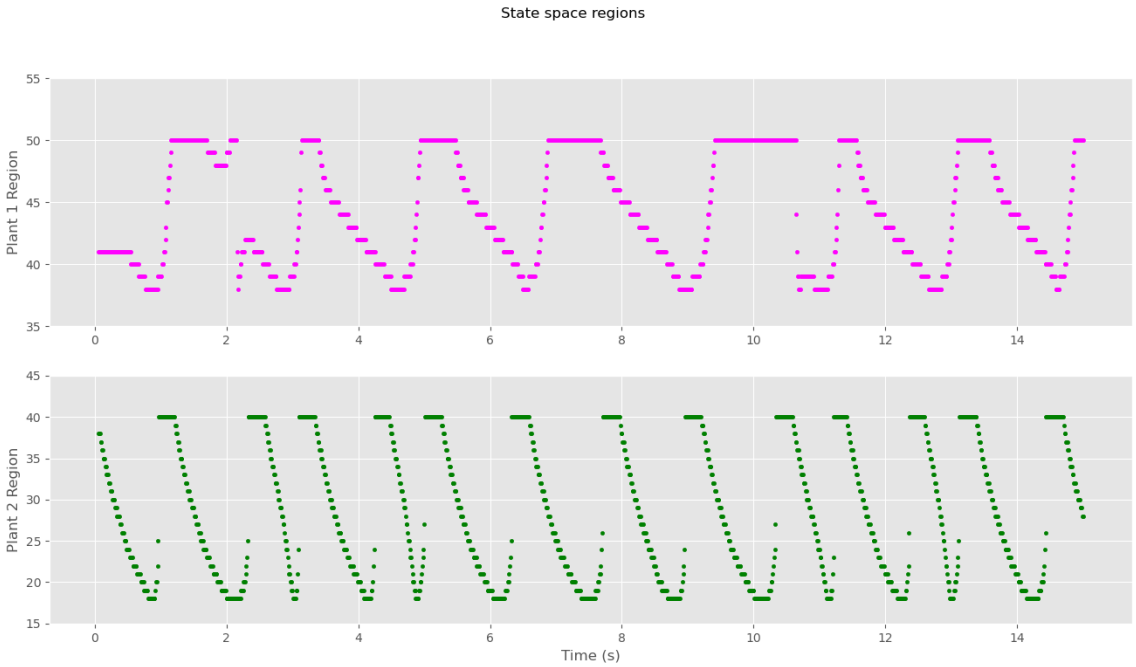

Figure 5-3: State space regions of the plants over time. Note the difference in the "frequency" based on plant dynamics.

5-3 Scalability of Abstraction Models

Since the traffic model involves computing transitions and tracking clock values for each case, we record the number of states generated by the strategy for increasing numbers of state space partitions, since this has a direct implication in terms of the scheduler's memory footprint.

Table 5-2 shows that the number of states stored grows almost ten-fold as we move from 40 partitions for each loop to 50 partitions. Accordingly, the amount of memory consumed while generating the strategies also increases from ~750 MiB to almost 5.5 GiB. Another point to note is that the number of states increases by different amounts depending on the control loop which is partitioned further (for instance, in the two cases wherein the total number of states is 85, the number of states for one case is more than twice the other).

Timed automata being defined around clock comparisons, each additional clock adds to the number of states exponentially, and hence the approach is not scalable used *as is*. Interested readers may consult [9] for more results on scalability of TGA model-based approaches.

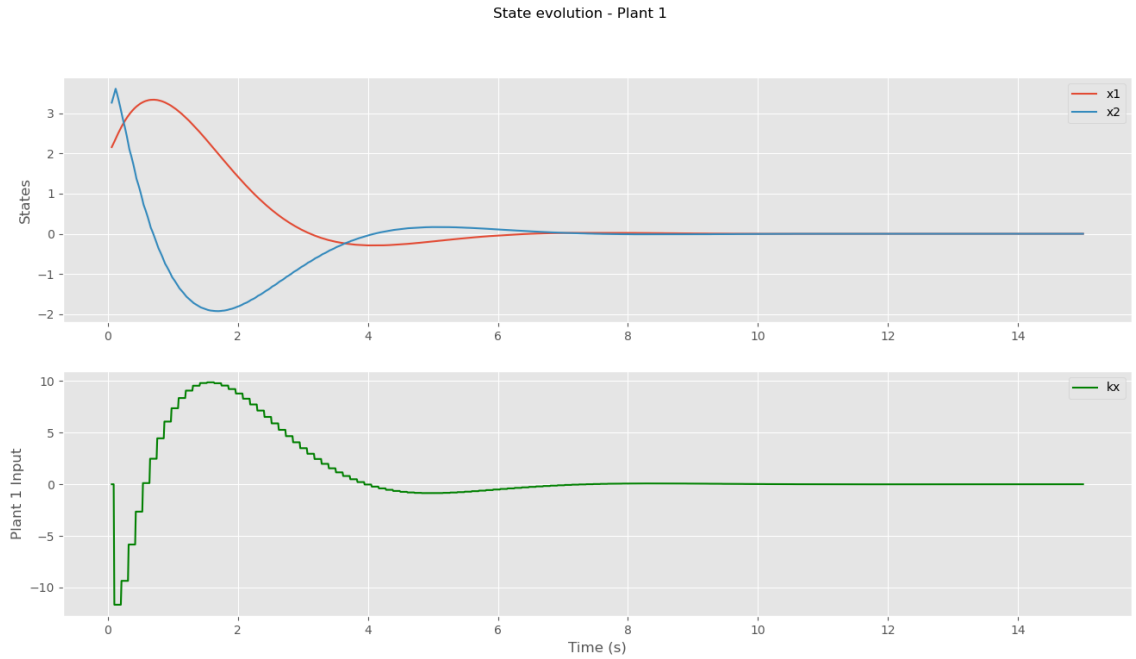

Figure 5-4: State evolution of plant 1 with periodic triggering. (Period = 200 ms)

Table 5-2: Memory footprint of scheduling strategies

CL 1 partitions	CL 2 partitions	States stored	Virtual memory (KiB)
40	40	186325	750844
40	45	236235	963412
40	50	272242	1102044
45	40	561079	1997540
45	45	702682	2577756
45	50	821923	2978768
50	40	1091524	3621396
50	45	1360620	4688048
50	50	1613242	5478480

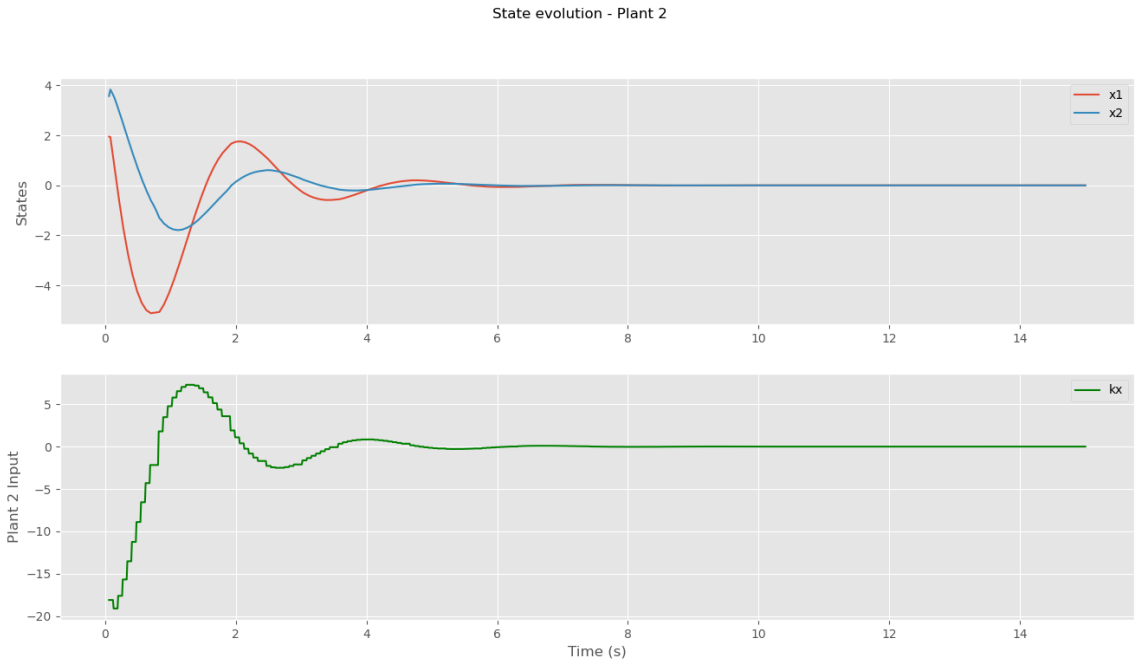

Figure 5-5: State evolution of plant 2 with periodic triggering. (Period = 100 ms)

Figure 5-6: Visualizing CAN network usage by showing triggering instances for the two control loops in (i) event-triggered, and (ii) periodic control cases. The mean number of triggering signals per unit time is computed as 0.0889 in the event-triggered case and 0.109 in the periodic case

Chapter 6

Conclusions and Recommendations

The main goals of this thesis were: (i) to create TGA models that can be used for generating scheduling strategies while considering *retransmission* of data over the network if it is already in use, and (ii) to demonstrate ETC in a practical setup involving a CAN network and simulated control loops.

In the various chapters, new TGA models suited for the retransmission requirement were described for generating scheduling strategies, a LabVIEW framework was created to implement ETC over a CAN network, and simulation results were shown to demonstrate the idea as a physical implementation.

A comparison was performed to demonstrate the performance-bandwidth tradeoff between ETC and periodic control in an NCS environment, especially when implemented over a physical network. The suitability of CAN was also shown for such applications, while at the same time the limitations of the new TGA models were shown by recording the memory footprint for a small increase in complexity.

In this thesis, though a certain approach was followed in terms of the traffic model and the triggering condition used, the physical implementation is also applicable to other combinations (for instance, with isotropic partitioning of the state space).

6-1 Directions for Future Work

There are clear directions in which further research can be extended from this thesis, either in the form of theoretical modifications or practical implementations.

- In the abstraction models used in this thesis, a fundamental assumption is that channel occupation time (Δ) does not interfere with the reachability analysis, i.e., it is small enough to be neglected. But for more accurate representations (and possibly slow networks), it should be taken into consideration.

- The *relaxation* provided through retransmission of data is considered to be *sufficiently small* in terms of the number of attempts and thus not affecting the transition relations, but in principle it requires a transition map of its own (with additional LMI computations to gauge the maximum relaxation in terms of clock ticks).
- Since this thesis showed an implementation of TGA models on a CAN network involving LTI systems, the concept can in principle be extended to nonlinear systems as well (depending on their modelling) [30].
- The triggering condition from [23] was not modified in this thesis, but the TGA models introduced can be combined with different traffic models (such as with a relaxed triggering condition introduced in [31]).
- The CAN protocol was used for implementing the connecting network between the control loops. There are several other industrial protocols (such as FlexRay, LIN, and EtherCAT) which can be considered for implementation.
- One fundamental assumption that is made in the CAN implementation in this thesis is that in case of failure to send packets over the network, the CAN driver handles retries and ensures that packets are sent eventually. However, in wireless ETC implementations, it may not be guaranteed (depending on the protocol), and implementation of approaches from [19] can be a direction to follow.
- In this thesis, a simple safety objective was considered to generate strategies, and hence the outcome may be quite conservative in terms of network usage. Pricing (using UPPAAL Stratego, for instance) may yield less conservative strategies and incorporating weights on edges in the TGA models may provide interesting results.

Appendix A

Code snippets

This appendix consists of relevant code snippets and some implementation details from various sections of the thesis.

- **Regular expression** for extracting strategy from raw strings to sets of relevant data (cf. Section 4-4-4):

```
{'\nState: \( .*\).*cl[A-z0-9]+\.
from_region=([0-9]+).*cl[A-z0-9]+\.from_region=([0-9]+).*\n.*'
'[\n]*When you are in \((.*)\).*[\n]+([A-z0-9]+)\.Trans.*,
from_region := to_region.*}\n'


;
```

- The front panel of simulation loop in LabVIEW is shown in Figure A-1;
- The plants are loaded in LabVIEW from their DLLs and run in a continuous loops as seen in Figure A-2;
- The code for generating a region's Q matrices from JSON files in LabVIEW is shown in Figure A-3;
- Figure A-4 shows how a region can be determined from a given state and the Q matrices;
- Some *type definitions* are shown in Figure A-5 and Figure A-6. In Figure A-5, the values stored by the scheduler across simulation loop iterations are shown - each control loop's last determined region, clock value, and ZOH input value;
- Figure A-6 shows the data structure for the generated strategy. Based on the regular expression, the following information is stored *per control loop, per region* as a list of sets:

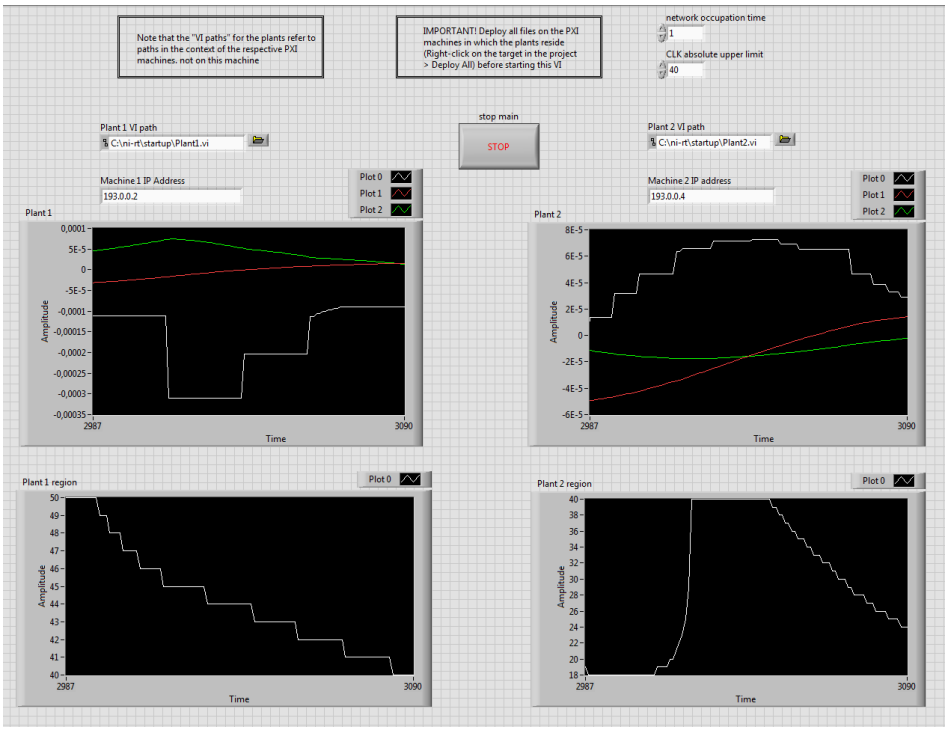

Figure A-1: Front panel of simulation loop

– Lower and upper clock bounds,

– If the clock values of the control loops need to be equal,

– If the clock values are related by an inequality (for instance, $CLK1 - CLK2 \leq 22$),

– The control loop to be triggered as per the strategy;

Given a region, its list of such sets is retrieved from the strategy JSON file loaded before the simulation loop begins execution. For each control loop, these conditions are checked - if all conditions are met, and the resulting *loop to be triggered* has the same value from both control loops' conditions, that loop is triggered over the CAN network.

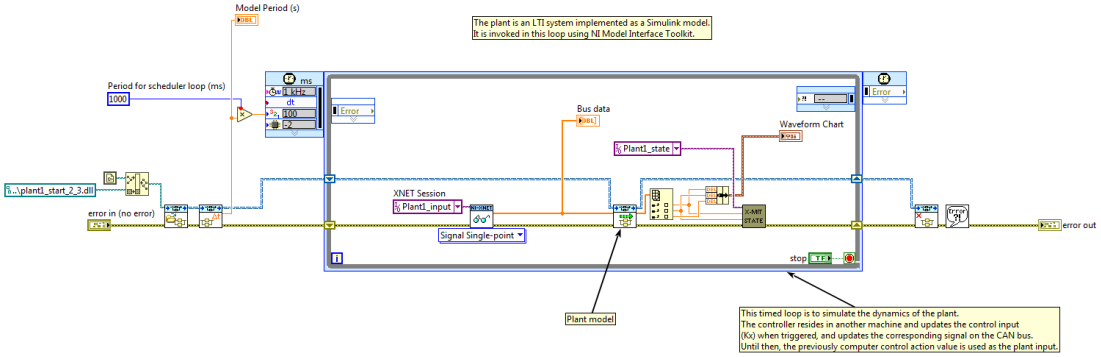

Figure A-2: Running a plant model in LabVIEW

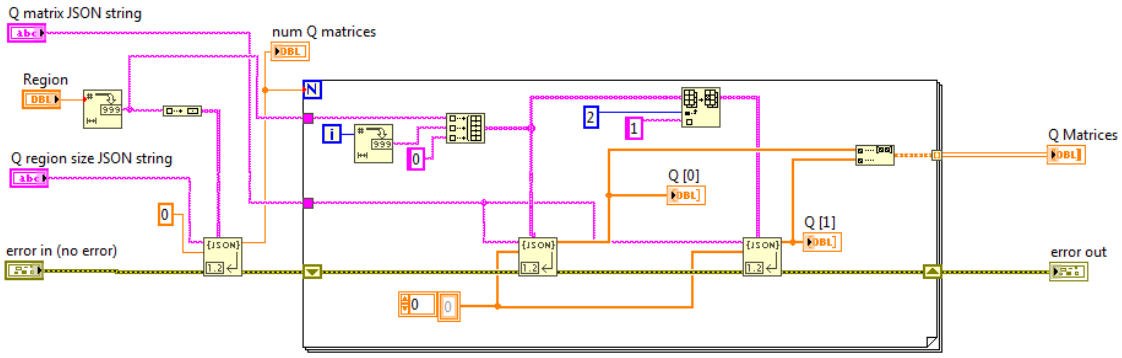

Figure A-3: Generating Q matrices for a region in LabVIEW

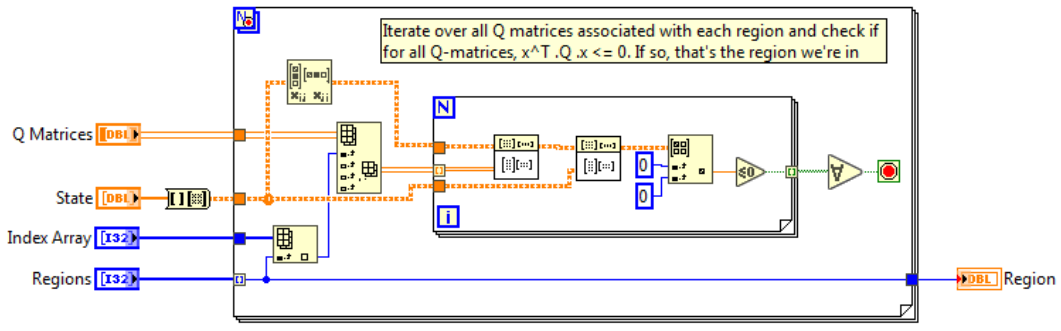

Figure A-4: Code for determining the current region

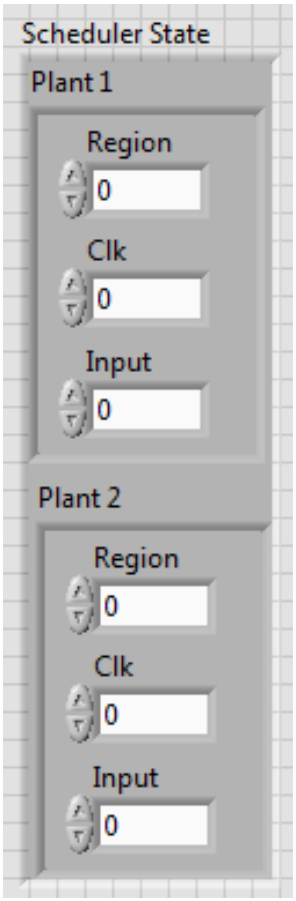

Figure A-5: Typedef of the scheduler's held information

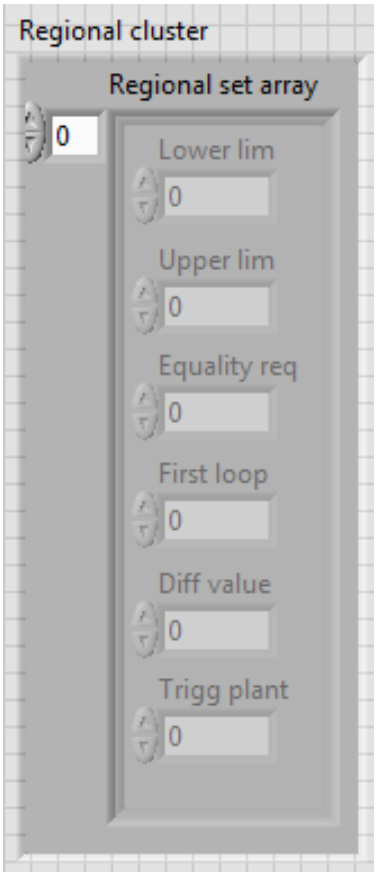

Figure A-6: Typedef for scheduler strategy

Bibliography

[1] J. Lunze and L. Grüne, *Introduction to Networked Control Systems*, pp. 1–30. Heidelberg: Springer International Publishing, 2014.

[2] D. Adzkiya and J. Mazo, Manuel, "Scheduling of Event-Triggered Networked Control Systems using Timed Game Automata," *arXiv e-prints*, p. arXiv:1610.03729, Oct 2016.

[3] W. P. M. H. Heemels, M. C. F. Donkers, and A. R. Teel, "Periodic event-triggered control for linear systems," *IEEE Transactions on Automatic Control*, vol. 58, pp. 847–861, April 2013.

[4] C. Fiter, L. Hetel, W. Perruquetti, and J.-P. Richard, "A state dependent sampling for linear state feedback," *Automatica*, vol. 48, no. 8, pp. 1860 – 1867, 2012.

[5] T. Koshy, "Chapter 11 - formal languages and finite-state machines," in *Discrete Mathematics with Applications* (T. Koshy, ed.), pp. 733 – 802, Burlington: Academic Press, 2004.

[6] A. David and K. Larsen, "A tutorial on uppaal 4.0," 01 2006.

[7] G. Behrmann, A. Cougnard, R. David, E. Fleury, K. G. Larsen, and D. Lime, "Uppaal tiga user-manual."

[8] National Instruments, *NI-XNET Hardware and Software Manual*, 7 2014.

[9] P. Schalkwijk, "Automating scheduler design for networked control systems with event-based control," Master's thesis, 2019.

[10] S. Bennett, "A brief history of automatic control," *IEEE Control Systems Magazine*, vol. 16, no. 3, pp. 17–25, 1996.

[11] W. P. M. H. Heemels and N. van de Wouw, *Stability and Stabilization of Networked Control Systems*, pp. 203–253. London: Springer London, 2010.

[12] K. G. Shin and P. Ramanathan, "Real-time computing: a new discipline of computer science and engineering," *Proceedings of the IEEE*, vol. 82, pp. 6–24, Jan 1994.

[13] G. C. Buttazzo, *Hard Real-Time Computing Systems: Predictable Scheduling Algorithms and Applications*. USA: Kluwer Academic Publishers, 1997.

[14] R. Alur and D. L. Dill, "A theory of timed automata," *Theoretical Computer Science*, vol. 126, no. 2, pp. 183 – 235, 1994.

[15] P. Tabuada, "Event-triggered real-time scheduling of stabilizing control tasks," *IEEE Transactions on Automatic Control*, vol. 52, pp. 1680–1685, Sep. 2007.

[16] M. Mazo Jr., A. Sharifi-Kolarijani, D. Adzkiya, and C. Hop, *Abstracted Models for Scheduling of Event-Triggered Control Data Traffic*, pp. 197–217. Cham: Springer International Publishing, 2018.

[17] W. P. M. H. Heemels, K. H. Johansson, and P. Tabuada, "An introduction to event-triggered and self-triggered control," in *2012 IEEE 51st IEEE Conference on Decision and Control (CDC)*, pp. 3270–3285, Dec 2012.

[18] A. Girard, "Dynamic triggering mechanisms for event-triggered control," *IEEE Transactions on Automatic Control*, vol. 60, pp. 1992–1997, July 2015.

[19] X. Wang and M. D. Lemmon, "Event-triggering in distributed networked systems with data dropouts and delays," in *Hybrid Systems: Computation and Control* (R. Majumdar and P. Tabuada, eds.), (Berlin, Heidelberg), pp. 366–380, Springer Berlin Heidelberg, 2009.

[20] G. de A. Gleizer and M. Mazo, "Self-triggered output feedback control for perturbed linear systems," *IFAC-PapersOnLine*, vol. 51, no. 23, pp. 248 – 253, 2018. 7th IFAC Workshop on Distributed Estimation and Control in Networked Systems NECSYS 2018.

[21] P. Tabuada, *Verification and Control of Hybrid Systems: A Symbolic Approach*. 06 2009.

[22] A. Sharifi Kolarijani and M. Mazo, "Formal traffic characterization of lti event-triggered control systems," *IEEE Transactions on Control of Network Systems*, vol. 5, pp. 274–283, March 2018.

[23] G. de Albuquerque Gleizer and M. M. Jr, "Scalable traffic models for scheduling of linear periodic event-triggered controllers," 2020.

[24] A. Fu and M. Mazo, "Traffic models of periodic event-triggered control systems," *IEEE Transactions on Automatic Control*, vol. 64, pp. 3453–3460, Aug 2019.

[25] A. Chutinan and B. H. Krogh, "Computing polyhedral approximations to dynamic flow pipes," 1998.

[26] T. Henzinger, X. Nicollin, J. Sifakis, and S. Yovine, "Symbolic model checking for real-time systems," *Information and Computation*, vol. 111, no. 2, pp. 193 – 244, 1994.

[27] J. Bengtsson and W. Yi, *Timed Automata: Semantics, Algorithms and Tools*, pp. 87–124. Berlin, Heidelberg: Springer Berlin Heidelberg, 2004.

[28] Y. Abdeddaım, E. Asarin, and O. Maler, "Scheduling with timed automata," *Theoretical Computer Science*, vol. 354, no. 2, pp. 272 – 300, 2006. Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2003).

[29] J. Krammer, P. Bornat, G. Dengel, S. Kuttenkeuler, R. Lukas, K. Oberhofer, J. Ruh, J. Stroop, and H. Quecke, "FIBEX – An Exchange Format for Networks Based on Field Busses," in *2nd Embedded Real Time Software Congress (ERTS'04)*, (Toulouse, France), 2004.

[30] G. Delimpaltadakis and M. Mazo, "Isochronous partitions for region-based self-triggered control," *IEEE Transactions on Automatic Control*, pp. 1–1, 2020.

[31] A. Szymanek, G. Gleizer, and M. Mazo Jr, "Periodic event-triggered control with a relaxed triggering condition," pp. 1656–1661, 12 2019.

Glossary

List of Acronyms

RR	Round-robin
EDF	Earliest deadling first
FP	Fixed-priority
CAN	Controller Area Network
NCS	Networked Control System
LTI	Linear Time-Invariant
ETC	Event-Triggered Control
PETC	Periodic Event-Triggered Control
LMI	Linear Matrix Inequality
TGA	Timed Game Automata
NTGA	Network of Timed Game Automata
ECU	Engine Control Unit
IET	Inter-Event Time
RTOS	Real-Time Operating System
ZOH	Zero-Order Hold