SL2SF: Refactoring Simulink to Stateflow

SL2SF: Refactoring Simulink to Stateflow

By

Stephen Wynn-Williams, B.Eng.

A Thesis

Submitted to the School of Graduate Studies

in Partial Fulfillment of the Requirements for the Degree

Master of Applied Science

McMaster University

Master of Applied Science (2021) McMaster University

(Software Engineering) Hamilton, Ontario

Title: SL2SF: Refactoring Simulink to Stateflow

Author: Stephen Wynn-Williams, B.Eng. (McMaster University)

Supervisor: Dr. Mark Lawford

Number of Pages: viii, 116

Abstract

The adoption of Model-Based Design in the development of embedded control systems across industries has led to the widespread use of Matlab Simulink/ Stateflow for modelling controller software. Within the Simulink environment, component systems may be modelled as either Simulink block diagrams or as Stateflow state charts. The choice of modelling formalism is informed by the nature of the system being modelled, with state charts being the recommended approach for systems which rely on stateful decision logic. However, in practice, systems with stateful decision logic are often modelled with complex block diagrams instead, hindering their readability.

This thesis presents a methodology for improving the maintainability and understandability of embedded software models by refactoring component systems which use block diagrams to model stateful logic into functionally equivalent state charts that represent the intended behaviour more naturally. This methodology establishes strategies for identifying such component systems within large industrial models, and for translating block diagrams into state charts.

The translation methodology uses Mealy machines as a semantic model for both block diagrams and state charts, and tabular expressions are used as an intermediate representation to bridge the syntactic gap between the

two modelling languages. Both identification and translation methodologies are designed to be automated. A prototype translation tool was developed, implementing the strategy presented in this work.

Acknowledgments

First and foremost, I express my most sincere gratitude to my supervisor, Dr. Mark Lawford. His guidance and encouragement were truly appreciated, and it was an honour to work for him.

Next, I would like to thank Dr. Zinovy Diskin for taking endless hours to teach me about category theory. I am incredibly grateful for our enlightening discussions and for your endless patience.

The invaluable expertise of Dr. Vera Pantelic is gratefully acknowledged. Thank you for your insightful comments, thoughtful edits, and excellent advice throughout my graduate studies.

To my colleagues from McSCert–thank you for your friendly company and for sharing your knowledge with me.

My unnamed industrial partner is gratefully acknowledged for their financial and technical assistance. Thank you for providing me with the opportunity to work on such an interesting engineering problem.

Lastly, I would like to thank my family and friends, especially Daniella Pallisco, for their unwavering support during my graduate studies.

Contents

List of Figures

List of Acronyms

HCT Horizontal Condition Table

MAB MathWorks Advisory Board

MBD Model-Based Design

STG State Transition Graph

STT State Transition Table

Chapter 1

Introduction

1.1 Motivation

Model-Based Design (MBD) is a software development methodology in which models specify the desired characteristics of software systems. In particular, this thesis focuses on the application of MBD for specifying behaviour. During the development process, models can be simulated in isolation to perform unit testing, or integrated with models of the environment to perform so-called "Model in the Loop" testing of system level functionality. Code is automatically generated from the developed models.

In MBD software development effort is focused on the models rather than the code they generate. In fact, developers are heavily discouraged from modifying generated software. Shifting from "source code" to "source models" means that characteristics of high quality software (namely maintainability) are assessed based on the quality of the models themselves. While metrics and tools exist for measuring and improving the quality of software written in traditional languages, applying these on the code in an MBD process offers

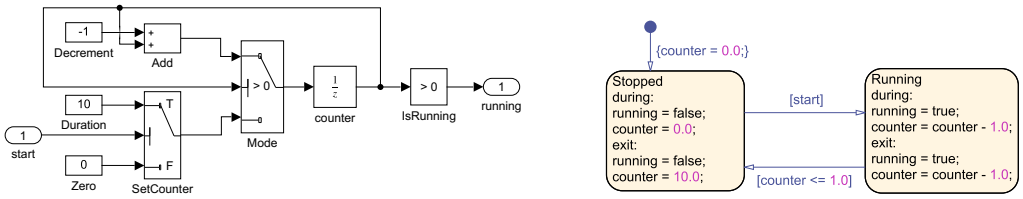
diminished value—maintainability of generated code is immaterial when it is overwritten automatically. Instead, new tools and metrics aimed at improving the quality of models offer much greater value, impacting the software in a way that is visible to the developer.

The adoption of MBD in the development of embedded control systems across industries has led to the wide use of Matlab/Simulink/Stateflow as a supporting environment. To support the development of high-quality software, guidelines such as the ones provided by the MathWorks Advisory Board (MAB) [The MathWorks 2020a] (formerly MAAB, the MathWorks Automotive Advisory Board) have been developed along with supporting tools such as Simulink Check [The MathWorks 2020b]. Simulink Check is a Simulink package available from MathWorks which evaluates compliance to various standards (including modelling standards, such as MAB guidelines, as well as standards that govern development of safety-critical software-intensive systems, such as ISO 26262 [ISO 2011]). Simulink Check also offers automated correction of certain offending design patterns. Such tools are often limited to supporting relatively simple rules, but modelling best practices can extend beyond such cases.

The modelling capabilities provided by Simulink block diagrams and Stateflow state charts complement each other by providing languages for functional and stateful system specifications. Due to their individual strengths, one modelling formalism may be preferable for specifying certain classes of behaviours. The MAB guidelines advise the use of Stateflow over Simulink for modelling stateful logic. This is because Simulink block diagrams that are used to model mode switching logic are often cumbersome and difficult to understand. In this case, Stateflow state charts should be used to implement the same logic

resulting in a structure which is easier to read, maintain, and verify.

For example, each model in Fig. 1.1 executes periodically to update its state and outputs. When the Simulink block diagram in Fig. 1.1a updates, each signal line is given a value and each block uses the values of the incoming signals to determine the values of the outgoing signals. When the state chart in Fig. 1.1b updates, it checks each condition on transitions leaving its current mode (i.e. state node). If a condition is satisfied, the state chart transitions to the associated target mode and executes the *exit* actions of the mode it is leaving, the actions on the transition it is taking, and the *entry* actions of the mode it is entering. If no transitions are valid, the state chart remains in its current mode and executes the *during* actions of that mode.



(a) Before: Simulink Block Diagram (b) After: Stateflow State Chart

Figure 1.1: Model of a Timer in Simulink and Stateflow.

The Simulink and Stateflow models shown in Fig. 1.1 are functionally equivalent. Both models capture a timer with one boolean input, *start*, and one boolean output, *running*. When *start* becomes true, the system starts counting down from ten to zero. While the system is counting down, *running* is true. Once the counter reaches zero, *running* is set to false and becomes true again if *start* is true. Although there are relatively few blocks in Fig. 1.1a, it is difficult to understand how this model achieves the behaviour while the state chart in Fig. 1.1b clearly captures the system's modes and the conditions triggering mode changes.

Practice shows that Simulink is often used to specify stateful logic even though Stateflow would be a more appropriate implementation language. This might occur during model evolution when modes of operation are added to previously mode-free block diagrams, and developers find it easier to modify the existing Simulink logic to accommodate the change than to reproduce the behaviour from scratch in a state chart. Other times, a developer's preference dictates the choice of modelling formalism. Manual refactoring from Simulink to Stateflow, although feasible, is a time consuming and error prone process which requires that the behaviour of complex Simulink models is completely understood. The main goal of this thesis is to develop a method and tool support for automated refactoring of Simulink block diagrams to Stateflow state charts for easier comprehension and maintenance.

1.2 Related Work

Several papers propose translating Simulink block diagrams to formal languages to enable their verification using existing tools (e.g., [Agrawal, Simon, and Karsai 2004]; [Dragomir, Preoteasa, and Tripakis 2015]; [Liebrenz, Herber, and Glesner 2018]; [Sfyrla et al. 2010]; [Tripakis et al. 2005]; [Zhan, Wang, and Zhao 2017]). Only a few, however, translate Simulink block diagrams to state transition diagrams. In [Minopoli and Frehse 2016], Simulink block diagrams are converted into an extended version of hybrid automata, with each block in a block diagram converted to a hybrid automaton, leading to an explosion in the number of states of the resulting model. In [Zhou and Kumar 2012], Simulink models are converted to finite state machines, but transitions between states represent the small execution steps of individual blocks updates,

not changes in the high level system modes. Both studies [Minopoli and Frehse 2016]; [Zhou and Kumar 2012], as well as [Manamcheri et al. 2011], do not aim to capture the high-level mode-switching logic of an entire block diagram. This is exactly what the approach in this thesis targets, with maintainability of the refactored model as a prime motivator.

The approach presented in this thesis shares many similarities with the approach presented in [Mealy 1955] for analysing sequential digital logic circuits. While [Mealy 1955] focuses on translating state transition diagrams to logic circuits, a brief description of the reverse translation (i.e. logic circuits to state transition diagrams) is given as well. While Simulink block diagrams and Stateflow state charts utilize higher levels of abstraction than simple digital logic, the core methodology used for digital logic is similar to the one presented here.

It is desirable to support model evolution and refactoring activities with tools automating complex tasks, and tool support for refactoring state charts has been proposed. In [Abadi and Feldman 2009], automated support for refactoring state charts is motivated, and some refactoring patterns are introduced. In [Harbird 2011], additional refactoring and refinement patterns for Contractual State Machines (i.e. annotated state charts) are presented in the form of transform-in-place model transformation rules.

Analysing the semantics of block diagrams as a whole requires a compositional approach. The general trend in behaviour modelling is to describe such compositional structures using monoidal categories. For example, monoidal categories have been used to describe interactions of quantum processes [Coecke and Kissinger 2017], labelled transition systems [Katis, Sabadini, and Walters 1997], and control systems [Baez and Erbele 2015]. The algebra of

(traced symmetric) monoidal categories is similar to the algebra used to describe block diagrams in [Dragomir, Preoteasa, and Tripakis 2015], but our approach uses a standard mathematical framework with a rich history and many known results. For example, the results of [Hofmann, Pierce, and Wagner 2011] can be reused to establish many useful facts about the structure used in this thesis to describe the behaviour of Simulink block diagrams. Specifically, their proof that Symmetric Lenses form a symmetric monoidal category can be directly adapted to show that our semantic model of Simulink block diagrams has the same categorical structure. Due to the similarities in the algebra for Simulink block diagrams, the strategy for building composite predicate transformers in [Dragomir, Preoteasa, and Tripakis 2015] is similar to the strategy used in this thesis for obtaining a tabular expression from a block diagram from component blocks and wires.

Simulink and Stateflow do not have a unified formal semantics, however there have been numerous semantic models proposed for assorted subsets of Simulink block diagrams and Stateflow state charts. For example, a small-step operational semantics of Stateflow state charts is presented in [Hamon and Rushby 2007], and a denotational semantics is presented in [Hamon 2005]. Similar examples exist for Simulink, such as the operational semantics for Simulink's simulation engine presented in [Bouissou and Chapoutot 2012].

Mode-switching logic uses decision logic to govern the behaviour of the whole system, i.e. not just one part of it. Therefore, a requirement for modelling mode-switching logic is that it must capture high level decision logic. Tabular expressions such as those presented in [Janicki and Wassyng 2005] provide a useful structure for modelling nested decision logic in systems with multiple inputs and multiple outputs. They have been given formal seman-

tics such as those established in [Jin and Parnas 2010], [Janicki and Wassyng 2005], and [Abraham 1997], where tabular expressions are given generalized evaluation rules.

Additionally, tabular expressions have been applied to model systems in Simulink and Stateflow. In [Singh et al. 2015] Stateflow state charts are translated to tabular expressions. The tables are given additional structure which establishes a one-to-one correspondence between Stateflow state charts and tabular expressions with the given format. In [Eles and Lawford 2011] a tabular expression toolbox is introduced as a means of specifying the behaviour of a block in Simulink, along with support for verifying certain conditions of well-formed tables.

Composition of tabular expressions has been studied in [Mohrenschildt 2000]. The composition of tabular expressions plays a key role in the translation strategy presented in this thesis, as it allows for the compositional translation of Simulink block diagrams to tabular expressions: blocks are individually translated to tabular expressions, and those tabular expressions are subsequently composed.

1.3 Contributions

This thesis presents an approach to translating implicit mode-switching logic modelled by Simulink block diagrams into behaviourally equivalent Stateflow state charts. The translation aims to improve the maintainability of Simulink models by modelling mode-switching logic more directly, making the models easier to comprehend and modify. Through collaboration with a large automotive OEM, block diagrams implementing mode-switching logic were found

to be difficult to maintain, and ensuring behavioural equivalence is challenging when manually refactoring models. The translation approach presented in this thesis was developed to provide tool support for such scenarios.

The translation approach utilizes tabular expressions [Parnas 1992] as an intermediate step between block diagrams and state charts. The approach converts individual blocks into tabular expressions to expose their latent state variables and decision logic. The data flow between blocks is then used to combine tables into a single, larger table describing the entire block diagram. Then, the elements of state charts (states, transitions) are extracted from the combined table. Behavioural equivalence is established by giving semantics to block diagrams, state charts, and the intermediate tables as Mealy machines.

The main contributions of this thesis are:

- A method for identifying implicit mode-switching logic in Simulink block diagrams. To the best of the author's knowledge, no other method exists that identifies mode-switching (stateful) logic in Simulink block diagrams. The method can be used in a model-based software development process to find cumbersome fragments of large industrial Simulink block diagrams which would be more suitably implemented as a Stateflow state chart.

- A refactoring approach which converts Simulink block diagrams to Stateflow state charts using tabular expressions as an intermediate representation. Once cumbersome fragments of block diagrams are identified, they may be automatically translated to more understandable Stateflow state charts. To the best the author's knowledge, this refactoring approach, together with the identification method, represents a unique approach

to improving quality of Simulink designs by translating stateful logic implemented in Simulink to Stateflow.

- A categorical framework for composing Mealy machines by combining their update functions which is used as the semantic basis of the translation. Behavioural equivalence is a key requirement of automated refactoring to ensure the developed models function correctly.

- A prototype tool implementing the translation from Simulink to Stateflow [McSCert 2019]. Proper automation of the method is crucial for its adoption in industry.

The translation strategy proposed by this thesis was presented at FASE 2019, with the accompanying paper [Wynn-Williams et al. 2019] providing an overview of the translation steps presented in chapter 5 and chapter 6. This thesis elaborates on the translation strategy, including a more detailed description and proofs for behavioural equivalence. Additionally, this is the first account of the identification strategy presented in chapter 4.

1.4 Outline

This thesis first provides background information in chapter 2, covering an introduction to the behavioural models and theoretical background used in this work. In chapter 3, the link between data flow (block diagram) and control flow (state chart) languages is established, giving a high level description of how this thesis extends previous work. Subsequently, chapter 4, chapter 5, and chapter 6 provide more detail on the steps involved in the refactoring strategy, and discuss how each step has been automated. Finally, chapter 7 concludes

with a discussion of the strategy's effectiveness, and possible directions for future work to improve the strategy.

Chapter 2

Preliminaries

2.1 Notation

This work uses the following notation:

- $f : X \to Y$ – a function f with domain X and codomain Y
- Y^X – the set of all functions with domain X and codomain Y
- $\{x \mapsto e \mid x \in X\}$ – a function with domain X, $x \in X$ is omitted when it is given by context
- $\{(x_1, \ldots, x_n) \mapsto (e_1, \ldots, e_m)\}$ – a function whose inputs/outputs are tuples
- $(c_1 : e_1, \ldots c_n : e_n)$ – a tuple with named components
- $\{(x_1 : v_1, \ldots, x_n : v_n) \mapsto (y_1 : e_1, \ldots, y_m : e_m) \mid x_i \in X_i\}$ – a function with multiple inputs and outputs where the inputs and outputs are named. If the input parameter names are used directly in the output expressions $x_i : x_i$ may be simply written as x_i.

- $\{e_l \leftrightarrow e_r \mid C\}$ – a relation which links the values of e_l and e_r if condition C is satisfied

2.2 Simulink and Stateflow Models

Suppose we wanted to model a space heater. Specifically, consider the thermostat used to turn on and off the fan and heat coil to warm a room to a desired temperature. This system would need inputs to determine the desired temperature and the actual ambient temperature of the room. Additionally it would produce as outputs the electrical signals used to power the fan and the heater coil. We will abstract away the sensors and signal conditioning electronics and say that our model of the thermostat has two real-valued input variables: `setpoint` and `temp`; and two Boolean-valued output variables: `fanOn` and `coilOn`.

2.2.1 Simulink Block Diagrams

In Simulink such a system would appear as a box such as the one illustrated in Fig. 2.1. Each input variable is illustrated as a *port* on the left hand side of the block with an arrow indicating that data enters through it. Similarly, output variables are illustrated as ports on the right hand side of the block with arrows indicating that data exits through them. For this reason, the inputs and outputs to systems are often referred to as input or output ports.

In our thermostat example, the simplest way to achieve the desired temperature is with a so-called On/Off controller: when the temperature is below the setpoint, turn on the heat coil and fan; and when the temperature is above the setpoint turn them off. To implement this functionality in Simulink, we could

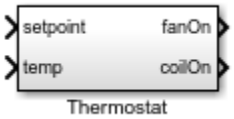


Figure 2.1: A Simulink subsystem block

look inside the Thermostat system and find the blocks pictured in Fig. 2.2. Each input/output is pictured with a block (called an inport/outport block) which produces/receives the values passed into and out of the ports of the subsystem block. Values are passed between blocks via wires or *signal lines*. For example, the signal lines coming from the inport blocks setpoint and temp are routed to the block labelled IsTooCold. The IsTooCold block will compare the values of *setpoint* and *temp*, and if the setpoint is higher than the current temperature then the block will produce the value true, and otherwise it will produce the value *false*. This value then gets routed to both *coilOn* and *fanOn* outport blocks via a branching signal line. Branching signal lines route information from a single source to multiple destinations, whatever value is produced by IsTooCold will be received by both outport blocks.

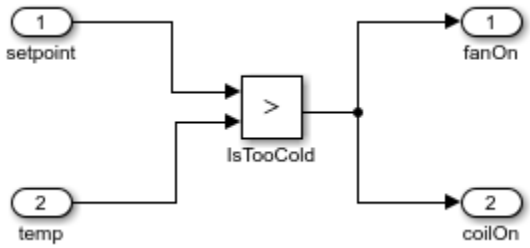


Figure 2.2: A Simulink block diagram

This block diagram can be used to generate embedded software and deployed on a microcontroller in a space heater to keep a room at the desired temperature. To achieve the desired functionality, the space heater would peri-

odically check the setpoint, measure the temperature of the room, and execute the function described by the block diagram to turn the fan and heating coil on or off. For example, Fig. 2.3 shows the inputs and outputs to the block diagram if the space heater were to check the temperature and setpoint once every 60 seconds. Initially the temperature is above the setpoint, so there is no need for the heater to turn on, thus the system in Fig. 2.2 keeps both the fan and the heating coil off. After 60 seconds however the setpoint has been increased to 23°C, and the IsTooCold block produces the value true which is routed to both *fanOn* and *coilOn*, so the space heater starts heating the room. The room will continue to heat up for the next few minutes until it reaches the desired temperature. At this point the IsTooCold block will once again produce the value false which is routed to the outputs, turning the fan and heating coil off.

Sample	0	1	2	3	4	5	...
Time (s)	0	60	120	180	240	300	...
setpoint (°C)	20	23	23	23	23	23	...
temp (°C)	21	21	21.5	22	22.5	23	...
fanOn	F	T	T	T	T	F	...
coilOn	F	T	T	T	T	F	...

Figure 2.3: Example input and output sequences for the block diagram in Fig. 2.2

This simple example introduces three types of blocks: inport blocks, outport blocks, and logical relation blocks. Simulink is equipped with a large library of block types for different purposes, but the examples in this thesis will draw from a selected subset of commonly used block types.

Systems may also be used as a packaged module that appears as a single block in another block diagram. This is achieved using subsystem blocks, such as the block labelled "Beeper" in Fig. 2.4.

One type of block of particular interest in this thesis is the unit delay. Unit delay blocks store the value of an input signal and produce the stored value one time step later. This allows Simulink block diagrams to act according to not only the current inputs, but also previous inputs. For example, we may want to introduce a safety feature to the space heater to produce a beeping noise whenever the heating coil is turned on. To achieve this functionality, we can design a beeper module which receives a signal indicating whether the heating coil is on. We could describe this module as a system with one input ($coilOn \in \mathbb{B}^{\mathbb{N}}$) and one output ($beep \in \mathbb{B}^{\mathbb{N}}$), as illustrated by the subsystem block in Fig. 2.4.

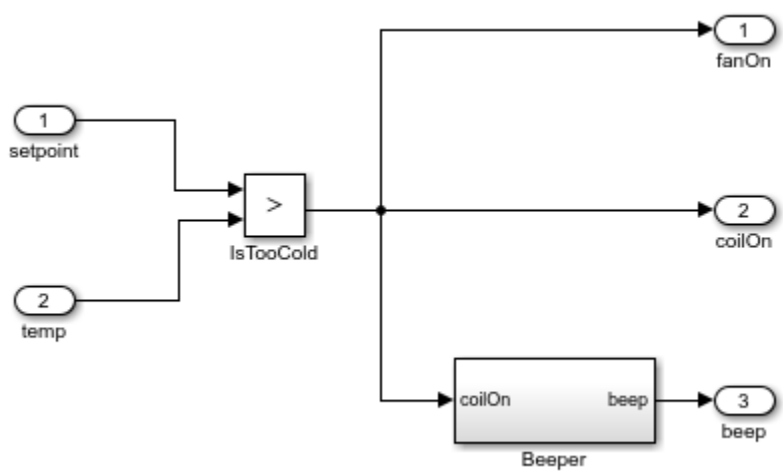


Figure 2.4: Adding a subsystem block to the block diagram in Fig. 2.2

To implement this subsystem, we need to know not only whether the heating coil is on now, but also whether it was on or off in the previous time step. If the coil is on now, and it was off before, then the beeper module should issue a signal telling some speaker to issue a beep. In order for the subsystem to know whether the coil was off before, it can delay the *coilOn* signal so its

value is available in the next time step, as is done with the *coilWasOn* delay block in Fig. 2.5. By negating *coilWasOn* and taking the conjunction of the result with *coilOn*, the value of *beep* can be calculated.

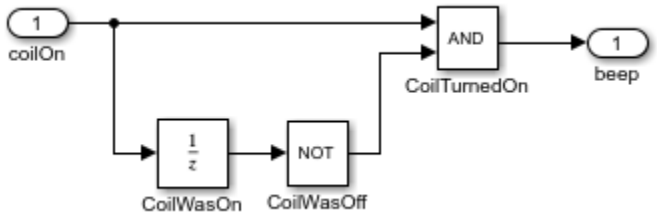


Figure 2.5: The internals of the Beeper subsystem

Once again this block diagram can be used to generate embedded software to deploy in a space heater. If the Beeper module executes every second, then an example of the inputs and outputs to the beeper module are shown in Fig. 2.6. Additionally, the value produced by the unit delay block included in the table to illustrate how it changes over time. Note that in the first time step *coilWasOn* produces the value false, even though there is no previous value of *coilOn*. To handle the exceptional case of the value output by unit delay blocks in the first time step, there is a configurable *Initial Value* parameter. In the first time step the unit delay produces the configured initial value, and from that point on it produces the same value it consumed (from *coilOn*) in the previous time step. In each time step where *coilOn* is *true*, and *coilWasOn* is *false*, the logical operator blocks will produce a value of *true* for the output *beep*. This occurs in our example run from Fig. 2.6 after 1 second, and again after 5 seconds.

Sample	0	1	2	3	4	5	6	7	...
Time (s)	0	1	2	3	4	5	6	7	...
coilWasOn	F	F	T	T	F	F	T	T	...
coilOn	F	T	T	F	F	T	T	T	...
beep	F	T	F	F	F	T	F	F	...

Figure 2.6: Example input and output sequences for the block diagram in Fig. 2.5

2.2.2 Stateflow State Charts

One of the strengths of the Simulink environment is that it has block diagrams for describing data flow between components, but those components can be implemented in more than one way. For example, the component shown in Fig. 2.2 was implemented using a block diagram subsystem, where elementary blocks were combined to achieve the desired behaviour. While this approach can be useful for modelling very simple decision procedures, complex decision procedures which depend on the history of inputs quickly becomes cumbersome to represent with block diagrams. This is where Stateflow state charts can be employed to help manage more complex logic.

For example, suppose the thermostat from Fig. 2.2 were used to produce a prototype space heater. Testing might uncover an unfortunate side effect of its control strategy, as illustrated by Fig. 2.7, which continues the run from Fig. 2.3. When the room reaches a temperature which is close to the setpoint, the heater will rapidly turn on and off as the temperature oscillates around the setpoint. To reduce the frequency of these oscillations it may be desirable to introduce some hysteresis so that when the heater turns on, it remains on until the room reaches a temperature slightly higher than the setpoint before turning off. Likewise, when the heater turns off, it remains off until the room cools down slightly below the setpoint before turning off again. While it is possible

to implement this logic by combining elementary blocks in a block diagram, it is more natural to represent this logic using a Stateflow state chart.

Sample	...	6	7	8	9	10	...
Time (s)	...	360	420	480	540	600	...
setpoint (°C)	...	23	23	23	23	23	...
temp (°C)	...	22.75	23.25	23	22.75	23.25	...
fanOn	...	T	F	F	T	F	...
coilOn	...	T	F	F	T	F	...

Figure 2.7: Extending the input and output sequences from Fig. 2.3

Stateflow state charts can be placed in Simulink block diagrams in the same manner as any other block, but they look much different when you look inside. For example, similarly to how Fig. 2.1 shows a subsystem block containing the implementation of the on/off thermostat, Fig. 2.8 shows a Stateflow block which contains a state chart implementing the thermostat with hysteresis.

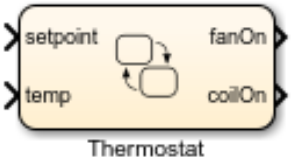


Figure 2.8: A Stateflow chart block

The state chart which defines the behaviour of the Stateflow chart block is shown in Fig. 2.9. Each input port of the Stateflow chart block in Fig. 2.8 is associated with an input variable which may be referenced in the chart. Similarly, each output port of the chart block is associated with an output variable whose value is assigned in the chart. The chart itself contains two *modes*[1]: *Off* and *Heat*, and three *transitions*: one from *Off* to *Heat*, one from

[1]While the state nodes of state charts are typically called "states", this thesis calls them "modes" to distinguish them from the states of Mealy machines which incorporate both modes and state variable values.

Heat to *Off*, and a *default transition* which leads to *Off*.

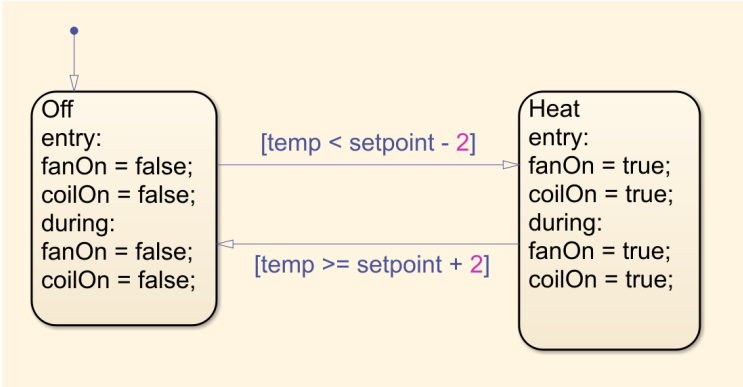


Figure 2.9: A Stateflow state chart

Similarly to the block diagrams, state charts are executed in time steps. At every time step, the conditions guarding each transition leaving the current mode are evaluated. If any of the conditions are satisfied, then the state chart will follow that transition and enter the targeted mode. When a transition is taken, the state chart executes any *exit actions* associated with the mode it is leaving, then executes any *transition actions* associated with the transition it is taking, then executes any *entry actions* associated with the mode it is entering. If none of the transitions leaving the current mode are valid, then the state chart will remain in the current mode and execute the during action. Once the chart updates, either by transitioning to a new mode, or staying in the current mode, the state chart will rest in the current mode until the next time step.

The default transition controls the initialization of the state chart, however there is a configuration (Execute at Initialization) which determines whether the default transition is taken *during* the first time step, or *before* the time step. For example, the state chart in Fig. 2.9 will either unconditionally enter the *Off* mode during the first time step, or it will start in the off mode and its

behaviour will depend on the input values. The state charts discussed in this thesis will be designed assuming that the default transition will be taken *before* the first time step (i.e. they are configured to execute upon initialization).

Using the same inputs from Fig. 2.3 and Fig. 2.7, Fig. 2.10 shows how the state chart from Fig. 2.9 would react. It starts in the *Off* mode, and in the first time step it remains in *Off*, executing the during action which sets *fanOn* and *coilOn* to *false*. Then in the second time step it reacts to the new setpoint by transitioning to the *Heat* mode and executes the entry action, setting *fanOn* and *coilOn* to *true*. It then stays in *Heat* setting *fanOn* and *coilOn* to *true* until the temperature of the room reaches 25°C at which point it transitions to *Off* and sets *fanOn* and *coilOn* to *false*.

Sample	0	1	2	3	4	5	6	7	8	9	10	...
Time (s)	0	60	120	180	240	300	360	420	480	540	600	...
Previous Mode	Off	Off	Heat	Heat	Heat	Heat	Heat	Heat	Heat	Heat	Off	...
setpoint (°C)	20	23	23	23	23	23	23	23	23	23	23	...
temp (°C)	21	21	21.5	22	22.5	23	23.5	24	24.5	25	24.75	...
fanOn	F	T	T	T	T	F	T	T	T	F	F	...
coilOn	F	T	T	T	T	F	T	T	T	F	F	...

Figure 2.10: Example output sequence using the input sequences from Fig. 2.3 and Fig. 2.7 with the state chart from Fig. 2.9

In addition to modes, state charts can be equipped with state variables which can store more detailed state information. For example we may want to introduce a timer variable which will turn the heater off after some fixed amount of time so that it does not overheat.

Stateflow state charts have a rich feature set which we do not discuss here as we focus on only a subset of the language. Notably absent from this subset are junction nodes, hierarchy of states, parallel states, and events.

2.3 Modelling Behaviours

The goal of refactoring is to convert block diagrams into state charts without changing their behaviours. This section defines what a behaviour is, how behaviours are modelled, and how to prove that models preserve behaviour.

2.3.1 Behaviours as a subset of possible observations

Willems describes behaviours of systems in terms of what outcomes can be observed [Willems 1989]. The set of all describable observations is denoted $\mathbb{V}$ and is called the *universum*, it contains both possible and impossible outcomes. The goal of modelling a system is to describe a subset $\mathcal{B}$ of observations which are possible.

Willems' model of behaviour can be refined by enriching the universum with additional structure, or the model can be refined with additional structure for specifying the subset $\mathcal{B}$.

To model complex systems, it is useful to project observations into components, and describe the behaviours in terms of those projections. For example, a universum $\mathbb{V}$ can be given the structure of a product: $\mathbb{V} \equiv V_1 \times V_2 \times \cdots \times V_n$. In particular when modelling embedded software, it is useful to partition observations into two components: an independent component Σ; and a dependant component Λ. This means that we may model a behaviour using a function f; in this case the behaviour is given as $\mathcal{B} \equiv \{(\sigma, \lambda) \in \Sigma \times \Lambda \,|\, \lambda = f(\sigma)\}$. Of course, these projections may also be given the structure of a product: $\Sigma \equiv \Sigma_1 \times \Sigma_2 \times \cdots \times \Sigma_m$ and $\Lambda \equiv \Lambda_1 \times \Lambda_2 \times \cdots \times \Lambda_n$ where Σ_i and Λ_j are the types of individual input and output signals. This is useful for modelling systems with multiple inputs and multiple outputs. Behaviours may then be

modelled as functions of type $\Sigma_1 \times \Sigma_2 \times \cdots \times \Sigma_m \to \Lambda_1 \times \Lambda_2 \times \cdots \times \Lambda_n$.

For example, the behaviour of the simple thermostat in Fig. 2.2 can be described as a function which takes the pair $(\mathit{setpoint}, \mathit{temp}) \in \mathbb{R} \times \mathbb{R}$ to the pair $(\mathit{fanOn}, \mathit{coilOn}) \in \mathbb{B} \times \mathbb{B}$. This function can be given as:

$$\left\{ (\mathit{setpoint}, \mathit{temp}) \mapsto \left\{ \begin{array}{ll} (\mathit{fanOn} : \mathit{true}, \mathit{coilOn} : \mathit{true}) & \mathit{setpoint} > \mathit{temp} \\ (\mathit{fanOn} : \mathit{false}, \mathit{coilOn} : \mathit{false}) & \mathit{setpoint} \leq \mathit{temp} \end{array} \right. \right\} \tag{2.1}$$

This refinement allows for the description of functional systems with multiple inputs and multiple outputs.

Willems' description of a system can be also refined by considering a class of system where observations form *trajectories* over time. The universum is equipped with a set $\mathbb{T}$ of time moments at which the system evolves, and a set $\mathbb{W}$ of instantaneous observations. If the system is observed at each time moment, the universum is given as the set of all mappings $\mathbb{W}^{\mathbb{T}}$ from $\mathbb{T}$ to $\mathbb{W}$. Common choices for $\mathbb{T}$ are the set $\mathbb{R}^+$ of positive real numbers for continuous systems, and the set $\mathbb{N}$ of natural numbers for discrete systems. This refinement is useful for modelling dynamic systems, where the behaviour of the system somehow evolves from one time moment to the next.

For example, if the space heater is turned off and the heating coil was still warm from use, then the temperature of the heating coil would slowly decrease, given by a function of time of the following form:

$$T(t) = (T_0 - T_a)e^{-rt} + T_0 \tag{2.2}$$

where T_0 is the initial temperature of the coil, T_a is the ambient temperature,

and r is a time constant which is determined, for example, by the material and shape of the heating coil.

Thus, the heating coil would be described as a dynamic system where $\mathbb{W} = \mathbb{R}$ is the temperature (in Kelvin) of the heating coil, and $\mathbb{T} = \mathbb{R}^+$ is the time (in seconds) that the coil has been observed for. Assuming some value for r (say, $r = 0.05$), the behaviour of this system is given by the family of functions:

$$\mathcal{B} = \{\{t \mapsto (T_0 - T_a)e^{-0.05t} + T_0\}\}_{T_0 \in \mathbb{R}^+, T_a \in \mathbb{R}^+} \tag{2.3}$$

This refinement for describing dynamic systems can be combined with the refinement for decomposing the universum into components to give a description in terms of components which evolve over time. By considering each component as the trajectory of an individual signal's value, we can describe the behaviour of functional discrete dynamic systems. The universum in this case is given by $\mathbb{V} \equiv \Sigma^{\mathbb{N}} \times \Lambda^{\mathbb{N}}$, and behaviours are given by functions of type $\Sigma^{\mathbb{N}} \to \Lambda^{\mathbb{N}}$.

For example, if we wanted the space heater to beep whenever the heating coil turned on, we could describe a system with one input ($\mathit{coilOn} \in \mathbb{B}^{\mathbb{N}}$) and one output ($\mathit{beep} \in \mathbb{B}^{\mathbb{N}}$). The behaviour of the beeper module would be described by the function:

$$f = \left\{ \mathit{coilOn} \mapsto \left\{ n \mapsto \left\{ \begin{array}{ll} (\mathit{coilOn}(n) \wedge \mathit{coilOn}(n-1) & n > 0 \\ (\mathit{false} & n = 0 \end{array} \right\} \right\} \right. \tag{2.4}$$

and would be given by the subset of the universum:

$$\mathcal{B} = \{(\mathit{coilOn}, f(\mathit{coilOn}))\}_{\mathit{coilOn} \in \mathbb{B}^{\mathbb{N}}} \tag{2.5}$$

2.3.2 Mealy Machines

While the class of deterministic discrete dynamic systems is of particular interest for this work, working with the higher-order definition above is often cumbersome. Software systems do not process the entire stream of inputs at once; they process inputs incrementally, reacting to inputs at each time moment based the current input. In order for the system to access the values of previous inputs, it must store them in some manner. Typically the system does not need to know the entire history of inputs, and instead stores only the information it needs about previous inputs. This stored information is the state tracked by the system. Such systems can be described using Mealy Machines:

Definition 1. *A* Mealy Machine *m is a tuple $(S, s_0, \Sigma, \Lambda, ud)$, where*

- *S is a set of states, called the* state space
- *$s_0 \in S$ is the initial state;*
- *Σ is a set of input values, called the* input alphabet
- *Λ is a set of output values, called the* output alphabet
- *$ud : \Sigma \times S \to \Lambda \times S$ is a function which computes the current output and next state from the current input and previous state, called the* update function.

Mealy machines describe functional discrete dynamic systems by modelling

behaviours as a subset of $\Sigma^{\mathbb{N}} \times \Lambda^{\mathbb{N}}$ in the following way:

$$\mathcal{B} \equiv \left\{ (\sigma, \lambda) \left| \begin{array}{l} \exists s \in S^{\mathbb{N}} \bullet \\ (\forall n \in \mathbb{N} \bullet (\lambda(n), s(n+1)) = ud(\sigma(n), s(n))) \\ \qquad \wedge (s(0) = s_0) \end{array} \right. \right\}_{\sigma \in \Sigma^{\mathbb{N}}} \tag{2.6}$$

The behaviour of the beep module from Section 2.2.1 can be described as a Mealy machine with input alphabet $\Sigma = \mathbb{B}$, output alphabet $\Lambda = \mathbb{B}$, and state space $S = \{On, Off\}$. The initial state of the Mealy machine is $s_0 = Off$ and the update function is given by:

$$\left\{ (coilOn, state) \mapsto \left\{ \begin{array}{ll} (beep : false, state' : Off) & \neg coilOn \\ (beep : true, state' : On) & coilOn \wedge (state = Off) \\ (beep : false, state' : On) & coilOn \wedge (state = On) \end{array} \right. \right\} \tag{2.7}$$

In many contexts, Mealy machines are considered to be as expressive as *Moore machines*, a model of behaviour similar to Mealy machines where the output is expressed solely in terms of the state. The model of behaviour described by Mealy machines is given in Equation 2.6 and associates input and output values with moments in discrete time. Mealy machines transition from one point in the state space to another when an input is received, and simultaneously produce the associated output value. Like Mealy machines, Moore machines can also be used to map sequences of inputs to sequences of outputs. However, to compare the behaviours of Mealy machines and Moore machines, it is important to consider what output is produced by a Moore machine when an input is received at time step n. Since the Moore machine

is transitioning from the previous state to the next state at time step n, the output could be determined by either the source state of the transition or the target state. If the output is determined by the source state, then not all behaviours representable as a Mealy machine can be represented as a Moore machine, since the Moore machine cannot produce an output reacting to an input received in the same time step. Alternatively, if the output is determined by the target state, then Moore machines can express any behaviour that can be expressed as a Mealy machine [Babcsanyi 2000]. In general however Moore machines require a state space larger than an equivalent Mealy machine, since the state must not only track values required for the purpose of tracking history, but also values responsible for tracking what to output in the current time step. To avoid the introduction of superfluous states, Mealy machines are chosen over Moore machines for modelling behaviours.

2.3.3 State Transition Graphs & Tables

Mealy machines are often illustrated in the form of state transition graphs. State transition graphs represent the flow of time by showing how the system may evolve as it transitions from one state to the next.

Definition 2. *A* State Transition Graph (STG) *is a directed graph* $G(S, T, src, tar)$*, where*

- S *is a set of states (nodes)*
- T *is a set of transitions (edges)*
- $src, tar : T \to S$ *indicate the starting and ending state of the transition*

with additional structure for labelling transitions $STG(G, s_0, \Sigma, \Lambda, \sigma, \lambda)$*, where*

- $s_0 \in S$ *is the initial state*
- Σ *is a set of input symbols*
- Λ *is a set of output symbols*
- $\sigma : T \to \Sigma$ *indicates the input which triggers the transition*
- $\lambda : T \to \Lambda$ *indicates the output produced when the transition is triggered*

An example of a state transition graph is illustrated in Fig. 2.11. It has four states: $\{A, B, C, D\}$; two input symbols: $\{\alpha, \beta\}$; two output symbols: $\{\gamma, \omega\}$; and eight transitions. The arrow which originates from no state indicates that $s_0 = A$.

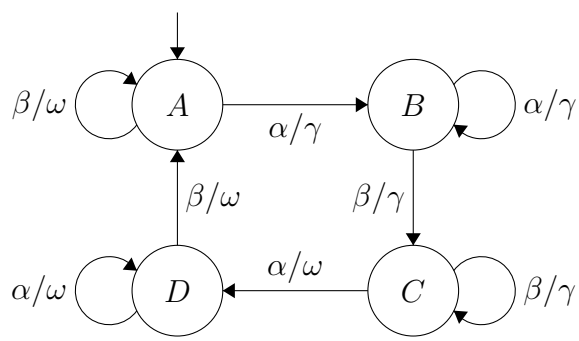


Figure 2.11: State Transition Graph

State transition graphs encode Mealy machines where each transition represents one maplet of the update function. The source state paired with the trigger symbol map to the target state and the output symbol in the transition/output functions (respectively).

$$ud = \{(\sigma(t), src(t)) \mapsto (\lambda(t), tar(t))\}_{t \in T} \tag{2.8}$$

The remaining elements of the Mealy machine are obtained trivially from

the definition of the STG (input/output/state symbols and initial state are defined directly).

Mealy machines can be alternatively represented in a similar manner using *State Transition Tables (STTs)*. These tables contain the same information as the graph, but instead of representing transitions as edges of a graph, each transition is represented as a row in a table. This table has columns for source, and target states, as well as for triggers and outputs. The Mealy machine from Fig. 2.11 is represented as a table in Fig. 2.12.

	$M[n]$	$in[n]$	$out[n]$	$M[n+1]$
	A	β	ω	A
	A	α	γ	B
	B	β	γ	C
	B	α	γ	B
	C	β	γ	C
	C	α	ω	D
	D	β	ω	A
	D	α	ω	D
$n = 0$	A			

Figure 2.12: State Transition Table

While it is more common to see functions represented in the form of a table, where inputs are organized in rows next to their precomputed outputs, the STTs and STGs contain the same data. Each has syntax for representing a function as a set where each element pairs the input with the output, and so they may be called *pairwise* representations of a function.

While the syntax of STGs is fundamental to state charts, they require additional structure which is covered in Section 2.3.6.

2.3.4 Combining Mealy Machines

When studying models of behaviour, it is typical to examine how instances of these models can be combined to produce complex systems comprised of simpler components. Mealy machines can be combined in two ways to produce complex systems: sequential (cascade) composition, where the output of one Mealy machine is fed into the input of another; and parallel (side-by-side) composition. These composition operations are well understood, and the following definitions are similar to the ones given in [Lee and Varaiya 2011].

Definition 3. *Given two Mealy machines* $m_1 = (S_1, s_0^1, \Sigma, \Theta, ud_1)$ *and* $m_2 = (S_2, s_0^2, \Theta, \Lambda, ud_2)$*, where the output alphabet of* m_1 *is the same as the input alphabet of* m_2*, the sequential composite Mealy machine* m' *is given by* $(S_1 \times S_2, (s_0^1, s_0^2), \Sigma, \Lambda, \{(\sigma, (s_1, s_2)) \mapsto (\lambda, (s_1', s_2'))\})$ *where* $(\theta, s_1') = ud_1(\sigma, s_1)$ *and* $(\lambda, s_2') = ud_2(\theta, s_2)$*.*

Definition 4. *Given two Mealy machines* $m_1 = (S_1, s_0^1, \Sigma_1, \Lambda_1, ud_1)$ *and* $m_2 = (S_2, s_0^2, \Sigma_2, \Lambda_2, ud_2)$*, the parallel composite Mealy machine* m' *is given by* $(S_1 \times S_2, (s_0^1, s_0^2), \Sigma_1 \times \Sigma_2, \Lambda_1 \times \Lambda_2, \{((\sigma_1, \sigma_2), (s_1, s_2)) \mapsto ((\lambda_1, \lambda_2), (s_1', s_2'))\})$ *where* $(\lambda_1, s_1') = ud_1(\sigma_1, s_1)$ *and* $(\lambda_2, s_2') = ud_2(\sigma_2, s_2)$*.*

2.3.5 Behavioural Equivalence

One of the primary benefits of Mealy machines for showing behavioural equivalence is that it is possible to show that two Mealy machines have the same behaviours by examining their update functions. This is done using the notion of a *simulation relation*. A simulation relation between two Mealy machines sharing an input and output alphabet $m_1 = (S_1, s_0^1, \Sigma, \Lambda, ud_1)$ and

$m_2 = (S_2, s_0^2, \Sigma, \Lambda, ud_2)$ is a relation $\mathcal{R} : S1 \leftrightarrow S2$ satisfying:

$$s_0^1 \mathcal{R} s_0^2 \tag{2.9}$$

and

$$\forall (s_1, s_2) \in S_1 \times S_2 \bullet s_1 \mathcal{R} s_2 \Rightarrow \forall \sigma \in \Sigma \bullet (\lambda_1 = \lambda_2) \wedge (s_1' \mathcal{R} s_2') \tag{2.10}$$

where $(\lambda_1, s_1') = ud_1(\sigma, s_1)$ and $(\lambda_2, s_2') = ud_2(\sigma, s_2)$.

Simulation relations are typically studied in the context of non-deterministic and non-total automata, where instead of an update function, the automata transition according to an update relation. For this reason, simulation relations are typically formulated in terms of the set of possible results of two automata updating, which are compared by the relationship of subsets, rather than equality. In this context, the simulation relationship is directional. One automaton simulates another if it exhibits all behaviours of the automata it simulates. However in the context of deterministic total automata (e.g. Mealy machines), the number of behaviours is equal to the number of possible inputs to the automata. For this reason, any two automata over the same input alphabet have the same number of behaviours. If one is a subset of the other, then the two sets of behaviours are necessarily equal. This breaks the typical asymmetry of simulation relations: if a Mealy machine simulates another, then the second also simulates the first. This is typically referred to as a bisimilar pair of automata.

2.3.6 Extended State Transition Graphs

Representing a Mealy machine using an STG or simple STT is useful in many scenarios where the set of possible inputs, outputs, and states are small. However in practical applications, it is often the case that the inputs and outputs of a system are given by multiple variables, and those variables have values with some known structure (e.g. they may have integer or floating point values). In these cases, the structure of the input/output variables can be leveraged to simplify the representation of the Mealy machine. Extended state transition graphs extend the notion of state transition graphs by replacing the labels on the transitions: single input triggers are replaced by *guards* which use symbolic representations of predicates to specify a subset of input variable values which trigger the transition, and single output values are replaced by actions which modify the values of the output variables. In addition, the nodes of the transition graph only represent one component of the Mealy machine's state. The remainder of the Mealy machine's state is encoded using state variables. State variables can be referenced in guards, and assigned to in actions to generalize over equivalence classes of states of the Mealy machine. For example, an extended state transition graph may have a looping transition where a state variable is incremented. While a traditional state transition graph would have a different node for each value of that state variable, with transitions leading from one node to the node corresponding to the next value, all those states and transitions get glued together in an extended state transition table.

While state charts have mechanisms for much more than what can be simply described using extended state transition graphs, this thesis only uses features of state machines that fit into this simple model, barring a few syntac-

tic features of state charts that have an apparent analogue in extended state transition graphs, namely entry, during, and exit actions which are equivalent to actions on incoming, self-looping, or outgoing transitions (respectively).

2.4 Tabular Expressions

Both block diagrams and state charts can specify decision logic, but in rather distinct ways. We unify the presentation of decision logic in the two formalisms using two similar forms of tabular expressions: *horizontal condition tables* (HCTs) as presented in [Janicki and Wassyng 2005]; and *state transition tables* (STTs), which specialize HCTs to describe state charts similarly to the ones presented in [Singh et al. 2015].

		running	*counter′*
start	$counter > 0$	*true*	$counter - 1$
	$counter \leq 0$	*false*	10
$\neg start$	$counter > 0$	*true*	$counter - 1$
	$counter \leq 0$	*false*	0

(a) Horizontal Condition Table

Source	**Condition**	*running*	*counter′*	**Target**
Running	$counter - 1 > 0$	*true*	$counter - 1$	*Running*
	$counter - 1 \leq 0$	*true*	$counter - 1$	*Stopped*
Stopped	*start*	*false*	10	*Running*
	$\neg start$	*false*	0	*Stopped*

(b) State Transition Table

Figure 2.13: Tabular Expressions

2.4.1 Horizontal Condition Tables

An HCT is represented in Fig. 2.13a. It is a tabular representation of the update function of a Mealy machine which models the block diagram from Fig. 1.1a. Given the variable values $start = true$ and $counter = 0$, the table can be evaluated from left to right in the following way. Since the first condition *start* of the first column is satisfied, and the sub-condition $counter \leq 0$ in the second row of the second column is satisfied, we use the second row to

determine that *running* is given the value of *false*, and $counter'$ is given a value of 10.

2.4.2 Extended State Transition Tables

The second tabular representation, extended state transition tables, are also used to represent the update function of Mealy machines. Their special format closely matches the state charts (i.e. extended state transition graphs) they model. For example, the STT in Fig. 2.13b represents the state chart in Fig. 1.1b. Each mode is listed in the first column, and the condition of each transition is listed in the second column, adjacent to the mode it leaves. The columns after the double bars describe how each output/state variable is updated by the actions of the associated transition. The final column of each row indicates which mode the associated transition leads to.

Tabular expressions were given a precise semantics in [Jin and Parnas 2010]. The structure of tables can be rearranged without changing the function they describe, e.g., conditions can be reordered as in [Bialy et al. 2015]; conditions can be combined with sub-conditions (via conjunction) to flatten the hierarchy of conditions; and normal expressions in the table can be simplified by assuming the conditions to their left hold.

2.5 Category Theory

The key idea of block diagrams is to combine simple, predefined blocks to describe a behaviour. The language of *monoidal categories* explains how to break down the complex data flow of block diagrams and describe it in terms of simpler data flow [Coecke and Kissinger 2017] (i.e. cascading blocks in

sequence, placing blocks in parallel, and feeding outputs of blocks back to their inputs).

2.5.1 Categories

Definition 5. *A* Category $\boldsymbol{C}$ *consists of:*

- *A collection of* objects $|\boldsymbol{C}|$
- *For each pair of objects* $X, Y \in |\boldsymbol{C}|$ *a collection of* morphisms $\boldsymbol{C}(X, Y)$ *with* domain X *and* codomain Y.
- *A composition operation* $(-)\,;(-)$*, which takes any pair of morphisms* $f : X \to Y$ *and* $g : Y \to Z$ *where the codomain of the first morphism is the same as the domain of the second morphism and produces a third morphism* $f; g : X \to Z$*: their composite. The composition operation must satisfy the following axioms:*
  - *Associativity: For any morphisms* $f : W \to X$*,* $g : X \to Y$*,* $h : Y \to Z$*,*

$$(f; g); h = f; (g; h)$$

- *A family of identity functions* $\{\mathrm{id}_X\}_{X \in |\boldsymbol{C}|}$ *satisfying the following axioms:*
  - *Unit: For any morphism* $f : X \to Y$*,*

$$\mathrm{id}_X; f = f$$

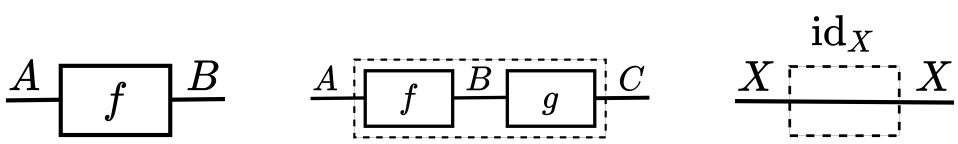
and

$$f; \mathrm{id}_Y = f$$

For example, in the category **Set** objects are sets, morphisms are functions, composition is function composition, and the identity morphisms are the identity functions on any given set.

2.5.2 String Diagrams – Categories

String diagrams are used to visualize the morphisms and objects of a category in such a way that if two morphism are equal according to the axioms, then their string diagram representations are visually similar. String diagrams visualize arrows as boxes and their domain/codomain as wires protruding from the left/right side of the box (respectively). For example an arrow $f : A \to B$ is illustrated as a string diagram in Fig. 2.14a. The strength of string diagrams comes from how they illustrate arrows such as composites and identities. For some $g : B \to C$ the composite arrow $f; g : A \to C$ is visualized by connecting the boxes via their common wire B (see Fig. 2.14b), and for any object X the identity id_X is visualized as a wire with no box (see Fig. 2.14c).



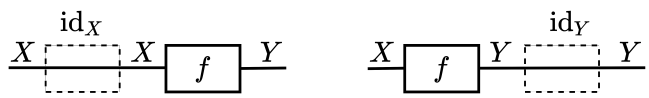
(a) $f : A \to B$ (b) $f; g : A \to C$ (c) $id_X : X \to X$

Figure 2.14: String Diagrams For Categories

As is illustrated by Fig. 2.15, the associativity axiom is visualized as the fact that for any morphisms $f : W \to X$, $g : X \to Y$, and $h : Y \to Z$, the string diagram for $(f; g); h$ is equivalent to the string diagram for $f; (g; h)$. Similarly, Fig. 2.16a and Fig. 2.16b illustrate the left and right unit laws of the identity morphisms.

(a) $(f;g);h$ (b) $f;(g;h)$

Figure 2.15: Associativity of composition with string diagram



(a) $id_X;f$ (b) $f;id_Y$

Figure 2.16: Unit laws with string diagrams

2.5.3 Monomorphisms, Epimorphisms, and Isomorphisms

Monomorphisms Monomorphisms generalize injective mappings between sets.

Definition 6. *A* monomorphism *is a morphism $f : Y \to Z$ satisfying the following:*

- *Monic: For any morphisms $g, h : X \to Y$,*

$$g;f = h;f \implies g = h$$

If a morphism is a monomorphism, it is often described as "monic."

Epimorphisms Epimorphisms generalize surjective mappings between sets.

Definition 7. *An* epimorphism *is a morphism $f : X \to Y$ satisfying the following:*

- *Epic: For any morphisms $g, h : Y \to Z$,*

$$f;g = f;h \implies g = h$$

If a morphism is an epimorphism, it is often described as "epic".

Isomorphisms Isomorphisms generalize bijective mappings between sets.

Definition 8. *An* isomorphism *is a morphism $f : X \to Y$ satisfying the following:*

- *Invertible: There exists some morphism $f^{-1} : Y \to X$ such that*

$$f; f^{-1} = \mathrm{id}_X$$

and

$$f^{-1}; f = \mathrm{id}_Y$$

2.5.4 Functors

Functors are structure-preserving maps between categories.

Definition 9. *A* Functor *F from $\boldsymbol{C}_1$ to $\boldsymbol{C}_2$ consists of:*

- *A function F_{ob} from $|\boldsymbol{C}_1|$ to $|\boldsymbol{C}_2|$*
- *For each pair of objects $X, Y \in |\boldsymbol{C}_1|$ a function $F_{X,Y}$ from $\boldsymbol{C}_1(X, Y)$ to $\boldsymbol{C}_2(F_{ob}(X), F_{ob}(Y))$ satisfying the following axioms:*
  - *Composition Preservation: For any morphisms $f : X \to Y$, and $g : Y \to Z$,*

$$F_{X,Z}(f; g) = F_{X,Y}(f); F_{Y,Z}(g)$$

  - *Identity Preservation: For any object $X \in |\boldsymbol{C}_1|$,*

$$F_{X,X}(\mathrm{id}_X) = \mathrm{id}_{F_{ob}(X)}$$

The subscripts of the object and morphism mappings are often omitted, instead using just the functor symbol for both. Brackets are also often omitted. For example $F_{ob}(X)$ is often denoted FX, and $F_{X,Y}(f)$ is denoted Ff.

2.5.5 Natural Transformations

Natural transformations are structure preserving maps between functors.

Definition 10. *A* Natural Transformation α *from* $F : \boldsymbol{C}_1 \to \boldsymbol{C}_2$ *to* $G : \boldsymbol{C}_1 \to \boldsymbol{C}_2$ *consists of:*

- *For each object* $X \in |\boldsymbol{C}_1|$*, a morphism* α_X *in* $\boldsymbol{C}_2$ *from* FX *to* GX *satisfying the following axiom:*
  - *Naturality: For any morphism* $f : X \to Y$ *in* $\boldsymbol{C}_1$*,*

$$\alpha_X ; Gf = Ff ; \alpha_Y$$

Each morphism α_X is called the *component* of α at X.

Natural Isomorphisms Natural isomorphisms are natural transformations whose components are isomorphisms.

2.5.6 Categorical Product

Categorical products generalize the Cartesian product of sets.

Definition 11. *Given family of objects* $\{X_i\}_{i \in I}$ *from a category* $\boldsymbol{C}$*, the product (if it exists) is an object* $\prod_{i \in I} X_i$ *equipped with a projection morphism* π_j *for each* $j \in I$ *from* $\prod_{i \in I} X_i$ *to* X_j *satisfying the following axiom:*

- *Universal: For any object $Q \in |\boldsymbol{C}|$ and any family of morphisms $\{f_i : Q \to X_i\}_{i \in I}$, there exists a unique morphism*

$$(f_i)_{i \in I} : Q \to \prod_{i \in I} X_i$$

such that for all $i \in I$,

$$(f_i)_{i \in I}; \pi_i = f_i$$

For any family of morphisms $\{f_i : X \to Y_i\}_{i \in I}$ the unique morphism $(f_i)_{i \in I} : X \to \prod_{i \in I} Y_i$ is called the *tuple* of those morphisms.

Binary products (where $I = \{1, 2\}$) are denoted $X_1 \times X_2$. Tuples with finite morphisms can also be denoted (f_1, f_2).

2.5.7 Categorical Coproduct

Categorical coproducts generalize the disjoint union of sets.

Definition 12. *Given family of objects $\{X_i\}_{i \in I}$ from a category $\boldsymbol{C}$, the* coproduct *(if it exists) is an object $\coprod_{i \in I} X_i$ equipped with an inclusion morphism ι_j for each $j \in I$ from X_j to $\coprod_{i \in I} X_i$ satisfying the following axiom:*

- *Universal: For any object $Q \in |\boldsymbol{C}|$ and any family of morphisms $\{f_i : X_i \to Q\}_{i \in I}$, there exists a unique morphism*

$$[f_i]_{i \in I} : \coprod_{i \in I} X_i \to Q$$

such that for all $i \in I$,

$$\iota_i; [f_i]_{i \in I} = f_i$$

For any family of morphisms $\{f_i : X_i \to Y\}_{i \in I}$ the unique morphism $[f_i]_{i \in I} : \coprod_{i \in I} X_i \to Y$ is called the *cotuple* of those morphisms.

Binary coproducts (where $I = \{1, 2\}$) are denoted $X_1 \sqcup X_2$. Cotuples with finite morphisms can also be denoted $[f_1, f_2]$.

2.6 Monoidal Category Theory

Categories where morphisms have multiple input components and multiple output components (i.e. where domains and codomains are products) are of key importance, since they are useful for describing Simulink block diagrams, and for describing the update functions of Mealy machines. Such categories have a structure which is described by *Monoidal Categories*. The string diagrams introduced in Section 2.5.2 have a graphical language which is useful for illustrating the structure of monoidal categories.

2.6.1 Monoidal Categories

Definition 13. *A* monoidal *category is a category $\boldsymbol{C}$ equipped with:*

- *A function $(-) \otimes (-) : |\boldsymbol{C}| \times |\boldsymbol{C}| \to |\boldsymbol{C}|$ called the monoidal product on objects.*
- *For any 4 objects $A, B, X, Y \in |\boldsymbol{C}|$, a function $(-) \otimes (-) : \boldsymbol{C}(A, X) \times \boldsymbol{C}(B, Y) \to \boldsymbol{C}(A \otimes B, X \otimes Y)$ called the monoidal product on morphisms satisfying:*
  - *Composition Preservation: For any four morphisms $f : X \to Y$ $g : Y \to Z$, $h : A \to B$, $k : B \to C$, $(f; g) \otimes (h; k) = (f \otimes h); (g \otimes k)$.*

  - *Identity Preservation: For any two objects* $X, Y \in |\boldsymbol{C}|$, $\mathrm{id}_X \otimes \mathrm{id}_Y = \mathrm{id}_{X \otimes Y}$.

- *An object* $I \in |\boldsymbol{C}|$ *called the monoidal unit.*

- *Natural isomorphisms* α *(associator),* λ *(left unitor),* ρ *(right unitor), with components of the form:*

  - $\alpha_{X,Y,Z} : (X \otimes Y) \otimes Z \to X \otimes (Y \otimes Z)$
  - $\lambda_X : I \otimes X \to X$
  - $\rho_X : X \otimes I \to X$

  satisfying:

  - *Pentagon Identity: For any objects* $W, X, Y, Z \in |\boldsymbol{C}|$,

$$(\alpha_{W,X,Y} \otimes \mathrm{id}_Z); \alpha_{W,X \otimes Y,Z}; (id_W \otimes \alpha_{X,Y,Z}) = \alpha_{W \otimes X,Y,Z}; \alpha_{W,X,Y \otimes Z}$$

  - *Triangle Identity: For any objects* $X, Y \in |\boldsymbol{C}|$,

$$\rho_X \otimes \mathrm{id}_Y = \alpha_{X,I,Y}; \mathrm{id}_X \otimes \lambda_Y$$

While α, λ, and ρ are natural transformations, they are commonly described via their naturality conditions, rather than the explicit definitions of the functors they transform:

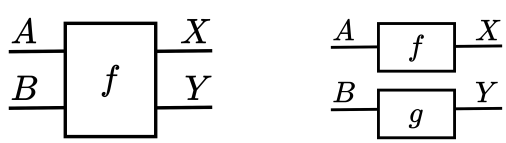
- Naturality (α): For any morphisms $f : A \to X$, $g : B \to Y$, $h : C \to Z$, $\alpha_{A,B,C}; (f \otimes (g \otimes h)) = ((f \otimes g) \otimes h); \alpha_{X,Y,Z}$

- Naturality (λ): For any morphism $f : X \to Y$, $\lambda_X; f = (\mathrm{id}_I \otimes f); \lambda_Y$.

- Naturality (ρ): For any morphism $f : X \to Y$, $\rho_X; f = (f \otimes \mathrm{id}_I); \rho_Y$.

2.6.2 String Diagrams – Monoidal Categories

The language of string diagrams can be enhanced with operations for representing the monoidal product on objects and morphisms. When the domain or codomain of a morphism is given by the monoidal product of two objects, the morphism is illustrated with multiple wires protruding from the left/right side of box (respectively). For example, a morphism $f : A \otimes B \to X \otimes Y$ is illustrated in Fig. 2.17a. If a morphism is given as the monoidal product of two morphisms $f : A \otimes X$, and $g : B \otimes Y$, the monoidal product of those two morphisms is illustrated by placing the string diagrams for each morphism side-by-side, as illustrated by Fig. 2.17b.

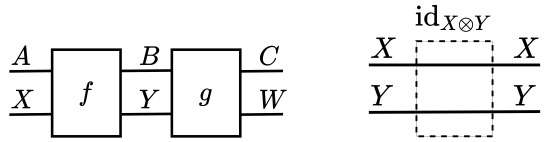


(a) $f : A \otimes B \to X \otimes Y$ (b) $f \otimes g$

Figure 2.17: Monoidal products with string diagrams

Since there is a possibility of multiple wires, the rules for representing composite and identity morphisms are slightly generalized from the single wire case. Composite morphisms connect *all* common wires between the morphisms being composed (see Fig. 2.18a), and identities are visualized as *multiple* parallel wires with no box (see Fig. 2.18b).

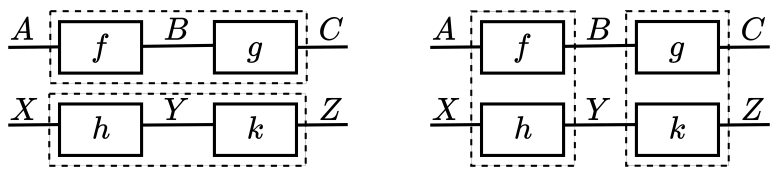
As is illustrated by Fig. 2.19, preservation of composition is visualized as the fact that the morphisms on either side of the equality have the same string diagrams (dotted boxes are used like parentheses to illustrate how each string



(a) $f; g$ (b) $\mathrm{id}_{X\otimes Y}$

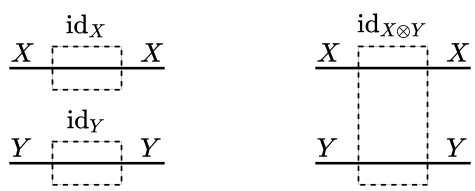
Figure 2.18: String diagrams for Monoidal Categories

diagram is built). Similarly, Fig. 2.20 illustrates the preservation of composition (dotted boxes are used to highlight the individual identity morphisms).



(a) $(f; g) \otimes (h; k)$ (b) $(f \otimes h); (h \otimes k)$

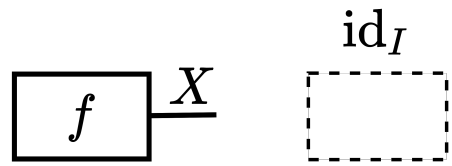
Figure 2.19: Preservation of composition with string diagram



(a) $\mathrm{id}_X \otimes \mathrm{id}_Y$ (b) $\mathrm{id}_{X\otimes Y}$

Figure 2.20: Preservation of identity with string diagram

The monoidal unit is represented in string diagrams as the absence of a wire. For example, Fig. 2.21a illustrates a morphism from I to X. This notation extends to the representation of various other special morphisms, for example, the identity morphism id_I is also represented by the absence of a wire (see Fig. 2.21b).



(a) $f : I \to X$ (b) id_I

Figure 2.21: String diagrams for Monoidal Categories

2.6.3 Symmetric Monoidal Categories

Monoidal categories can be equipped with a suitable notion of commutativity via a natural transformation called braiding.

Definition 14. *A* Symmetric Monoidal Category *is a monoidal category* $\boldsymbol{C}$*, equipped with a natural isomorphism* Br *with components of the form* $\mathrm{Br}_{X,Y} : X \otimes Y \to Y \otimes X$ *satisfying:*

1. *Symmetry: For any objects* $X, Y \in |\boldsymbol{C}|$, $\mathrm{Br}_{X,Y} = \mathrm{Br}_{Y,X}^{-1}$.

2. *Hexagon Identity: For any objects* $X, Y, Z \in |\boldsymbol{C}|$,

$$(\mathrm{Br}_{X,Y} \otimes \mathrm{id}_Z); \alpha_{Y,X,Z}; (\mathrm{id}_Y \otimes \mathrm{Br}_{X,Z}) = \alpha_{X,Y,Z}; (\mathrm{Br}_{X,Y\otimes Z}); \alpha_{Y,Z,X}$$

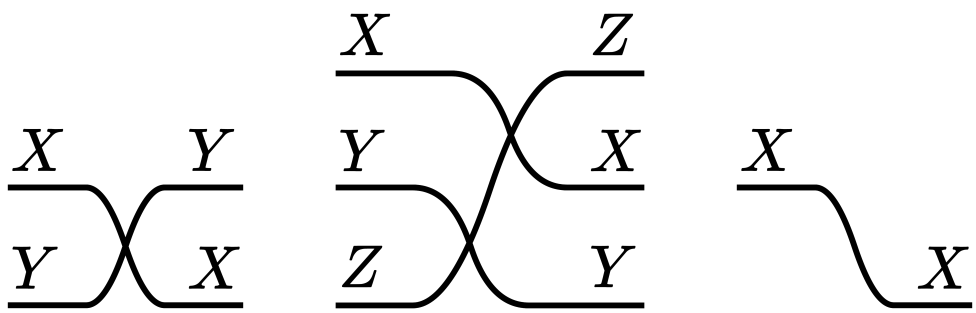
Much like the associator and unitor natural isomorphisms, the braiding natural isomorphism is commonly described in terms of its naturality conditions rather than the functors it transforms:

1. Naturality (Br): For any morphisms $f : A \to X$, $g : B \to Y$,

$$\mathrm{Br}_{A,B}; (g \otimes f) = (f \otimes g); \mathrm{Br}_{X,Y}$$

2.6.4 String Diagrams – Symmetric Monoidal Categories

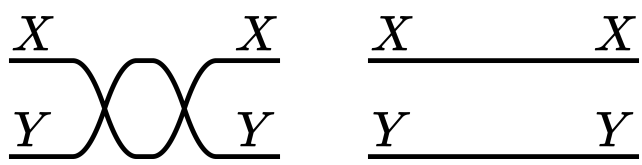
As with identity morphisms, the braiding morphism is visualized in string diagrams using plain wires. The wires from each component cross over each other (see Fig. 2.22a); if there are multiple wires (composite objects) or no wires (identity object) in either component, the wires are crossed over each other in (potentially empty) groups (see Fig. 2.22b and Fig. 2.22c).



(a) $\mathrm{Br}_{X,Y}$ (b) $\mathrm{Br}_{X\otimes Y,Z}$ (c) $\mathrm{Br}_{X,I}$

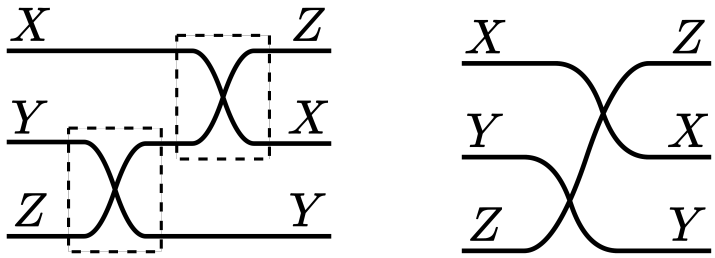
Figure 2.22: String Diagrams For Symmetric Monoidal Categories

String diagrams containing braiding morphisms are considered equivalent if the connections between dangling wires and blocks are preserved. For example, the Symmetry axiom (restated as $\mathrm{Br}_{X,Y};\mathrm{Br}_{Y,X} = \mathrm{id}_{X\otimes Y}$) is illustrated in Fig. 2.23, and Fig. 2.24 illustrates the Hexagon identity.



(a) $\mathrm{Br}_{X,Y};\mathrm{Br}_{Y,X}$ (b) $\mathrm{id}_{X\otimes Y}$

Figure 2.23: Symmetry of braiding with string diagrams



(a) $(\mathrm{id}_X \otimes \mathrm{Br}_{Y,Z}); (\mathrm{Br}_{X,Z} \otimes \mathrm{id}_Y)$ (b) $\mathrm{Br}_{X \otimes Y,Z}$

Figure 2.24: Hexagon identity with string diagrams

2.6.5 Traced Monoidal Categories

Definition 15. *A* Traced monoidal category *is a symmetric monoidal category* $\boldsymbol{C}$*, equipped with a family of* trace *operations of the form* $\mathrm{Tr}^X_{A,B} : \boldsymbol{C}(X \otimes A, X \otimes B) \to \boldsymbol{C}(A, B)$ *satisfying the following axioms:*

- *Tightening I (Naturality in A): For any two morphisms* $f : A \to B$*, and* $g : X \otimes B \to X \otimes C$*,*

$$\mathrm{Tr}^X_{A,C}((\mathrm{id}_X \otimes f); g) = f; \mathrm{Tr}^X_{B,C}(g)$$

- *Tightening II (Naturality in B): For any two morphisms* $f : X \otimes A \to X \otimes B$*, and* $g : B \to C$*,*

$$\mathrm{Tr}^X_{A,C}(f; (\mathrm{id}_X \otimes g)) = \mathrm{Tr}^X_{A,B}(f); g$$

- *Sliding (Dinaturality in X): For any two morphisms* $f : X \to Y$*, and* $g : Y \otimes A \to X \otimes B$*,*

$$\mathrm{Tr}^X_{A,B}((f \otimes \mathrm{id}_A); g) = \mathrm{Tr}^Y_{A,B}(g; (f \otimes \mathrm{id}_B))$$

- *Vanishing (I): For any morphism $f : A \to B$,*

$$\mathrm{Tr}^{I}_{A,B}(f) = f$$

- *Vanishing (II): For any morphism $f : X \otimes Y \otimes A \to X \otimes Y \otimes B$,*

$$\mathrm{Tr}^{X}_{Y\otimes A, Y\otimes B}(\mathrm{Tr}^{Y}_{A,B}(f)) = \mathrm{Tr}^{X\otimes Y}_{A,B}$$

- *Superposing: For any two morphisms $f : X \otimes A \to X \otimes B$, and $g : C \to D$,*

$$\mathrm{Tr}^{X}_{A\otimes C, B\otimes D}(f \otimes g) = \mathrm{Tr}^{X}_{A,B}(f) \otimes g$$

- *Yanking: For any object $X \in |\boldsymbol{C}|$,*

$$\mathrm{Tr}^{X}_{X,X}(\mathrm{Br}_{X,X}) = \mathrm{id}_X$$

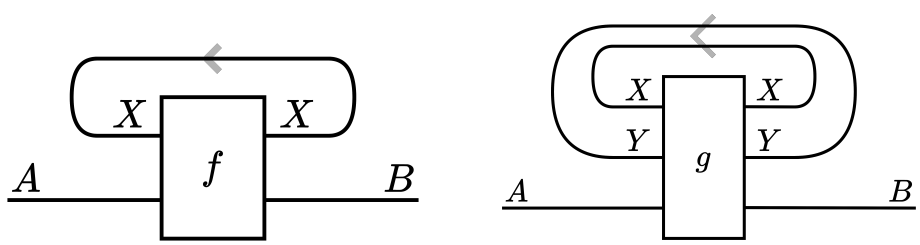
While only some categories admit a trace operation, any symmetric monoidal category can be equipped with a trace-like operation that follows all the above axioms, and where the trace-like operation is only defined if the morphism can be factored into a form where it is possible to remove the operation via the above axioms. For example, for any object X in a symmetric monoidal category, the trace-like operation Tr is defined for the morphism $\mathrm{Br}_{X,X}$, as $Tr^{X}_{X,X}(\mathrm{Br}_{X,X}) = \mathrm{id}$ follows from the yanking axiom. However, for any object X other than the monoidal unit I, the trace-like operation Tr is not defined for $\mathrm{id}_X \otimes \mathrm{id}_Y$, as $\mathrm{Tr}^{X}_{Y,Y}(\mathrm{id}_X \otimes \mathrm{id}_Y)$ is not able to be expressed without a trace operation.

The notion of this trace-like operation is discussed in [Goncharov and

Schröder 2018], where it is the guarded trace operation on the vacuously guarded category. In this thesis, the trace-like operation will be referred to as a trace, with the understanding that it is only applicable to some morphisms, and that for the remaining morphisms, the operation is undefined.

2.6.6 String Diagrams – Traced Monoidal Categories

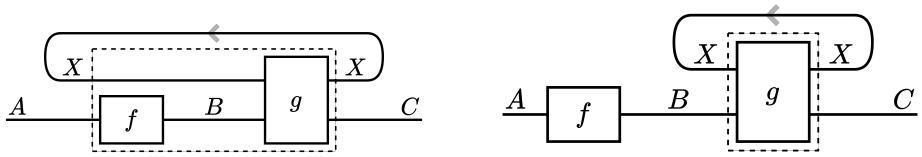
The Trace operation is represented in string diagrams by routing wires (corresponding to the traced component) from the output back into the input. As with other diagramatic notation, this can be applied to (potentially empty) groups of wires to represent traces where the traced component is a product, or the monoidal unit.



(a) $\mathrm{Tr}^{X}_{A,B}(f)$ (b) $\mathrm{Tr}^{X\otimes Y}_{A,B}(g)$

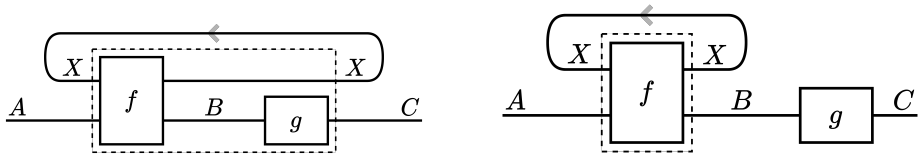
Figure 2.25: String diagrams for traced categories

Much like for symmetric monoidal categories, string diagrams for traced monoidal categories are considered equivalent if the connections between dangling wires and blocks are preserved. The axioms of traced monoidal categories are illustrated by the string diagrams in Fig. 2.26, Fig. 2.27, Fig. 2.28, Fig. 2.29, Fig. 2.30, Fig. 2.31, and Fig. 2.32, where the dashed boxes are used to indicate the parts of the diagram to which the traces are being applied, and the grey chevrons are used to indicate the wires being fed back as part of a trace.



(a) $\mathrm{Tr}^X_{A,C}((\mathrm{id}_X \otimes f); g)$ (b) $f; \mathrm{Tr}^X_{B,C}(g)$

Figure 2.26: Tightening (I) with string diagrams



(a) $\mathrm{Tr}^X_{A,C}(f; (\mathrm{id}_X \otimes g))$ (b) $\mathrm{Tr}^X_{A,B}(f); g$

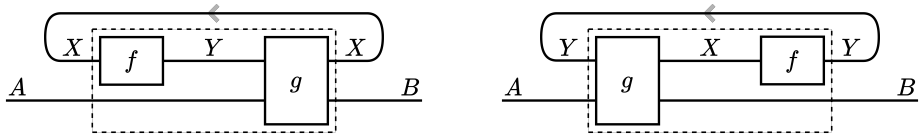
Figure 2.27: Tightening (II) with string diagrams

2.6.7 Cartesian Monoidal Categories

A cartesian monoidal category is a category equipped with two natural transformations: the *diagonal* transformation with components of the form Δ_X; and the *deletion* transformation with components of the form $!_X$.

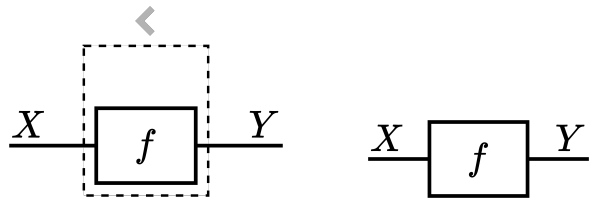
Definition 16. *A* cartesian monoidal category *is a symmetrical monoidal category* $\boldsymbol{C}$ *equipped with:*

- *Natural transformations* Δ *(diagonal) and* $!$ *(deletion) with components of the form:*
  - $\Delta_{\mathrm{X}} : \mathrm{X} \to \mathrm{X} \otimes \mathrm{X}$
  - $!_{\mathrm{X}} : \mathrm{X} \to \mathrm{I}$



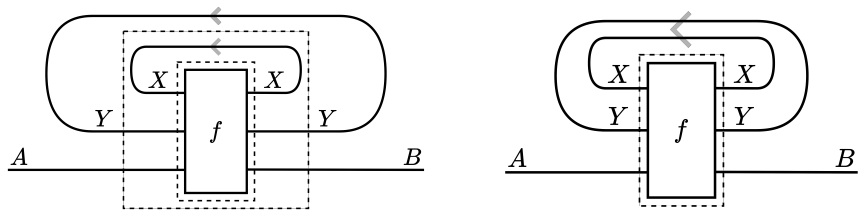
(a) $\mathrm{Tr}^X_{A,B}((f \otimes \mathrm{id}_A); g)$ (b) $\mathrm{Tr}^Y_{A,B}(g; (f \otimes \mathrm{id}_B))$

Figure 2.28: Sliding with string diagrams



(a) $\mathrm{Tr}^{I}_{A,B}(f)$ (b) f

Figure 2.29: Vanishing (I) with string diagrams



(a) $\mathrm{Tr}^{X}_{Y\otimes A,Y\otimes B}(\mathrm{Tr}^{Y}_{A,B}(f))$ (b) $\mathrm{Tr}^{X\otimes Y}_{A,B}$

Figure 2.30: Vanishing (II) with string diagrams

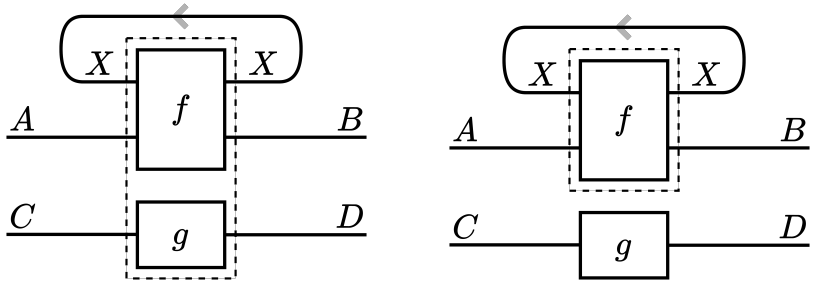
satisfying:

– *Unit: For any object* $X \in |C|$,

$$\Delta_X; (!_X \otimes \mathrm{id}_X); \lambda_X = \mathrm{id}_X$$

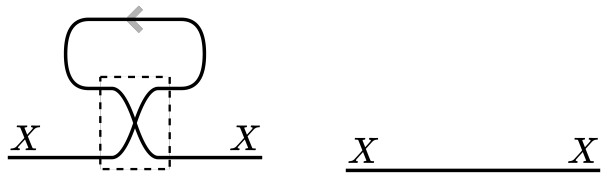
and

$$\Delta_X; (\mathrm{id}_X \otimes !_X); \rho_X = \mathrm{id}_X$$



(a) $\mathrm{Tr}^{X}_{A\otimes C,B\otimes D}(f \otimes g)$ (b) $\mathrm{Tr}^{X}_{A,B}(f) \otimes g$

Figure 2.31: Superposing with string diagrams



(a) $\mathrm{Tr}^{X}_{X,X}(\mathrm{Br}_{X,X})$ (b) id_X

Figure 2.32: Yanking with string diagrams

The naturality conditions for Δ and ! are:

- Naturality (Δ): For any morphism $f : X \to Y$,

$$f; \Delta_Y = \Delta_X; (f \otimes f)$$

- Naturality (!): For any morphism $f : X \to Y$,

$$f; !_Y = !_X; \mathrm{id}_I$$

In addition, the natural transformations must satisfy the following:

- Monoidal Product Preservation (Δ): For any objects $X, Y \in |C|$,

$$\Delta_{X \otimes Y} = (\Delta_X \otimes \Delta_Y); (id_X \otimes \mathrm{Br}_{X,Y} \otimes id_Y)$$

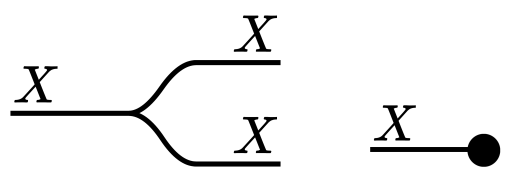
- Terminal (I): For any Morphism $f : X \to I$,

$$f = !_X$$

The above structure is sufficient to determine that the monoidal product on objects ($\otimes$) is the categorical product ($\times$) as defined in Section 2.5.6 (see ["Lectures on categorical quantum mechanics (2012)"]).

2.6.8 String Diagrams – Cartesian Monoidal Categories

As with identity morphisms and braiding morphisms, the diagonal and deletion morphisms are visualized in string diagrams using plain wires. Diagonal morphisms are represented by branching a (potentially empty) group of wires, and deletion morphisms are represented by terminating a (potentially empty) group of wires with a dot.

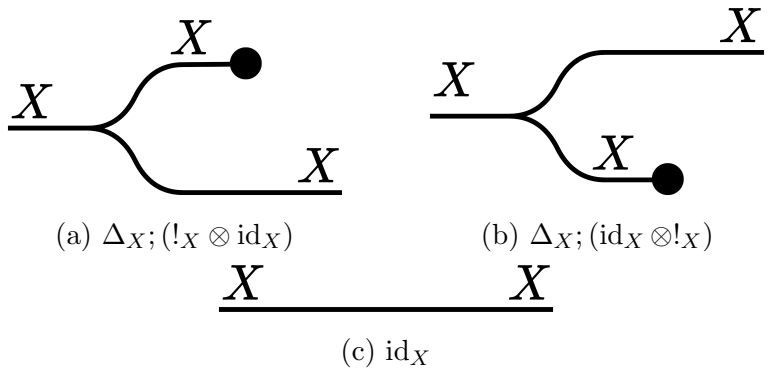


(a) Δ_X (b) $!_X$

Figure 2.33: String diagrams for Cartesian monoidal categories

The equivalence of string diagrams thus far has been simply determined by preserving the connections between blocks and dangling wires. A similar rule governs equivalence of combinations of diagonal and deletion morphisms, where string diagrams are equivalent if connections are preserved, accounting for branching wires connecting a single source to multiple destinations, and terminating wires connecting a single source to no destinations. So long as the same sources are connected to the same destinations by some wiring, the string diagrams are equivalent, as for example is illustrated in Fig. 2.34.

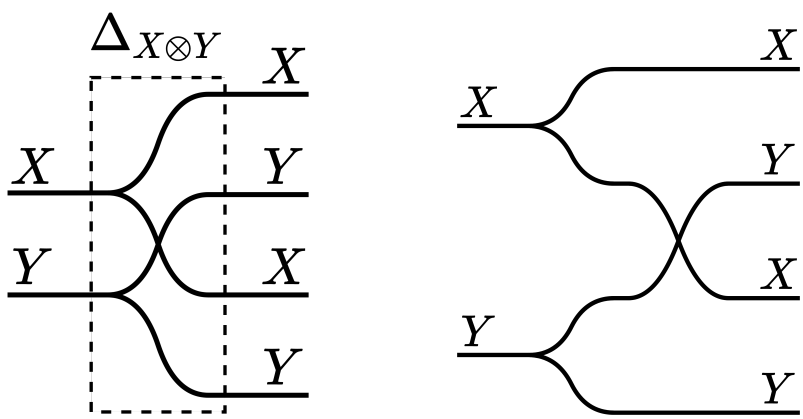
However, maintaining connections between blocks is not the only way to preserve equivalence between string diagrams containing diagonal and deletion morphisms. This is clearly illustrated by the naturality conditions in Fig. 2.36 and Fig. 2.37, illustrating that blocks can be introduced or removed from a string diagram, while maintaining equivalence. These equivalences can be summarized by a simple manipulation: one may replace blocks lead-



(a) $\Delta_X; (!_X \otimes \mathrm{id}_X)$ (b) $\Delta_X; (\mathrm{id}_X \otimes !_X)$

(c) id_X

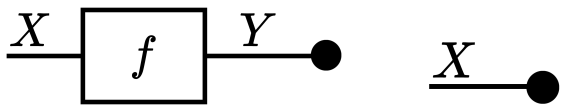
Figure 2.34: Counit axiom with string diagrams



(a) $\Delta_{X \otimes Y}$ (b) $(\Delta_X \otimes \Delta_Y); (\mathrm{id}_X \otimes \mathrm{Br}_{X,Y} \otimes \mathrm{id}_Y)$

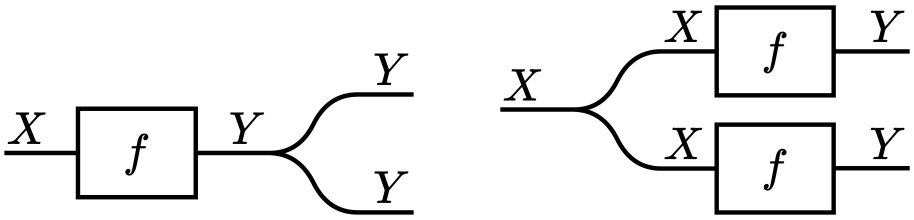
Figure 2.35: Uniform copying with string diagrams

ing to the input of a branch/termination point with the same blocks applied to each wire leading out. For termination points, there are no wires leading out, and so the block is simply no longer present in the block diagram. Two string diagrams are equivalent if one can be manipulated into the other by a sequence of changes that either perform the above manipulation, reverses the above manipulation, or preserves connections between blocks. While this is less straightforward than the simple equivalences for all other wiring morphisms, this rule of equivalence is well known in disciplines where block diagrams are used.



(a) $f; !_X$ (b) $!_X$

Figure 2.36: Naturality (!) with string diagrams



(a) $f; \Delta_X$ (b) $\Delta_X; (f \otimes f)$

Figure 2.37: Naturality (Δ) with String Diagrams

Chapter 3

Refactoring Strategy Overview

In order to improve the quality of embedded software models, this thesis presents a refactoring strategy for replacing fragments of block diagrams with behaviourally equivalent state charts. A key feature of the strategy is that it accounts for circumstances where block diagrams do not encode state machines, and a state chart would be a less intuitive representation of the system. Such cases are described in version 3 of the MAB guidelines on mixed use of Simulink and Stateflow [The MathWorks 2012]:

na_0006: Guidelines for mixed use of Simulink and Stateflow

> The choice of whether to use Simulink or Stateflow to model a given portion of the control algorithm functionality should be driven by the nature of the behavior being modeled.
>
> - If the function primarily involves complicated logical operations, use Stateflow diagrams.
> - Stateflow should be used to implement modal logic – where the control function to be performed at the cur-

rent time depends on a combination of past and present logical conditions.

- If the function primarily involves numerical operations, use Simulink features.

Similar to the Model Advisor tool designed by Mathworks to support many of the MAB guidelines, the presented strategy automates two important refactoring activities. First, the *detection* of cumbersome block diagram fragments which would be easier to understand as state charts, and second, automated *correction*. Supporting both detection and correction activities facilitates a proactive approach to improving the quality of Simulink/Stateflow models. An identification strategy for detection is presented together with a translation strategy for correction.

For both identification and translation, two classes of Simulink blocks are given special consideration. The first are *storage* blocks, such as unit delays or data stores. These blocks have some notion of *memory* which allows them to produce outputs which depend on previous inputs, not just the current one. These blocks are essential to state machines implemented as block diagrams, as their memory stores the system state between time steps. The second important class are *decision logic* blocks, such as switch blocks, or logical operators. These blocks calculate their output in distinct ways depending on certain conditions being met. For example, switch blocks will output the value received on the top input or the bottom input depending on the value of the middle input.

3.1 Digital Logic Circuits

State transition graphs and state transition tables have historically been used as an aid for understanding the behaviour of sequential digital logic circuits. Sequential logic circuits contain *shift registers* which output a stored value which gets updated at each rising edge of a clock signal. Combinatorial logic circuit components are then used to use the inputs and values stored by the shift registers to produce output values, and the new values to be stored by the shift register at the next clock pulse.

The translation between sequential digital logic circuits and state transition diagrams has been extensively studied. One of the seminal papers on the topic describes the process for taking a digital logic circuit and producing from it a state transition graph [Mealy 1955]. The process is to enumerate the behaviours of the circuit by constructing a truth table which records the values produced by the circuit for each possible combination of input and state variable. Then each combination of state variable values is interpreted as a distinct state in the state space, each combination of input values is interpreted as a distinct input symbol from the input alphabet, and each combination of output values is interpreted as a distinct output symbol from the output alphabet. By interpreting the truth table this way, the truth table can be rewritten as a state transition table. From the state transition table, a state transition graph is easily obtained by creating a transition for each row in the table. This strategy can be summarized as the following three steps:

1. Convert the sequential logic circuit to a truth table

2. Convert the truth table to a state transition table

3. Convert the state transition table to a state transition graph

The key to this translation strategy is that state transition graphs and state transition tables are alternative representation of the same data. For the Mealy machine describing any digital logic circuit, transition graphs and tables just enumerate the *maplets* of the update function.

3.2 Generalizing for Simulink

The process of translating digital logic circuits involves enumerating each possible combination of input and state variable values, and is therefore not directly applicable to translation of general block diagrams which may have inputs which are not practically enumerable (e.g. floating point numbers). However by tweaking this strategy to not consider individual values, but rather *equivalence classes* of values, it can be applied practically to Simulink block diagrams. This is achieved by replacing truth tables with tabular expressions.

For example, consider the the block diagram in Fig. 3.1. Modelled as a Mealy machine, its update function and initial state is given by the tabular expression in Fig. 3.2. When q is true, the space of inputs is split into two equivalence classes: if $temp < setpoint - 2$, q will remain true and so will *coilOn* and *fanOn*; if $temp \geq setpoint - 2$, q will become false and so will *coilOn* and *fanOn*. Similarly to how each combination of state variable values was interpreted as a distinct mode, we can assign a mode to each value of the state variable, yielding the state transition table illustrated in Fig. 3.3. Much like the truth tables used in the analysis of digital logic circuits, each row in the table now corresponds to a transition between two modes, as illustrated by Fig. 3.4.

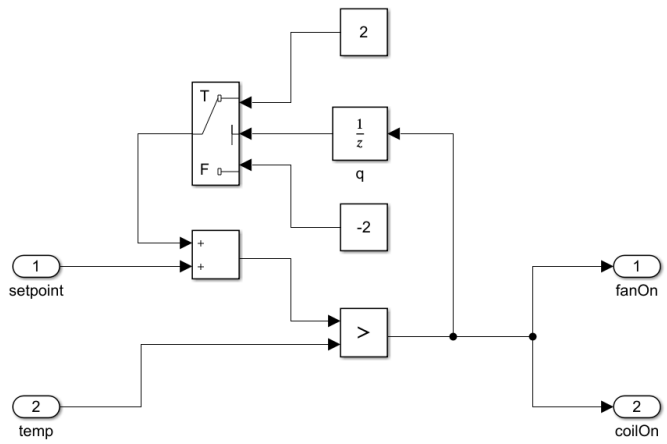


Figure 3.1: Simple Simulink block diagram to be translated

		$fanOn$	$coilOn$	q'
$\neg q$	$temp < setpoint - 2$	T	T	T
	$temp \geq setpoint - 2$	F	F	F
q	$temp < setpoint + 2$	T	T	T
	$temp \geq setpoint + 2$	F	F	F

Figure 3.2: HCT representing the behaviour of the block diagram in Fig. 3.1

Source	**Condition**	$fanOn$	$coilOn$	**Target**
Off	$temp < setpoint - 2$	T	T	*Heat*
	$temp \geq setpoint - 2$	F	F	*Off*
Heat	$temp < setpoint + 2$	T	T	*Heat*
	$temp \geq setpoint + 2$	F	F	*Off*

Figure 3.3: STT which replicates the behaviour of the HCT in Fig. 3.2

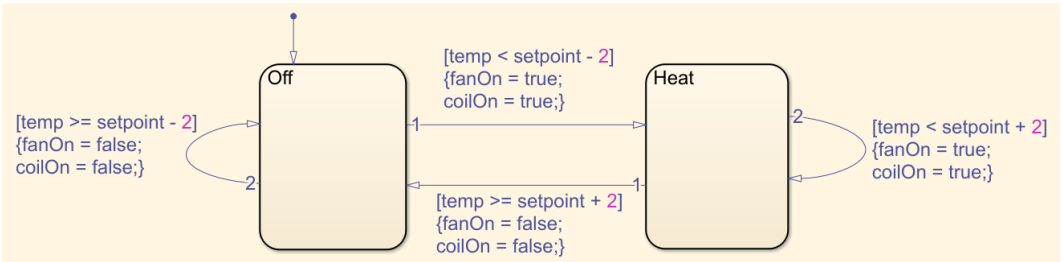


Figure 3.4: Stateflow state chart equivalent to the STT in Fig. 3.3

The previous example grouped inputs into equivalence classes based on the individual value of the state variables, however state variables may also be grouped into equivalence classes. To illustrate how state variables can be grouped into equivalence classes, consider the block diagram in Fig. 3.5. Modelled as a Mealy machine, its update function and initial state is given by the tabular expression in Fig. 3.6. This block diagram behaves in the same way as the one in Fig. 3.1, only its state space is given by a floating point value, rather than a Boolean one. By examining the tabular expression, it is clear that the exact value of *error* does not actually matter, all that matters is whether *error* is greater than 0, or less than 0. Instead of interpreting each distinct state in the state space as a mode, by interpreting each equivalence class of states as a mode, the table can be used to enumerate each way the state chart can react from a given starting mode, as illustrated by Fig. 3.7. However, unlike the example above, each row in Fig. 3.7 does not lead to a distinct target mode. For example, the first row can lead to a state where *error* is either greater than 0 or less than 0, depending on the values of *temp* and *setpoint*. This means that the first row could generate two transitions: one which leads to the mode corresponding to $error > 0$, and one which leads to the mode corresponding to $error \geq 0$. Each transition is thus guarded by an additional condition which determines whether the corresponding state condition of the target mode (e.g. $error > 0$) holds for the updated value of *error*. Each additional guarding condition for these transitions is generated by substituting the updated value for *error* from the row into the state condition of the target mode. The result of adding such conditions to each row in Fig. 3.7 is illustrated by an additional condition column, as illustrated in Fig. 3.8. In some cases the existing conditions obviate the newly added conditions. This is

the case for the conditions introduced in Fig. 3.8, where some are implied by the prior condition(s), and others contradict prior conditions. Each row that is guarded by a contradictory sub-condition (shaded red in Fig. 3.8) can be removed, as they are inaccessible; in a similar manner sub-conditions implied by prior conditions (shaded green in Fig. 3.8) can be removed, as they can be absorbed by the conditions that imply their truth. Removing the superfluous cells from Fig. 3.8 yields the STT in Fig. 3.9. Note that rows with contradictory conditions may contain updated variable values that do not match the target modes, as is the case in this example, where the target mode of the removed rows does not match the values of the output variables *fanOn* and *coilOn*.

Now that each row in the table has a unique source and target mode, a state chart can be created similarly to how it was done in the previous example. Note that while the starting block diagram was more complex, the final state chart shown in Fig. 3.10 is almost the same.

Tabular expressions play a key role in this translation as they act to bridge the gap between block diagrams and state charts. The main attributes of the behaviour modelled in the syntax of state charts are decision logic and modes (i.e. states). These same attributes however are not modelled by the syntax of block diagrams, which model the data flow and connections between blocks without regard for what kind of behaviours those blocks have. By first translating to tabular expressions, the internal decision logic and stateful behaviour of each block in a block diagram is explicitly represented, and a compositional approach can be applied to capture these aspects of the system as a whole. Having made the decision logic and state variables of the block diagram explicit, the second step of the translation leverages the syntax of HCTs to identify the conditions that encode the modes of operation, namely those

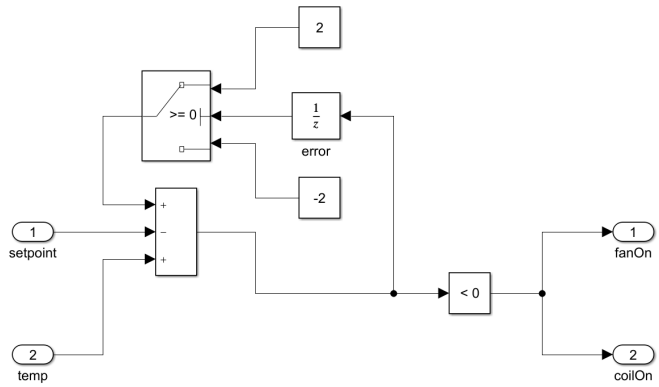


Figure 3.5: Simulink block diagram with numerical state

		fanOn	*coilOn*	*error′*
$error \geq 0$	$temp < setpoint - 2$	T	T	$temp - setpoint + 2$
	$temp \geq setpoint - 2$	F	F	$temp - setpoint + 2$
$error < 0$	$temp < setpoint + 2$	T	T	$temp - setpoint - 2$
	$temp \geq setpoint + 2$	F	F	$temp - setpoint - 2$

Figure 3.6: HCT representing the behaviour of the block diagram in Fig. 3.5

which depend only on the value of the state variables. Identifying these same conditions directly from the block diagram is a complex task; these stateful conditions arise through the implicit behaviour of stateful blocks, the implicit behaviour of decision logic blocks, and an analysis of how those blocks combine with each other, and with the rest of the block diagram. Contrastingly, condition cells of HCTs can simply be analysed to determine which variables they reference.

This translation strategy can be summarized as the following three steps

		fanOn	*coilOn*	*error′*
Off	$temp < setpoint - 2$	T	T	$temp - setpoint + 2$
	$temp \geq setpoint - 2$	F	F	$temp - setpoint + 2$
Heat	$temp < setpoint + 2$	T	T	$temp - setpoint - 2$
	$temp \geq setpoint + 2$	F	F	$temp - setpoint - 2$

Figure 3.7: STT without Target state

Source	Conditions		*fanOn*	*coilOn*	*error′*	**Target**
Off	$temp < setpoint - 2$	$temp - setpoint + 2 \geq 0$	T	T	$temp - setpoint + 2$	*Off*
		$temp - setpoint + 2 < 0$	T	T	$temp - setpoint + 2$	*Heat*
	$temp \geq setpoint - 2$	$temp - setpoint + 2 \geq 0$	F	F	$temp - setpoint + 2$	*Off*
		$temp - setpoint + 2 < 0$	F	F	$temp - setpoint + 2$	*Heat*
Heat	$temp < setpoint + 2$	$temp - setpoint - 2 \geq 0$	T	T	$temp - setpoint - 2$	*Off*
		$temp - setpoint - 2 < 0$	T	T	$temp - setpoint - 2$	*Heat*
	$temp \geq setpoint + 2$	$temp - setpoint - 2 \geq 0$	F	F	$temp - setpoint - 2$	*Off*
		$temp - setpoint - 2 < 0$	F	F	$temp - setpoint - 2$	*Heat*

Figure 3.8: Adding conditions to the table in Fig. 3.7 to determine target states

		fanOn	*coilOn*	*error′*	**Target**
Off	$temp < setpoint - 2$	T	T	$temp - setpoint + 2$	*Heat*
	$temp \geq setpoint - 2$	F	F	$temp - setpoint + 2$	*Off*
Heat	$temp < setpoint + 2$	T	T	$temp - setpoint - 2$	*Heat*
	$temp \geq setpoint + 2$	F	F	$temp - setpoint - 2$	*Off*

Figure 3.9: Removing superfluous cells from the STT in Fig. 3.8

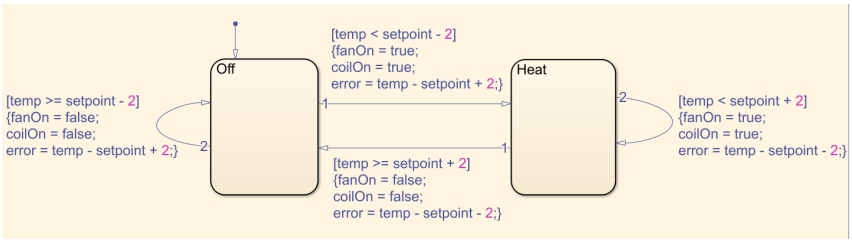


Figure 3.10: Stateflow state chart equivalent to Fig. 3.5

Digital Logic	**SL2SF**
Sequential Logic Circuit	Discrete-Time Block Diagram
Combinational Logic	Decision Logic Blocks
Register	Storage Block
Truth Table	Horizontal Condition Table
State Transition Table	Extended State Transition Table
State Transition Graph	State Chart
Input Symbol	Guarding Condition
Maplet	Partial Function

Figure 3.11: Digital circuit analysis vs Simulink to Stateflow refactoring strategy

which mirror the ones from the digital logic circuits case.

1. Convert the Simulink block diagram to a tabular expression (i.e. HCT)
2. Convert the tabular expression to an (extended) state transition table (i.e. STT)
3. Convert the STT to a state chart

Where the rows and transitions in the case of digital logic circuits represented individual maplets in the update function of mealy machines, the expressions in state charts and tabular expressions allow rows and transitions to represent equivalence classes of maplets given by a partial function.

The table in Fig. 3.11 maps concepts from digital sequential logic to their analogues in the approach presented in this thesis.

The analogy to sequential digital logic circuits has an additional benefit: it gives a framework for state machines in terms of data flow. To paraphrase the MAB guidelines, the type of behaviours that are amenable to being modelled as a state chart are those where "The present action can be broken down into

enumerable simpler actions" and those simpler actions can be "grouped based on some logical conditions evaluated in the past".

While [Mealy 1955] considers translation between sequential logic circuits and state transition diagrams in both directions, the goal of the generalized translation is to aid readability of Simulink models by capturing mode switching logic in a language that makes such mode switching logic explicit. If however, translation in the opposite direction were to be considered, the steps used to translate a block diagram to a state chart could be reversed. Converting a subset of state charts to state transition tables has been considered in [Singh et al. 2015], and a horizontal condition table could be produced from this state transition table by keeping the mode variable explicit, ranging over enumerated values associated with each state of the state chart. From there, step one of the translation could be reversed by producing a block diagram implementing the tabular expression. A naive approach to this would be to cascade switch blocks to implement the if-else structure of the nested conditions of the horizontal condition table, and use the logical operator and use logical/mathematical operator blocks to directly implement the expressions from the table to feed the conditions/values of the switch blocks. While it is likely that this block diagram could be subsequently simplified, it is unlikely that the resulting model will be as easy to understand as the original state chart.

3.3 Application of Category Theory

The role of category theory in this work is to provide a unified framework for composition. The categorical framework helps to unify the presentation of the

semantics via categories of functions, as well as categories of Mealy machines. Monoidal categories have proven to be a useful language for describing both the piecewise nature of decision logic functions, and for the componentwise nature of multi-input multi-output systems. This language often yields elegant descriptions for otherwise cumbersome operations.

Furthermore, without category theory, obtaining a HCT for an entire block diagram systematically is a non-trivial task. In [Mealy 1955] the prescribed process to obtain a truth table for a digital logic circuit was to "write the equations," and then "evaluate them for all possible input values." A truth table for a digital logic circuit can be created trivially by brute force: there is a unique truth table for any logic circuit, and it can be obtained by enumerating all possible inputs and evaluating the functions represented by each circuit component to determine what their outputs will be. Such a process is not possible for Simulink, where all inputs are not practically enumerable. Monoidal category theory provides a compositional structure that can be applied to make this task manageable. Using this compositional structure, this task will be achieved by first obtaining HCTs for each individual block in the block diagram, then by combining those tables according to the data flow between the blocks, yielding a combined HCT for the entire diagram.

Chapter 4

Identification: Finding Candidate Subsystems

4.1 Data Flow Schemas

The goal of the identification component of the refactoring strategy presented in this thesis is to analyze Simulink models to detect subsystems containing stateful decision logic which would be more appropriately represented in Stateflow. To achieve this, ideas about the construction of sequential logic circuits are adapted to Simulink block diagrams (Section 4.1.1), and extended with conditions for finding stateful decision logic (Section 4.1.2).

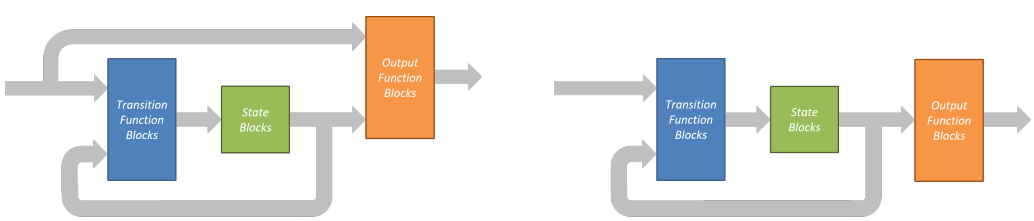
4.1.1 Characteristic Data Flow

As discussed in Section 3.1, sequential logic circuits are used to represent state machines, where the state is given by the values that are stored in each shift register (i.e. storage block). By examining the construction of sequential logic circuits, two classic schemas for combining shift registers and combinational

logic components will be used to produce schemas for identifying state machines implemented in block diagrams.

According to the definition of Mealy machines given in Section 2.3.2, the current output and next state of the state machine are calculated using a single function, the update function $ud : \Sigma \times S \rightarrow \Lambda \times S$. However in the study of sequential logic circuits, the update function is often considered as being comprised of two parts: the transition function ($tf : \Sigma \times S \rightarrow S$) which calculates the next state; and the output function ($of : \Sigma \times S \rightarrow \Lambda$) which calculates the current output. Given a Mealy machine defined in terms of these two component functions, the update function can be obtained by pairing the output function and transition function (i.e. $ud = (of, tf)$).

By considering the distinct parts of the update function, it is possible to construct a schema for how combinational logic circuits that implement these functions are combined with shift registers to implement the Mealy machine. The schema in Fig. 4.1a illustrates how the state blocks (i.e. shift registers) are connected to the combinational logic components that make up the transition function and output function.



(a) Mealy (b) Moore

Figure 4.1: Schemas for Mealy and Moore machines

A similar schema is provided in Fig. 4.1b. In this schema, the current input is *not* used to determine the current output; only the state impacts the output. This schema shows how *Moore machines* are constructed. Moore machines,

like Mealy machines, are used to model behaviours, and while Moore machines are not used to model behaviours in this thesis, the schema for circuit diagrams of Moore machines is important for capturing another form that state machines can take when expressed in a data flow language.

As Simulink block diagrams contain much more than the shift registers and combinational logic components of sequential logic circuits, the schemas in Fig. 4.1 cannot be directly applied to generalized block diagrams. These schemas can however be generalized by dividing Simulink blocks into two categories: *stateful* blocks with some form of dynamic behaviour (e.g. Delay blocks, and Data Store Read/Write blocks); and *stateless* blocks which calculate their output values using only the current values of their inputs (e.g. Switch blocks and Sum blocks). This classification of blocks can be used to match Simulink block diagrams to the classical schemas of Mealy and Moore machines. Instead of the transition function and output function being comprised of combinational logic components, they are instead made up of any stateless blocks. Similarly, the state blocks in the schemas are comprised of stateful Simulink blocks.

For example, Fig. 4.2 illustrates how a Simulink block diagram may conform to the schema in Fig. 4.1b. The requirement for one or more state blocks is satisfied by the single unit delay block, and the transition and output functions are a combination of stateless numerical blocks (Constant, Gain, and Sum blocks), and stateless decision logic blocks (Switch and Logical Operator blocks).

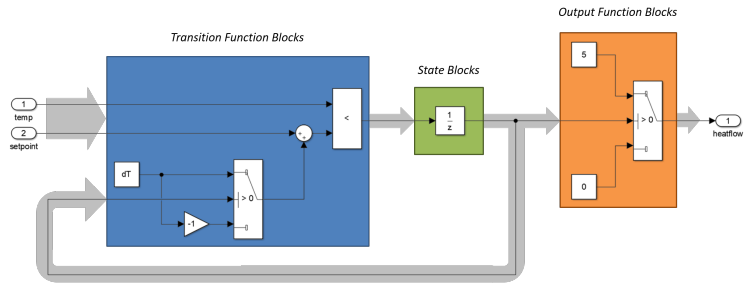


Figure 4.2: Block Diagram Conforming to Moore Schema

4.1.2 Augmenting with Decision Logic Requirement

It is however important to note that these schemas are not exclusive to the types of systems that are naturally represented by state charts (i.e. stateful mode-switching logic). The schemas will also match block diagrams representing dynamic systems, even when the model primarily involves numerical calculations. For example, a simple PID controller would match the Mealy schema, but it is best represented using a block diagram, not a state chart.

In order to avoid detection of block diagrams that are not suitable to be represented as state charts, we will supplement the schemas with an additional condition which will determine whether the state machine modelled by the block diagram is able to take advantage of the features of state charts. Notably, state charts encode decision logic by acting in distinct ways based on which transition is taken when moving from the previous mode to the next mode. In order for a block diagram to effectively leverage this feature of state charts, it must express a state machine with such decision logic. Since Simulink block diagrams are only able to express decision logic by the inclusion of decision logic blocks, it is clear that block diagrams must include decision logic blocks in order to be candidates for refactoring. Additionally, the decision logic encoded in the transitions of state charts implicitly uses the previous

mode when determining what the next mode should be. Therefore the decision logic blocks in candidate Simulink block diagrams must consider the previous state when determining the next state. Since the next state of a block diagram which matches the Mealy or Moore schema is governed by the transition logic, the decision logic blocks must lie on the feedback path between the output of the state blocks and inputs to the state blocks.

For example, the block diagram in Fig. 4.3 highlights the blocks and signals on the feedback path of the block diagram in Fig. 4.2. The feedback path contains decision logic blocks (i.e. the relational operator and switch blocks) and therefore it is identified as a candidate for refactoring.

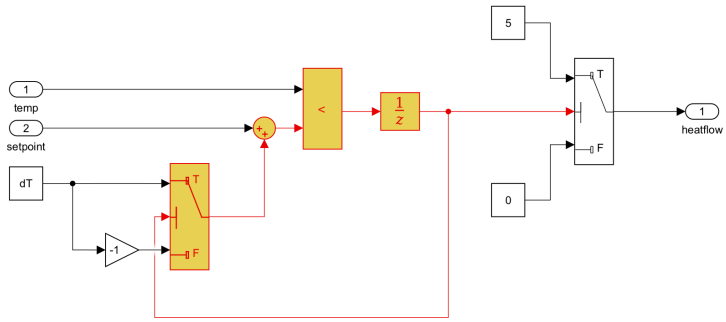


Figure 4.3: Feedback Containing Decision Logic

4.2 Identification Algorithm

The data flow schemas described in Section 4.1 can be used to describe block diagrams that are candidates for refactoring to stateflow. In this section, the data flow schemas are used as the basis of an algorithm which can identify subsystems in a Simulink model which may be partially or fully replaced by a Stateflow state chart block.

While there are various techniques that can be applied to matching patterns in models (e.g. [Kolovos and Paige 2017]), one large challenge is that of *implicit* data flow. One example of implicit data flow is the data dependency between GoTo and From blocks in Simulink. This detail means that models can conform to the schemas in terms of semantic data flow, but fail to conform to the schemas when it comes to their syntactic structure. The inclusion of GoTo and From blocks are quite common, especially in models which would otherwise have messy feedback loops. This means that the data flow schemas in Section 4.1 can be difficult to detect using traditional techniques. For this reason, an imperative algorithm was developed which uses the Reach/Coreach tool [McSCert 2018] to manage the intricacies of implicit data flow.

In order to automatically determine whether a block diagram conforms to the schemas in Fig. 4.1, the following steps are performed on each subsystem in a Simulink model. For a subsystem S:

1. Find all unit delay and memory store read/write blocks that are within S. This includes any blocks that are directly contained within S, and any blocks that are found by recursively searching within child subsystems, including masked subsystems and library links. Perform the following steps for found block B.

2. Find the sets D and U of all blocks and signals directly contained in S that are *downstream* or *upstream* of B (respectively).

    - Note: the downstream/upstream trace analysis is performed using the capabilities of the Reach/Coreach tool. This accounts for implicit data flow, e.g. data flow between GoTo/From blocks

3. Find the set F of all blocks in the feedback path of B by taking the intersection of D and U (the intersection is empty if there is no feedback).

4. If there are any decision logic blocks (e.g. Switch or RelationalOperator blocks) mark S as a candidate for refactoring.

Any subsystem identified by this algorithm will contain at least one fragment satisfying the conditions described in Section 4.1.2, since it is directly identifying feedback loops containing decision logic.

4.3 Evaluation

In order to evaluate this strategy for identifying parts of a Simulink model which may be replaced with a Stateflow state chart, the decisions made by the algorithm were compared to decisions made by an expert from industry, who was asked to identify subsystems within an industrial model which were candidates for being refactored into Stateflow state charts. The model being analysed was a large industrial-scale Simulink model containing many subsystems, for which the expert was responsible for maintaining.

In our test, in which over 50 subsystems were analysed, the algorithm agreed with the decision made by the expert roughly 75% of the time. Of the 25% of subsystems where the algorithm disagreed with the expert, some models were *false positives* (the expert identified them as not viable, the algorithm identified them as viable), and some were *false negatives* (the expert identified them as viable, the algorithm identified them as not viable).

The false negatives were investigated further, because a user may decide not to translate a subsystem if they do not agree with the result of the identification

algorithm, but if the tool never suggests a viable subsystem, the user may not consider it at all.

We discussed a sample of false negatives with the expert, and discovered the following reasons for the mismatches:

1. The subsystem was considered "potentially refactorable" at the time of labelling, but in hindsight, it is not refactorable

2. The subsystem could be refactored, but not into a Stateflow state chart

3. The subsystem can be refactored into a Stateflow state chart, but it does not contain the feedback pattern

Subsystems which were false negatives for reason 1 or reason 2 indicate that the identification algorithm worked as expected.

However, some false negatives were subsystems which could be understood as a state machine, but were not identified by the algorithm. While no single pattern accounted for all such cases, some examples nearly conformed to the Mealy schema, only they did not contain a feedback loop. By removing the feedback loop from the Mealy schema, a new schema can be produced, as illustrated by Fig. 4.4. Because it does not contain the characteristic feedback

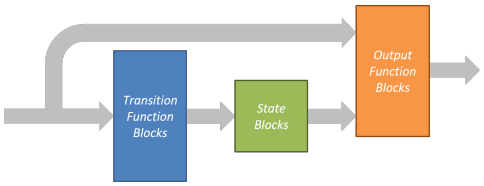


Figure 4.4: Schema Not Found By Algorithm

loop, block diagrams conforming to this schema are not considered refactorable by the methodologies described in Section 4.1, and will not be identified by the algorithm in Section 4.2.

Block diagrams which conformed to this schema would implement state machines where the previous mode is not considered in the determination of the next mode, only the current inputs. These state machines are somewhat unnatural, because any mode is accessible from any other mode, and the same conditions guard each transition leading to any mode. Ultimately, while some of these block diagrams may be better represented as a state chart, it is likely that they would be further improved by refactoring to a different representation.

4.4 Summary

The data flow schemas described in this section provide a framework for identifying subsystems which are candidates for refactoring. Furthermore, an algorithm is described for automated detection of subsystems conforming to these schemas. This algorithm facilitates proactive refactoring activities wherein large industrial models are analysed to identify subsystems within them which employ complex stateful decision logic. Once identified, these subsystems can be refactored using the strategy described in the following chapters.

Chapter 5

Translation I: Block Diagrams to Tables

The goal of this first step in the translation is to model an entire block diagram as a Mealy machine whose update function is represented as a tabular expression. This is achieved by modelling individual blocks as Mealy machines whose update functions are represented by some static tabular expression, and then combine these Mealy machines to get a Mealy machine for the entire block diagram. To combine the Mealy machines they are composed as morphisms in some monoidal category. Since no such category exists, we present a suitable category **Mealy** which frames the existing composability of Mealy machines as the operations of a monoidal category. One key to our presentation of **Mealy** is that instead of defining composition in terms of functions specifically, it defines composition in terms of the monoidal structure of **Set**; reasoning about the categorical structure of **Mealy** is founded in the categorical structure of **Set**. This gives us the ability to transform the task of composing Mealy machines as morphisms in the monoidal category **Mealy** into the task of composing their

update functions as morphisms in the monoidal category **Set**. This is used to take block diagrams where blocks are Mealy machines to produce a block diagram for its update function, consisting entirely of functional blocks, where each block is defined as a tabular expression.

This translation strategy can be applied to any block diagram valid for embedded code generation, so long as: (1) the update function of each block in the block diagram can be represented as a tabular expression; and (2) each block in the block diagram only contains data ports (i.e. input/output ports). The compositional framework would have to be extended in order to model port types used for control flow (e.g. action ports on If-Action subsystems), and so they are not covered by this translation strategy.

5.1 Combining HCTs

HCTs represent functions with multiple inputs and outputs. In this section two syntactic operations are described: one for combining two HCTs into an HCT representing the composite of the functions of the original HCTs; and one for combining two HCTs into an HCT representing the monoidal product of the functions of the original HCTs. These operations are similar to the operations presented in [Mohrenschildt 2000].

5.1.1 Sequential Composition of HCTs

When composing two HCTs sequentially, the conditions of the first HCT appear first in the composed HCT and the conditions of the second HCT are included as sub-conditions. The conditions from the second HCT are evaluated using the output values from the first one. Consider, for example, the

		y
$x_1 > 0$		$x_1 - 1$
$x_1 \leq 0$	x_2	10
	$\neg x_2$	0

(a) HCT_1

	z
$y > 0$	y
$y \leq 0$	0

(b) HCT_2

			z
$x_1 > 0$	$x_1 - 1 > 0$		$x_1 - 1$
	$x_1 - 1 \leq 0$		0
$x_1 \leq 0$	x_2	$10 > 0$	10
		$10 \leq 0$	0
	$\neg x_2$	$0 > 0$	0
		$0 \leq 0$	0

(c) $HCT_1; HCT_2$

Figure 5.1: Sequentially composing tables

composition of Fig. 5.1a with Fig. 5.1b, where the output y of the first table is routed to the input y of the second. Their composition is shown in Fig. 5.1c. The conditions $x_1 > 0$ and x_2 (and their complements) appear in the same configuration as the first HCT. However, the sub-conditions (e.g. $x_1 - 1 \leq 0$) come from the conditions ($y \leq 0$) in the second HCT, evaluated with the values ($y \mapsto x_1 - 1$) from the row in the first HCT associated with the parent condition ($x_1 > 0$). The conditions $10 > 0$ and $0 > 0$ (and their complements) are generated in a similar manner, however these express tautological or unsatisfiable conditions. In cases where condition is always true, it can be removed as a sub-condition of the table. Similarly, when a condition always evaluates to false the entire row can be removed. The removable conditions/rows are shaded (green and red respectively) in Fig. 5.1c).

Similarly to the conditions, the output expressions of the second HCT are evaluated with the corresponding values from the first HCT, and those are used as the output expressions of the combined HCT.

		z
$x_1 > 0$	$x_1 - 1 > 0$	$x_1 - 1$
	$x_1 - 1 \leq 0$	0
$x_1 \leq 0$	x_2	10
	$\neg x_2$	0

Figure 5.2: Simplified composite table

		y_1
$x_1 > 0$		$x_1 - 1$
$x_1 \leq 0$	x_2	10
	$\neg x_2$	0

(a) HCT_1

	y_2
$x_3 > 0$	x_3
$x_3 \leq 0$	0

(b) HCT_2

			y_1	y_2
$x_1 > 0$	$x_3 > 0$		$x_1 - 1$	x_3
	$x_3 \leq 0$		$x_1 - 1$	0
$x_1 \leq 0$	x_2	$x_3 > 0$	10	x_3
		$x_3 \leq 0$	10	0
	$\neg x_2$	$x_3 > 0$	0	x_3
		$x_3 \leq 0$	0	0

(c) $HCT_1 \otimes HCT_2$

Figure 5.3: Monoidal product of tables

5.1.2 Monoidal Product of HCTs

If two HCTs are combined in parallel, then the conditions from one table do not impact the conditions in the next. However, all conditions must be represented in the table. By convention, the tables will be combined such that the conditions from the first table appear at the root of the product table, and the conditions from the second table appear as subconditions of those conditions.

5.2 Mealy Category

The first step of the translation strategy is to model the entire block diagram as a Mealy machine whose update function is represented as a HCT. To achieve this, Simulink block diagrams are modelled in a category **Mealy**, where morphisms (i.e. blocks) are Mealy machines, not functions.

5.2.1 Definition

The basic construction is given below:

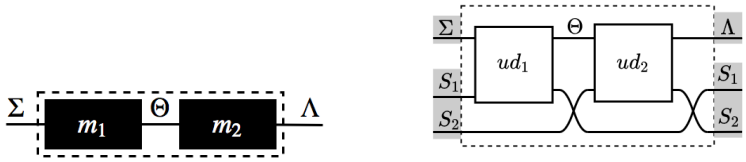
Definition 17. *By considering equivalence up to bisimilarity, Mealy machines form a category* **Mealy**, *where:*

- *Objects are sets*
- *Morphisms* $m : \Sigma \to \Lambda$ *are Mealy machines with input alphabet* Σ, *and output alphabet* Λ
- *Composition is given by cascade composition*
- *The identity morphism is given by the Mealy machine* $\mathrm{id}_X = (\mathbb{1}, (), X, X, \{(x, ()) \mapsto (x, ())\})$

To prove that Mealy machines form a category (where equivalence is considered up to bisimilarity), one must prove the axioms of a category hold. That is, one must show that composition is associative, and that the identity morphism is a left and right unit with respect to the composition. This has been shown for a similar structure (symmetric lenses) in [Hofmann, Pierce, and Wagner 2011]. The structure of symmetric lenses is sufficiently similar to Mealy machines that the proofs for the categorical structure have the same form, and will not be repeated here.

The following expression of cascade composition is equivalent to the definition given in Section 2.3.4:

Definition 18. *Given two Mealy machines* $m_1 = (S_1, s_0^1, \Sigma, \Theta, ud_1)$ *and* $m_2 = (S_2, s_0^2, \Theta, \Lambda, ud_2)$, *where the output alphabet of* m_1 *is the same as the input alphabet of* m_2, *the sequential composite Mealy machine* $m_1; m_2$ *is given by*



(a) $m_1; m_2$ (b) Update function of $m_1; m_2$

Figure 5.4: Composition of **Mealy** morphisms

$(S_1 \times S_2, (s_0^1, s_0^2), \Sigma, \Lambda, ud')$ *where* $ud' = (ud_1 \otimes id_{S_2}); (id_\Theta \otimes \mathrm{Br}_{S_1,S_2}); (ud_2 \otimes id_{S_1}); (id_\Lambda \otimes \mathrm{Br}_{S_2,S_1})$.

The update function of composite Mealy machines is shown in terms of the update functions of the component Mealy machines in Fig. 5.4.

While this definition is less compact than the one given previously, it will be useful to express the update function of Mealy machines using the monoidal categorical structure of sets.

5.2.2 Functions Embedded in Mealy

To address the fact that a large part of a Simulink block diagram *looks* very functional (i.e. stateless), we consider a class of Mealy machines which produce outputs as a function of only their current inputs. For example, wiring in block diagrams and many Simulink blocks can be modelled as functions, rather than Mealy machines.

Any function $f : X \to Y$ can be embedded into **Mealy** as the Mealy machine $\mathcal{M}f = (\mathbb{1}, (), X, Y, f \otimes \mathrm{id}_{\mathbb{1}})$, with one state, and update function $f \otimes \mathrm{id}_{\mathbb{1}}$ (see Fig. 5.5).

The mapping $\mathcal{M}$ *embeds* morphisms from **Set** into the category **Mealy**, because any two embedded functions $\mathcal{M}f$ and $\mathcal{M}g$ interact in **Mealy** very similarly to the way they interact as functions in **Set**.

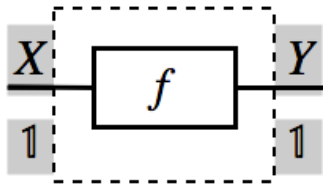


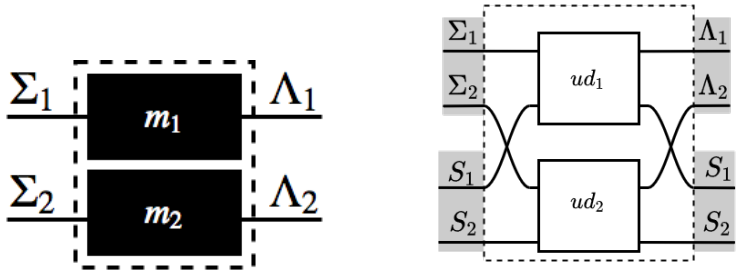
Figure 5.5: Update function of $\mathcal{M}f$

5.2.3 Additional Categorical Structure

The category **Mealy** forms a symmetric monoidal category, where:

- The monoidal product on objects is given by the Cartesian product on Sets
- The monoidal product on morphisms is given by the parallel composition of Mealy machines
- The monoidal unit is given by the singleton set, which contains only the empty tuple $\mathbb{1} = \{()\}$
- The natural transformations λ, ρ, α, and Br are defined in terms of the natural transformations from the monoidal category of sets and functions:
  - λ has components $\lambda_X = \mathcal{M}l_X$
  - ρ has components $\rho_Z = \mathcal{M}r_X$
  - α has components $\alpha_{X,Y,Z} = \mathcal{M}a_{X,Y,Z}$
  - $\mathrm{Br}_{X,Y}$ has components $\gamma_{X,Y}$

  where l, r, a, and γ are the left unitor, right unitor, associator, and braiding natural transformations from the monoidal category of sets and functions.



(a) $m_1 \otimes m_2$ (b) Update function of $m_1 \otimes m_2$

Figure 5.6: Monoidal product of **Mealy** morphisms

Again, the structure of **Mealy** is sufficiently similar to the structure discussed in [Hofmann, Pierce, and Wagner 2011] that the proofs for the properties of this categorical structure take the same shape, and will not be repeated here.

Similarly to sequential composition, parallel composition can be expressed in terms of monoidal operations on the category of sets and functions:

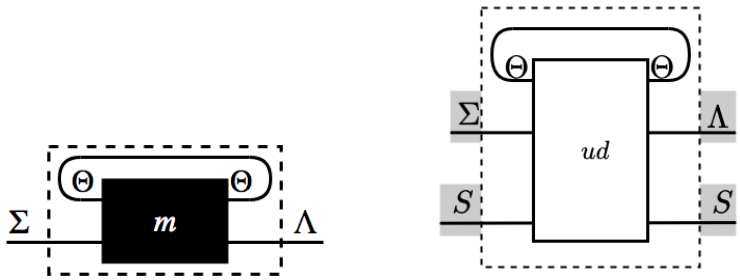
Definition 19. *Given two Mealy machines* $m_1 = (S_1, s_0^1, \Sigma_1, \Lambda_1, ud_1)$ *and* $m_2 = (S_2, s_0^2, \Sigma_2, \Lambda_2, ud_2)$*, the parallel composite Mealy machine* $m_1 \otimes m_2$ *is given by* $(S_1 \times S_2, (s_0^1, s_0^2), \Sigma_1 \otimes \Sigma_2, \Lambda_1 \otimes \Lambda_2, ud')$ *where*

$$ud' = (\mathrm{id}_{\Sigma_1} \otimes \mathrm{Br}_{\Sigma_2,S_1} \otimes \mathrm{id}_{S_2}); (ud_1 \otimes ud_2); (\mathrm{id}_{\Lambda_1} \otimes \mathrm{Br}_{S_1,\Lambda_2} \otimes \mathrm{id}_{S_2})$$

.

The update function of the monoidal product of Mealy machines is shown in terms of the update functions of the component Mealy machines in Fig. 5.6.

Additionally, **Mealy** is a *Cartesian* monoidal category, where the natural transformations Δ and ! are defined in terms of the natural transformations from the Cartesian monoidal category of sets and functions.



(a) $\mathrm{Tr}^{\Theta}_{\Sigma,\Lambda}(m)$ (b) Update function of $\mathrm{Tr}^{\Theta}_{\Sigma,\Lambda}(m)$

Figure 5.7: Traced **Mealy** morphisms

- Δ has components $\mathcal{M}\Delta_X$
- ! has components $\mathcal{M}!_X$

Finally, **Mealy** can be equipped with a trace-like operation, where the operation is defined in terms of the trace-like operation on functions as follows:

Definition 20. *Given a Mealy machine* $m = (S, s_0, \Theta \otimes \Sigma, \Theta \otimes \Lambda, ud)$*, the traced Mealy machine* $\mathrm{Tr}^{\Theta}_{\Sigma,\Lambda}$*, if it exists, is given by* $(S, s_0, \Sigma, \Lambda, \mathrm{Tr}^{\Theta}_{\Sigma\otimes S,\Lambda\otimes S}\, ud)$.

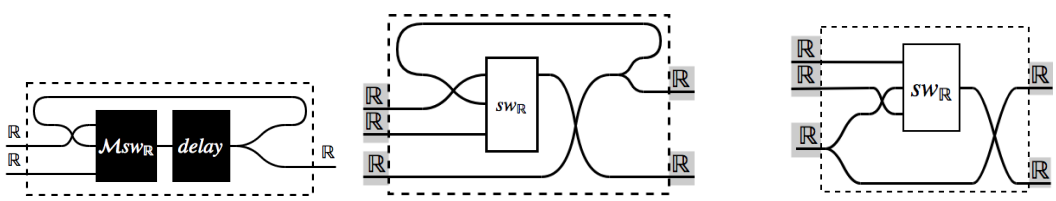
The update function of a traced Mealy machine is shown in terms of the update function of the Mealy machine being traced in Fig. 5.7.

Since the trace is not defined for some functions, it is also not defined for some Mealy machines. The trace operation is defined for a Mealy machine if and only if the trace operation on the update function as applied in Definition 20 is defined.

While this trace-like operation is quite restrictive, it maps quite well to the rules of Simulink block diagram construction. In Simulink, cycles in block diagrams are often invalid because they produce what is called an *algebraic loop*. Algebraic loops occur when the cycle creates a dependency where the value that will be input to a block depends on some computation which needs

the current output of the same block. These loops can often be resolved by adding a unit delay to the cycle, since the current output of a unit delay block does not depend on its current input. Since unit delays are modelled by the Mealy machine $(X, x_0, X, X, \mathrm{Br}_{X,X})$, adding the unit delay is introducing a symmetry operation to the cycle, such that the loop can be "yanked out". This is to say that the update function of a Mealy machine for a Simulink block diagram can be represented by combining component update functions without the use of trace operations.

Take for example the Mealy machines combined in the string diagram in Fig. 5.8a. Combining the update functions of the component Mealy machines according to the definitions in this section yields the string diagram in Fig. 5.8b. However, the string diagram in Fig. 5.8b is equivalent to the string diagram in Fig. 5.8c, which contains no traces.



(a) Mealy Machine (b) Traced Function (c) Trace-Free Function

Figure 5.8: Representing block diagrams without traces

All axioms for diagonal/deletion maps and trace operations of **Mealy** can be shown to follow from the axioms of diagonal/deletion maps and trace operations on sets and functions. For example, given a Mealy machine $m = (S, s_0, \Sigma, \Lambda, ud)$, the naturality condition for Δ is $m; \Delta_\Lambda = \Delta_\Sigma; (m \otimes m)$. From

the definitions, $m; \Delta_\Lambda$ is roughly equal[1] to $(S, s_0, \Sigma, \Lambda \otimes \Lambda, ud_l)$ where

$$ud_l = ud; (\Delta_\Lambda \otimes \mathrm{id}_S)$$

and $\Delta_\Sigma; (m \otimes m)$ is roughly equal to $(S \otimes S, (s_0, s_0), \Sigma, \Lambda \otimes \Lambda, ud_r)$ where

$$ud_r = (\Delta_\Sigma \otimes \mathrm{id}_{S \otimes S}); (\mathrm{id}_\Sigma \otimes \mathrm{Br}_{\Sigma,S} \otimes \mathrm{id}_S); (\mathrm{id}_\Lambda \otimes \mathrm{Br}_{S,\Lambda} \otimes \mathrm{id}_S)$$

These Mealy machines are bisimilar, where the bisimulation relation is Δ_S. The following shows that

$$ud_l; (\mathrm{id}_{\Lambda \otimes \Lambda} \otimes \Delta_S) = (\mathrm{id}_\Sigma \otimes \Delta_S); ud_r$$

.

[1]Note that these Mealy machines are already slightly simplified as to remove the $\mathbb{1}$ components from their state spaces; this avoids lengthy representations of the update functions, and clearly results in a bisimilar Mealy machine, as the bisimulation relation is given by an isomorphism.

$$
\begin{aligned}
& ud_l; (\mathrm{id}_{\Lambda\otimes\Lambda} \otimes \Delta_S) \\
&= ud; (\Delta_\Lambda \otimes \mathrm{id}_S); (\mathrm{id}_{\Lambda\otimes\Lambda} \otimes \Delta_S) && \text{definition of } ud_l \\
&= ud; ((\Delta_\Lambda; \mathrm{id}_{\Lambda\otimes\Lambda}) \otimes (\mathrm{id}_S; \Delta_S)) && \text{comp. pres. } (\otimes)^2 \\
&= ud; (\Delta_\Lambda \otimes \Delta_S) && \text{identity laws} \\
&= ud; \Delta_{\Lambda\otimes S}; (\mathrm{id}_\Lambda \otimes \mathrm{Br}_{S,\Lambda} \otimes \mathrm{id}_S) && \text{uniform copying} \\
&= \Delta_{\Sigma\otimes S}; (ud \otimes ud); (\mathrm{id}_\Lambda \otimes \mathrm{Br}_{S,\Lambda} \otimes \mathrm{id}_S) && \text{naturality } (\Delta) \\
&= (\Delta_\Sigma \otimes \Delta_S); (\mathrm{id}_\Sigma \otimes \mathrm{Br}_{\Sigma,S} \otimes \mathrm{id}_S); && \text{uniform copying} \\
&\qquad (ud \otimes ud); (\mathrm{id}_\Lambda \otimes \mathrm{Br}_{S,\Lambda} \otimes \mathrm{id}_S) \\
&= ((\mathrm{id}_\Sigma; \Delta_\Sigma) \otimes (\Delta_S; \mathrm{id}_{S\otimes S})); (\mathrm{id}_\Sigma \otimes \mathrm{Br}_{\Sigma,S} \otimes \mathrm{id}_S); && \text{identity laws} \\
&\qquad (ud \otimes ud); (\mathrm{id}_\Lambda \otimes \mathrm{Br}_{S,\Lambda} \otimes \mathrm{id}_S) \\
&= (\mathrm{id}_\Sigma \otimes \Delta_S); (\Delta_\Sigma \otimes \mathrm{id}_{S\otimes S}); (\mathrm{id}_\Sigma \otimes \mathrm{Br}_{\Sigma,S} \otimes \mathrm{id}_S); && \text{comp. pres. } (\otimes) \\
&\qquad (ud \otimes ud); (\mathrm{id}_\Lambda \otimes \mathrm{Br}_{S,\Lambda} \otimes \mathrm{id}_S) \\
&= (\mathrm{id}_\Sigma \otimes \Delta_S); ud_r && \text{definition of } ud_r
\end{aligned}
$$

Since $m; \Delta_\Lambda = \Delta_\Sigma; (m \otimes m)$, the naturality condition for Δ holds.

5.3 Translation Strategy

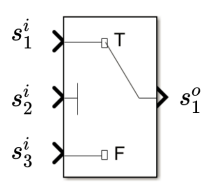
As described at the start of this section, the goal of this first step in the translation is to produce a HCT describing the update function of the entire

[2] comp. pres. $(\otimes)$ – composition preservation $(\otimes)$

block diagram to be transformed. This is broken down into steps that can be performed automatically. This section describes each of those steps, and provides an example.

5.3.1 Expressing The Block Diagram as a Collection of HCTs

To automate this first step, the translation tool generates a unique name for each input and output port on each block in the block diagram. Then, for each block, it uses the type of the block and associated parameters for that block to generate an instance of a HCT, where the input variables are named according to the input ports to the block and the output variables are named according to the output ports to the block. For example, the switch block in Fig. 5.9a would yield the HCT found in Fig. 5.9b.



(a) A switch block s

	s_1^o
s_2^i	s_1^i
$\neg s_2^i$	s_3^i

(b) HCT for switch block s

Figure 5.9: HCT for a switch block

Then, each wire in the block diagram is used to produce a single HCT which maps the values from its source port to one or more output ports that it is routed to. E.g., a simple wire which connects the first output port of block a to the second input port of block b would produce a HCT which achieves the mapping $\{a_1^o \mapsto b_2^i : a_1^o\}$. A branching wire from the first output port of block a to the second input port of block b and the first input port of block c

would produce a HCT which achieves the mapping $\{a_1^o \mapsto (b_2^i : a_1^o, c_1^i : a_1^o)\}$.

While each type of block requires a distinct strategy in order to perform this first step, any block diagram comprised of these blocks can be handled automatically. The prototype tool developed alongside this thesis handles a subset of block types (about 25 block types), which has been sufficient to work for the majority of block diagrams that are candidates for translation.

5.3.1.1 Generating HCTs for specific block types

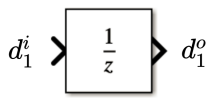
While the process for generating HCTs for individual block types is not generally important for the overall strategy, the treatment of some blocks are relevant.

Blocks with no logical conditions associated with their behaviour generate a HCT with a single row. The condition for that row is simply *true*. For example, a sum block would produce the HCT shown in Fig. 5.12.

Inport and Outport blocks each produce HCTs with an input/output variable that maps between the input/output of the block diagram to the internal variable associated with the output/input ports on the blocks. For example, an Inport block named x would generate an internal variable x_1^o (the first output of the input block). The HCT would reference an input variable x associated with this block, and would achieve the mapping $\{x \mapsto x_1^o : x\}$

Unit delay blocks produce a state variable for tracking the stored value. This variable is represented by two variables referenced by the HCTs: one for holding the previously stored value, and one for holding the value to be newly stored. Unit delay blocks are unique in that they produce two HCTs. The first HCT maps the value of the internal variable associated with the input of the unit delay block to the variable that holds the newly stored value, and the

second maps the variable holding the previously stored value to the output. For example, a unit delay block named d (Fig. 5.10a) would generate internal variables d_1^i and d_1^o, along with variables d and d' for the previously stored and newly stored values (respectively). The update function of d is represented by a single HCT in Fig. 5.10b, however, it can be decomposed into the monoidal product of the two HCTs in Fig. 5.11. Additionally, the initial value of the unit delay block is noted for future use, and is associated with the generated state variable.



(a) A delay block d

	d_1^o	d'
$true$	d	d_1^i

(b) HCT for delay block d

Figure 5.10: HCT for a delay block

	d_1^o
$true$	d

(a) HCT_1

	d'
$true$	d_1^i

(b) HCT_2

Figure 5.11: Decomposing Fig. 5.10b into two HCTs, where $HCT = HCT_1 \otimes HCT_2$

Logical operator blocks produce HCTs which capture the logical structure in its conditions. For example, a logical operator block performing the logical conjunction of two inputs would produce the HCT shown in Fig. 5.13.

5.3.2 Combining the Tabular Expressions

After a collection of HCTs has been produced from each block and signal line in the block diagram, the next step in automated processing is to combine them all into one large HCT representing the update function of the block

	m_1^o
$true$	$m_1^i + m_2^i$

Figure 5.12: HCT for a sum block m

		a_1^o
$\neg a_1^i$		$false$
a_1^i	$\neg a_2^i$	$false$
	a_2^i	$true$

Figure 5.13: HCT for a logical block a (AND)

diagram as a whole. In Section 5.2, it is established how the update functions of two Mealy machines are combined to form the update function of a combined Mealy machine. Since these operations are only able to combine two HCTs at once, this problem is solved using a *fold* operation, where starting with one element, each subsequent element is combined with the previous result, until all elements have been combined.

The fold operation will take two HCTs, and combining them according to the variables that are shared between the outputs of the first HCT and the inputs to the second. Since, in general, not all outputs of the first HCT will be inputs to the second HCT, any inputs/outputs that are not shared will remain as inputs/outputs of the combined HCT. Order matters in this operation, since the outputs of the first HCT are routed to the inputs of the second. To ensure that HCTs are combined in the correct order, the HCTs are sorted into a sequence where any input of a HCT is not an output of a HCT that appears later in the sequence.

To achieve this, the existing structure of the HCTs is leveraged. Note that the collection of HCTs form a directed graph, where the HCTs are nodes, and the internal variables are edges, from the single HCT with that variable as an output, to the single HCT with that variable as an input. This graph is acyclic,

as there can be no cycles in the data dependency of Simulink block diagrams (no algebraic loops). Any cycles in the initial block diagram must contain at least one unit delay block, and because the unit delay block produced two HCTs with no shared variables [3] in the previous step, that cycle does not exist amongst the collection of HCTs.

This directed acyclic graph is partially ordered, but already holds the property desired for the final sequence. Starting at any node (table) t_1, for each incoming edge (input variable) e there exists no path to a node (table) t_2 which has e as an outgoing edge (output variable), since if such a path were to exist, adding e to that path would form a cycle. To produce a sequence that maintains this property, the nodes of the existing graph can be sorted using any algorithm for topologically sorting a directed acyclic graph (e.g. Kahn's algorithm [Kahn 1962]).

Once all HCTs produced from blocks/signals lines in the block diagram have been combined, the only variables remaining will be input/output variables, and the variables used to represent the previous/next values of the state variables. This combined HCT along with the collection of initial values of each state variable describes the behaviour of the block diagram as a whole. Thus, the first step of the translation strategy is achievable in an automated fashion.

5.4 Example

Consider the block diagram in Fig. 5.14. Each block in the block diagram has an associated HCT in Fig. 5.15. The wire that starts from the output port

[3] The variables holding the previous/new stored values are considered distinct

of the start block and leads to the second input port of the SetCounter block is shown in Fig. 5.16b. Since most wires in the block diagram yield a similar definition, their definitions are omitted. For the purposes of this example, the HCTs corresponding to each wire will be referred to as $wire^{srcblk}_{tgtblk}$, where *srcblk* and *tgtblk* are the source and target blocks of the wire. One wire however is unique in that it branches. The HCT for this wire is shown in Fig. 5.16a.

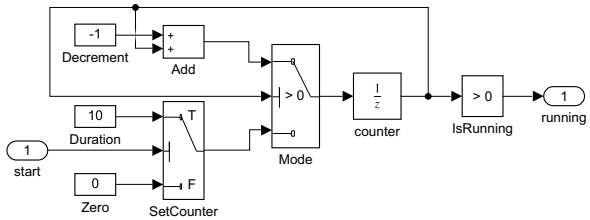


Figure 5.14: Simulink Block Diagram

$start^o_1$	
true	*start*

(a) *start*

$counter^o_1$	
true	$counter^s_1$

(b) *counter*

$decrement^o_1$	
true	-1

(c) *decrement*

$duration^o_1$	
true	10

(d) *duration*

$zero^o_1$	
true	0

(e) *zero*

add^o_1	
true	$add^i_1 + add^i_2$

(f) *add*

$setcounter^o_1$	
$setcounter^i_2$	$setcounter^i_1$
$\neg setcounter^i_2$	$setcounter^i_3$

(g) *setcounter*

$mode^o_1$	
$mode^i_2 > 0$	$mode^i_1$
$mode^i_2 \leq 0$	$mode^i_3$

(h) *mode*

$counter^{s'}_1$	
true	$counter^i_1$

(i) *counter′*

$isrunning^o_1$	
$isrunning^i_1 > 0$	*true*
$isrunning^i_1 \leq 0$	*false*

(j) *isrunning*

running	
true	$running^i_1$

(k) *running*

Figure 5.15: HCTs for blocks in Fig. 5.14

To combine these tabular expressions, they must be sorted topologically. Topological ordering is not unique, but one such ordering for these HCTs would be: $\langle start,\ wire^{start}_{setcounter},\ counter,\ wire^{counter}_{isrunning,add,mode},\ decrement,\ wire^{decrement}_{add},$

	$isrunning_1^i$	add_2^i	$mode_2^i$
$true$	$counter_1^o$	$counter_1^o$	$counter_1^o$

(a) $wire^{counter}_{isrunning,add,mode}$

	$setcounter_2^i$
$true$	$start_1^o$

(b) $wire^{start}_{setcounter}$

Figure 5.16: HCTs for selected wires in Fig. 5.14

$duration$, $wire^{duration}_{setcounter}$, $zero$, $wire^{zero}_{setcounter}$, add, $wire^{add}_{mode}$, $setcounter$, $wire^{setcounter}_{mode}$, $mode$, $wire^{mode}_{counter'}$, $isrunning$, $wire^{isrunning}_{running}$, $wire^{isrunning}_{running}$, $running\rangle$.

Then, combining the HCTs one by one in that order would yield the HCT in Fig. 5.17.

		$running$	$counter'$
$start$	$counter > 0$	$true$	$counter - 1$
	$counter \leq 0$	$false$	10
$\neg start$	$counter > 0$	$true$	$counter - 1$
	$counter \leq 0$	$false$	0

Figure 5.17: HCT representation of block diagram in Fig. 5.14

5.5 Summary

This section describes a strategy for producing a HCT representing the update function of a block diagram in its entirety. The HCTs produced using this strategy encode the state update decision logic which will be leveraged in the following chapter, where the HCT is translated into a state chart.

Chapter 6

Translation II: Tables to State Charts

The HCTs produced using the technique described in chapter 5 are an intermediate representation in our translation strategy. They illustrate the decision logic of the system as a whole, but the logic is not related to state the way it is for state charts, i.e., through modes. This section explains how HCTs are augmented with modes to form STTs, and finally state charts.

6.1 Introducing Modes

The STTs described in Section 2.4.2 have obvious similarities to state charts, but they can be viewed as HCTs with syntactic sugar for modes. In Section 2.3.6, the nodes in extended state transition graphs (or state charts) were described as being only a single component of the state space. This component, whose value is tracked by the nodes in the graph, is modelled as a special state variable *mode* which has values from an enumerated set M (see, e.g., ex-

tended state machines in [Alur 2015]). Each element in M corresponds with a node in the graph. The cells in the first column of STTs (labelled **Source** in Fig. 2.13b) express conditions of the form $mode = Running$ which compare the previous value of the *mode* variable to each element of M, and the last column (**Target**) identifies the updated value for *mode*. Therefore, the state spaces of Mealy machines modelling STTs and state charts have the form $Q = S \times M$, where M is the set of modes, and S contains tuples of the other state variable values.

A HCT produced via the techniques in the previous section describes the update function ud of a Mealy machine $m = (S, s_0, \Sigma, \Lambda, ud)$. We will enhance m with a state variable *mode* to produce a Mealy machine $m^+ = (S \times M, (s_0, mode_0), \Sigma, \Lambda, ud^+)$ whose update function is given by a HCT which matches the format of an STT (i.e. first column has conditions which check the value of *mode*). The introduction of modes here must not change the behaviour of the state machine, and more importantly, it must capture the high-level mode switching logic of the system. The following strategy achieves these goals by introducing modes that match the existing decision logic of the update function.

As mentioned in chapter 3, the existing decision logic modelled by HCTs naturally splits the state space into equivalence classes. For example, in Fig. 5.17 the conditions $counter > 0$ and $counter \leq 0$ divide the state space into two regions, and each region has a distinct set of behaviours associated with it. For example, in the region where $counter > 0$, the system will output a value of *true* for *running*, and decrement the *counter* state variable. In the region where $counter \leq 0$, two behaviours are possible, depending on the value of *start*: if *start* is *true*, then counter will be set to 10; if *start* is $false$, then

counter will be set to 0 (in both cases the system will output a value of $false$).

Any condition in the HCT which depends *only* on the previous state variable values will divide the state space in this way. Conditions such as these will be referred to as *state conditions*. This strategy relies on the existence of one or more state conditions in the HCT. At least one state condition is required for this translation strategy to apply. Any block diagram which produces an HCT with no state conditions in the previous step is not refactorable.

State conditions perform a similar role in HCTs as modes do in STTs, they limit the available behaviours based only on the previous state of the system. To highlight the similarities between HCTs and STTs, the conditions in the HCT can be reordered via the methods described in [Bialy et al. 2015] to move all the state conditions to the left. For example, reordering the conditions in Fig. 5.17 yields the HCT in Fig. 6.1.

		running	*counter'*
$counter > 0$		*true*	$counter - 1$
$counter \leq 0$	*start*	*false*	10
	$\neg start$	*false*	0

Figure 6.1: Reordering the columns in Fig. 5.17

Subsequently, if there are nested state conditions, the hierarchy can be flattened by conjunction. For example, a state condition sc_1 with two sub-conditions $sc_{1.1}$ and $sc_{1.2}$ can be replaced with two conditions: $sc_1 \wedge sc_{1.1}$ and $sc_1 \wedge sc_{1.2}$.

Now, the first row in the HCT contains a distinct state condition. For example, the HCT in Fig. 6.1 has set of state conditions: $\{(counter > 0), (counter \leq 0)\}$. The next step is to build a set of modes M such that each condition is associated with a distinct mode, say $Running \in M$ for the condition $counter > 0$ and *Stopped* for the condition $counter \leq 0$.

Going back to the general case of a Mealy machine $m = (S, s_0, \Sigma, \Lambda, ud)$, a set of modes M has been defined, but we still require definitions for $mode_0$ and ud^+. The introduction of modes should be linked to the state conditions, and for this reason a function $md : S \to M$ is introduced which maps each point in the state space to the mode associated with the state space it satisfies. For example, the HCT in Fig. 6.2 represents the md function for Fig. 6.1 as a HCT.

	mode
$counter > 0$	*Running*
$counter \leq 0$	*Stopped*

Figure 6.2: HCT representing the md function for Fig. 6.1

As an intermediate step, consider a Mealy machine $m^* = (S \times M, (s_0, m_0), \Sigma, \Lambda, ud^*)$, which has a mode variable which is updated to track which state condition holds, but is never used within the state machine. In order to have the mode variable track which state condition holds, it will be updated according to the md function. That is to say, that the update function is defined as $ud^* = (ud \otimes !_M); (\mathrm{id}_\Lambda \otimes (\Delta_S; (\mathrm{id}_S \otimes md)))$ and the mode component of the initial state is defined as $m_0 = md(s_0)$. For the running example, this intermediate update function is shown in Fig. 6.3. The HCT in Fig. 6.3a illustrates how conditions are added which apply each state condition to the updated value of the state variables for that row. In the first two rows, where $counter' = counter - 1$, the previous value of *counter* are required in order to determine which state condition will be satisfied. In contrast, the remaining 4 rows assign a constant value to $counter'$, and so the state condition will always hold or not hold depending on the updated value. Rows with unsatisfiable conditions can be removed, and trivially satisfied conditions can be removed, while keeping the

rest of the row. Simplifying, we get the tabular expression in Fig. 6.3b.

			running	*counter′*	*mode′*
$counter > 0$	$counter - 1 > 0$		*true*	$counter - 1$	*Running*
	$counter - 1 \leq 0$		*true*	$counter - 1$	*Stopped*
$counter \leq 0$	*start*	$10 > 0$	*false*	10	*Running*
		$10 \leq 0$	*false*	10	*Stopped*
	$\neg start$	$0 > 0$	*false*	0	*Running*
		$0 \leq 0$	*false*	0	*Stopped*

(a) Result of composition

		running	*counter′*	*mode′*
$counter > 0$	$counter - 1 > 0$	*true*	$counter - 1$	*Running*
	$counter - 1 \leq 0$	*true*	$counter - 1$	*Stopped*
$counter \leq 0$	*start*	*false*	10	*Running*
	$\neg start$	*false*	0	*Stopped*

(b) Simplified

Figure 6.3: HCTs representing ud^*

The enhanced update function trivially preserves the original behaviour by ignoring the value of *mode*, but updates *mode′* to maintain the invariant that $mode = md(s)$. For this reason, the enhanced Mealy machine operates within a subset of the state space $S \times M$ where the aforementioned invariant holds. Therefore, the truth value of any state condition can be deduced from the value of the mode variable (e.g. $(counter > 0) \Leftrightarrow (mode = Running)$). Thus, replacing the state conditions with the corresponding modes in the HCT representation of ud^* does not modify its behaviour. Replacing the state conditions in the HCT associated with ud^* transforms it into a new HCT (Fig. 6.4a) describing an update function ud^+. This HCT is already structured like an STT. Applying the syntactic sugar of STTs yields the table in Fig. 6.4b.

6.2 Behaviour Preservation

To show that the behaviour of the state machine does not change this section will show that m is bisimilar to m^*, and that m^* is bisimilar to m^+. Each

		running	*counter′*	*mode′*
mode = *Running*	$counter - 1 > 0$	*true*	$counter - 1$	*Running*
	$counter - 1 \leq 0$	*true*	$counter - 1$	*Stopped*
mode = *Stopped*	*start*	*false*	10	*Running*
	$\neg start$	*false*	0	*Stopped*

(a) HCT

Source	**Condition**	*running*	*counter′*	**Target**
Running	$counter - 1 > 0$	*true*	$counter - 1$	*Running*
	$counter - 1 \leq 0$	*true*	$counter - 1$	*Stopped*
Stopped	*start*	*false*	10	*Running*
	$\neg start$	*false*	0	*Stopped*

(b) STT

Figure 6.4: Tables representing ud^+

equivalence is witnessed by a bisimulation relation which relates to the intuitive justification for equivalence given in Section 6.1.

6.2.1 Bisimilarity of m and m^*

Since the Mealy machine $m^* = (S \times M, (s_0, md(s_0)), \Sigma, \Lambda, ud^*)$ was created by introducing a mode variable to $m = (S, s_0, \Sigma, \Lambda, ud)$ that is discarded, it can be shown to be bisimilar via the bisimulation relation that discards the mode component of the state. That is to say that the bisimilation relation is given by the function $\mathrm{id}_S \otimes !_M$.

The following shows that

$$ud^*; (\mathrm{id}_\Lambda \otimes \mathrm{id}_S \otimes !_M) = (\mathrm{id}_\Sigma \otimes \mathrm{id}_S \otimes !_M); ud$$

$$
\begin{aligned}
& ud^*; (\mathrm{id}_\Lambda \otimes \mathrm{id}_S \otimes !_M) \\
&= (ud \otimes !_M); (\mathrm{id}_\Lambda \otimes (\Delta_S; (\mathrm{id}_S \otimes md))); (\mathrm{id}_\Lambda \otimes (\mathrm{id}_S \otimes !_M)) && \text{definition of } ud^* \\
&= (ud \otimes !_M); ((\mathrm{id}_\Lambda; \mathrm{id}_\Lambda) \otimes (\Delta_S; (\mathrm{id}_S \otimes md)); (\mathrm{id}_S \otimes !_M)) && \text{comp. pres. } (\otimes) \\
&= (ud \otimes !_M); ((\mathrm{id}_\Lambda; \mathrm{id}_\Lambda) \otimes (\Delta_S; ((\mathrm{id}_S; \mathrm{id}_S) \otimes (md; !_M))) && \text{comp. pres. } (\otimes) \\
&= (ud \otimes !_M); (\mathrm{id}_\Lambda \otimes (\Delta_S; (\mathrm{id}_S \otimes (md; !_M))) && \text{identity laws} \\
&= (ud \otimes !_M); (\mathrm{id}_\Lambda \otimes (\Delta_S; (\mathrm{id}_S \otimes !_S)) && \text{naturality } (!) \\
&= (ud \otimes !_M); (\mathrm{id}_\Lambda \otimes \mathrm{id}_S) && \text{counit } (\Delta, !) \\
&= (ud \otimes !_M); \mathrm{id}_{\Lambda \otimes S} && \text{identity presevation} \\
&= (\mathrm{id}_{\Sigma \otimes S}; ud) \otimes (!_M; \mathrm{id}_{\mathbb{1}}) && \text{identity laws} \\
&= (\mathrm{id}_{\Sigma \otimes S} \otimes !_M); (ud \otimes \mathrm{id}_{\mathbb{1}}) && \text{comp. pres. } (\otimes) \\
&= (\mathrm{id}_\Sigma \otimes \mathrm{id}_S \otimes !_M); (ud \otimes \mathrm{id}_{\mathbb{1}}) && \text{identity presevation} \\
&= (\mathrm{id}_\Sigma \otimes \mathrm{id}_S \otimes !_M); ud && \text{naturality } (\rho)
\end{aligned}
$$

For the initial state condition, note that applying $\mathrm{id}_S \otimes !_M$ to $(s_0, md(s_0))$ yields (s_0).

6.2.2 Bisimilarity of m^* and m^+

The Mealy machine $m^+ = (S \times M, (s_0, md(s_0)), \Sigma, \Lambda, ud^+)$ is of course very similar to $m^* = (S \times M, (s_0, md(s_0)), \Sigma, \Lambda, ud^*)$. Because these two are both bimisimilar to m, they operate in a subset of the state space which maps to S. This subset is given by $\{(s, md(s)) | s \in S\}$. Since the modification that was

made to m^* to transform it relied on the fact that m^* and m^+ both operate within this state space, they can be shown to be bisimilar through the use of a bisimulation relation which enforces this property. In this case the bisimulation relation is given by $\{(s, md(s)) \leftrightarrow (s, md(s)) | s \in S\}$. Each element in S has a corresponding link in this relation, and both sides of the link are obtained by applying $\Delta_S; (\mathrm{id}_S \otimes md)$ to s.

In order to show that m^* and m^+ are bisimilar, consider the following equivalence:

$$(\mathrm{id}_\Sigma \otimes (\Delta_S; (\mathrm{id}_S \otimes md))); ud^* = (\mathrm{id}_\Sigma \otimes (\Delta_S; (\mathrm{id}_S \otimes md))); ud^+$$

Consider what each side of this equivalence would look like as an HCT. The HCT for $(\mathrm{id}_\Sigma \otimes (\Delta_S; (\mathrm{id}_S \otimes md)))$ is much the same as the HCT for md, in that it has the state conditions, and an output variable *mode* which is assigned in the same manner as before, the only difference is that it also outputs the state and input variables, unmodified. For the running example, this HCT is shown in Fig. 6.5a. Composing this HCT with the HCT for ud^* would yield a redundant set of conditions, since the leftmost column already contains the state conditions, and since the *mode* variable is unused. Because the conditions introduced in the first column are redundant, the columns in the second column are either trivially satisfied based on the identical condition in the adjacent cell of the first column; or it is impossible, since it is disjoint with the condition in the adjacent cell of the first column. Composing Fig. 6.5a with the HCT for ud^+ would also add a set of state conditions in the first column, and while the conditions in the second column are not identical to the conditions in the first column, they are trivially satisfied or impossible just the same as they are in

the first case. This is because when substituting the *mode* value from the HCT representing $(\mathrm{id}_\Sigma \otimes (\Delta_S; (\mathrm{id}_S \otimes md)))$, into the second column, the mode value corresponding to the state condition in the adjacent cell of the first column will only satisfy the mode-comparison condition which was introduced to replace that state condition. As is illustrated by Fig. 6.5b and Fig. 6.5c, composing the HCT for $(\mathrm{id}_\Sigma \otimes (\Delta_S; (\mathrm{id}_S \otimes md)))$ with the HCTs for ud^* and ud^+ yield HCTs where all cells in the column containing the only difference between them can be removed, since that column contains conditions that will always evaluate to either true or false.

	start	*counter*	*mode*
$counter > 0$	*start*	*counter*	*Running*
$counter \leq 0$	*start*	*counter*	*Stopped*

(a) HCT for $\mathrm{id}_\Sigma \otimes (\Delta_S; (\mathrm{id}_S \otimes md))$

			running	*counter'*	*mode'*
$counter > 0$	$counter > 0$	$counter - 1 > 0$	*true*	$counter - 1$	*Running*
		$counter - 1 \leq 0$	*true*	$counter - 1$	*Stopped*
	$counter \leq 0$	*start*	*false*	10	*Running*
		$\neg start$	*false*	0	*Stopped*
$counter \leq 0$	$counter > 0$	$counter - 1 > 0$	*true*	$counter - 1$	*Running*
		$counter - 1 \leq 0$	*true*	$counter - 1$	*Stopped*
	$counter \leq 0$	*start*	*false*	10	*Running*
		$\neg start$	*false*	0	*Stopped*

(b) HCT for $(\mathrm{id}_\Sigma \otimes (\Delta_S; (\mathrm{id}_S \otimes md))); ud^*$

			running	*counter'*	*mode'*
$counter > 0$	$Running = Running$	$counter - 1 > 0$	*true*	$counter - 1$	*Running*
		$counter - 1 \leq 0$	*true*	$counter - 1$	*Stopped*
	$Running = Stopped$	*start*	*false*	10	*Running*
		$\neg start$	*false*	0	*Stopped*
$counter \leq 0$	$Stopped = Running$	$counter - 1 > 0$	*true*	$counter - 1$	*Running*
		$counter - 1 \leq 0$	*true*	$counter - 1$	*Stopped*
	$Stopped = Stopped$	*start*	*false*	10	*Running*
		$\neg start$	*false*	0	*Stopped*

(c) HCT for $(\mathrm{id}_\Sigma \otimes (\Delta_S; (\mathrm{id}_S \otimes md))); ud^+$

Figure 6.5: HCTs showing equivalence after applying $(\mathrm{id}_\Sigma \otimes (\Delta_S; (\mathrm{id}_S \otimes md)))$

Composing these two update functions with the function $\mathrm{id}_\Sigma \otimes (\Delta_S; (\mathrm{id}_S \otimes md))$ yields the same function. This shows that updating either Mealy machine

from states that are related by the bisimulation relation yields equal outputs. To show that they produce states that are related by the bisimulation relation, consider the fact that the bisimulation relation is given by $\{(s, md(s)) \leftrightarrow (s, md(s)) | s \in S\}$. For two states to be related, they must satisfy both these conditions: they must be equal; they must be of the form $(s, md(s))$. The first condition can shown in the same way as it is shown that the outputs are equal. The second condition can be shown by the definition of ud^*, where the value for the *mode* variable is obtained in each time step by applying md to s.

Lastly, the initial states of ud^* and ud^+ are related by the bisimilation for the same reason; they are equal, and defined to be of the form $(s, md(s))$

6.3 Converting to State Charts & Simplifying

The state chart in Fig. 6.6 can be generated from the STT in Fig. 6.4b by creating a state for each state in the first column, and a transition for each row. Each transition originates from the associated mode in the **Source** column, and leads to the mode in the associated **Target** column. Furthermore, the transition is guarded by the condition from the **Condition** column, and has actions which assign to each output and state variable the associated value from the row of the STT. In addition, the initial state is used to generate a default transition which targets the initial mode, and assigns to each state variable its initial value. As discussed in chapter 2, STTs and STGs represent the same information, only in a slightly different format. As the state chart in Fig. 6.6 only uses a subset of state chart syntax that can be modelled as a state transition graph, expressing the data in an STT as a state chart will represent the same Mealy machine as was originally described.

Thus, a state chart, behaviourally equivalent to the initial block diagram has been produced. Furthermore, as is clear in the example in Fig. 6.6, this state chart captures the high level mode switching behaviour of the system.

In addition, there are often ways that these state charts can be simplified by moving common actions from transitions to entry or exit actions of modes, or by removing looping transitions and performing the corresponding actions as during actions. Performing these simplifications on the state chart from Fig. 6.6 result in the state chart from Fig. 6.7. These simplifications help highlight the effectiveness of the refactoring strategy, as capturing commonalities between transitions originating from the same state is a feature of this methodology. However, these simplifications are generally easy to perform by inspection, and so no prescriptive methodology has been developed for this simplification step.

Furthermore, as is illustrated in the example provided in chapter 3, where the block diagram in Fig. 3.1 is translated into the state chart shown in Fig. 3.4, the mode variable may obviate the need for one or more state variables. In this case, the value of those state variables will not be referenced in the final state chart, and so they can be pruned. This may happen if a Boolean state variable generates a state condition; knowing the value of *mode* can be sufficient to deduce the value of the original state variable. It may also be the case that a state variable from the block diagram stores more detailed information than necessary, and knowing the mode is sufficient for the state chart to act. However, this is not always the case, such as in the example given above, where it is crucial that the new state variable *mode* is tracked in addition to the existing variable *counter*. The *mode* variable tracks the high level system state, but the *counter* variable is still important for tracking

the detailed system state. This activity can be performed by inspection, and again, no prescriptive methodology has been developed.

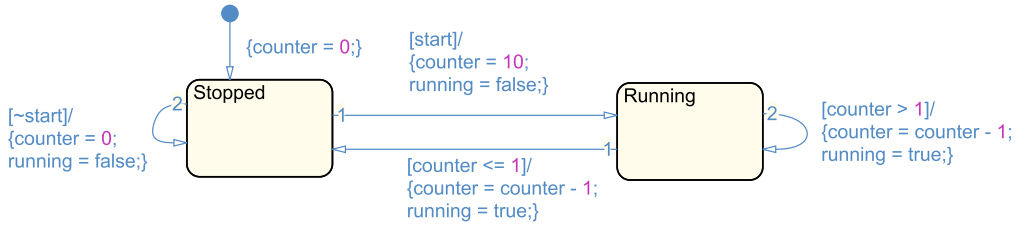


Figure 6.6: State Chart Equivalent to STT in Fig. 6.4b

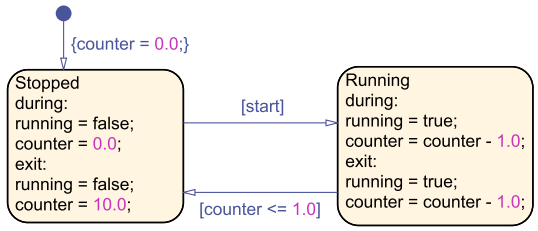


Figure 6.7: Simplified State Chart

The state charts in Fig. 6.6 and Fig. 6.7 illustrate a key advantage of this methodology over manual refactoring: they produce a model which is behaviourally equivalent to the original block diagram. Without following a structured refactoring approach, it is easy to incidentally change small details about how the model works. For example, in Fig. 6.6, it is clear that the output *running* does not change in the same time step that the state chart transitions from one state to the next, rather it outputs the value corresponding to the previous state. While this is the behaviour of the original model, it is easy to imagine that refactoring based only on an intuitive understanding of the model may miss this minor detail, producing a state chart which immediately updates the *running* variable upon changing modes.

6.4 Summary

In this section, the HCTs produced via the strategy in the previous chapter are analysed to identify the high-level modes of the block diagram's behaviours. Using these modes, a state chart is produced which is behaviourally equivalent to the HCT, and by extension, the original block diagram.

Chapter 7

Conclusion

This thesis presents a methodology for refactoring Simulink block diagrams into Stateflow state charts, using tabular expressions to represent their respective update functions. The theoretical foundation of the translation is provided by Mealy machines, and a categorical framework was developed to model their interactions in block diagrams. To the best of the author's knowledge, this is the first methodology for translating block diagrams to state transition systems which capture high-level mode switching logic. This work is applicable to industrial development, where it can assist refactoring efforts aimed at improving software maintainability and compliance with modelling guidelines.

The translation methodology succeeds at improving the readability of various small to medium sized block diagrams, such as the ones presented in this thesis. While factors such as readability can be subjective, this evaluation has been affirmed by experienced model developers, and aligns with the MAB modelling guidelines. Furthermore, the identification methodology matches many components of industrial models which had previously been selected for refactoring based on the developer's intuition.

The success of this methodology for small to medium sized models is promising, and serves as a foundation for future work in expanding the methodology to refactor larger models. Complex industrial models often contain multiple state machines interacting amongst themselves and combining with stateless condition logic. Elegantly representing these industrial models in Stateflow requires the use of more sophisticated capabilities of Stateflow state charts, such as hierarchical and parallel modes, or branching transitions. The STTs presented in [Singh et al. 2015] already model some of these capabilities, and it is likely that there are further analogues between tabular expressions and state charts that can be leveraged. For example, nested state conditions divide the state space into a hierarchy of equivalence classes, and it is possible that this can be leveraged to produce a state chart with a hierarchy of states.

Additionally, methods for simplifying tabular expressions such as those presented in [Bialy et al. 2015] may be leveraged to simplify the intermediate tabular expressions. Because each row in the tabular expression yields a distinct transition in the state chart, reducing the number of rows in the table can tangibly simplify the final state chart. Alternatively, the method used to produce HCTs for individual blocks may be improved to use context to generate fewer rows in the first place. For example, under the current strategy many logical blocks produce tabular expressions with multiple rows, outputting constant $true$ or $false$ values. These tabular expressions use logical expressions in the conditions to determine which value to return. This strategy can be augmented to sometimes produce HCTs which evaluate the same logical expressions to produce the output value on a single row, similar to how arithmetic expressions are represented. Generating conditions such as these is crucial when those conditions become state conditions in the composed HCT,

however they can lead to redundant transitions in the state chart when they become conditions that do not rely on state. Future work may explore how data flow in the block diagram can be used to inform the generation of HCTs to limit the generation of conditions that do not become state conditions.

Lastly, future work may be spent on situating this methodology within the broader work of model management by using declarative specification techniques rather than imperative ones used here. For example, using transformation languages such as ATL ([Jouault et al. 2008]) and ETL ([Kolovos, Paige, and Polack 2008]) to specify the translation strategy via a set of rules, or using traceability links between meta-models to specify the translation ([Diskin, Gómez, and Cabot 2017]).

Bibliography

Abadi, Moria and Yishai A Feldman (2009). "Refactoring of statecharts". In: *International Conference on Next Generation Information Technologies and Systems*. Springer, pp. 50–62 (cit. on p. 5).

Abraham, Ruth F (1997). *Evaluating generalized tabular expressions in software documentation*. Vol. 346. Citeseer (cit. on p. 7).

Agrawal, Aditya, Gyula Simon, and Gabor Karsai (2004). "Semantic translation of Simulink/Stateflow models to hybrid automata using graph transformations". In: *Electronic Notes in Theoretical Computer Science* 109, pp. 43–56 (cit. on p. 4).

Alur, Rajeev (2015). *Principles of cyber-physical systems*. MIT Press (cit. on p. 96).

Babcsanyi, Istvan (2000). "Equivalence of mealy and moore automata". In: *Acta Cybernetica* 14.4, pp. 541–552 (cit. on p. 26).

Baez, John C and Jason Erbele (2015). "Categories in control". In: *Theory and Applications of Categories* 30.24, pp. 836–881 (cit. on p. 5).

Bialy, M. et al. (2015). "A Methodology for the Simplification of Tabular Designs in Model-Based Development". In: *Proceedings of the 3rd FME Workshop on Formal Methods in Software Engineering (FormaliSE)*. Florence, Italy: IEEE Press, pp. 47–53 (cit. on pp. 33, 97, 109).

Bouissou, Olivier and Alexandre Chapoutot (2012). "An operational semantics for Simulink's simulation engine". In: *ACM SIGPLAN Notices*. Vol. 47. 5. ACM, pp. 129–138 (cit. on p. 6).

Coecke, Bob and Aleks Kissinger (2017). *Picturing quantum processes*. Cambridge University Press (cit. on pp. 5, 33).

Diskin, Z., A. Gómez, and Jordi Cabot (2017). "Traceability Mappings as a Fundamental Instrument in Model Transformations". In: *FASE* (cit. on p. 110).

Dragomir, Iulia, Viorel Preoteasa, and Stavros Tripakis (2015). "Translating hierarchical block diagrams into composite predicate transformers". In: *arXiv preprint arXiv:1510.04873* (cit. on pp. 4, 6).

Eles, Colin and Mark Lawford (2011). "A Tabular Expression Toolbox for Matlab/Simulink". In: *Proceedings of the 3rd NASA Formal Methods Symposium*. Ed. by Mihaela Bobaru et al. Vol. 6617. LNCS. Springer Berlin Heidelberg, pp. 494–499 (cit. on p. 7).

Goncharov, Sergey and Lutz Schröder (2018). "Guarded Traced Categories". In: *Foundations of Software Science and Computation Structures*. Ed. by Christel Baier and Ugo Dal Lago. Cham: Springer International Publishing, pp. 313–330. ISBN: 978-3-319-89366-2 (cit. on p. 47).

Hamon, Grégoire (2005). "A denotational semantics for stateflow". In: *Proceedings of the 5th ACM international conference on Embedded software*. Citeseer, pp. 164–172 (cit. on p. 6).

Hamon, Grégoire and John Rushby (2007). "An operational semantics for Stateflow". In: *International Journal on Software Tools for Technology Transfer* 9.5-6, pp. 447–456 (cit. on p. 6).

Harbird, Lishan (2011). "Patterns and model transformation tools for designing Contractual State Machines". PhD thesis. University of York (cit. on p. 5).

Heunen, Chris and Jamie Vicary. "Lectures on categorical quantum mechanics (2012)". In: *URL www. cs. ox. ac. uk/files/4551/cqm-notes. pdf* () (cit. on p. 51).

Hofmann, Martin, Benjamin Pierce, and Daniel Wagner (2011). "Symmetric lenses". In: *ACM SIGPLAN Notices* 46.1, pp. 371–384 (cit. on pp. 6, 80, 83).

ISO (2011). *Road vehicles – Functional safety*. Norm (cit. on p. 2).

Janicki, Ryszard and Alan Wassyng (2005). "Tabular expressions and their relational semantics". In: *Fundamenta Informaticae* 67.4, pp. 343–370 (cit. on pp. 6, 7, 32).

Jin, Ying and David Lorge Parnas (2010). "Defining the meaning of tabular mathematical expressions". In: *Science of Computer Programming* 75.11, pp. 980–1000 (cit. on pp. 7, 33).

Jouault, Frédéric et al. (2008). "ATL: A model transformation tool". In: *Science of Computer Programming* 72.1. Special Issue on Second issue of experimental software and toolkits (EST), pp. 31 –39. ISSN: 0167-6423. URL: `http://www.sciencedirect.com/science/article/pii/S0167642308000439` (cit. on p. 110).

Kahn, A. B. (Nov. 1962). "Topological Sorting of Large Networks". In: *Commun. ACM* 5.11, 558–562. ISSN: 0001-0782. URL: `https://doi.org/10.1145/368996.369025` (cit. on p. 92).

Katis, Piergiulio, Nicoletta Sabadini, and Robert FC Walters (1997). "Span (Graph): A categorical algebra of transition systems". In: *International*

Conference on Algebraic Methodology and Software Technology. Springer, pp. 307–321 (cit. on p. 5).

Kolovos, Dimitrios S, Richard F Paige, and Fiona AC Polack (2008). "The epsilon transformation language". In: *International Conference on Theory and Practice of Model Transformations*. Springer, pp. 46–60 (cit. on p. 110).

Kolovos, Dimitris S and Richard F Paige (2017). "The epsilon pattern language". In: *2017 IEEE/ACM 9th International Workshop on Modelling in Software Engineering (MiSE)*. IEEE, pp. 54–60 (cit. on p. 72).

Lee, Edward A and Pravin Varaiya (2011). *Structure and interpretation of signals and systems*. 2nd. LeeVaraiya.org (cit. on p. 29).

Liebrenz, Timm, Paula Herber, and Sabine Glesner (2018). "Deductive verification of hybrid control systems modeled in Simulink with KeYmaera X". In: *International Conference on Formal Engineering Methods*. Springer, pp. 89–105 (cit. on p. 4).

Manamcheri, Karthik et al. (2011). "A step towards verification and synthesis from Simulink/Stateflow models". In: *Proceedings of the 14th International Conference on Hybrid systems: Computation and Control*. ACM, pp. 317–318 (cit. on p. 5).

McSCert (2018). *Reach/Coreach Tool*. `https://www.mathworks.com/matlabcentral/fileexchange/51180-reach-coreach-tool`. [Online; accessed June, 2018] (cit. on p. 72).

— (2019). *Simulink-to-Stateflow*. `https://www.mathworks.com/matlabcentral/fileexchange/70317-simulink-to-stateflow`. [Online; accessed February 2019] (cit. on p. 9).

Mealy, George H (1955). "A method for synthesizing sequential circuits". In: *Bell System Technical Journal* 34.5, pp. 1045–1079 (cit. on pp. 5, 57, 65, 66).

Minopoli, Stefano and Goran Frehse (2016). *SL2SX Translator: From Simulink to SpaceEx Models*. URL: `http://www-verimag.imag.fr/~minopoli/SL2SX.pdf` (cit. on pp. 4, 5).

Mohrenschildt, Martin von (2000). "Algebraic composition of function tables". In: *Formal aspects of computing* 12.1, pp. 41–51 (cit. on pp. 7, 77).

Parnas, David Lorge (1992). *Tabular representation of relations*. Tech. rep. McMaster University (cit. on p. 8).

Sfyrla, Vassiliki et al. (2010). "Compositional translation of Simulink models into synchronous bip". In: *Industrial Embedded Systems (SIES), 2010 International Symposium on*. IEEE, pp. 217–220 (cit. on p. 4).

Singh, Neeraj Kumar et al. (2015). "Stateflow to Tabular Expressions". In: *Proceedings of the Sixth International Symposium on Information and Communication Technology (SolCT)*. ACM, p. 47 (cit. on pp. 7, 32, 65, 109).

The MathWorks (2012). *MathWorks Automotive Advisory Board (MAAB): Control Algorithm Modeling Guidelines Using MATLAB, Simulink, and Stateflow, Version 3.0*. URL: `https://www.mathworks.com/solutions/mab-guidelines/previous-versions.html` (cit. on p. 55).

— (2020a). *MathWorks Advisory Board (MAB): Control Algorithm Modeling Guidelines Using MATLAB, Simulink, and Stateflow, Version 5.0*. URL: `https://www.mathworks.com/solutions/mab-guidelines/` (cit. on p. 2).

— (2020b). *Simulink Check*. `https://www.mathworks.com/products/simulink-check.html`. [Online; accessed December, 2020] (cit. on p. 2).

Tripakis, Stavros et al. (2005). "Translating discrete-time Simulink to Lustre". In: *ACM Transactions on Embedded Computing Systems (TECS)* 4.4, pp. 779–818 (cit. on p. 4).

Willems, Jan C (1989). "Models for dynamics". In: *Dynamics reported*. Springer, pp. 171–269 (cit. on p. 21).

Wynn-Williams, Stephen et al. (2019). "SL2SF: Refactoring Simulink to Stateflow". In: *International Conference on Fundamental Approaches to Software Engineering*. Springer, Cham, pp. 264–281 (cit. on p. 9).

Zhan, Naijun, Shuling Wang, and Hengjun Zhao (2017). *Formal Verification of Simulink/Stateflow Diagrams*. Springer (cit. on p. 4).

Zhou, Changyan and Ratnesh Kumar (2012). "Semantic translation of Simulink diagrams to input/output extended finite automata". In: *Discrete Event Dynamic Systems* 22.2, pp. 223–247 (cit. on pp. 4, 5).