SL2SF: Refactoring Simulink to Stateflow

Stephen Wynn-Williams[1], Zinovy Diskin[1], Vera Pantelic[1], Mark Lawford[1], Gehan Selim[1], Curtis Milo[1], Moustapha Diab[2], and Feisel Weslati[2]

[1] McMaster Centre for Software Certification
McMaster University, Hamilton ON, Canada
{wynnwisj, diskinz, pantelv, lawford, selimg, milocj}@mcmaster.ca
[2] FCA US LLC, Auburn Hills MI, USA
moustapha.diab@external.fcagroup.com,faz.weslati@fcagroup.com

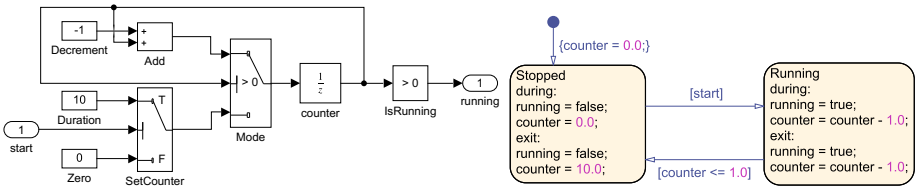
Abstract. In the Matlab Simulink environment, systems can be modelled using Simulink block diagrams and Stateflow state charts. While stateful logic is more naturally modelled using Stateflow, in practice complex block diagrams are often used instead, resulting in models that are hard to understand and maintain. In order to improve the maintainability and understandability of large industrial models, this paper presents a strategy for refactoring Simulink block diagrams implementing stateful logic into functionally equivalent Stateflow state charts that more naturally represent the intended behaviour. To bridge the gap between the syntax of block diagrams and state charts, Mealy machines represented by tabular expressions are used as an intermediate representation. The compositional language of block diagrams is used to combine tables modelling individual blocks into a table for the entire block diagram which describes the high level state machine encoded in the Simulink subsystem. A prototype tool that performs the translation from Simulink to Stateflow automatically is discussed.

Keywords: Simulink · Stateflow · Refactoring · Mealy Machines · Tabular Expressions · Monoidal Categories

1 Introduction

The adoption of Model-Based Design in the development of embedded control systems across industries has led to the wide use of Matlab/Simulink/Stateflow as a supporting environment. 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. For example, the MathWorks Automotive Advisory Board (MAAB) guidelines [25] 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 executes periodically to update its state and outputs. When the block diagram in Fig. 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. 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

Fig. 1: Model of a Timer in Simulink and Stateflow.

The Simulink and Stateflow models shown in Fig. 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. 1a, it is difficult to understand how this model achieves the behaviour while the state chart in Fig. 1b clearly captures the system's modes and the conditions triggering mode changes.

Our industrial experience has identified the need to refactor Simulink block diagrams to Stateflow state charts for easier comprehension and maintenance. More precisely, 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.

This paper presents an approach to translate block diagrams into behaviourally equivalent state charts. The approach converts individual blocks into tabular expressions [21] 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 identified by reconfiguring the combined tables into a form similar to state charts. Behavioural equivalence is established by giving semantics to block diagrams, state charts, and the intermediate tables as Mealy machines. The paper's main contributions are: (i) A method for translating Simulink block diagrams to Stateflow state charts via tabular expressions. (ii) A categorical framework for composing Mealy machines by combining their update functions as the basis of the translation. (iii) A prototype tool implementing the translation from Simulink to Stateflow.

This paper is organized as follows. Section 2 describes how we model systems and our categorical framework for combining them. Section 3 illustrates the translation method with a simple example. Section 4 describes the application of the categorical framework to convert block diagrams to tabular expressions. Section 5 explains how tabular expressions are converted to state charts. Section 6 describes the prototype tool. Related work is covered in Section 7 and the paper concludes with Section 8.

2 Background: Modelling Systems & Their Combinations

This section describes the formalisms underlying the proposed translation approach: Mealy machines, tabular expressions, and monoidal categories.

2.1 Mealy Machines: Modelling Stateful Systems

To preserve behaviour, the semantics of both block diagrams and state charts are modeled 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 (the *state space*), $s_0 \in S$ (the *initial state*), Σ is a set of input values (the *input alphabet*), Λ is a set of output values (the *output alphabet*), and $ud : \Sigma \times S \to \Lambda \times S$ is a function (the *update function*) which computes the current output and next state from the current input and current state.

For example, the unit delay $\frac{1}{z}$ block labelled *counter* in Fig. 1a can be modelled as the Mealy machine $delay = (\mathbb{R}, 0, \mathbb{R}, \mathbb{R}, shift)$. The block has an input variable (port) i, an output variable (port) o, and an internal state variable $counter$, where $i, o, counter \in \mathbb{R}$. When the block updates, it outputs the current state value $o = counter$, and updates the state to store the current input value $counter' = i$, i.e. $(o, counter') = shift(i, counter)$, where $shift : \mathbb{R}^2 \to \mathbb{R}^2$ is defined as $shift(i, counter) = (counter, i)$.

While Simulink has no formal semantics, our use of Mealy machines to model their behaviours is consistent with the informal semantics described in Chapter 3 of the Simulink User Guide [26].

2.2 Tabular Expressions: Representing Conditional Behaviours

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 [28]; and *state transition tables* (STTs), which specialize HCTs to describe state charts similarly to the ones presented in [24].

		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

Fig. 2: Intermediate Representations

An HCT is represented in Fig. 2a. It is a tabular representation of the update function of a Mealy machine which models the block diagram from Fig. 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.

The second tabular representation, STTs, are also used to represent the update function of Mealy machines. Their special format closely matches the state charts they model. For example, the STT in Fig. 2b represents the state chart in Fig. 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 they leave. 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 [10]. The structure of tables can be rearranged without changing the function they describe, e.g., conditions can be reordered as in [4]; 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.

2.3 Categorical Framework: Combining Systems

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 [5] (i.e. cascading blocks in sequence, placing blocks in parallel, and feeding outputs of blocks back to their inputs).

Monoidal categories describe data flow in an abstract setting where blocks are called *morphisms*. Simple data flow constructs are described as operations on morphisms, which can be visualized using block diagrams called *string diagrams* [5,22]. In this section, we discuss the wiring constructs in the concrete setting of the category **Set**, where morphisms are functions from an input set of tuples to an output set of tuples (called the *domain*/*codomain objects* of the morphism).

A fragment of the block diagram from Fig. 1a can be used to illustrate the idea behind the basic data flow operations. The string diagram in Fig. 3 describes a function that is broken down into sub-functions combined via two operations: sequential combination (denoted ";") and parallel combination (denoted "⊗"). The fragment describes a function g from $\mathbb{R} \times \mathbb{B}$ to $\mathbb{R}$. Each wire extending from

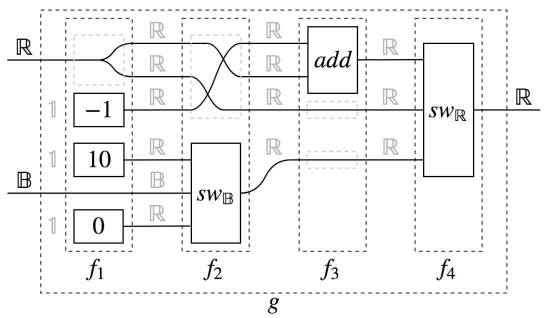


Fig. 3: Functional Fragment of Timer Example

the left/right of the large compound function indicates an input/output value, respectively. The wire is labelled with the set from which the value comes. If there are multiple wires, the domain or codomain of the function is given as the Cartesian product of those sets. In monoidal categories, the Cartesian product is generalized as an operation called the *monoidal product on objects*.

The function g is composed of a sequence of sub-functions, $g = f_1; f_2; f_3; f_4$. The sub-functions (except for f_4) consist of functions composed in parallel with wires and other functions. The wiring "data routing functions" are then defined as follows: a normal wire is the *identity* function $\mathrm{id}_X = \{(x) \mapsto (x)\}$; wires crossing over each other define the *braiding* function $\mathrm{Br}_{A,B} = \{(a,b) \mapsto (b,a)\}$; and branching wires are called the *diagonal* function $\Delta_X = \{(x) \mapsto (x,x)\}$. The functions are indexed with the set(s) over which they are defined. Morphisms like these functions have special status in monoidal categories and must satisfy some axioms to verify that they "act like wiring" in the host category.

Sub-function f_3 can now be described as $f_3 = add \otimes \mathrm{id}_{\mathbb{R}} \otimes \mathrm{id}_{\mathbb{R}}$. Functions combined in parallel have domains/codomains which are the Cartesian products

of the domain/codomain of the component functions. The parallel combination uses each component function independently to calculate each component of the output. For example, taking $add = \{(x_1, x_2) \mapsto (x_1 + x_2)\}$, the function $add \otimes \mathrm{id}_{\mathbb{R}} \otimes \mathrm{id}_{\mathbb{R}}$ is given by $\{(x_1, x_2, x_3, x_4) \mapsto (x_1 + x_2, x_3, x_4)\}$. In monoidal categories this operation is generalized as the *monoidal product on morphisms*, where the domain/codomain of a product morphism is given by the monoidal product of the domain/codomain objects of the component morphisms. It is notable that we can also describe sub-function f_3 as $f_3 = add \otimes \mathrm{id}_{\mathbb{R}^2}$, where the two wires are treated as one function. This is useful, for example, when describing the sub-function f_2 as $f_2 = Br_{\mathbb{R}^2,\mathbb{R}} \otimes sw_{\mathbb{B}}$.

Describing f_1 requires modelling constant blocks as functions. Therefore, constants are described as functions with inputs from the singleton set $\mathbb{1} = \{()\}$, and we draw functions with domain/codomain $\mathbb{1}$ as blocks with no wires extending from the left/right side, respectively. Functions modelling constant blocks, $[k] = \{() \mapsto (k)\}$, always take the empty tuple as input, and always produce the same value k as output. The function f_1 can now be described as $f_1 = \Delta_{\mathbb{R}} \otimes [-1] \otimes [10] \otimes \mathrm{id}_{\mathbb{B}} \otimes [0]$. Objects like $\mathbb{1}$ have special status in monoidal categories and are called the *monoidal unit*. Taking their monoidal product with any other object X yields the same object X. Intuitively, this means that concatenating any tuple $(x_1, .., x_n)$ with the empty tuple $()$ does nothing. This explains why the product of the domains of the functions in f_1 is the set $\mathbb{R} \times \mathbb{1} \times \mathbb{1} \times \mathbb{B} \times \mathbb{1}$, but the domain of f_1 is described as $\mathbb{R} \times \mathbb{B}$—the former simplifies to the latter.

We now describe the entire function g in terms of simple data flow:

$$g = (\Delta_{\mathbb{R}} \otimes [-1] \otimes [10] \otimes \mathrm{id}_{\mathbb{B}} \otimes [0]); (\mathrm{Br}_{\mathbb{R}^2,\mathbb{R}} \otimes sw_{\mathbb{B}}); (add \otimes \mathrm{id}_{\mathbb{R}} \otimes \mathrm{id}_{\mathbb{R}}); sw_{\mathbb{R}}$$

However, this example does not contain feedback loops. Loops are obtained when inputs and outputs of a function are connected by some common wire(s), such as the wire connecting the first input and first output of the inner box in Fig. 4a. Adding looping wires to a function $f : X \times A \to X \times B$ yields a new function $f^* : A \to B$ (e.g., the outer box in Fig. 4a) where $f^*(a) = b$ if there exists a unique $x \in X$ such that $f(x, a) = (x, b)$. When such an x exists for each $a \in A$, the loop configuration is considered *well-formed*. Following [11], we encode the addition of such loops with a *trace* operation: $\mathrm{Tr}^X_{A,B}(f) = f^*$.

For example, consider the function $f = \{(x, y) \mapsto (x + x, x + y)\}$. In the function $\mathrm{Tr}^{\mathbb{R}}_{\mathbb{R},\mathbb{R}}(f)$ the trace applies the constraint that the first input is equal to the first output (i.e. $x = x + x$) to which there is a unique solution: $x = 0$. Given any $y \in \mathbb{R}$, $f(0, y) = (0, y)$, therefore $\mathrm{Tr}^{\mathbb{R}}_{\mathbb{R},\mathbb{R}}(f) = \{(y) \mapsto (y)\}$. This approach uses *fixed point equations* to specify traces, which is generalized by the approach from [8]. Since these fixed point equations are not guaranteed to have a unique solution, the trace operation is *partial*—it is only defined for loop configurations that are well-formed. Partial traces have been described in [15], and the guarded structure introduced in [7] compositionally describes which feedback configurations are valid. For the loops to "act like wiring", certain axioms must be satisfied, e.g., the *yanking* axiom (as shown in Fig. 4b) states that $\mathrm{Tr}^X_{X,X}(\mathrm{Br}_{X,X}) = \mathrm{id}_X$ for any set X.

(a) $\mathrm{Tr}^X_{A,B}(f)$ (b) Yanking: $\mathrm{Tr}^X_{X,X}(\mathrm{Br}_{X,X}) = \mathrm{id}_X$

Fig. 4: String diagrams For traced categories

3 Translation Strategy

The translation strategy is composed of three steps. This section illustrates these steps by considering the example from Fig. 1.

First, the decision logic implemented by the block diagram is encoded as the HCT in Fig. 8a. This step is described in Section 4. In the second step, the representation is simplified as, depending on the value of *counter*, only some rows of the table can be valid. By associating a certain range of state variable values with a mode of operation, we simplify the representation by considering only the conditions which are possible. This allows us to leverage the conditions from HCTs to determine the modes of operation by rearranging HCTs into equivalent STTs such as Fig. 2b. The final step trivially rearranges the information from STTs into a state chart by creating a transition for each row. The conversion from HCTs to STTs to state charts is described in Section 5, and possible simplifications to the resulting state chart are discussed.

Even with such a simple example, the importance of automated refactoring becomes apparent. If the model were to be refactored manually, a state chart that is not equivalent to the block diagram could be created unintentionally. For example, one can manually produce a state chart that transitions out of the *Running* mode when *counter* is zero, rather than one.

4 Block Diagrams to HCTs: Mealy Composition

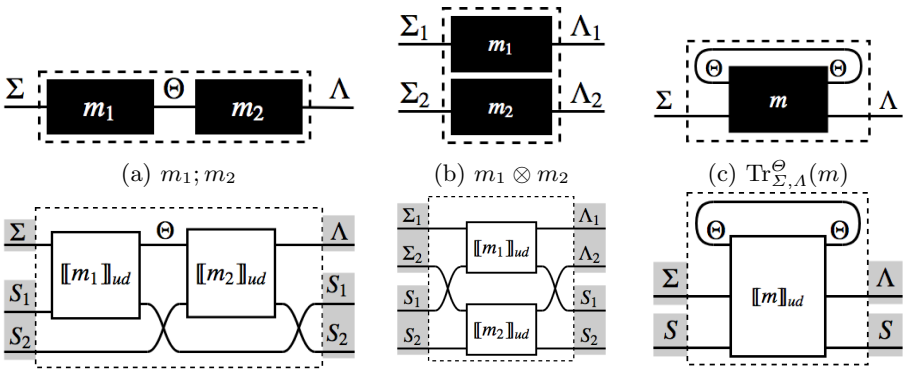
The first step of the translation strategy is to model the entire block diagram as a Mealy machine whose update function is represented as a HCT. To achieve this, Simulink block diagrams are modelled in a category **Mealy**, where morphisms (i.e. blocks) are Mealy machines, not functions. We then show how the update functions of composite Mealy machines built from the operations described in Section 2.3 can be built from the update functions of the component Mealy machines using the same operations on functions. Then, the predefined update functions of individual blocks can be represented using HCTs and combined according to the functional combinations derived from the block diagram.

4.1 Mealy Machines & Their Combinations via Functions

In this section, we consider a category **Mealy** whose objects are sets, and whose morphisms $m : \Sigma \to \Lambda$ are Mealy machines with input alphabet Σ, and output

alphabet Λ. Composition of morphisms is given by the usual definition of cascade composition of Mealy machines [13]. We also introduce a monoidal product, giving the category a monoidal structure. It is defined on objects as the Cartesian product of sets, and on morphisms as the parallel composition of Mealy machines. The unit of the monoidal product is the same as for sets, the set containing one element: $\mathbb{1}$. Considering equality of morphisms up to bisimilarity results in a structure similar to the one used in [9] to describe symmetric lenses—according to [9], this structure forms a (symmetric) monoidal category.

While the cascade/parallel composition of Mealy machines is well understood (see, e.g. [13]), we introduce a definition for the update functions of the composed machines which wires together the update functions of the individual machines. Because string diagrams are used to represent both Mealy machines and their update functions, let us introduce some graphical notation to differentiate them. For Mealy machines, the string diagrams use black boxes to denote component Mealy machines (e.g. Fig. 5a). The update function ud of a Mealy machine m can be expressed using the projection mapping $[\![m]\!]_{ud} = ud$. For update functions, the string diagram is decorated with grey backing to group the inputs/outputs of the update function into two main components: the upper components describe the inputs/outputs to the Mealy machine, and the lower components describe the current/next state (e.g. Fig. 5d).



(a) $m_1; m_2$ (b) $m_1 \otimes m_2$ (c) $\mathrm{Tr}^{\Theta}_{\Sigma,\Lambda}(m)$

(d) $[\![m_1; m_2]\!]_{ud}$ (e) $[\![m_1 \otimes m_2]\!]_{ud}$ (f) $[\![\mathrm{Tr}^{\Theta}_{\Sigma,\Lambda}(m)]\!]_{ud}$

Fig. 5: Composite Mealy machines and their update functions

Two Mealy machines $m_1 = (S_1, s_0^1, \Sigma, \Theta, ud_1)$, and $m_2 = (S_2, s_0^2, \Theta, \Lambda, ud_2)$ can be composed in sequence as illustrated by Fig. 5a to form the composite Mealy machine $m_1; m_2 = (S_1 \times S_2, (s_0^1, s_0^2), \Sigma, \Lambda, ud')$. The update function ud' for $m_1; m_2$ with the string diagram in Fig. 5d, is defined as:

$$[\![m_1; m_2]\!]_{ud} = ([\![m_1]\!]_{ud} \otimes id_{S_2}); (id_{\Theta} \otimes \mathrm{Br}_{S_1,S_2}); ([\![m_2]\!]_{ud} \otimes id_{S_1}); (id_{\Lambda} \otimes \mathrm{Br}_{S_2,S_1})$$

The parallel composition of m_1 and m_2 is the Mealy machine $m_1 \otimes m_2 = (S_1 \times S_2, (s_0^1, s_0^2), \Sigma_1 \times \Sigma_2, \Lambda_1 \times \Lambda_2, ud')$ as illustrated by Fig. 5b. The update

function ud' for $m_1 \otimes m_2$, with string diagram Fig. 5e, is defined as:

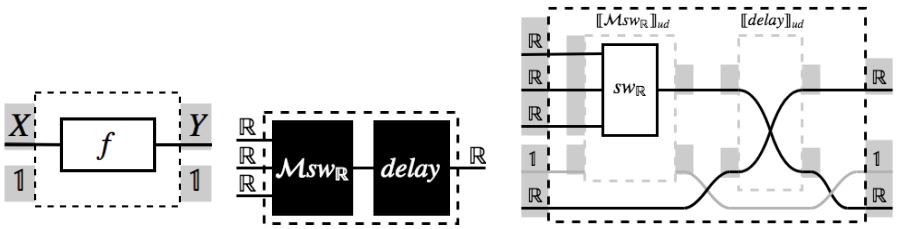
$$[\![m_1 \otimes m_2]\!]_{ud} = (\mathrm{id}_{\Sigma_1} \otimes \mathrm{Br}_{\Sigma_2,S_1} \otimes \mathrm{id}_{S_2}); ([\![m_1]\!]_{ud} \otimes [\![m_2]\!]_{ud}); (\mathrm{id}_{\Lambda_1} \otimes \mathrm{Br}_{S_1,\Lambda_2} \otimes \mathrm{id}_{S_2})$$

Feedback configurations of Mealy machines (e.g., Fig. 5c) can be defined with fixed-point equations, such as in [13]. We give an equivalent description in terms of the trace operation in **Set**. A Mealy machine $m = (S, s_0, \Theta \times \Sigma, \Theta \times \Lambda, ud)$ can be traced to form the machine $\mathrm{Tr}^{\Theta}_{\Sigma,\Lambda}(m) = (S, s_0, \Sigma, \Lambda, ud')$ where the update function ud' is defined as $[\![\mathrm{Tr}^{\Theta}_{\Sigma,\Lambda}(m)]\!]_{ud} = \mathrm{Tr}^{\Theta}_{\Sigma\times S,\Lambda\times S}([\![m]\!]_{ud})$ as illustrated by Fig. 5f. Since this operation is defined in terms of traces in **Set**, many of the properties of traces can be derived from traces in **Set**.

The above results mean that if we know the update functions of individual Simulink blocks, then we can model the update functions of block diagrams which configure those blocks in sequence, in parallel, and with feedback.

4.2 Functional Embedding & Wiring Morphisms

In this section, we address the fact that a large part of a Simulink block diagram *looks* very functional (i.e. stateless). For example, many of the blocks and wiring in Fig. 1a can be modelled as functions. For this reason, we consider a class of Mealy machines which produce outputs as a function of only their current inputs. Any function $f : X \to Y$ can be described as the Mealy machine $\mathcal{M}f = (\mathbb{1}, (), X, Y, f)$, with one state, and update function f (see Fig. 6a). 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**.



(a) $[\![\mathcal{M}f]\!]_{ud} = f$ (b) $\mathcal{M}sw_{\mathbb{R}}; delay$ (c) $[\![\mathcal{M}sw_{\mathbb{R}}; delay]\!]_{ud}$

Fig. 6: Embedded functions and their interactions

This explains how functional aspects of Simulink block diagrams can be modelled with Mealy machines. For example, the block labelled *Mode* in Fig. 1a can be modelled with the Mealy machine $\mathcal{M}sw_{\mathbb{R}}$. Perhaps more importantly, the morphisms introduced to describe wiring in functional diagrams (i.e. id_X, Δ_X, $\mathrm{Br}_{A,B}$) can again be used to describe the same (functional) wiring for Mealy machines. Therefore, in string diagrams representing Mealy machines, plain wires

represent the morphism $\mathcal{M}\,\mathrm{id}_X$ which carries data without changing it, branching wires represent the morphism $\mathcal{M}\Delta_X$ which duplicates data, and crossing wires represent the morphism $\mathcal{M}\,\mathrm{Br}_{A,B}$ which reorders the components of data. The fact that $\mathcal{M}\,\mathrm{id}_X$ and $\mathcal{M}\,\mathrm{Br}_{A,B}$ "act like wiring" is established in [9].

This establishes how to model wiring and functional blocks in Simulink block diagrams as Mealy machines. We can now use the operations from Section 4.1 to describe block diagrams which use complex wiring and functional blocks in combinations with stateful blocks.

4.3 Block Diagrams to Horizontal Condition Tables

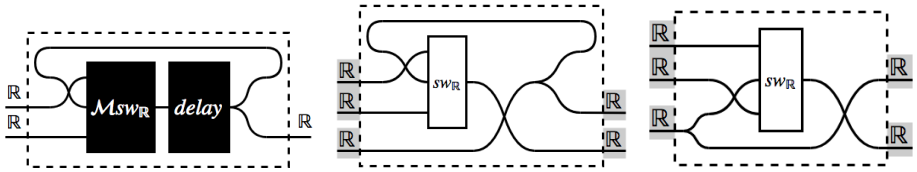
We have explained how the categorical structure from Section 2.3 applies to **Mealy**, and related it to the same structure in **Set**. This framework allows us to combine update functions of individual blocks into update functions of entire block diagrams using the above definitions. For example, the update function $[\![\mathcal{M}sw_{\mathbb{R}}; \mathit{delay}]\!]_{ud}$ of the machine from Fig. 6b is equal to

$$([\![\mathcal{M}sw_{\mathbb{R}}]\!]_{ud} \otimes \mathrm{id}_{\mathbb{R}}); (\mathrm{id}_{\mathbb{R}} \otimes \mathrm{Br}_{\mathbb{1},\mathbb{R}}); ([\![\mathit{delay}]\!]_{ud} \otimes \mathrm{id}_{\mathbb{1}}); (\mathrm{id}_{\mathbb{R}} \otimes \mathrm{Br}_{\mathbb{R},\mathbb{1}}),$$

as shown in Fig. 6c, where the "$\mathbb{1}$" wire is drawn in grey to illustrate how it achieves the data flow described by Fig. 5d (normally, this wire is not drawn). This can be simplified, e.g., the final sequential sub-function $\mathrm{id}_{\mathbb{R}} \otimes \mathrm{Br}_{\mathbb{R},\mathbb{1}}$ is given by $\{(x,(y,()))\mapsto(x,((),y))\}$ which simplifies to $\{(x,y)\mapsto(x,y)\}$ by flattening tuples. Our presentation of monoidal categories skips the formalities which describe this simplification, but it can be intuitively understood by considering the data flow described in Fig. 6c if the grey wire were absent (as usual). Taking $[\![\mathit{delay}]\!]_{ud} = \mathit{shift}$ (as defined in Section 2.1) which we now describe as $\mathrm{Br}_{\mathbb{R},\mathbb{R}}$ and using $[\![\mathcal{M}sw_{\mathbb{R}}]\!]_{ud} = sw_{\mathbb{R}}$ along with appropriate axioms over the wiring morphisms, $[\![\mathcal{M}sw_{\mathbb{R}}; \mathit{delay}]\!]_{ud}$ simplifies to $(sw_{\mathbb{R}} \otimes \mathrm{id}_{\mathbb{R}}); \mathrm{Br}_{\mathbb{R},\mathbb{R}}$. This simplification can be intuitively understood by considering only the black data flow in Fig. 6c. In the same way that we describe the functional data flow of Fig. 3, this approach can be repeated to describe the entire block diagram in Fig. 1a, not just the combination of blocks labelled *Mode* and *counter*.

This example illustrates how our categorical algebra for Mealy machines is structurally similar to the one used in [6] which describes the algorithm that represents block diagrams in terms of sequential/parallel/feedback configurations of components. The algorithm from [6] constructs descriptions which contain no feedback operations. A similar result can be shown in our framework, allowing us to produce trace-free descriptions of update functions in terms of the update functions of their components.

As mentioned in Section 2.3, not all feedback configurations are valid. The validity of a feedback configuration describing a Mealy machine is decided by determining whether or not the trace on its update function is defined. In many settings, the trace is defined if the aforementioned fixed-point equations have a unique solution [13]. However, for Simulink models that are used to generate embedded software, the configuration must satisfy a more strict validity condition:



(a) Mealy Machine (b) Update Function (c) Function Rearranged

Fig. 7: The update function of a Mealy machine with feedback

there must be no *algebraic loops*. This means there can be no cyclic dependencies in the underlying update function, any feedback can be trivially removed by rearranging the components and wiring to "yank out" the loops while preserving the connections between blocks. For example, Fig. 7 illustrates how the update function of a simple feedback configuration can be rearranged to remove loops. This can be formalized by the notion of vacuous guardedness introduced in [7].

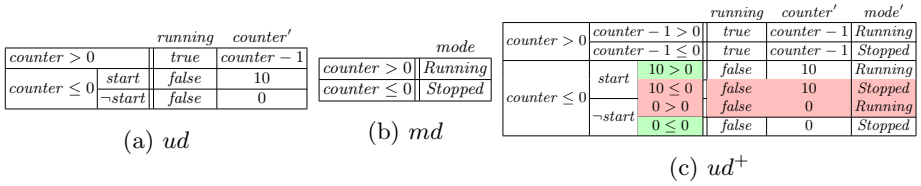
This means that the update functions of well-formed block diagrams can be modelled without traces. In this manner, the update function of the block diagram in Fig. 1a can be described as

$$\mathrm{Br}_{\mathbb{B},\mathbb{R}}; ([-1]\otimes\Delta_{\mathbb{R}}\otimes[10]\otimes\mathrm{id}_{\mathbb{B}}\otimes[0]); (add\otimes\Delta_{\mathbb{R}}\otimes sw_{\mathbb{B}}); (\mathrm{id}_{\mathbb{R}^2}\otimes\mathrm{Br}_{\mathbb{R},\mathbb{R}}); (sw_{\mathbb{R}}\otimes gtz); \mathrm{Br}_{\mathbb{R},\mathbb{B}}$$

where each individual function has a fixed definition, and can be represented as a predefined tabular expression. Here *gtz* denotes the > 0 block labelled *IsRunning*. Functions whose behaviours are not conditional are trivially represented by a table with a single condition: *true*.

HCTs—being representations of functions—can be composed like functions. We modify the composition operation in [20] to describe HCTs so that we can compose predefined tabular expressions as stated above. 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 composition of Fig. 8a with Fig. 8b, where the output $counter'$ of the first table is routed to the input *counter* of the second (ignore the *running* output for now). Their composition is shown in Fig. 8c (ignore the *running* and $counter'$ outputs). The conditions $counter > 0$ and *start* (and their complements) appear in the same configuration as the first HCT. However, the sub-conditions (e.g. $counter - 1 \leq 0$) come from the conditions ($counter \leq 0$) in the second HCT, evaluated with the values ($counter \mapsto counter - 1$) from the row in the first HCT associated with the parent condition ($counter > 0$). The conditions $10 > 0$ and $0 > 0$ (and their complements) are generated in a similar manner, but because they are trivially satisfied/impossible conditions, the sub-conditions/entire row can be removed (the removable conditions/rows are shaded in Fig. 8c).

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



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

(a) *ud*

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

(b) *md*

			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*

(c) ud^+

Fig. 8: Introducing Modes

as the output expressions of the combined HCT. In Fig. 8b, the output values for *mode* are constants, therefore they appear unchanged in Fig. 8c. For HCTs composed in parallel, the conditions from the second HCT are once again used as sub-conditions, but they are not modified. Similarly, the output expressions from both HCTs are placed in the combined table unchanged.

The predefined HCTs representing each function in the equation above can be combined using the operations described above to achieve a tabular expression for the entire block diagram. For example, the tabular expression in Fig. 2a can be obtained this way.

5 HCTs to STTs: Modes via Tables

The HCTs produced using the technique described in Section 4 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.

5.1 Defining Modes

The STTs described in Section 2.2 have obvious similarities to state charts, but they are just syntactic sugar for HCTs. STTs and state charts are modelled as Mealy machines with a special state variable *mode* with values from an enumerated set M (see, e.g., extended state machines in [2]). The cells in the first column of STTs (see Fig. 2b) express conditions of the form $mode = Running$ which compare the value of *mode* to each element of M. The last column identifies the updated value of $mode'$. 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. To achieve the goal of improving readability, we leverage the existing decision logic in HCTs.

When a state chart updates, it only considers the transitions leaving its current mode, i.e., depending on its *state*, only some behaviours are possible. The same dependence on state is expressed in HCTs by conditions which depend only on the values of state variables, which will be referred to as *state conditions*. For example, in Fig. 8a, if the condition $counter > 0$ is satisfied, the system can only do one thing: decrement *counter* and set *running* to true. Our strategy associates the condition $counter > 0$ with a mode of operation $Running \in M$, and replaces the original condition with $mode = Running$. We augment the HCTs into STTs in a way that preserves the behaviour of the Mealy machines.

As the modes are all listed in the first column of an STT, the first augmentation reorders conditions in HCTs so that the state conditions appear first. For example, the conditions in Fig. 2a can be rearranged via the methods in [4] to obtain Fig. 8a. While our example contains only one pair of state conditions, HCTs describing general block diagrams may contain multiple nested state conditions. The second augmentation uses conjunction to flatten nested state conditions into a single column with a condition for each branch of the stateful logic.

The augmented HCT now has a specific form (Fig. 8a) which superficially resembles an STT, but the behaviour is unchanged. We now introduce a set of modes M with each element associated with a distinct condition in the first column of the augmented HCT. This association is defined by a function $md : S \to M$ which maps tuples of state variable values to the mode whose associated state condition is satisfied. This function is represented by an HCT with the state conditions from the augmented HCT, and distinct values from M as outputs. The md function for the timer example is given by the HCT in Fig. 8b.

Next, the Mealy machine is enhanced by introducing a state variable *mode* with values from M. We design the enhancement to maintain the invariant that the value of *mode* always corresponds with the state condition which the other state variables satisfy. The invariant is satisfied by the initial state $(s_0, md(s_0))$. The enhanced update function trivially preserves the original behaviour by ignoring the value of *mode*, but updates $mode'$ to maintain the invariant by evaluating md with the updated state variable values. The update function is therefore defined as $ud^+ = (ud \otimes !_M); (id_\Lambda \otimes (\Delta_S; (id_S \otimes md)))$, where $!_M : M \to \mathbb{1} = \{(mode) \mapsto ()\}$ introduces an input whose value is discarded. Since ud and md are given as HCTs (e.g. Fig. 8a and Fig. 8b), the enhanced update function can be achieved through composition of tables (e.g. Fig. 8c).

This enhanced Mealy machine operates within a subset of the state space $S \times M$ where the aforementioned invariant holds. The validity of any state condition can now be deduced from the value of the mode variable (e.g. $(counter > 0) \Leftrightarrow (mode = Running)$). Thus, replacing those conditions with the corresponding modes in the HCT representation of ud^+ does not modify its behaviour. This is the final step in rearranging the HCT from Fig. 8c into the STT in Fig. 2b.

5.2 Converting to State Charts & Simplifying

The state chart in Fig. 9 implements the STT in Fig. 2b by creating a transition for each row and by creating assignment actions to update state and output

variables. State charts produced in this manner can often be simplified by moving common actions from transitions to *entry*/*exit* actions of modes, or by removing transitions and performing the corresponding actions as *during* actions. For example, the state chart in Fig. 9 simplifies to the one in Fig. 1b.

In the example given above, 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 additional information is not always important, i.e., sometimes the mode is sufficient and the old state variable may be removed from the description of the Mealy machine. 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 is also possible 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. In these cases, the unnecessary state variables can be removed from the state chart.

6 Prototype, Evaluation, and Future Work

The methodologies presented here have been used to develop a prototype tool which automatically refactors Simulink model fragments to Stateflow [18]. The tool supports a large subset of discrete Simulink blocks typically used for implementation of embedded software. The refactoring tool is implemented in Matlab and integrates with Simulink allowing the user to select the blocks they would like to replace. When the tool is invoked, it generates a Stateflow chart and uses the Simulink Design Verifier [17] to verify that it is equivalent to the selected blocks.

The prototype tool improves the readability of small to medium sized block diagrams such as the one in Fig. 1a. However, we found that the stateful logic of complex industrial-scale models incorporates multiple state machines interacting with each other and with stateless conditional logic. To elegantly represent these complex block diagrams in Stateflow, the translation methodologies presented here can be enhanced to utilize the more sophisticated mechanisms of state charts such as hierarchical/parallel modes. We believe that many state chart mechanisms have analogies in tabular expressions, e.g., using hierarchies of state conditions can be leveraged to specify sub-modes. We found that block diagrams

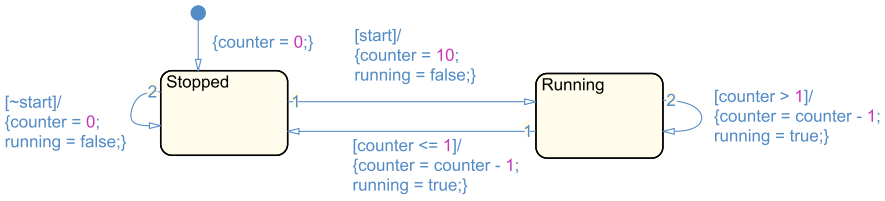


Fig. 9: State Chart Equivalent to STT

encoding more than 4 high-level modes can often become difficult to understand without these mechanisms.

We also recognize the importance of finding refactorable fragments in large models. In fact, the translation methodology presented in this paper was developed in parallel with an identification strategy that pinpoints block diagrams which are candidates for refactoring—it searches for certain patterns of logical and stateful blocks which indicate complex state update logic. An elaborated description of both translation and identification strategies will be presented in the master's thesis of the first author[29].

7 Related Work

Several papers propose translating Simulink block diagrams to formal languages to enable their verification using existing tools (e.g., [1,6,14,23,27,30]). Only a few, however, translate Simulink block diagrams to state transition diagrams. In [19], 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 [31], 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 [19,31], as well as [16], do not aim to capture the high-level state machine of an entire block diagram. This is exactly what our approach does, with maintainability of the resulting model as a prime motivator.

Our approach to modelling Mealy machines and their interactions using the monoidal category **Mealy** follows a general trend in behavioural modelling. For example, monoidal categories have been used to describe interactions of quantum processes [5], labelled transition systems [12], and control systems [3]. The algebra of (traced symmetric) monoidal categories is similar to the algebra used to describe block diagrams in [6], but our approach uses a standard mathematical framework with a rich history and many known results. For example, the results of [9] indicate that by considering equivalence up to bisimilarity, the category **Mealy** is symmetric monoidal, meaning the appropriate axioms and resulting properties of this structure are already known.

8 Conclusion

In this paper, we proposed a method for translating Simulink block diagrams to Stateflow state charts via tabular expressions representing their respective Mealy machines update functions. A categorical framework for composing Mealy machines provides a theoretical basis for the translation. To the best of our knowledge, this is the first method for Simulink to Stateflow translation. Our proposed method is relevant to industrial development where it can help improve software maintainability and aid compliance with modelling guidelines.

References

1. Agrawal, A., Simon, G., Karsai, G.: Semantic translation of Simulink/Stateflow models to hybrid automata using graph transformations. Electronic Notes in Theoretical Computer Science **109**, 43–56 (2004)
2. Alur, R.: Principles of cyber-physical systems. MIT Press (2015)
3. Baez, J.C., Erbele, J.: Categories in control. Theory and Applications of Categories **30**(24), 836–881 (2015)
4. Bialy, M., Lawford, M., Pantelic, V., Wassyng, A.: 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). pp. 47–53. IEEE Press (May 2015)
5. Coecke, B., Kissinger, A.: Picturing quantum processes. Cambridge University Press (2017)
6. Dragomir, I., Preoteasa, V., Tripakis, S.: Translating hierarchical block diagrams into composite predicate transformers. arXiv preprint arXiv:1510.04873 (2015)
7. Goncharov, S., Schröder, L.: Guarded traced categories. In: Baier, C., Dal Lago, U. (eds.) Foundations of Software Science and Computation Structures. pp. 313–330. Springer International Publishing, Cham (2018)
8. Hasegawa, M.: Recursion from cyclic sharing: traced monoidal categories and models of cyclic lambda calculi. In: International Conference on Typed Lambda Calculi and Applications. pp. 196–213. Springer (1997)
9. Hofmann, M., Pierce, B., Wagner, D.: Symmetric lenses. ACM SIGPLAN Notices **46**(1), 371–384 (2011)
10. Jin, Y., Parnas, D.L.: Defining the meaning of tabular mathematical expressions. Science of Computer Programming **75**(11), 980–1000 (2010)
11. Joyal, A., Street, R., Verity, D.: Traced monoidal categories. In: Mathematical Proceedings of the Cambridge Philosophical Society. vol. 119, pp. 447–468. Cambridge University Press (1996)
12. Katis, P., Sabadini, N., Walters, R.F.: Span (graph): A categorical algebra of transition systems. In: International Conference on Algebraic Methodology and Software Technology. pp. 307–321. Springer (1997)
13. Lee, E.A., Varaiya, P.: Structure and interpretation of signals and systems. LeeVaraiya.org, 2nd edn. (2011)
14. Liebrenz, T., Herber, P., Glesner, S.: Deductive verification of hybrid control systems modeled in Simulink with KeYmaera X. In: International Conference on Formal Engineering Methods. pp. 89–105. Springer (2018)
15. Malherbe, O., Scott, P.J., Selinger, P.: Partially traced categories. Journal of Pure and Applied Algebra **216**(12), 2563–2585 (2012)
16. Manamcheri, K., Mitra, S., Bak, S., Caccamo, M.: A step towards verification and synthesis from Simulink/Stateflow models. In: Proceedings of the 14th International Conference on Hybrid systems: Computation and Control. pp. 317–318. ACM (2011)
17. MathWorks: Simulink Design Verifier. https://www.mathworks.com/products/sldesignverifier.html (2018), [Online; accessed Nov 18th, 2018]
18. McSCert: Simulink-to-Stateflow. https://www.mathworks.com/matlabcentral/fileexchange/70317-simulink-to-stateflow (2019), [Online; accessed February 2019]
19. Minopoli, S., Frehse, G.: Sl2sx translator: From Simulink to SpaceEx models (April 2016), uRL: http://www-verimag.imag.fr/~minopoli/SL2SX.pdf

20. von Mohrenschildt, M.: Algebraic composition of function tables. Formal aspects of computing **12**(1), 41–51 (2000)
21. Parnas, D.L.: Tabular representation of relations. Tech. rep., McMaster University (October 1992)
22. Selinger, P.: A survey of graphical languages for monoidal categories. In: New structures for physics, pp. 289–355. Springer (2010)
23. Sfyrla, V., Tsiligiannis, G., Safaka, I., Bozga, M., Sifakis, J.: Compositional translation of Simulink models into synchronous bip. In: Industrial Embedded Systems (SIES), 2010 International Symposium on. pp. 217–220. IEEE (2010)
24. Singh, N.K., Lawford, M., Maibaum, T.S., Wassyng, A.: Stateflow to tabular expressions. In: Proceedings of the Sixth International Symposium on Information and Communication Technology (SolCT). p. 47. ACM (2015)
25. The MathWorks: MathWorks Automotive Advisory Board (MAAB): Control Algorithm Modeling Guidelines Using MATLAB, Simulink, and Stateflow, Version 3.0 (2012), www.mathworks.com/solutions/automotive/standards/maab.html
26. The MathWorks: Simulink user's guide. http://www.mathworks.com/help/releases/R2018b/pdf_doc/simulink/sl_using.pdf (Sep 2018), http://www.mathworks.com/help/releases/R2015b/pdf_doc/simulink/sl_using.pdf, version R2018b. [Online; accessed Feb 2019]
27. Tripakis, S., Sofronis, C., Caspi, P., Curic, A.: Translating discrete-time Simulink to Lustre. ACM Transactions on Embedded Computing Systems (TECS) **4**(4), 779–818 (2005)
28. Wassyng, A., Janicki, R.: Tabular expressions in software engineering. Proceedings of 2003. International Conference on Software and System Engineering (ICSSEA'03) pp. 1–46 (2003)
29. Wynn-Williams, S.: SL2SF: Refactoring Simulink to Stateflow (2019), unpublished thesis
30. Zhan, N., Wang, S., Zhao, H.: Formal Verification of Simulink/Stateflow Diagrams. Springer (2017)
31. Zhou, C., Kumar, R.: Semantic translation of Simulink diagrams to input/output extended finite automata. Discrete Event Dynamic Systems **22**(2), 223–247 (2012)