NTNU
Norwegian University of
Science and Technology

Application of Discrete-Event Dynamic Systems in Plant Analysis and Control

Mandar Thombre

Chemical Engineering
Submission date: June 2017
Supervisor: Heinz A. Preisig, IKP

Norwegian University of Science and Technology
Department of Chemical Engineering

Abstract

Chemical process plants typically follows continuous dynamics. However, for various plant operations like start-up, shut-down and maintaining safe operability, it is necessary to obtain information about the discrete state of the system under consideration. This is achieved by observing the continuous system with discrete sensors (temperature sensors, level sensors, etc.) which emit a signal when a process variable crosses a certain value, as opposed to at constant time intervals. This results in a 'quantized system' where the state-space is discretized by these discrete sensors. Each partition of this discretized state-space - known as a hypercube - corresponds to a different discrete state of the system. An 'event' in this context is when the system makes a transition from one discrete state to another adjacent discrete state. This quantized system can thus be said to be a discrete-event dynamic system (DEDS). The DEDS that is abstracted from a plant with continuous dynamics can be modelled as an automaton. This thesis broadly covers the modelling of such DEDS and two related aspects where these models can be used - hazard and operability (HAZOP) analysis and supervisory control in plants.

Conventional HAZOP techniques are not reliable for identifying low-frequency, high-risk hazards caused by multiple simultaneous failures. The total number of all possible failure overlaps is very high, making it is infeasible to analyse them using conventional techniques. The analysis can be done computationally using the DEDS model of the plant. HAZOP involves defining a region of 'safe' operation within the entire state-space. The plant automaton and this defined safe operability region can be combined to identify regions in the state-space where the state has the possibility to go out of safe limits - the so-called 'leaks'.

Synthesis of a supervisory controller - or supervisor - for plants to follow given specifications, is based on the automaton model of the plant. In addition to the automaton information, the control strategy also makes use of the underlying gradient information from the continuous dynamics. The control action happens through the use of discrete inputs, where some of the inputs can be used to 'force' a particular transition between adjacent discrete states. A supervisor so synthesized is itself a DEDS.

All the techniques presented in the thesis are explained using illustrative case studies and examples. The discussion points pertinent to the different concepts are also included.

Preface

This Master's thesis was written in the Spring 2017 semester. It concludes the 2 year Master's Degree program in the Department of Chemical Engineering at the Norwegian University of Science and Technology (NTNU), leading to an M.Sc. in Chemical Engineering. The final year of my studies was spent at the research group in Process Systems Engineering within the Department. The work performed in this thesis is an extension of the specialization project carried out in the Fall 2016 semester in the same research group.

I would like to thank my supervisor, Professor Heinz Preisig, for his support throughout the duration of this thesis. I am truly grateful to him for giving me the opportunity to work with him on some of his projects, while also giving me the freedom to develop my own ideas. Not only this, he has been an excellent mentor to me ever since I first came to Norway. I have certainly learnt a lot through the many long discussions I have had with him.

I wish to extend my gratitude to Professor Tore Haug-Warberg for helping me with various aspects of this thesis, and to Arne Tobias Elve for taking the time to review my thesis report. I would also like to thank my classmates for making these last 2 years a lot of fun. A special thanks to all the people who shared the study room with me this last semester - Cristina, Melissa, Petter, Eirik and others - for the many breaks, laughs and random discussions.

Finally, my thoughts go to my family and friends in India, without whose support my stay in Trondheim would not have been possible.

Declaration of Compliance

I declare that this is an independent work according to the exam regulations of the Norwegian University of Science and Technology (NTNU)

Trondheim, Norway
June 26, 2017

Mandar Thombre

Table of Contents

List of Tables

List of Figures

List of Symbols and Acronyms

Latin Symbol	Description
e	Event
E	Discrete set of events
G	Automaton
G_d	Deterministic automaton
G_{nd}	Non-deterministic automaton
h	Output function
$H_x(\tilde{x})$	Hypercube associated with $\tilde{x}$
$H_u(\tilde{u})$	Hypercube associated with $\tilde{u}$
$int(H_x(\tilde{x}))$	Interior of the hypercube associated with $\tilde{x}$
n	Dimension of the state-space
S	Supervisor
u	Continuous input
$\tilde{u}$	Discrete input
$\widetilde{U}$	Set of discrete inputs
x	Continuous state
$\tilde{x}$	Discrete state
$\widetilde{X}$	Set of discrete states
$\tilde{y}$	Discrete output
$\widetilde{Y}$	Set of discrete outputs
x_i, u_i	ith state, input
x^i, u^i	ith component of state, input

Greek Symbol	Description
β^i	Boundaries in the state-space in the ith dimension
γ^i	Boundaries in the input-space in the ith dimension
λ^i	Limits of the safe operability region in the ith dimension
ϕ	Transition function
Ψ	Safe operability region
$bd(\Psi)$	Boundary (hyper)surface of the safe operability region
Ω	State-space
ξ	Discretely controlled state trajectory

Acronym	Description
HAZOP	Hazard and Operability study
DEDS	Discrete-Event Dynamic System

Chapter 1

Introduction

The processes in chemical plants typically follow continuous dynamics, and can generally be described by differential equations. On the other hand, the control systems used to operate these processes are discrete, by virtue of being implemented by a digital computer. The closed-loop behaviour of this plant-controller system can thus be thought of as following both continuous and discrete dynamics (Koutsoukos et al., 2000; Stiver et al., 1996).

On the regulatory level of the plant, control usually involves implementation of control laws such as the PID control. A digital computer is capable of taking measurements at extremely small time intervals. Due to this fast sampling, a control system on the regulatory level is assumed to be continuous. The discrete part of the plant-controller system is considered insignificant in the design of controllers at the regulatory level.

On the supervisory level of the plant, however, tasks such as start-up and shutdown have to be implemented. Moreover, issues such as safety and operability - ensuring that the overall system stays within specified limits - may have to be considered. It thus becomes important to identify the *discrete state* of the system under consideration. Consequently, it is not possible to ignore the discrete nature of the controller on the supervisory level.

Understanding the interaction between the continuous plant and the discrete controller is important, since this is useful both in coming up with the model of the plant and in designing the controller. The interaction typically happens through an interface. The interface consists of a analogue-to-digital converter to relay information from the continuous plant to the discrete controller and a digital-to-analogue converter to relay information in the opposite direction.

Analysing the interactions in an overall system that is following both continuous and discrete dynamics is not straightforward. To make the analysis easier, the

plant and the interface can be modelled *together* as a discrete system (Lunze and Raisch, 2002; Lunze and Steffen, 2002). The overall plant-interface-controller system can then be represented by two discrete systems, one for the discrete controller and the other for the discrete plant-interface combination, as shown in Figure 1.1. Now the analysis relates to interactions between two discrete systems.

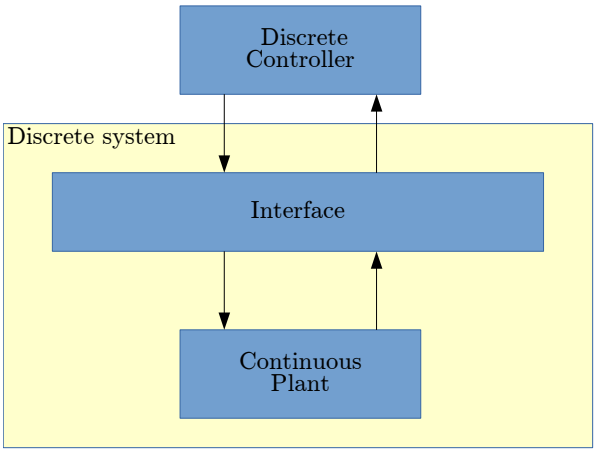


Figure 1.1: Interaction between a plant and a controller.

The plant-interface combination can be represented by two types of discrete systems at two different hierarchical levels in the plant. The first is the discrete-time system at the regulatory level of the plant (Santina et al., 2010). This is based on the discretization of time. The regulatory controller also follows discrete-time dynamics (albeit with extremely fast sampling). This abstraction thus involves interactions between two discrete-time systems (plant-interface combination and the controller).

The second is the discrete-event system at the supervisory level of the plant. This is based on the discretization of the state-space via the use of *event sensors*. Supervisory controllers are also typically discrete-event systems [1]. This abstraction thus involves interactions between two discrete-event systems (plant-interface combination and the controller [2]). This thesis deals with this latter type of abstraction: discrete-event systems, their modelling, analysis and control.

In case of discrete-event systems, the use of event sensors results in a discretized state-space, made of so-called *hypercubes*. Obtaining the discrete-event model of the continuous plant is then based on determining the direction of the

[1] Systems where the plant follows continuous dynamics and the controller follows discrete-event dynamics are commonly referred to as *hybrid* systems in literature.

[2] Also commonly referred to as the *supervisor*.

continuous state trajectory at the boundaries of each of these hypercubes. The direction is determined by checking the sign of the state derivative at the boundaries of the hypercubes, to check whether a particular transition is possible (Philips, 2001). The knowledge of component equilibrium *hypersurfaces* obtained from the continuous model can be exploited here (Preisig and Manenti, 2012). The model, known as an automaton, represents all possible transitions from each hypercube in the state-space.

Broadly, this thesis deals with the use of discrete-event dynamic systems (DEDS) in two areas: plant analysis and plant control. Specifically, the analysis part covers the hazard and operability (HAZOP) analysis in plants whereas the control part relates to automated supervisory control of plants. These topics are introduced here.

1.1 HAZOP study

The chemical industry has been prioritizing the safety of plants not only to ensure the well-being of the plant personnel, but also to make certain that the concerned stakeholders have confidence in how the plants are managed. To ensure that this confidence is maintained, it is important to account for the safety of the plant and the involved personnel in the design phase itself.

The HAZOP study is a systematic and structured technique to investigate a process, with the objective of identifying potential hazards and operability problems in the process (Thomas Marlin, 2014). Conventionally, HAZOP involves a team of experts examining the P&ID diagrams of the plant and analysing the effect of potential changes to process variables like temperature and pressure. The team relies on brainstorming, intuition and experience based on prior studies to determine the possibility of these deviations and how they might affect the safety and operability of the plant (Crawley et al., 2000; Dunj et al., 2010). Such HAZOP studies are usually time-consuming and expensive. Nonetheless, the approach is widely used in the process industry, and international standards have been established (IEC 61882:2001, 2001). A brief description of how a HAZOP analysis is performed conventionally is given in Appendix A.

An important aspect of doing a HAZOP analysis is identifying the potential causes of operational failure, or a hazard. There are various ways in which a failure or a hazard may occur in a process plant. It may be a result of sequential series of faulty events, one event causing the next. A very simple but realistic example of this kind would be a fault in the cooling system for an exothermic reaction, leading to increased reaction rate resulting in a runaway reaction. The pressure in the reaction tank would increase and possibly result in an explosion. Such failures or hazards, where a cause-and-effect relationship can be established, are fairly easy

to capture with the conventional HAZOP study described above.

In a large but highly interlinked plant with many process variables, a deviation in one part of the plant may affect a completely different part of the plant. Small changes to process parameters may have an unforeseen knock-on effect elsewhere. A simple cause-and-effect analysis is inadequate to identify hazards in such complex systems (Thomas Marlin, 2014). This is especially true when a failure or a hazard is a result of a chance overlap of two or more, possibly unrelated, occurrences in the plant. The total number of all possible overlaps in a complex plant is so high that failures or hazards resulting from these cannot realistically be identified by a conventional HAZOP study. The safety analysis of a complex plant should, therefore, not depend only on conventional HAZOP analyses.

This report focuses on a different, more quantitative approach to the HAZOP analysis that employs the discrete-event models abstracted from the continuous plants. The idea is to use the DEDS model of the plant i.e. the automaton in combination with a defined *safe operability region*. The automaton can be used to identify parts of this region where the state has a possibility of moving out of the safe region. Specifically, since the automaton provides a list of possible transitions from each hypercube in the state-space, it is possible to identify those hypercubes where the state has an outward transition with respect to the safe region (Preisig and Manenti, 2012).

Such a quantitative analysis has the benefit that it can handle systems where a lot of process variables have to be considered. It can identify hazards arising out of all different combinations of (discretized) process variable values, since all combinations are considered in the automaton. The approach may be used as a complement rather than as an alternative to the conventional HAZOP analyses.

1.2 Supervisory control of DEDS

Normal control strategies on the regulatory level involve some insight from experts or prior knowledge of the workings of the plant. There are various rules relating to the various different parameters, different operating conditions, etc. that one needs to be aware of when implementing such a control strategy. Examples of such strategies include implementation of control laws such as the PID control.

As discussed, on the supervisor level, one is concerned with information pertaining to the discrete state of the system. Event sensors are used to discretize the state-space of the system, resulting in a discrete-event equivalent of the plant - an event being the transition between the different discrete states. Supervisory control strategies then need to be devised for this discrete-event equivalent of the plant.

Now, the topic of supervisory control of discrete-event systems is well studied in literature. The supervisory control methods developed are based on the frame-

work of so-called formal language models (Ramadge and Wonham, 1987a,b; Wonham, 1989). These methods were first proposed in the 1980s and several extensions have been proposed since then (Charbonnier et al., 1999; Koutsoukos et al., 2000; Kumar and Garg, 1995; Stiver et al., 1996; Thistle, 1996). Notably, though, most of this research has been developed for purely discrete systems, like the ones encountered in manufacturing systems.

However, in case of chemical plants, the actual process dynamics are continuous and discrete-event equivalents are only abstractions of this underlying continuous dynamics. The additional information provided by the continuous nature of the process, specifically gradient information, can be very useful when developing supervisory control methods. Control methods that exploit the continuous nature have been presented in Philips et al. (1999a,b).

Supervisory control may have to be developed for a variety of specifications, and there are various ways in which these specifications may be modelled into the overall system (Cassandras and Lafortune, 2010). One common specification relates to steering the system from one state in the discretized state-space to the other. This is the reachability specification and it's the specification primarily considered in this work.

The supervisory control synthesis is an automated procedure and does not require case-by-case experience or expert insight. Nor is there any requirement to follow specific rules pertaining to the process operating conditions, etc.. The supervisory controller is itself a discrete-event system and is described by an automaton. The automaton based procedure ensures that the supervisory controller is prepared for all kinds of situations that may occur in the plant, since the automaton quantitatively encapsulates all the relevant process information. This is an advantage, especially in case of complex systems.

1.3 Objectives of the thesis

Given the preceding discussion, the objectives of this thesis can be broadly stated as follows:

1. To present a modelling formalism for obtaining a DEDS model from a continuous plant model described by differential equations.

2. To combine the DEDS model with a defined safe operability region for performing a quantitative HAZOP analysis

3. To investigate supervisory control of the DEDS models abstracted from continuous plants, especially with regards to designing a supervisor to meet the reachability specification.

The thesis mainly presents a theoretical analysis and consolidation of the various methodologies in a systematic manner. Case studies are also used to illustrate the techniques of quantitative HAZOP and supervisory controller synthesis.

1.4 Structure of the report

Chapter 2 aims to highlight the main difference between discrete-time and discrete-event systems obtained from an underlying continuous system. This difference has to do specifically with the interface that is used between the continuous plant and the discrete controller.

Initially in Chapter 3, some basics of automata theory are presented. This includes the definitions of different types of automata, like deterministic, non-deterministic and input/output automata. Next, the main modelling formalism of abstracting DEDS models from continuous plants in presented in a systematic manner. This includes the discretization of state-space and input-space into hypercubes, and the computation of the transition function. The notion of hypercubes and hypersurfaces is formally described in this chapter. The computational cost involved in obtaining the DEDS model, specifically relating to the problem of combinatorial explosion, is briefly discussed The chapter concludes with a discussion on some pertinent issues/limitations specific to this modelling formalism.

Chapter 4 discusses the quantitative HAZOP analysis. The notion of a safe operability region is formalized. The chapter explains how to use the knowledge provided by the DEDS model of the plant to identify those regions in the state-space where the safe operability region may be breached. These are referred to as *leaks* and are formally defined. An illustrative case study involving two interacting tanks is presented to demonstrate the use of this quantitative HAZOP technique. The chapter concludes with a discussion on some pertinent issues related to such a HAZOP analysis.

Chapter 5 deals with supervisory control of DEDS models obtained from continuous plants. It introduces the different kinds of specifications that a supervisor may have to fulfil. First, a simple example demonstrating the supervisor synthesis for realizing the reachability specification in a purely discrete system is given to build intuition. Next, specific supervisory control actions that rely on the underlying continuous dynamics of the DEDS model are presented. A modified two-tanks case study is given to demonstrate the synthesis of the supervisor using the plant automaton and the control actions. The last part of Chapter 5 presents pertinent discussion points in supervisor synthesis, and control.

It must be noted that the discussion sections relevant to the different methodologies are presented within the corresponding chapters. Finally, Chapter 6 gives recommendations for future work and concludes the thesis.

1.5 Previous work

This thesis is an extension of the work conducted by the author on a specialization project in the Fall of 2016. Some of the content presented in the specialization project report is reused in this thesis, but with several changes and improvements made after the completion of the specialization project.

Furthermore, during the course of this thesis, an article (Thombre and Preisig, 2017) based on the thesis work was written and sent to the ESCAPE-27 conference (European Symposium on Computer-Aided Process Engineering, Barcelona, 2017) for peer review. The article was accepted for the publication and will be published in the ESCAPE-27 conference proceedings in October 2017. The author of this thesis is the first author of the said article. Some parts of Chapter 3 and Chapter 4 overlap with the contents of the article due to be published.

Chapter 2

Going from continuous to discrete systems

This chapter serves to make clear the distinction between the following types of systems, especially with respect to the terminology used:

- Continuous systems
- Discrete-time equivalents of continuous systems
- Discrete-event equivalents of continuous systems

To form a basis for the rest of the discussion in this chapter, it is necessary to define a continuous system described by a set of differential equations. Mathematically, a continuous-time state-space model is represented as:

$$\dot{x}(t) = f(x(t), u(t), t); \quad x(t_0) = x_0 \tag{2.1}$$

where $x(t)$, $u(t)$, f and x_0 represent the continuous-time state, continuous-time input, continuous function, and initial value vectors respectively. The following sections discuss how discrete-time and discrete-event systems result from this continuous system, according to the use of the type of the plant-controller interface.

2.1 Discrete-time equivalents of continuous systems

Time is a continuous variable in the physical sense. However, most advanced computations require the help of digital computers. For example, some complex differential equations can only be solved using numerical methods. This necessitates the approach based on discretization of time. The design of a discrete-time control system for a continuous plant typically happens in the following steps:

- Finding a discrete-time equivalent of the continuous plant
- Designing a discrete-time controller for this discretized version of the plant

The time-based discretization typically happens through sampling. A *sampler* takes in a continuous-time function as an input and gives a discrete-time sequence as an output. This is also referred to as analogue-to-digital (A/D) conversion. Figure 2.1 shows a continuous-time function $f(t)$ being converted into a discrete-time sequence $f(k)$ using an analogue-to-digital converter. The samples may or may not be taken at evenly spaced intervals.

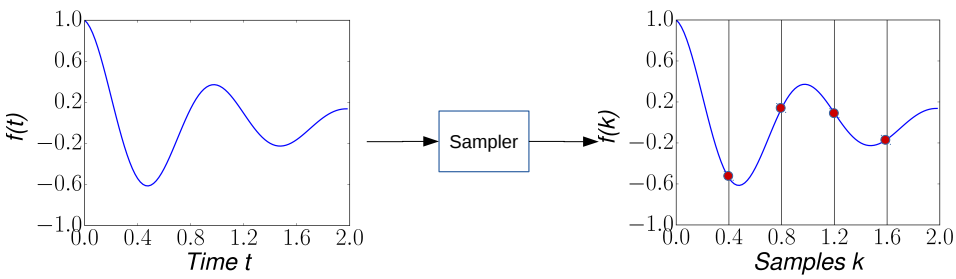


Figure 2.1: Sampler: Conversion of a continuous-time function into a discrete-time sequence.

In the sampler, the discrete-time sequence is represented using a finite number of bits. The precision of the converter thus depends on the number of bits used in the representation. For example, a 16-bit converter would mean a maximum sampling error of $2^{-16} = 0.0015\%$. This error is found to be acceptable for most control system applications (Santina et al., 2010).

The controller takes the discrete-time sequence as an input, takes necessary control action and returns a discrete-time sequence as an output. The returned sequence needs to be reconstructed into a continuous-time signal for the plant. This is known as digital-to-analogue (D/A) conversion and typically happens through a *hold* of some order. For example, a zero-order hold produces a stepwise signal from incoming samples. The binary data is converted into a voltage and the voltage is held constant until the next sample is available. This is represented in Figure 2.2.

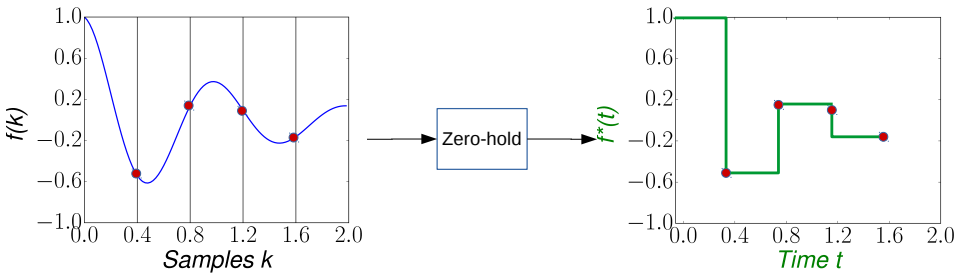


Figure 2.2: Zero-hold: Conversion of discrete-time sequence into an analogue signal.

It is apparent that the zero-order hold will have better approximation as the frequency of sampling increases. Another way to improve the accuracy of the approximation is to use higher-order holds. A hold of order n will construct a n-degree polynomial in each interval.

The design of the discrete-time controller is based on this discrete-time equivalent of the plant. The continuous plant and the interface can be described together as a discrete-time system. The obtained discrete-time system is then modelled as the discrete-time equivalent of the plant, as shown by the shaded region in Figure 2.3.

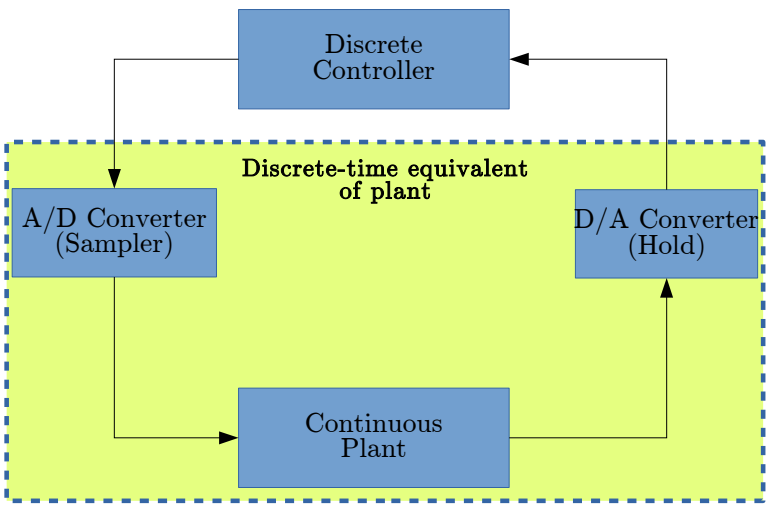


Figure 2.3: Discrete-time equivalent of a continuous plant.

The discrete-time representation (Figure 2.2) of the state-space model is then:

$$x(k+1) = f(x(k), u(k), k); \quad x(0) = x_0 \tag{2.2}$$

where $x(k)$ and $u(k)$ are the state and input vectors at sample k. Thus, one goes from differential equations (Equation 2.1) to difference equations (Equation 2.2).

An important point to note here is that discretization of time does not result in the discretization of the state-space, since the state can still take any real value. In other words, the state-space of the system is described by a continuous set. This is one of the main points where discrete-time systems differ from discrete-event systems, described in the next section.

2.2 Discrete-event equivalents of continuous systems

A system is called a discrete-state system when the state-space of a system may be modelled as a discrete set, taking only specific values. The transition from one state to another happens at discrete points in time. These transitions are referred to as *events* and are instantaneous in nature. It can be noted that the set of all possible events, i.e the set of all possible transitions in the state space, is a discrete set itself. In common notation, an event is denoted by e and the discrete-set of all possible events is denoted by E (Cassandras and Lafortune, 2010).

In such a discrete-state system, time may only be noted when an event $e \in E$ occurs. However, the state transitions themselves do not depend on the evolving time variable. This is to say that the events are asynchronous in nature. Such discrete-state systems with event-driven dynamics are more commonly referred to as discrete-event dynamic systems (DEDS) and these are the focus of this work.

To be precise, a discrete-event equivalent of a continuous plant results when the sampler discussed in Section 2.1 is replaced with a *quantizer* (Lunze, 1994, 1999, 2000). This means that the continuous plant is observed with *event sensors* which emit a signal only when a process variable crosses a certain value, as opposed to at constant time intervals. This is illustrated in Figure 2.4. For discrete-controller design purposes, the DEDS describing the plant and the interface together is modelled as a discrete-event equivalent of the plant (Figure 2.5). This is similar to how it is done for discrete-time systems.

Modelling of DEDS is not as straightforward as modelling the continuous-time system (Equation 2.1) or the discrete-time system (Equation 2.2). Like mentioned previously, the state-space in the DEDS is described by a discrete set. Each event sensor is represented by a 'boundary' in the state-space. The quantizer is thus represented by a set of boundaries in the state-space.

From the perspective of modelling an system with an n-dimensional state, the resulting 'quantized' state-space consists of n-dimensional *hypercubes*. An event

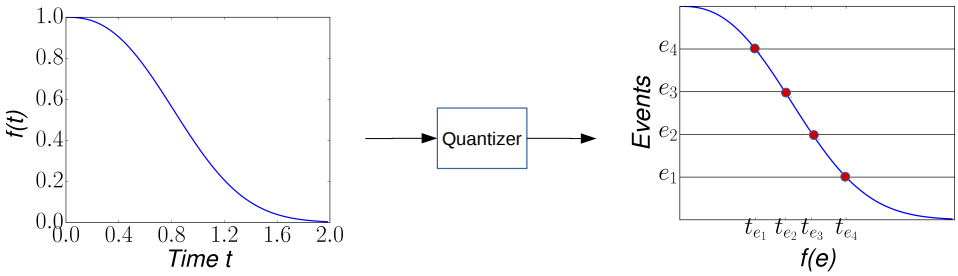


Figure 2.4: Quantizer: Conversion of a continuous-time function into a discrete-event sequence.

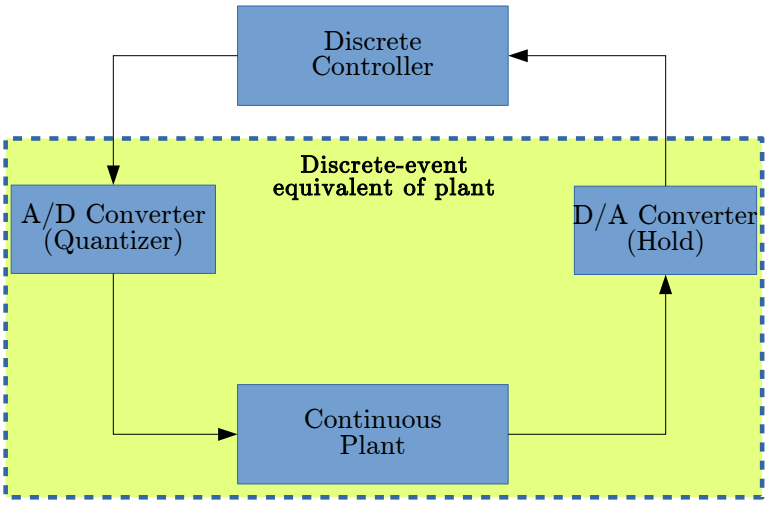


Figure 2.5: Discrete-event equivalent of a continuous plant.

in this context is defined as the state crossing the surface of the hypercube, or equivalently, making a transition from one hypercube to another. Since the system is event-driven, the only available information is the current hypercube that the state lies in. A continuous path of the state in a continuous-time system is translated into a discrete set of transitions between hypercubes in a discretized state-space. A representative example is shown in Figure 2.6. The notion of events inducing hypercubes in the state-space and the modelling of DEDS is discussed more formally in Chapter 3.

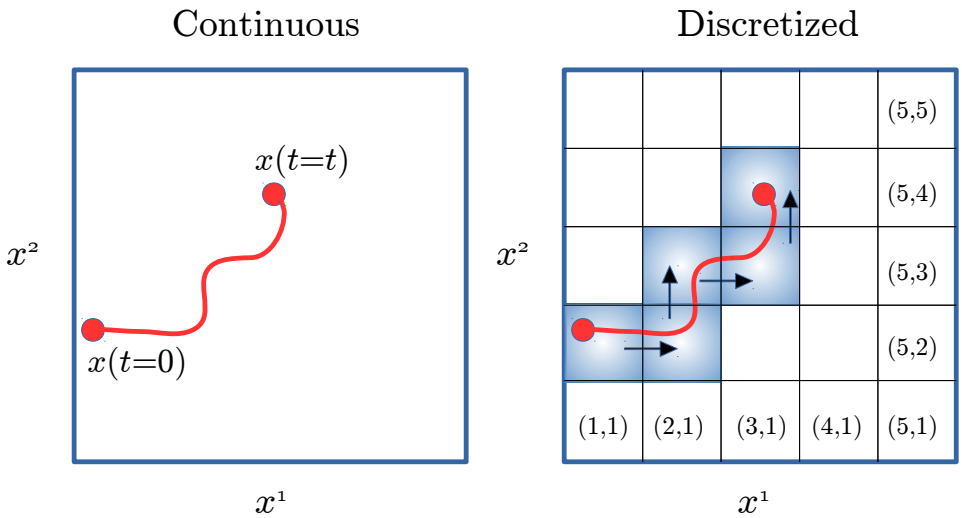


Figure 2.6: Trajectories in continuous (red path) and discrete (black arrows) state-spaces. Since the state-space is two-dimensional (x^1 and x^2 denote components of the state x), the hypercubes are rectangles. The grid lines indicate the boundaries induced by the event sensors.

2.3 Practical implementation of control in a plant

Practical implementation of DEDS involves plantwide control (regulatory as well as supervisory control). The system includes both time sampling and event sampling, and in sequence. The discrete-time sequence is used for computer implementation of control laws on the process level (regulatory control). The discrete-event sequence is used for supervisory control such that the event detection happens on a discrete-time signal coming from the discrete-time equivalent of the plant on the process level (rather than on a continuous-time signal). This is illustrated in Figure 2.7.

This necessitates that the time sampling of the continuous plant signal is 'fast enough' for event detection. This is because if two consecutive time samples are taken before and after an event, information about that event is lost. This would obviously lead to errors in the DEDS model. For the purposes of this thesis, however, the focus will be on the interaction between the supervisory controller - also referred to as the *supervisor* - and the discrete-event equivalent of the plant. The underlying interaction between the regulatory controller and the plant at the lower, process level is not addressed explicitly in the discussions pertaining to the topic of this thesis.

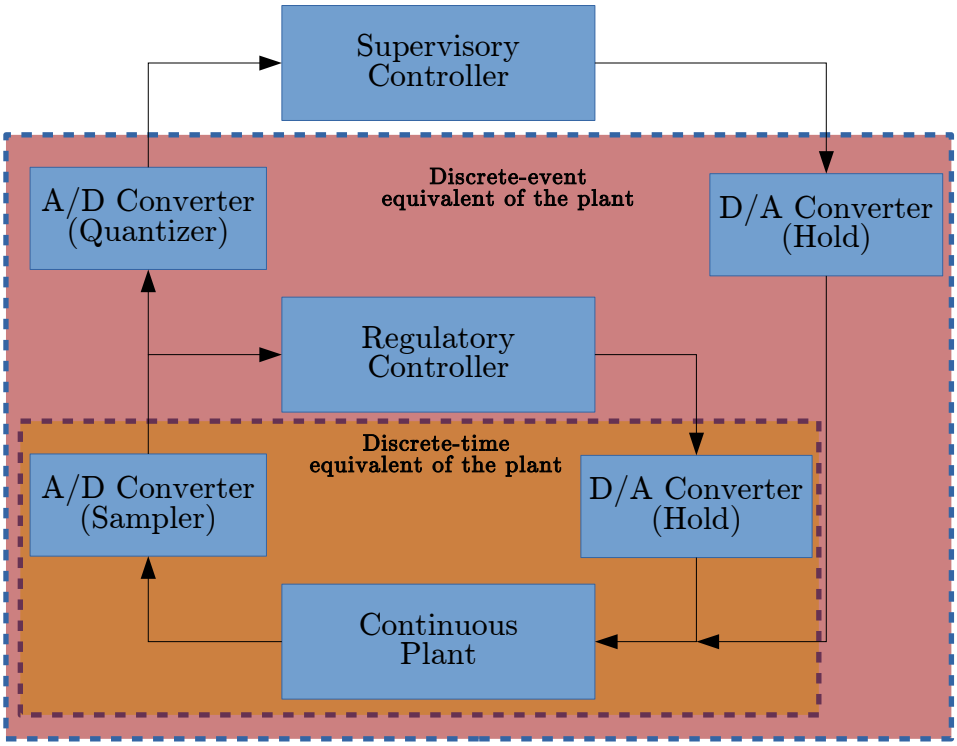


Figure 2.7: Practical implementation of a control system.

Chapter 3

Modelling of Discrete-Event Dynamic Systems

As mentioned in Chapter 2, DEDS are discrete-state systems with event-driven dynamics. It was further noted that in DEDS, time may only be noted when an event $e \in E$ occurs, where E is the set of all possible events. When this is done, a timed sequence of events: $(e_1, t_{e_1}), (e_2, t_{e_2}) \ldots (e_l, t_{e_l})$ is obtained. This means that the event e_1 occurs at time $t = t_{e_1}$, and so on. Thus, it is possible to obtain the state of the system at any point in time if the times sequence of events is available. In this context, the set of all possible timed sequences of events is known as the *timed language* model of the system (Cassandras and Lafortune, 2010) [1].

If, however, the information pertaining to the time occurrence of events is removed from this model, an *untimed language* model of the system is obtained. This model only contains information about the possible orderings in which events could occur in a system. For the above given timed sequence of events, the corresponding untimed sequence of events would be: $e_1, e_2 \ldots e_l$. Untimed language models are also referred to as *logical* models or simply *language* models.

As was discussed in Chapter 2, issues such as start-up, shut-down and operating within safety limits are of concern at the supervisor level of a plant. These issues relate to the logical behaviour of the system, satisfying a given set of specifications in the plant. In other words, the specific ordering of the events is of particular interest, rather than the exact timing of the events. Thus it is sufficient to model only the untimed behaviour i.e. to consider the logical model of the system.

[1] A spoken language consists of an alphabet which contains letters used to make words. Similarly, the behaviour of a DEDS model can be represented as a 'language' where the set of events E is an 'alphabet', the individual events $e \in E$ can be thought of as 'letters' of this alphabet and the finite sequences of these events can be thought of as 'words' describing the behaviour.

It is thus clear that a model representing an evolving DEDS should account for the different events that take place and also the sequence of these events. For example, the untimed sequence of events: $e_1, e_2 \ldots e_l$ approximates a 'path' in the state-space. The most popular logical models for DEDS are automata (Hopcroft et al., 2006) and Petri nets (Reisig, 1985). The former alternative is employed in this work.

This chapter first gives a brief introduction to automata theory, describing how automata are defined and laying out the basics needed to discuss these DEDS models in general. It may be recalled from Chapter 2 that the objective here is to model a discrete-event equivalent for the continuous plant (Figure 2.5). It is worth mentioning here that automata are also extensively used when describing *purely* discrete-event systems. The discussion in Section 3.1 pertains to these systems as well. Typical examples of purely discrete systems are queuing systems, communication systems and computer systems.

The chapter then proceeds to describe *how* to obtain a DEDS model of a continuous system. More specifically, the methodology used to derive an automaton from the given set of differential equations describing a continuous system (plant), is shown formally. A brief discussion regarding the computational cost of obtaining the automaton, and other relevant issues, is also included.

3.1 Automata theory

This section serves only as a brief introduction to the theory behind automata and some types of it. Automata theory has been studied extensively in the domain of computer science (Hopcroft et al., 2006). It also has various systems theory applications like supervisory control (Lin and Wonham, 1988b; Skoldstam et al., 2007; Tousi et al., 2008), fault diagnosis (Bouyer et al., 2005; Chang and Chen, 2011; Tripakis, 2002; Xi et al., 2001) and HAZOP analysis (Preisig and Manenti, 2012; Srinivasan and Venkatasubramanian, 1996).

Simply stated, an automaton is a model that performs computations by moving through a predetermined sequence of configurations (or states). In an automaton, the next state in the sequence is determined by the current state and an associated transition function. If the automaton takes in inputs (as in case of a Mealy automaton, discussed later in this section), the transition function incorporates the inputs in addition to the states to determine the next states in the sequence, as shown in Figure 3.1.

Automata may be deterministic or non-deterministic; and with or without inputs and outputs. The different types are defined below.

Figure 3.1: Illustration of transition function of an automaton with inputs.

3.1.1 Deterministic automata

Definition 3.1. A deterministic automaton G_d is a four-tuple

$$G_d = (\widetilde{X}, E, \phi, \tilde{x}_0)$$

where:

- $\widetilde{X}$ is the set of discrete states
- E is the finite set of events associated with G_d
- $\phi : \widetilde{X} \times E \to \widetilde{X}$ is the partial transition function: $\phi(\tilde{x}, e) = \tilde{y}$ implies that event e leads to the transition of the system from state $\tilde{x}$ to state $\tilde{y}$
- $\tilde{x}_0$ is the initial state of the system

The automaton G_d starts at $\tilde{x}_0$, the initial state of the system. When an event $e \in E$ occurs, it makes a transition to the state $\phi(\tilde{x}_0, e) \in \widetilde{X}$. Further transitions follow according to the transition function ϕ. The following example shows a deterministic automaton.

Consider the event set $E = \{e_1, e_2\}$ and the state set $\widetilde{X} = \{\tilde{x}_1, \tilde{x}_2, \tilde{x}_3, \tilde{x}_4\}$. Further consider the following transition function ϕ:

$$\begin{aligned} \phi(\tilde{x}_1, e_1) &= \tilde{x}_2 & \qquad \phi(\tilde{x}_1, e_2) &= \tilde{x}_3 \\ \phi(\tilde{x}_2, e_1) &= \tilde{x}_4 & \qquad \phi(\tilde{x}_2, e_2) &= \tilde{x}_3 \\ \phi(\tilde{x}_3, e_1) &= \tilde{x}_3 & \qquad \phi(\tilde{x}_4, e_2) &= \tilde{x}_1 \end{aligned}$$

This automaton can be represented as a directed graph where the states are the nodes and the events are the arcs, as shown in Figure 3.2. The automaton is deterministic because a state cannot make transitions to multiple states for the same event. This implies that for a state, a specific event results only in a specific transition. Thus the transition function ϕ maps $\widetilde{X} \times E \to \widetilde{X}$, as can be seen from the given example.

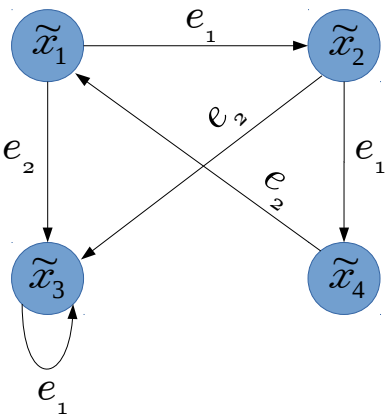


Figure 3.2: Directed graph of the deterministic automaton

3.1.2 Non-deterministic automata

In a non-deterministic automaton, an event e at state $\tilde{x}$ may cause transitions to multiple states. In other words, $\phi(\tilde{x}, e)$ does not just represent a single state but a set of states. Further, it may be the case that the initial state of the automaton is itself be a set of states.

Definition 3.2. A non-deterministic automaton G_{nd} is a four-tuple

$$G_{nd} = (\widetilde{X}, E, \phi, \tilde{x}_0)$$

where:

- $\widetilde{X}$ is the set of discrete states
- E is the finite set of events associated with G_{nd}
- $\phi : \widetilde{X} \times E \to 2^{\widetilde{X}}$ is the partial transition function such that $\phi(\tilde{x}, e) \subseteq \widetilde{X}$ whenever it is defined
- $\tilde{x}_0$ is the initial state of the system, which may be a set of states i.e. $\tilde{x}_0 \subseteq \widetilde{X}$

Consider the following transition function for the same sets E and $\widetilde{X}$ defined in Section 3.1.1:

$$\begin{aligned}
\phi(\tilde{x}_1, e_1) &= \{\tilde{x}_2, \tilde{x}_3\} & \phi(\tilde{x}_1, e_2) &= \tilde{x}_3 \\
\phi(\tilde{x}_2, e_1) &= \tilde{x}_4 & \phi(\tilde{x}_2, e_2) &= \tilde{x}_3 \\
\phi(\tilde{x}_3, e_1) &= \tilde{x}_3 & \phi(\tilde{x}_4, e_2) &= \{\tilde{x}_1, \tilde{x}_3\}
\end{aligned}$$

Here, the transition function maps from $\widetilde{X} \times E \to 2^{\widetilde{X}}$. The state transition diagram is shown in Figure 3.3.

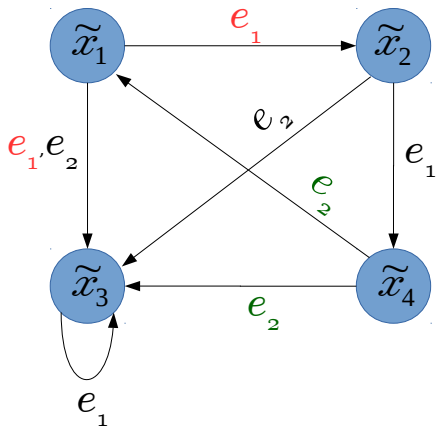


Figure 3.3: Directed graph of a non-deterministic automaton. The events marked in red and green respectively show transitions from states $\tilde{x}_1$ and $\tilde{x}_4$ to multiple states for the same event.

3.1.3 Automata with inputs and outputs

Automata may also be classified based on inputs and outputs of the system. The two classes of automata based on this classification are the Moore automata and the Mealy automata (Cassandras and Lafortune, 2010).

- In Moore automata, the output of the automaton depends only on the current state of the automaton. Each state is associated with a corresponding output via an output function. This means that the automaton 'gives out' the corresponding output when some state is reached.

- In Mealy automata, the output of the automaton depends on the current state as well as the input to the automaton. Inputs and outputs can be thought of in terms of 'input events' and 'output events'. This means that if the automaton is in a particular state $\tilde{x}$ and receives an input event e_i, it makes a transition to state $\tilde{y}$ according to the transition function and an output event e_o corresponding to this $\tilde{x} \rightarrow \tilde{y}$ transition is then 'given out' in the process, according to the output function.

Viewed in the context of inputs and outputs, a non-deterministic automaton is defined as shown next.

Definition 3.3. A non-deterministic input/output automaton G is a six-tuple

$$G = (\widetilde{X}, \widetilde{U}, \phi, \widetilde{Y}, h, \tilde{x}_0)$$

where:

- $\widetilde{X}$ is the set of discrete states
- $\widetilde{U}$ is the set of discrete inputs (input events)
- $\phi : \widetilde{X} \times \widetilde{U} \to 2^{\widetilde{X}}$ is the transition function
- $\widetilde{Y}$ is the set of discrete outputs (output events)
- $h : \widetilde{X} \times \widetilde{U} \to \widetilde{Y}$ is the output function
- $\tilde{x}_0$ is the initial state of the system

If the output $\tilde{y} \in \widetilde{Y}$ is only a function of the state $\tilde{x} \in \widetilde{X}$, i.e. if $\tilde{y} = h(\tilde{x})$, then G is a Moore automaton. If it is a function of the state as well as the input $\tilde{u} \in \widetilde{U}$, i.e. if $\tilde{y} = h(\tilde{x}, \tilde{u})$, then G is a Mealy automaton. The typical way of representing such automata is as shown in Table 3.1.

	Input events ...
Current States	Next states (and associated output events) ...
⋮	⋮
⋮	⋮

Table 3.1: Typical tabular automaton representation.

3.2 Obtaining the discrete-event model of a continuous system

A continuous system (plant) is represented by a set of differential equations. This section discusses a method to obtain an automaton from this set of differential equations. Consider the following set of differential equations [2]:

$$\dot{x}(t) = f(x(t), u(t)); \quad x(t_0) = x_0 \tag{3.1}$$

where $x(t) \in \mathbb{R}^n$, $u(t) \in \mathbb{R}^m$, $f : \mathbb{R}^{n+m} \to \mathbb{R}^n$ and x_0 is the initial state vector. This continuous plant has inputs but no outputs. This is to say that an explicit mapping from the state to output is not considered in this discussion [3]. To develop a DEDS model for this system, it is necessary to correspondingly define the set of discrete sets $\widetilde{X}$,the set of discrete inputs $\widetilde{U}$ and the transition function ϕ.

3.2.1 State discretization

The concept of a quantizer (event sensors) leading to formation of so-called *hypercubes* in the state-space was briefly mentioned in Section 2.2. Consider the state $x = (x^1 \quad x^2 \quad ... \quad x^n)$. The boundaries β^i for each state component x^i, induced by the event sensors, can be represented as:

$$\beta_0^i \leq \beta_1^i \leq \beta_2^i \ldots \leq \beta_{p_i}^i \quad (p_i \geq 1) \tag{3.2}$$

The region of interest is then determined by the state-space represented by:

$$\Omega = \{x \in \mathbb{R}^n | \beta_0^i \leq x^i \leq \beta_{p_i}^i, i = 1, 2, \ldots, n\} \tag{3.3}$$

The state-space Ω can be thought of as being partitioned into n-dimensional hypercubes by these boundaries. Each hypercube thus represents a discrete state $\tilde{x}$. Let $\tilde{x}$ be represented by a n-dimensional hypercube that is labelled by an n-tuple of integers $a = (a^1, a^2, \ldots, a^n)$ with $1 \leq a^i \leq p_i$, for each i. Then the hypercube $H_x(\tilde{x})$ is the bounded region given by (Philips, 2001):

$$H_x(\tilde{x}) := \{x \in \mathbb{R}^n | \beta_{a^i-1}^i \leq x^i \leq \beta_{a^i}^i, i = 1, 2, \ldots, n\} \tag{3.4}$$

[2] This is a time-invariant case of Equation 2.1. In a *time-invariant* dynamic system, the same input always produces the same output. Here, it implies that f does not explicitly depend on time.

[3] Obtaining the DEDS model for the plant with outputs is more involved and is covered in Ushio and Takai (2009) and Philips (2001).

Put simply, this shows how the 'location' of the hypercube is identified in the n-dimensional state-space Ω, given the n-tuple of integers representing $\tilde{x}$. It can be seen that the total number of hypercubes in Ω is given by the product $\prod_i p_i$ The simplest example of a 2-dimensional state-space having four hypercubes is shown in Figure 3.4.

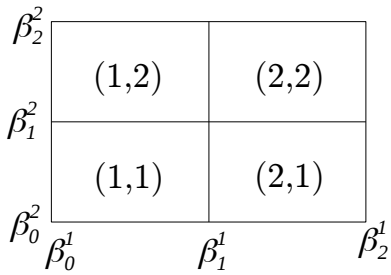


Figure 3.4: Hypercube labels and boundaries in a state-space

Further, if the hypercubes $H_x(\tilde{x}_1)$ and $H_x(\tilde{x}_2)$ share a boundary hypersurface $H_x(\tilde{x}_1) \cap H_x(\tilde{x}_2)$ that is of $(n-1)$ dimension, then the corresponding discrete states $\tilde{x}_1$ and $\tilde{x}_2$ are said to be *adjacent* to each other. The transition from one hypercube to other across the boundary hypersurface is called as an *event* and is denoted by $\tilde{x}_1 \rightarrow \tilde{x}_2$ [4].

3.2.2 Input discretization

An analogous argument to state discretization can be made in case of discretization of continuous inputs. Consider the input $u = (u^1 \quad u^2 \quad ... \quad u^m)$. The boundaries γ^i for each input component u^i, can be represented as:

$$\gamma_0^i \leq \gamma_1^i \leq \gamma_2^i \ldots \leq \gamma_{q_i}^i \quad (q_i \geq 1) \tag{3.5}$$

The input-space is then made of m-dimensional hypercubes. Each hypercube thus represents a discrete input $\tilde{u}$. Let $\tilde{u}$ be represented by a m-dimensional hypercube that is labelled by an m-tuple of integers $b = (b^1, b^2, \ldots, b^m)$ with $1 \leq b^i \leq q_i$, for each i. Then the hypercube $H_u(\tilde{u})$ is the bounded region given by:

$$H_u(\tilde{u}) := \{u \in \mathbb{R}^m | \gamma_{b^i-1}^i \leq u^i \leq \gamma_{b^i}^i, i = 1, 2, \ldots, m\} \tag{3.6}$$

[4] It is assumed here that only transitions between adjacent hypercubes are allowed. This means that the state trajectory is not allowed to cross the edges or corners of the hypercubes. This is a limitation of the model and will be elaborated on in Section 3.3.3.

The total number of hypercubes in the input-space is given by the product $\prod_i q_i$. The analysis then involves studying how the discretized inputs $\tilde{u}$ cause transitions between the discretized states $\tilde{x}$.

However, for the sake of this discussion (and the report), a simplification is made. It is assumed that the set of inputs is *already* discrete by default and input discretization is not needed. Equivalently, it may be assumed that the set of inputs U is continuous but piecewise constant such that each element in U is mapped directly onto one of the discrete inputs in $\widetilde{U}$. This means that $\widetilde{U} = \{u_1, u_2, \ldots, u_k\}$ [5].

This is a reasonable assumption to make especially in case of process plants where inputs are mainly discrete valve positions (open/closed). The reason for this assumption is that it simplifies the discussion to a visualization of transitions in the state-space for a given discrete input. It helps in avoiding the *simultaneous* visualization of both the state-space and the input-space, or the *combined* visualization of the whole state-input-space, significantly condensing the analysis. The arguments that follow, however, hold in the general even if this assumption is relaxed.

The discrete-event model of the continuous system should describe all possible transitions (events) between hypercubes in the state-space for given discrete inputs. It is clear from the discussion in Section 3.1 that, in this context, this model is a non-deterministic Mealy automaton. The non-deterministic nature comes from the fact that a hypercube has multiple adjacent hypercubes (in multiple dimensions) where transitions are possible for a given discrete input. Definition 3.3 will hold for this automaton, but without the inclusion of the set of discrete outputs $\widetilde{Y}$ and the output mapping h, since outputs are not explicitly considered.

3.2.3 Computing transition function

As discussed, a continuous trajectory in the continuous state-space corresponds to discrete transitions from one hypercube to the next in the discretized state-space (see Figure 2.6). This implies that the boundary surface between the two adjacent hypercubes will be crossed when a transition happens between adjacent discrete states. This fact can be exploited in the computation of the transition function (Philips, 2001; Preisig, 1996). The derivative of the state trajectory is given by the continuous model of the plant represented by Equation 3.1. This derivative is checked at boundary surfaces between two adjacent hypercubes. The sign of the derivative will then determine the direction of the state trajectory. If the derivative does not exist at the boundary, there will be no transition between the hypercubes.

[5] The 'tilde' notation generally used for discrete variables is not used here even though the inputs are assumed discrete. This is because they may equivalently be assumed to be continuous but piecewise constant.

Being more precise in mathematical terms (Philips, 2001), consider two adjacent states $\tilde{x}_1$ and $\tilde{x}_2$ according to the following:

- $\tilde{x}_1$ and $\tilde{x}_2$ are represented by hypercubes that are labelled by the n-tuples $(a^1, \ldots, a^r, \ldots, a^n)$ and $(a^1, \ldots, a^r + 1, \ldots, a^n)$, respectively. This is to say that the two hypercubes are adjacent in the rth dimension of the state-space.
- The boundary hypersurface between these two hypercubes is denoted by $H_x(\tilde{x}_1) \cap H_x(\tilde{x}_2)$. In this case, this is given by the locus of the points $\{x \in \mathbb{R}^n | x^r = \beta^r_{a^r}\}$.
- $x \in H_x(\tilde{x}_1) \implies x^r \leq \beta^r_{a^r}$ and $x \in H_x(\tilde{x}_2) \implies x^r \geq \beta^r_{a^r}$. This specifies the positions of the adjacent hypercubes relative to each other i.e. $H_x(\tilde{x}_1)$ comes 'before' $H_x(\tilde{x}_2)$, when moving in the positive direction in the rth dimension.

Also, let the rth element of the f vector in Equation 3.1 be denoted by f^r. Then the transition $\tilde{x}_1 \to \tilde{x}_2$ in the rth dimension is possible with a given discrete input $u \in \widetilde{U}$, if and only if

$$\exists x \in H_x(\tilde{x}_1) \cap H_x(\tilde{x}_2) \text{ such that } f^r(x, u) > 0 \tag{3.7}$$

The mathematical proof is not covered here but can be found in literature (Blanchini, 1999; Philips, 2001).

So, computing the transition function ϕ essentially consists of checking the value, or rather the sign, of trajectory derivative f^r at the boundary hypersurfaces between every pair of adjacent hypercubes. If the sign is positive, the transition is possible and if it's not positive, the transition is impossible. In fact, if the sign is negative, the *opposite* transition $\tilde{x}_2 \to \tilde{x}_1$ is possible.

3.2.4 Getting the final DEDS model

Now that $\widetilde{X}, \widetilde{U}$ and ϕ have been determined, the final model can be obtained. Going back to the assumption made in Section 3.2.2, the inputs are considered to take only discrete values. This makes checking of f^r at various boundaries relatively easier, demonstrated as follows.

For a given input $u \in \widetilde{U}$, $f^i(x, u) = \dot{x}^i = 0, i \in \{1, 2, \ldots, n\}$ represents a $(n-1)$-dimensional hypersurface in the state-space. Specifically it is the corresponding component equilibrium hypersurface, since x^i will be the same on all points on the hypersurface. The intersection of all the component equilibrium hypersurfaces, if it exists, will be the global equilibrium point of the system.

Now, the equilibrium hypersurface across the range of the state-space [6] will divide the state-space into two distinct subregions. The natural tendency of a system is to move towards equilibrium. Since in a dynamic system the state component x^i will always move towards the corresponding equilibrium hypersurface $f^i = 0$, it can be seen that these two subregions will have opposing trajectories for x^i. This is illustrated in Figure 3.5.

Since the direction of the state trajectory is now known in the entirety of the continuous state-space, knowing the transition in the event-discretized state space boils down to ascertaining whether the hypercube that the state lies in, is on one side of the equilibrium hypersurface or the other.

Note here that it is thus *not* necessary to check the sign of f^r at *every* boundary hypersurface, between each pair of adjacent hypercubes. One only needs to know the location of the hypercube with respect to the equilibrium hypersurface, and extrapolate that information to the hypercubes further away from the equilibrium hypersurface. This classifies the hypercubes in the event-discretized state space into three distinct types:

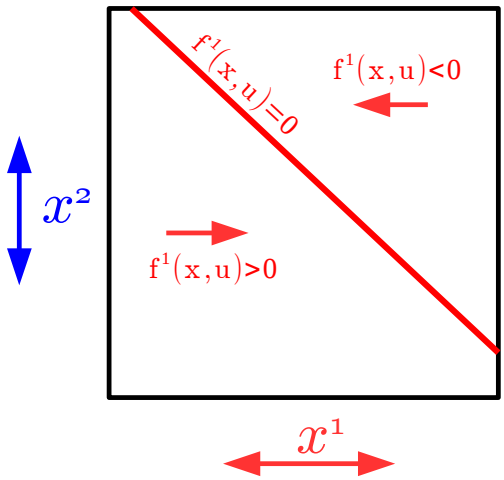
- Type 1 (T1): Hypercubes where transitions with respect to a state component are in the positive direction with respect to the corresponding component equilibrium hypersurface

- Type 2 (T2): Hypercubes where transitions with respect to a state component are in the negative direction with respect to the corresponding component equilibrium hypersurface

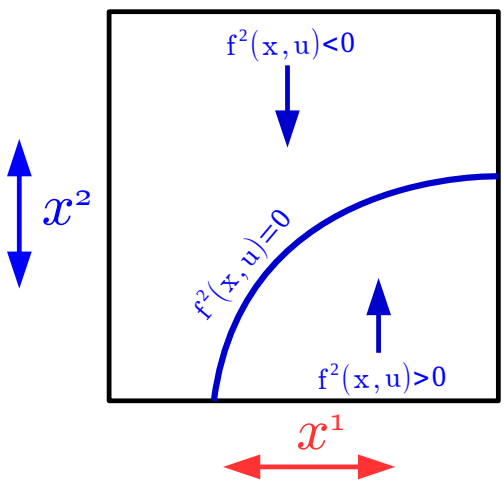
- Type 3 (T3): Hypercubes where transitions with respect to a state component cannot be ascertained

Type 3 corresponds to hypercubes through which the component equilibrium surface passes. Since the size of the hypercubes is the limit of the resolution that can be obtained in an discretized state-space, what happens within the hypercube is 'hidden' from the observer. Hence, the transitions in these Type 3 hypercubes cannot be ascertained i.e. it is not possible to know whether the transitions will be positive or negative - unless additional information is provided, for instance through local refinement (discussed in Section 3.3.2). The three types are illustrated in Figure 3.6.

[6] The range of the state-space will be defined by the physical limits of the system.

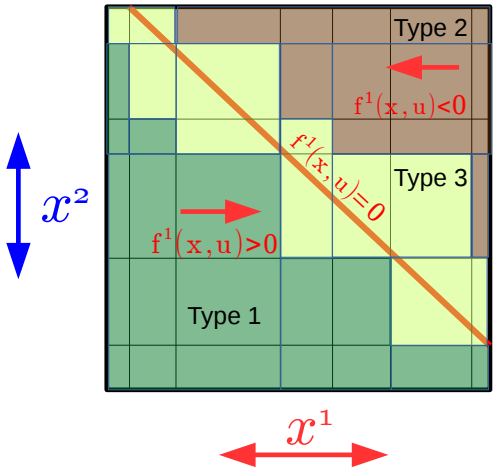


(a)

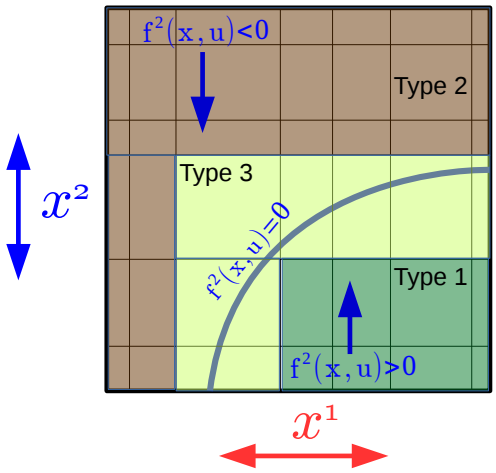


(b)

Figure 3.5: A two dimensional continuous state-space. The component equilibrium hypersurface (here a 2-D line/curve) divides the state-space into subregions of opposing gradients, with respect to (a) x^1 and (b) x^2.



(a)



(b)

Figure 3.6: A two dimensional discretized state-space. The three different colors represent the three types of hypercubes, based on the type of transitions for (a) x^1 and (b) x^2.

Combining the knowledge of transitions from all the state components and given a hypercube in the discretized state-space, one can determine exactly the list of the possible transitions from this hypercube that may happen - which determines the automaton that was sought.

The result here is a list of possible transitions and not an exact transition because each state component will try to drive the state in the corresponding dimension. It may be noted that this is precisely what gives the automaton its non-deterministic nature. This is illustrated in Figure 3.7.

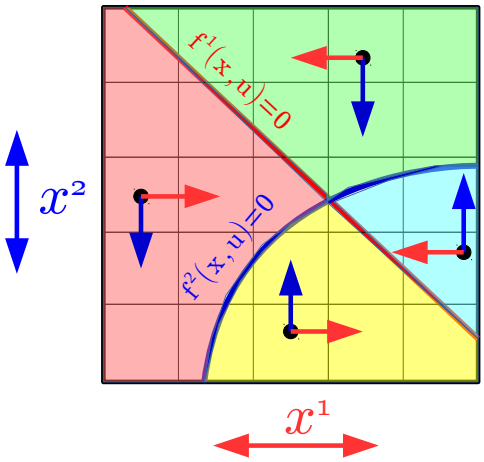


Figure 3.7: Possible transitions in a discretized state-space - non-deterministic nature.

3.3 Some pertinent issues

3.3.1 Computational effort

It may seem that with increasing state dimensionality and finer discretization, the computational cost blows up, since the number of hypercubes in the state-space then increases rapidly - the state-explosion problem. However, in practical applications with higher dimensionality, the interdependencies between the various state components are usually *sparse* - meaning that some components of the state are not influenced by all other components or inputs.

This sparsity is only natural because if one is to think of a complex plant in a modular way, only modules that are 'close' have an effect on each other. When computing the state transitions with respect to a particular state-component, this sparseness can be exploited to reduce the dimensionality of the corresponding sub-problem.

Further, since one is only interested in knowing whether a given hypercube lies on one side of a component equilibrium hypersurface or the other, one only needs to check this condition for those hypercubes that are close to the hypersurface and use this information to extrapolate to the hypercubes further away. This does away the need to check the sign of f^r at boundaries of each pair of adjacent hypercubes, significantly reducing the computational effort.

Moreover, it suffices to compute the intersection points of the equilibrium hypersurface with the various event boundaries since one is only interested in knowing which hypercubes the equilibrium hypersurface passes through and not its shape within the hypercube (this however assumes that the equilibrium hypersurface is monotonic within the hypercube). This implies that only root solvers are required. The computational cost for linear systems, especially, is significantly lesser compared to non-linear systems. A Python code for generating the automaton is given in Appendix B.

3.3.2 Selective finer discretization

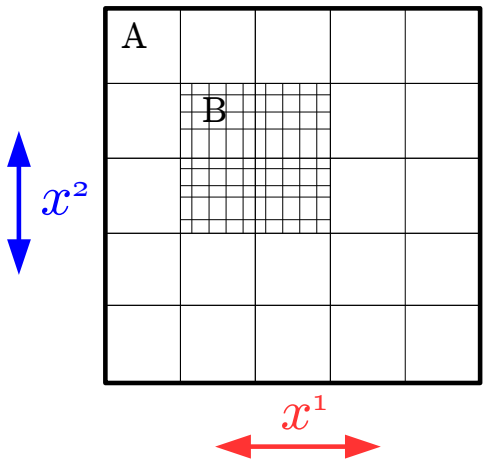


Figure 3.8: Selective finer discretization.

In some cases, a region within the state-space may be of particular interest in terms of the model analysis or control. That is, more information or accuracy may be sought from the model in this region. A reason for this may be that one needs much 'tighter' control in this region due to stricter specifications.

This higher accuracy can be achieved through finer discretization of the state-space (in the physical sense, this would mean higher number of event sensors in the system). As mentioned previously, the number of hypercubes in the state-space

determines the resolution of the DEDS model, since what happens in the interior of the hypercube is hidden from the observer. Finer discretization of the state-space implies higher number of hypercubes and thus higher resolution.

However, resorting to finer discretization in the *entire* state-space unnecessarily increases the computation cost of obtaining the automaton, since now the state component gradients (f^r) have to be checked at many more boundary hypersurfaces between hypercubes [7]. A better idea is to do the finer discretization in a *selective* manner. This means that a finer discretization is only sought within a hypercube or a group of hypercubes, but not in the entire state-space. The idea is illustrated in Figure 3.8. For instance, one can have finer discretization for the hypercubes that are closer to the equilibrium hypersurfaces, since this is a region where the hypercubes with uncertain transitions (Type 3) are encountered. In general, this can be done for any region in the state-space where more accuracy might be needed. This is also termed as local refinement (Philips, 2001).

The automaton for the finely discretized parts may then be computed separately and the supervisor can then 'switch' between automata as needed. For example, the supervisor can initiate the use of Automaton A in a coarse region A of the state-space and switch to Automaton B in another, finer, region B of the state-space. The switching mechanism may be extended to any finite number of automata. This is illustrated in Figure 3.9.

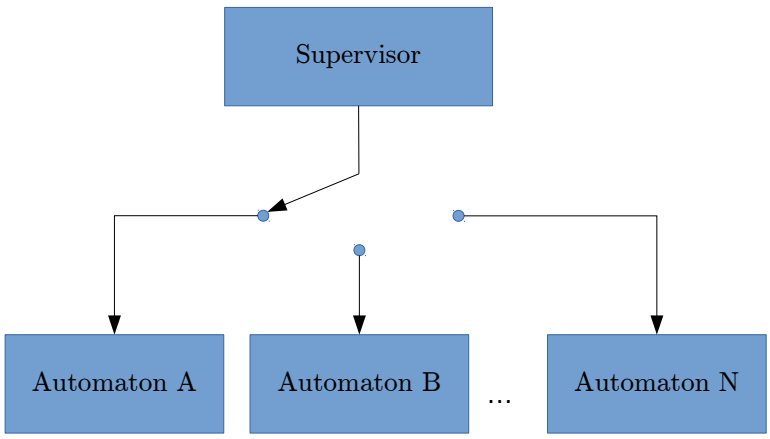


Figure 3.9: The supervisor can switch between multiple automata models as needed.

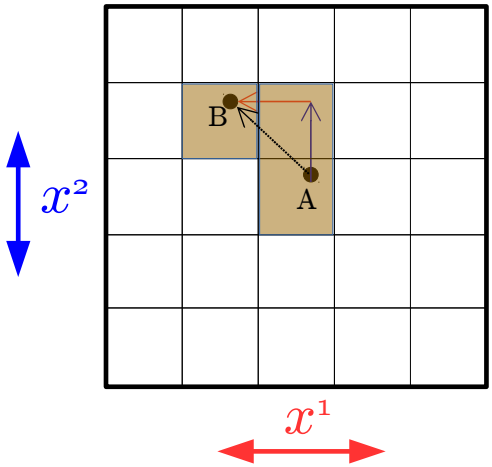
[7]The computational cost would not be *that much* higher, however, since the sparsity argument from Section 3.3.1 is still valid.

3.3.3 State trajectory passing through a corner of hypercube

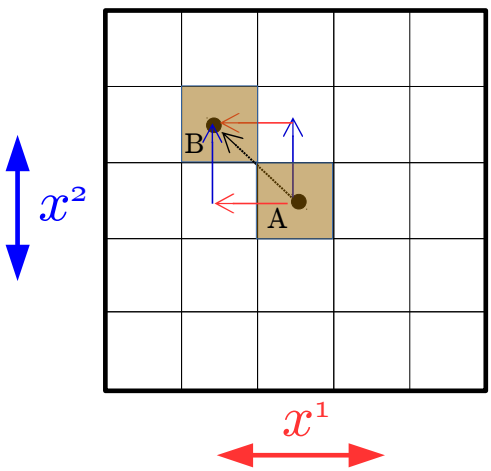
A limitation of this modelling approach can be seen when the continuous state trajectory passes through a corner or an edge of the hypercube. For illustration, compare the continuous trajectory from point A to point B (shown in black) in Figures 3.10(a) and 3.10(b). When the continuous trajectory does not pass through a hypercube corner, it can be described by a particular sequence of discrete transitions (upwards and then leftwards as shown in Figure 3.10(a)). However, when it passes through a corner, there is no definite way to represent this as a fixed sequence of discrete transitions (it could be upwards and then leftwards, or leftwards and then upwards, as shown in Figure 3.10(b)).

This is a major limitation, especially if it happens in a region of interest. Since the supervisory control action is determined by the sequence of transitions given by the automaton, choosing one or the other sequence from Figure 3.10(b) may lead to completely different control actions. This might also have implications in safety analyses of plants where incorrect or ambiguous information may have serious consequences.

In view of this limitation, it will be assumed for all analyses pertaining to this report that the trajectory does not pass through a point that belongs to more than two hypercubes. This means that two different components x^i and x^j $(i \neq j)$ of a state cannot both cross a hypercube boundary simultaneously. Stated differently, it is assumed that only transitions between adjacent hypercubes are allowed.



(a)



(b)

Figure 3.10: The continuous state trajectory (black) and the corresponding hypercube transitions (red,blue) when (a) when trajectory does not pass through a hypercube corner and (b) when it does.

Chapter 4

Using DEDS in HAZOP Analysis: A Case Study

A HAZOP analysis deals with not only the safety issues in a process plant, but also with it's operability issues. Although operability issues may not lead to hazardous or unsafe conditions in the plant, they may lead to poorer economic performance. Thus it becomes necessary to define a *safe operability region* for the plant in performing a HAZOP analysis. The choice of the safe operability region is a decision that the team of engineers performing HAZOP have to make. The process plant already has a range imposed by the physical limits that the process variables may take, for example the mole fractions may only be between 0 and 1. The safe operability region is a subset of the entire range spanned by the process plant. This is because operating at a physically allowable point does not necessarily mean that such an operation is safe or economically viable. For example, some equipment may be designed to operate within a certain specified range - a very low flow rate is not desirable for a compressor.

4.1 The idea

Consider the continuous state-space of the kind that was discussed in Chapters 2 and 3. The state-space Ω, as defined in Section 3.2.1 represents the region of interest in the process plant. In this context, the safe operability region, denoted by Ψ, is a subregion within this state-space Ω (Figure 4.1).

The boundaries of Φ with respect to each state component x^i can be represented by λ^i_{min} and λ^i_{max}, such that $\lambda^i_{min} < \lambda^i_{max}$ for each i. These boundaries are decided by the team of engineers like mentioned previously. Then,

$$\Psi = \{x \in \mathbb{R}^n | \lambda^i_{min} \leq x^i \leq \lambda^i_{max}, i = 1, 2, \ldots, n\} \tag{4.1}$$

Note that Ψ is a hypercube.

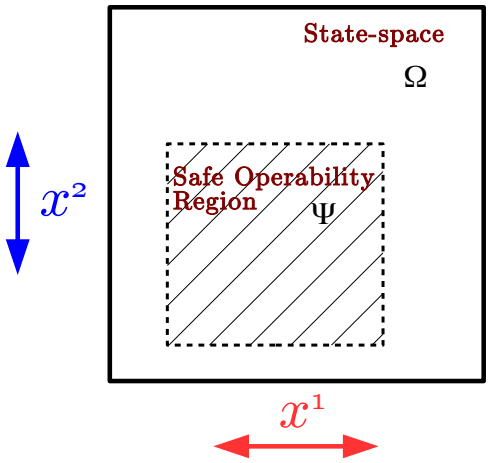


Figure 4.1: Safe operability region ψ within a state-space Ω.

The use of event sensors induces hypercubes in the state-space. This discretized state-space is then superimposed with the Ψ. The DEDS model ,say an automaton G, of the continuous plant is obtained according to the methodology described in Chapter 3. Now, equipped with the automaton G and the defined safe operability region Ψ, one can assess the possibilities of the system breaching the boundaries of Ψ. This is because the automaton G contains information about the *directionality* of the state trajectory throughout the state-space whereas the safe region Ψ contains information about the *boundaries* within the state-space where the process is considered 'safe.

If the state reaches a hypercube that is in the vicinity of the boundary of the Ψ and if for that hypercube the automaton G shows a transition that points outwards of Ψ, one can ascertain that there is a possibility of a so-called *leak* happening (Preisig and Manenti, 2012). A leak is possible state transition out of the safe operability region. A leak in this context is defined as follows:

Definition 4.1. Consider the continuous system given by Equation 3.1 and a defined safe operability region Ψ. Further, let the boundary surface of Ψ be a continuous set denoted by $bd(\Psi)$. A leak in the rth dimension of Ψ is defined as part of $bd(\Psi)$ where, for a given input u and any state $x \in bd(\Psi)$, $f^r(x, u)$ has a sign (positive or negative) that implies a state trajectory out of Ψ.

However, it is not necessary to check sign of f^r on the entire boundary $bd(\Psi)$, because this information is readily available if the automaton G has been computed. One only needs to know which hypercubes the boundary $bd(\Psi)$ passes through and the automaton G would provide the transition information for those hypercubes. If the transitions are in outward direction with respect to Ψ, then the corresponding part of the boundary $bd(\Psi)$ is a leak hypersurface. Moreover, if this outward transition happens, then a leak is said to have occurred. It may be seen that the input u that causes a leak to occur is 'undesirable' - a notion that will be expanded upon in the discussion of supervisory control in Chapter 5. The concept of leaks is illustrated in Figure 4.2.

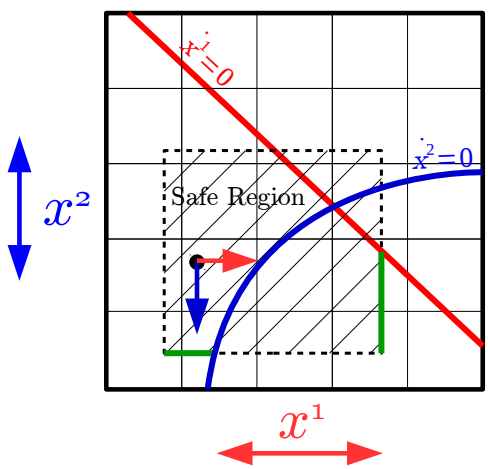


Figure 4.2: Leaks in the safe operability region, highlighted by bold green lines.

This technique has a major advantage over the conventional HAZOP in that it can handle complex plants with numerous process variables as well, provided the automaton is computed with respect to all relevant variables. It has the potential to identify failures or hazards caused by multiple simultaneous failures since all possible combinations of (discretized) process variable values are explicitly enumerated in the automaton. Obviously, the higher the level of discretization of the

state-space (implying more event sensors), the lower the variance of the measured process variable value. The technique thus gives a quantitative basis for doing HAZOP, as opposed to the conventional HAZOP that relies considerably on human judgement. A summary of the conventional HAZOP procedure is given in Appendix A.

4.2 Case study: Two Tanks

Consider a system of two interacting tanks connected in series, as shown in Figure 4.3. This configuration is common in the process industry where the second tank is a buffer tank to dampen disturbances in the inflow rate of the first tank.

The valve on the inlet of the first tank (Valve 0) has a valve constant of C_0 $(m^{2.5}/min)$. The flow rate into Tank 1 when this valve is open is Q (m^3/min). The heights of liquids in the two tanks are H_1 and H_2 (m) respectively. Both tanks have a base area of A (m^2). The valves on the outlets of the two tanks (Valve 1 and Valve 2) have valve constants of C_1 and C_2 $(m^{2.5}/min)$ respectively. The 3 valve positions are denoted by W_0, W_1 and W_2 respectively. Seven level sensors each are placed in both tanks.

The numerical values for the constants in the model are given in Table 4.1. The locations of the level sensors are: 0 m, 1.5 m, 3 m, 4.5 m, 6 m, 7.5 m, 9 m for Tank 1, and 0 m, 1.6 m, 3.1 m, 4.6 m, 6 m, 7.5 m and 9 m for Tank 2 [1].

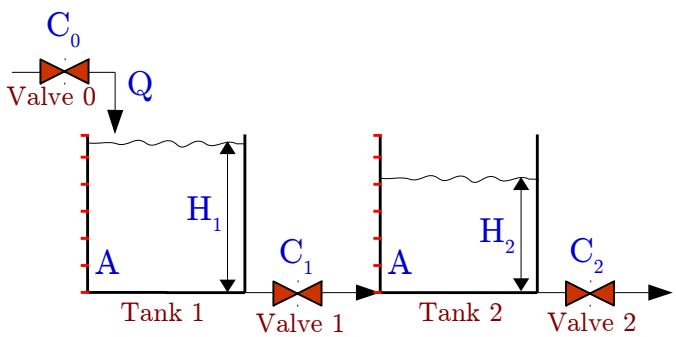


Figure 4.3: System of two tanks connected in series.

[1] Some of these values are chosen to avoid violating the non-corner-crossing assumption made in Section 3.3.3.

Constants	Value	Units
Q	0.3	m^3/min
C_0	0.15	$m^{2.5}/min$
C_1	0.15	$m^{2.5}/min$
C_2	0.15	$m^{2.5}/min$
A	0.5	m^2

Table 4.1: Constants in the two tanks model.

4.2.1 Model Derivation

The objective here is to get a simple model for the two tanks system in the continuous state-space representation (Equation 3.1). The states, inputs and constants in the model are determined. The states in the model are the heights H_1 and H_2. Thus the continuous state vector is $x = (H_1\ H_2)$.

The inputs in the model are determined by the positions of the 3 valves. The valve positions are components of the input vector and are considered to be binary - open or closed. This is to say that the input vector $u = (W_0\ W_1\ W_2)$ [2], where each of the valve positions W_0, W_1 and W_2 is either 0 (closed) or 1 (open). For example the input $u = (1\ 0\ 0)$ represents the case where the Tank 1 inlet valve is open but both outlet valves are closed. It may be seen that in this case there are a total of $2^3 = 8$ different discrete inputs.

The inflow rate is Q when the valve position $W_0 = 1$. The constants in the model are thus the inflow rate Q, the base areas of the tanks A and the valve constants C_0, C_1 and C_2, with values as shown in Table 4.1.

The continuous state-space model is obtained from the total mass balances in the two tanks. Assuming constant densities for the liquid contents in the tanks, this boils down to a volume balance. Further, the flow rate through the outlet valve in each tank is the function of the square root of the 'net' heights in the corresponding tanks [3]. The following model thus results:

$$\begin{aligned} A\frac{dH_1}{dt} &= W_0Q - W_1C_1\mathbf{sgn}(H_1 - H_2)\sqrt{|H_1 - H_2|} \\ A\frac{dH_2}{dt} &= W_1C_1\mathbf{sgn}(H_1 - H_2)\sqrt{|H_1 - H_2|} - W_2C_2\sqrt{H_2} \end{aligned} \tag{4.2}$$

where, $\mathbf{sgn}$ represents the sign function.

[2] To be clear, H_1 and H_2 are the 2 components of the state x, i.e. $x^1 = H_1$ and $x^2 = H_2$. Similarly $u^1 = W_0$, $u^2 = W_1$ and $u^3 = W_2$.

[3] The terms *inlet* and *outlet* valves refer to the locations of the valves and not the flow direction through the valve. Note that it is possible to have a flow *into* Tank 1 through the Tank 1 outlet valve if $H_2 > H_1$.

4.2.2 Performing HAZOP on the model

For the scope of this analysis, in addition to being discrete, the input is considered to be constant also. All 3 valves are considered to be in the 'open' position i.e. a constant $u = (1\ 1\ 1)$ is considered.

Further, the 'safe' or feasible operating region for this system is considered to be when the H_1 is between 2 m and 7 m, whereas H_2 is between 2 m and 6 m. The use of seven level sensors in each tank implies that there are a total of 6×6 i.e. 36 states to be considered.

The two component equilibrium hypersurfaces are obtained by setting the derivatives in the respective model equations to 0. These hypersurfaces are lines since the dimensionality of the state-space is 2.

$$\begin{aligned} W_0 Q &= W_1 C_1 \mathbf{sgn}(H_1 - H_2)\sqrt{|H_1 - H_2|} \\ W_1 C_1 \mathbf{sgn}(H_1 - H_2)\sqrt{|H_1 - H_2|} &= W_2 C_2 \sqrt{H_2} \end{aligned} \tag{4.3}$$

Putting in the values given in Table 4.1 for $u = (1\ 1\ 1)$ and squaring both sides, the following model equations are obtained:

$$\begin{aligned} H_2 &= H_1 - 4 \\ H_2 &= 0.5 H_1 \end{aligned} \tag{4.4}$$

The equilibrium lines in the state-space are shown in the Figure 4.4. The event-discretized state-space and the chosen safe operability region are together shown in Figure 4.5. The tuple numbering of the hypercubes is also shown.

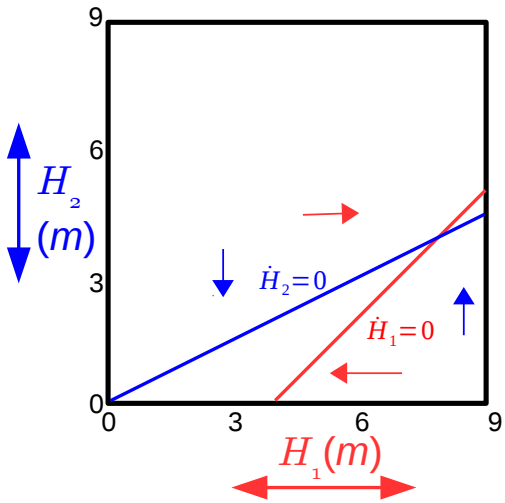


Figure 4.4: Equilibrium lines for the 2D model.

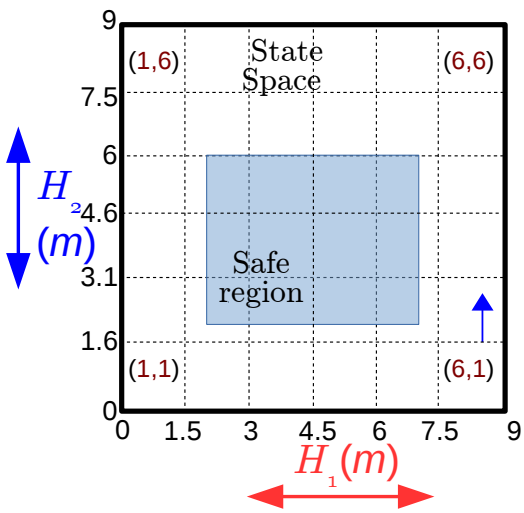


Figure 4.5: Event-discretized state-space, the chosen safe operability region and the numbering of hypercubes (here rectangles) for the system.

The automaton is typically represented as shown in Table 3.1. In this analysis, only one input $u = (1\ 1\ 1)$ is considered. According to Table 3.1, there will be 36 rows for each of the 36 states. The transitions caused by the given input u will be considered for each of these 36 states. The next state (or list of possible next states) will be shown next to each of the 36 current state entries for the given input u.

However, for the purposes of this analysis, a much better representation as shown in Table 4.2 is be used. This representation has the advantage that it directly maps the discretized state-space onto the automaton table, making for better visualization of the system [4]. So instead of showing the next states, it shows the *transitions* associated with each current state (hypercube). The next state associated with each state and the given input, although not explicitly shown, is implicitly obvious from the type of transition shown (T1, T2 or T3).

	Event boundaries (state component i) ...
Event boundaries (state component j)	Possible events (type of transitions) ...
$\vdots$	$\vdots$

Table 4.2: Automaton representation for the interacting tanks analysis for a given input u.

Using the DEDS modelling methodology explained in Chapter 3 for the interacting tanks system, the automaton shown in Table 4.3 is obtained. The automaton in Table 4.3 maps directly onto the discretized state-space shown in Figure 4.5. This means that the 6×6 entries in the automaton can be superimposed directly onto the 6×6 grid squares (hypercubes) in Figure 4.6. Thus the automaton denotes the type of transitions that the state is likely to make from a particular region in the $(H_1\ H_2)$ state-space for the given input u.

For example, the tuple marked $(1, 6)$ in Figure 4.6 corresponds to the top left entry in Table 4.3: (T1,T2). This means that for this hypercube, the transition with respect to H_1 will be T1 i.e. positive (rightwards) and with respect to H_2 will be T2 i.e negative (downwards). This is because both components will move towards their corresponding equilibrium hypersurfaces (left or right for H_1, up or down for H_2).

With the knowledge of the automaton and the safe region, the so-called leak-hypercubes can be identified. The leaks in hypercubes $(2, 2)$ and $(5, 4)$ will be

[4]Note that this representation may only be used for 2D or 3D systems, since such a tabular representation for higher dimensions cannot be realized in print.

	Automaton					
	(T1,T2)	(T1,T2)	(T1,T2)	(T1,T2)	(T1,T2)	(T1,T2)
	(T1,T2)	(T1,T2)	(T1,T2)	(T1,T2)	(T1,T2)	(T1,T2)
$\uparrow$	(T1,T2)	(T1,T2)	(T1,T2)	(T1,T2)	(T1,T2)	(T3,T2)
H_2	(T1,T2)	(T1,T2)	(T1,T2)	(T1,T2)	(T3,T3)	(T3,T3)
$\downarrow$	(T1,T2)	(T1,T2)	(T1,T3)	(T3,T3)	(T3,T3)	(T2,T1)
	(T1,T3)	(T1,T3)	(T3,T3)	(T3,T1)	(T2,T1)	(T2,T1)
			$\leftarrow H_1 \rightarrow$			

Table 4.3: Model automaton for the system for $u = (1\ 1\ 1)$. The first entry in each tuple corresponds to transition types with respect to H_1 and the second entry to those with respect to H_2.

detected by the automaton. The hypercube $(2, 2)$ has the T2 transition associated with it in the H_2 direction and this transition will take the state out of the safe region. Similarly the hypercube $(5, 4)$ has the T1 transition associated with it in the H_1 direction and this transition will take the state out of the safe region. This is illustrated in Figure 4.6.

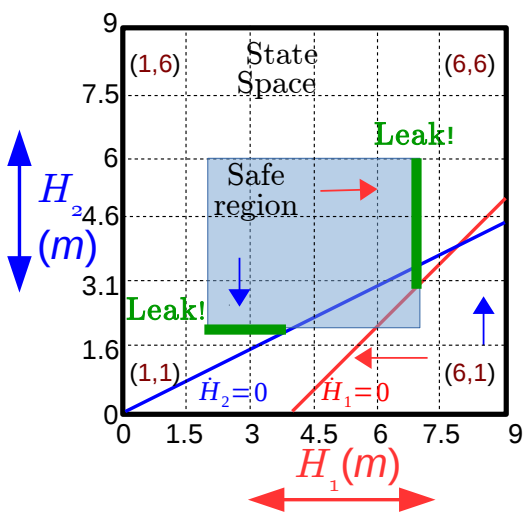


Figure 4.6: The equilibrium lines, the discretization, the safe region and the leaks in the system. The tuples show the numbering of the hypercubes.

However, the leaks in hypercubes $(3, 2)$ and $(5, 3)$ will *not* be detected since these correspond to T3 transitions (since the equilibrium lines passes through these hypercubes). In cases such as these, selective finer discretization (as discussed in-Section 3.3.2) can be employed near the equilibrium lines to get better resolution. The automated nature of such a HAZOP analysis makes it possible to take preventive actions even in complex plants where the failures are seemingly 'random' (i.e those failures occurring due to a chance overlap of simultaneous events rather than due a cause-and-effect mechanism). When a leak detection is made, the supervisor can initiate an appropriate control action to 'steer' the system back into the safe operability region.

The leaks in the interacting tanks case are easy to understand instinctively due to the simplistic nature of the case. Since the inflow valve is open, it is to be expected that a possible breach of the safe region will be when the level in Tank 1 becomes too high. Also, the outflow valve on Tank 2 is open, continuously draining the tank. Thus another possible breach is when the level in Tank 2 becomes too low. In the absence of any control action, which leak will happen first, or if any leak happens at all, is decided by the initial state of the system i.e. which hypercube the system starts in. It may be recalled from Chapter 3 that the initial state of the system is also generally modelled into the automaton. Some relevant issues relating to the automated HAZOP analysis are discussed in the next section.

4.3 Some pertinent issues

4.3.1 Model robustness

There is always a certain gap between the developed model and the actual system dynamics. A perfect model representation of the actual system dynamics is not possible. It therefore becomes imperative this automated HAZOP approach to take into consideration the *robustness* of the model. This is because, in terms of hazard and safety issues, a model mismatch may turn out to be very dangerous, not to mention costly.

For example, the automaton would identify a wrong leak-hypercubes or not identify any leak-hypercubes at all if there is a serious model mismatch. One way of dealing with model robustness is to 'shrink' the safe operability region as shown in Figure 4.7. That is, one can have tighter specifications on the operability conditions than those deemed conventionally necessary in the system. This creates a 'buffer' to account for any modelling errors. Thus, a model that is more robust would need a smaller buffer in the safe operability region. A less robust model, on the other hand, would have to account for larger deviations from expected behaviour and thus need a larger buffer in the safe operability region.

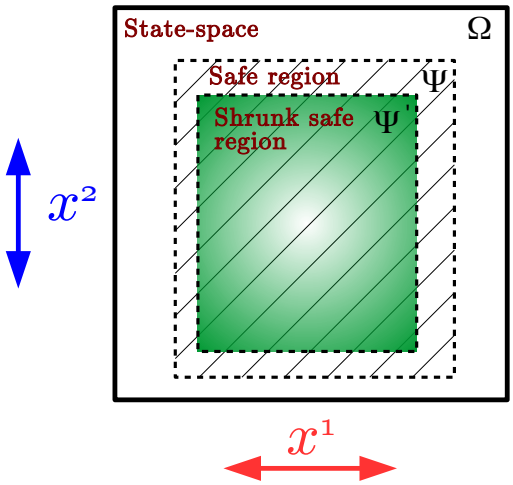


Figure 4.7: The shrunk safe region ψ'.

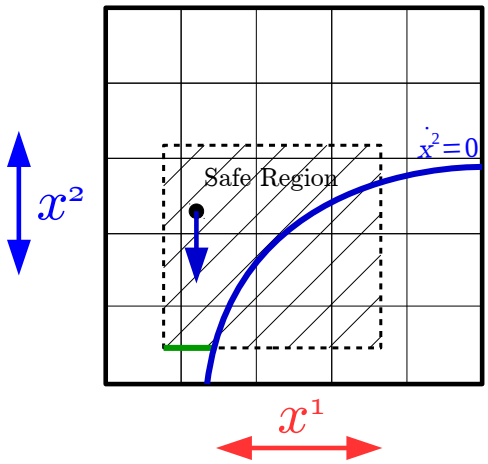
4.3.2 Changing inputs to ensure safe operability

The automaton is used to identify regions in the state-space where there are possibilities of leaks in the safe operability region i.e. the so-called leak-hypercubes. The exact leaks themselves (from a modelling perspective, not an operability perspective), however, do not depend on the discretization of the state-space (see Definition 4.1). They depend on the equilibrium hypersurfaces as defined by the continuous model and their intersections with the boundaries of the chosen safe region (see Figure 4.2).

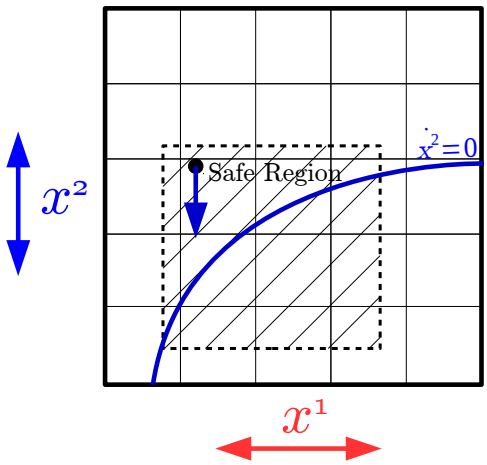
So one can change the environmental conditions i.e. the model inputs to 'move' the equilibrium hypersurfaces such that safe operability is ensured. This is a supervisory control action that may be implemented in response to the automaton identifying a leak-hypercube. The idea is illustrated in Figure 4.8.

In the case of the interacting tanks, the control action may be to stop the inflow rate Q by closing the inlet valve so as to prevent H_1 from rising. Similarly the outlet valve on Tank 2 can be closed to prevent H_2 to go too low. Another possibility is to add an extra inlet valve for Tank 2 (this would increase number of components of input u from 3 to 4). From the modelling perspective, these changes would shift the equilibrium lines for H_1 and H_2 and possibly eliminate the leaks.

All such control actions in the plant can be automatically taken by a supervisory controller designed according to given specifications. A supervisor thus needs to be designed for the given DEDS model and the given specifications. The topic of supervisory control and supervisor synthesis are the subject of Chapter 5.



(a)



(b)

Figure 4.8: The leak (shown in bold green) in (a) is eliminated in (b) by moving the equilibrium line for x^2.

Chapter 5

Supervisory Control of Discrete-Event Dynamic Systems

The discussion up to this point was limited to the modelling of DEDS and analysing how they evolve; and using this analysis to make predictions about plant behaviour (such as HAZOP). In this chapter, however, the focus shifts to the *control* of such discrete-event systems. This chapter relates to the supervisory control of plants. More precisely, it relates to how the behaviour of the DEDS describing the plant can be modified by a so-called supervisor - denoted by S - in such a way that the DEDS meets certain given *specifications*. Note here that supervisory control systems are typically themselves DEDS, described by automata.

Specifications for plant behaviour may be needed for a variety of reasons. Suppose that the plant behaviour is modelled by an automaton G. Certain discrete states in G may be undesirable because of safety and operability issues, as in case of HAZOP discussed in Chapter 4. In context of HAZOP, all discrete states in G that are outside the safe operability region are undesirable.

In some cases, it might be the case that G reaches a state which is in 'deadlock' (meaning that the system cannot escape the state), making it undesirable. These situations are common in purely discrete-event systems like complex scheduling problems or computer systems. A possible specification might relate to the physical inadmissibility of a certain discrete state - for example negative mass fractions are not possible. It should however be noted here that it is best to avoid such scenarios in the modelling phase itself, rather than in the controller design phase.

A common specification is about the *reachability* of certain states in the plant. This means that given an initial state, the task of the supervisor S is to direct the trajectory of the state to the desired final state via a path in the state-space that is

considered 'optimal'. The optimality might be related to time, if it is considered. That is, the supervisor S should be able to take the system from state a to state b as fast as possible. More commonly though, the optimality factors in the profitability of the plant. For instance, even though there may be multiple 'paths' for a plant to go from state a to state b, one of these paths would be optimal in terms of profits (due to lower energy costs, better resource usage, lessening the burden on the lower level regulatory control operations, etc.).

Stated in rudimentary terms, supervisor synthesis would involve the following steps:

- Define a specification relating to the desired behaviour of the system - this would be the 'control goal' that needs to be realized
- Next, a control strategy needs to be devised in order meet the desired specification
- Finally, the supervisor may be designed according to the chosen control strategy

5.1 A simple example about the reachability specification

A representative example is given here to demonstrate the reachability specification. How to devise the control strategy and how to synthesise a supervisor for this specification, is also shown.

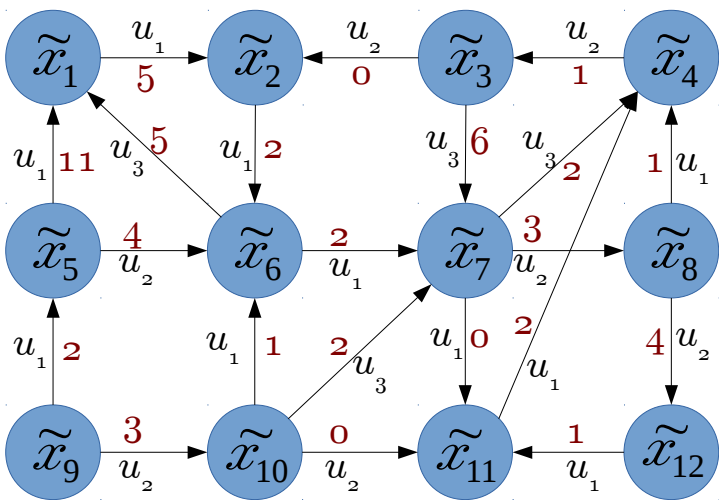


Figure 5.1: Directed graph of a purely discrete-event system with weights for transitions.

Consider the automaton represented by the directed graph shown in Figure 5.1. The system has 12 discrete states i.e. $\widetilde{X} = \{\tilde{x}_1, \ldots, \tilde{x}_{12}\}$, and 3 discrete inputs i.e. $\widetilde{U} = \{u_1, u_2, u_3\}$. The inputs are input events that cause transitions between the discrete states. The numbers associated with each transition show the 'weights' associated with each transition. Each number may be thought of as a 'cost' of making the transition. The weights can be thought of as being related to the specifications of the desired behaviour. Transitions that are desired are given low weights and those that are undesired are given higher weights.

Suppose that the system is initially in the discrete state $\tilde{x}_9$ and it is desired to reach state $\tilde{x}_2$, in the most optimal way. That is, the reachability of $\tilde{x}_2$ from $\tilde{x}_9$ has to be ensured by the supervisor in a way that minimises the associated cost.

It is evident from Figure 5.1 that there are multiple paths from $\tilde{x}_9$ to $\tilde{x}_2$. The most cost-effective path can be found using Dijkstra's algorithm, which is an algorithm conceived by computer scientist Edsger Dijkstra for finding the shortest path between nodes in a graph (Dijkstra, 1959). A Python code for Dijkstra's algorithm is provided in Appendix B.2. Running the algorithm for the graph shown in Figure 5.1, the most optimal path is found to follow the trajectory:

$$\tilde{x}_9 \rightarrow \tilde{x}_{10} \rightarrow \tilde{x}_{11} \rightarrow \tilde{x}_4 \rightarrow \tilde{x}_3 \rightarrow \tilde{x}_2$$

Now that the path is known, a supervisor can be synthesised based on the knowledge of which inputs cause the needed transitions so that the most optimal path is followed [1]. To be precise, when the system is in state $\tilde{x}_9$, the supervisor will tell the system to activate the input u_2 to allow the transition to $\tilde{x}_{10}$. When the system is in state $\tilde{x}_{10}$, the supervisor will again tell the system to activate the input u_2 to allow the transition to $\tilde{x}_{11}$, and so on. The supervisory action would stop when the system reaches the target state $\tilde{x}_2$. This can be represented by the following sequence:

$$\tilde{x}_9 \xrightarrow{u_2} \tilde{x}_{10} \xrightarrow{u_2} \tilde{x}_{11} \xrightarrow{u_1} \tilde{x}_4 \xrightarrow{u_2} \tilde{x}_3 \xrightarrow{u_2} \tilde{x}_2$$

The supervisor thus directs the system to execute the input sequence: $u_2u_2u_1u_2u_2$.

Suppose now that the system makes the undesirable transition from $\tilde{x}_9$ to $\tilde{x}_5$ by activating the input u_1 (say due to a disturbance in the system). Now the supervisor can implement a new optimal trajectory from the new state $\tilde{x}_5$ to $\tilde{x}_2$ Applying the Dijkstra's algorithm from $\tilde{x}_5$ to $\tilde{x}_2$, the following sequence is obtained:

$$\tilde{x}_9 \xrightarrow{undesirable} \tilde{x}_5 \rightarrow \tilde{x}_6 \rightarrow \tilde{x}_7 \rightarrow \tilde{x}_4 \rightarrow \tilde{x}_3 \rightarrow \tilde{x}_2$$

The input sequence will then be: $u_1u_2u_1u_3u_2u_2$, with the input in red being the one causing the undesirable transition.

[1] Note that the system automaton considered here is deterministic.

Such a 'corrective' control strategy can be implemented every time an undesirable transition occurs. It should be noted here, however, that the supervisor cannot direct the system between every given pair of states. For instance, supervisor cannot direct the system from $\tilde{x}_2$ to $\tilde{x}_9$, since such path does not exit. This, however is the property of the system and not a limitation of the supervisor.

The system considered was purely discrete. The main objective in this chapter, however, is to design a supervisor for a continuous plant with a discretized state-space and discrete inputs. The example presented here was meant to build some intuition about devising a control strategy and synthesising a supervisor, to meet the reachability specification. The basic idea for devising the control strategy for DEDS models abstracted from continuous systems, discussed in the next section, will be similar.

5.2 Control of DEDS models of continuous systems

Supervisor design here is based on the automaton model abstracted from the plant, but in addition, the gradient information provided by the underlying continuous model of the plant can also be used. The gradient information allows for the use of control actions that are not possible to implement if only the information from the automaton model is used. Much of the theory that follows is inspired from Philips (2001) and Philips et al. (2003).

It may be recalled from Chapter 3 that the continuous system under consideration is as follows:

$$\dot{x}(t) = f(x(t), u(t)); \quad x(t_0) = x_0 \tag{5.1}$$

where the boundaries β^i for each state component x^i, induced by the event sensors, can be represented as:

$$\beta_0^i \leq \beta_1^i \leq \beta_2^i \ldots \leq \beta_{p_i}^i \quad (p^i \geq 1) \tag{5.2}$$

The boundaries lead to the discretization of the state-space into hypercubes $H_x(\tilde{x})$ representing the different discrete states. The inputs are considered to be already discrete or piecewise continuous so that u belongs to some discrete set $\widetilde{U}$.

As discussed before, any observer - the supervisor in this case - would only have information about the hypercube, and not the actual continuous state within the hypercube. The supervisor should then be able to apply inputs from the discrete set $\widetilde{U}$ to the system so as to meet the control specifications. The state trajectory that evolves under such supervisory control is referred to as a *discretely controlled* trajectory, denoted by ξ. Similar to Section 5.1, the reachability specification will be considered here as well.

5.2.1 The reachability specification

Consider the continuous system given by Equation 5.1, the discretized state-space induced by the boundaries given by Equation 5.2 and the set of discrete inputs $\widetilde{U}$. Suppose that the initial state of the system x_0 corresponds to the hypercube $H_x(\tilde{x}_0)$ i.e. $x_0 \in H_x(\tilde{x}_0)$. Suppose further that it is desired to reach the discrete state represented by the hypercube $H_x(\tilde{x}_e)$.

The objective here is to design a supervisor such that for any $x_0 \in H_x(\tilde{x}_0)$, the discretely controlled trajectory ξ (with $\xi(0) = x_0$) implemented by the supervisor intersects with the hypercube $H_x(\tilde{x}_e)$. Put simply, the task for the supervisor is to control the trajectory from the initial hypercube to the final specified hypercube in the discretized state-space.

An important point discussed in Section 3.3.3 must be recalled here. The assumption was made that only transitions between adjacent hypercubes are allowed. This also necessitates a further assumption that the discretely controlled trajectory ξ will be implemented by the supervisor in such a way that only transitions between adjacent hypercubes are allowed. That is, the trajectory will never reach a point in the continuous state-space that belongs to more than two hypercubes (corners or edges).

5.2.2 Control actions

Again, consider the continuous system 5.1, the discretized state-space induced by the boundaries 5.2 and the set of discrete inputs $\widetilde{U}$. And again, consider two adjacent states $\tilde{x}_1$ and $\tilde{x}_2$, according to the following:

- $\tilde{x}_1$ and $\tilde{x}_2$ are represented by hypercubes that are labelled by the n-tuples $(a^1, \ldots, a^r, \ldots, a^n)$ and $(a^1, \ldots, a^r + 1, \ldots, a^n)$, respectively. This is to say that the two hypercubes are adjacent in the rth dimension of the state-space.

- The boundary hypersurface between these two hypercubes is denoted by $H_x(\tilde{x}_1) \cap H_x(\tilde{x}_2)$. In this case, this is given by the locus of the points $\{x \in \mathbb{R}^n | x^r = \beta^r_{a^r}\}$.

- $x \in H_x(\tilde{x}_1) \implies x^r \leq \beta^r_{a^r}$ and $x \in H_x(\tilde{x}_2) \implies x^r \geq \beta^r_{a^r}$. This specifies the positions of the adjacent hypercubes relative to each other i.e. $H_x(\tilde{x}_1)$ comes 'before' $H_x(\tilde{x}_2)$, when moving in the positive direction in the rth dimension.

Also, let the rth element of the f vector in Equation 5.1 be denoted by f^r.

The supervisor implements the control through the discrete inputs. With respect to transitions between hypercubes, the so-called preventing inputs and moving inputs are considered (Philips et al., 2003; Philips, 2001). These types are explained here.

Preventing inputs

A preventing input prevents a transition between hypercubes from happening. It is assumed that the continuous trajectory reaching the boundary between two hypercubes is detected instantaneously. Further the control action is also assumed to be instantaneous i.e. the preventing input is considered to be applied immediately when the continuous trajectory reaches the boundary between two hypercubes. The preventing input is shown in Figure 5.2.

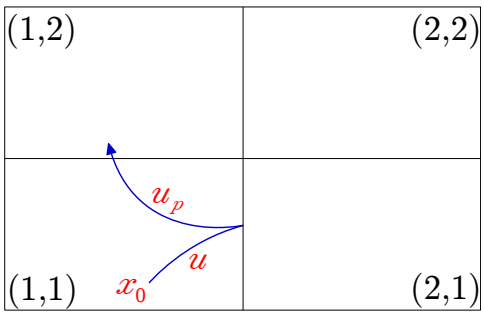


Figure 5.2: The input u_p is preventing for the transition $H_x((1,1)) \to H_x((2,1))$.

Definition 5.1. (Philips et al., 2003) An input $u \in \widetilde{U}$ is preventing for the transition $\tilde{x}_1 \to \tilde{x}_2$ if $\tilde{x}_2 \notin \phi(\tilde{x}_1, u)$.

It was discussed that in Section 3.2.4 that, for a given discrete input, the transition $\tilde{x}_1 \to \tilde{x}_2$ is *possible* if and only if $f^r > 0$ at *some* point on the boundary surface between the hypercubes (see Equation 3.7). It thus follows that an input *will* be preventing for the same transition if $f^r \leq 0$ for *all* points on the boundary surface. This is because the given input will *never* result in the transition if $f^r \leq 0$ for *all* points on the boundary surface.

An input $u \in \widetilde{U}$ is preventing for the transition $\tilde{x}_1 \to \tilde{x}_2$ if and only if

$$f^r(x, u) \leq 0, \ \forall x \in H_x(\tilde{x}_1) \cap H_x(\tilde{x}_2) \tag{5.3}$$

This should be interpreted as the sufficient condition for an input to prevent the transition.

Moving inputs

A moving input simply makes certain that the state trajectory moves in the desired direction. Applying this input ensures that when the continuous state trajectory leaves a hypercube, it is closer to the desired target hypercube.

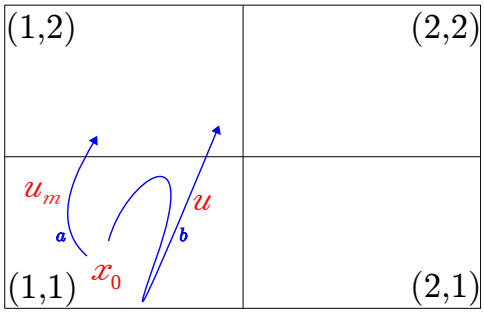


Figure 5.3: The input u_m corresponding to trajectory a is moving for the transition $H_x((1,1)) \to H_x((1,2))$. Another input u corresponds to trajectory b. The input u still takes the state in the desired direction, but is not considered in the definition of the moving input, since the state derivative does not have the same sign throughout $H_x((1,1))$.

For the continuous state to maintain the same directionality throughout the interior of the hypercube for a given discrete input, it is necessary that f^r has the same sign at *all* points in the interior of the hypercube for a given discrete input. This corresponds to the trajectory a in Figure 5.3.

Definition 5.2. (Philips et al., 2003) An input $u \in \widetilde{U}$ is moving for the transition $\tilde{x}_1 \to \tilde{x}_2$ if

$$f^r(x,u) > 0,\ \forall x \in H_x(\tilde{x}_1) \tag{5.4}$$

This is a how a moving input is *defined*, it should not be interpreted as the *condition* for an input to move a state trajectory in the direction of the desired hypercube. This is because even if f^r has different signs at different points within the hypercube, the discrete input may still ensure that the continuous state eventually exits the hypercube in the desired direction. This is the case for trajectory b shown in Figure 5.3. But, for the purposes of this chapter, it is assumed that this does not happen and the possibility is excluded from the definition of the moving input, for convenience.

5.2.3 Forceable transitions

The control strategy can be devised using the available control actions discussed in Section 5.2.2. The idea is to abstract a directed graph like the one shown in the

simple reachability example of Section 5.1, so that the reachability specification may be realized using the kind of 'shortest path' algorithms like the Dijkstra's algorithm.

The directed graph considered in the simple reachability example was that of a deterministic automaton (for a given state, a particular input only caused a particular transition). However, it has been discussed that when abstracting DEDS models from continuous systems, the resulting automata are typically non-deterministic (Mealy automata). So for a given state, a particular input may cause one of the multiple possible transitions.

The idea, then, is to avoid non-determinism by finding those transitions between adjacent discrete states (hypercubes) that can be 'forced' by the available control actions. That is, to find those inputs, from the discrete set $\widetilde{U}$, that ensure that a particular transition happens while ruling out other possible transitions.

Definition 5.3. (Philips et al., 2003) A transition $\tilde{x}_1 \rightarrow \tilde{x}_2$ between two adjacent discrete states is called *forceable*, if from each initial continuous state $x_0 \in H_x(\tilde{x}_1)$ there exists a discretely controlled trajectory ξ (with $\xi(0) = x_0$) for which there exists a $t_1 > 0$ such that $\xi(t) \in H_x(\tilde{x}_1) \cup H_x(\tilde{x}_1)\ \forall t \in [0, t_1]$ and $\xi(t_1) \in int(H_x(\tilde{x}_2))$.

Put simply, if there exists a ξ starting in $H_x(\tilde{x}_1)$ that only crosses the boundary hypersurface between $H_x(\tilde{x}_1)$ and $H_x(\tilde{x}_2)$, without crossing any other boundary hypersurface of $H_x(\tilde{x}_1)$, then the transition between $H_x(\tilde{x}_1)$ and $H_x(\tilde{x}_2)$ is forceable.

The sufficient condition for a transition to be forceable can be formulated in terms of the control actions. Consider two adjacent discrete states $\tilde{x}_1$ and $\tilde{x}_2$ and consider a set of inputs that are moving for the transition $\tilde{x}_1 \rightarrow \tilde{x}_2$. Now, if in this set of moving inputs there exists an input that is also preventing for a transition $\tilde{x}_1 \rightarrow \tilde{x}_3 \neq \tilde{x}_2$ to any state $\tilde{x}_3$ that is also adjacent to $\tilde{x}_1$, then such an input will force the transition $\tilde{x}_1 \rightarrow \tilde{x}_2$. So an input $u \in \widetilde{U}$ will force the transition $\tilde{x}_1 \rightarrow \tilde{x}_2$ if:

- u is moving for the transition $\tilde{x}_1 \rightarrow \tilde{x}_2$, and
- u is preventing for any transition $\tilde{x}_1 \rightarrow \tilde{x}_3$, where $\tilde{x}_3$ is any state adjacent to $\tilde{x}_1$, other than $\tilde{x}_2$

All such forceable transitions then define a directed graph like the one shown in the simple reachability example of Section 5.1. Such a directed graph is referred to as the *forceabilty graph* (Philips, 2001). The shortest path algorithms can now be used on the forceability graph to realize the reachability specification.

It is, however, important to note that the existence of a path from some state a to state b within the forceability graph is a sufficient but not *necessary* condition for

realizing the reachability specification. This is because the trajectory may go from state a to state b through non-forceable transitions as well, and the forceability graph does not provide this information. So state b may or may not be reachable from state a if a direct path does not exist within the forceability graph. But it is *definitely* reachable if a direct path *does* exist.

5.3 Case study: Two Tanks extended

Consider the case of two interacting tanks that was used for the HAZOP analysis in Chapter 4. To have more options for controlling the levels in the two tanks, an inlet to Tank 2 is additionally considered, as shown in Figure 5.4. The inflow rate for this second tank is same as that of the first tank i.e. Q (m^3/min). The valve constant on this inlet line is $C_3(m^{2.5}/min)$. Further, 6 level sensors each are considered for both the tanks.

The values of these constants are given in Table 5.1. The locations of the level sensors are: 0 m, 1 m, 3 m, 4 m, 6 m, 7 m for Tank 1 and 0 m, 1.9 m, 3.6 m, 4.5 m, 6.5 m, 7.5 m for Tank 2. These sensors would partition the $(H_1\ H_2)$ state-space into $5 \times 5 = 25$ hypercubes.

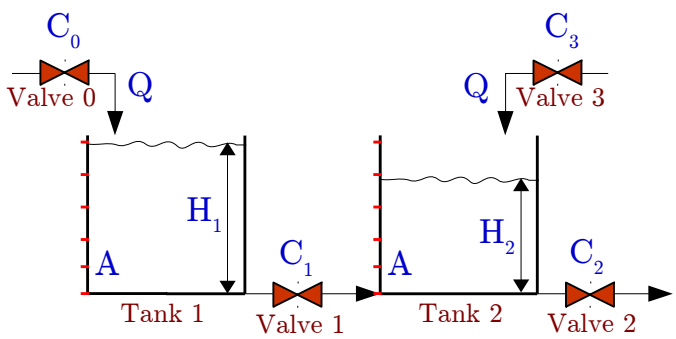


Figure 5.4: System of two tanks connected in series - control example.

Constants	Value	Units
Q	0.3	m^3/min
C_0	0.15	$m^{2.5}/min$
C_1	0.15	$m^{2.5}/min$
C_2	0.15	$m^{2.5}/min$
C_3	0.15	$m^{2.5}/min$
A	0.5	m^2

Table 5.1: Constants in the two tanks control model.

5.3.1 Model Derivation

The model derivation is similar to the HAZOP case study considered in Chapter 4. Now, there are 4 valves in the system i.e. $u = (W_0 \quad W_1 \quad W_2 \quad W_3)$. This means that there are a total of $2^4 = 16$ discrete inputs. For example, the input $u = (1\ 0\ 0\ 1)$ represents the case where both the inlet valves tanks are open but both outlet valves are closed.

The model in Equations 4.3 are modified to incorporate the extra inlet valve W_3.

$$\begin{aligned} A\frac{dH_1}{dt} &= W_0 Q - W_1 C_1 \mathbf{sgn}(H_1 - H_2)\sqrt{|H_1 - H_2|} \\ A\frac{dH_2}{dt} &= W_3 Q + W_1 C_1 \mathbf{sgn}(H_1 - H_2)\sqrt{|H_1 - H_2|} - W_2 C_2 \sqrt{H_2} \end{aligned} \tag{5.5}$$

where, $\mathbf{sgn}$ represents the sign function.

In the HAZOP case, only a single input was considered for analysis, where all valves were considered to be open. In this case, however, the different inputs are to be used for control purposes. The next section discusses how (and if) the different inputs can force various transitions among the 25 hypercubes.

5.3.2 Getting the forceability graph

The task here is to find all the forceable transitions between adjacent hypercubes using the different discrete inputs. As was discussed in Section 5.2.3, finding forceable transitions involves checking for moving and preventing inputs, for the given pair of hypercubes.

According to the discussion in Section 5.2.2, finding these inputs boils down to checking the values of f^r at various points and boundaries in the state-space, for a given pair of adjacent hypercubes and a given discrete input. The task of finding all forceable transitions can be accomplished by a code that does this for every pair of adjacent discrete states and for each discrete input.

In this case there are 25 discrete states and 16 discrete inputs. Presenting the analysis for every possible combination of discrete state pairs and discrete inputs is infeasible. Since this is a fairly simple case, however, most of the forceable transitions can be ascertained intuitively, as follows.

It can be seen the input $u = (1\ 0\ 0\ 0)$ causes Tank 1 to fill up without affecting the level in Tank 2. This is because only the Tank 1 inlet valve is open while all others are closed. This implies that all transitions where only Tank 1 level is increasing are forceable with $u = (1\ 0\ 0\ 0)$, as shown in Figure 5.5.

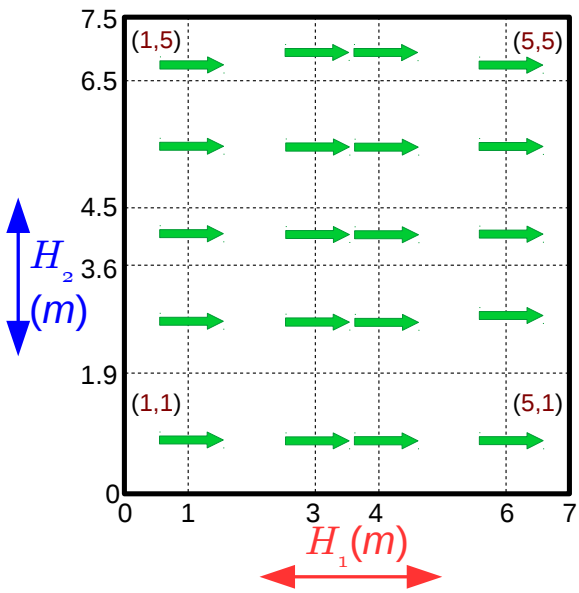


Figure 5.5: Forceable transitions for $u = (1\ 0\ 0\ 0)$, shown with green arrows.

This can also be verified by checking the sign of f^r. For $u = (1\ 0\ 0\ 0)$, the model equations are:

$$f^1 = \frac{dH_1}{dt} = \frac{Q}{A} = 0.3 \qquad > 0$$
$$f^2 = \frac{dH_2}{dt} \qquad = 0$$

Since $f^1 > 0$ everywhere in the state-space, the input is moving for all transitions in the positive H_1 direction, and preventing for all transitions in the negative H_1 direction. And $f^2 = 0$ everywhere in the state-space, which implies that the input is preventing for all transitions in the H_2 direction. Thus all transitions in the positive H_1 direction are forceable with the input $u = (1\ 0\ 0\ 0)$, as shown in Figure 5.5.

It can also be seen that the input $u = (0\ 0\ 0\ 1)$ causes Tank 2 to fill up without affecting the level in Tank 1, since only the inlet valve of Tank 2 is open while all other valves are closed. Moreover, the input $u = (0\ 0\ 1\ 0)$ causes Tank 2 to drain without affecting the level in Tank 1, since only the outlet valve of Tank 2 is open while all other valves are closed.

This means that all transitions in the positive H_2 direction are forceable with $u = (0\ 0\ 0\ 1)$, whereas all the transitions in the negative H_2 direction are forceable with $u = (0\ 0\ 1\ 0)$, as shown in Figure 5.6.

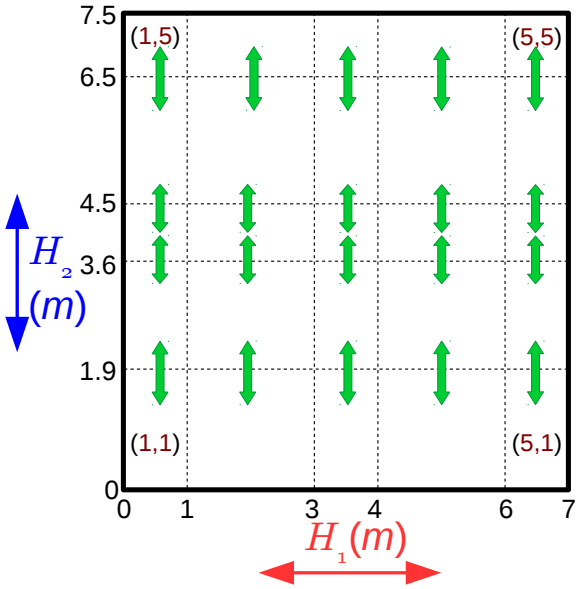


Figure 5.6: The transitions in the positive H_2 direction are forceable with $u = (0\ 0\ 0\ 1)$, and the transitions in the negative H_2 direction are forceable with $u = (0\ 0\ 1\ 0)$.

Verifying this through the gradient information, for $u = (0\ 0\ 0\ 1)$:

$$f^1 = \frac{dH_1}{dt} \qquad = 0$$
$$f^2 = \frac{dH_2}{dt} = \frac{Q}{A} = 0.3 \qquad > 0$$

Since $f^2 > 0$ everywhere, the input is moving for all transitions in the positive H_2 direction. Since $f^1 = 0$ everywhere, the input is also preventing for all transitions in the H_1 direction. Thus it can be seen that all transitions in the positive H_2 direction are forceable with $u = (0\ 0\ 0\ 1)$.

For $u = (0\ 0\ 1\ 0)$:

$$f^1 = \frac{dH_1}{dt} \qquad = 0$$
$$f^2 = \frac{dH_2}{dt} = -C_2\sqrt{H_2} \qquad < 0$$

Again, it can be seen that all transitions in the negative H_2 direction are forceable with $u = (0\ 0\ 1\ 0)$.

From the preceding analysis, it is clear that all transitions in the positive H_1, positive H_2 and negative H_2 directions are forceable. The analysis is interesting for finding inputs that can force transitions in the negative H_1 direction.

Tank 1 has no 'drain' valve since the flow through the outlet valve of Tank 1 depends on the difference in heights of the two tanks. So if $H_2 > H_1$ and the outlet valve on Tank 1 is open, the flow will be *into* Tank 1, increasing its level [2]. Thus it is clear that transitions in the negative H_1 direction are not possible with any input if $H_2 > H_1$. This is to say that the level in Tank 1 cannot be decreased if it is lower than the level in Tank 2.

If $H_1 > H_2$, transitions in the negative H_1 direction *may* be possible under the following circumstances:

- The Tank 1 inlet valve should obviously be in the 'closed' position, to avoid increase in H_1.
- The Tank 1 outlet valve should be in the 'open' position to allow for the decrease in H_1.
- The Tank 2 inlet valve should be in the 'closed' position to avoid $H_2 > H_1$.
- The Tank 2 outlet valve should be open to prevent the tendency of H_2 to increase simultaneously along with decrease in H_1 (recall that to force transitions in the negative H_1 direction, all transitions in the H_2 direction should be preventable).

Thus, the input $u = (0\ 1\ 1\ 0)$ is the best bet to force transition in the negative H_1 direction, provided $H_1 > H_2$. Note that even if $H_1 > H_2$, the input $u = (0\ 1\ 1\ 0)$ may not be able to force *all* transitions in the negative H_1 direction, because in some cases it might not be able to prevent transitions in the H_2 direction.

For $u = (0\ 1\ 1\ 0)$, the model equations become:

$$A\frac{dH_1}{dt} = -C_1\mathbf{sgn}(H_1 - H_2)\sqrt{|H_1 - H_2|}$$
$$A\frac{dH_2}{dt} = C_1\mathbf{sgn}(H_1 - H_2)\sqrt{|H_1 - H_2|} - C_2\sqrt{H_2}$$

The derivatives are set to zero and the following equilibrium lines result after simplification:

$$H_2 = H_1$$
$$H_2 = 0.5H_1$$

[2] Again, recall that the terms *inlet* and *outlet* valves refer to the locations of the valves and not the flow direction through the valve.

The exact transitions that can be forced in the negative H_1 direction by $u = (0\ 1\ 1\ 0)$ are shown in Figure 5.7.

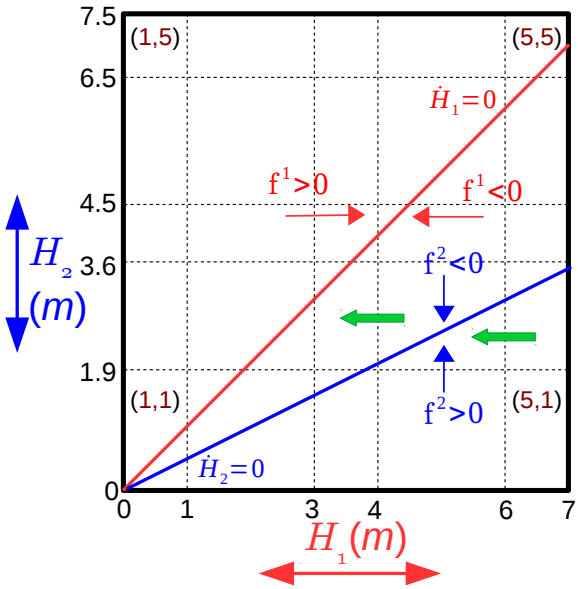


Figure 5.7: The forceable transitions for $u = (0\ 1\ 1\ 0)$, shown in green arrows. The equilibrium lines in the model in case of $u = (0\ 1\ 1\ 0)$ are also shown, with the red and blue arrows indicating the tendency of the state to move towards the corresponding equilibrium lines.

The possibility for forceable transitions in the negative H_1 direction can only be checked for those hypercubes where $H_1 > H_2$. Further, those hypercubes through which the $\dot{H}_1 = 0$ equilibrium line passes are ruled out since these correspond to Type 3 transitions in the H_1 direction. This leaves the hypercubes $(5, 3)$, $(5, 2)$, $(5, 1)$, $(4, 2)$, $(4, 1)$ and $(3, 1)$ as the 'eligible' hypercubes for checking for forceable transitions in the negative H_1 direction.

Consider the hypercube $(4, 2)$ in Figure 5.7. The hypercubes adjacent to $(4, 2)$ are the hypercubes $(5, 2)$, $(3, 2)$, $(4, 1)$ and $(4, 3)$. The red arrows indicate the tendency of the state to move towards the red equilibrium line. Since $f^1 < 0$ everywhere in this hypercube, the input $u = (0\ 1\ 1\ 0)$ is moving for the transition $(4, 2) \rightarrow (3, 2)$ and preventing for the transition $(4, 2) \rightarrow (5, 2)$.

Further it can be seen that the blue equilibrium line $\dot{H}_2 = 0$ lying within the hypercube $(4, 2)$, lies *entirely within* the boundaries of the hypercube in the H_2 direction i.e. between the 1.9 m and 3.6 m boundaries of Tank 2. Since the state has a tendency to move towards this blue equilibrium line, it means that the input $u = (0\ 1\ 1\ 0)$ will prevent the state from leaving the hypercube $(4, 2)$ in the H_2

direction i.e. the input is preventing for the $(4,2) \rightarrow (4,3)$ and $(4,2) \rightarrow (4,1)$ transitions. This can also be verified from the gradient information. Since $f^2 > 0$ at the boundary between the hypercubes $(4,2)$ and $(4,1)$, the input is preventing for the transition $(4,2) \rightarrow (4,1)$. And since $f^2 < 0$ at the boundary between the hypercubes $(4,2)$ and $(4,3)$, the input is preventing for the transition $(4,2) \rightarrow (4,3)$.

In conclusion, the transition $(4,2) \rightarrow (3,2)$ is forceable with the input $u = (0\ 1\ 1\ 0)$, since it is moving for this transition while also being preventing for transitions to all other hypercubes adjacent to $(4,2)$. A similar analysis for all the 'eligible' hypercubes reveals that only the transition $(4,2) \rightarrow (3,2)$ and the transition $(5,2) \rightarrow (4,2)$ are forceable with $u = (0\ 1\ 1\ 0)$, as shown in Figure 5.7.

Combining the results from Figures 5.5, 5.6 and 5.7, the forceability graph can be constructed, as shown in Figure 5.8. All transitions in the positive H_1 direction are forceable with the input $u = (1\ 0\ 0\ 0)$. All transitions in the positive H_2 direction are forceable with the input $u = (0\ 0\ 0\ 1)$. All transitions in the negative H_2 direction are forceable with the input $u = (0\ 0\ 1\ 0)$. Finally, those transition that are forceable in the negative H_1 direction are forceable with the input $u = (0\ 1\ 1\ 0)$.

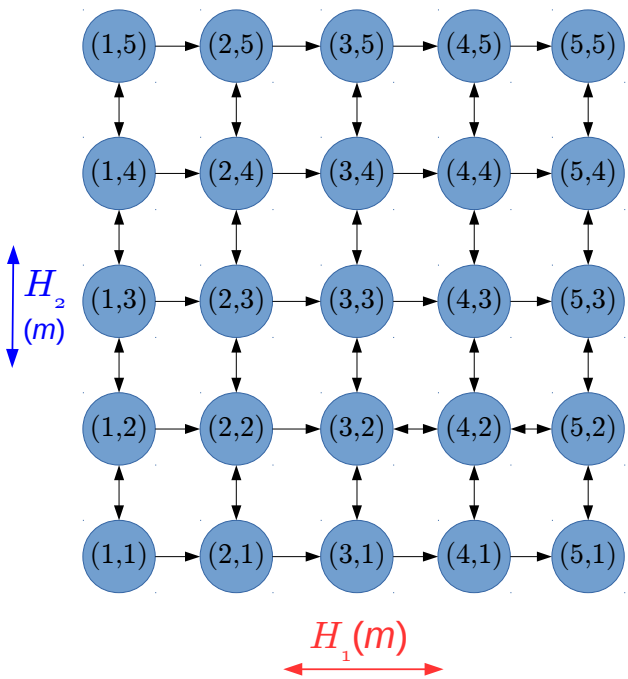


Figure 5.8: The resulting forceability graph for the two tanks system.

5.3.3 Reachability

Now, the reachability specification can be realized in a way similar to what was shown in the simple reachability example of Section 5.1. Suppose that the reachability specification states that the system needs to be steered from the starting point within hypercube $(5, 5)$ to a target hypercube $(4, 4)$. The Dijkstra's algorithm for the 'shortest path' can be applied for this forceability graph for the given initial and target states. Note that all transitions are given equal weights here, since there is no reason to believe that one transition is better than the other. However, if there is a more precise specification where different transitions are weighted differently, this can be handled by the algorithm.

The following sequence results:

$$H_x(5,5) \xrightarrow{(0,0,1,0)} H_x(5,4) \xrightarrow{(0,0,1,0)} H_x(5,3) \xrightarrow{(0,0,1,0)} H_x(5,2)$$

$$\xrightarrow{(0,1,1,0)} H_x(4,2) \xrightarrow{(0,0,0,1)} H_x(4,3) \xrightarrow{(0,0,0,1)} H_x(4,4)$$

A supervisor can thus be synthesized such that the following input sequence is implemented: $(0, 0, 1, 0)$, $(0, 0, 1, 0)$, $(0, 0, 1, 0)$, $(0, 1, 1, 0)$, $(0, 0, 0, 1)$, $(0, 0, 0, 1)$.

The first three transitions correspond to decrease in the level of Tank 2 by opening the Tank 2 outlet valve. Then the Tank 1 outlet valve is opened to exploit the height differential to decrease the level in Tank 1 to the desired level, but the Tank 2 outlet valve is also simultaneously kept open to prevent Tank 2 level from rising and reducing this height differential. When Tank 1 reaches the desired level, both outlet valves are closed and the Tank 2 inlet valve is opened to get the desired level in Tank 2.

As another example, to steer the system from hypercube $(5, 5)$ to $(3, 3)$, the following input sequence would be implemented: $(0, 0, 1, 0)$, $(0, 0, 1, 0)$, $(0, 0, 1, 0)$, $(0, 1, 1, 0)$, $(0, 1, 1, 0)$, $(0, 0, 0, 1)$.

5.4 Some pertinent issues

5.4.1 Use of correcting inputs

The preventing input is suitable for use when the sensor emits a signal instantaneously when a hypercube boundary is reached. The application of the preventing input is also assumed to be immediate. Practically though, such instantaneous relay of signals is not realizable by most sensors. If there is even a slight delay in relay of the signal, or if control action cannot be applied immediately, the state will cross the hypercube boundary and make the undesirable transition. In these cases, since prevention is not possible, the transitions need to be *corrected*.

A correcting input corrects a transition that has just occurred. This input steers the state back into the original hypercube. To use this input, it is sufficient to identify that a transition has taken place between two hypercubes. This control action need not be instantaneous. The correcting input is shown in Figure 5.9.

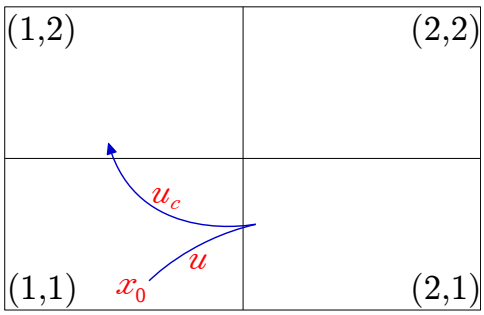


Figure 5.9: The input u_c is correcting for the transition $H_x((1,1)) \to H_x((2,1))$.

As discussed previously, for a given discrete input, the transition $\tilde{x}_1 \to \tilde{x}_2$ is *possible* if $f^r > 0$ at *some* point on the boundary surface between the hypercubes. It follows that the reverse transition *will happen* for a given discrete input if $f^r < 0$ for *all* points on the boundary.

Definition 5.4. (Philips et al., 2003) An input $u \in \widetilde{U}$ is correcting for the transition $\tilde{x}_1 \to \tilde{x}_2$ if

$$f^r(x,u) < 0, \ \forall x \in H_x(\tilde{x}_1) \cap H_x(\tilde{x}_2) \tag{5.6}$$

Like in the case of moving inputs, this should not be interpreted as the *condition* for an input to correct a transition. This is because the reverse transition $\tilde{x}_2 \to \tilde{x}_1$ is also possible if $f^r < 0$ at *some* point (instead of *all* points) on the boundary surface between the hypercubes. Again, though, it is assumed that this does not happen and such a possibility is excluded when defining the correcting input. Note that the inequality here is strict.

5.4.2 Other control strategies

The notion of forceability as defined in this chapter requires that all undesirable transitions be preventable for a particular transition between hypercubes to be forceable. This limits the scope of control actions. In some cases, a better control strategy is to force transitions between different subregions of the state-space rather than forcing transitions between particular discrete states themselves.

Consider the subregions A, B and C of the state-space as shown in Figure 5.10. There exists no forceable path from the discrete state a_1 to c_1. Thus a 'shortest path' algorithm cannot be employed.

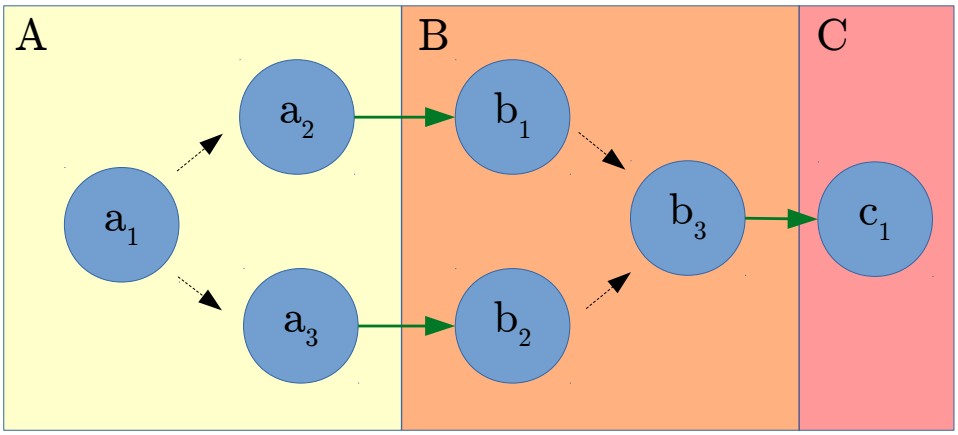


Figure 5.10: Control using forceable transitions between subregions of state-space. The arrows in green show forceable transitions between the shown discrete states. The black dashed arrows show transitions for which moving inputs exist, but cannot be forced.

The transitions $a_1 \rightarrow a_2$ and $a_1 \rightarrow a_3$ cannot be forced, but if they were to happen, then the subsequent transitions $a_2 \rightarrow a_1$ or $a_3 \rightarrow b_2$ can be forced. Similarly, the transitions $b_1 \rightarrow b_3$ and $b_2 \rightarrow b_3$ cannot be forced, but if they were to happen, the subsequent transition $b_3 \rightarrow c_1$ can be forced and the target state can be reached.

Thus the supervisor should 'allow' for transitions to states, from where subsequent transitions into other subregions 'closer' to the target can be forced. The supervisor thus forces transitions between subregions rather than between states. In the above example, the subregion transitions $A \rightarrow B$ and $B \rightarrow C$ can be forced by the supervisor. Further details and a more comprehensive analysis can be found in Philips (2001).

5.4.3 Note on formal verification

The discrete-event equivalent of the plant and the supervisor follow a feedback control scheme as shown in Figure 5.11. Both the plant (G) and the supervisor (S) follow discrete-event dynamics and can be modelled as automata. The output of the plant is the input to the supervisor whereas the output of the supervisor is the input to the plant. This will be reflected in the individual automata representation of the plant and the supervisor.

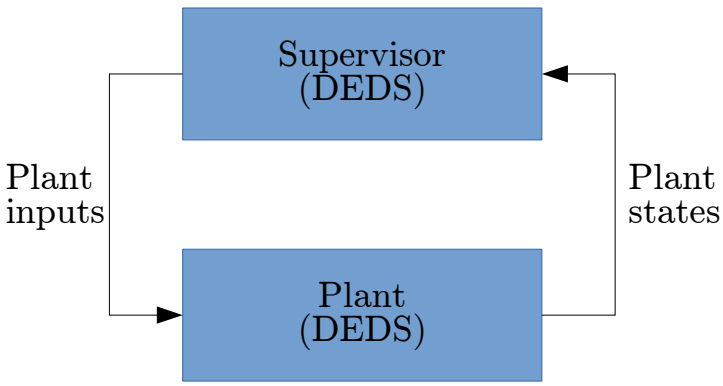


Figure 5.11: The plant-supervisor feedback loop.

The interesting thing to note here is that the two automata can be *combined* to form a single plant-supervisor automaton. The properties of this automaton can then be studied using *formal verification* of models. Stated simply, given a combined automaton S/G (where S/G denotes G under the supervision of S) and given a property P, the aim of the formal verification is to check whether S/G satisfies P, in an automated way.

Examples of properties that can be checked are the so-called 'blocking' properties of deadlock and livelock. A deadlock is when the system reaches a state in the automaton where no further transition is possible. A livelock is when the systems reaches a set of states that cannot be escaped, meaning that there is no transition going out of the set. The combined automaton can be checked for transitions that can lead to such deadlocks or livelocks.

The formal verification of the combined automaton ensures efficient supervisor synthesis. Verification techniques have been widely studied in literature (Huuck et al., 2002; Kowalewski, 2002; Tripakis and Dang, 2009; Wang, 2006).

5.4.4 Note on the 'Ramadge-Wonham' framework

Supervisory control of DEDS has been studied extensively in literature. The foundation has been laid by P. J. Ramadge and W. M. Wonham in the late 1980s in what is now called as the 'Supervisory Control Theory' (SCT) or the 'Ramadge-Wonham' (RW) framework (Ramadge and Wonham, 1987a,b; Wonham, 1989). This theory is based on automata and language models of the DEDS (Cassandras and Lafortune, 2010; Hopcroft et al., 2006).

In the RW framework, logical models of DEDS are studied i.e. chronological time is not considered. The process described by an automaton G evolves through different discrete states, brought on by events. The events are assumed to be generated spontaneously by the process. The behaviour of such a process is thus described by a sequence of events. Such a behaviour is modelled as an *untimed language* over the alphabet of events (see Chapter 3). The behaviour is denoted as $L(G)$.

The objective of the RW framework is to design a supervisor S for a given 'uncontrolled' process G such that the supervised process (S/G) behaves according to given specifications. The supervisor does this by restricting the behaviour of the process to a subset of $L(G)$. That is, the supervisor has the ability to *disable* some of the events that are being generated by process so as to modify the behaviour of the process. The framework also incorporates the fact that not all events may be 'observable' to the supervisor and not all observable events may be 'controllable' to the supervisor. It also formalizes the 'controllability' property i.e. the condition for the existence of a supervisor.

The framework has been extended for hierarchical supervision (Wong and Wonham, 1996; Zhong and Wonham, 1990) and decentralized control (Lin and Wonham, 1988a). Further additions to the framework have been proposed in Koutsoukos et al. (2000); Kumar and Garg (1995); Stiver et al. (1996); Thistle (1996) and Charbonnier et al. (1999).

The RW framework, however, proposes supervisor synthesis methods based only on the the discrete-event models of the systems i.e. it is based purely on automata and corresponding languages. This makes it attractive for systems that are purely discrete, like scheduling, queueing, communications, etc.

In context of this report, however, the focus is on discrete-event equivalents of continuous plants (Section 2.2). This means that, in addition to information about the 'discretized' plant behaviour through an automaton, knowledge about the plant's underlying continuous dynamics can also be exploited in designing the supervisor. The RW framework has no such provision to exploit information that can be obtained from the continuous origin of the automaton.

Furthermore, it can be seen from Section 3.2 that automata obtained from continuous systems have a high degree of non-determinism associated with them (re-

call that the DEDS equivalent of the continuous plant is a Mealy automaton). This is because in the discretized state-space described by hypercubes, an input may cause a state to transition to one of multiple adjacent hypercubes in the state-space. This also makes the RW framework unsuitable for controller synthesis in such systems.

The preceding discussion is meant to give only a superficial overview of the RW framework and why it is *not* suitable for synthesizing supervisors for the kind of systems that are studied in this report. It is, nonetheless, important to mention the RW framework in context of the topic of this thesis because it is the most widely cited theory when it comes to supervisory control of discrete-event systems.

Chapter 6

Conclusion and further work

Discrete-event dynamic systems that are abstracted from continuous systems are the focus of this thesis. The thesis discusses DEDS in context of three issues: modelling, analysis and control.

Modelling

DEDS are based on the premise of sampling events rather than time. The continuous state-space is discretized by event sensors into hypercubes that correspond to different discrete states. The framework for modelling such DEDS derived from continuous systems is presented, where the models are automata. The formalism involves the discretization of the state-space , discretization of the input-space and obtaining the transition function of the automaton. For the latter, gradient information from the original continuous dynamics, described by differential equations, is used. Implementation is fairly straightforward in that the gradient information can be easily obtained from the knowledge of the component equilibrium hypersurfaces, obtained by setting the derivatives in the continuous model to zero.

The computational cost of this procedure can be reduced by exploiting the spareness that is inherent in most continuous plant systems. Another way to reduce the computational effort is to modify the modelling procedure with the use of selective finer discretization to increase resolution in parts of the state-space. The automated nature of this procedure allows for 'switching' between different automata representing different regions in the state-space. A limitation of the framework is when the state trajectory passes through a point belonging to more than two hypercubes, since such a continuous trajectory cannot be represented by a fixed sequence of discrete transitions.

Analysis

The analysis presented in this thesis is the hazard and operability (HAZOP) analysis. The DEDS approach is a viable option for conducting quantitative HAZOP analysis in plants where low-frequency, high-risk hazards may be caused by combination of different failures. It provides a framework that forms an important part of the overall plant automation scheme. A region of safe operability is defined as part of this analysis. Detection of potential movements out of the safe operability region - so-called leaks - can be done with the help of the transition information available from the automaton model of the plant.

The quantitative HAZOP analysis is demonstrated through a benchmark case of two tanks connected in series. It is also shown how the selected state-space discretization and the chosen safe operability region affect the ability of the automaton to detect regions of potential leaks. An important conclusion is that while some hypercubes can be definitely identified as having positive or negative transitions with respect to a state-component, some others will always represent uncertainty in transitions - the so-called Type 3 transitions. It might be necessary to resort to selective finer discretization of such hypercubes to get higher accuracy.

Further it is important to account for model robustness when choosing the safe operability region - the safe region may have to be 'shrunk' to counter the inaccuracies in the model. Also, leaks may be eliminated by changing the inputs to 'move' the equilibrium hypersurfaces.

Control

With respect to the control hierarchy in a plant, the supervisory control of DEDS is discussed in this thesis. Synthesis of a supervisor involves defining a specification for the desired behaviour of the system, devising a control strategy to meet the desired specification and finally designing the supervisor according to the chosen control strategy. The reachability specification - steering the system from given initial state to a target state - is used for demonstrating the control strategy and supervisor design. The use of Dijkstra's 'shortest path' algorithm for directed graphs in shown on a simple reachability example to give a flavour of the procedure that has to be followed.

It is emphasized that when devising control strategies for DEDS abstracted from continuous systems, it is worthwhile to use the gradient information available from the underlying continuous dynamics rather than relying purely on the automaton model of the plant. The relevance of the widely cited 'Ramdge-Wonham' framework for supervisory control of discrete-event systems is discussed in this context. The control actions through the so-called preventing, correcting and moving inputs can be used to 'force' certain transitions in the discretized state-space.

All the forceable transitions define a forceability graph on which the Dijkstra's algorithm can be used to find the sequence of inputs that the supervisor needs to implement.

The case study of two tanks from HAZOP is extended with some modifications in this control discussion. The forceability graph can be obtained by systematically checking the gradients at various points and boundaries in the state-space, for various discrete inputs. An intuitive explanation of the two tanks forceability graph is presented, and reachability in this context is discussed. Control strategies can also be devised such that transitions between different subregions - rather than states - are forced by the supervisor.

6.1 Further work

When it comes to modelling, the presented formalism has some limitations that are discussed. Specifically, the continuous state trajectory is assumed not to pass through a point belonging to more than two hypercubes. The methodology can be developed further so as to relax this assumption.

Further, a more practical case study representing some complex process in the industry can be developed, and the utility of the presented methodologies, especially HAZOP, can be studied. The use of DEDS models can be extended to incorporate logistical aspects of the process plant, for example the automation of the supply chain.

Concerning supervisory control, better control strategies can be formulated and specifications other than reachability can be considered. The formal verification for the kind of plant-supervisor automata discussed in this report has not been covered in literature, although the background automata theory has been extensively developed. A major avenue of further research in this domain is to use this developed automata theory to do the formal verification of supervisors synthesized for DEDS abstracted from continuous plants.

Finally, only untimed models of DEDS were considered in this work. The time information can be incorporated into the automaton - the so-called timed automaton. The time it takes for a state trajectory to cross two different boundaries of the same hypercube (entry and exit) can be ascertained, providing information on the transitions from the hypercube that take maximum and minimum times. This information can be used to devise better control strategies where time is a factor in the given control specification.

6.2 Final remarks

Discrete-event dynamic systems can easily be abstracted from real continuous plants. The models of these systems can be used in the industry for a variety of important analyses, particularly related to safety issues in plants. The control of plants based on these models is not only easy, but also cheap, to implement.

The biggest advantage of using the DEDS approach is that it is very exhaustive in it's scope, while hardly requiring any human intervention. In a world that is becoming increasingly automated, the methodologies presented in this work definitely point in the direction of the automation of the process industry.

Bibliography

Blanchini, F., 1999. Set invariance in control. Automatica 35, 1747–1767.

Bouyer, P., Chevalier, F., D'Souza, D., 2005. Fault Diagnosis Using Timed Automata. Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 219–233.
URL http://dx.doi.org/10.1007/978-3-540-31982-5_14

Cassandras, C. G., Lafortune, S., 2010. Introduction to Discrete Event Systems. Springer US.

Chang, C.-T., Chen, C. Y., 2011. Fault diagnosis with automata generated languages. Computers & Chemical Engineering 35 (2), 329 – 341.
URL http://www.sciencedirect.com/science/article/pii/S0098135410003285

Charbonnier, F., Alla, H., David, R., March 1999. The supervised control of discrete-event dynamic systems. IEEE Transactions on Control Systems Technology 7 (2), 175 – 187.

Crawley, F., Preston, M., Tyler, B., of Chemical Engineers (Great Britain), I., Centre, E. P. S., 2000. HAZOP: Guide to Best Practice : Guidelines to Best Practice for the Process and Chemical Industries. Institution of Chemical Engineers.

Dijkstra, E. W., Dec. 1959. A note on two problems in connexion with graphs. Numer. Math. 1 (1), 269–271.
URL http://dx.doi.org/10.1007/BF01386390

Dunj, J., Fthenakis, V., Vlchez, J. A., Arnaldos, J., 2010. Hazard and operability (hazop) analysis. a literature review. Journal of Hazardous Materials 173 (13), 19 – 32.
URL http://www.sciencedirect.com/science/article/pii/S0304389409013727

Eppstein, D., 2002. Dijkstra's algorithm for shortest paths (python recipe). http://code.activestate.com/recipes/119466-dijkstras-algorithm-for-shortest-paths/ [Online: accessed 20 June 2017].

Hopcroft, J. E., Motwani, R., Ullman, J. D., 2006. Introduction to Automata Theory, Languages, and Computation (3rd Edition). Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

Huuck, R., Lukoschus, B., Frehse, G., Engell, S., 2002. Compositional verification of continuous-discrete systems. In: Engell, S., Frehse, G., Schnieder, E. (Eds.), Modelling, Analysis, and Design of Hybrid Systems, Lecture Notes in Control and Information Science. Springer, pp. 225–246.

IEC 61882:2001, 2001. Hazard and operability studies (HAZOP studies) Application guide. Standard, International Electrotechnical Commission.

Koutsoukos, X., Antsaklis, P., Stiver, J., Lemmon, M., 2000. Supervisory control of hybrid systems. Proceedings of the IEEE 88 (7).

Kowalewski, S., 2002. Introduction to the analysis and verification of hybrid systems. In: Engell, S., Frehse, G., Schnieder, E. (Eds.), Modelling, Analysis, and Design of Hybrid Systems, Lecture Notes in Control and Information Science. Springer, pp. 153–173.

Kumar, R., Garg, V., 1995. Modeling and Control of Logical Discrete Event Systems. Springer US.

Lin, F., Wonham, W., December 1988a. Decentralized control and coordination of discrete-event systems. Proceedings of the 27th IEEE Conference on Decision and Control.

Lin, F., Wonham, W., 1988b. Decentralized supervisory control of discrete-event systems. Information Sciences 44 (3), 199 – 224.
URL http://www.sciencedirect.com/science/article/pii/0020025588900023

Lunze, J., Mar. 1994. Qualitative modelling of linear dynamical systems with quantized state measurements. Automatica 30 (3), 417–431.
URL http://dx.doi.org/10.1016/0005-1098(94)90119-8

Lunze, J., 1999. A timed discrete-event abstraction of continuous-variable systems. International Journal of Control 72 (13), 1147 – 1164.

Lunze, J., 2000. Diagnosis of quantised systems by means of timed discrete-event representations. In: Lynch, N., Krogh, B. H. (Eds.), Hybrid Systems: Computation and Control: Third International Workshop, HSCC 2000 Pittsburgh, PA, USA, March 23–25, 2000 Proceedings. Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 258–271.
URL http://dx.doi.org/10.1007/3-540-46430-1_23

Lunze, J., Raisch, J., 2002. Discrete models for hybrid systems. In: Engell, S., Frehse, G., Schnieder, E. (Eds.), Modelling, Analysis, and Design of Hybrid Systems, Lecture Notes in Control and Information Science. Springer, pp. 76–80.

Lunze, J., Steffen, T., 2002. Hybrid reconfigurable control. In: Engell, S., Frehse, G., Schnieder, E. (Eds.), Modelling, Analysis, and Design of Hybrid Systems, Lecture Notes in Control and Information Science. Springer, pp. 267–284.

Philips, P., Heemels, W., Preisig, H., Bosch, P. V. D., 2003. Control of quantized systems based on discrete event models. International Journal of Control 76 (3), 277 – 294.

Philips, P., Weiss, M., Preisig, H. A., 1999a. Control based on discrete-event models of continuous systems. Proceedings of the European Control Conference, Karlsruhe, Germany.

Philips, P., Weiss, M., Preisig, H. A., 1999b. A design strategy for discrete control of continuous systems. Proceedings of the 1999 American Control Conference, San Diego, USA, 2097–2101.

Philips, P. P., 2001. Modelling, control and fault detection of discretely-observed systems. Ph.D. thesis, TU Eindhoven, Eindhoven, The Netherlands.

Preisig, H. A., 1996. A mathematical approach to discrete-event dynamic modelling of hybrid systems. Computers & Chemical Engineering 20, S1301 – S1306.
URL http://www.sciencedirect.com/science/article/pii/0098135496002244

Preisig, H. A., Manenti, F., 2012. HAZOP an automaton-inspired approach. In: Proceedings of the 22nd European Symposium on Computer Aided Process Engineering, 17 20 June 2012, London.

Ramadge, P., Wonham, W., January 1987a. The control of discrete event systems. Proceedings of the IEEE 77 (1), 81–98.

Ramadge, P., Wonham, W., 1987b. Supervisory control of a class of discrete event processes. Control and Optimization 25 (1), 206–230.

Reisig, W., 1985. Petri Nets: An Introduction. Springer-Verlag New York, Inc., New York, NY, USA.

Santina, M. S., Stubberud, A. R., Hostetter, G. H., 2010. Discrete-Time Equivalents to Continuous-Time Systems. In: Levine, W. S. (Ed.), The Control Handbook. CRC Press, Ch. 13.

Skoldstam, M., Akesson, K., Fabian, M., 2007. Modeling o f Discrete Event Systems using Finite Automata With Variables. In: Proceedings of the 46th IEEE Conference on Decision and Control, 12 14 Dec 2007, New Orleans, LA, USA.

Srinivasan, R., Venkatasubramanian, V., 1996. Petri net-Digraph models for automating HAZOP analysis of batch process plants. Computers & Chemical Engineering 20, S719 – S725.
URL http://www.sciencedirect.com/science/article/pii/0098135496001299

Stiver, J., Antsaklis, P., Lemmon, M., 1996. A logical des approach to the design of hybrid control systems. Mathematical and Computer Modelling 23 (11), 55 – 76.
URL http://www.sciencedirect.com/science/article/pii/0895717796000647

Thistle, J., 1996. Supervisory control of discrete event systems. Mathematical and Computer Modelling 23 (11), 25 – 53.
URL http://www.sciencedirect.com/science/article/pii/0895717796000635

Thomas Marlin, 2014. Safety. In: Operability in process design: Achieving safe, profitable, and robust process operations. McMaster University, Ch. 5.

Thombre, M. N., Preisig, H. A., October 2017. Use of discrete-event dynamic systems for hazop analysis. Proceedings of the 27 th European Symposium on Computer Aided Process Engineering ESCAPE 27.

Tousi, M. M., Karuei, I., Hashtrudi-Zad, S., Aghdam, A. G., 2008. Supervisory control of switching control systems. Systems & Control Letters 57 (2), 132 – 141.
URL http://www.sciencedirect.com/science/article/pii/S0167691107001077

Tripakis, S., 2002. Fault Diagnosis for Timed Automata. In: Damm, W., Olderog, E. R. (Eds.), Formal Techniques in Real-Time and Fault-Tolerant Systems: 7th International Symposium, FTRTFT 2002 Co-sponsored by IFIP WG 2.2 Oldenburg, Germany, September 9–12, 2002 Proceedings. Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 205–221.
URL http://dx.doi.org/10.1007/3-540-45739-9_14

Tripakis, S., Dang, T., 2009. Modeling, verification and testing using timed and hybrid automata. In: Nicolescu, G., Mosterman, P. J. (Eds.), Model-Based Design for Embedded Systems. CRC Press.

Ushio, T., Takai, S., 2009. Supervisory control of discrete event systems modeled by mealy automata with nondeterministic output functions. Transactions of the Institute of Systems, Control and Information Engineers 22 (4), 154–160.

Wang, F., 2006. Symbolic implementation of model-checking probabilistic timed automata. Ph.D. thesis, The University of Birmingham, Birmingham, United Kingdom.

Wong, K. C., Wonham, W. M., 1996. Hierarchical control of discrete-event systems. Discrete Event Dynamic Systems 6 (3), 241–273.
URL http://dx.doi.org/10.1007/BF01797154

Wonham, W. M., 1989. On the control of discrete-event systems. In: Nijmeijer, H., Schumacher, J. M. (Eds.), Three Decades of Mathematical System Theory: A Collection of Surveys at the Occasion of the 50th Birthday of Jan C. Willems. Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 542–562.
URL http://dx.doi.org/10.1007/BFb0008476

Xi, Y.-X., Lim, K.-W., Ho, W.-K., Preisig, H. A., 2001. Fault diagnosis using dynamic finite-state automaton models. In: Industrial Electronics Society, 2001. IECON '01. The 27th Annual Conference of the IEEE. Vol. 1. pp. 484–489 vol.1.

Zhong, H., Wonham, W., October 1990. On the consistency of hierarchical supervision in discrete-event systems. IEEE Transactions on Automatic Control 35 (10), 1125 – 1134.

Appendix A

Brief description of conventional HAZOP

This appendix serves as an overview of how HAZOP studies are currently implemented in the industry. HAZOP stands for Hazard and Operability Studies and these studies were developed in Imperial Chemical Industries (ICI) in the mid-1960s. A standard (IEC 61882:2001, 2001) for the application guide for HAZOP has been established by the International Electrotechnical Commission (IEC), an international standards body. Moreover, Dunj et al. (2010) provides a extensive literature review on the subject.

There are various tools used for Process Hazard Analyses (PHA). These include:

- Checklists
- Fault Tree Analysis
- Fault Modes and Effects Analysis
- HAZOP

Out of these, checklists are a relatively less detailed analysis tool whereas the others are more detailed.

A.1 The basic methodology of HAZOP

As mentioned in Chapter 1, a HAZOP study is a systematic and structured technique to investigate a process, with the objective of identifying potential hazards

and operability issues in the process (Thomas Marlin, 2014). A HAZOP study is performed by a multidisciplinary team. The team aims to look at meaningful (physically possible) deviations from the intended design intention [1] by studying the P&ID diagrams of the process. A key prerequisite of a good HAZOP study is that the process design, described by the P&ID diagrams must be firm. The team concentrates on the deviations that could lead to potential hazards to safety, health or the environment (Crawley et al., 2000). The team also looks for deviations in operability conditions, in addition to the hazards. This is because operability issues, although not hazardous, may affect factors such as security and economic profitability.

The team, using intuition, judgment and experience, evaluates the consequences of those deviations for which a cause can be determined. The consequences are then measured up against the existing safety mechanisms. If the implemented safety measures are found to be inadequate, the team makes a written record recommending a change or calling for a more focused investigation of the problem. A good HAZOP study also takes into account the changes that might happen during the lifetime of operation and attempts to identify the problems that may occur at a later stage in the plant life, or those caused by human error (Crawley et al., 2000).

It is important to note that HAZOP does not implement solutions *per se*, but rather allows engineers to identify potential hazards in the design phase of a process plant such that adequate safety performance is ensured. The knowledge obtained from HAZOP can be very useful in implementing appropriate solutions to ensure plant safety.

A.2 The HAZOP study procedure

This section describes the step-by-step HAZOP procedure, shown in Figure A.1.

The first step in the procedure is to formulate a detailed design intention. This refers not only to the conditions on the process variables but also on other factors such as plant equipment, materials, control structures, etc.

The next step is to introduce a meaningful deviation. This is done by combining a *guideword* and a parameter. Some guidewords and their meaning are shown in Table A.1. Parameters may include flow, pressure, temperature, viscosity, composition, phase, etc. A deviation may be introduced by taking a parameter and combining it with a guideword too see if a meaningful deviation occurs. This is referred to as the parameter-first approach. The counter approach is to take a guideword and apply it to each parameter. This is the guideword-first approach.

[1] This design intention is usually more limiting than the physical design conditions. The idea of the intended design intention more descriptive than, but similar to, that of the safe operability region shown in Figure 4.1.

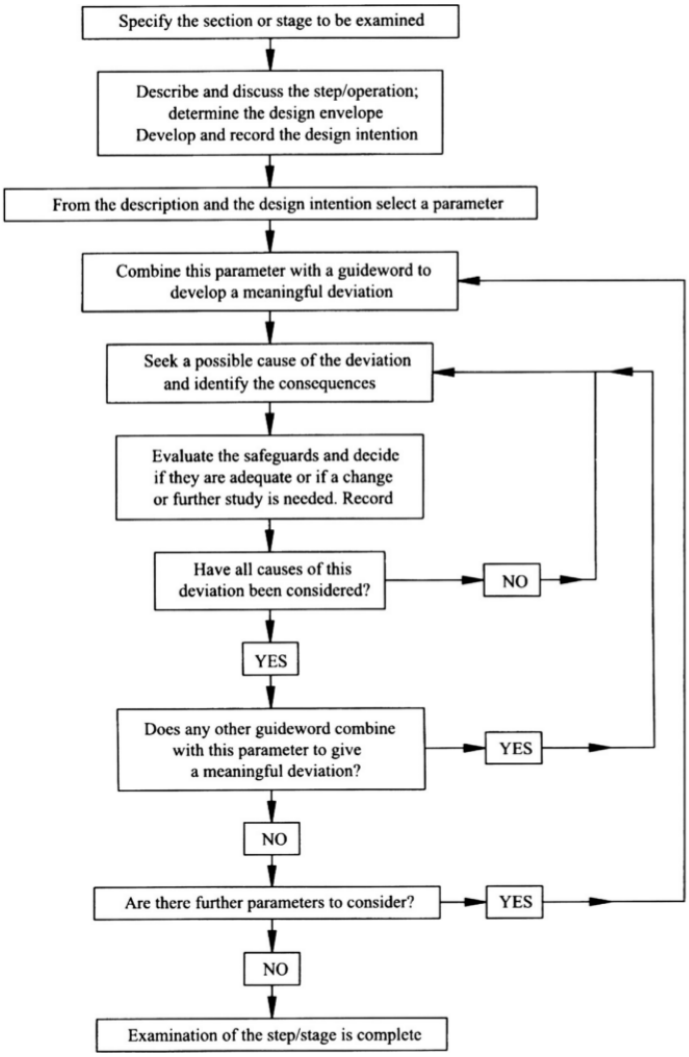


Figure A.1: HAZOP study procedure for a section of operation (Crawley et al., 2000).

Guideword	Meaning
No (not, none)	None of the design intent is achieved
More (more of, higher)	Quantitative increase in a parameter
Less (less of, lower)	Quantitative decrease in a parameter
As well as (more than)	An additional activity occurs
Part of	Only some of the design intention is achieved
Reverse	Logical opposite of the design intention occurs
Other than (other)	Complete substitution another activity takes place OR an unusual activity occurs or uncommon condition exists

Table A.1: Some guidewords and their meanings (Crawley et al., 2000).

Parameter	Applicable Guidewords
Flow	No, more, less, reverse
Temperature	Higher, lower
Pressure	Higher, lower
Composition	No, more of, less of, more than, other than
Phase	No, more of, less of, more than, other than
Time Sequence	Sooner, later, longer, shorter

Table A.2: Some HAZOP parameter and guidewords (Thomas Marlin, 2014).

An example of the former is shown in Table A.2. It is obvious than not every guideword can be combined with a parameter to result in a meaningful deviation, and these combinations may be avoided.

After identifying a meaningful deviation, the next step is to identify the cause of that deviation. If the consequences are trivial, the team need not look for causes. The triviality or the non-triviality of the consequences can be made by evaluating the acceptable risk, since it involves both frequency and severity. In some cases, it might be possible to group causes together, but this should only be done when the team is sure that the consequences are same for every cause (Crawley et al., 2000).

Next, the consequences associated with each cause have to be studied to check whether they take the system outside intended operation range (Crawley et al., 2000). This analysis of consequences must include those that are immediate or delayed, inside or outside the plant section that is under study. The team should be especially careful to note consequences that appear 'farther' from the causes, in a physical sense.

The next step in the procedure is to analyze the consequences against safeguards. One way of doing this is to ignore the existence of existing safety mecha-

nism, analyze the worst possible consequences and then measure them up against the existing safeguards. Another approach is to analyze the consequences by keeping in mind the existing safety mechanisms.

Once the hazards or operability problems are identified, the team may refer the problem for further investigation. If the team can unanimously agree on a proposed solution and if they are within their authority to do so, they may make the recommendation in the written report. The recommended actions may be generic or specific, but should be unambiguous and non-repetitive (Crawley et al., 2000). A complete record of the conclusions of the HAZOP study should be made by the team. The record is typically in form of a detailed form, an example of which is shown in Table A.3.

All these steps are looped first through all corresponding guidewords corresponding to a parameter and then through each individual parameter (in context of the parameter-first approach). Once this is done, the HAZOP study for that section of operation is considered complete.

Company: XYZ Polymer Limited				Facility: Hamilton Works	
Design Intent: Raise circulating oil stream temperature flowing at 100 m^3/h from 250 to 400 °C				HAZOP Team Members:	
Drawing: Figure 5.20					Date: Jan 2, 2011
1.0 Node: Pipe after feed pump before entering heater					
Parameter: Flow					
ID. No.	Guideword / Deviation	Causes	Consequences	Safeguards/ checks	Actions
1.1	No Flow	a. pump motor failure	a. Fluid in pipe being overheated pipe metal overheated and damaged Pipe bursting and releasing oil into the firebox (in contact with flame) Shutdown and loss of production	a. Reliable power supply to motor low flow alarm	a. feed flow sensor and SIS on low flow • Close fuel valves • Open air valve • Alarm with SIS • Manual reset • Short delay to guard against noise • Manual activation of SIS possible • Open stack damper Low flow alarm using controller sensor
		b. coupling failure	b. Hazard from metal pieces at high velocity		b. Install guard over coupling
		c. feed valve closure	See (a) above	c. Flow controller, valve fail open	See (a) above
		d. Foreign material blocking flow	See (a) above	d. Filter upstream in process	See (a) above
		e. No fluid available to pump	See (a) above	e. Level low alarm on vessel supplying pump	See (a) above
1.2	Further deviations …..				

Table A.3: Typical HAZOP form (Thomas Marlin, 2014).

A.3 Limitations of conventional HAZOP

Some of the limitations of conventional HAZOP are as follows (Thomas Marlin, 2014):

- The procedure might not identify a low-frequency, high-consequence hazard caused by multiple, simultaneous failures
- The risks are not quantitatively estimated; thus, considerable judgment is required in deciding the actions
- Hazards may not be identified for a process fault that influences a nearby process
- Since HAZOP is typically performed on finished designs or operating processes, fundamental changes to chemistry or equipment is usually not possible, without incurring large costs
- The team may tend to provide overly complex safety barriers, especially control and SIS systems, that could have low reliability. Also, they might recommend a large number of alarms
- HAZOP does not evaluate chronic hazards

Additionally, other aspects of HAZOP such as the organizational (team dynamics, etc.), managerial (meeting the client, etc.) and company procedures are covered in literature (Crawley et al., 2000; IEC 61882:2001, 2001).

Appendix B

Python codes

B.1 Computing automaton

```python
"""
B{Course:}      TKP4550 Master's Thesis
B{Task:}        Computes an automaton w.r.t a stateindex given intersection points and stateindex

@author       : Mandar Thombre
@contact      : mandart@stud.ntnu.no
@organization: Department of Chemical Engineering, NTNU, Norway
@since        : Nov 1, 2016
@change       : June 15, 2017
@version      : 1.2
@todo         : see comments at the bottom
@requires     : Python v2.7
"""


import numpy as np
import copy

## Assumption! For now, I assume I have all the points of intersection between hypercubes and equilibrium
## hypersurfaces, and they are not on intersection of gridlines

class Grid:

  def __init__(self, no_of_states, boundaries):
    self.no_of_states = no_of_states
    self.boundaries = boundaries
    self.partitions = [len(_)-1 for _ in boundaries]
```

```
def ComputeGridSquare(self, x):
    '''
    Computes the grid square(hypercube) that a given point x lies in
    Indexing of grid in this representation is {(1..n1) X (1..n2) X (1..n3) X ....}
    '''
    gridindex = [0]*self.no_of_states

    for stateindex in range(self.no_of_states):
        if min(self.boundaries[stateindex]) < x[stateindex] < max(self.boundaries[stateindex]):
            gridindex[stateindex] = np.searchsorted(self.boundaries[stateindex],x[stateindex])
        else:
            raise ValueError("Point outside grid!")

    return gridindex


def NextGridSquare(self, gridindex, stateindex):
    '''
    Takes to next grid square (+ve) in the given stateindex
    '''

    newgridindex = copy.deepcopy(gridindex)
    if gridindex[stateindex] < self.partitions[stateindex]:
        newgridindex[stateindex] = newgridindex[stateindex]+1
    else:
        raise ValueError("Can't go forward! Grid in this state direction ends here!")

    return newgridindex

def PrevGridSquare(self, gridindex, stateindex):
    '''
    Takes to previous grid square (-ve) in the given stateindex
    '''

    newgridindex = copy.deepcopy(gridindex)
    if gridindex[stateindex] > 1:
        newgridindex[stateindex] = newgridindex[stateindex]-1
    else:
        raise ValueError("Can't go backword! Grid in this state direction ends here!")

    return newgridindex
```

```
  def PullApart(self, x):
    '''
    Pulls apart the given point x in each state direction
    '''
    epsilon = 1e-12 #some very small number

    xplus =  [0]*self.no_of_states
    xminus = [0]*self.no_of_states

    for stateindex in range(self.no_of_states):
      if x[stateindex] == min(self.boundaries[stateindex]):
        xplus[stateindex]  = x[stateindex]+epsilon
        xminus[stateindex] = x[stateindex]+epsilon
      if x[stateindex] == max(self.boundaries[stateindex]):
        xplus[stateindex]  = x[stateindex]-epsilon
        xminus[stateindex] = x[stateindex]-epsilon
      if min(self.boundaries[stateindex]) < x[stateindex] < max(
    self.boundaries[stateindex]):
        xplus[stateindex]  = x[stateindex]+epsilon
        xminus[stateindex] = x[stateindex]-epsilon

    return xplus, xminus


def cartesian(arrays, out=None):
    """
    Generate a cartesian product of input arrays.
    """

    arrays = [np.asarray(x) for x in arrays]
    dtype = arrays[0].dtype

    n = np.prod([x.size for x in arrays])
    if out is None:
        out = np.zeros([n, len(arrays)], dtype=dtype)

    m = n / arrays[0].size
    out[:,0] = np.repeat(arrays[0], m)
    if arrays[1:]:
        cartesian(arrays[1:], out=out[0:m,1:])
        for j in xrange(1, arrays[0].size):
            out[j*m:(j+1)*m,1:] = out[0:m,1:]
    return out

if __name__ == '__main__':

  ## 3D system
  boundaries1 = [0,1,2,3]
  boundaries2 = [0,1,2,3,4]
```

```
boundaries3 = [0,1,2]

g = Grid(3,[boundaries1,boundaries2, boundaries3])

automaton     = np.zeros(g.partitions)
type3squares = []

## intersection points
intpoints     = [[1.5,4,0], [1.5,3,0], [1,2.5,0], [1.5,4,1],
  [1.5,3,1], [1,2.5,1], [1.5,4,2], [1.5,3,2], [1,2.5,2]]
pulledpoints = []

for point in intpoints:
  xplus, xminus = g.PullApart(point)
  pulledpoints.append(xplus)
  pulledpoints.append(xminus)

for point in pulledpoints:
  gridindex = g.ComputeGridSquare(point)
  if gridindex not in type3squares:
    type3squares.append(gridindex)

  grid_python_index  = [i-1 for i in gridindex] #for indexing in
   python, 0 start
  automaton[tuple(grid_python_index)] = 3

## Automaton based on stateindex = 0
## This part won't fill up any array in the direction of
  stateindex that does not have a type3square
automaton_stateindex = 0
for gsq in type3squares:
  tempgsq = copy.deepcopy(gsq)
  while tempgsq[automaton_stateindex] < g.partitions[
  automaton_stateindex]:
    tempgsq = g.NextGridSquare(tempgsq, automaton_stateindex)
    nextgsq_python_index  = [i-1 for i in tempgsq]
    if automaton[tuple(nextgsq_python_index)] == 0:
      automaton[tuple(nextgsq_python_index)] = 2

  tempgsq = copy.deepcopy(gsq)
  while tempgsq[automaton_stateindex] > 1:
    tempgsq = g.PrevGridSquare(tempgsq, automaton_stateindex)
    prevgsq_python_index  = [i-1 for i in tempgsq]
    if automaton[tuple(prevgsq_python_index)] == 0:
      automaton[tuple(prevgsq_python_index)] = 1


## Now we fill up all the empty arrays
allsquares    = cartesian([range(1,g.partitions[i]+1) for i in
```

```
    range(g.no_of_states)])
  zerosquares  = []

  for gsq in allsquares:
    gsq_python_index = [i-1 for i in gsq]
    if automaton[tuple(gsq_python_index)] == 0:
      zerosquares.append(gsq)

  iflag = 0
  motherload = None
  for gsq in zerosquares:
    for stateindex in range(g.no_of_states):

      tempgsq = copy.deepcopy(gsq)
      while tempgsq[stateindex] < g.partitions[stateindex]:
        tempgsq = g.NextGridSquare(tempgsq, stateindex)
        nextgsq_python_index  = [i-1 for i in tempgsq] # for
    python

        if automaton[tuple(nextgsq_python_index)] == 3:
          break

        if automaton[tuple(nextgsq_python_index)] != 0 and
    automaton[tuple(nextgsq_python_index)] != 3:
          fillall = automaton[tuple(nextgsq_python_index)]
          iflag = 1
          break

      if iflag == 1:
        break

    if iflag == 1:
      break

      tempgsq = copy.deepcopy(gsq)
      while tempgsq[stateindex] > 1:
        tempgsq = g.PrevGridSquare(tempgsq, stateindex)
        prevgsq_python_index  = [i-1 for i in tempgsq] # for
    python

        if automaton[tuple(nextgsq_python_index)] == 3:
          break

        if automaton[tuple(prevgsq_python_index)] != 0 and
    automaton[tuple(prevgsq_python_index)] != 3:
          fillall = automaton[tuple(prevgsq_python_index)]
          iflag = 1
          break
      if iflag == 1:
```

```
        break
    if iflag == 1:
      break

  for gsq in zerosquares:
    gsq_python_index = [i-1 for i in gsq]
    automaton[tuple(gsq_python_index)] = fillall

  print automaton

  '''
  The presence of "nan" in the automaton indicates that the equilibrium hypersurface is "parallel"
  or "close to parallel" to direction of chosen stateindex, since automaton computation is based
  only on intersection points (and corresponding grid hypercubes) and not the surface itself.

  This happens when, before the fillall calculation, arrays along a stateindex are either all 3 or all 0.
  Knowledge of the hypersurface may be exploited here. For example, the normal from any point in any of the
  'zerosquares' to the hypersurface may be calculated and the sign of normal would determine the state directionality
  of all zerosquares (1 or 2). This is straightforward for linear hypersurfaces but trickier for non linear hypersurfaces
  (future implementation)
  '''
```

B.2 Dijkstra's algorithm

This Python code for the Dijkstra's algorithm is taken from Eppstein (2002) and modified according to the examples used in this thesis. The copyright of the code remains with the author, David Eppstein.

```
# Dijkstra's algorithm for shortest paths
# David Eppstein, UC Irvine, 4 April 2002

# http://aspn.activestate.com/ASPN/Cookbook/Python/Recipe/117228
from __future__ import generators

class priorityDictionary(dict):
    def __init__(self):
        '''Initialize priorityDictionary by creating binary heap
of pairs (value,key).  Note that changing or removing a dict entry will
not remove the old pair from the heap until it is found by smallest() or
until the heap is rebuilt.'''
        self.__heap = []
        dict.__init__(self)

    def smallest(self):
        '''Find smallest item after removing deleted items from heap.'''
        if len(self) == 0:
            raise IndexError, "smallest of empty priorityDictionary"
        heap = self.__heap
        while heap[0][1] not in self or self[heap[0][1]] != heap[0][0]:
            lastItem = heap.pop()
            insertionPoint = 0
            while 1:
                smallChild = 2*insertionPoint+1
                if smallChild+1 < len(heap) and \
                        heap[smallChild] > heap[smallChild+1]:
                    smallChild += 1
                if smallChild >= len(heap) or lastItem <= heap[smallChild]:
                    heap[insertionPoint] = lastItem
                    break
                heap[insertionPoint] = heap[smallChild]
                insertionPoint = smallChild
        return heap[0][1]

    def __iter__(self):
```

```
        '''Create destructive sorted iterator of priorityDictionary.'''
        def iterfn():
            while len(self) > 0:
                x = self.smallest()
                yield x
                del self[x]
        return iterfn()

    def __setitem__(self,key,val):
        '''Change value stored in dictionary and add corresponding pair to heap.  Rebuilds the heap if the number of deleted items grows
too large, to avoid memory leakage.'''
        dict.__setitem__(self,key,val)
        heap = self.__heap
        if len(heap) > 2 * len(self):
            self.__heap = [(v,k) for k,v in self.iteritems()]
            self.__heap.sort()  # builtin sort likely faster than O(n) heapify
        else:
            newPair = (val,key)
            insertionPoint = len(heap)
            heap.append(None)
            while insertionPoint > 0 and \
                    newPair < heap[(insertionPoint-1)//2]:
                heap[insertionPoint] = heap[(insertionPoint-1)//2]
                insertionPoint = (insertionPoint-1)//2
            heap[insertionPoint] = newPair

    def setdefault(self,key,val):
        '''Reimplement setdefault to call our customized __setitem__.'''
        if key not in self:
            self[key] = val
        return self[key]

def Dijkstra(G,start,end=None):
  """
  Find shortest paths from the start vertex to all
  vertices nearer than or equal to the end.

  The input graph G is assumed to have the following
  representation: A vertex can be any object that can
  be used as an index into a dictionary.  G is a
  dictionary, indexed by vertices.  For any vertex v,
  G[v] is itself a dictionary, indexed by the neighbors
  of v.  For any edge v->w, G[v][w] is the length of
  the edge.  This is related to the representation in
```

```
<http://www.python.org/doc/essays/graphs.html>
where Guido van Rossum suggests representing graphs
as dictionaries mapping vertices to lists of neighbors,
however dictionaries of edges have many advantages
over lists: they can store extra information (here,
the lengths), they support fast existence tests,
and they allow easy modification of the graph by edge
insertion and removal.  Such modifications are not
needed here but are important in other graph algorithms.
Since dictionaries obey iterator protocol, a graph
represented as described here could be handed without
modification to an algorithm using Guido's representation.

Of course, G and G[v] need not be Python dict objects;
they can be any other object that obeys dict protocol,
for instance a wrapper in which vertices are URLs
and a call to G[v] loads the web page and finds its links.

The output is a pair (D,P) where D[v] is the distance
from start to v and P[v] is the predecessor of v along
the shortest path from s to v.

Dijkstra's algorithm is only guaranteed to work correctly
when all edge lengths are positive. This code does not
verify this property for all edges (only the edges seen
before the end vertex is reached), but will correctly
compute shortest paths even for some graphs with negative
edges, and will raise an exception if it discovers that
a negative edge has caused it to make a mistake.
"""

D = {}  # dictionary of final distances
P = {}  # dictionary of predecessors
Q = priorityDictionary()   # est.dist. of non-final vert.
Q[start] = 0

for v in Q:
  D[v] = Q[v]
  if v == end: break

  for w in G[v]:
    vwLength = D[v] + G[v][w]
    if w in D:
      if vwLength < D[w]:
        raise ValueError, \
"Dijkstra: found better path to already-final vertex"
    elif w not in Q or vwLength < Q[w]:
      Q[w] = vwLength
      P[w] = v
```

```
  return (D,P)

def shortestPath(G,start,end):
  """
  Find a single shortest path from the given start vertex
  to the given end vertex.
  The input has the same conventions as Dijkstra().
  The output is a list of the vertices in order along
  the shortest path.
  """

  D,P = Dijkstra(G,start,end)
  Path = []
  while 1:
    Path.append(end)
    if end == start: break
    end = P[end]
  Path.reverse()
  return Path

# Simple reachability example
SimpleR  = {'x1':{'x2':5                    },
       'x2':{'x6':2                      },
       'x3':{'x2':0,  'x7':6             },
       'x4':{'x3':1                      },
       'x5':{'x1':11, 'x6':4             },
       'x6':{'x1':5,  'x7':2             },
       'x7':{'x4':2,  'x8':3,  'x11':0},
       'x8':{'x4':1,  'x12':4            },
       'x9':{'x5':2,  'x10':3            },
       'x10':{'x6':1,  'x7':2,  'x11':0},
       'x11':{'x4':2                     },
       'x12':{'x11':1                    },
    }

# Two Tanks example
TwoTanks = {'h11':{'h21':1,'h12':1},
      'h21':{'h31':1,'h22':1},
      'h31':{'h41':1,'h32':1},
      'h41':{'h51':1,'h42':1},
      'h51':{'h52':1},

      'h12':{'h22':1,'h13':1,'h11':1},
      'h22':{'h32':1,'h23':1,'h21':1},
      'h32':{'h42':1,'h33':1,'h31':1},
      'h42':{'h52':1,'h43':1,'h41':1, 'h32':1},
      'h52':{'h53':1,'h51':1,'h42':1},

```

```
        'h13':{'h23':1,'h14':1,'h12':1},
        'h23':{'h33':1,'h24':1,'h22':1},
        'h33':{'h43':1,'h34':1,'h32':1},
        'h43':{'h53':1,'h44':1,'h42':1},
        'h53':{'h54':1,'h52':1},

        'h14':{'h24':1,'h15':1,'h13':1},
        'h24':{'h34':1,'h25':1,'h23':1},
        'h34':{'h44':1,'h35':1,'h33':1},
        'h44':{'h54':1,'h45':1,'h43':1},
        'h54':{'h55':1,'h53':1},

        'h15':{'h25':1,'h14':1},
        'h25':{'h35':1,'h24':1},
        'h35':{'h45':1,'h34':1},
        'h45':{'h55':1,'h44':1},
        'h55':{'h54':1},

    }

print shortestPath(SimpleR, 'x5', 'x2')
print shortestPath(TwoTanks, 'h55', 'h44')
```