NTNU

Department of Engineering Cybernetics

Multibody Flow Networks

John Eivind Rømma Helset

Supervisor: Sverre Hendseth

Jun 06, 2022

I would like to thank the academic necromancy of Ellen Beate Hove, without whom I would not have been able to finish my degree.

Abstract

The decision making process of hydropower production planning is supported by a suite of computations, many of which involve a watercourse model. Models of tunnel system flow used in this context commonly assume that each tunnel system is inhabitated by a single body of water. Furthermore that tunnel system networks conform to some fixed topology. These assumptions can, for some computations, lead to inadequate results.

Continuous models based on differential algebraic equations enables a modeller to describe flow through tunnel networks without the assumption of a fixed topology. The hybrid system theoretical framework of hybrid automata, whose continuous dynamics are described in terms of differential algebraic equations, enables a modeller to directly describe arbitrary distributions of waterbodies in tunnel networks; to model a multibody flow network. This in turn enables more complex patterns of flow between the reservoirs connected to a tunnel network.

A multibody flow network model can be constructed at a high enough level of detail to serve as a basis for tunnel system flow models in the computational models used in the context of hydrowpower production planning. This would be a more efficient construction, than constructing every computational model independently.

In this report the modelling framework of hybrid automata is refined into two new frameworks: structured discrete automata, and structured hybrid automata. These frameworks are tailored to the construction of hybrid automata, with a high level of discrete detail, such as a multibody flow network. The framework of structured discrete automata is then used to construct a discrete model of a multibody flow network. This model can be used as a basis for constructing a hybrid model, in the form of a structured hybrid automaton, of a multibody flow network.

Sammendrag

Produksjonsplanlegging av vannkraft er understøtta av forskjellige beregninger. Mange av beregningsmodellene legger til grunn en modell av et vassdrag. Modellene som brukes av tunnellsystem er ofte lagd med en antagelse om at en enkelt vannkropp har tilhold i tunnellen, samt at tunnellsystemet har en spesifikk topologisk struktur.

Kontinuerlige modeller basert på differensial-algebraiske likningssett lar derimot en modellmaker beskrive flyt gjennom generelle tunnellnettverk, uten en del topologiske antagelser. Det hybrid-system-teoreriske rammeverket, hybride tilstandsmaskiner, hvis kontinuerlige dynamikk er representert av differensial-algebraiske likningssett, lar en modellmaker direkte beskrive vilkårlige fordelinger av flere vannkropper i et tunnellsystem. Modellen kan derfor representere mer komplekse flytmønster mellom magasinene som er tilkobla tunnellnettverket.

En flerkropps-flyt-nettverk-modell kan settes opp på et høyt detaljnivå, og brukes til å sette opp resten av beregningsmodellene som brukes innafor feltet. Dette er en mer effektiv måte å sette opp modeller på, enn å sette opp hver modell isolert.

I denne rapporten blir rammeverket til den hybride tilstandsmaskinen raffinert til to nye modelleringsrammeverk, kalt strukturerte diskret og hybride tilstandsmaskiner. Disse er skreddersydd for konstruksjon av hybride modeller, som har et høyt diskret detaljnivå, som for eksempel et flerkropps-flyt-nettverk. Rammeverket blir så brukt til å sette opp en diskret modell av et flerkropps-flyt-nettverk, som kan brukes som utgangspunkt for oppsett av en hybrid modell.

CONTENTS

1 Introduction **1**
- 1.1 Context and Motivation . . . 1
- 1.2 Goals . . . 6
- 1.3 Contribution . . . 6

2 Background **9**
- 2.1 Related Modelling . . . 9
- 2.2 Differential Algebraic Equations . . . 12
- 2.3 Hybrid System Theory . . . 16

3 Results **31**
- 3.1 Structured Discrete Automata . . . 31
- 3.2 Structured Hybrid Automata . . . 40
- 3.3 Logical Description of a Flow Network . . . 48
- 3.4 Discrete Flow Network Model . . . 53

4 Discussion **69**
- 4.1 Modelling Framework and Modelling Experience . . . 69
- 4.2 Implementation Experience . . . 70
- 4.3 Future work . . . 71

5 Conclusion **75**

A Notation **77**

B Discrete Model Diagrams **79**
- B.1 Diagrams For Filling The Simple Network Top Down . . . 79
- B.2 Diagrams For Draining The Simple Network Top Down . . . 85

C External Content **91**

D Glossary **93**

Bibliography **95**

Index **99**

CHAPTER ONE

INTRODUCTION

1.1 Context and Motivation

1.1.1 The Tunnel Systems of a Hydropower Producing Watercourse

A hydropower producing watercourse is a complex system that can be represented in terms of multiple connected subsystems consisting of reservoirs, catchments of rainwater and snowmelt, river systems, and tunnel systems. The tunnel systems of such a watercourse are networks of gates, tunnels, units, and valves, where a unit is either a generator or a pump. An example of a hydropower producing watercourse based on the Novle and Røldal hydropower plants is shown as a diagram in Fig. 1.1.

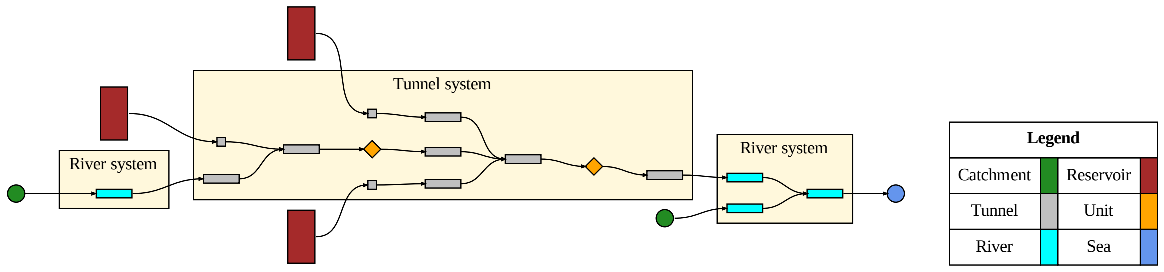

Fig. 1.1: Example hydropower producing topology based on the Novle and Røldal hydropower plants.

A tunnel system can encompass multiple hydropower plants, be connected to multiple catchments, reservoirs and river-systems, and contain multiple waterbodies. These waterbodies can be used independently for production. Consider, for example, a water column resting on the closed left-most unit in Fig. 1.1, while the right-most unit is producing from one or both of the reservoirs connected to its upstream tunnel system.

This report is partly a result of my previous work with existing models of such systems in the context of production planning. These models are often built on two simplifying assumptions. The first assumption is that the topology of the system conforms to a given topological structure. In particular that the tunnel system is turnip-like (see Fig. 1.2). A turnip-like system can be decomposed into an upstream tunnel-tree (the leaves of the turnip), a plant *DAG* (the root of the turnip), and a downstream river or reservoir. The leaf vertices of the tree are reservoirs. The *DAG* has a single source and sink, and consists of trash racks, penstocks, units, and draft-tubes.

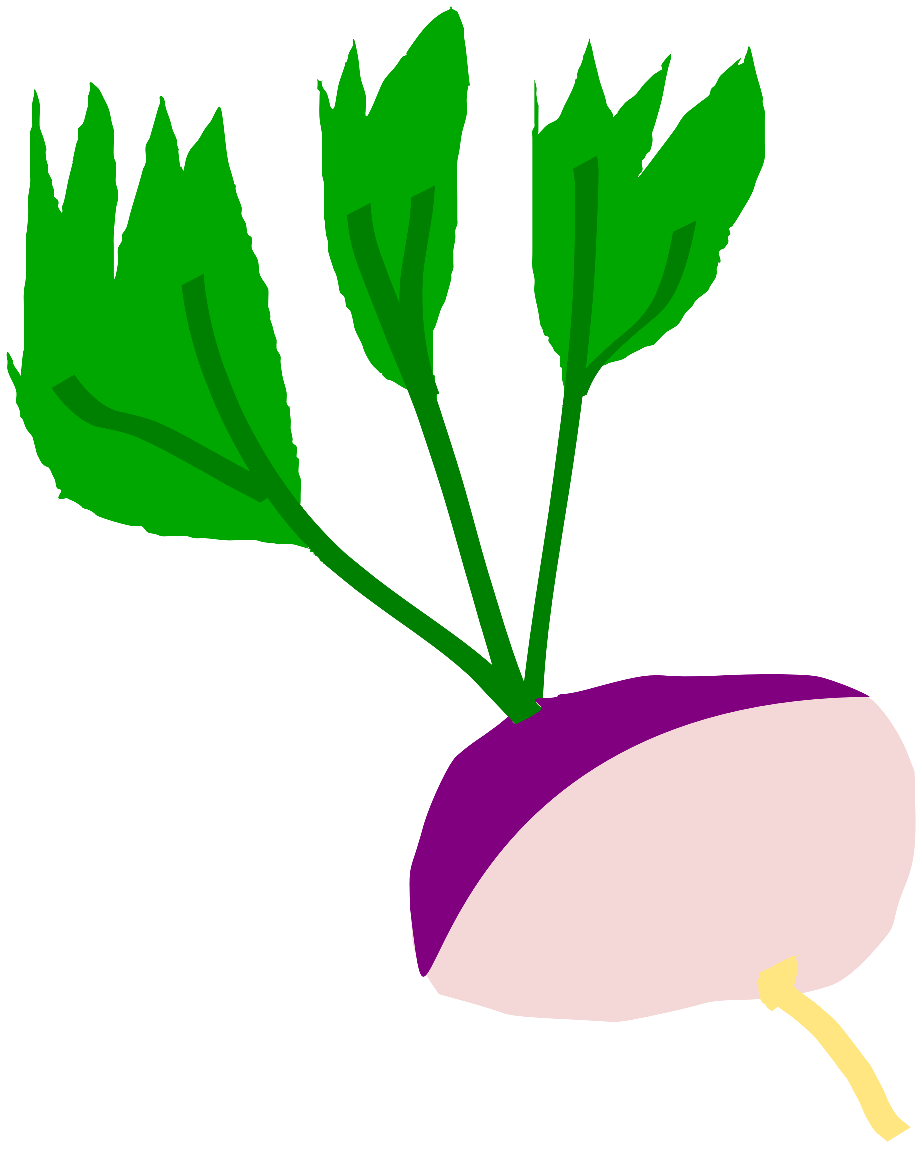

Fig. 1.2: Rendition of a turnip.

An example of a turnip-plant is shown in Fig. 1.3. The upstream tunnel-tree has two reservoirs, the plant has 2 units, and there is a single river downstream. The system in Fig. 1.1 however does not conform to this topological structure. It is difficult to find a topological mould that is generally applicable due to the variety of hydropower plant topologies.

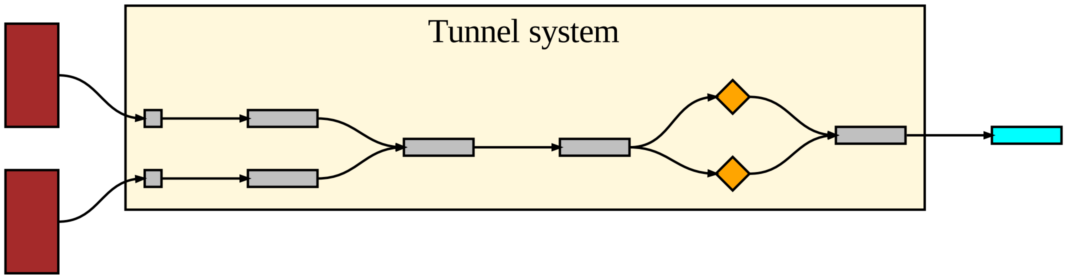

Fig. 1.3: Turnip-like hydropower plant (see legend in Fig. 1.1).

The second assumption is that the tunnel system is fully submerged; that there is one waterbody per tunnel system. The distribution of waterbodies in a tunnel system shape the patterns of flow, and the patterns of flow determine the directions of waterbody growth and recession, that in turn cause waterbodies to merge and split. The two scenarios shown in Fig. 1.4 and Fig. 1.5 illustrate the problem of assuming a fixed waterbody distribution. In the former scenario both reservoirs affect the dynamics of the waterbody in the tunnel system. In the latter scenario only the left-most reservoir does. This effect is compounded by large tunnels which are often found in complex tunnel systems. In [KSS07], a review of hydropower plant models, submersion is not included as a qualitative feature as all the reviewed models assume a fully submerged tunnel system.

Computations based on a fixed topology, or a fixed distribution of waterbodies and fixed patterns of flow, might produce misleading results. This can in turn lead to the implementation of ad-hoc solutions without a formal model. A formal model is an effective form of communication, and an ad-hoc solution will, in comparison, make a computation more difficult to communicate, understand, correctly implement, use, and maintain.

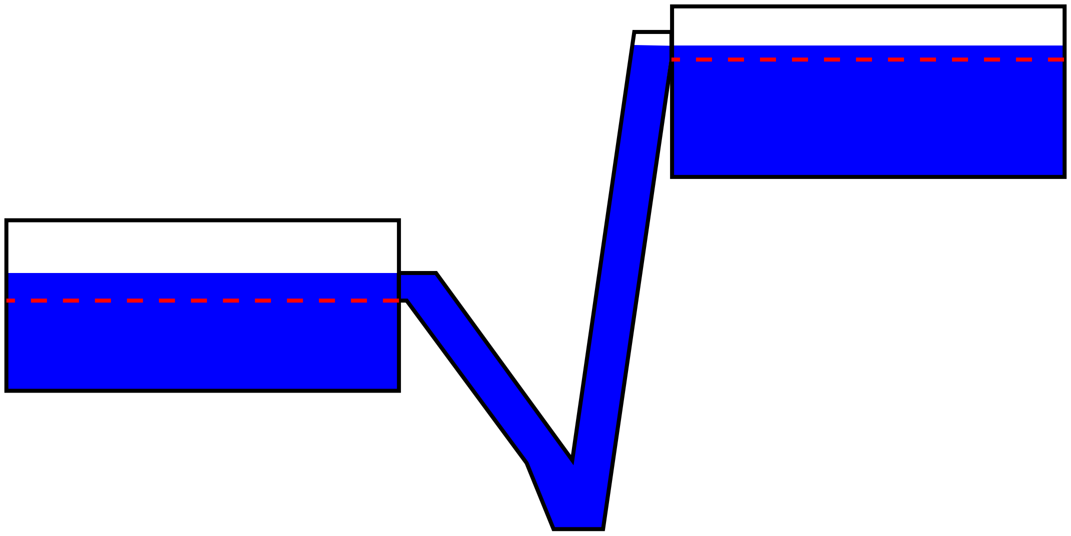

Fig. 1.4: Wet gate configuration.

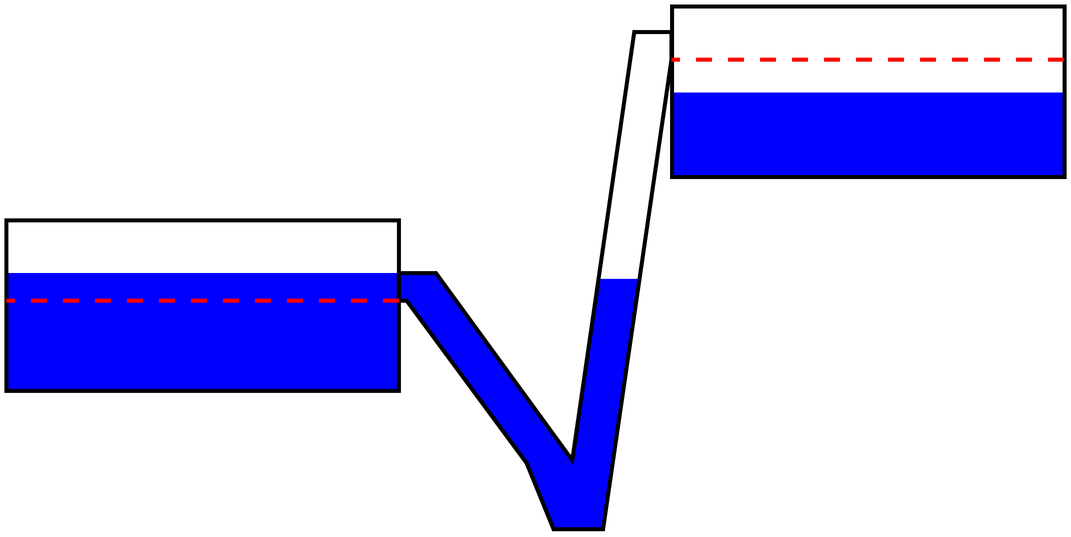

Fig. 1.5: Dry gate configuration.

1.1.2 Modelling and Computation

A mathematical simulation is a coded description of an experiment with a reference (pointer) to the model to which this experiment is to be applied.
- Cellier, [GC91]

The process of modelling concerns itself with the extraction of knowledge from the physical plant to be simulated, organizing that knowledge appropriately, and representing it in some unambiguous fashion. We call the end product of the modelling cycle the model of the system to be simulated.
- Cellier, [CK06]

Modelling and simulation for Cellier, as described and visualised in **Figure 1.4** of [CK06], are coded descriptions of experiments; Cellier considers the simulation in isolation. In [GC91] and [CK06] a simulation, or mathematical simulation, is used in a general sense. In this report the word computation is used instead, since simulation has a more specific meaning in the context of production planning.

Technical labour processes often involve multiple interrelated computations. The models of these computations might represent the same system, be formalised in the same theoretical framework, be subcomputations of one another, and so on. To effectively communicate, design, implement, and maintain the totality of computational software needed to support such a labour process one should build on the mental model of Cellier to represent relations between multiple computations, models, and systems; a model of models.

In Fig. 1.6 the construction of model, as it appears in [CK06], has been deconstructed into a stepwise process involving multiple systems and computations. What was previously a singular model has become a modelling phase. Models materialising on its left side are immediate products of system modelling. These models are constructed using expressive and flexible frameworks, are open and composable, and are constructed at a high enough level of detail to eventually lend their clean, formal hands to any computation that needs them. These left-sided models will be referred to as system models. Models about to exit on the right side of the modelling phase are products of a series of stepwise left-to-right transformations. These models are constructed using computationally tractable frameworks, highly specialised and closed, ready to get their hands dirty in some particular computation. These right-sided models will be referred to as computational models. The people who construct the models on the left side might differ from those constructing the models on the right side, who again might differ from those that implement and maintain the computations, or those that use the computations or work with the actual systems. This underlines the need for clear and unambiguous communication; which in this context and in this report is taken to be synonymous with formal modelling.

The *DAE* and *HA* formalisms are examples of expressive frameworks for continuous and hybrid system modelling respectively. A relevant application of *DAE* is the modelling of general flow networks described in terms of graphs in [JT14]. A *HA* can model temporary flow paths in the same manner that an *MLD* is used in [OM10]. Temporary flow paths can be used to model waterbody distributions and arbitrary flow patterns in tunnel systems. Together these frameworks seem well-suited for constructing a watercourse system model that obviates the assumptions of fixed tunnel system topology and fixed waterbody distribution. These assumptions might still need to make their reappearance as the system model is transformed to one of the many computational models that involve a watercourse.

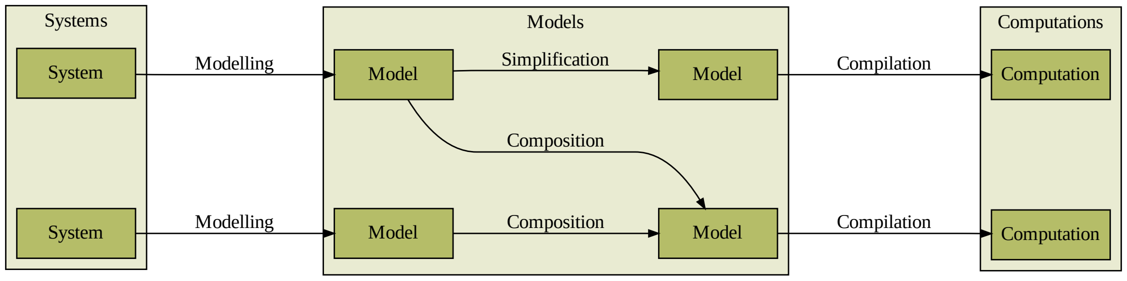

Fig. 1.6: Systems, Models, and Computations.

1.1.3 Computations of Hydropower Production Planning

The planning of hydropower production involves several computations that aid the decision making process of production planners; inflow forecasting, historical inflow estimation, planning, plan simulation, and more. A potential organisation of the construction of their computational models is illustrated in Fig. 1.7. The watercourse model is a reoccurring component and is involved directly in several computational models, albeit at different levels of detail. Reusing a watercourse system model for each of these constructions is more efficient than constructing independent watercourse models.

The corresponding computations of Fig. 1.7 can be described in terms of input and output. Simulation of production plans, roughly speaking, takes as input an inflow forecast, a set of production plans for the plants in the watercourse and computes the trajectories of reservoir-levels, the aggregate movement of water, as well as the production and consumption of power. To model the aggregate movement of water it must be able to represent temporary flow paths caused by flooding or bypassing from reservoirs, similar to the wastewater model of [OM10]. It must also be able to represent arbitrary flow-patterns through tunnel systems, as discussed in *The Tunnel Systems of a Hydropower Producing Watercourse*, and this is where current models often fall short. Estimation of historical inflow takes historical measurements of water levels, consumption and production of power as input, and produces an estimate of historical inflow as output. This is done by computing a steady-state tunnel system flow that matches the historical production and consumption, and then computing the historical inflow based on the mass-balance of reservoirs. Production planning takes, roughly speaking, inflow and price forecasts as inputs and computes an economically efficient production plan for the watercourse for a given period as output(see for example [Skj19]). The watercourse models of historical inflow estimation and production planning can be constructed by doing further simplification of the one used for simulation.

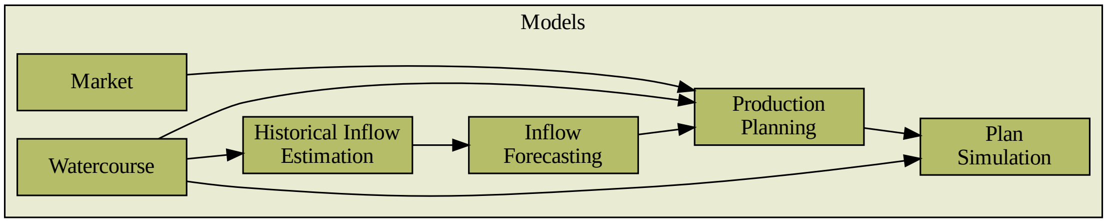

Fig. 1.7: Interrelated Models used in Hydropower Production Planning.

1.2 Goals

The overarching question of this report is how to effectively organise the construction, implementation, use and maintenance of a suite of computations supporting a technical labour process. It is assumed that an efficient construction of these computational models is done through left-to-right transformations of system models. This report focuses on the left side of Fig. 1.6; on the construction of system models and system modelling frameworks. Particular attention will be given to their potential for practical computation. Practical computation is a useful modelling tool for directly testing, debugging, verifying and experimenting with designs. The particular system and labour process that frames this question is that of a hydropower producing watercourse and the associated production planning.

The goal of this report is to construct a system model of a hydropower producing watercourse in a suitable system modelling framework, that can serve as a basis for the efficient construction of the associated computational models, and meet the challenges of topological complexity and waterbody distribution. Such a system model would be efficient both in the conceptual sense of constructing several computational models by applying different transforms to the same system models, and also in the sense of constructing models that can be implemented directly which reaps the benefits of the unambiguous communicative power a formal model, all the way from modeller to megawatt hour.

1.3 Contribution

This report has two contributions that I would like to highlight. Firstly it proposes a new modelling framework called structured automata consisting of structured discrete automata and structured hybrid automata. This hybrid system theoretical framework is yet another take on the hybrid automaton. It is tailored to the construction of hybrid automata with a high level of discrete detail. The execution of a structured hybrid automaton is a formalisation of the event detecting approach to the simulation of a hybrid system, lifting the process of constructing discrete events from continuous roots into the formal definition of a hybrid execution. Models constructed in this framework, and the construction of executions of those models, can be directly represented even as the set of discrete states grows large. Direct representation and computation is a useful tool for debugging, testing, and exploring system models. This new modelling framework and the algorithms described for constructing executions, was implemented in [RH22]. This implementation was used during the course of this report to construct executions, generate figures, and to assist in system modelling. It is another simulator capable of handling variable structure systems in the same vein as [Zim10] and [CN21].

The framework of structured discrete automata was used to define the discrete part of a new hybrid system theoretical model called a multibody flow network. This model combines the flow networks of [JMT15] with the hybrid system theoretical approach to temporary flow paths of [OM10], and is shown to be deterministic and zenofree. A discrete multibody flow network can represent arbitrary distributions of waterbodies and patterns of flow in a network of valves and pipes. It also detects discrete events representing the movement, collisions, and separations of waterbodies. This discrete model is a base on which the continuous dynamics and continuous actions of a hybrid multibody flow model can be constructed.

CHAPTER TWO

BACKGROUND

The background is divided into three parts. The first part, *Related Modelling*, reviews existing models that are interesting, similar, or relevant to the model worked towards in *Results*. The second part, *Differential Algebraic Equations*, introduces the framework that is used to represent continuous dynamics of a *HA* in *Hybrid System Theory*, as well as in *Results*. The third part, *Hybrid System Theory*, provides an overview of some hybrid system theoretical frameworks, and in particular the hybrid automaton, which will be used in *Results*. The simulation of the *HA* defined in *Example (HA Flower System)* was done with using the hybrid system theoretical framework defined and implemented in *Structured Hybrid Automata*.

2.1 Related Modelling

2.1.1 Hydropower Plant Models

The computational models found in the context of production planning of hydropower production are often 0-dimensional in the sense of [Yan19]; their units (and the other elements of the tunnel system) are modelled as vertices or edges in a graph, and their variables are defined in terms of said graph. The structure of the graph is often constrained according to some typical hydropower plant taxonomy, usually some variation of the turnip-like structure discussed in *The Tunnel Systems of a Hydropower Producing Watercourse*. [KSS07] classifies hydropower plant models in terms of the complexity of their continuous dynamics (linearity and elasticity of water columns), as well as the types of components that make up the plant (whether or not the topology includes surge tanks), and assumes a fixed topology.

Short-term Hydro Optimisation

An example of a computational model of a hydropower plant is found in [Skj19], which contains a computational model for short-term optimisation of production plans. The computational model is an optimisation problem (a *MILP*) defined in terms of the discretised steady-state dynamics of the tunnel system. This report is not concerned with economic optimisation or optimal control, only the representation of the topological structure of the tunnel system is of interest. This model assumes turnip-like topology, and separates the tunnel system of the plant from the tunnel system of the upstream reservoir-tree. Plants are considered in isolation, and their tunnel systems are divided into 4 parts, as seen in Fig. 2.1. These are the main tunnel, the penstocks, the units and the outlet. It is assumed that the graph of the hydropower plant has exactly one sink and one source, and that the topology can be decomposed in an upstream and downstream tree, whose leaves are the units of the plant. The two resulting computational models of the reservoir-tree and the plant are later combined to construct the computational model of the entire tunnel system. The two separate models are solved in turn, until a satisfactory output from both are achieved. It is also assumed that there is a single body of water in the entire tunnel system.

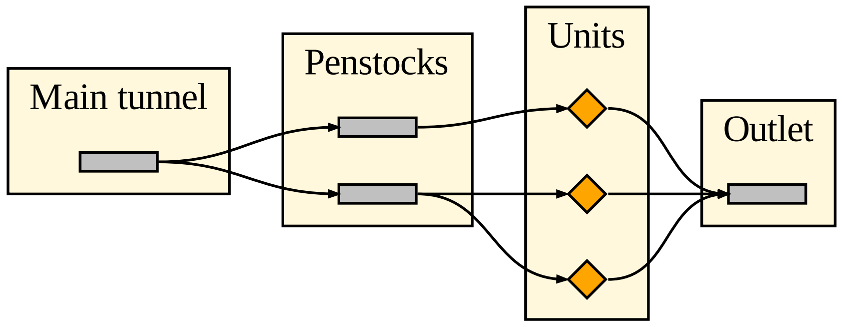

Fig. 2.1: Fixed plant topology with 4 parts.

2.1.2 Wastewater Control

In [OM10] wastewater systems are represented in terms of a typed directed graph which serves as the basis for constructing a hybrid system theoretical model, in particular an *MLD*. The wastewater system is decomposed into subsystems, some of which are modelled with hybrid dynamics. The *MLD* is then used as a basis for *MPC*. Ocampo's wastewater model leveraged hybrid dynamics to model what is called "temporary flow paths", [OM10]:

> "The presence of intense precipitation causes some sewer mains and virtual tanks to surpass their limits. When this happens, any excess above the maximum volume flows to another tank downstream. In this way, temporary flow paths are triggered that depend on the system state and inputs. Since this behaviour is observed in most parts of the sewer network, a modelling methodology is needed that can consider and incorporate overflows and other logical dynamics"

The modelling approach of [OM10] was one of the initial inspirations of the one planned in this report, even though this report will use another hybrid system theoretical framework. The focus in this report is on modelling multiple bodies of water in the tunnel systems of a hydropower producing watercourse. These bodies can also been seen as defining temporary flow paths between the reservoirs and rivers that are connected to the tunnel system. One would need to model the same type of flood behaviour in reservoirs when constructing a model of an entire watercourse, which is needed in by some of the computations of production planning.

As mentioned the hybrid subsystems of [OM10] are defined in terms of the vertices of a graph. There is a one-to-one mapping between subsystem and vertex. The discrete dynamics of a particular subsystem only affects its own continuous dynamics. In this report however, the discrete and continuous description of the system is not overlapping. There is no one-to-one mapping between the elements of the graph and a hybrid subsystem. Instead there is a one-to-many mapping between elements and discrete variables, and similar for continuous variables. The discrete variables associated with the elements of the graph, together, define the discrete dynamics of the entire tunnel system. The discrete state in turn determines the continuous dynamics of the entire tunnel system. The inclusion or exclusion of a particular continuous relation can be dependent on the discrete valuation of more than one discrete variable, even though the discrete variables involved in such a computation are, topologically speaking, close to each other.

2.1.3 Flow Networks

[JT14] presents a unified modelling approach for what they term flow networks. It abstracts from the specific content of the flow. A flow can be a current, water, gas, blood and so on. It represents the system in terms of a directed graph, and then defines the dynamics that govern the systems with reference to this graph. It also introduces switching elements, vertices in the flow network that are able to break or connect flow; elements that can be modeled as hybrid subsystems like in [OM10]. The approach taken in [JT14] is similar to the one taken here, even though this report focuses specifically on water-networks. The model presented here introduces two orthogonal refinements of the models in [JT14], in particular it introduces the discrete representation of the direction of flow, as well as multibody versus single-bodied flow networks. The model in [JT14] is described in terms of *DAE*, this report instead intends to develop the model within a hybrid system theoretical framework where only the continuous dynamics of the model are represented in terms of a *DAE*.

2.1.4 Water Networks

[JMT15] establishes global unique solvability for a what is called a quasi-stationary water network:

> We assume the water network to be dominated by laminar flows, which allows us to consider the water motion as a one-dimensional flow along the length of the pipes. Furthermore we assume a network with significant time-dependent changes of flow, but without hydraulic shocks.

The modelling approach is similar to the one in [JT14]. The continuous dynamics of this model are similar to the ones worked towards in this report. In [JT14] the ports of the model are pressure and demand nodes. The pressure at pressure nodes and flow through demand nodes are exogenous variables. This report only considers pressure nodes, which here are modeled both in terms of discrete and continuous variables describing discrete submersion and continuous pressure. When simulating a full hydropower producing watercourse it is often useful to model demand nodes. In particular catchment flow can enter a hydropower through tunnel inserts, and this flow is sometimes modeled by inflow forecasting. When waterbodies in the tunnel system are explicitly modelled, the need for demand nodes disappears. Instead the demand node can be modelled as a minuscule reservoir outside of the tunnel system, and in the event that the tunnel is full, the catchment flow would flood. The results established in [JMT15] would be useful for defining and analysing the continuous dynamics worked towards in this report.

2.2 Differential Algebraic Equations

A differential algebraic equation (*DAE*) is a collection of differential relations and algebraic constraints. As mentioned in *Modelling and Computation* the *DAE* framework is well suited for systems modelling, and in the words of Stephen L. Campbell, [Cam15]:

> One major reason given for the usefulness of DAEs is that they are the initial way that many complex systems are most naturally modeled. This is especially true in chemical, electrical, and mechanical engineering and with models formed by interconnecting various submodels.

*DAE*s have seen application in modelling chemical processes, electrical circuits, multibody systems and more ([Dao14], [Ria13], [Arn17], [JT14]). Considering the similarities between the 0-dimensional tunnel networks of hydropower production planning and circuit modelling, and that the goal of the report is to model multiple bodies of water moving through these networks, and finally that the model is simply a subcomponent of the larger watercourse system, the *DAE* would seem like good fit. Here I give a short introduction, and restate the definitions needed for the definition of the hybrid automaton, which is discussed in *Hybrid System Theory*. See [Bre96] or [Kun06] for a thorough treatment of *DAE*s, or [Sim17] for a historical tour. I will follow [HBG+05] and use x_d to refer to the differential variables, and x_a to refer to the algebraic variables.

Definition 2.2.1 (Differential Algebraic Equations):

A system of differential algebraic equations is a set of equations:

$$F(t, x, \dot{x}) = 0$$

Where:

- $F\colon \mathbb{I} \times \mathbb{D}_x \times \mathbb{D}_{\dot{x}} \to \mathbb{C}^m$
- $\mathbb{I} \subseteq \mathbb{R}$ is a (compact) interval;
- $\mathbb{D}_x, \mathbb{D}_{\dot{x}} \subseteq \mathbb{C}^n$ are open;
- $m, n \in \mathbb{N}$.

The definition in [Kun06] of the *DAE* in its most general form was restated in *Definition 2.2.1 (Differential Algebraic Equations)*, and is a daunting formalism. A *DAE* can be classified according to its to its shape as in *Definition 2.2.2 (The regularity of a DAE)*. Note that the definition of regularity used in this report is a superficial classification, and does not imply anything about the solvability. In [Kun06] regularity is defined in opposition to singularity which is a much more useful classification. *DAE*s can also be classified according to the structure and complexity of F. Some common forms are tabulated in Table 2.1, which was reproduced from [CastelloGrinoBasanez98]. A *DAE* that was real, regular, nonlinear, semi-explicit and linear in its derivative, and would take the form of equation (2.1).

$$\begin{aligned} \dot{x}_d &= f(t, x_d, \dot{x}_d, x_a) \\ 0 &= g(t, x_d, x_a) \end{aligned} \tag{2.1}$$

Definition 2.2.2 (The regularity of a DAE)

A *DAE* is regular if $m = n$, and irregular otherwise.

Before even doing a computation, it is interesting to know whether or not the computation can be expected to give a sensible result. An important quality of a *DAE*s in this regard is its solvability, which is treated in for example [Bre96] and [Kun06]. A solvable *DAE* guarantees that a solution exists, and it is up to the computation to find it. Unique solvability guarantees that the solution is unambiguous. The approach taken in this report is one of practical computation in the context of system modelling, and Cellier's perspective on "The Solvability Issue" in [CK06] bears repeating:

> To us, solvability is a non–issue. It is the typical worry of a mathematician who puts the mathematical formulation first, and then tries to interpret the ramifications of that formulation. … Saying that a DAE is unsolvable is equivalent to saying that the phenomenon described by it is "defying causality" in the

sense that the outcome of an experiment is non–deterministic, which in turn is almost equivalent to saying that the phenomenon is non–physical.

This does not detract from the usefulness of solvability. It would be ideal to build models that are uniquely solvable by construction, but this still needs to be shown, and accidents happen. Perhaps the modeller has made some unfortunate modelling decision, like choosing an ill-suited set of variables, as is the case in the pendulum model discussed in [CK06]. Diagnosing unsolvable or ambiguous models as often, and as early as possible is particularly useful in a context of system modelling whose framewroks make it easy to accidentally formulate a non-physical model. Kunkel's definition of solvability in [Kun06] is reproduced here in *Definition 2.2.3 (The solvability of a DAE)*, where a problem is solvable if it has at least one solution. In [Bre96] solvability implies uniqueness of solution. The latter definition might be more in line with Cellier's remark. Solvability is tied to the initial value problem, which imposes an initial constraint on x:

$$x(t_0) = x_0 \tag{2.2}$$

Definition 2.2.3 (The solvability of a DAE):

Let $C^k(\mathbb{I}, \mathbb{C}^n)$ denote the vector space of all k-times continuously differentiable functions from the real interval $\mathbb{I}$ into the complex vector space $\mathbb{C}^n$.

- A function is called a solution of *Definition 2.2.1 (Differential Algebraic Equations)* if the *DAE* is satisfied pointwise.
- The function $x \in C^1(\mathbb{I}, \mathbb{C}^n)$ is called a solution of the initial value problem with initial conditions x_0, if it furthermore satisfies equation (2.2).
- An initial condition is called consistent with F, if the associated initial value problem has at least one solution.

Table 2.1: Morphology of common forms of DAE

Linear	Semi explicit	Constant	$\dot{x}_d + B_{11}x_d + B_{12}x_a = f_1(t)$ $B_{21}x_d + B_{22}(t)x_a = f_2(t)$
		Variable time	$\dot{x}_d + B_{11}(t)x_d + B_{12}(t)x_a = f_1(t)$ $B_{21}(t)x_d + B_{22}(t)x_a = f_2(t)$
	Fully implicit	Constant	$A\dot{x} + Bx = f(t)$
		Variable time	$A(t)\dot{x} + B(t)x = f(t)$
Nonlinear	Semi explicit	General	$f(t, x_d, x_a, \dot{x_d}) = 0$ $0 = g(t, x_d, x_a)$
		Linear derivative	$\dot{x}_d = f(t, x_d, x_a)$ $0 = g(t, x_d, x_a)$
	Fully implicit	General	$f(t, \dot{x}, x) = 0$
		Linear derivative	$A(t, x)\dot{x} + f(t, x) = 0$

The solvability of a *DAE* can be investigated in terms "the indices" of a *DAE*, like the differentiation index, perturbation index, tractability index and more (see [Gea90] for a through treatment of indices). Kunkel remarks on the plethora of these classifiers in [Kun06]:

> Unfortunately, the simultaneous development of the theory in many different research groups has led to a large number of slightly different existence and uniqueness results, particularly based on different concepts of the so-called index. The general idea of all these index concepts is to measure the degree of smoothness of the problem that is needed to obtain existence and uniqueness results.

An index is an indicator of how difficult a *DAE* is to solve. In this report only the differentiation will briefly be discussed, as the computational software used specifies the requirements on input *DAE*s, for certain computations, in terms of such an index. In particular the computation of consistent initial conditions in [HBG+05] is restricted to semi-explicit index-one models.

Definition 2.2.4 (The Differentiation Index of a DAE):

> The minimum number of times that all or part of *Definition 2.2.1 (Differential Algebraic Equations)* must be differentiated, with respect to t, in order to determine $\dot{x}$ as a continuous function of x and t, is the index of the DAE. An index 0 model is called an implicit *ODE*, and the *ODE*-form of a *DAE*, is called the underlying *ODE* of the *DAE*.

Example 2.2.1 (The Differentiation Index of a DAE)

> Consider a regular *DAE* with $n = 3$, and:
>
> $$F = \begin{cases} \dot{x}_0 - f(x_1, x_2) \\ \dot{x}_1 - g(x_0) \\ x_1 - x_2 \end{cases}$$
>
> It is not possible to determine $\dot{x}$ as a function of x and t, as $\dot{x}_2$ does not appear in any of the 3 equations; $\frac{\partial F}{\partial \dot{x}}$ is singular. Differentiating the last equation yields, $\dot{x}_1 + \dot{x}_2 = 0$, this new $\frac{\partial F}{\partial \dot{x}}$ is no longer singular. The *DAE* can be formulated as an *ODE* with simple algebraic manipulation:
>
> $$\begin{aligned} \dot{x}_0 &= f(x_1, x_2) \\ \dot{x}_1 &= g(x_0) \\ \dot{x}_2 &= -g(x_0) \end{aligned}$$
>
> The differentiation index of F was thus 1.

2.2.1 Simulation of a DAE

While a *DAE* is an expressive formalism, and as such is less computationally tractable than more specialised formalisms, practical computation is doable and there exists open-source computational software for this purpose. One example is IDA in SUNDIALS from [HBG+05], which solves the initial value problem of a regular *DAE*. When formulating the initial value problem for a *DAE*, one must ensure that the initial values satisfy the algebraic constraints of the model, [PAN88]:

> The initial values for variables in mixed Differential-Algebraic (DAE) systems must satisfy not only the original equations in the system but also their differentials with respect to time.

And the problem of finding such values is called the consistent initialisation problem, [BLG91]:

> . . . given specified information about the initial state of the problem that is sufficient to specify a unique solution to a DAE, determine the complete initial vector $(x(t_0), \dot{x}(t_0))$ corresponding to this unique solution.

IDA can compute consistent initial conditions for semi-explicit index-one models based on an initial guess; it computes $\dot{x}_d(t_0)$ and $x_a(t_0)$, given $x_d(t_0)$ and a guess $x_a(t_0)$. This capability makes practical computation more ergonomic, in particular when one is working with *HA* whose initial value problems involves zero or more *DAE* initial value problems. There also exists algorithmic techniques for transforming *DAE*s into *ODE*s, see for example [OE17] or [CK06]. The *ODE* can in turn be solved by one of the many open-source software packages available for *ODE*s.

2.3 Hybrid System Theory

Hybrid system theoretical models are, roughly speaking, models with both continuous and discrete dynamics, whose trajectories are interleavings of continuous and discrete trajectories. The continuous dynamics are at any point in continuous time determined by the discrete state, and the discrete dynamics are in turn driven by the evolution of the continuous state. Here I give a brief introduction to some hybrid system theoretical frameworks, and the definitions necessary for the subsequent modelling. See [Sch00] or [LLL09] for a more comprehensive introduction to hybrid system theory.

2.3.1 Mixed Logical Dynamical Systems

First consider the Mixed Logical Dynamical (*MLD*) from [BM99]:

Definition 2.3.1 (Mixed Logical Dynamical System):

$$\begin{aligned} x(t+1) &= A_t x(t) + B_{1t} u(t) + B_{2t}\delta(t) + B_{3t} z(t) \\ y(t) &= C_t x(t) + D_{1t} u(t) + D_{2t}\delta(t) + D_{3t} z(t) \\ E_{2t}\delta(t) + E_{3t} z(t) &\leq E_{1t} u(t) + E_{4t} x(t) + E_{5t} \end{aligned}$$

- where $t \in \mathbb{Z}$, $x = \begin{bmatrix} x_c \\ x_l \end{bmatrix}$, $x_c \in \mathbb{R}^{n_c}$, $x_l \in \mathbb{B}^{n_l}$, $n \triangleq n_c + n_l$ is the state of the system;
- and $y = \begin{bmatrix} y_c \\ y_l \end{bmatrix}$, $y_c \in \mathbb{R}^{p_c}$, $y_l \in \{0,1\}^{p_l}$, $p \triangleq p_c + p_l$, is the output vector;
- and $u = \begin{bmatrix} u_c \\ u_l \end{bmatrix}$, $u_c \in \mathbb{R}^{m_c}$, $u_l \in \{0,1\}^{m_l}$, $m \triangleq m_c + m_l$ is the command input.
- $\delta \in \{0,1\}^{r_l}$ and $z \in \mathbb{R}^{r}_{c}$ represent respectively auxiliary logical and continuous variables.

The *MLD* is a discretised-time framework specified with an eye on computational tractability:

> We restrict the dynamics to be linear and discrete-time in order to obtain computationally tractable control schemes…
> - [BM99]

Nevertheless, the discrete variables of the framework can be separated into discrete and discretised continuous variables that express discrete and continuous relations. The discrete dynamics of the system models logical relations with algebraic equalities and inequalities (see equations **2a** through **2f**, and **4a** through **4e** in [BM99]). This representation is computationally tractable, but it is not conceptually efficient.

Consider the relation $[f(x) \leq 0] \to [\delta = 1]$, where $[f(x) \leq 0]$ is the assertion that $f(x) \leq 0$, and similar for $[\delta = 1]$. It is a very direct expression of a logical relation. In a *MLD* model it would be represented as $f(x) \leq \epsilon + (m - \epsilon)\delta$, where ϵ is some small tolerance, and $m \triangleq \min_{x \in \mathcal{X}} f(x)$ for some bounded set $\mathcal{X}$ assumed to contain x. This is a more indirect representation, and it is also coupled to the actual computation, through ϵ which is the machine epsilon. An *MLD* trades computational tractability for conceptual efficiency, which is only natural for a computational model.

2.3.2 Piecewise Affine Systems

A closely related modelling framework is the *PWA*s, a continuous time-variant of which is found in [RJ00]. Its definition is restated in *Definition 2.3.2 (Piecewise Affine System)*. This is another computationally tractable model, for which one for example can do stability analysis and optimal control. In addition it is shown that the framework can be used to approximate smooth non-linear models up to arbitrary precision. This would be an example of the left-to-right model transformation discussed in *Modelling and Computation*. Here the logical description of the system takes the form of matrices, representing polyhedral cells that partition the state space. Each partition has its own continuous linear dynamics. The polyhedral cells are also an indirect way of expressing logical relations. In *Example (PWA Flower System)* below, the logical relation $[x_0 + x_1 \leq 0] \wedge [-x_0 + x_1 \leq 0]$, is represented by the matrix:

$$\begin{bmatrix} 1 & 1 & 0 \\ -1 & 1 & 0 \end{bmatrix}$$

This is, again, a rather indirect representation. It is a computational detail, and not an essential aspect of the underlying logic, but it is a suitable representation for this particular computational model.

Definition 2.3.2 (Piecewise Affine System):

A piecewise affine system is a set of affine systems:

$$\begin{cases} \dot{x} = a_i + A_i x + B_i u \\ y = c_i + C_i x + D_i u \end{cases} \qquad \text{for } x \in X_i$$

- where $\{X_i\}_{i \in I} \subseteq \mathbb{R}^n$ partitions the state space into a a number of closed (possibly unbounded) polyhedral cells denoted by an index set I,
- and represented in terms of matrices $\bar{E}_i = \begin{bmatrix} E_i & e_i \end{bmatrix}$, $\bar{F}_i = \begin{bmatrix} F_i & f_i \end{bmatrix}$;
- such that $\bar{E}_i \bar{x} \geq 0$ where $x \in X_i, i \in I$;
- and $\bar{F}_i \bar{x} = \bar{F}_j \bar{x}$ where $x \in X_i \cap X_j$, $i, j \in I$, and $\bar{x} = \begin{bmatrix} x \\ 1 \end{bmatrix}$.

Example (PWA Flower System)

Here the *PWA* flower system from **Example 1** in [Hed99] is restated. The flower system cuts the two-dimensional state space into 4 polytopes:

$$G_0 = \begin{bmatrix} 1 & 1 & 0 \\ -1 & 1 & 0 \end{bmatrix} \qquad G_2 = \begin{bmatrix} -1 & -1 & 0 \\ 1 & -1 & 0 \end{bmatrix}$$
$$G_1 = \begin{bmatrix} 1 & -1 & 0 \\ 1 & 1 & 0 \end{bmatrix} \qquad G_3 = \begin{bmatrix} -1 & 1 & 0 \\ -1 & -1 & 0 \end{bmatrix}$$

The dynamics of the system is specified as:

$$\dot{x} = A_i x$$

$$A_0 = \begin{bmatrix} -0.1 & 1 \\ -5 & -0.1 \end{bmatrix} \qquad A_1 = \begin{bmatrix} -0.1 & 5 \\ -1 & -0.1 \end{bmatrix}$$

$$i = \begin{cases} x \in X_0 \cup X_2 & \rightarrow 0 \\ x \in X_1 \cup X_3 & \rightarrow 1 \end{cases}$$

The cell defined by G_0 is shown in Fig. 2.2, and the phase plot of the dynamics A_1 is shown in Fig. 2.3.

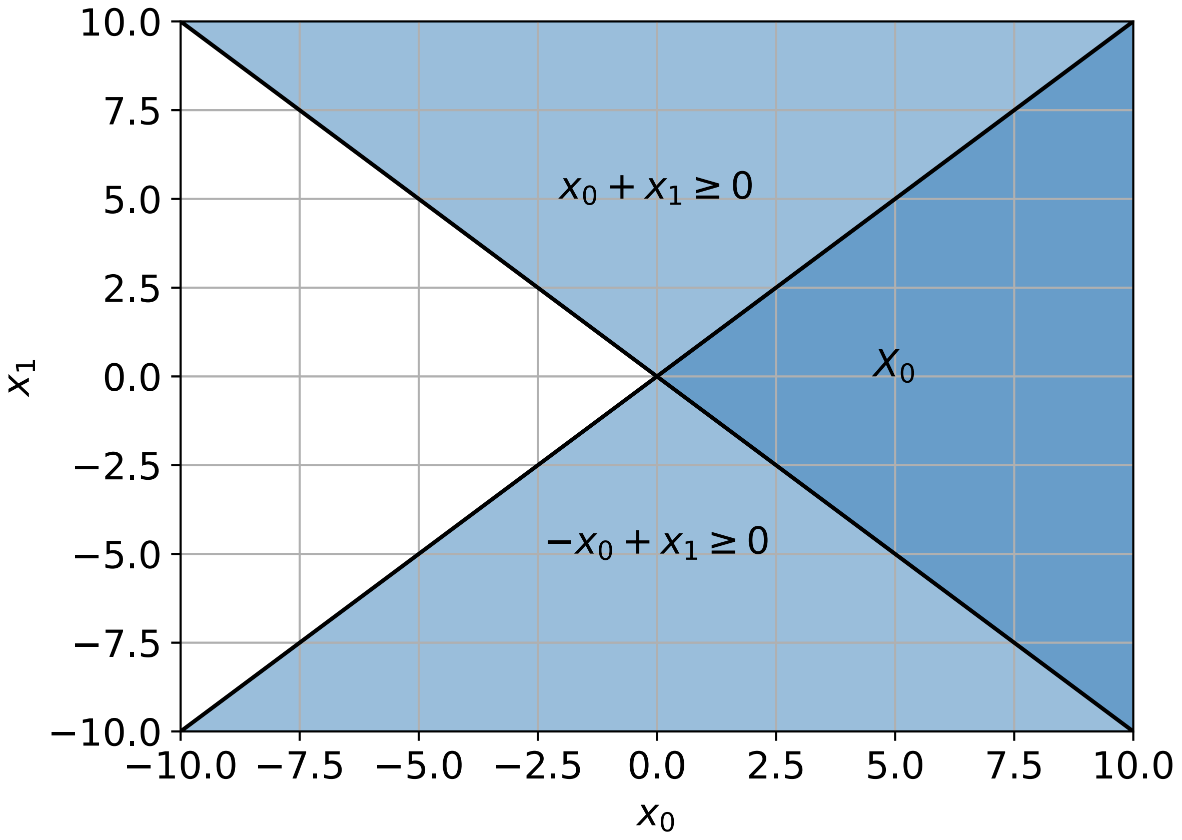

Fig. 2.2: Shaded polyhedron defined by the two inequalities of G_0, $x_0 + x_1 \leq 0$ and $-x_0 + x_1 \leq 0$.

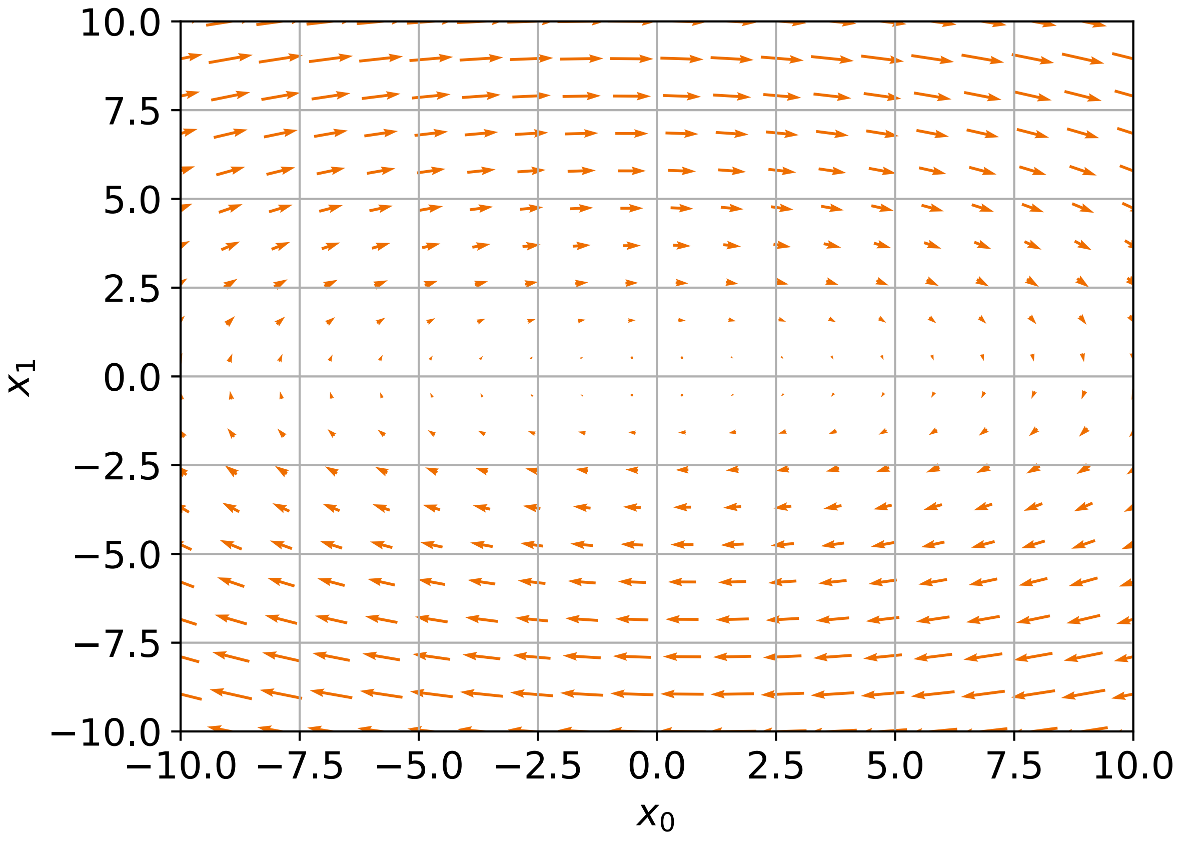

Fig. 2.3: Phase plot of the dynamics of $\dot{x} = A_1 x$, active in X_1 and X_3

2.3.3 Hybrid Automata

The Hybrid Automaton (*HA*) is an expressive hybrid system theoretical modelling framework which lets a modeller directly describe complex discrete dynamics and changes between different modes of behavior in a direct manner. See [KGG+09] for an introduction, or [SHS21] and [AGH+19] for some recent applications. In short, the *HA*, annotates the states and transitions of the finite state automaton from discrete system theory with continuous constraints, actions, and dynamics. Here, unlike the *MLD* and the *PWA*, the expressiveness and directness of the framework is paid for in computational tractability, [DSHLP09]:

> The choice of a modeling framework is a trade-off between two conflicting criteria: the modeling power and the decisive power. The modeling power indicates the size of the class of systems allowing a reformulation in terms of the chosen model description. The decisive power is the ability to prove quantitative and qualitative properties of individual systems in the framework. A model structure that is too broad (like the hybrid automaton) cannot reveal specific properties of a particular element in the model class.

In the same manner that the *DAE* is a natural continuous model for complex systems, the *HA* is a natural hybrid model for complex systems. A system being initially modeled as a *HA* does not preclude the construction of simpler, more computationally tractable models, through the left-to-right model transformations. The focus in this report is more on modelling more than computation. The modelling framework needs to efficiently express models on the left side of Fig. 1.6 and serve as a basis for constructing other models. In this context modelling power becomes more important, and the qualities of frameworks like the *HA* come into focus.

There are multiple variants of the *HA* which differ in for example the specificity of its continuous dynamics (contrast for example [Sch00] with [LLL09]), or how the transitions are modeled. At its most general, the continuous dynamics are specified in terms of a *DAE*. The *HA* from [ZJLS01] is reproduced in *Definition 2.3.3 (Hybrid Automaton)*.

Definition 2.3.3 (Hybrid Automaton):

A hybrid automaton H is a collection $H = (Q, X, Init, f, D, E, G, R)$, where

- Q is a finite collection of discrete variables;
- X is a finite collection of continuous variables with $X = \mathbb{R}^N$;
- $Init \subseteq Q \times X$ is a set of initial states;
- $f\colon Q \times X \to X$ is a vector field;
- $D\colon Q \to \mathcal{P}(X)$ is a map assigning to each $q \in Q$ a subset X called the domain of q;
- $E \subset Q \times Q$ is a set of edges;
- $G\colon E \to \mathcal{P}(X)$ is a map assigning to each edge $e \in E$ a subset of X called the guard of e; and
- $R \to E \times X \to \mathcal{P}(X)$ is a reset map, assigning to each edge $e \in E$ and each $x \in X$ a subset of X

Executions of a Hybrid Automaton

The execution of a *HA* is similar to the concept of a solution of a *DAE* discussed in *Differential Algebraic Equations*. The executions of a hybrid automaton are interleavings of discrete executions and continuous solutions (which in this report also will be referred to as executions). Continuous executions are constructed until the continuous state leaves the domain of the current discrete state q, $D(q)$, and / or enters the edge of a guard $G(e)$, $e \in E(q)$. This event marks the start of a discrete execution. This discrete execution happens instantaneously in continuous time. The final state of the discrete execution marks the start of a new continuous execution. The discrete state is sometimes referred to as the "mode" of the hybrid system, and the construction of a discrete execution is referred to as "the mode selection problem" in [Sch00]:

The problem of finding the next discrete state is called the mode selection problem.

The definitions of time trajectory and execution from [ZJLS01] are reproduced here in *Definition 2.3.4 (Hybrid Time Trajectory)* and *Definition 2.3.5 (Hybrid Execution)*.

Definition 2.3.4 (Hybrid Time Trajectory):

A hybrid time trajectory is a finite or infinite sequence of intervals $\tau = \{I_i\}_{i=0}^N$, such that:

- $I_i = [\tau_i, \tau'_i]$ for all $0 \leq i < N$,
- if $N < \infty$ then either $I_N = [\tau_N, \tau'_N]$ or $I_N = [\tau_N, \tau'_N)$,
- $\tau_i \leq \tau'_i$ for all i and $\tau' = \tau_{i+1}$ for all $0 \leq i < N$.

Definition 2.3.5 (Hybrid Execution):

An execution of a hybrid automaton H is a collection $\chi = (\tau, q, x)$ where τ is a hybrid time trajectory, $q\colon \langle\tau\rangle \top \mathbf{Q}$ is a map, and $x = \{x^i : i \in \langle\tau\rangle\}$ is a collection of C^1 maps $x^i : I_i \to \mathbf{X}$ such that:

- $(q(0), x^0(0)) \in Init$
- for all $i \in \langle\tau\rangle$ and for all $t \in I_i$, $\dot{x}^i(t) = f(q(i), x^i(t))$ and for all $t \in [\tau_i, \tau'_i)$, $x^i(t) \in D(q(i))$,
- for all $i \in \langle\tau\rangle$, $e = (q(i), q(i+1)) \in E$, $x^i(\tau'_i) \in G(e)$, and $x^{i+1}(\tau_{i+1}) \in R(e, x^i(\tau'_i))$.

$\mathcal{E}^M$, $\mathcal{E}^*$, and $\mathcal{E}^\infty$ denotes the set of all maximal, finite and infinite executions respectively. $\mathcal{E}$ denotes the set of all executions of $x_0, q_0 \in Init$.

Definition 2.3.5 (Hybrid Execution) sets the stage for the definition of some useful qualitative properties of a *HA*. These properties are similar to the concept of solvability of a *DAE* and are important in the context of practical computation, as they can be used to determine whether or not an execution will be unique and finite. Before continuing on, a simple example will demonstrate the execution of a *HA*:

Example (HA Flower System)

Here *Example (PWA Flower System)* is formulated and simulated as a *HA*. The cell $X_0 \cup X_2$ is expressed by the relation $x_0^2 \leq x_1^2$, and $X_1 \cup X_3$ by the relation $x_1^2 \leq x_0^2$. Assuming A_i is defined as before, the system on *HA* form is:

- $Q = \begin{bmatrix} q_0 \end{bmatrix}$ taking values in $\mathbb{Q} \in \{0, 1\}$
- $X = \begin{bmatrix} x_0 \\ x_1 \end{bmatrix}$ taking values in $\mathbb{X} \triangleq \mathbb{R}^2$.
- $Init = \mathbb{Q} \times \mathbb{X}$.
- $f(q) = \begin{cases} [q_0 = 0] \rightarrow A_0 x \\ [q_0 = 1] \rightarrow A_1 x \end{cases}$
- $D(q) = \begin{cases} [q_0 = 0] \rightarrow \{x \in \mathbb{X} \mid x_0^2 \leq x_1^2\} \\ [q_0 = 1] \rightarrow \{x \in \mathbb{X} \mid x_1^2 \leq x_0^2\} \end{cases}$
- $E = \{(\begin{bmatrix} 0 \end{bmatrix}, \begin{bmatrix} 1 \end{bmatrix}), (\begin{bmatrix} 1 \end{bmatrix}, \begin{bmatrix} 0 \end{bmatrix})\}$
- $G(e) = \{x \in X \mid x_0^2 = x_1^2\}$
- $R(e) = \emptyset$

Snapshots of an execution starting out at $x = \begin{bmatrix} 1 \\ 0 \end{bmatrix}$ for this *HA* is shown in the figures Fig. 2.4, Fig. 2.5 and Fig. 2.6. The start, as well as the states, of transitions are marked with a small circle. The rays $x_0 = x_1$ and $-x_0 = x_1$ are drawn in dashed green and pink, and the phase plane of the current continuous dynamics are drawn in the background.

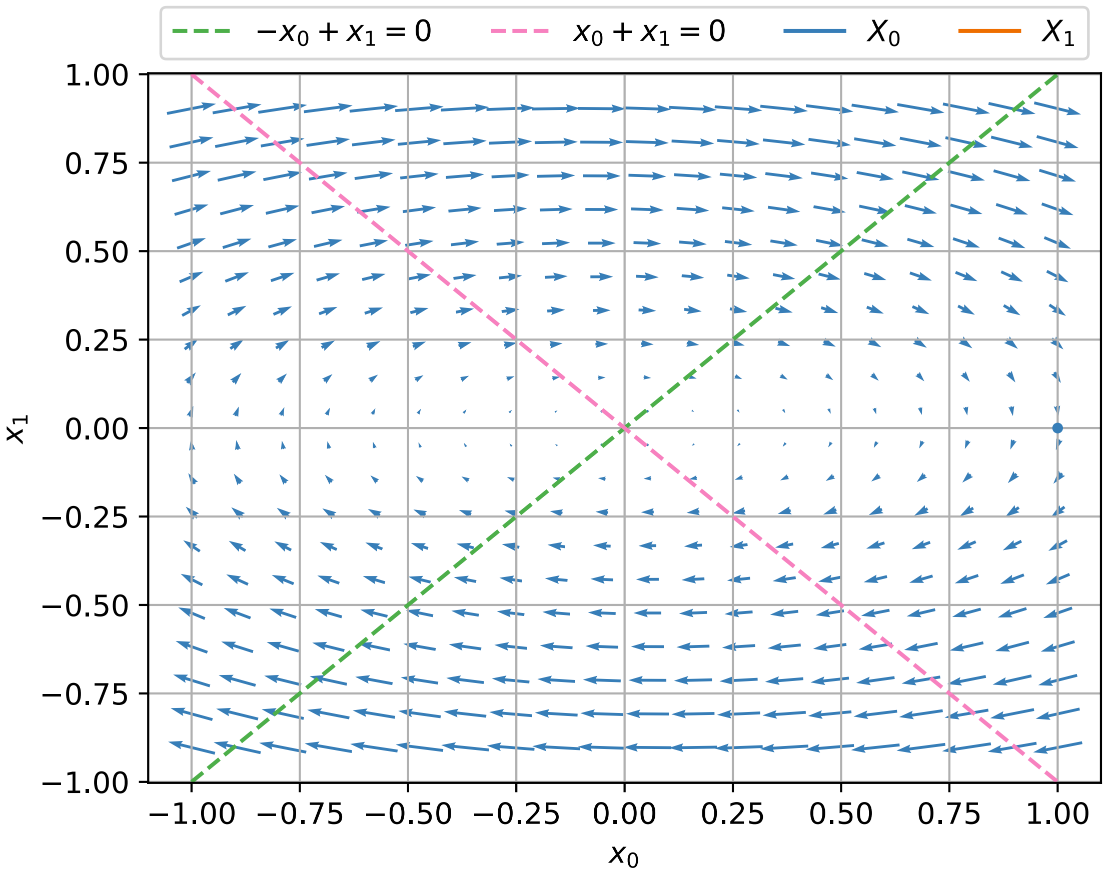

Fig. 2.4: The initial state of an execution of the *HA* flower system, at $t = 0.00$, $x = \begin{bmatrix} 1.00 \\ 0.00 \end{bmatrix}$, in state $q = [0]$.

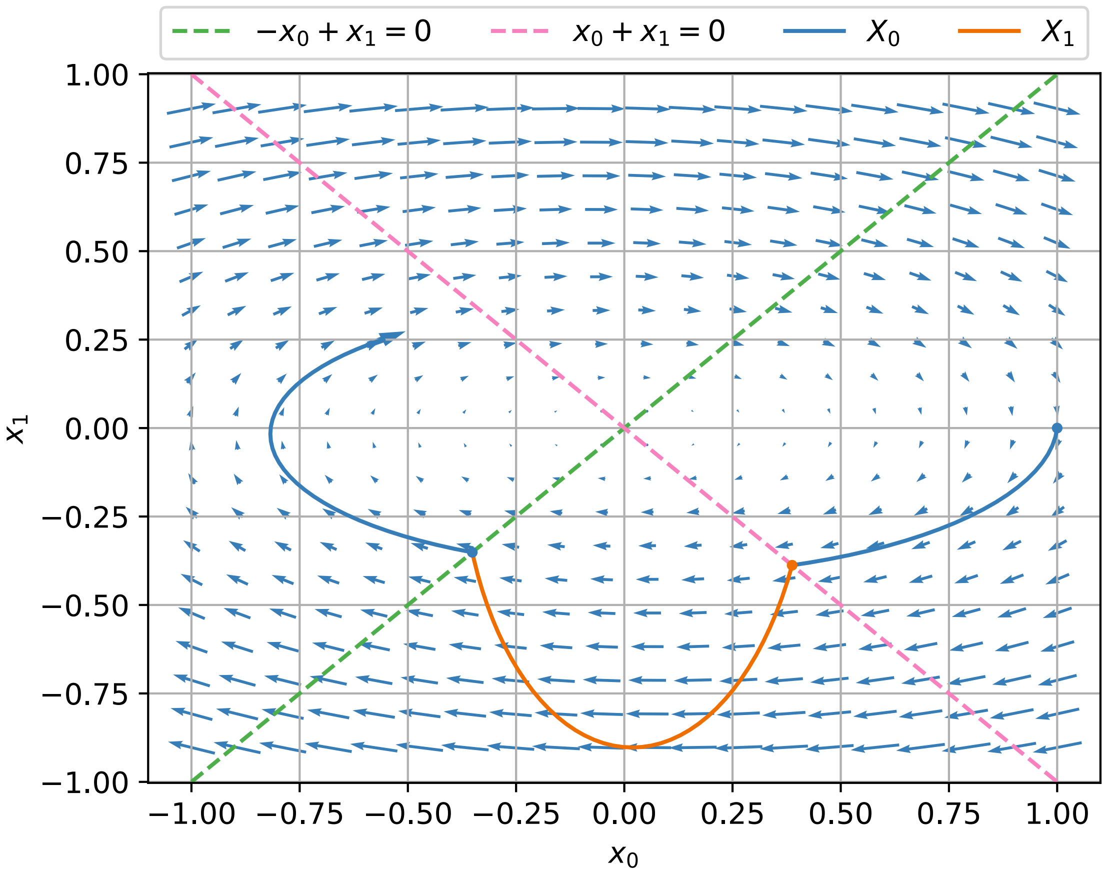

Fig. 2.5: The eventual state of an execution of the *HA* flower system, $x = \begin{bmatrix} -0.56 \\ 0.25 \end{bmatrix}$, at $t = 2.43$, in $q = [0]$.

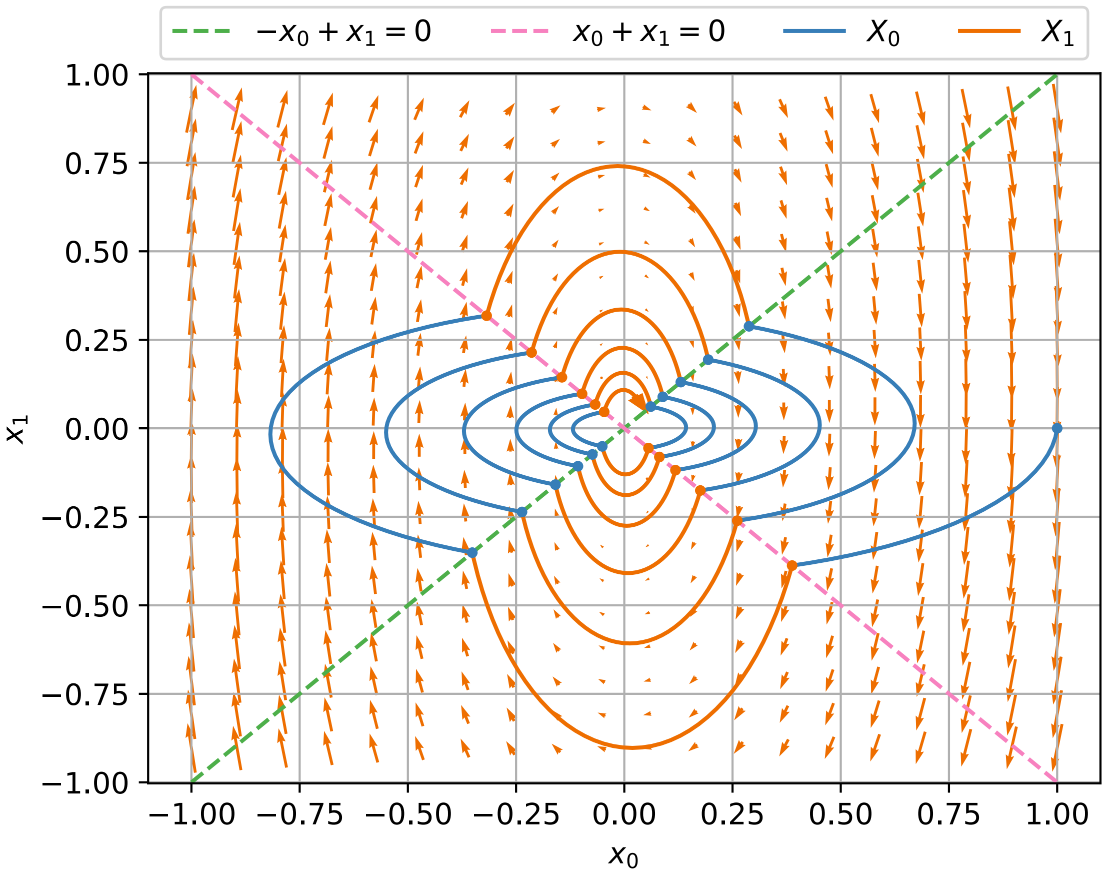

Fig. 2.6: The eventual state of an execution of the *HA* flower system, $x = \begin{bmatrix}0.03\\0.09\end{bmatrix}$, at $t = 24.14$, in $q = [1]$.

Determinism of a Hybrid Automaton

Since a hybrid model combines continuous and discrete dynamics into hybrid dynamics, it also combines the potentials for continuous and discrete nondeterminism. Not only continuous and discrete nondeterminism in isolation, but also nondeterminism between the discrete and the continuous; hybrid nondeterminism. The discrete form of determinism in is described in [Hop79] in the context of automata theory:

> The term “deterministic” refers to the fact that on each input there is one and only one state to which the automaton can transition from its current state. In contrast, “nondeterministic” finite automata, the subject of Section 2.3, can be in several states at once.

A deterministic finite automaton in [Hop79] is a 5-tuple $A = (Q, \Sigma, \delta, q_0, F)$. A similar automaton can be constructed by considering the discrete components of the *HA*. Discrete nondeterminism happens during the construction of a hybrid execution whenever:

$$\exists e_0, e_1 \in E \times E \mid e_0 \neq e_1 \wedge x(t) \in G(e_0) \cap G(e_1)$$

The continuous form of determinism is equivalent to uniqueness of solution. Hybrid nondeterminism, then, is the ambiguity between the continued construction of the current continuous execution, or the beginning of a new discrete execution, by taking some discrete transition e, where $x \in G(e)$. It is an ambiguity between the continuous and discrete dynamics of a hybrid model. The union of these three forms of determinism is captured in [LJS+03] using the set of all executions $\mathcal{E}$ from *Definition 2.3.5 (Hybrid Execution)*.

Definition 2.3.6 (Deterministic Hybrid Automaton):

> A hybrid automaton H is called deterministic if $\mathcal{E}^M(q_0, x_0)$ contains at most one element for all $(q_0, x_0) \in Init$.

Zenoness of a Hybrid Automaton

The definition of the hybrid execution includes infinite executions, and automata that generates such executions are called nonblocking.

Definition 2.3.7 (Blocking Hybrid Automaton):

> A hybrid automaton H is called nonblocking if $\mathcal{E}^\infty(q_0, x_0)$ is non empty for all $(q_0, x_0) \in Init$.

An execution can be infinite without progressing in continuous time. This is property called zenoness, and is described for example in [DSHLP09]:

> *Zeno behaviour* is the phenomenon that for a dynamical system an infinite number of events occur in a finite length time-interval.

[ZJLS01] defines it in terms of hybrid execution:

> An execution is finite if τ is a finite sequence ending with a compact interval, it is called infinite if τ is either an infinite sequence or if $\tau_\infty(\mathcal{X}) = \infty$, and it is called Zeno if it is infinite but $\tau_\infty(\mathcal{X}) < \infty$. The execution time of a Zeno execution is also called the Zeno time.

The definition of zenoness from [ZJLS01] is reproduced in *Definition 2.3.8 (Zeno Hybrid Automaton)*.

Definition 2.3.8 (Zeno Hybrid Automaton):

> A hybrid automaton H is Zeno if there exists $(q_0, x_0) \in Init$ such that all executions in $\mathcal{E}^\infty(q_0, x_0)$ are Zeno.

Zenoness causes hybrid executions to get "stuck" in an infinite discrete execution that renders the "mode selection problem" of Schaft unsolvable, [Sch00]:

> … it may happen that a cycling between different modes occurs ("livelock"), and the simulator does not return to a situation in which motion according to some continuous dynamics is generated, so that effectively the simulation stops.

It is important to note that Zeno behaviour can be a consequence of both modelling and / or computation. It is not a property of the system being modelled. In this sense it is similar to the solvability issue of a *DAE* discussed in *Differential Algebraic Equations*, and is another "non-issue" in the words of Cellier. However, it does have negative implications for practical computation and analysis. It might stall computations, and give false negatives for reachability and stability problems. Establishing that a hybrid system is zenoless is a useful result, and is something that modellers need to strive for, and robust computations need to account for, [Sch00]:

> In such situations the simulation software should provide a warning to the user, and if it is difficult to make a definitive choice between several possibilities perhaps the solver should even work out all reasonable options in parallel.

Simulation of Hybrid Automata

A model is not always perfectly constructed ex nihilo, and is rather the intermediate result of a modelling process involving both modelling and computation. During the construction of hybrid models with a high level of discrete detail, one needs tooling to verify, test, and explore designs. Campbell's observations about *DAE*s can similarly be applied to *HA*s, [Cam95]:

> Being able to do engineering design and computer simulation directly on these original equations would lead to faster simulation and design, permit easier model variation, and allow for more complex models.

A modelling framework should thus be directly usable for practical computation, even though the completed model might not necessarily be used *as* a computational model. Perhaps it will only serve as a basis for constructing other computational models. The *HA*, like the *DAE*, is an expressive formalism, that does admit practical computation. See [Zim10] for an overview of existing software solutions of variable structure or hybrid systems. [Sch00] discusses approaches to the simulation of a hybrid system, such as a *HA*. One of these approaches is smoothing:

> In this method, one tries to replace the hybrid model by a smooth model which is in some sense close to it. For instance, diodes in an electrical network may be described as ideal diodes (possibly plus some other elements), which will give rise to regime-switching dynamics, or as strongly nonlinear resistors, which gives rise to smooth dynamics.

Smoothing can be seen as a right-to-left model transformation of the type discussed in *Modelling and Computation*. However, to smooth a model, one first needs a model to smooth. [Sch00] also discusses a more direct approach that seems apt in the light of Campbell's observation. This is called the event tracking method:

> The idea is to simulate the motion in some given mode using a time-stepping method until an event is detected, either by some external signal (a discrete input, such as the turning of a switch) or by violation of some constraints on the continuous state. If such an event occurs, a search is made to find accurately the time of the event and the corresponding state values, and then the integration is restarted from the new initial time and initial condition in the "correct" mode; possibly a search has to be performed to find the correct mode.

What is here called "the correct mode" is in *Definition 2.3.3 (Hybrid Automaton)* the vector-field $f(q)$. Events happen when the continuous state of the system, x, leaves the domain, $D(q)$, or, when x enters the guard of an edge, $G(e)$. As mentioned in *Differential Algebraic Equations*, there exists open-source computation software for working with *DAE*s directly. Some of these, like [HBG+05], include root-finding capabilities. These capabilities can be used to drive the event detection algorithms. A root does not necessarily constitute an event, as the guards of an *HA* is defined in terms of subsets, and the continuous trajectory might have to produce several zero-crossings to enter, or leave, a subset. Zero-crossing and event detection also poses their own set of technical challenges, but these will be left aside in this report (see [ZYM08] for an in-depth treatment).

After detecting an event and computing a new *DAE*, the continuous integration must be restarted. A *DAE* can contain algebraic constraints, which complicate the consistent initialisation problem. The execution of a *HA* will need to find consistent initial values for every continuous execution in the hybrid execution, [Sch00]:

> In the context of hybrid systems, start-up procedures for DAE solvers should receive particular attention since re-initializations are expected to occur frequently.

When transitioning from one discrete state to another, $q_0 \to q_1$, and thus from one continuous dynamic to another, the continuous state might need to be subjected to the resets $R(e, x)$ of *Definition 2.3.3 (Hybrid Automaton)*. Let x^0 and x^1, and F^0 and F^1 denote the continuous state and dynamics before and after a transition. It is not necessarily the case that $F^0(x^0) = 0$ will imply $F^1(x^0) = 0$. Even with the help of software to find x_a^1 and $\dot{x}_d^1$, as discussed in *Simulation of a DAE*, one still needs to compute x_d^1 according to the reset map. This computation might induce zenoness in *HA* that are zenoless by *Definition 2.3.8 (Zeno Hybrid Automaton)*, [Sch00]:

> Theoretically, the state after the jump should satisfy certain constraints exactly; finite word length effects however will cause small deviations in the order of the machine precision. Such deviations may cause an interaction with the mode selection module; in particular it may appear that a certain constraint is violated so that a new event is detected.

[And94] provides an informative high-level description of such an event tracking, or event detecting, simulation algorithm of a hybrid system in Omola (which also, mutatis mutandis, works for *HA*s), and illustrates it in **Figure 5.10** which is reconstructed in Fig. 2.7.

Another interesting algorithm, that refines the "Solve DAE problem" block of Fig. 2.7, is found in [Zim13], where Zimmer presents the Sol-framework. This is a framework for working with models of variable (mathematical) structure, such as a *HA*. Zimmer introduces the notion of a dynamic *DAE* processor (*DDP*) whose function is to turn an implicit *DAE* into an explicit *ODE*, as discussed in *Differential Algebraic Equations*. This is done during, not before, the construction of the execution. Such an approach is essential when working with *HA*s of high levels of discrete detail. The number of potential *DAE*s makes the cost of dealing with all of them eagerly, *before* the construction of an execution, prohibitive compared to the added cost and complexity of dealing with them lazily and on demand *during* the construction. The lazy approach can be combined JIT-compilation-schemes to generate efficient executable code, see [CN21] for a recent report using LLVM ([LA04]).

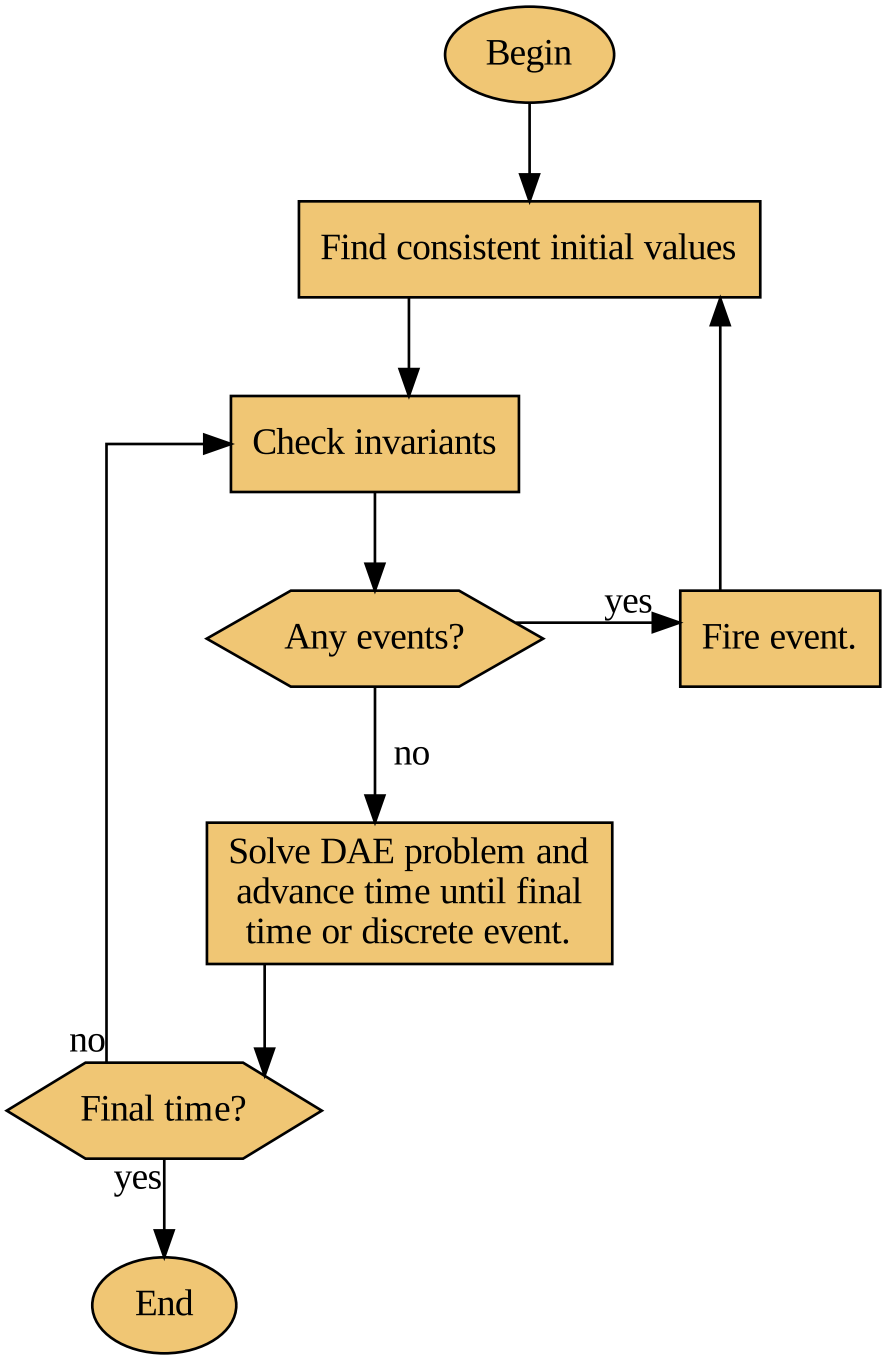
Begin
Find consistent initial values
Check invariants
Any events?
yes
Fire event.
no
Solve DAE problem and advance time until final time or discrete event.
no
Final time?
yes
End

Fig. 2.7: The simulation algorithm for Omola Hybrid Models

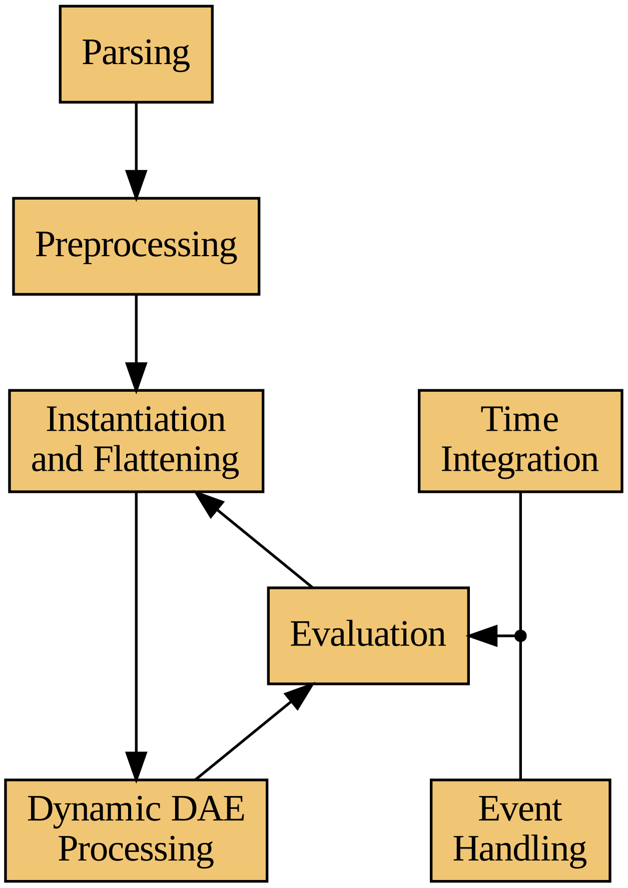

Fig. 2.8: Dynamic processing of Sol

CHAPTER

THREE

RESULTS

The results are divided in two parts. The first part is about modelling frameworks and consists of *Structured Discrete Automata* and *Structured Hybrid Automata*. Here the *HA* framework is decomposed and refined into what is called a structured discrete automaton (*SDA*) and a structured hybrid automaton (*SHA*). This separates the discrete dynamics from the hybrid dynamics of the *HA*, so that one can construct and reason about the discrete aspects of the automaton in isolation. These re-definitions more directly lend themselves to practical computations for automata with a high level of discrete detail. The second part is about modelling and consists of *Logical Description of a Flow Network* and *Discrete Flow Network Model*. It gives a logical description of a tunnel-network in terms of a *DAG*, and sets out a discrete model in the form of a structured discrete automaton, which could form the basis of a *SHA* model of the tunnel systems of a watercourse.

3.1 Structured Discrete Automata

3.1.1 Definition of a Structured Discrete Automaton

[Hop79] defines a discrete automaton in terms of states, input symbols, start state, final state and a transition function, reproduced in equation (3.1).

$$A \triangleq (Q, \Sigma, \delta, q_0, F) \tag{3.1}$$

The *HA* from *Definition 2.3.3 (Hybrid Automaton)* was defined in terms of the set of valid states $Init$. Initial and accepting states will not be a part of the definition of a *SDA*, and are instead considered to be parameters of the computational problems in which the model appears. The symbols of the *DA* in Hopcroft are not required to generate an execution of a *HA* or a *SDA*, so these are also dropped. However, annotating transitions with symbols might still be useful for modelling purposes, and symbols make their re-appearance in the construction of the discrete model in *Discrete Flow Network Model*.

The transitions of a *HA* was defined in terms of edges, $E \subset \mathbb{Q} \times \mathbb{Q}$, guards, $G\colon E \to \mathbb{P}(X)$ and resets, $R \to E \times X \to \mathbb{P}(X)$. The guards and resets pertain only to the continuous dynamics of the automaton. For highly detailed discrete automata $\mathbb{Q}$ is a large set, which makes it difficult to directly represent E in a computation. Direct computation is useful for testing and debugging during the construction of a model. A more structured definition of a discrete automaton, that is designed to be a starting point for constructing hybrid automata with a high level of discrete detail, is suggested in *Definition 3.1.1 (Structured Discrete Automaton)*.

Definition 3.1.1 (Structured Discrete Automaton):

A structured discrete automaton (*SDA*) is a pair $A \triangleq (Q, E)$:

1. where Q a finite collection of N discrete variables, taking values in $\mathbb{Q} \subseteq \mathbb{Z}^N$;
   - with derivatives $\dot{Q}$ taking values in $\dot{\mathbb{Q}} \subseteq \mathbb{Z}^N$;
2. and $E \subset \mathbb{F} \times \mathbb{G}$ is a finite collection of M transitions;
   - where $\mathbb{F}$ is the set of transition equations, $f \colon \mathbf{\Psi} \to \mathbb{Z}$ with $\mathbf{\Psi} \triangleq \mathbb{I} \times \mathbb{Q} \times \dot{\mathbb{Q}}$;
   - and $\mathbb{G}$ is the set of transition guards $g \colon \mathbf{\Upsilon} \to \mathbb{B}$ with $\mathbf{\Upsilon} \triangleq \mathbb{I} \times \mathbb{Q}$ and $\mathbb{I} \subseteq \mathbb{Z}$.

The sets $\mathbf{\Upsilon}$ and $\mathbf{\Psi}$ are called the states and solutions of A respectively.

The *SDA* has a global set of transitions instead of the source, symbol, and target triplet of [Hop79], or a mapping from the set of states, as in for example [Sch00]. Each transition is a tuple of an equation and a guard. This decouples the set of transitions from the set of states, and the *SDA* can be represented directly even for highly detailed discrete dynamics. The guard functions of *Definition 2.3.3 (Hybrid Automaton)* are lowered into the discrete automaton, and made dependent on an independent discrete variable representing discrete time, $t \in \mathbb{I}$. Explicitly representing discrete time enables the construction of models whose determinism (*Definition 2.3.6 (Deterministic Hybrid Automaton)*) and zenoness (*Definition 2.3.8 (Zeno Hybrid Automaton)*) are easier to reason about. The transition guards are in this report assumed to be logical expressions over the discrete variables and boolean literals $\top$, $\bot$. These are composed by a subset of the operators used in [BM99] from *Mixed Logical Dynamical Systems*: conjunction, $\wedge$, disjunction $\vee$, complement, $\neg$, as well as algebraic statements, or assertions, $[a = b]$, $[a < b]$, and $[a \leq b]$. The guards of the *HA* are functions of continuous variables, but the guards of the *SDA* are functions of discrete variables. This ensures the complete separation of the *SDA* from the *SHA*. The relations between discrete variables and continuous roots will be dealt with during the definition of a *SHA*. For now recall that the *MLD* of Bemporad in *Mixed Logical Dynamical Systems* used the syntax $[f(x) > 0] \leftrightarrow [\delta = 1]$ to bridge a (discretised) continuous relation with a discrete variable.

A transition, $e \triangleq (f, g)$, is active in $\upsilon \in \Upsilon$ when $g(\upsilon) = \top$, and inactive when $g(\upsilon) = \bot$. The active transitions of υ are denoted $E^\top(\upsilon)$ and the inactive guards are denoted $E^\bot(\upsilon)$. The active and inactive transition guards, G, and equations, F are denoted in a similar manner. E and F will be used as shorthand for $E^\top$ and $F^\top$ respectively. The active equations form a system of m discrete differential algebraic equations (3.2), which determine $\dot{q}$, or ψ, in υ.

$$F(\psi) = 0 \tag{3.2}$$

It is not necessarily the case that $N = m$ or $M = m$, and it might not determine, or uniquely determine ψ. Equation (3.2) might for example be underdetermined. This is dealt with by defining the solutions to F to be the set of $\dot{q} \in \dot{\mathbb{Q}}$ such that $F(t, q, \dot{q}) = 0$ where $\dot{q}_i = 0$ if $\frac{\partial F}{\partial \dot{q}_i} = 0$. If $\dot{q}_i$ is not determined then q_i should remain constant. Equation (3.2) might also be overdetermined. The significance of an overdetermined system of equations is here one of nondeterminism; a bifurcation point in the construction of the execution of a *SDA*. If F contains both $\dot{q}_i - 1 = 0$ and $\dot{q}_i + 1 = 0$, the execution has two potential branches to continue along. A similar source of nondeterminism is systems of equations that admit multiple solutions. The equation $\dot{q}_i^2 - 1 = 0$ has the same branches as the previous pair of overdetermining equations. A *SDA* that can produce bifurcating transition equations is nondeterministic. In this report nondeterminism caused by overdetermination will be put aside, and determinism will only be discussed with respect to the solvability and unique solvability of F. This is captured in *Definition 3.1.2 (Discrete Derivatives)*.

Definition 3.1.2 (Discrete Derivatives):

Let A be a *SDA*, $\upsilon \in \boldsymbol{\Upsilon}$ be a state of A, and $F \triangleq F^{\top}(\upsilon)$ be the active equations of this state.

The discrete derivatives of A at υ is a map $\dot{\mathbf{Q}} \colon \boldsymbol{\Upsilon} \to \mathbb{P}(\dot{\mathbb{Q}})$:

1. where $\dot{\mathbf{Q}}(\upsilon)$ are the elements $\dot{q} \in \dot{\mathbb{Q}}$ that satisfy $F(t, q, \dot{q}) = 0$;
2. and $\dot{q}_i \triangleq 0$ for any $\dot{q}_i$ such that $\frac{\partial F}{\partial \dot{q}_i} = 0$

3.1.2 Executions of a Structured Discrete Automaton

The structured discrete counterparts of *Definition 2.3.4 (Hybrid Time Trajectory)* and *Definition 2.3.5 (Hybrid Execution)* are reworked into *Definition 3.1.3 (Discrete execution)*. The set of maximal executions, $\mathcal{E}^M$ has been dropped, since an execution here instead is maximal by definition. The time trajectory has been inlined into the definition of the execution.

Definition 3.1.3 (Discrete Execution)

Let A be a *SDA*.

An execution of A is a tuple $\delta \triangleq (q_0)$:

1. with initial state $q_0 \in \mathbb{Q}$;
2. and time trajectory $T \subseteq \mathbb{I}$, which is an interval with $\min(T) \triangleq 0$;
3. and state trajectory $q \colon \mathbb{I} \to \mathbb{Q}$, which is a map:

$$q(i) \triangleq \begin{cases} [i = 0] \to q_0 \\ [i \neq 0] \to q(i-1) + \dot{q}_{i-1} \end{cases}$$

   where $\dot{q}_j \in \dot{\mathbf{Q}}(\upsilon_j)$ from *Definition 3.1.2 (Discrete Derivatives)*;
4. and if $\exists t \in T \mid \dot{q}_t = 0$ then $t_m \triangleq t$ and $T \triangleq [t_0, t_m]$, else $t_m \triangleq \infty$ and $T \triangleq [t_0, t_m)$.

$\mathcal{E}(q_0)$ denotes union of all executions of A starting from q_0. $\mathcal{E}^*(q_0)$, and $\mathcal{E}^{\infty}(q_0)$ denotes the set of all finite and infinite executions respectively. T_m is used to denote a time trajectory whose last element is t_m.

The ticker and delay automata of *Example 3.1.1 (Ticker Automaton)* and *Example 3.1.2 (Delay Automaton)* demonstrate the *SDA* and its execution. The delay automaton demonstrates how the independent variable can be utilized in a guard to sequence a transition.

Example 3.1.1 (Ticker Automaton)

A ticker automaton is an automaton A:

- with $Q \triangleq \{q_0\}$ and $q \in |Z| < 2$;
- and $E \triangleq \{(\dot{q}_0 - 1, \top)\}$.

An execution of this automaton can be seen in Fig. 3.1.

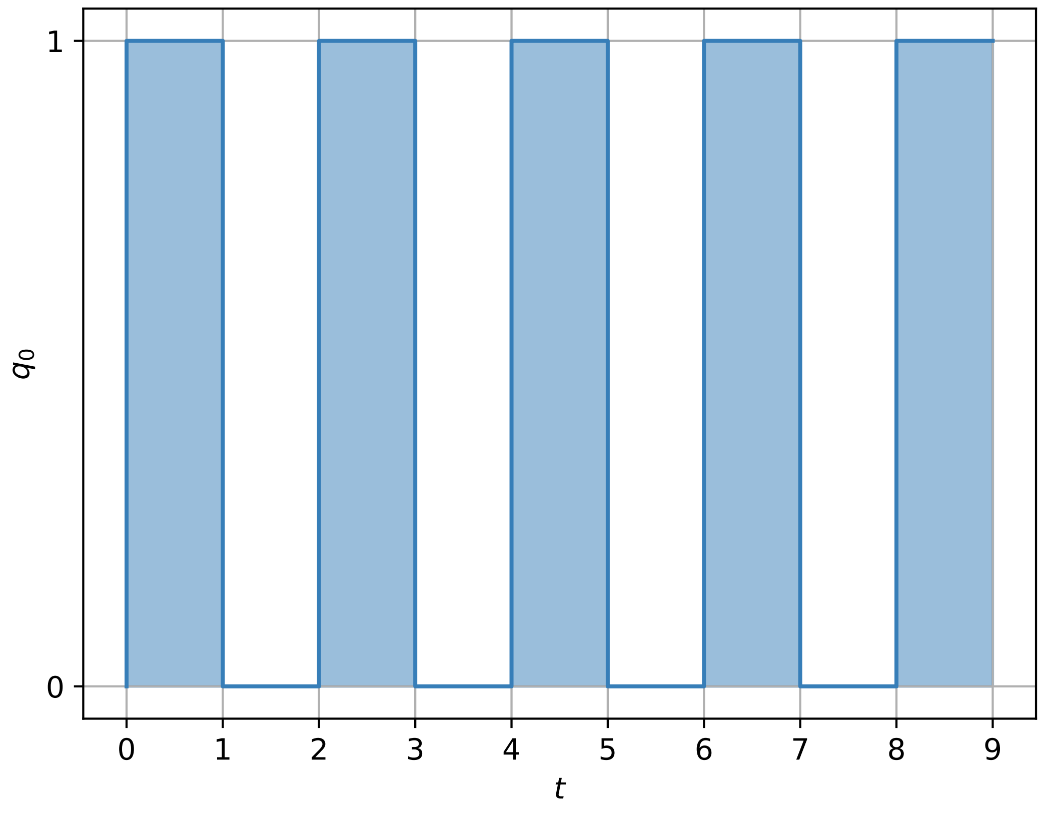

Fig. 3.1: The execution $\delta \triangleq ([0])$ over T_9 of the simple ticker automaton in *Example 3.1.1 (Ticker Automaton)*.

Example 3.1.2 (Delay Automaton)

Guards and equations are functions of the independent variable t and transitions can thus be sequenced in time. This is demonstrated with a delay automaton A:

- with $Q \triangleq \{q_0\}$ where $q_0 \in Z < 2$,
- and $E \triangleq \{(\dot{q}_0 - 1, [t \geq 1])\}$

The transition e_0 will only be active when $t \geq 0$. An execution of this automaton can be seen in Fig. 3.2.

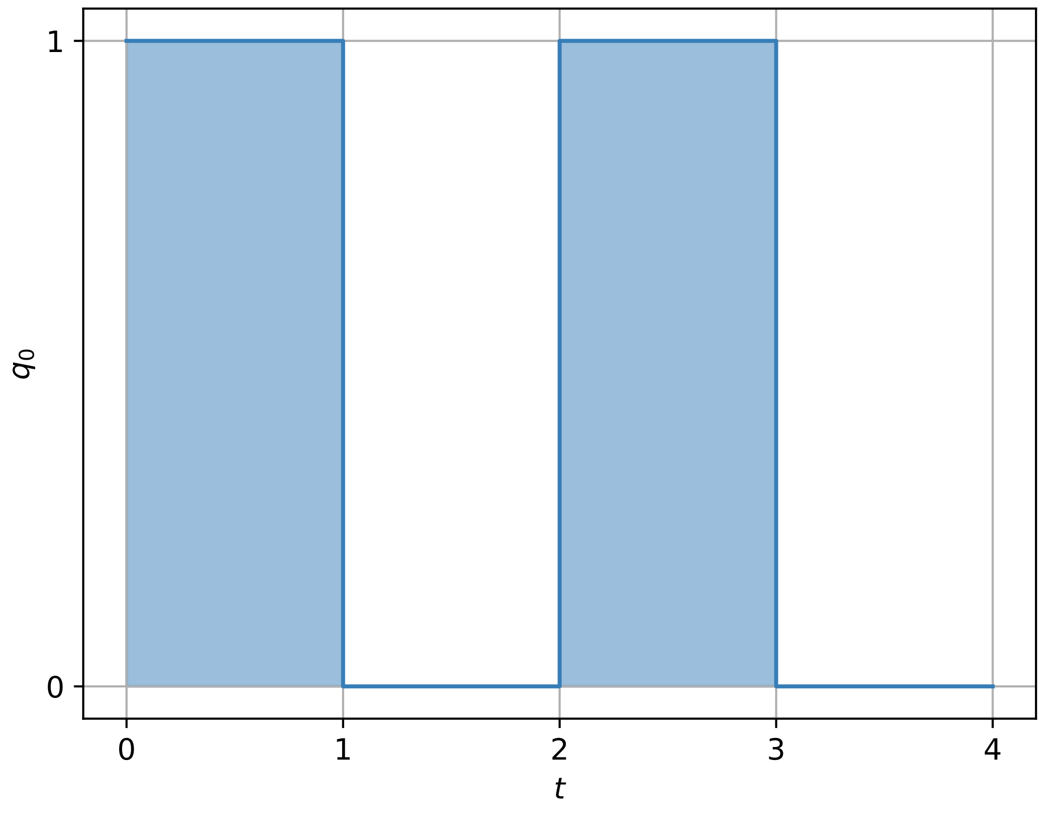

Fig. 3.2: The execution $\delta \triangleq ([1])$ over T_4 of the delay automaton in *Example 3.1.2 (Delay Automaton)*. Note that this execution does not satisfy the 4th point of *Definition 3.1.1 (Structured Discrete Automaton)*, and indeed this constraint was suspended to cleanly demonstrate the delay automaton.

One of the techniques used in the construction of the discrete model in *Discrete Flow Network Model* is "counting variables" which is preemptively demonstrated here. A counting variable tracks the sum of some expression over the other variables of an automaton, and an example of such a variable is demonstrated in *Example 3.1.3 (Counting Automaton)*.

Example 3.1.3 (Counting Automaton)

A counting automaton is an automaton A:

- with $Q \triangleq \{q_0, q_1 \ldots\}$ with $|Q| > 1$, $q_0 \in Z < |Q| - 1$, and $q_i \in Z < 2$,
- and $E \triangleq \{e_0\}$, where f_0, and g_0 are functions:

$$f_0(v) \triangleq \dot{q}_0 - (q_0 - \sum_{q \in Q^+} q)$$

$$g_0(v) \triangleq [(q_0 - \sum_{q \in Q^+} q) \neq 0]$$

- with $Q^+ \triangleq Q \setminus \{q_0\}$

The variable q_0 tracks the number of non-zero variables in Q^+. An execution of this automaton can be seen in Fig. 3.3.

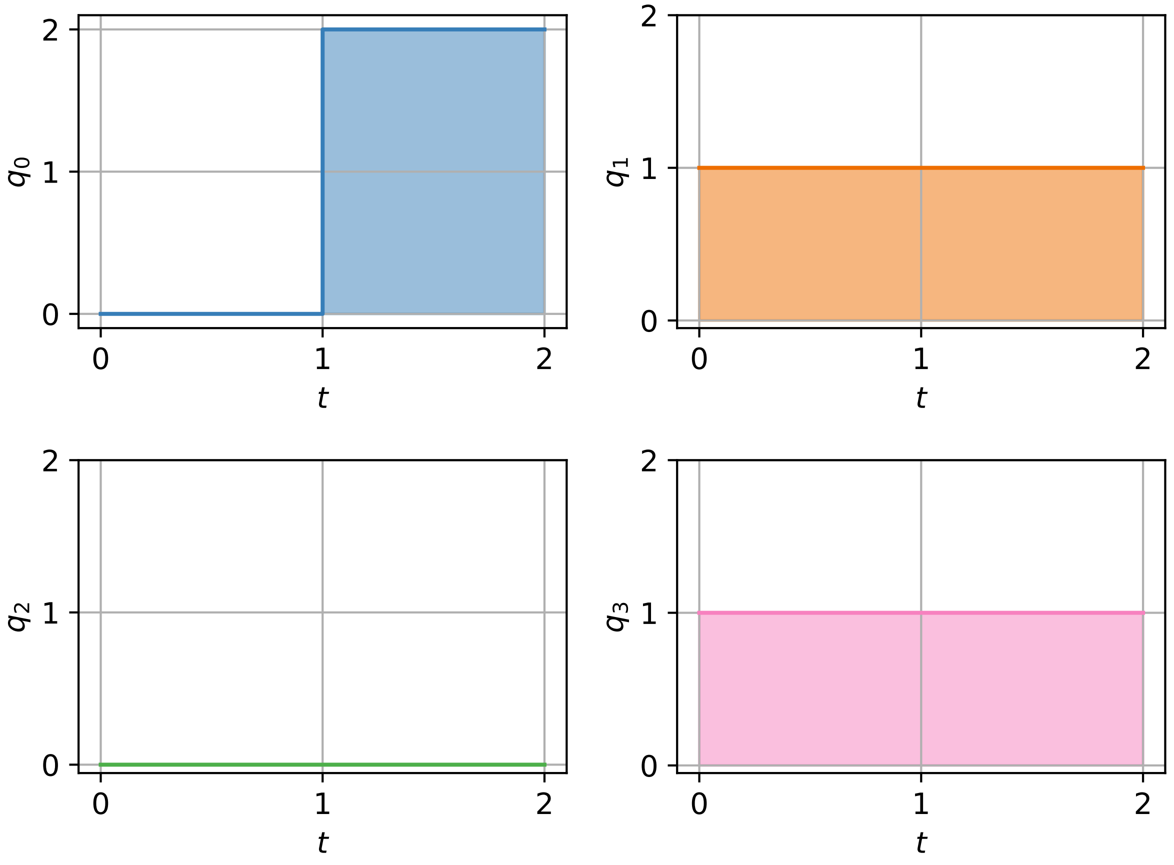

Fig. 3.3: The execution $\delta \triangleq \left(\begin{bmatrix}0 & 1 & 0 & 1\end{bmatrix}^T\right)$ of the automaton over T_2 from *Example 3.1.3 (Counting Automaton)*, with $|Q| = 4$.

Determinism of a Structured Discrete Automaton

As discussed in *Definition of a Structured Discrete Automaton* the determinism of a SDA is dependent on the solvability of equation (3.2). Nondeterminism is caused by ambiguity, and three separate sources of ambiguity were identified. One is underdetermination, which was dealt with in *Definition 3.1.2 (Discrete Derivatives)*. The second is due to a particular *f*, and thus the Fs that include it, having multiple solutions. And the third is due to the potential for an overdetermined F. An overdetermined F might have no solutions if the overdetermining equations are not redundant. The latter two cases were both be interpreted as bifurcations, branching the execution into several potential constructions. The former branches the executions on the solutions of F, and the latter branches the executions on the maximal non-overdetermined subsets of F. Overdetermination was set aside, and determinism will be defined in terms of solvability, thus excluding overdetermination that leads to unsolvable Fs. Solvability and determinism of a SDA are defined in *Definition 3.1.4 (Discrete Solvability)* and *Definition 3.1.5 (Discrete Determinism)*. These are the structured discrete counterparts of *Definition 2.2.3 (The solvability of a DAE)* and *Definition 2.3.6 (Deterministic Hybrid Automaton)*.

Definition 3.1.4 (Discrete Solvability)

A structured discrete automaton is:

1. locally solvable for $\upsilon \triangleq (t, q)$ if $F(t, q, \dot{q}) = 0$ is solvable.
2. locally solvable for $\Upsilon \subset \boldsymbol{\Upsilon}$ if it is locally solvable $\forall \upsilon \in \Upsilon$.
3. globally solvable if it is locally solvable $\forall \upsilon \in \boldsymbol{\Upsilon}$.

It is uniquely solvable in each of these three cases if the solutions to $F(t, q, \dot{q})$ are unique.

Definition 3.1.5 (Discrete Determinism)

A structured discrete automaton is:

1. locally deterministic for q if is uniquely locally solvable for q.
2. locally deterministic for $Q \subset \mathbb{Q}$ if is uniquely locally solvable $\forall q \in Q$.
3. globally deterministic if it is uniquely globally solvable.

Two automata which demonstrate unsolvability and nondeterminism are defined in *Example 3.1.4 (Unsolvable Automaton)* and *Example 3.1.5 (Nondeterministic Automaton)*.

Example 3.1.4 (Unsolvable Automaton)

An example of an unsolvable automaton is A:

- with $Q \triangleq \{q_0\}$ with $q_0 \in Z < 2$,
- and $E \triangleq \{(\dot{q}_0 - 1, \top), (\dot{q}_0, [t > 0])\}$.

For any q, when $t > 0$, f_1 is included in F and $\dot{q}_0$ becomes overdetermined. f_0 and f_1 are not mutually redundant, F has no solution, and $|\dot{\mathbf{Q}}(t, q)| = 0$. In this report overdetermination was set aside, and the combination of this automaton and initial state has no execution.

Example 3.1.5 (Nondeterministic Automaton)

An example of a nondeterministic automaton is A:

- with $Q \triangleq \{q_0\}$ with $q_0 \in Z < 2$,
- and $E \triangleq \{(q_0 - \dot{q}_0^2, \top)\}$.

For $q \triangleq [1]$, f_0 has two solutions; $\dot{q}_0 = 1$, and $\dot{q}_0 = -1$. Thus $|\dot{\mathbf{Q}}(t, q)| = 2$, and the execution, $\delta \triangleq ([1])$ is nondeterministic. It is locally deterministic at $q \triangleq [0]$.

Zenoness of a Structured Discrete Automaton

Zenoness is defined in terms of the execution of the automaton in *Definition 3.1.6 (Discrete Zenoness)*, just as in *Definition 2.3.8 (Zeno Hybrid Automaton)*.

Definition 3.1.6 (Discrete Zenoness)

A structured discrete automaton is:

1. locally zeno for $q \in \mathbb{Q}$ if $\mathcal{E}^\infty(q) \neq \emptyset$.
2. locally zeno for $Q \subset \mathbb{Q}$ if it is locally zeno $\forall q \in Q$.
3. globally zeno if it is locally zeno $\forall q \in \mathbb{Q}$.

An automaton that is not zeno is called a zenoless automaton.

The counting automata of *Example 3.1.3 (Counting Automaton)* is the only zenoless example automata defined so far. The sum of a set of variables that converges will also converge, and so an automaton composed only of constant and counting variables will be zenoless.

3.1.3 Simulation of Structured Discrete Automata

The *SDA* and an algorithm for constructing the execution of a globally deterministic *SDA* was implemented in [RH22]. It was implemented as a C++ library with Python bindings. The algorithm, illustrated in Fig. 3.4, was implemented using an event-based coroutine. It was used to generate the executions in *Example 3.1.1 (Ticker Automaton)*, *Example 3.1.2 (Delay Automaton)*, and *Example 3.1.3 (Counting Automaton)* as well as the executions in *Discrete Flow Network Model*.

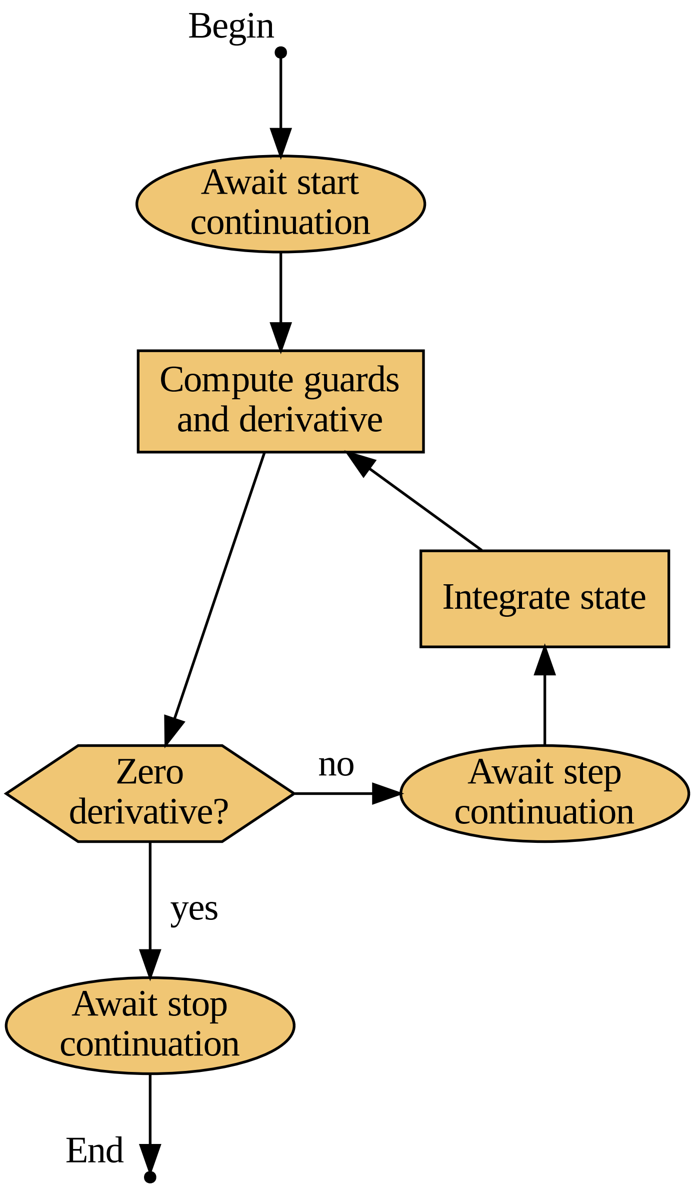

Fig. 3.4: An event-based coroutine for simulating a Structured Discrete Automata

3.2 Structured Hybrid Automata

3.2.1 Definition of a Structured Hybrid Automaton

The *HA* from *Definition 2.3.3 (Hybrid Automaton)* is reconstructed as a *SHA* in *Definition 3.2.1 (Structured Hybrid Automaton)*. The discrete parts of its definition is defined in terms of a *SDA* from *Definition 3.1.1 (Structured Discrete Automaton)*,

Definition 3.2.1 (Structured Hybrid Automaton):

Let $\mathbb{I} \subseteq \mathbb{R}$ be a compact interval, $A^D \triangleq (Q, E)$ be a *SDA* from *Definition 3.1.1 (Structured Discrete Automaton)* with states $\mathbf{\Upsilon}^D$.

A structured hybrid automaton (*SHA*) is a tuple $A \triangleq (A^D, X, U, V, W)$:

1. where X is a finite collection of N continuous variables taking values in $\mathbb{X}$;
   - with derivatives $\dot{X}$ taking values in $\dot{\mathbb{X}}$;
   - and where $\mathbb{X}, \dot{\mathbb{X}} \subseteq \mathbb{C}^N$ are open sets;
   - and where $\mathbf{\Upsilon}^C \triangleq \mathbb{I} \times \mathbb{X}$, $\mathbf{\Upsilon} \triangleq \mathbb{Q} \times \mathbf{\Upsilon}^C$, and $\mathbf{\Psi}^C \triangleq \mathbb{I} \times \mathbb{X} \times \dot{\mathbb{X}}$;
2. where $U \subset \mathbb{F} \times \mathbb{G}$ is a finite collection of M activities;
   - where $\mathbb{F}$ is the set of activity equations $f\colon \mathbf{\Psi}^C \to \mathbb{C}$;
   - and $\mathbb{G}$ is the set of activity guards $g\colon \mathbb{Q} \to \mathbb{B}$;
3. where $V \subset X \times \hat{\mathbb{H}} \times \hat{\mathbb{G}}$ is a finite collection of P actions;
   - where $\hat{\mathbb{H}}$ is the set of action functions $\hat{h}\colon \mathbf{\Psi}^C \to \mathbb{C}$;
   - and $\hat{\mathbb{G}}$ is the set of action guards $\hat{g}\colon \mathbf{\Upsilon}^D \to \mathbb{B}$;
   - where $\mathbf{\Upsilon}^D$ are the states of A^D;
4. where $W \subset Q \times \check{\mathbb{H}} \times \check{\mathbb{G}}$ is a finite collection of O roots;
   - where $\check{\mathbb{H}}$ is the set of root functions $\check{h}\colon \mathbf{\Psi}^C \to \mathbb{C}$;
   - and $\check{\mathbb{G}}$ is the set of root guards $\check{g}\colon \mathbb{Q} \to \mathbb{B}$;
   - and if $w_0, w_1 \in W$ then $q_0 = q_1 \implies w_0 = w_1$.

The sets $\mathbf{\Upsilon}^C$ and $\mathbf{\Psi}^C$ are called the continuous states and continuous solutions of A respectively, and the set $\mathbf{\Upsilon}$ is called the states of A.

The continuous dynamics of the *SHA* is a *DAE* defined in line with *Definition 2.2.1 (Differential Algebraic Equations)*. The domain of the *HA*, $D(q)$, has been dropped, as it is redundant and introduces additional complexity to *Definition 2.3.6 (Deterministic Hybrid Automaton)*. As discussed in *Determinism of a Hybrid Automaton*, if it is the case that $x \in D(q) \cap G(e)$, the construction of the hybrid execution could either continue constructing the current continuous execution, or start constructing a new discrete execution. This is in addition to the ambiguity caused by $x \in G(e_0) \cap G(e_1)$ with $e_0 \neq e_1$, where the construction could branch into either transition. The former ambiguity is redundant, because the domain is, after all, just another subset. A modeller could always construct a transition whose guard is $X/D(q)$. Finally, one has to deal with the case where x is about to leave $D(q)$, but $\nexists e$ such that $x \in G(e)$. Instead of having different types of bifurcations, the domain is dropped, leaving only the ambiguity of $x \in G(e_0) \cap G(e_1)$.

The continuous dynamics, which was represented as a vector field in *Definition 2.3.3 (Hybrid Automaton)*, has been structured into a set of tuples of equation and guard, just like the discrete dynamics in *Definition 3.1.1 (Structured Discrete Automaton)*. The *SHA* can be represented directly in a computation with a large $\mathbb{Q}$ as this set is decoupled from the definition of the continuous dynamics. The guards of the activities, $g \in \mathbb{G}$, are assumed to be of the same form as the guards of a *SDA*. The active and inactive activities are defined in the same manner as the transitions of a

SDA. Thus an activity is active in a discrete state if $g(q) = \top$. G is used as a shorthand for $G^\top$ and similarly for F. The active activity equations form a system of m differential algebraic equations (3.3), which determine $\dot{x}$, or ψ^C, in q.

$$F(\psi^C) = 0 \tag{3.3}$$

The resets of *Definition 2.3.3 (Hybrid Automaton)* are here called the actions of the automaton. The actions have been structured into tuples of variable, function, and guard. The same notation used for active and inactive activities is used to denote the active and inactive actions. $\hat{x}$ and $\hat{H}$ deontes the action variables and functions active in υ^D. For the sake of simplicity it will be assumed that $\nexists(\upsilon^D, \upsilon_0, \upsilon_1) \in \mathbf{\Upsilon}^D \times V \times V$ such that $\upsilon_0 \neq \upsilon_1$ and $x_0 = x_1$.

Roots are a new addition to the hybrid automaton. The guards of the *SDA* are functions of Υ^D. The *SHA* needs to mediate the continuous and discrete dynamics of the automaton. This is done in a similar way to how the relation $[f(x) > 0] \leftrightarrow [\delta = 1]$ from the *MLD*s of [BM99] bridged a (discretised) continuous relation with a discrete relation. The introduction of roots, and their function in the execution of the automaton, is a formalisation of the event detecting approach to the simulation of a hybrid system discussed in *Simulation of Hybrid Automata*. For the sake of simplicity it was assumed in the definition of *Definition 3.2.1 (Structured Hybrid Automaton)* that there are no two roots referring to the same discrete variable. The same notation used for active and inactive activities is used to denote the active and inactive roots. $\check{q}$ and $\check{H}$ denotes the root variables and functions active in q.

3.2.2 Executions of a Structured Hybrid Automaton

The execution of a *SHA* is an interleaving of continuous and discrete executions. Two types of specifically hybrid executions are used to mediate between the continuous and the discrete executions. The action execution deals with the application of actions to the continuous state after a discrete execution has converged. The root execution deals with the transformation of continuous roots into discrete state. These four types of executions are termed subexecutions, and are the components from which an execution of a *SHA* is constructed. The discrete, action, continuous, and root subexecutions are denoted δ^D, δ^U, δ^C, and δ^W respectively. The length of a subexecution is defined as:

$$|\delta| \triangleq \max(T) - \min(T) \tag{3.4}$$

Where T is the discrete or continuous time trajectory of the subexecution. An infinite subexecution has $|\delta| = \infty$, a finite subexecution has $|\delta| \neq \infty$, an empty discrete subexecution has $|\delta^D| = 1$, and a non-empty discrete subexecution has $|\delta^D| > 1$.

Before defining the root and continuous subexecutions, a helper function is constructed in *Definition 3.2.2 (Root Function)* which maps the sign of the root functions of a *SHA* to discrete values. This definition does not account for root functions that are identically zero. This is a simplification in line with IDA from [HBG+05], which is the software used in this report for simulating *DAE*s, [HSB+20]:

> However, if an exact zero of any g_i is found at a point t, ida computes g at $t + \delta$ for a small increment δ, slightly further in the direction of integration, and if any $g_i(t + \delta) = 0$, ida stops and reports an error.

Definition 3.2.2 (Root Function):

Let A be a *SHA*.

The root function is a map $\check{F}\colon \mathbb{Q} \times \mathbf{\Psi}^C \to \mathbb{Q}$:

1. whose elements $\check{f}_i\colon \mathbf{\Psi} \to \mathbb{Q}_i$ is defined as:

$$\check{f}_i \triangleq \begin{cases} \exists \check{q}_j \in \check{q} | q_i = \check{q}_j \to \check{r}_j(\psi^C) \\ \nexists \check{q}_j \in \check{q} | q_i = \check{q}_j \to q_i \end{cases}$$

   - where $R\colon \mathbf{\Psi}^C \to \mathbb{B}^o$ is a map:

$$\check{r}_i \triangleq \begin{cases} [\check{h}_i(\psi) > 0] \vee ([\check{h}_i(\psi) = 0] \wedge [\dot{\check{h}}_i(\psi) > 0]) \to \top \\ [\check{h}_i(\psi) < 0] \vee ([\check{h}_i(\psi) = 0] \wedge [\dot{\check{h}}_i(\psi) < 0]) \to \bot \end{cases}$$

   - and where $\check{x}$ and $\check{H}(\psi^C)$ are the o active root variables and functions, with $\check{H} \triangleq \check{H}^\top(q)$ and $\check{H}\colon \mathbf{\Psi} \to \mathbb{C}^o$.

The root subexecution in *Definition 3.2.3 (Root Subexecution)* transfers the continuous roots detected at some continuous solution, ψ_0^C, to the current discrete state, q_0. It is a fixed size subexecution in two steps.

Definition 3.2.3 (Root Subexecution):

Let A be a *SHA*.

A root subexecution is a tuple $\delta \triangleq (q_0, \psi_0^C)$:

1. with initial state $q_0 \in \mathbb{Q}$, and $\psi_0^C \in \gneqq^C$;
2. and time trajectory $\mathbb{I} \triangleq \mathbb{Z} < 2$;
3. and state trajectory $q\colon \mathbb{I} \to \mathbb{Q}$, which is a map:

$$q(i) \triangleq \begin{cases} [i = 0] \to q_0 \\ [i = 1] \to \check{F}(q_0, \psi_0^C) \end{cases}$$

   - where $\check{F}$ is the root function from *Definition 3.2.2 (Root Function)*.

The execution of a *SHA* will generate several *DAE* initial value problems. The continuous subexecution in *Definition 3.2.4 (Continuous Subexecution)* restates the solution to such a problem from *Definition 2.2.3 (The solvability of a DAE)*. This solutions defined to end on the detection of a root. If a continuous subexecution ends with one or more roots, the next subexecution will be a root subexecution to process these roots. There is no longer any ambiguity between the continued construction of the continuous subexecution and the construction of a new discrete subexecution, as was the case in a *HA* when $x \in D(q) \cap G(e)$. A continuous root of a *SHA* always ends the current continuous subexecution.

Definition 3.2.4 (Continuous Subexecution):

Let A be a *SHA*, and $C^k(\mathbb{I}, \mathbb{X})$ denote the vector space of all k-times continuously differentiable functions from the real interval $\mathbb{I}$ into the vector space $\mathbb{X}$.

A continuous subexecution is a tuple $\delta \triangleq (\upsilon_0)$:

1. with initial state $\upsilon_0 \triangleq (q_0, t_0, x_0) \in \boldsymbol{\Upsilon}$;
2. and time trajectory $T \subseteq \mathbb{I}$, which is an interval with $\min(T) \triangleq t_0$;
3. and state trajectory $x \in C^1(T, \mathbb{X})$ which is a solution of the *DAE* initial value problem (F, t_0, x_0),
   - where $F \triangleq F^\top(q_0)$ are the active activity equations of q_0;
4. and if $\exists t_m \in T$ such that $\check{F}(q_0, \psi(t_0)) \neq \check{F}(q_0, \psi(t_m))$, then $T \triangleq [t_0, t_m]$, else $t_m \triangleq \infty$ and $T \triangleq [t_0, t_m)$;
   - where $\psi^C(t) \triangleq (t, x(t), \dot{x}(t))$ is the solution of x at t;
   - and where $\check{F} \triangleq \check{F}^\top(q_0)$ is the root function from *Definition 3.2.2 (Root Function)*.

The detection of a root is one of the potential discrete events discussed in *Simulation of Hybrid Automata*. The corresponding change in discrete state, which was captured by *Definition 3.2.3 (Root Subexecution)*, is followed by a mode selection problem. The mode selection problem is here called a discrete subexecution. The discrete subexecution needs to propagate the continuous state of the hybrid execution. The discrete execution from *Definition 3.1.3 (Discrete execution)* is extended with continuous state in *Definition 3.2.5 (Discrete Subexecution)*. A discrete subexecution is otherwise identical to a discrete execution. For a discrete subexecution it will always be the case that there exists an earlier subexecution that defines some continuous state trajectory. The discrete subexecution simply passes this along.

Definition 3.2.5 (Discrete Subexecution):

Let A be a *SHA*.

A discrete subexecution is a tuple $\delta \triangleq (\upsilon_0)$ with:

1. initial state $\upsilon_0 \triangleq (q_0, t_0, x_0) \in \boldsymbol{\Upsilon}$;
2. and that is otherwise equivalent to the discrete execution $\delta^D \triangleq (q_0)$ in *Definition 3.1.3 (Discrete execution)*.

At the end of a discrete subexecution the *SHA* might apply a set of actions to the continuous stat. This is done to, for example, ensure the satisfaction of the algebraic constraints of the new active activity equations. This application is captured in a helper function in *Definition 3.2.6 (Action Function)*.

Definition 3.2.6 (Action Function):

Let A be a *SHA*.

The continuous action function is a map $\hat{F} \colon \mathbb{Q} \times \boldsymbol{\Psi}^C \to \mathbb{X}$:

1. whose elements $\hat{f}_i \colon \gneqq^C \to \mathbb{C}$ are maps:

$$\hat{f}_i(\psi^C) \triangleq \begin{cases} \exists \hat{x}_j \in \hat{x} | x_i = \hat{x}_j \to \hat{h}_j(\psi^C) \\ \nexists \hat{x}_j \in \hat{x} | x_i = \hat{x}_j \to x_i \end{cases}$$

   where $\hat{x}$ and $\hat{H}(\psi^C)$ are the p active action variables and functions, with $\hat{H} \triangleq \hat{H}^\top(q)$ and $\hat{H} \colon \boldsymbol{\Psi} \to \mathbb{C}^p$.

The action subexecution of υ applies the active actions to the continuous state, using the action function in *Definition 3.2.6 (Action Function)*, and then recomputes the discrete state based on the new continuous state, using the root function in *Definition 3.2.2 (Root Function)*. The root function was defined in terms of a continuous solution $\psi^C \in \boldsymbol{\Psi}^C$, which requires the computation of a continuous solution from υ. This is captured in *Definition 3.2.7 (Implicit Continuous Solution)*.

Definition 3.2.7 (Implicit Continuous Solution):

Let A be a *SHA*.

The implicit continuous solution of $\upsilon_0 \triangleq (q_0, t_0, x_0) \in \mathbf{\Upsilon}$ is $\phi\colon \mathbf{\Upsilon} \to \Psi^C$, which is a map:

- where $\phi(\upsilon_0) \triangleq (t_0, x(t_0), \dot{x}(t_0))$ is a solution of x at t_0,
- and where x is a solution to the *DAE* initial value problem (F, t_0, x_0);
- and where $F \triangleq F^\top(q_0)$ are the active activity equations of q_0.

The action subexecution is a fixed size subexecution in two steps. It might require consistent initialisation of the continuous state to correctly compute the next discrete state. Consistent initialisation was discussed in *Simulation of a DAE* and is also relevant to the continuous subexecution. It is still a matter of ergonomics, as the modeller could instead be required to ensure that actions result in a continuous state consistent with the new continuous dynamics. The modeller could also be required to specify an initial continuous state that is consistent with the initial continuous dynamics. Ergonomic practical computation requires consistent initialisation to be done, when possible, by the implementation, for both continuous and action subexecutions.

Definition 3.2.8 (Action Subexecution):

Let A be a *SHA*.

An action subexecution is a tuple $\delta \triangleq (\upsilon_0)$:

1. with initial state $\upsilon_0 \triangleq (q_0, t_0, x_0) \in \mathbf{\Upsilon}$;
2. and time trajectory $\mathbb{I} \triangleq \mathbb{Z} < 2$;
3. and continuous state trajectory $x\colon \mathbb{I} \to \mathbb{X}$, which is a map:

$$x(i) \triangleq \begin{cases} [i = 0] \to x_0 \\ [i = 1] \to \hat{F}(q_0, \psi_0^C) \end{cases}$$

   - where $\psi_0^C \triangleq \phi(\upsilon_0)$ is the implicit continuous solution of υ_0 from *Definition 3.2.7 (Implicit Continuous Solution)*;

4. and discrete state trajectory $q\colon \mathbb{I} \to \mathbb{Q}$, which is a map:

$$q(i) \triangleq \begin{cases} [i = 0] \to q_0 \\ [i = 1] \to \check{F}(q_0, \psi_1^C) \end{cases}$$

   - where $\psi_1^C \triangleq \phi(\upsilon_0')$ is the implicit continuous solution of $\upsilon_0' \triangleq (q_0, t_0, x(1))$ from *Definition 3.2.7 (Implicit Continuous Solution)*;
   - and x is the continuous state trajectory defined above.

The execution of a *SHA* is a sequence of subexecutions of a particular structure. A finite continuous subexecution will be followed by a root subexecution, an empty discrete subexecution will be followed by a continuous subexecution, and so on. This structure is shown on diagram form in Fig. 3.5.

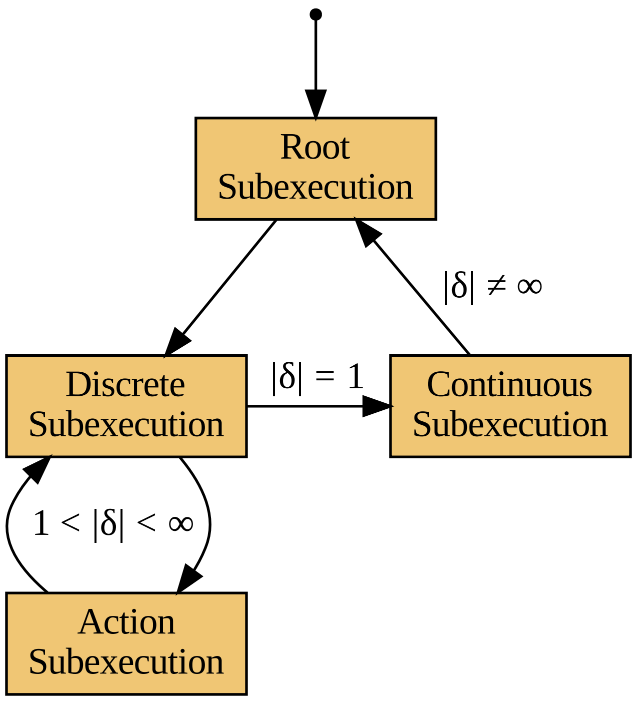

Fig. 3.5: The execution of a Structured Hybrid Automaton

The execution of a *SHA* is finally stated in *Definition 3.2.9 (Hybrid Execution)*. Note that since $|\delta^U| = 2$ and $|\delta^W| = 2$; it is never the case that a root or action subexecution is the last subexecution in such an execution.

Definition 3.2.9 (Hybrid Execution):

Let A be a *SHA*.

A hybrid execution is a tuple $\omega \triangleq (\upsilon_0, \Delta)$:

1. with initial state $\upsilon_0 \in \boldsymbol{\Upsilon}$;
2. and subexecutions $\Delta \triangleq \{\delta_0, \ldots\}$, with initial subexecution $\delta_0^W \triangleq (q_0, \psi_0^C)$:
   - where q_0 is the initial discrete state of υ_0;
   - and where $\psi_0^C \triangleq \phi(\upsilon_0)$ is the implicit continuous solution of υ_0 from *Definition 3.2.7 (Implicit Continuous Solution)*;
3. and time trajectory $I \subseteq \mathbb{Z}$, which is a sequence with $\min(I) \triangleq 0$:
   - and if $\exists i \in I$ such that $|\delta_i| = \infty$, then $i_m \triangleq i$ and $I \triangleq [i_0, i_m]$;
   - else $i_m \triangleq \infty$ and $I \triangleq [i_0, i_m)$;
4. and $\forall \delta_i^C \triangleq (q_{i,0}, t_{i,0}, x_{i,0}) \in \Delta$, where $i \neq i_m$, there $\exists \delta_{i+1}^W \triangleq (q_{i,0}, \psi_{i,m}^C) \in \Delta$:
   - where $\psi_{i,m}^C \triangleq (t_{i,m}, x(t_{i,m}), \dot{x}(t_{i,m}))$ is the solution of x at $t_{i,m} \triangleq \max(T)$;
   - and where x and T are the continuous state and time trajectories of δ_i^C;
5. and $\forall \delta_i^W \triangleq (q_{i,0}, \psi_{i,0}^C) \in \Delta$ there $\exists \delta_{i+1}^D \triangleq (q_{i,m}, t_{i,0}, x_{i,0}) \in \Delta$:
   - where $q_{i,m} \triangleq q(t_{i,m})$ is the state of q at $t_{i,m} \triangleq \max(T)$;
   - and where q and T is the discrete state and time trajectories of δ_i^W;
   - and where $t_{i,0}$ and $x_{i,0}$ is the state of $\psi_{i,0}^C \triangleq (t_{i,0}, x_{i,0}, \dot{x}_{i,0})$;
6. and $\forall \delta_i^D \triangleq (q_{i,0}, t_{i,0}, x_{i,0}) \in \Delta$, where $i \neq i_m \wedge |\delta_i^D| \neq 1$, there $\exists \delta_{i+1}^V \triangleq (q_{i,m}, t_{i,0}, x_{i,0}) \in \Delta$:
   - where $q_{i,m} \triangleq q(t_m^D)$ is the state of q at $t_{i,m}^D \triangleq \max(T)$;
   - and where q and T is the discrete state and time trajectories of δ_i^D;
7. and $\forall \delta_i^V \triangleq (q_{i,0}, t_{i,0}, x_{i,0}) \in \Delta$ there $\exists \delta_{i+1}^D \triangleq (q_{i,m}, t_{i,0}, x_{i,m}) \in \Delta$:
   - where $q_{i,m} \triangleq q(t_{i,m})$ and $x_{i,m} \triangleq x(t_{i,m})$ is the state of q and x at $t_{i,m} \triangleq \max(T)$;
   - and where q, x, and T are the discrete state, continuous state, and time trajectories of δ_t^V;

3.2.3 Simulation of Structured Hybrid Automata

Here the simulation algorithms of Omola and Sol are revisited in light of the execution of a *SHA*.

This execution is a formalisation of the event detecting approach to the simulation of hybrid systems. It can be mapped to the blocks of the event detecting simulation algorithm described in Fig. 2.7. The "Solve DAE problem" block corresponds to the continuous subexecutions. Final time was dropped from the definition of the continuous subexecutions, and all executions in this report are maximal by definition. The root and discrete subexecution covers the "Check Invariants" and "Any events" blocks, and the action subexecution covers the "Fire Event" and "Find Consistent Initial Values" block.

In the dynamic processing algorithm of Sol, the *DDP* transforms *DAE*s to *ODE*s. This is done in response to changes in the continuous dynamics of the model. This is a relatively expensive procedure compared to integration, and one that should be avoided if possible. The execution of a *SHA* formalises the notion of potential structural change. Any discrete subexecution with state trajectory q, with activity guards $G(q_0) \neq G(q_m)$, is a potential structural change. The execution of a *SHA* identifies the moments the *DDP* needs to be run. This is only in the case of a continuous subexecution, or an action subexecution with a non-empty action set. In the former case it needs to run because a continuous subexecution is about to be constructed. In the latter case it needs to run because the action subexecution needs the implicit solution of the current state defined in *Definition 3.2.7 (Implicit Continuous Solution)*. This gives a modeller and algorithm a better chance of synchronising structural changes of the continuous dynamics.

3.3 Logical Description of a Flow Network

A flow network is here described logically in terms of a *DAG* of valves and pipes (see [Die16] for an introduction to *DAG*s and graph theory). This network can contain multiple bodies of water. The valve can represent both gates and actual valves, and each valve is potentially connected to an outside system. For a hydropower plant, gates would typically be connected to a reservoir in a real system and valves internal to its tunnel system would have no external connection. Openings between valves and pipes, and between valves and the outside system, are assumed to be closeable. The elements of the graph are annotated with auxiliary logical entities, called components. This serve as a frame of reference for the variables of the discrete and continuous model. The flow network representation was implemented in [RH22], and the diagrams below were programatically generated.

3.3.1 Flow Network Graph

The flow network is described by a *DAG*, $G \triangleq (V, E)$. Its vertices, V, and edges, E, represent valves and pipes respectively. The directions of the graph are denoted $D \triangleq Z < 2$, where d_0 is in, and d_1 is out. The complement of a direction is denoted d'. These definitions are made with hydropower producing watercourses in mind. These watercourses admit a natural direction along the flow of water, and the edges are assumed to point downstream.

Definition 3.3.1 (Flow Network Graph):

A flow network graph is a directed graph $G \triangleq (C^0, C^1)$, where:

- C^0, C^1 are the valves and pipes of the network respectively.

Example 3.3.1 (Basic Flow Network)

The simplest possible, non-trivial system is the graph:

$$G_0 \triangleq (\{c^0_0, c^0_1\}, \{(c^0_0, c^0_1)\})$$

A diagram of the system can be seen in Fig. 3.6.

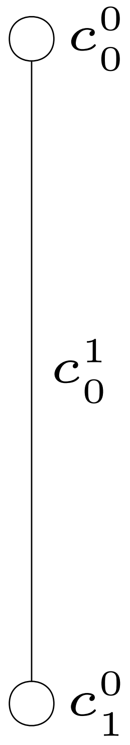

Fig. 3.6: Diagram of the system G_0 in *Example 3.3.1 (Basic Flow Network)* annotated with labels of valves and pipes. Edge directions are not drawn, as they all go from top to bottom in the diagrams of this report.

3.3.2 Flow Network Components

For the purposes of the subsequent modelling the elements of the network are annotated with components. Each element is a component in its own right, and the valves and pipes are termed the primary components of the network. The other components are termed auxiliary components. Each valve, c_i^0 is associated with an outside source, called a tub, $c_i^{0,0}$. Each pipe, c_i^1, is associated with a midpipe, $c_i^{1,0}$, two halfpipes, $c_{i,j}^{1,1}$, and two endpipes, $c_{i,j}^{1,2}$, where $j \in D$. The halfpipe and endpipe $c_{i,0}^{0,k}$ is physically located incident to the source of the corresponding pipe, and $c_{i,1}^{0,k}$ is incident to its target. The union of valves, tubs, pipes, midpipes, halfpipes, and endpipes are called the components of the system, and are tabulated in Table 3.1. The auxiliary components of G_0 are illustrated in Fig. 3.7.

Table 3.1: Flow Network Components

Name	Symbol	Domain	Count
Valve	c_i^0	$Z_c^0 = \mathbb{Z} < \lvert V\rvert$	$N_c^0 = \lvert V\rvert$
Tub	$c_i^{0,0}$	$Z_c^{0,0} = Z_c^0$	$N_c^{0,0} = N_c^0$
Pipe	c_i^1	$Z_c^1 = \mathbb{Z} < \lvert E\rvert$	$N_c^1 = \lvert E\rvert$
Midpipe	$c_i^{1,0}$	$Z_c^{1,0} = Z_c^1$	$N_c^{1,0} = N_c^1$
Halfpipe	$c_{i,j}^{1,1}$	$Z_c^{1,1} = Z_c^1 \times D$	$N_c^{1,1} = 2 \cdot N_c^1$
Endpipe	$c_{i,j}^{1,2}$	$Z_c^{1,2} = Z_c^1 \times D$	$N_c^{1,2} = 2 \cdot N_c^1$

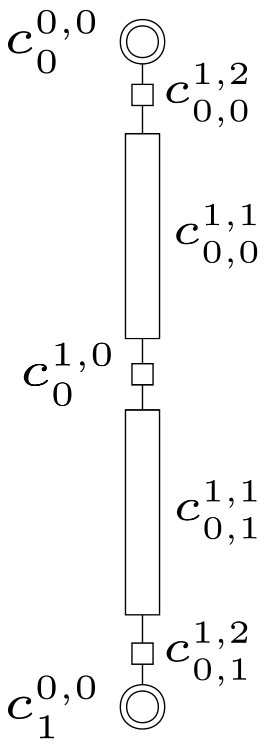

Fig. 3.7: Diagram of the tubs, midpipes, halfpipes, and endpipes of G_0.

Definition 3.3.2 (Flow Network Components):

The components of a flow network, $G \triangleq (C^0, C^1)$ is a tuple $C(G) \triangleq (C^0, C^1, C^{0,0}, C^{1,0}, C^{1,1}, C^{1,2})$:

- where $C^{0,0} \triangleq \{c_i^{0,0} \mid i \in Z < |C^0|\}$;
- and $C^{1,0} \triangleq \{c_i^{1,0} \mid i \in Z < |C^1|\}$;
- and $C^{1,1} \triangleq \{c_{i,j}^{1,1} \mid (i,j) \in Z < |C^1| \times D\}$;
- and $C^{1,2} \triangleq \{c_{i,j}^{1,2} \mid (i,j) \in Z < |C^1| \times D\}$.

Flow Network Incidence

During the definition of the discrete model, discrete expressions and equations will need to refer to groups of variables associated with components that are topologically near each other. The necessary incidence relations are preemptively defined below.

Definition 3.3.3 (Incident Valves):

- The incident valves of a pipe component $c_0^{\mathbf{k}} \triangleq (c_0^0, c_1^0)$ is the set $\Pi_0^{\mathbf{k}} \triangleq \{c_0^0, c_1^0\}$.
- The directed incident valve of a pipe or midpipe $c_0^{\mathbf{k}} \triangleq (c_0^0, c_1^0)$ is denoted $\pi_{d|1}^{\mathbf{k}}(c_0^{\mathbf{k}}) \triangleq c_d^0$, where $d \in D$.
- The incident (singular) valve of an halfpipe or endpipe, $c_{i,j}^{1,k}$ is $\pi_{j|0}^{1,k}(c_i^{1,k})$.

Definition 3.3.4 (Incident Pipes):

- The directed incident pipe or midpipe of a valve c_i^0 is defined as $\pi_{d|\mathbf{k}}^0(c_i^0) \triangleq \{c_j^{\mathbf{k}} \in C^{\mathbf{k}} \mid \pi_{d'|0}^{\mathbf{k}}(c_j^{\mathbf{k}}) \triangleq c_i^0\}$, where $d \in D$.
- The directed incident halfpipes or endpipes of a valve c_i^0 is defined as $\pi_{d|\mathbf{k}}^0(c_i^0) \triangleq \{c_{j,d'}^{\mathbf{k}} \in C^{\mathbf{k}} \mid \pi_{d'|0}^{\mathbf{k}}(c_{j,d'}^{\mathbf{k}}) \triangleq c_i^0\}$, where $d \in D$.
- The incident pipes, midpipes, halfpipes or endpipes of valve is the union $\Pi_{\mathbf{k}}^0(c_i^0) \triangleq \bigcup_{d \in D} \pi_{d|\mathbf{k}}^0(c_i^0)$.

Example 3.3.2 (Incident Pipes)

For the system, $G_1 \triangleq (\{c_1^0, c_1^0, c_2^0, c_3^0\}, \{(c_0^0, c_1^0), (c_1^0, c_2^0), (c_1^0, c_3^0)\})$, shown in Fig. 3.8, the incident endpipes of c_1^0 are $\{c_{0,1}^{1,2}, c_{1,0}^{1,2}, c_{1,0}^{1,2}\}$, or with indices instead of subscripts: $\{c_{\mathbf{1}}^{\mathbf{5}}, c_{\mathbf{2}}^{\mathbf{5}}, c_{\mathbf{4}}^{\mathbf{5}}\}$

Definition 3.3.4 (Incident Tubs):

The incident tubs of a valve c_i^0 is $\pi_{0,0}^0(c_i^0) \triangleq c_i^{0,0}$. Conversely the incident valve of a tub $c_i^{0,0}$ is $\pi_0^{0,0}(c_i^{0,0}) \triangleq c_i^0$.

Definition 3.3.5 (Incident Components):

The incident components of a valve c_i^0 is $\Pi^0(c_i^0) \triangleq \{\pi_{0,0}^0(c_i^0)\} \bigcup \Pi_{1,2}^0(c_i^0)$.

Example 3.3.3 (Incident Components)

For the system, G_0, shown in Fig. 3.7, the incident components of c_1^0 is the set $\{c_1^{0,0}\} \bigcup \{c_{0,1}^{1,2}\}$.

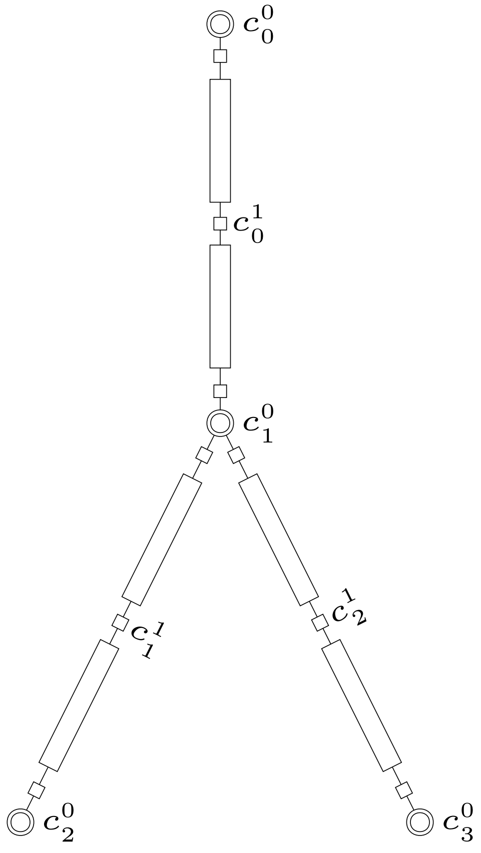

Fig. 3.8: Diagram of the system defined by G_1, with labeled pipes and valves.

3.4 Discrete Flow Network Model

The discrete model constructed here describes arbitrary distributions of waterbodies, and patterns of flow, in a network of pipes and valves. It is a discrete model of a multibody flow network constructed using the *SDA* framework defined in *Structured Discrete Automata*. The construction is done with a *SHA* model of a multibody flow network in mind. It is assumed that the continuous dynamics of the waterbodies of this *SHA* are inelastic.

3.4.1 Discrete Domains, Variables and States

The set of discrete domains of the variables in the model is $\mathbb{Q}^z \subset \mathbb{Z}$, with $z \in Z$. Each domain and its members are associated with a particular semantic: volumes, flow, and so on. The domains are tabulated in Table 3.2. The variables are defined in terms of sets, $Q^{\mathbf{w}}$, where $W = \{\mathbf{w}\ldots\}$. $z^{\mathbf{w}} \in Z$ refers to the corresponding domain index of a variable $q_{\mathbf{i}}^{\mathbf{w}}$, and $Z^{\mathbf{w}}$ denotes the corresponding domain. The variables of the automaton is union in equation (3.5), and the domain of the automaton the product in equation (3.6)

$$\bigcup_{\mathbf{w}\in W} Q^{\mathbf{w}} \tag{3.5}$$

$$\mathbb{Q} = \prod_{\mathbf{w}\in W} \mathbb{Q}^{z^{\mathbf{w}}} \tag{3.6}$$

Every discrete variable is associated with a system component from *Logical Description of a Flow Network*. More than one discrete variable can be associated with a component. The subscripts of each variable refer to the subscripts of the corresponding component. If variables of type $Q^{\mathbf{w}}$ are associated with halfpipes, then the discrete variable $q_{\mathbf{i}}^{\mathbf{w}}$ is associated with the halfpipe $c_{\mathbf{i}}^{1,1}$. $c^{\mathbf{w}}$ denotes the corresponding type of the component of a discrete variable, and $C^{\mathbf{w}}$ denotes the set of such components. The sets of discrete variables are tabulated in Table 3.4 through Table 3.7. These variables are split into two groups. The first group consists of variables representing waterbodies. This group consists of volumes, joints, flows, and control variables. The second group of variables detect events. This group consists of logic and event variables. The cardinalities of $|Q|$ and $|\mathbb{Q}|$ are defined in equation (3.7) and equation (3.8).

$$|Q| = \sum_{\mathbf{w}\in W} |C^{\mathbf{w}}| = 14|V| + 25|E| \tag{3.7}$$

$$|\mathbb{Q}| = \prod_{\mathbf{w}\in W} |Z^{\mathbf{w}}|^{|C^{\mathbf{w}}|} = 2^{14|V|} \cdot 2^{25|E|}) \tag{3.8}$$

For the simplest, non-trivial network, as seen in Fig. 3.6, there is $2^{14\cdot 2} \cdot 2^{25\cdot 1} = 9007199254740992$ different discrete states, which underlines the rationale of *Definition 3.1.1 (Structured Discrete Automaton)*.

Table 3.2: Discrete Domains

Domain			
Semantic	Notation	Members	Count
Volume	$\mathbb{Q}^0$	$\begin{cases}0 \rightarrow \textit{dry}\\ 1 \rightarrow \textit{wet}\end{cases}$	$\lvert\mathbb{Q}^z\rvert = 2$
Joint	$\mathbb{Q}^1$		
Flow	$\mathbb{Q}^2$	$\begin{cases}0 \rightarrow \textit{non-negative}\\ 1 \rightarrow \textit{non-positive}\end{cases}$	
Logic	$\mathbb{Q}^3$	$\begin{cases}0 \rightarrow \textit{bot}\\ 1 \rightarrow \textit{top}\end{cases}$	
Event	$\mathbb{Q}^4$	$\begin{cases}0 \rightarrow \textit{none}\\ 1 \rightarrow \textit{detected}\end{cases}$	
Control	$\mathbb{Q}^5$	$\begin{cases}0 \rightarrow \textit{off}\\ 1 \rightarrow \textit{on}\end{cases}$	

Table 3.3: Discrete Valve Variables

Discrete Valve Variables			
Domain	Semantic	Description	Notation
$\mathbb{Q}^1$	Joint	Joint	$q_i^{0,0}$
$\mathbb{Q}^4$	Event	Seeding	$q_i^{0,1}$
		Colliding	$q_i^{0,2}$
		Receding	$q_i^{0,3}$

Table 3.4: Discrete Tub Variables

Discrete Tub Variables			
Domain	Semantic	Description	Notation
$\mathbb{Q}^0$	Volume	Volume	$q_i^{0,0,0}$
$\mathbb{Q}^1$	Joint	Joint	$q_i^{0,0,1}$
$\mathbb{Q}^2$	Flow	Flow	$q_i^{0,0,2}$
$\mathbb{Q}^3$	Logic	Source	$q_i^{0,0,3}$
$\mathbb{Q}^4$	Event	Branching	$q_i^{0,0,4}$
		Seeding	$q_i^{0,0,5}$
		Cutting	$q_i^{0,0,6}$
		Receding	$q_i^{0,0,7}$
$\mathbb{Q}^5$	Control	Leg Control	$q_i^{0,0,8}$
		Flow Control	$q_i^{0,0,9}$

Table 3.5: Discrete Midpipe Variables

Discrete Midpipe Variables			
Domain	Semantic	Description	Notation
$\mathbb{Q}^1$	Joint	Joint	$q_i^{1,0,0}$
$\mathbb{Q}^4$	Event	Branching	$q_i^{1,0,1}$
		Colliding	$q_i^{1,0,2}$
		Receding	$q_i^{1,0,3}$
$\mathbb{Q}^5$	Control	Joint Control	$q_i^{1,0,4}$

Table 3.6: Discrete Halfpipe Variables

Discrete Halfpipe Variables			
Domain	Semantic	Description	Notation
$\mathbb{Q}^1$	Volume	Leg	$q_{i,j}^{1,1,0}$
$\mathbb{Q}^3$	Control	Leg Control	$q_{i,j}^{1,1,1}$

Table 3.7: Discrete Endpipe Variables

Discrete Endpipe Variables			
Domain	Semantic	Description	Notation
$\mathbb{Q}^1$	Joint	Joint	$q_{i,j}^{1,2,0}$
$\mathbb{Q}^2$	Flow	Flow	$q_{i,j}^{1,2,1}$
$\mathbb{Q}^3$	Logic	Source	$q_{i,j}^{1,2,3}$
$\mathbb{Q}^4$	Event	Branching	$q_{i,j}^{1,2,4}$
		Seeding	$q_{i,j}^{1,2,5}$
		Receding	$q_{i,j}^{1,2,6}$
		Cutting	$q_{i,j}^{1,2,7}$
$\mathbb{Q}^5$	Control	Flow Control	$q_{i,j}^{1,2,8}$

Discrete Waterbody Variables

The control variables correspond to the discrete half of the roots of in *Definition 3.1.1 (Structured Discrete Automaton)*. This is the mechanism whereby the discretisation of continuous roots propagate into the discrete automaton and drive the discrete dynamics. Control variables are in this automaton modelled as impulses that are cleared out after the first timestep.

Volume variables describe which halfpipes and tubs are part of some waterbody. A waterbody is composed by volume variables called legs. The joint variables connect legs together. Even if both legs of a halfpipe are wet they are only part of the same waterbody if the midpipe joint is wet. Similarly, for the valve joints and the legs of the incident components (*Definition 3.3.5 (Incident Components)*). Flow variables of wet legs describe the direction of flow. In a hybrid model the corresponding flow control variables might track the sign of the continuous flow. The flow variables of dry legs will limit the seeding of waterbodies. Perhaps the outside pressure of a dry tub is high enough to prevent a waterbody internal to the pipe network from entering the tub. It might even be about to push water away from the valve, potentially cutting a waterbody into several pieces.

To visualise the discrete state of the model, discrete state diagrams will be used. These annotate the network diagrams of *Logical Description of a Flow Network*, like Fig. 3.6, with discrete variables in some particular state. These diagrams are also programatically generated with [RH22]. Variables of a particular component is drawn on top of the component. Blue coloring indicates wet volumes, joints or active sources, while white coloring means dry. The arrows indicate the value of the directions of flow. Non-negative states point downwards. Note that non-negative flow in a tub variable means the tub is not draining.

Consider for example a structured discrete automaton, A_0, of the system G_0 from *Example 3.3.1 (Basic Flow Network)* in the discrete state, q_0, as visualised in Fig. 3.9. Water is entering the pipe from both valves, but the pipe is not yet filled, as can be seen by the white coloring of the midpipe. The blue, downward pointing triangles of each tub indicates that both tubs are sources, and are not draining water from the connected waterbody. In a hybrid simulation, during the course of the construction of a continuous trajectory, the pipe would eventually fill. This would cause a root to be detected for $[V_{max} - (V_{0,0}^{1,1} + V_{0,1}^{1,1}) < 0]$, where $V_{0,j}^{1,1}$ is the continuous variable of the volume in the halfpipe. The hybrid execution would break out of the continuous subexecution, and construct a discrete subexecution, δ^D. The final state of δ^D, $q(t_m)$, would eventually be used to construct the next continuous subexecution.

The initial state $q(t_0)$ of this discrete trajectory would have the midpipe joint control variable set high, $q_0^{1,0} = 1$, to indicate that the pipe has just been filled. This would cause a collision event to be detected at the midpipe, and a action subexecution would apply actions to ensure that the new set of algebraic constraints are satisfied. For an inelastic continuous model, this might mean computing new values for the flows in the system. After the collision, the flow through $c_0^{0,0}$ must now be equal the flow through $c_1^{0,0}$. The eventual discrete state of an inelastic collision, q_1, is visualised in Fig. 3.9. The midpipe joint variable is here wet and colored blue.

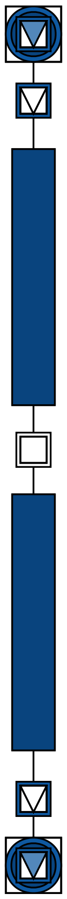

Fig. 3.9: Visualisation of q_0 for A_0 before collision. Water is entering the network from both valves.

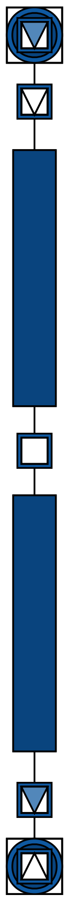

Fig. 3.10: Visualisation of q_1 for A_0 after collision. Water is flowing through the network from the top valve to the bottom valve.

Discrete Event Detecting Variables

The logic and event variables together detect events during the course of an execution. Event variables are set high whenever an event is detected. An event might spuriously be detected during the construction of an execution, but disappear before the execution is complete. Only the final state of a discrete execution determines which events were detected. The events detected in the previous executions be set high in the first timestep of the next discrete execution. All event variables are equipped with an initial transition that resets detected events. This mechanism structures the execution. In the initial step of an execution, reactions to the previously detected events are computed. In the eventual steps these reactions propagate and new events will be detected.

Source variables are logic variables that describe whether or not the incident component of a valve (*Definition 3.3.5*

(Incident Components)) is supplying water to the valve. The count of these variables is used to determine whether a dry valve is about to become wet, or whether a wet valve is about to become dry. This counting principle was illustrated in *Example 3.1.3 (Counting Automaton)*. In Fig. 3.9 both tubs are sourcing their valve and their flow triangles are shaded blue. Neither halfpipe is sourcing and so their flow triangles are shaded white. After the collision Fig. 3.10 the flow through the lower halfpipe and valve have changed sign. The lower tub is no longer a source to the valve, instead the lower halfpipe has become a source.

Event variables come in five types. Branching and seeding describe the imminent expansion of a waterbody. Branch events are detected when a waterbody is about to expand into a valve, or a midpipe. This might cause water to flow into dry halfpipes and tubs. This is represented by seed events. Receding events capture the drying up of wet tubs and halfpipes. This might cause water to stop flowing into wet halfpipes and tubs. This is represented by cutting events. Collision events describe when a waterbody branches into an already wet valve, or when more than one waterbody branches into a dry valve or midpipe.

Lets revisit the example in Fig. 3.9. This is a collision of two waterbodies, and a collision event was detected. This detection would happen in two steps. First, because the midpipe joint control variable was set high in the initial state, q_0, the transition that guards the branching event would evaluate to true, and in the next discrete state, q_2, the branching event variable for the midpipe would go high, as seen in Fig. 3.11. Branching is highlighted in dark yellow. A collision event would then be detected as both halfpipes are already wet. In the next state, q_3, the midpipe collision event would also go high, as seen in Fig. 3.12.

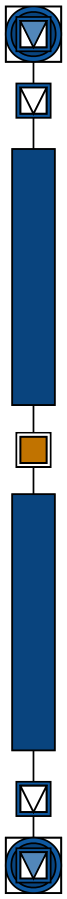

Fig. 3.11: Visualisation of q_2 for A_0. The pipe has become completely full, and a midpipe branching event has been detected.

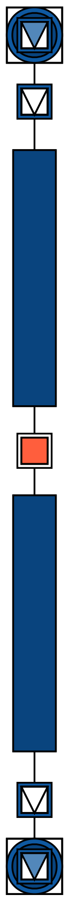

Fig. 3.12: Visualisation of q_3 for A_0. A midpipe collision event has been detected, as the midpipe was branching, and both sides of the midpipe were wet.

Speeding happens only in and around valves, and not midpipes. All dry incident components of a valve can potentially be seeded. A common scenario is when the tub of a completely dry network suddenly becomes wet. Perhaps a gate was opened, and power production is about to start. The waterbody in the tub branches into the valve, and causes the valve and endpipe $c_{0,0}^{1,2}$ to seed. The resulting end-state, q_4, is visualised in Fig. 3.13. Branching events are colored in dark orange, and seeding events in bright orange.

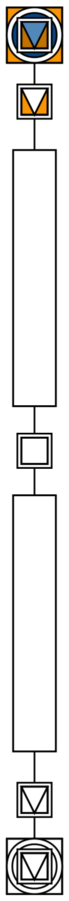

Fig. 3.13: Visualisation of q_4 for A_0.

Receding events are controlled with the halfpipe and tub leg control variables, and would be controlled by continuous dynamics. A wet tub recedes, for example, when the water level in the tub sinks below the valve. Recall that valves also represent gates. A wet halfpipe recedes when its volume goes to zero. If a recession causes the number of sources in a valve to be zero it causes a cutting event in any other wet leg incident to the valve. A hybrid model would then set the flows into these legs, or tubs, to zero.

3.4.2 Discrete Transitions

The symbols of the discrete automaton in [Hop79] were dropped in *Definition 3.1.1 (Structured Discrete Automaton)*, yet are still useful to describe the dynamics of the model. Every transition is associated with a symbol. The symbols of the discrete model are grouped into $|Z|$ different sets, $S^z = \{s^z_y \dots\}$, each uniquely associated with a discrete domain. The set of the symbols, or the alphabet, of the discrete model, is the union in equation (3.9), and the symbols are tabulated in table Table 3.8.

$$S = \bigcup_{z \in Z} S^z \tag{3.9}$$

Table 3.8: Discrete Symbols

Symbols		
Semantic	Notation	Count
Volume	$S^0 = \begin{cases} s^0_0 \to \textit{seed} \\ s^0_1 \to \textit{reap} \end{cases}$	$\lvert S^0 \rvert = 2$
Joint	$S^z = \begin{cases} s^z_0 \to \textit{flip} \end{cases}$	$\lvert S^z \rvert = 1$
Flow		
Logic		
Event		
Control		

A transition, $e^{\mathbf{w},j}_{\mathbf{i}}$, will be defined in terms of a variable, $q^{\mathbf{w}}_{\mathbf{i}}$ and a symbol, $s_j \in S^{z^{\mathbf{w}}}$. The transition equations, f, of every transition $e^{\mathbf{w},j}_{\mathbf{i}}$ is defined in equation (3.10). If the top level expression of a transition guard is a disjunction, as in equation (3.10), the K operands of the disjunction are called guard alternatives denoted $h^{\mathbf{w},j,k}$. The transitions of the automaton is the union in equation (3.12).

$$f(q) \triangleq \dot{q}^{\mathbf{w}}_{\mathbf{i}} = 1 \tag{3.10}$$

$$g^{\mathbf{w},j}_{\mathbf{i}}(q) = \bigwedge_{k \in K} h^{\mathbf{w},j,k}_{\mathbf{i}}(q) \tag{3.11}$$

$$E = \bigcup_{\mathbf{w} \in W} Q \times S^{\mathbf{z}^{\mathbf{w}}} \tag{3.12}$$

The number of transitions, $|E|$ for the simplest, non-trivial network, G_0 is 57, which is tiny compared to the size of the discrete domain. This again, underlines the rationale of *Definition 3.1.1 (Structured Discrete Automaton)*.

Discrete Control, Flow and Event Reset Guards

The guards of both control and flow is determined by the control variables. As mentioned in *Discrete Waterbody Variables*, control variables are reset in the first timestep. Let $\mathbf{w}$ denote the superscript of a control variable:

$$g^{\mathbf{w},0}_{\mathbf{i}} \triangleq [t = 0] \wedge [q^{\mathbf{w}}_{\mathbf{i}} = 1]$$

Flow is flipped in reaction to a high flow control variable. Let $\mathbf{w}_0$ denote the superscripts of a flow variable, and let $\mathbf{w}_1$ denote the corresponding superscript of the flow control variable, then:

$$g^{\mathbf{w}_0,j}_{\mathbf{i}} \triangleq [q^{\mathbf{w}_1,0}_{\mathbf{i}} = 1]$$

Events are reset in the first timestep.

$$h^{\mathbf{w},0}_{\mathbf{i}} \triangleq [t = 0] \wedge [q^{\mathbf{w}}_{\mathbf{i}} = 1]$$

Every event variable transition has a reset guard alternative in addition to the alternatives that detect the event and clear out spurious detections. These will be treated in turn below.

Discrete Source Guards

The tub source guard has two alternatives. The first one changes a false variable to true if the tub leg is wet and the tub flow is positive. Recall that a positive tub flow means the tub is not draining water from the pipe network. The second changes a true variable to false whenever the tub leg is dry, or the tub flow is negative. The guards of the endpipe source variables are specified in the same manner, with respect to the halfpipe leg and endpipe flow. However, the flow through the endpipe is not necessarily sourcing if the flow is positive, as was the case with the tub. It depends on which endpipe it is. The sourcing flow constant, l, is defined in equation (3.13) to compute the sourcing flow for any source.

$$l_{\mathbf{i}}^{\mathbf{w}} \triangleq \begin{cases} \mathbf{w} = (0,0,3) & \rightarrow 1 \\ \mathbf{w} = (1,2,1) \wedge \mathbf{i} = (i,j) & \rightarrow \neg j \end{cases} \tag{3.13}$$

Let $\mathbf{w}_0$, $\mathbf{w}_1$, $\mathbf{w}_2$ denote the superscripts of the source, tub and flow variables respectively, with subscript $\mathbf{i}$. Let the sourcing flow be $l = l_{\mathbf{i}}^{\mathbf{w}}$. The two alternatives of a source guard are then:

$$\begin{aligned} h_{\mathbf{i}}^{\mathbf{w}_1} &\triangleq [q_{\mathbf{i}}^{\mathbf{w}_0} = 0] \wedge [q_{\mathbf{i}}^{\mathbf{w}_1} = 1] \wedge [q_{\mathbf{i}}^{\mathbf{w}_2} = l] \\ h_{\mathbf{i}}^{\mathbf{w}_0,0,1} &\triangleq [q_{\mathbf{i}}^{\mathbf{w}_0} = 1] \wedge \neg([q_{\mathbf{i}}^{\mathbf{w}_1} = 1] \wedge [q_{\mathbf{i}}^{\mathbf{w}_2} = l]) \end{aligned}$$

Discrete Leg Guards

A tub leg is seeded and reaped via the tub leg control. This control would be set high or low by a hybrid model when a waterbody external to the system connects or disconnects from the valve.

$$g_{\mathbf{i}}^{0,0,0,j} \triangleq [q_{\mathbf{i}}^{0,0,0} = j] \wedge [q_{\mathbf{i}}^{0,0,9} = 1]$$

Halfpipe legs are wetted whenever some wet incident valve starts pushing water into it, or when the opposite halfpipe has completely filled the pipe. A halfpipe leg variable can be seeded either from its incident valve or through the midpipe. It is reaped in reaction to its endpipe receding or when the halfpipe leg control is set high:

$$\begin{aligned} g_{i,j}^{1,1,0,0} &\triangleq [t = 0] \wedge [q_{i,j}^{1,1,0} = 0] \wedge ([q_{i,j}^{1,2,5} = 1] \vee [q_{i}^{1,0,1} = 1]) \\ g_{i,j}^{1,1,0,1} &\triangleq [t = 0] \wedge [q_{i,j}^{1,1,0} = 1] \wedge ([q_{i,j}^{1,0,3} = 1] \vee [q_{i}^{0,0,9} = 1]) \end{aligned}$$

Discrete Joint Guards

Tub joints wet whenever the corresponding tub is either seeding or branching. They dry when the tub is either cutting or receding. Similarly the endpipe joint wets when the endpipe is either seeding or branching, and dries when the endpipe is either cutting or receding:

$$\begin{aligned} h_{\mathbf{i}}^{0,0,1,0} &\triangleq [t = 0] \wedge [q_{\mathbf{i}}^{0,0,1} = 0] \wedge ([q_{\mathbf{i}}^{0,0,4} = 1] \vee [q_{\mathbf{i}}^{0,0,5} = 1]) \\ h_{\mathbf{i}}^{0,0,1,1} &\triangleq [t = 0] \wedge [q_{\mathbf{i}}^{0,0,1} = 1] \wedge ([q_{\mathbf{i}}^{0,0,6} = 1] \vee [q_{\mathbf{i}}^{0,0,7} = 1]) \\ h_{\mathbf{i}}^{1,2,0,0,0} &\triangleq [t = 0] \wedge [q_{\mathbf{i}}^{1,2,0} = 0] \wedge ([q_{\mathbf{i}}^{1,2,4} = 1] \vee [q_{\mathbf{i}}^{1,2,5} = 1]) \\ h_{\mathbf{i}}^{1,2,0,0,1} &\triangleq [t = 0] \wedge [q_{\mathbf{i}}^{1,2,0} = 0] \wedge ([q_{\mathbf{i}}^{1,2,6} = 1] \vee [q_{\mathbf{i}}^{1,2,7} = 1]) \end{aligned}$$

The valve joint variable wets in response to a seeding event, and dries in response to a receding event:

$$\begin{aligned} h_{\mathbf{i}}^{0,0,0,0} &\triangleq [t = 0] \wedge [q_{\mathbf{i}}^{0,0} = 0] \wedge [q_{\mathbf{i}}^{0,2} = 1] \\ h_{\mathbf{i}}^{0,0,0,1} &\triangleq [t = 0] \wedge [q_{\mathbf{i}}^{0,0} = 1] \wedge [q_{\mathbf{i}}^{0,4} = 1] \end{aligned}$$

The midpipe joint variable wets in reaction to a branching event and dries in reaction to a recession event:

$$\begin{aligned} h_{\mathbf{i}}^{1,0,0,0,0} &\triangleq [t = 0] \wedge [q_{\mathbf{i}}^{1,0,0} = 0] \wedge [q_{\mathbf{i}}^{1,0,1} = 1] \\ h_{\mathbf{i}}^{1,0,0,0,1} &\triangleq [t = 0] \wedge [q_{\mathbf{i}}^{1,0,0} = 1] \wedge [q_{\mathbf{i}}^{1,0,3} = 1] \end{aligned}$$

Discrete Branching Event Guards

A midpipe branching event is detected when for a dry midpipe joint when the joint control variable is high. This control variable would be set high by a root of the *SHA* when the volume of the halfpipes fill the entire pipe:

$$h_{\mathbf{i}}^{1,0,1,0,1} \triangleq [q_{\mathbf{i}}^{1,0,0} = 0] \wedge [q_{\mathbf{i}}^{1,0,4} = 1]$$

A tub branching event is detected when a wet tub with flow into the network branches into a dry tub joint:

$$h_{\mathbf{i}}^{0,0,4,0,1} \triangleq [q_{\mathbf{i}}^{0,0,4} = 0] \wedge [q_{\mathbf{i}}^{0,0,0} = 1] \wedge [q_{\mathbf{i}}^{0,0,2} = 1] \wedge [q_{\mathbf{i}}^{0,0,1} = 0]$$

An endpipe branching event is detected in a dry endpipe joint when the corresponding halfpipe is part of a waterbody about to enter its incident valve. Let $l = l_{\mathbf{i}}^{1,2,0}$ denote the sourcing flow from equation (3.13), the guard alternative is then:

$$h_{\mathbf{i}}^{1,2,4,0,1} \triangleq [q_{\mathbf{i}}^{1,2,4} = 0] \wedge [q_{\mathbf{i}}^{1,2,0} = 0] \wedge [q_{\mathbf{i}}^{1,0,0} = 1] \wedge [q_{\mathbf{i}}^{1,2,1} = l]$$

Discrete Seeding Event Guards

Seeding happens as a result of branching. When a waterbody branches into a dry valve it is seeded. This might cause the dry tubs and endpipes incident to the valve to be seeded as well. The seeding of a valve is defined in terms of a branching count, b, in equation (3.16). Seeding is detected in a dry valve with a positive branching count, and the event is disabled if the branching count goes to zero:

$$\begin{aligned} h_{\mathbf{i}}^{0,1,0,0} &\triangleq [q_{\mathbf{i}}^{0,1} = 0] \wedge [q_{\mathbf{i}}^{0,0} = 0] \wedge [b_{\mathbf{i}}^{0,0} > 1] \\ h_{\mathbf{i}}^{0,1,0,1} &\triangleq [q_{\mathbf{i}}^{0,1} = 1] \wedge [b_{\mathbf{i}}^{0,0} = 0] \end{aligned}$$

Let $\mathbf{j}$ denote the subscript of the incident valve of a tub or endpipe. Let $l = l_{\mathbf{i}}^{\mathbf{w}}$ denote the sourcing flow from equation (3.13). The guard alternatives of seeding event detection for tubs and endpipes is then defined in equation (3.14). In addition a seeding event in a tub is disabled if a branching event is detected. This alternative is defined in equation (3.15).

$$\begin{aligned} h_{\mathbf{i}}^{0,0,5,0,1} &\triangleq [q_{\mathbf{i}}^{0,0,5} = 0] \wedge [q_{\mathbf{i}}^{0,0,1} = 0] \wedge [q_{\mathbf{i}}^{0,0,2} = \neg l] \wedge [q_{\mathbf{i}}^{0,0,4} = 0] \wedge [q_{\mathbf{j}}^{0,1} = 1] \\ h_{\mathbf{i}}^{1,2,5,0,1} &\triangleq [q_{\mathbf{i}}^{1,2,5} = 0] \wedge [q_{\mathbf{i}}^{1,2,0} = 0] \wedge [q_{\mathbf{i}}^{1,2,1} = \neg l] \wedge [q_{\mathbf{i}}^{1,1,1} = 0] \wedge ([q_{\mathbf{j}}^{0,1} = 1] \vee [q_{\mathbf{j}}^{0,0,1} = 1]) \end{aligned} \tag{3.14}$$

$$h_{\mathbf{i}}^{0,0,5,0,2} \triangleq [q_{\mathbf{i}}^{0,0,5} = 1] \wedge [q_{\mathbf{i}}^{0,0,4} = 1] \tag{3.15}$$

Discrete Collision Event Guards

Valve collisions are defined in terms of a counter called the branching count. Let $B_{\mathbf{i}}^{\mathbf{w}}$ denote the incident branching event variables of a valve variable $q_{\mathbf{i}}^{\mathbf{w}}$, where the incident components are defined in *Definition 3.3.5 (Incident Components)*. The branching count of a valve variable is defined in equation (3.16).

$$b_{\mathbf{i}}^{\mathbf{w}} \triangleq \sum_{q \in B_{\mathbf{i}}^{\mathbf{w}}} [q = 1] \tag{3.16}$$

A valve collision is detected when a wet valve has a positive branching count, or when the branching count is greater than one. It is disabled when either a wet valve has zero branching count, or when the collision count is less than one. This guard is a tracking guard, similar to the automaton in *Example 3.1.3 (Counting Automaton)*.

$$\begin{aligned} h_{\mathbf{i}}^{0,2,0,1} &\triangleq [q_{\mathbf{i}}^{0,2} = 0] \wedge (([q_{\mathbf{i}}^{0,0} = 1] \wedge [b_{\mathbf{i}}^{0,0} > 1]) \vee ([q_{\mathbf{i}}^{0,0} = 0] \wedge [b_{\mathbf{i}}^{0,0} > 2])) \\ h_{\mathbf{i}}^{0,2,0,1} &\triangleq [q_{\mathbf{i}}^{0,2} = 1] \wedge \neg(([q_{\mathbf{i}}^{0,0} = 1] \wedge [b_{\mathbf{i}}^{0,0} > 1]) \vee ([q_{\mathbf{i}}^{0,0} = 0] \wedge [b_{\mathbf{i}}^{0,0} > 2])) \end{aligned}$$

A midpipe collision happens if a branching has been detected, and there is a waterbody in each halfpipe. This is the case when each halfpipe has either a wet halfpipe leg, or a seeding endpipe on each side.

$$h_{\mathbf{i}}^{1,0,2,0,1} \triangleq [q_{\mathbf{i}}^{1,0,1} = 1] \wedge ([q_{\mathbf{i},0}^{1,1,0} = 1] \vee [q_{\mathbf{i},0}^{1,2,5} = 1]) \wedge ([q_{\mathbf{i},1}^{1,1,0} = 1] \vee [q_{\mathbf{i},1}^{1,2,5} = 1])$$

Discrete Cutting Event Guards

Cutting events detect the severing of a waterbody. A tub or endpipe is cut whenever it is sourcing its incident valve, and its leg control is set high. Let $\mathbf{w}_0$, $\mathbf{w}_1$, $\mathbf{w}_2$ denote the cutting event, control and source variables respectively. Then both $h^{0,0,6,0,1}$ and $h_{\mathbf{i}}^{1,2,7,0,1}$ are defined as:

$$h_{\mathbf{i}}^{\mathbf{w}_0} \triangleq [t = 0] \wedge [q^{\mathbf{w}_0} = 0] \wedge [q_{\mathbf{i}}^{\mathbf{w}_1} = 1] \wedge [q_{\mathbf{i}}^{\mathbf{w}_2} = 1]$$

Discrete Receding Event Guards

Tubs, valves, midpipes, and endpipes all recede when the waterbodies that inhabit them are about to leave. Recession spreads from valves and midpipes. Receding events will be defined in terms of a source count, s. This is defined in a similar manner to the branching count in equation (3.16). Let $S_{\mathbf{i}}^{\mathbf{w}}$ denote the incident source event variables of a valve variable $q_{\mathbf{i}}^{\mathbf{w}}$. The incident components are defined as in *Definition 3.3.5 (Incident Components)*. The source count of a valve variable is then defined in equation (3.17).

$$s_{\mathbf{i}}^{\mathbf{w}} \triangleq \sum_{q \in S_{\mathbf{i}}^{\mathbf{w}}} [q = 1] \tag{3.17}$$

Receding is detected in wet valves with zero source count:

$$h_{\mathbf{i}}^{0,3,0,1} \triangleq [q_{\mathbf{i}}^{0,3} = 1] \wedge [s_{\mathbf{i}}^{0,0} = 0]$$

A tub recedes when its incident valve is receding. This variable tracks the valve recession and if the valve at some later point stops receding then the tub will no longer recede:

$$h_{\mathbf{i}_0}^{0,0,7,0,1} \triangleq [q_{\mathbf{i}_0}^{0,0,7} = 0] \wedge [q_{\mathbf{i}_0}^{0,0,6} = 0] \wedge [q_{\mathbf{i}_1}^{0,4} = 1]$$
$$h_{\mathbf{i}_0}^{0,0,7,0,2} \triangleq [t \neq 0] \wedge [q_{\mathbf{i}_0}^{0,0,7} = 1] \wedge \neg[q_{\mathbf{i}_1}^{0,4} = 1]$$

A midpipe receding event is detected when any of its endpipes are receding. In this scenario the pipe is about to go from fully to partially submerged; the valve supplying water to the pipe has just run dry. This is a tracking event, and only done in eventual timesteps, based on events detected in the current execution.

$$h_{\mathbf{i}}^{1,0,3,0,1} \triangleq [t \neq 0] \wedge [q^{1,0,3} = 0] \wedge ([q_{\mathbf{i},0}^{1,2,6} = 1] \vee [q_{\mathbf{i},0}^{1,2,6} = 1])$$
$$h_{\mathbf{i}}^{1,0,3,0,2} \triangleq [t \neq 0] \wedge [q^{1,0,3} = 1] \wedge \neg([q_{\mathbf{i},0}^{1,2,6} = 1] \vee [q_{\mathbf{i},0}^{1,2,6} = 1])$$

The endpipe recede variable tracks the recession of its valve, but is disabled in the event of a cutting:

$$h_{\mathbf{i}}^{1,2,6,0,1} \triangleq [q_{\mathbf{i}}^{1,2,6} = 0] \wedge [q_{\mathbf{i}}^{1,2,7} = 0] \wedge [q_{\mathbf{j}}^{0,3} = 1]$$
$$h_{\mathbf{i}}^{1,2,6,0,2} \triangleq [t \neq 0] \wedge [q_{\mathbf{i}}^{1,2,6} = 1] \wedge [q_{\mathbf{j}}^{0,3} = 0]$$

3.4.3 Determinism Of The Discrete Model

Recall from *Determinism of a Structured Discrete Automaton* that the determinism of a *SDA* was defined in terms of the solvability of the active set of transitions, $F^{\top}$ for some υ. The transitions of the model were defined in terms of variables and symbols in equation (3.12). For a given variable $q_{\mathbf{i}}^{\mathbf{w}}$ there were $|S^{z^{\mathbf{w}}}|$ transitions, one per domain symbol. Lets denote this set $E_{\mathbf{i}}^{\mathbf{w}}$. Each transition equation in this set is on the form $f \triangleq \dot{q}_{\mathbf{i}}^{\mathbf{w}} = 1$, and the transition guards in this set are mutually exclusive. This means that if $j_0, j_1 \in S^{z^{\mathbf{w}}}$ for some variable $q_{\mathbf{i}}^{\mathbf{w}}$, then:

$$g_{\mathbf{i}}^{\mathbf{w},j_0}(\upsilon) \implies \neg g_{\mathbf{i}}^{\mathbf{w},j_1}(\upsilon) \tag{3.18}$$

As a consequence, $\forall q \in \mathbb{Q}$, $F \triangleq F^{\top}(\upsilon)$ will be on the form $\dot{\bar{q}} = F(\upsilon)$, and $\nexists k_0, k_1 \in |F^{\top}(q)|$ such that $\bar{q}_{k_0}$, and $\bar{q}_{k_1}$ are the same variables. That would contradict equation (3.18). Therefore the automaton is globally deterministic. Here, because the right hand side is a constant one, even if such k_0 and k_1 existed, F would still be uniquely solvable as one of them is a redundant equation. However, the more interesting result is that an automaton constructed in this manner, is globally deterministic for any right hand side $f(\upsilon)$.

3.4.4 Zenolessness Of The Discrete Model

Eventually it will be the case that $\nexists(f, g) \in E$ such that $g(v) = \top$; eventual $F^{\top}(v) = \emptyset$, and the execution will stop. The zenolessness of the model is showed by exhaustion. Every variable will eventually converge, and since every variable converges the model is zenoless.

The guards in *Discrete Control, Flow and Event Reset Guards* are only potentially active in $t = 0$. In addition any guard reacting to a detected event is constrained to the first timestep. These guards are termed event reacting guards. As discussed in *Discrete Control, Flow and Event Reset Guards*, the detected events from the last discrete execution are active in the first timestep. execution.

The guards in *Discrete Leg Guards* are either conjoined to an active control variable, which converges to zero after one step, or are event reacting. All the guards in *Discrete Joint Guards*, *Discrete Cutting Event Guards* and *Discrete Cutting Event Guards* are also event reacting.

The *Discrete Source Guards* are logic variables defined in terms of the leg and flow of a tub, or halfpipe, and endpipe. Because the leg and flow will converge after the first timestep, the source variables will also converge. The equation (3.17) will, thus, also converge.

The guard of the midpipe branching event in *Discrete Branching Event Guards* is conjoined with a control variable being high, and thus the midpipe branching event will converge in the first timestep. Tub branching events conjoins a tub leg, tub flow, and valve joint, which all converge in the first timestep. Thus tub branching events also converge. Similarly endpipe branching events conjoins a halfpipe leg, endpipe flow, and valve joint which all converge in the first timestep. Thus all branching events converge. This means that the branching count in equation (3.16) will also converge.

Finally the guards of *Discrete Seeding Event Guards* and *Discrete Collision Event Guards* are all defined in terms of convergent branching counts and variables. The guards of *Discrete Receding Event Guards* are all defined in terms of a convergent source count and convergent variables. Thus all these guards and the variables whose transitions they guard also converge. And since every variable converges, the model is zenoless.

3.4.5 Implementing The Discrete Model

The discrete model was implemented in [RH22]. Structured discrete automata were constructed, and executions generated for a sample of initial value problems. The sequence of discrete simulations mirror the expected discrete subexecutions that a hybrid execution of a hypothetical *SHA* would produce. The events detected in these sequences would be instrumented by a *SHA* with actions to ensure that the continuous state would satisfy the next continuous subexecution. In the diagrams of these executions, $q_{i,j}$ will be used denote the jth state of the ith discrete subexecution. Similarly for the discrete time trajectories and $t_{i,j}$.

The initial state of the first execution in each of these sequences, $q_{0,0}$ is known a priori, and the initial state of the ith execution, $q_{i,0}$, is the final state of the $i - 1$th execution, $q_{i-1,m}$, sometimes modified with certain control variables set to one. These control variables are roots of the *SHA* and would be computed by the root subexecution.

The sequences are visualised with programatically drawn state diagrams using [RH22]. Control variables are not drawn.

Filling The Simple Network Top Down

In this sequence the simple system in *Example 3.3.1 (Basic Flow Network)* is filled from top down. The discrete model starts dry and empty in $q_{0,0}$, with the upstream tub leg control variable set high. The executions eventually fill the pipe network from top to bottom, detecting branching and seeding events in the process. The sequence of $q_{0,0}$, $q_{i,m}$ is visualised in the figures Fig. 2.1 through Fig. 2.6 in *Diagrams For Filling The Simple Network Top Down*.

Draining The Simple Network Top Down

In this sequence the simple system in *Example 3.3.1 (Basic Flow Network)* starts out filled, just like it ended in *Filling The Simple Network Top Down*. The tub control of $c_{0,0}^{0,0}$ is set high, indicating that the tub has just run dry. The sequence eventually drains the pipe network from top to bottom, detecting cutting and receding events in the process. Continuous flow and volume variables of the receding components would for example be set to zero. The sequence of $q_{0,0}$, $q_{i,m}$ is visualised in the figures Fig. 2.7 through Fig. 2.11 in *Diagrams For Draining The Simple Network Top Down*.

CHAPTER

FOUR

DISCUSSION

The overarching question of the report, as set out in *Goals*, was the organisation of computations and in particular the construction of a suite of interrelated computational models supporting a technical labour process. It was assumed that construction of computational models through left-to-right transformations of system models was an efficient way to do this. Practical computation was highlighted as a useful system modelling tool. The framing of the report was one of hydropower production planning, and the goal was to construct a system model of a watercourse. The *DAE* and *HA* were identified as suitable system modelling frameworks for this construction. The *HA* enables the combination of the *DAE* flow networks in [JT14] with the hybrid system theoretical approach to the modelling of temporary flow paths of [OM10]. The former meets the challenges of topological complexity and the latter meets the challenges of complex waterbody distributions and transient patterns of flow. These two challenges were assumed to be main obstacles to the direct implementation of a satisfactory computational model for simulation, and the cause of ad-hoc solutions in computational software. It was assumed that direct implementation was preferable to ad-hoc solutions.

4.1 Modelling Framework and Modelling Experience

This report is a snapshot of a work in progress - the construction of a new hybrid system theoretical model called a multibody flow network. This model pushes the envelope of hybrid automata in terms of discrete detail. The tooling required to support such a complex construction, the tooling that enables practical computation with hybrid automata, is an area of active research ([Zim10], [KM18], [CN21]). During the construction of the discrete part of this model there was a definite need for practical computation to debug, verify, and test designs - and the lack of a directly representable model made it difficult. The modelling frameworks set out in *Structured Discrete Automata* and *Structured Hybrid Automata* was a response to that problem. The observation in [Cam95] will be repeated for a second time:

> Being able to do engineering design and computer simulation directly on these original equations would lead to faster simulation and design, permit easier model variation, and allow for more complex models.

The discrete and hybrid structured automata were designed with an eye on direct computation in the face of highly detailed discrete dynamics. The separation of the hybrid automaton into a discrete and hybrid part enabled the definition and analysis of the discrete aspects of the model to be done separately from the hybrid construction. Defining the transition guards as functions of discrete time made it easier to construct a model whose zenoness could be reasoned about. The execution of a *SHA* formalises the event detecting approach described in *Simulation of Hybrid Automata* and enables a direct implementation of its construction. Future work in this direction is outlined in *Iterating on Structured Automata*.

The discrete multibody flow network defined in *Discrete Flow Network Model* is the discrete part of a hybrid multibody flow network model. It can represent arbitrary distributions of waterbodies and patterns of flow in a network of valves and pipes and is a basis for the continuous dynamics of a *SHA*. It can detect the discrete events a *SHA* would need to handle the changing continuous algebraic constraints of a multibody flow network. In this respect, it is a sufficient basis for the construction of the *SHA*. Executions of the discrete model were successfully constructed and illustrated using an implementation of the algorithm in *An event-based coroutine for simulating a Structured Discrete Automata*. The discrete model was shown to be deterministic and zenofree, which are useful results in the context of practical computation. However, it does fall short of being a satisfactory basis for the construction of a system model of a tunnel network in the context hydropower production planning. It thus falls even shorter of being a system model for a hydropower producing watercourse. The latter was set out as an explicit goal in *Goals*, and has not been met. There is still some way to go before such a model can be constructed. This will be addressed in *Iterating the Discrete Model*, *Constructing the Hybrid Model*, *Constructing a Watercourse Model*, and *Controlling a Watercourse Model*.

4.2 Implementation Experience

The implementation of these frameworks were, like [Zim10], written in C++. In this report a library-based approach was taken. This sidesteps the issue of designing a modelling language, which is a common approach when it comes to the design of computational software for hybrid systems. Both [Zim10] and [CN21] take a language-based approach and discuss previous language-based solutions. A dedicated modelling language has the potential for ergonomic and elegant model definitions in a way that is hard to replicate, even with a well-designed library *API*. However, in an operative context, like the one assumed in this report (see Fig. 1.7), no computation or model transformation can assume to have a monopoly on the representation of a system model. Its representation needs to be accessible for a plethora of different computations and applications. A textual representation, in the form of a modelling language designed for humans, is not a practical data format in this setting. General purpose data formats designed for programmatic consumption, supported across a variety of multi-purpose languages, is likely to be used instead.

A library for an existing multi-purpose programming language, alternatively combined with bindings in an existing scripting language, lets the modeller take advantage of an already existing ecosystem of tooling and packages. A language and ecosystem that the modeller is, likely, already somewhat comfortable with. This makes practical computation more ergonomic and flexible without the need for intermediate formats to define models and store computational results. There is no contradiction between programmatic and language-based construction, but the benefits of the latter do not seem to be worth the effort in this setting, and here, a library-based approach is more pragmatic.

Structured hybrid automata were defined in terms of structured discrete automata. As a result the implementation of the latter and the construction of its execution could be directly reused by the implementation of the former. The algorithms for constructing executions were implemented as event-based coroutines. The construction of the continuous executions were implemented using the root finding *DAE*-solver IDA from [HBG+05]. The library was fitted with Python bindings. The events of a construction are coroutine suspension points, the events are derived directly from the defintions of the executions. This in turn means that the constructions are suspendable. This enable fast iteration as a construction can be suspended according to experiment-specific logic.

Python has a rich ecosystem of data analysis and visualisation tools which can be used directly during the course of such a construction. A modeller is able to define an event continuation in Python that can suspend the construction and / or inspect its current state. The state of the construction includes access to a symbolic representation of the constructions components. This is a consequence of the modelling framework being directly implemented. The computational software itself needs a direct symbolic representation of these components to construct the execution. The discrete diagrams seen in *Discrete Flow Network Model* and *Discrete Model Diagrams* were programmatically generated in this manner by combining the logical description of the system, with the state of the construction, and rendering these with Python packages [P+22] and [N+22]. The visualisation of complex discrete states in diagram form was of great help during the construction of the discrete model. The simulations, and associated diagrams ,of *Example (HA Flower System)* were also generated in a similar manner using the Python package [Hun07]. Another useful tool was the rendering of the currently active set of transitions, activities, actions, and roots in LaTeX for debugging purposes during the course of an execution.

Coroutines is a new addition to C++ (see [BXHP21] for a recent application) and can not be considered to be a production-ready feature yet. The implementation turned out more difficult than it had to be, as the compiler kept running into internal compiler errors. Nevertheless, being able to suspend the construction of an execution is a useful quality in the context of practical computation. Further work in this direction is described in *Iterating on Structured Automata*.

4.3 Future work

4.3.1 Iterating on Structured Automata

This is the initial iteration of structured automata. There are no doubt errors, both in definitions and implementation, which need to be worked out. In addition this report left out the definition of determinism and zenoness of the *SHA*, which are important qualities in the context of practical computation and system modelling. Even though a *SDA* is zenoless, the hybrid execution can still get stuck in a sequence of discrete executions. This potential zenoness is inherent to the *Definition 3.2.9 (Hybrid Execution)*, and made obvious in Fig. 3.5.

In *Structured Discrete Automata* the case of an overdetermined set of transition equations was set aside. *Definition 3.1.3 (Discrete execution)* was constructed in terms of solvable transition equations. An overdetermined set of transition equations has an obvious interpretation; that of nondeterminism. A more powerful definition of the execution of a *SDA* would also include executions that bifurcate as a result of overdetermination. This would be needed be able to represent $x \in D(q) \cap G(e)$ as $x \in G(e_0) \cap G(e_1)$. The convergence of a discrete execution was defined with respect to $\dot{q}$ instead of $G^\top$, which is less ergonomic in the case one wants to construct a non-blocking automaton, as it requires a slightly more complex discrete dynamics than one defined in terms of $G^\top$.

In *Structured Hybrid Automata* the actions were defined in terms of a variable, function, and guard. It was assumed that no two actions could be active at the same time, for the same variable. A more powerful definition would use an implicit equation instead of a variable and function. This would be similar to the definitions of the continuous dynamics of the *SHA* or discrete dynamics of the *SDA*. This creates potential for ambiguity and bifurcation, but it is a necessary generalization to efficiently represent the actions required by a *SHA* of a multibody flow network. In particular it would be required when handling collisions between inelastic waterbodies. These collisions require solving affine and acausal systems of equations, constructed based on the current discrete state. It was also assumed that no two roots referred to the same discrete variable. This is an unnecessary simplification, but a more powerful definition will again create potential for ambiguity and bifurcation.

The discrete multibody flow network model was manually shown to be zenofree and deterministic by considering the convergence of guards, variables and counting variables. This is a tedious and fragile approach which should be improved. The manual approach suggests an algorithmic solution for *SDA*s like the discrete multibody flow network. In this case determinism could be sufficiently checked, for example, by checking mutual exclusivity of the transition guards of a variable, combined with uniqueness of solution of the corresponding transition equations.

Structured automata are designed for systems modelling, and only need to enable practical computation as a modelling tool. However, the computations of production planning (*Computations of Hydropower Production Planning*) are done in an operative setting with highly varied hardware, and a high degree of regularity. Computational input varies only slightly in two consecutive computations. This is an interesting context for investigating the efficient computational architecture of hybrid systems. A *JIT*-compilation based approach, as demonstrated in [CN21], can simplify the software distribution model, and at the same time generate efficient executable code in the face of hardware variation. The high degree of regularity of computations provides interesting possibilities with regards to memoisation of causalised & compiled continuous dynamics in commonly inhabited discrete states of hybrid executions. This would come at the cost of the *JIT*-compilation itself, as well as the added complexity of distributing a *JIT*-compilation framework.

The construction of the executions of a nondeterministic *SDA* or *SHA* will need to handle bifurcation points. The initial implementation took the liberty of assuming that the automata were deterministic. *Zenoness of a Hybrid Automaton* quoted Schaft discussing the handling of such a bifurcation [Sch00]:

> In such situations the simulation software should provide a warning to the user, and if it is difficult to make a definitive choice between several possibilities perhaps the solver should even work out all reasonable options in parallel.

Implementation approaches to the construction of nondeterministic executions of structured automata is a very interesting problem, particularly in the light of the implementation strategy in this report. The frame of a C++ coroutine does not, currently, have value-semantics. It is an opaque object, and there is no simple way to copy the entire state of a coroutine if the members of the frame are not trivial. However, if coroutine frames had value-semantics, such that a coroutine frame was copyable if all its members were copyable, it would become trivial to implement bifurcations. The current coroutine-frame of an execution, suspended at a bifurcation event could be copied, and each copy instructed to follow a different execution.

4.3.2 Iterating the Discrete Model

The discrete model in *Discrete Flow Network Model* has a shortcoming as it does not model valve openings. The modelling of valve openings is crucial for an accurate representation of a watercourse, and a necessary addition before constructing a watercourse system model that meets the demands of *Goals*. The upside is that the logical description of the flow network system already can facilitate the opening and closing of valves on an per-incident component basis (*Definition 3.3.5 (Incident Components)*) by adding suitable discrete endpipe and tub variables. In addition it would greatly increase the expressive power of the model.

Nor is this iteration of the discrete model likely to be a minimal expression of its discrete dynamics, and might have both redundancies and errors. The sequencing of transitions were done in a coarse manner, and the guards of the transitions were designed to be mutually exclusive so that it would become easier to reason about determinism and zenoness. Perhaps there is potential for relaxation and improvement. One such example would be to make the tracking variables function as PD-regulators instead of P-regulators. This would ensure faster convergence of the discrete state. On the other hand it would also cause a $F^\top(v)$ that was linear in $\dot{q}$ to have a more complex structure, and impose sequencing on its computation.

Nor is the proposed discrete model able to support elastic waterbodies. Consider again the discrete sequence of Fig. 3.9 and Fig. 3.10. Here waterbodies were assumed to be inelastic, and so the endpipe flow variables of the pipe would be aligned during the course of collision resolution. In the case of an elastic waterbody, the flow directions would remain the same until the pressure of the water column in the pipe eventually flips one or both of the continuous flows. Now imagine the scenario where, at the point of collision, both valves recede. The waterbody in the pipe would float midair, as it were. To model this state discretely, one might need to add a midpipe volume or droplet variable. This is not required for being an adequate model for the computations of hydropower production planning, where only the aggregate movement of water, on a scale much larger than the volume of a pipe, is of interest.

Finally, there is an issue with discrete “droplets”. Small waterbodies might become stuck in the pipe network. A waterbody can only move through a pipe if it is big enough to fill the entire pipe, even though it can always retreat back into the valve. In the case where gravity is pulling water into the pipe network from a tub, and the tub runs dry, or the valve is shut before the emergent waterbody is large enough to fill a pipe, it will simply get stuck. It will stay there until the tub starts feeding it water again, or it is merged with another waterbody from below. The scale of droplets is also too small to cause much concern in this context.

4.3.3 Constructing the Hybrid Model

The discrete multibody flow network is the point of departure for a hybrid multibody flow network. The addition of opening and closing of valves will require the consideration of over- and underdetermined continuous dynamics. Consider for example a pipe that is fully submerged but closed at both ends. A construction of the continuous dynamics of a *SHA*, based only on local information, might add two algebraic constraints - one for each end of the pipe. For an inelastic body of water, where the flow into a submerged pipe is equal to the flow out of a submerged pipe, this will lead to an overdetermined system of equations with redundant algebraic constraints. Similarly, with no connected valves, the equations governing the pressure in model with quasi-stationary continuous dynamics will be underdetermined. This underdetermination is not just a transient consequence of symbolic processing, as in [Zim13], it is the persistent continuous dynamics of the system which last until one of the valves are opened. Elastic waterbodies would have consequences on the discrete model as mentioned in *Iterating the Discrete Model*.

The computational software used to solve the consistent initialisation problem and to construct the continuous execution in this report is based on regular *DAE*s, and use a *DAE* index that is not suitable for over- and underdetermined systems, [Kun06]:

> Although the concept of the differentiation index is widely used, it has a major drawback, since it is not suited for over- and underdetermined systems. The reason for this is that it is based on a solvability concept that requires unique solvability.

The unique and global solvability result of [JMT15] requires at least one demand and pressure node connected to the network. The problem of over- and underdetermination of the *DAE*s, generated during the construction of an execution of a hybrid automaton, will likely become a point of focus for the construction of the hybrid model.

Another problem that needs to be dealt with is that of continuous actions. The discrete model detects branching, collisions, separations, and recessions. Recessions and separations are easy enough to deal with; they simply require setting the corresponding continuous flow variables to zero. Collisions and branching require more careful consideration. In particular, the collision of two inelastic waterbodies requires solving an affine system of algebraic equations of flow. This system of equations must be constructed based on the discrete state at the point of collision. The challenges of event detection and zero-crossing, described in [ZYM08], poses yet another problem; is it even feasible to simulate a model with such a high level of discrete detail?

4.3.4 Constructing a Watercourse Model

A *SHA* of a multibody flow network is one of many system models that make up a watercourse. Constructing a system model of an entire hydropower producing watercourse was the goal set out in *Goals*. Models of reservoirs, catchments, river systems, and power production remains to be constructed. These systems, in the context of production planning, are not required to be modelled at a high level of discrete detail, and are trivial constructions in comparison to the one in *Discrete Flow Network Model*. With a *SHA* of a watercourse in hand, the watercourse models, of the associated computational models, could then be constructed through suitable left-to-right transformations.

4.3.5 Controlling a Watercourse Model

Optimal control is a cornerstone computation in the context of production planning. The question of control is even relevant for plan simulations. It is not always the case that the production planning computation of *Computations of Hydropower Production Planning* computes plans detailed enough for simulation. The plan might be computed for a lumped group of units, whereas the simulation treats each unit separately, which requires the plan simulation itself to deal with the problem of optimal control.

This might lead to ad-hoc solutions in simulation software, which faces the problems described in *The Tunnel Systems of a Hydropower Producing Watercourse*. Some producers do not do production planning per se, and rely only on this limited planning functionality in the plan simulation; maiming two very different animals with one stone. [OM10] formulates an *MPC* problem based on a hybrid model for wastewater systems. Treating the limited planning that the plan simulation does as a wastewater system, and simply limiting the flood, while softly satisfying production plans might be a good starting point for constructing a formal model of this limited form of planning. An *MLD* similar to the one used in [OM10] could be constructed by applying left-to-right model transformations of a watercourse *SHA*.

CHAPTER

FIVE

CONCLUSION

Practical computation is an important tool for construction of hybrid automata with a high level of discrete detail. Structured automata is a modelling framework designed with such constructions in mind. It decomposes the hybrid automata into a structured discrete automaton and structured hybrid automaton. The transition guards of a structured discrete automaton are dependent on the discrete time of an execution which enables the sequencing of transitions in time, which makes it easier to reason about the zenoness of a model. The execution of a structured hybrid automaton is a formalisation of the event detecting approach to simulating hybrid systems and in particular makes the relation between a continuous root and a discrete event explicit. It is a step in the direction of a more formalised approach to the simulation of a hybrid automaton.

A multibody flow network is a new hybrid system theoretical model combining the continuous dynamics of the flow networks of [JT14] with the temporary flow paths [OM10] to support arbitrary distributions of waterbodies and arbitrary flow paths in a tunnel network. Its discrete part was implemented and simulated as a structured discrete automaton, and shown to be deterministic and zenofree. The discrete multibody flow network is able to represent arbitrary distributions of waterbodies and patterns of flow in a network of pipes and valves, and detects the branching, splitting, colliding, and receding of waterbodies as discrete events. It is a discrete base from which a structured hybrid automaton of a multibody flow network can be constructed.

Structured automata, the construction of their executions, and the discrete multibody flow network, was implemented in [RH22]. Both the framework and the construction of executions were implemented as C++ libraries with Python bindings. The construction of the executions were implemented as event-based coroutines. These events are derived directly from the definition of the execution of the automaton. Because the construction of the execution was implemented in a direct manner, the implementation of the construction needed a symbolic representation of every single component of its formal definition. In the continuation of an event the modeller gets direct access to this symbolic representation in the current state of the construction of an execution. The combination of this access, suspendable constructions, and bindings in a scripting language made it easy to debug and test designs. It also enabled the programmatic construction of diagrams that helped visualise complex discrete dynamics. This framework and implementation approach is well-suited to support the construction of hybrid automata with practical computation.

APPENDIX

A

NOTATION

Below is a collection of notational conventions that are considered idiosyncratic and thus are defined explicitly.

Definition

A definition is denoted $A \triangleq B$, while a relation is denoted $A = B$.

Powersets

Powersets are denoted with $\mathbb{P}$.

Index sets

$\{N, \ldots, M - 1\}$ is denoted $N \leq \mathbb{Z} < M$, if N is omitted it is taken to be zero. $\mathcal{B}$ is defined as the set $\mathcal{B} = \mathbb{Z} < 2$. If a natural number is used where a set expected it is taken to mean an equivalent index set. $x \in 5$ thus means $x \in Z < 5$.

Structured and Unstructured Super and Subscripts

Symbols with structured super and subscript $x^{i\cdots}_{j\cdots}$ where $i_k \in \mathbb{Z} < N_k$ and $j_k \in \mathbb{Z} < M_k$ are alternatively written with an unstructured $x^{\mathbf{i}}_{\mathbf{j}}$, with **i** defined as:

$$\mathbf{i} = \sum_{k_0} (\prod_{k_1} N_{k_1}) \cdot i_{k_0}$$

- with $I = \{i \ldots\}$;
- and with $K_0 = Z < |I|$, and $k_0 \in K_0$;
- and with $K_1(k_0) = k_0 < Z \leq |K|$, and $k_1 \in K_1(k_0)$;
- and with $N_{|I|} = 1$.

and similarly for **j** and M.

APPENDIX

B

DISCRETE MODEL DIAGRAMS

B.1 Diagrams For Filling The Simple Network Top Down

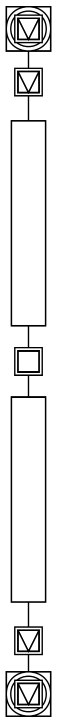

Fig. 2.1: State diagram for G_0 at $t_{0,0}$, with the tub leg control variable set high.

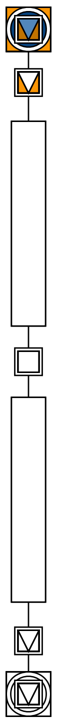

Fig. 2.2: State diagram for G_0 at $t_{0,m}$. Here the tub $c_0^{0,1}$ has become a source, as can be seen by the flow-arrow being drawn in light blue. A branching event has been detected for the tub, and seeding events has been detected for the valve and incident endpipe.

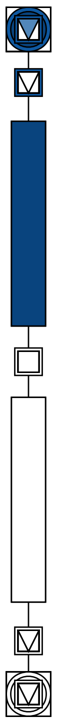

Fig. 2.3: State diagram for G_0 at $t_{1,m}$. This execution had no control signals set high. The discrete execution converged after the newly formed waterbody has entered the halfpipe $c^{1,1}_{0,0}$.

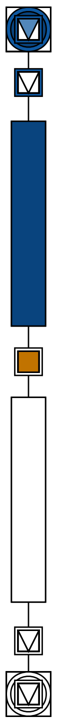

Fig. 2.4: State diagram for G_0 at $t_{2,m}$. Here the midpipe joint control $q_0^{1,0}$, is set high, which indicates that the combined volume of halfpipes $c_{0,j}^{1,1}$ has filled the pipe. The execution converges by detecting a midpipe branch event.

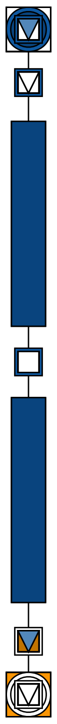

Fig. 2.5: State diagram for G_0 at $t_{3,m}$. This execution has no control signals set high, and is simply processing the events detected in τ_3. The waterbody fills the pipe, and the endpipe $c^{1,2}_{0,1}$ has become a source for the valve c^0_1, as indicated by the light blue shading of the flow triangle. A branch event has been detected for the endpipe, and seding events have been detected for the valve and tub.

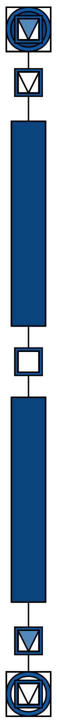

Fig. 2.6: State diagram for G_0 at $t_{4,m}$. This execution has no control signals set high, and is simply processing the events detected in τ_4. The waterbody has now completely filled the network and water is in freefall into the tub $c_1^{0,0}$.

B.2 Diagrams For Draining The Simple Network Top Down

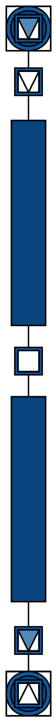

Fig. 2.7: State diagram for G_0 at $t_{0,0}$, with the tub leg control variable set high. The upstream tub is about to run dry.

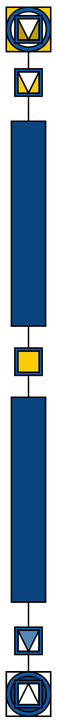

Fig. 2.8: State diagram for G_0 at $t_{1,m}$. Here a tub cutting has been detected, as a consequence, the valve, upstream endpipe, and midpipe are receding.

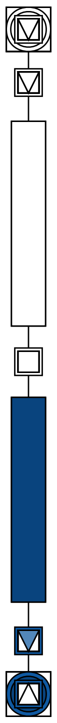

Fig. 2.9: State diagram for G_0 at $t_{2,m}$, with the downstream halfpipe leg control variable set high. The upstream tub, valve, and halfpipe have all run dry, and the downstream halfpipe is about to recede.

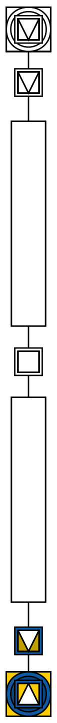

Fig. 2.10: State diagram for G_0 at $t_{3,m}$. The downstream halfpipe has receded, and and the downstream endpipe has been cut. The source count of the valve is 0, and the valve and the tub are now receding.

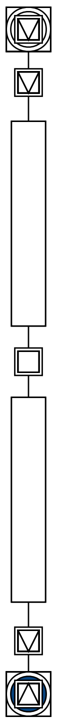

Fig. 2.11: State diagram for G_0 at $t_{4,0}$. The network has finally run dry. Note that the tub leg is still wet, but the tub flow is negative. The pressure inside the valve network is preventing the waterbody from entering.

EXTERNAL CONTENT

The implementation of the modelling frameworks of *SDA*, *SHA*, as well as the construction of their executions were implemented as a C++ library with python bindings in [RH22]. This is a git repository hosted on GitLab. The discrete model of a multibody flow network is implemented in the same repository on the models-multibody_loss_network branch. The following attached python-scripts used this library to generate the executions and diagrams used in this report:

- *ha_flower_figures.py*
- *pwa_flower_figures.py*
- *discrete_model_simulation_figures.py*
- *discrete_model_state_figures.py*
- *structured_da_figures.py*

Some snippets from *ha_flower_figures.py* used to generate the figures in *Example (HA Flower System)* are shown in *Symbolic representation of A*, *Constructing the SDA*, *Constructing the SHA*, and *Constructing the initial value problem.*.

Listing 3.1: Symbolic representation of A

```
A = [ [ [ program.negation(epsilon), program.multiplication([alpha,omega]) ],
        [ program.negation(omega),   program.negation(epsilon)             ] ],
      [ [ program.negation(epsilon), omega                                 ],
        [ program.multiplication([program.negation(alpha), omega]), program.
↪negation(epsilon) ] ] ]
```

Listing 3.2: Constructing the *SDA*

```
dautomaton = dautomatons.BasicAutomaton(
  variables = dvariables,
  transitions = [
    dautomatons.Transition(
      variable = 0,
      difference = program.one(),
      guard = program.conjunction([program.complement(W[i]),region[i]])
    ) for i in regions
  ])
```

Listing 3.3: Constructing the *SHA*

```
hautomaton = hautomatons.BasicAutomaton(
  discrete_automaton = dautomaton,
  continuous_variables = cvariables,
  activities = [
    hautomatons.Activity(
      equation = equations[i][j],
      guard    = W[i]
    ) for i,j in product(regions,variables)
  ],
  actions = [])
```

Listing 3.4: Constructing the initial value problem.

```
problem = hsimulators.Problem(
  program = program,
  automaton = hautomaton,
  state = hsolvers.BasicState(
    discrete   = dsolvers.BasicState(
      independent = 0,
      dependent = [ locations[0], [0] ]),
    continuous = csolvers.BasicValuation(
      independent = t_begin,
      dependent  = initial_continuous_state).to_state()),
  solver = hsolvers.BasicSolver(
    discrete = dsolvers.BasicSolver(),
    continuous = csolvers.BasicSolver(
      config = csolvers.BasicConfig(
        tolerance = csolvers.BasicTolerance(relative = 1.0e-2,absolute = 1.0e-3),
        step = csolvers.BasicSolver.fixed_step(step = t_delta),
        stop = t_end))))
```

Listing 3.5: Constructing a hybrid trajectory.

```
solution = hsimulators.solve(problem = problem,control = lambda xs,xe:(xs,hsimulators.
↪make_event(xe)))
trajectories = []
while xframe := solution():
  xsolution,xevent = xframe
  match xevent.tag:
    case dsimulators.init:
      trajectories.append((xevent.state,[]))
    case dsimulators.stop:
      trajectories.append((xevent.state,[]))
    case csimulators.step | csimulators.start | csimulators.root:
      trajectories[-1][-1].append(xevent.state.to_valuation(problem.automaton.
↪continuous_variables))
    case csimulators.stop:
      break
    case _:
      pass
```

APPENDIX

D

GLOSSARY

API
: Application Programming Interface

DA
: Discrete Automaton

DAG
: Directed Acyclic Graph

DAE
: Differential Algebraic Equation

DDP
: Dynamic Differential Algebraic Equation Processor

HA
: Hybrid Automaton

JIT
: Just In Time

MILP
: Mixed integer linear programming.

MLD
: Mixed Logical-Dynamical System, see [LLL09] for more.

MPC
: Model Predictive Control

ODE
: Ordinary Differential Equation

PWA
: Piecewise Affine

SDA
: Structured Discrete Automaton

SHA
: Structured Hybrid Automaton

BIBLIOGRAPHY

[AGH+19] Rajeev Alur, Mirco Giacobbe, Thomas A. Henzinger, Kim G. Larsen, and Marius Mikučionis. Continuous-time models for system design and analysis. In *Computing and Software Science*, Lecture Notes in Computer Science, pages 452–477. Springer International Publishing, Cham, 2019.

[And94] Mats Andersson. Object-oriented modeling and simulation of hybrid systems. 1994.

[Arn17] Martin Arnold. Dae aspects of multibody system dynamics. In *Surveys in Differential-Algebraic Equations IV*, Differential-Algebraic Equations Forum, 41–106. 2017.

[BLG91] L. R. Petzold B. Leimkuhler and C. W. Gear. Approximation methods for the consistent initialization of differential-algebraic equations. *SIAM Journal on Numerical Analysis vol. 28 iss. 1*, feb 1991.

[BXHP21] Bruce Belson, Wei Xiang, Jason Holdsworth, and Bronson Philippa. C++20 coroutines on microcontrollers—what we learned. *IEEE Embedded Systems Letters*, 13(1):9–12, 2021. doi:10.1109/LES.2020.2973397.

[BM99] Alberto Bemporad and Manfred Morari. Control of systems integrating logic, dynamics, and constraints. *Automatica (Oxford)*, 35(3):407–427, 1999.

[Bre96] Kathryn Eleda Brenan. Numerical solution of initial-value problems in differential-algebraic equations. 1996.

[Cam15] Stephen L Campbell. The flexibility of dae formulations. In *Surveys in Differential-Algebraic Equations III*, Differential-Algebraic Equations Forum, pages 1–59. Springer International Publishing, Cham, 2015.

[Cam95] Stephen L. Campbell. Linearization of daes along trajectories. *Zeitschrift für angewandte Mathematik und Physik*, 46(1):70–84, 1995.

[CastelloGrinoBasanez98] Ramón Costa Castelló, Robert Griñó, and Luis Basañez. Dae methods in constrained robotics system simulation. In *Computación y Sistemas*, Computación y Sistemas. 1998.

[CK06] François E Cellier and Ernesto Kofman. *Continuous System Simulation*. Springer US, Boston, MA, 2006. ISBN 9780387261027.

[CN21] Guerric Chupin and Henrik Nilsson. Modular compilation for a hybrid non-causal modelling language. *Electronics (Basel)*, 10(7):814, 2021.

[Dao14] Prodromos Daoutidis. Daes in model reduction of chemical processes: an overview. In *Surveys in Differential-Algebraic Equations II*, Differential-Algebraic Equations Forum, pages 69–102. Springer International Publishing, 2014.

[DSHLP09] B De Schutter, W. P. M. H Heemels, J Lunze, and C Prieur. Survey of modeling, analysis, and control of hybrid systems. In *Handbook of Hybrid Systems Control*, pages 31–56. Cambridge University Press, 2009.

[Die16] Reinhard Diestel. *Graph Theory: 5th Electronic Edition*. Springer, Berlin, 2016. ISBN 3662536218.

[Gea90] C. W. Gear. Differential algebraic equations, indices, and integral algebraic equations. *SIAM Journal on Numerical Analysis vol. 27 iss. 6*, dec 1990.

[GC91] Jurgen Greifeneder and Francois E Cellier. *Continuous System Modeling*. Springer, 1991. ISBN 9780387975023.

[Hed99] Sven Hedlund. Computational methods for hybrid systems. 1999. Licentiate Thesis. URL: https://lup.lub.lu.se/search/files/4499784/8566368.pdf.

[HSB+20] Hindmarsh, Alan, Radu Serban, Cody Balos, Gardner David, Woodward Carol, and Reynolds Daniel. *User Documentation for IDA*. Lawrence Livermore National Laboratory, 2020.

[HBG+05] Alan Hindmarsh, Peter Brown, Keith Grant, Steven Lee, Radu Serban, Dan Shumaker, and Carol Woodward. Sundials: suite of nonlinear and differential/algebraic equation solvers. *ACM transactions on mathematical software*, 31(3):363–396, 2005.

[Hop79] John E Hopcroft. Introduction to automata theory, languages, and computation. 1979.

[Hun07] J. D. Hunter. Matplotlib: a 2d graphics environment. *Computing in Science & Engineering*, 9(3):90–95, 2007. doi:10.1109/MCSE.2007.55.

[JMT15] Lennart Jansen, Michael Matthes, and Caren Tischendorf. Global unique solvability for memristive circuit daes of index 1. *International Journal of Circuit Theory and Applications*, 43(1):73–93, 2015. URL: https://onlinelibrary.wiley.com/doi/abs/10.1002/cta.1927, arXiv:https://onlinelibrary.wiley.com/doi/pdf/10.1002/cta.1927, doi:https://doi.org/10.1002/cta.1927.

[JT14] Lennart Jansen and Caren Tischendorf. A unified (p)dae modeling approach for flow networks. In *Progress in Differential-Algebraic Equations*, Differential-Algebraic Equations Forum, pages 127–151. Springer Berlin Heidelberg, Berlin, Heidelberg, 2014.

[KSS07] Nand Kishor, R.P Saini, and S.P Singh. A review on hydropower plant models and control. *Renewable and Sustainable Energy Reviews*, 11(5):776–796, 2007.

[KGG+09] S Kowalewski, M Garavello, H Guéguen, G Herberich, R Langerak, B Piccoli, J. W Polderman, and C Weise. Hybrid automata. In *Handbook of Hybrid Systems Control*, pages 57–86. Cambridge University Press, 2009.

[Kun06] Peter Kunkel. Differential-algebraic equations : analysis and numerical solution. 2006.

[KM18] Peter Kunkel and Volker Mehrmann. Regular solutions of dae hybrid systems and regularization techniques. *BIT*, 58(4):1049–1077, 2018.

[LA04] Chris Lattner and Vikram Adve. LLVM: a compilation framework for lifelong program analysis and transformation. In *CGO*, 75–88. San Jose, CA, USA, Mar 2004.

[LLL09] Jan Lunze and Françoise Lamnabhi-Lagarrigue. *Handbook of Hybrid Systems Control: Theory, Tools, Applications*. Cambridge University Press, 2009. ISBN 9780521765053.

[LJS+03] J Lygeros, K.H Johansson, S.N Simic, Jun Zhang, and S.S Sastry. Dynamical properties of hybrid automata. *IEEE transactions on automatic control*, 48(1):2–17, 2003.

[N+22] Stephen North and others. Graphviz. https://gitlab.com/graphviz/graphviz, 2022.

[OM10] Carlos Ocampo-Martinez. Model predictive control of wastewater systems. 2010.

[OE17] Marting Otter and Hilding Elmqvist. Transformation of differential algebraic array equations to index one form. In *Proceedings of the 12th International Modelica Conference, Prague, Czech Republic, May 15-17, 2017*. Linköping University Electronic Press, July 2017.

[P+22] Keith Packard and others. cairo. https://gitlab.freedesktop.org/cairo/cairo, 2022.

[PAN88] C. C PANTELIDES. The consistent initialization of differential-algebraic systems. *SIAM journal on scientific and statistical computing*, 9(2):213–231, 1988.

[RJ00] A Rantzer and M Johansson. Piecewise linear quadratic optimal control. *IEEE transactions on automatic control*, 45(4):629–637, 2000.

[Ria13] Ricardo Riaza. Daes in circuit modelling: a survey. 2013.

[RH22] John Eivind Rømma Helset. hysj. https://gitlab.com/jehelset/hysj, 2022.

[Sch00] A.J. van der Schaft. An introduction to hybrid dynamical systems. 2000.

[Sim17] Bernd Simeon. On the history of differential-algebraic equations: a retrospective with personal side trips. In *Surveys in Differential-Algebraic Equations IV*, Differential-Algebraic Equations Forum, pages 1–39. Springer International Publishing, Cham, 2017.

[Skj19] Hans Ivar Skjelbred. Unit-based short-term hydro scheduling in competitive electricity markers. 2019.

[SHS21] Miriam Garc\'ıa Soto, Thomas A. Henzinger, and Christian Schilling. Synthesis of hybrid automata with affine dynamics from time-series data. In *Proceedings of the 24th International Conference on Hybrid Systems: Computation and Control*. ACM, may 2021. URL: https://doi.org/10.1145%2F3447928.3456704, doi:10.1145/3447928.3456704.

[Yan19] Weijia Yang. *Hydropower Plants and Power Systems: Dynamic Processes and Control for Stable and Efficient Operation*. Springer Theses. Springer International Publishing, 2019. ISBN 978-3-030-17241-1.

[ZYM08] Fu Zhang, Murali Yeddanapudi, and Pieter J Mosterman. Zero-crossing location and detection algorithms for hybrid system simulation. *IFAC Proceedings Volumes*, 41(2):7967–7972, 2008.

[ZJLS01] Jun Zhang, Karl Henrik Johansson, John Lygeros, and Shankar Sastry. Zeno hybrid systems. *International journal of robust and nonlinear control*, 11(5):435–451, 2001.

[Zim10] Dirk Zimmer. Equation-based modeling of variable-structure systems. 2010.

[Zim13] Dirk Zimmer. A new framework for the simulation of equation-based models with variable structure. *Simulation (San Diego, Calif.)*, 89(8):935–963, 2013.

INDEX

A

API, 93

D

DA, 93
DAE, 93
DAG, 93
DDP, 93

H

HA, 93

J

JIT, 93

M

MILP, 93
MLD, 93
MPC, 93

O

ODE, 93

P

PWA, 93

S

SDA, 93
SHA, 93