# McFSM: Near Turing-Complete Finite-State Based Programming

Florian Murr
Siemens AG, Corporate Research
florian.murr@siemens.com

Wolfgang Mauerer
Technical University of Applied Sciences Regensburg
Siemens AG, Corporate Research
wolfgang.mauerer@othr.de

## ABSTRACT

Finite state machines (FSMs) are an appealing mechanism for simple practical computations: They lend themselves to very efficient and deterministic implementation, are easy to understand, and allow for formally proving many properties of interest. Unfortunately, their computational power is deemed insufficient for many tasks, and their usefulness has been further hampered by the state space explosion problem and other issues when naïvely trying to scale them to sizes large enough for many real–life applications.

This paper expounds on theory and implementation of *multiple coupled finite state machines* (McFSMs), a novel mechanism that combines benefits of FSMs with near Turing-complete, practical computing power, and that was designed from the ground up to support static analysis and reasoning. We develop an elaborate category–theoretical foundation based on non–deterministic Mealy machines, which gives a suitable algebraic description for novel ways of blending different computing models. Our experience is based on a domain specific language and an integrated development environment that can compile McFSM models to multiple target languages, applying it to use-cases based on industrial scenarios. We discuss properties and advantages of McFSMs, explain how the mechanism can interact with real–world systems and existing code without sacrificing provability, determinism or performance.

We discuss how McFSMs can be used to replace and improve on commonly employed programming patterns, and show how their efficient handling of large state spaces enables them to be used as core building blocks for distributed, safety critical, and real–time systems of industrial complexity, which contributes to the long-desired goal of providing executable specifications.

## KEYWORDS

Finite state machines, Mealy machines, automata, coupled machines, executable specification, category theory, generative approaches, formal models, static analysis

**ACM Reference Format:**
Florian Murr and Wolfgang Mauerer. 2020. McFSM: Near Turing-Complete Finite-State Based Programming. In *Proceedings of 35th IEEE/ACM International Conference on Automated Software Engineering (ASE 2020).* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
*ASE 2020, 21–25. Sep. 2020, Melbourne, Australia*
© 2020 Association for Computing Machinery.
ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

## 1 INTRODUCTION

State information is omnipresent in systems: Physical and logical properties like speed, temperature, time, duration, or error status are required to describe almost any kind of machine. Such properties are intuitively considered to be part of the *state* of a system, and are typically used in specifications to describe its workings or failures, or to make runtime decisions, by referring to them. Different properties often belong to different *components* of a system, and multiple components may influence one another. Correctly handling such interdependencies is crucial in system design and implementation.

Many advances in computer science have focused on the objective of *localizing* effects, as is evident from functional or object-oriented programming [51]. A major purpose of this is limiting side-effects to manageable portions of the code, as do common techniques like *information hiding* or *separation of concerns* [15]. While the total state is composed of all the different component states, the effects of any event on this total state may not be local, but spread across multiple components and set off an avalanche of internal changes. Orchestrating such cascades of internal events and internal reactions into a reliable, working system is one of the main challenges of software engineering [9]. Developers often try to cope with this by using design patterns like the ubiquitous observer pattern [32]. As Maier and Odersky show [44], the pattern may provide a false sense of security, while actually making the system more complex and harder to debug. What is really needed are compiletime means of describing and debugging these interdependencies, while ensuring as much locality as possible.

This paper introduces a novel mechanism – multiple coupled finite state machines (McFSMs) – together with a domain specific language (DSL) for model-based development and abstract specification. Relating changes to states and properties of different parts of a system is possible by choosing a super-ordinate system description with a *finite state machine* (FSM), which is a very well-known and intensively studied mechanism [36]. It allows us to *communicate* and *reason* about practically any deterministic system. In contrast to describing a system with a single, very large finite state machine that can quickly become unwieldy for realistic system sizes, our mechanism employs multiple FSMs coupled by notifications to make the structure of a cooperating system explicit. It aims at retaining the simplicity and advantages of local FSMs by enveloping them with a *global superordinate structure* that takes care of the intricacies brought about by their interdependencies. As a *low-level* mechanism, it lends itself to an efficient implementation that can, thanks to properties discussed in detail below, serve as basis for real-time and safety-critical industrial systems. It avoids the pitfalls of scattered observers acting largely agnostic of their interdependencies. Owing to a *theoretical* foundation based

DRAFT

on category theory, it lends itself to rigorous scrutiny and formal verification.

The composition logic required to orchestrate components does often *not* require Turing–complete code (for instance, simple dispatchers may suffice), but is usually implemented in the same Turing–complete mechanism used for the core code. McFSMs aim primarily at explicitly describing and handling interdependencies between components, and therefore the global structure of a composite system, at compiletime and with non-Turing–complete means. Although our description is synchronous and deterministic at the core, it also allows for handling asynchronous external processes and events. They can be used to reduce implicit or undesired dependencies between components, and foster a consistent and well-structured global description of interdependencies.

Figure 1 illustrates the well-known relation between expressiveness of various classes of languages as defined by the Chomsky hierarchy [56], and the increasing complexity of proving properties of programs. are at opposite borders of this spectrum. We have deliberately designed our approach to fall in between FSMs and Turing–machines: It want to benefit from provability properties of simpler approaches, but still be sufficiently expressive to conveniently handle and describe non–trivial industrial problems.

Established model checking approaches [20, 21, 38], for instance LTL or CTL [59], describe a given implemented system with an additional *model* that captures its essential properties, and then prove derived properties. Our approach combines modelling and implementation: We extract and model the core *decision structures* of programs into a superordinate description based on non-Turing–complete means. To ascertain practicality (for instance, to interact with OS-level features), the approach allows for including components given in Turing–complete languages that specify behavioural properties that can be assumed in proofs. Provability, then, is of course at the discretion of the guarantees specified and satisfied by the extension.

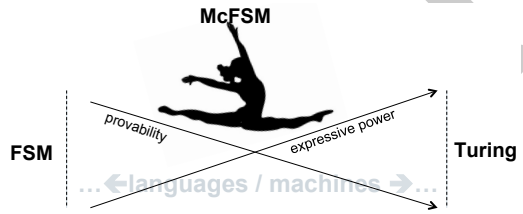


**Figure 1: Expressiveness in languages complicates proofs of correctness or other relevant properties. Our approach is designed to perform the balancing act of bridging between sufficient expressivity for real-world problems (as given by Turing-complete mechanisms) and simplicity (as provided by FSMs) that enables provability.**

Our considerations base on Mealy machines [46, 56] that consume input–events and issue output–events. McFSMs provide a *deterministic* mechanism to connect them into larger compounds. A third entity, *Pseudo–FSMs*, provides a connection with Turing–complete portions of a system. It is known that when systems comprising multiple components are modelled using a straightforward product automaton approach, an exponential increase in both the number of states and edges can occur. Our approach allows us to describe industrially relevant coupled systems while avoiding such an exponential "state space explosion". We achieve this by providing an *algebraic* description of FSMs using appropriate category-theoretical [29, 41] constructions, in contrast to the usual approach of *co-algebraically* [6] describing FSMs and related mechanisms for formal verification. A co-algebraic description implies that by *observing* the behaviour of a system, it is possible to construct an equivalent system that may or may not be structurally identical. This often leads to the construction and design of systems relying internally on different variables and functions than the ones used during their specification.[1]

We claim the following contributions:

(1) A novel mechanism to described coupled systems designed to facilitate direct static analysis, to separate the global superordinate coordination structure of systems from implementation details of components.
(2) A complete formal treatment based on category theory.
(3) Generative mechanisms to produce practically deployable code in common low-level and scripting languages usable in many classes of industrial systems.
(4) An human-readable, domain specific language (DSL) similar to a programming language, and an associated industrial strength integrated development environment available as open source software available on the companion website (hyperlink available in PDF).

Overall, our approach provides a step towards an *executable specification*, which allows for verifying that a given generated system behaves as intended from one single, unified system description. While our work also includes techniques to reason about programs and perform static analysis, we cannot, for the lack of space, address these aspect in the current paper, and focus on formalisms and software engineering issues. Details on reasoning will be provided in subsequent work; practical aspects of the integrated development environment are described in an online supplement (see the companion website).

## 2 ILLUSTRATIVE EXAMPLE

We start explaining the syntactic features of our DSL by considering the illustrating example in Fig. 2, which uses the McFSM formalism to implement an elementary light bulb (`FSM class Bulb`), that may toggle between the states `on` and `off` (lowercase) and another bulb (`McFSM class McBulb`) that uses one such bulb `B` (`Bulb inst B`) and lets it additionally blink in periodic intervals.

Consider `class Bulb`: Line 1 declares a list of strings named `OO` containing the two string values `On` and `Off`. This list is used multiple times as an *xvar variable* `OO` in a *xvar pattern* `%OO`. These patterns indicate implicit for–loops, and have the format *%var*, where

[1] The difference is similar to the difference between FSMs and deterministic finite automata (DFAs): The latter mechanism is an acceptor with the purpose of deciding a language; internal states and transitions are only a means of determining which words are a part of the language, contrary to FSMs, where such transitions are the main reason for the existence of the computational mechanism. Two structurally different DFAs serve the same purpose as long as they accept the same language, whereas this not the case for FSMs.

DRAFT

```
1  set 00 {On Off}
2  FSM class Bulb {
3      hop -loops      %00_%00  i%00   -> oIs%00
4      hop               *_%00  i%00   -> o%00
5      hop               *_%00  iBlink -> oBlink
6  }
7  McFSM class McBulb {
8      Bulb inst B {
9          initialState off
10         iconnect i%00    <-  /i%00
11         iconnect iBlink  <-  /tBlink  /iBlink
12         when oBlink call emitT(/tBlink,0.7sec,/i%00)
13     }
14     oconnect /o%00    <- B/o%00
15     oconnect /oBlink  <- B/oBlink
16 }
```

**Figure 2: Modelling a switchable light bulb, and an extension that can blink with the McFSM DSL.**

*var* is the name of a list-variable. They implement a string replacement called *xvar resolution.*

Line 3 multiply contains the same xvar pattern %00. It starts with the command hop, which works as a constructor for states, state transitions, input- and output–events. It triggers string replacement by handing its list of parameter strings (here %00_%00, i%00 and oIs%00) to xvar resolution. This function first determines the amount of *different* xvar patterns in a strings (here only one: %00), and then creates as many nested implicit for–loops as required to replace each xvar pattern with the value of the corresponding loop variable. As a result, all combinations of the string elements of the respective xvar variables are created, and the result is returned to the triggering command. In the example, this delivers `on_on iOn -> oIsOn", "off_off iOff -> oIsOff`. hop knows where to expect state transitions (written with an underscore as *state1_state2*), input–events, and outputevents. It creates all elements referenced in the above string, which in the example equates to two states on, off, two input–symbols iOn, iOff, two state transitions on_on, off_off, and two output–symbols oIsOn, oIsOff. The option -loops instructes hop to accept cyclic state transitions.

Line 4 shows a *glob pattern*, identified by an asterisk *, that performs pattern matching against the elements so far defined in the current class. The glob patterns in the example, *_on and *_off, are matched against the list of already defined states on and off. Since the flag -loops is not given, line 4 is equivalent to hop off_on iOn -> oOn and hop on_off iOff -> oOff. Likewise, line 5 is equivalent to hop off_on iBlink -> oBlink. and hop on_off iBlink -> oBlink. Class Bulb has three input–symbols iOn, iOff, iBlink and five output–symbols oOn, oOff, oIsOn, oIsOff and oBlink. The idea is to have Bulb return oOn if it was not on, and has switched to on and oIsOn if it was already on in the first place—analogously for off.

A major difference between *xvar patterns* and *glob patterns* is that the former perform string transformations agnostic of the current context (class and command), while the latter match elements depending on the current context. Xvars use the same string lists across different classes, and thus work globally, while glob uses knowledge about the local context to fill in information.

Line 7 defines a McFSM named McBulb, which in line 8 creates its first (and only) *member* B (Bulb inst B). This is an *instance* of class Bulb. Line 9 sets the initial state of B to off. Lines 10 and 11 start with command iconnect (mnemonic for "input connect") and it creates the input–events of the surrounding McFSM indicated by the slash / as prefix. The event names without slash are the ones provide by class Bulb.

Lines 14 and 15 analogously connect the output–symbols of members (only B) with the ones of the surrounding McFSM (McBulb).

Now McBulb has the input–symbols iOn, iOff, iBlink and the timer input tBlink and the output–symbols oOn, oOff, oBlink.

Allowing the bulb to periodically blink is implemented with the help of *timer event* tBlink, issued by the when command in line 12. It triggers the scheduler, which is part of the McFSM runtime environment, to issue an event tBlink with a temporal delay, here 0.7 seconds (this default value can be reconfigured at runtime), should it not be already scheduled. The last parameter in the emitT() symbol expands to /iOn, /iOff and indicates *timeout* events, that means the event tBlink gets canceled or ignored, should event iOn or iOff happen for the McFSM before the timer event tBlink has arrived.

It is important to note that the DSL focuses on the pure event structure, and does *not*, in contrast to orthodox programming languages, describe any actions or side effects. Based on the description in the DSL, our system generates code in an object-oriented target language (TL)—this can be C++, Python, Tcl, …—, and provides a class structure that corresponds to the classes defined in the DSL. The interaction with the physical system is left to routines dispatched by these classes.

This separation between event structure and generic code is supposed to make both easier to understand and maintain. Each TL has to provide a slim runtime library, whose detailed design is not of interest here. However, we remark that our reference implementation implements transitions in a single thread, but allows input–events to come from other threads. Given that the runtime library appropriately handles such multi-threaded interactions, McFSM objects can typically act as some kind of "mega-mutex" handling concurrency issues and letting the developers work as if all other pieces were single-threaded.

## 3 MCFSM: FORMALISM

A **McFSM** $m$ is a Mealy machine equipped with an internal structure given by a list of smaller *member*-Mealy machines $m_1, \ldots, m_n$. The members $m_i$ can be combined and interconnected by using the output of some as input to others. In contrast to other models, most importantly the widely used UML state machines [48, 54], McFSMs use a deterministic and synchronous coupling, which results in a strictly sequential internal event distribution protocol. This ascertains that an McFSM and the connection protocol between members is deterministic, given that all members are deterministic.

Apart from Mealy machines, a McFSM may contain other elements: External events (input and timers) feed input to a McFSM, and provide a notion of temporal progress compatible with the model; both can also be handled deterministically. pseudo–FSMs and plugins allow for interfacing with external components written in regular programming languages without re-introducing the many issues of Turing–complete code. Plugins and pseudo–FSMs can introduce non–determinism, and require a careful and detailed

DRAFT

treatment that we will provide in Sec. 6. For now, let us make do with just mentioning their existence and purpose, and suffice it to say that our formalism can state general requirements that ensure they behave like a Mealy machine.

To simplify some considerations, we add a virtual member $m_0$ that uses the input–events of a McFSM as its output–events, and a virtual member $m_{n+1}$ that has the output–events of the McFSM as its input–events. The processing of an input–event corresponds then to a sequence of output-to-input event connections starting at $m_0$ and ending at $m_{n+1}$. We refer to this as the *internal connection network*.

To make McFSMs work, an internal algorithm and data-structure is needed that takes responsibility of creating and operating the right such sequence. How events are distributed and processed is derived from elementary requirements:

**Keep order.** Events must keep their order, that is no later event may overtake a previous one.

**Run to completion.** Every event is processed until done. Internal event processing shall not be interrupted by other events.

**Distribution order.** The members get each event in a deterministic, predefined, possibly event-specific, order.

A formal description of the distribution mechanism turns out to be astonishingly complicated, so we omit it here (details are provided in the companion website). Essentially, it resembles a sequence of function calls. Assume that machine $m_i$ accepts inputs $E_i = \{e_{i,1}, \ldots, e_{i,\#E_i}\}$ and returns as outputs one of $O_i = \{o_{i,1}, \ldots, o_{i,\#O_i}\}$ for $i \in \{0, \ldots, n+1\}$. We can take as a simple example some input–event $e_{i,j}$ to $m_i$ and exemplify the internal distribution mechanism of its output $o_{i,k}$ to two coupled machines $m_a, m_b$, with $o_{i,k}$ connected to $e_{a,u}$ and $e_{b,v}$. This can be viewed as a sequence of function-calls in pseudo-code:

```
O_i m_i::e_{i,j}() {y_a=m_a.e_{a,u}(); y_b=m_b.e_{b,v}(); ... return o_{i,x};}
```

The return value $o_{i,x}$ usually gets chosen depending on the return-values $y_a, y_b$, which would unnecessarily complicate our presentation.

The order of calls for $o_{i,k}$ corresponds to a sequence of the members $m_i$. If not specified otherwise, we use their order in the McFSM. The absence of cycles makes sure that the sequences for individual events combine into an ordered *tree* of calls.

To distribute events across the system, a stack is required to store any intermediate values; as usual, each function may call other functions in turn. In the absence of recursive cycles in these calls, the call sequence terminates in a finite number of steps. The exact amount can (contrary to regular programming languages) be computed at compiletime.

Checking and guaranteeing properties at compiletime is an essential feature of FSMs and McFSMs. For instance, one straightforward condition that ascertains the absence of undefined behaviour or unhandled cases in a system at runtime is the Mealy condition: It requires that both, $\delta : Q \times \Sigma \to Q$ and $\omega : Q \times \Sigma \to \Omega$, are total functions. While it is syntactically easy to violate the condition in a McFSM specification, the violation can always be identified at compiletime, which is one of the main goals of static analysis.

# 4 CATEGORIES FOR STATE-MACHINES

This section develops a suitable mathematical model, starting with FSMs and extending it to finite and infinite, deterministic and non–deterministic general state machines (SMs) and Mealy machines.[2] Mathematical category theory (CT) is pertinent because it allows us to concisely expound the essence of our concepts. We do not assume close familiarity with CT beyond basic concepts— recall that CT is essentially about *objects* connected by *morphisms* (*arrows*) that satisfy *associativity* for their composition.

The objects we define represent models of computation, and our morphisms represent the interrelations between these models. We also automatically benefit from established definitions and propositions in CT, for example the definition of when two objects are considered isomorphic.

Let us define a category **FSM** of finite-state-machines (bold font is used for categories) that respects their internal structure, at least as far as reachable states are concerned. We do not equate the *algebraic* model of FSMs with the *co–algebraic* model of FAs. These two alternatives consider different perspectives: In FSMs, the *states* bear semantics, while FAs focus on the *accepted language*, and states are just means to perform the separation between elements inside and outside this language.

A McFSM is based on a state space $Q \approx Q_1 \times \cdots \times Q_n$, which is roughly the product space of $n$ *member* FSMs $m_i$ ($0 < i < n; n, i \in \mathbb{N}$). Each individual state space $Q_i$ bears individual semantics. We want to emphasize that isomorphic FSMs are equivalent as DFAs, but not vice versa: We aim at an algebraic definition of FSMs that allows us to see an equivalence of FSMs based on their inner state structure as opposed to the usual co-algebraic definition of DFAs, which is based on their input-/output- relations and equivalence based on observed behaviour, that is, their accepted language.

We use the following notation: Let $X, Y$ be sets, $f : X \to Y$ a function, $\emptyset$ the empty set; $\mathcal{P}(X)$ the power set of $X$ and $\mathcal{P}(f) : \mathcal{P}(X) \to \mathcal{P}(Y), A \mapsto f(A)$ the corresponding function; $X^n$ the set of n-tuples (we treat *tuples*, *strings* and *sequences* as synonyms). ($n \in \mathbb{N}$); $X^* = \bigcup_{n\in\mathbb{N}} X^n$ the set of all finite strings[3] of $X$ including the empty string $\varepsilon$; $X^+ := X^* \backslash \{\varepsilon\}$ the set of non-empty finite sequences; string concatenation $\cdot : X^* \times X^* \to X^*$, that is $u, v \in X^* \Rightarrow uv := u \cdot v := \cdot(u, v) \in X^*$. $f$ may be extended to $X^*$ as $f^* : X^* \to Y^*$ with $f^*(\varepsilon) := \varepsilon$; $f^*(u \cdot a) := f^*(u) \cdot f(a)$;

As additional notations: $\mathcal{P}_+(X) := \mathcal{P}(X) \backslash \{\emptyset\}$ the set of non--empty subsets of X, $\mathcal{P}_+(f)$ the restriction of $\mathcal{P}(f)$ to $\mathcal{P}_+(X)$ and $\iota_X : X \rightarrowtail \mathcal{P}(X), \quad x \mapsto \{x\}$ the injective embedding of $X$ into $\mathcal{P}(X)$. Finally, we use von Neumann's convention to equate 0 with the empty set, and define $n + 1 := n \cup \{n\}$ as successor function (especially $2 := \{0, 1\}$ is of interest for us).

A *deterministic state machine* is a quadruple $dsm := \langle Q, \Sigma, \delta, q_0 \rangle$, where $Q$ is a (finite) set of states, $\delta : Q \times \Sigma \to Q$ the transition-function, $q_0 \in Q$ an initial state [37]. The (finite) sets $\Sigma, \Omega$ are called *alphabets* and their elements *symbols* or *events*. The domain of $\delta$

[2] The prefixes 'D', 'N', 'F' are used to indicate the particular kind, like: *NFSM* := non–deterministic finite state machine, *DFMealy* := deterministic finite Mealy machine, … The terms *FSM* and *Mealy* are more lax. Usually they are short for the deterministic and finite variants, but may sometimes encompass other kinds as well. Likewise we use this convention for automata: FAs, DFAs, NFAs.

[3] We typically use as variable names: $a, b, c \in X$; $u, v, w \in X^*$, if not stated otherwise.

DRAFT

can easily be extended to strings: $\tilde{\delta} : Q \times \Sigma^* \to Q; \quad \tilde{\delta}(q, \varepsilon) := q;$ $\tilde{\delta}(q, u \cdot a) := \delta(\tilde{\delta}(q, u), a)$; and further extended to sets of states: $\hat{\delta} : \mathcal{P}(Q) \times \Sigma^* \to \mathcal{P}(Q); \quad \hat{\delta}(\emptyset, w) := \emptyset; \quad \hat{\delta}(\{q\}, w) := \{\tilde{\delta}(q, w)\};$ $\hat{\delta}(R, w) := \bigcup_{q \in R} \hat{\delta}(\{q\}, w)$; We may write $\delta$ for any of the three functions $\delta$, $\tilde{\delta}$ and $\hat{\delta}$, because the context usually makes clear what is meant and $\delta(\{q_0\}, \Sigma^*)$ for the set of *reachable* states.

Since $q_0$ is fixed, *dsm* does correspond to a function $m : \Sigma^* \to Q$; with $m(\varepsilon) = q_0; \quad m(a) = \delta(q_0, a); \quad m(v \cdot a) = \delta(m(v), a)$; This allows us to define *categories* **DSM** and **DFSM** where the *objects* are such functions $m : \Sigma^* \to Q$. Morphisms $f : m \to m'$ are pairs of functions $f = \langle f_1, f_2 \rangle$, that make the following diagram commute

$$\begin{array}{ccc} \Sigma^* & \xrightarrow{m} & Q \\ {\scriptstyle f_2^*}\downarrow & & \downarrow{\scriptstyle f_1} \\ \Sigma'^* & \xrightarrow{m'} & Q' \end{array} \qquad \begin{array}{l} f_1 : Q \to Q'; \quad f_2 : \Sigma \to \Sigma'; \\ f_2^* : \Sigma^* \to \Sigma'^*; \\ \text{that is } \; f_1 \circ m = m' \circ f_2^*. \end{array}$$

*Short notation*: $m \xrightarrow{\langle f_1, f_2 \rangle} m'$, or $\langle f_1, f_2 \rangle : m \to m'$. *Identities* are the *pairs* of identity functions $id_m = \langle id_\Sigma, id_Q \rangle$ and *composition* $g \circ f : m \to m''$ is defined using component–wise function–composition $g \circ f := \langle g_1 \circ f_1, g_2 \circ f_2 \rangle$, which is *associative* as required.

These categories yield an *algebraic* definition of *deterministic (finite) state machines* and thus provide us with definitions of (*epi-*, *mono-*, *iso-*) *morphisms* between them, respecting their reachable states.

*Deterministic Mealy* machines issue an *output* when treating an *input*, therefore we define *mealy* $:= \langle Q, \Sigma, \Omega, \delta, \omega, q_0 \rangle$, where $\text{DSM}_{mealy} := \langle Q, \Sigma, \delta, q_0 \rangle$ is its internal SM, $\Omega$ a (finite) *output-alphabet* and $\omega : Q \times \Sigma \to \Omega$ its *output-function*. Both functions $\delta$ and $\omega$ can be extended as before to use string domains and sets of states, yielding $\hat{\delta} : \mathcal{P}(Q) \times \Sigma^* \to \mathcal{P}(Q)$ and $\hat{\omega} : \mathcal{P}(Q) \times \Sigma^* \to \mathcal{P}(\Omega)$ with $\omega(x) = \emptyset \Leftrightarrow x \in \{(q, \varepsilon) : q \in \mathcal{P}(Q)\} \cup \{(\emptyset, w) : w \in \Sigma^*\}$, which allows us to talk about *reachable output* symbols.

The arrows $\langle f_1, f_2, f_3 \rangle : m \to m'$ between Mealy machines are triples, extending the arrows $\langle f_1, f_2 \rangle$ between the underlying DSMs, which are pairs, with a mapping $f_3$, so that these three functions work together, meaning that the following diagram has to commute

$$\begin{array}{ccc} Q \times \Sigma & \xrightarrow{\omega} & \Omega \\ {\scriptstyle f_1 \times f_2}\downarrow & & \downarrow{\scriptstyle f_3} \\ Q' \times \Sigma' & \xrightarrow{\omega'} & \Omega' \end{array} \qquad \begin{array}{l} f_1 : Q \to Q'; \quad f_2 : \Sigma \to \Sigma'; \quad f_3 : \Omega \to \Omega'; \\ f_1 \times f_2 : \langle q, a \rangle \mapsto \langle f_1(q), f_2(a) \rangle; \\ f_3 \circ \omega = \omega' \circ (f_1 \times f_2). \end{array}$$

Thus, we get a *category* **DMealy** of deterministic Mealy machines by defining the composition of Mealy-arrows component–wise. From this it follows, that Mealy machines are *isomorphic*, iff there is an arrow $j = \langle j_1, j_2, j_3 \rangle : m \to m'$ between them, that is an isomorphism between their underlying DSMs $\langle j_1, j_2 \rangle : m \to m'$ so that $j_3$ is invertible and $j^{-1} = \langle j_1^{-1}, j_2^{-1}, j_3^{-1} \rangle : m' \to m$ is another Mealy-arrow.

Although we mentioned *finiteness* of $Q$, $\Sigma$, $\Omega$ and the construction of $m$ using $\delta$ in the definitions above, it is not needed for defining a category. Dropping any of these finiteness conditions will not change that all previous diagrams still commute and form categories, albeit of course different ones. The very same formal abstraction $m : \Sigma^* \to Q$ can be used for any kind of computation, including FSMs and Turing–complete code, bringing them together inside the same categories[4] and we can use this to compare their behaviour using Mealy-arrows and diagrams. [5] — We will come back to all this, when we discuss Pseudo–FSMs and the influence of data.

To construct a category **NSM** of *non–deterministic* SMs, we essentially repeat the constructions above, but with the proviso that $\delta$ and $\omega$ get a different codomain.

An NSM $nsm := \langle Q, \Sigma, \delta, q_0 \rangle$ is a quadruple, where $Q$ is a (finite) set of *states*, $\Sigma$ a (finite) input *alphabet*, $\delta : Q \times \Sigma \to \mathcal{P}_+(Q)$ the *transition-function*, $q_0 \in Q$ an *initial state*.
$\tilde{\delta} : Q \times \Sigma^* \to \mathcal{P}(Q), \quad \tilde{\delta}(q, \varepsilon) := \{q\}, \quad \tilde{\delta}(q, u{\cdot}a) := \bigcup_{q' \in \tilde{\delta}(q,u)} \delta(q', a);$
$\hat{\delta} : \mathcal{P}(Q) \times \Sigma^* \to \mathcal{P}(Q), \quad \hat{\delta}(\emptyset, w) := \emptyset, \quad \hat{\delta}(R, w) := \bigcup_{q \in R} \tilde{\delta}(q, w).$
Again we may write $\delta$ for any of the three functions $\delta$, $\tilde{\delta}$ and $\hat{\delta}$, because the context usually makes clear what is meant.

Since $q_0$ is fixed for $nsm = \langle Q, \Sigma, \delta, q_0 \rangle$, *nsm* does correspond to a function $m : \Sigma^* \to \mathcal{P}_+(Q); \quad m(\varepsilon) = \{q_0\}; \quad m(v \cdot a) = \hat{\delta}(m(v), a)$; The *objects* of **NSM** are such $m$ and the *morphisms* $f : m \to m'$ are pairs of functions $f = \langle f_1, f_2 \rangle$, that make it form a *pointwise subset* relation as shown in Fig. 3. Here we have relaxed the requirement

$$\begin{array}{ccc} \Sigma^* & \xrightarrow{m} & \mathcal{P}_+(Q) \\ {\scriptstyle f_2^*}\downarrow & & \downarrow{\scriptstyle \mathcal{P}_+(f_1)} \\ \Sigma'^* & \xrightarrow{m'} & \mathcal{P}_+(Q') \end{array} \qquad \begin{array}{l} f_1 : Q \to Q'; \quad f_2 : \Sigma \to \Sigma'; \\ \forall w \in \Sigma^* : (\mathcal{P}(f_1) \circ m)(w) \subset (m' \circ f_2^*)(w). \end{array}$$

**Figure 3: *NSM morphisms*** $\langle f_1, f_2 \rangle$

of a commuting diagram (*i.e.*, function equality) to the subset relation $\subset$, which results in more arrows, while still encompassing equality. These additional arrows do also represent relations between machines, but now we can also compare finer models to coarser ones. We say $m$ *complies* with $m'$ (via morphism $f$).

Let us emphasise that the diagram in Fig. 3, which ill be key to understanding the semantics of our more advanced ways of to expand the definitions of McFSMs by the plugin interface introduced in Sec. 6.

The lower arrow, that is $m'$, represents the finite, non–deterministic compiletime model of some McFSM member and the upper arrow, that is $m$, represents the same member at runtime including the influence of data, which is deterministic, but potentially infinite. The vertical arrows show how our interfaces constrain this relation. We do not allow arbitrary $m$, but only those that *comply* with the compiletime description $m'$, which means that for any sequence of events $w$ the resulting state $m(w)$ has to fit to one of those purported at compiletime $m'(w)$. — This diagram is the reason, why infinities in our models do not cause any trouble and our *reasoning* can be confined to finite compiletime models! — It does also show how we use other (novel) means to separate wheat from

[4] Turing–complete code may employ infinite Q, FSMs only finite Q. That both can be viewed as objects in the same category does not imply that they are similar in computing power. FSMs can be seen to form subcategories inside these bigger categories, but we do not elaborate this in detail.

[5] We can even view FAs as objects in these categories, namely as $a : \Sigma^* \to 2$. The simplest non–trivial $Q$ is $2 = \{0, 1\} \cong \{reject, accept\}$. The *accepted* language is $L(a) := a^{-1}(\{accept\})$. Usually $a = \alpha \circ m$ with some $\alpha : Q \to 2$.

DRAFT

chaff than FAs do: It is not the "language recognised" and the internal make–up of $m$ that decide, but the *compliance* with a given (compiletime) model $m'$.

Analogous to the Mealy constructions above we get *NMealy* machines $nmealy := \langle Q, \Sigma, \Omega, \delta, \omega, q_0 \rangle$, where we have an underlying $\text{NSM}_{nmealy} := \langle Q, \Sigma, \delta, q_0 \rangle$ and a finite *output-alphabet* $\Omega$.

To define its *output-function* $\omega$, let $\Delta$ represent the set of (hyper)-edges in the corresponding graph of $\delta$, to ensure that $\omega$ can be used exactly where $\delta$ is defined. We first set $\Delta_1 := \{(q, e, q') : q \in Q;\ e \in \Sigma;\ q' \in \delta(q, e)\}$; $\Delta_2 := \{(q, \varepsilon, q) : q \in Q\}$; $\Delta := \Delta_1 \cup \Delta_2$ and then any $\omega : \Delta \to \mathcal{P}(\Omega)$ with $\omega(x) = \emptyset \Leftrightarrow x \in \Delta_2$. This ensures that $\delta$ and $\omega$ are in sync. The diagram in Fig. 4 has to commute.

$$\begin{array}{ccc} \Delta & \xrightarrow{\omega} & \mathcal{P}(\Omega) \\ {\scriptstyle f_3}\downarrow & & \downarrow{\scriptstyle \mathcal{P}(f_1)} \\ \Delta' & \xrightarrow{\omega'} & \mathcal{P}(\Omega') \end{array}$$

$f_1 : Q \to Q';\ f_2 : \Sigma \to \Sigma';\ f_3 : \Delta \to \Delta',$
$f_3(q, e, q') = (f_1(q), f_2(e), f_1(q'))$ on $\Delta_1$,
$f_3(q, \varepsilon, q) = (f_1(q), \varepsilon, f_1(q))$ on $\Delta_2$.

**Figure 4: $\omega$-*part of NMealy-morphisms***

Similar to $\delta$ we can extend $\omega$ to strings with $\tilde{\omega} : Q \times \Sigma^* \to \mathcal{P}(\Omega)$ using $\tilde{\omega}(q, \varepsilon) := \omega(q, \varepsilon, q) = \emptyset$ and

$$\tilde{\omega}(q, u \cdot a) := \bigcup_{q' \in \tilde{\omega}(q,u),\ q'' \in \delta(q',a)} \{\omega(q', a, q'')\}.$$

An extension to sets of states is given by $\hat{\omega} : \mathcal{P}(Q) \times \Sigma^* \to \mathcal{P}(\Omega)$ with $\hat{\omega}(\emptyset, w) := \emptyset$ and $\hat{\omega}(R, w) := \bigcup_{q \in R} \tilde{\omega}(q, w)$.

Again $q_0$ is fixed for *nmealy*, and we may combine $\delta$ and $\hat{\omega}$ so that *nmealy* corresponds to a function $m : \Sigma^* \to \mathcal{P}_+(Q) \times \mathcal{P}(\Omega)$; $m(w) = (\delta(q_0, w), \hat{\omega}(q_0, w))$. These $m$, with the aforementioned constraints, constitute the objects of our **NMealy** category. The functions and constraints of Fig. 3 and Fig. 4 together define its morphisms $f = \langle f_1, f_2, f_3 \rangle$. The identities are $id_m = \langle id_Q, id_\Sigma, id_\Delta \rangle$ and arrow composition is done component–wise $g \circ f := \langle g_1 \circ f_1, g_2 \circ f_2, g_3 \circ f_3 \rangle$.

We know deterministic models of computation are a special case of non–deterministic ones and unsurprisingly we can find a *full subcategory*[6] isomorphic to **DSM** inside **NSM**. Given an object $m : \Sigma^* \to Q$ in **DSM** we can identify it with the object $(\iota_Q \circ m) : \Sigma^* \to \mathcal{P}_+(Q)$ in **NSM**. And the arrows can be mapped likewise. For the opposite direction, we can translate any object $m$ with $\forall q, a : |\delta(q, a)| = 1$ back into an object in **DSM** and any arrow between such objects as well. All in all, this means we also have a full subcategory isomorphic to **DMealy** inside **NMealy**. — Therefore for our purposes **NMealy** is usually all we need.

Now let's get back from mathematics to software constructions.

## 5 LANGUAGE SPECIFICATION & FORMALISM

Utilising our formalism in practical applications should be possible with minimal changes to accustomed workflows, including the creation and manipulation of programs in text-based representation. Consequently, we present a domain specific language (DSL) to describe FSMs—Mealy-machines, McFSMs and Pseudo–FSMs—in a unified framework. The basic syntactic elements of our DSL are: strings, patterns, lists, names, events, states, transitions, FSM classes, McFSM classes, FSM instances, McFSM instances, Plugins, Pseudo–FSM instances, guarantees, pragmas and functions that describe and connect these elements. A complete language specification is available on the companion website (hyperlink available in PDF); we omit some details in the following discussion.

FSMs and McFSMs need only specify their interface in the DSL to get a working implementation, but plugins and Pseudo–FSMs have to use the plugin–interface to provide technical information on how to integrate these external components into the system and the DSL and sufficient information for reasoning and static analysis. In a tight question and answer game between both, the plugin–interface on the other hand does provide information about the context in the DSL the plugin is getting used and some information of its fellow members and the current McFSM.

We provide a reference implementation in Tcl [49]. Consequently, the DSL is inspired by some of Tcl's features and peculiarities.[7]. We do not assume familiarity with Tcl to understand the DSL, albeit the implicit availability of helper functions from the Tcl standard library might be of service for McFSM development.

Recall the example given in Fig. 2, which we used to detail syntactic elements of the DSL. All member- and element-names are *relative* to the current FSM. From the point of view of some member *Inst*, one of its elements *elem* is referred to, by simply using its name *elem*. From the point of view of the current top-McFSM "/" or some sibling member *Sibling*, the same element has the name *Inst*/*elem*.[8] The names of elements of the current top-FSM "/" are prefixed with '/'. This can be observed in the `oconnect` lines that select output–events for the McFSM from its members.

We obey a strict rule of *locality*: Every member may *subscribe* to any output of any member and use this to adapt its own behaviour. Changing the behaviour of any other FSM is impossible. This is, of course, closely related to the *publish-subscribe* pattern [44], where the publishing FSMs do not need to know anything about its subscribing siblings. As a result, the functionality of each member can be understood by observing its code, together with its surrounding McFSM.

To demonstrate how plugins and pseudo–FSMs blend into the DSL Fig. 5 and 6 show an excerpt of a McFSM application in a real industrial system that used hand gestures to control stepper motors (switching, change rotation direction and speed), while observing temperature constraints from heat sensors (the motors can overheat if direction or speed are adjusted too quickly). The code in Fig. 5 contains `package require` calls. These import code from files that employ the plugin interface. Line 14 uses *j-event* syntax, which defines `j0k` as an abbreviation for all the input–events following after the colon. There is a similar syntax for output–events, namely the *u-events* (line #27 in Fig. 6). Lines 7, 20, 23 copy the

[6] *subcategory*: Every object/morphism in **DSM** is an object/morphism in **NSM**. *full*: There are no further arrows between **DSM** objects in **NSM**.

[7] In particular, we inherit the following conventions: (1) variable names are arbitrary strings and their values are denoted by prepending their name with a '$', (2) lists are written using curly braces starting on the same line as the list elements, separating elements with white–space (setting a list-variable *myList* is done by `set myList {a b c};`). (3) Function calls employ square brackets to delimit arguments.

[8] We decided against the seemingly more natural notation *../Inst/elem* that resembles file-system syntax, and the use of hierarchies with more than two levels. Thus, we have exactly 2 levels in a McFSM: *top* or some member *mem*, which result in these relative calls: (1) *local* views onto itself, $top \rightsquigarrow top$, $mem \rightsquigarrow mem$, and (2) views onto *others*, $top \rightsquigarrow mem$, $mem \rightsquigarrow top$, $mem_1 \rightsquigarrow mem_2$.

DRAFT

```
1  package require dsl::fqueue 0.1
2  package require dsl::counter 0.1
3  package require dsl::smartactor 0.5
4  package require dsl::stupidactor 0.3
5  set LR  {Left Right}
6  set LRS {Left Right Stop}
7  set lrs $LRS
8  FSM class Rotation {
9      hop %lrs_%LRS   i%LRS -> o%LRS
10 }
11 set ERR {TooHot}
12 set LH  {Low High}
13 FSM class HeatSensor {
14     In: i%ERR {jOk: i%LH}
15     hop %ERR_ok     jOk -> oOk
16     hop *_%ERR      i%ERR -> o%ERR
17 }
18 PRAGMA layout HeatSensor -colors {ok black  * red}
19 set OO {On Off}
20 set oo $OO
21 FSM class Switch {
22     hop %oo_%OO   i%OO -> o%OO
23 }
24 PRAGMA layout Switch -colors {off black  * violet}
25 set SMF {Slow Medium Fast}
26 set smf $SMF
27 FSM class SpeedGear {
28     hop  %smf_%SMF  i%SMF -> o%SMF
29 }
30 PRAGMA layout SpeedGear -colors {slow black  m* orange  * violet}
```

**Figure 5: Excerpt of an industrial motion controller (part 1, elementary components).**

values of some list to another list variable. This is used in lines 9, 22, 28 to make the xvar resolution algorithm generate all combinations of the respective list values as transitions—the impicit iteration ranges over two formal variables in these cases.

The PRAGMA lines provide *hints* on visualisation. Such meta-information is passed to generated target language classes, but can also be used at compiletime. For instance, Sec. 6 discusses how worst-case execution time estimates are be handled this way.

Class MotionController, shown in Fig. 6 is the main class. It contains members Rotation, Speed, HS, OnOff, Buf, Count, Stupid, Smart, the order of which gives the (default) event distribution order. Rotation, Speed, HS, OnOff are instances of regular FSMs, Buf, Count are pseudo–FSMs and Smart and Stupid are plugins that represent certain engines to be controlled. The functions in lines 26 (config), 28 (push), 29 (pop), 39 (someEvent), 40 (onOff), 45 (onOff), 46 (speed) are provided via the plugin–interface. They connect the global, superordinate structure handled by the McFSM with implementation details like issuing instructions to physical devices which requires interaction with libraries and the operating system, but does *not* need to be aware of the larger system architecture, not unlike microservices. Buf is a pseudo–FSM that queues up to a maximum number of events and re–emits them after a short delay, thus ensuring that these events do not come in too quick succession (line 6). The appliance includes two stepper motors as actors: A smart device that allows for directly setting a desired rotation speed, and a legacy device for which the mechanism must send explicit stepper pulses dispatched from timer events.

Each FSM has an *initial-state* $q_0$. For a McFSM class the initial-state is therefore the cross-product of initial-states of its members.

```
1  McFSM class MotionController {
2      In:   i%LR i%SMF i%OO i%ERR i%LH tTime0 tTime1 tTick
3      Out:  oOk oOff oErr
4      Rotation inst Rot {
5          initialState stop
6          sconnect i%LRS <- Buf/o%LRS
7          iconnect iStop <- /tTime*
8          when oLeft  call emitT(/tTime0,3sec,/i%LRS)
9          when oRight call emitT(/tTime1,3sec,/i%LRS)
10     }
11     SpeedGear inst Speed {
12         initialState  slow
13         iconnect i%SMF <- /i%SMF
14     }
15     HeatSensor inst HS {
16         initialState  ok
17         iconnect  i%ERR  <- /i%ERR  and  i%LH <- /i%LH
18     }
19     Switch inst OnOff {
20         initialState  on
21         iconnect i%OO  <- /i%OO
22         sconnect iOff  <-  HS/o%ERR
23     }
24     # Slow down event-bursts by buffering and re-emitting
25     PsFQueue inst Buf {
26         config max 4
27         Out: oUnderflow uIncr oOverflow {uDecr: o%LRS}
28         push o%LRS     <- i%LRS
29         pop            <- iTick
30         iconnect i%LRS  <- /i%LRS
31         iconnect iTick  <- /tTick
32         when oIncr1 oOverflow uDecr call emitT(/tTick,0.2sec)
33     }
34     PsCount32 inst Count {
35         add 1          <- iI1
36         iconnect iI1   <- /i%OO  /i%ERR
37     }
38     PxStupidActor inst Legacy {
39         someEvent      <- iTick
40         onOff          <- i%OO
41         sconnect i%OO  <- OnOff/o%OO
42         iconnect iTick <- /i%LRS /i%SMF
43     }
44     PxSmartActor inst Smart {
45         onOff          <- i%OO
46         speed          <- i%SMF
47         sconnect i%OO  <- OnOff/o%OO
48         sconnect i%SMF <- Speed/o%SMF
49     }
50     oconnect default /oOk
51     oconnect /oErr     <- HS/o%ERR
52     oconnect /oOff     <- OnOff/oOff
53 }
```

**Figure 6: Excerpt of an industrial motion controller (part 2, coupling between machines).**

A member inherits the initial-state from its class, but may change it to some other state that is reachable from there. A simple FSM instance can set it with function initialState and a McFSM instance can use an initialSequence of events. This sequence is a purely logical statement that gets resolved at compiletime, omitting thereby any side-effects like when statements.

## 6 PSEUDO-FSM AND PLUGIN

### 6.1 Construction

A McFSM contains a sequence of *members* $m_1, \ldots, m_n$. These may be *instances* of Mealy machines that have been defined using FSM

DRAFT

`class` or `McFSM class`, or be foreign code that assures to behave like a Mealy machine, that is a *Pseudo–FSM*, or a *Plugin*.

Both, Pseudo–FSMs and Plugins, are foreign code, and must therefore provide information about their runtime behaviour, and specify guarantees that can be used to consider their behaviour in static analyses. The respective interfaces are almost identical; the difference is mostly in how their runtime code behaves.

Pseudo–FSMs implement a non–deterministic Mealy machine (satisfying category **NMEALY** in defined in Sec. 4). Presented with an input–event, they proceed to an next internal state, and return an output–event. Their code is executed in the same thread as the implementation of the McFSM proper.

Plugins run code that can cause side-effects, and may delegate the actual work into a separate thread from the McFSM dispatcher. Plugins are not required to internally rely on states; the resulting **NMEALY** instance if typically trivial. Regardless of the type of a McFSM member $m_i$, reasoning requires at least:

**Possible Output.** The set of possible output–events for a given input–event, which is a mapping $p_i : E_i \to \mathcal{P}(O_i)$, where each $o_k \in p_i(e_{i,j})$ is reachable from $e_{i,j} \in E_i$.

**Cost estimate.** An estimate how much time is required to perform a mapping $p_i$ in the worst-case, which is given by another mapping $u_i : E_i \times p_i(E_i) \to (\mathbb{R} \times U)^+$, where $U \subset \Sigma^*$ gives the units of measurement and $(\mathbb{R} \times U)^+$ is a set of aggregates, that is, a symbolic sum of products $value \cdot unit$, with $value \in \mathbb{R}$ and $unit \in \Sigma^*$. The mapping assigns to each pair of input- and possible output–event a cost given by a *symbolic* expression.

Plugins can provide the cost of their operations in different units. Given *base units* $B \subset U$, then a global mapping $\psi : B \to (\mathbb{R} \times U)^+$, which ensures that there are no cycles in the resulting exchange rates, converts between units. This is not just to give a more convenient way of providing temporal durations: By varying $\psi$, we are able to gauge costs with respect to varying target platforms (computers, programming languages, libraries, …).

More importantly, the symbolic expressions can be added up along the event-distribution chains of any McFSM, and then be evaluated with respect to any given set of mappings $\psi$. If we use, for instance, functions that provide lower and upper bounds, this results in cost-range expressions $[from \ldots to]$ for each input–event $e_{i,j}$ of each $m_i$, and make sure these stay in an acceptable range. The utility for real-time and safety critical workloads is evident.

## 6.2 Handling external data

Input and output events of the McFSM formalism are represented in the DSL by symbols (names) and at runtime by objects of the generated target language code. So a single symbol in the DSL does correspond to a class for runtime event objects and eventually to an arbitrary number of runtime objects of this class. As long as these objects can be seen as being identical *w.r.t.* the McFSM we have a logical one–to–one correspondence between a DSL event symbol and its class of runtime event objects. But we do allow these objects to carry arbitrary ("external") *data* besides the information required for the McFSM base system. Of course, data are inherently runtime information and not accessible at compiletime. This is necessary to interact with external components outside the McFSM formalism, for instance via system or API calls that may eventually turn into side effects like reading data from sensors, displaying messages, etc.

Pure McFSMs do not consider data—from their perspective, events are members of the finite set of symbols—, but data can be utilised by Pseudo–FSM members. This changes the game: When a pseudo–FSM gets influenced by external data, it may yield different output events, even when confronted with a sequence of input events corresponding to an identical sequence of input symbols. Events are atomic without external data, but receive an additional internal structure in their presence, thus they have to be considered different if they carry different data.

To not unnecessarily restrict the interaction with external components, there is no limit on the amount of data that may accompany an event. This is equivalent to considering an *infinite* number of events that may hide behind each event symbol. Consequently, $\Sigma$ at runtime can no longer be regarded as finite during compiletime. A similar argument holds for the set of states $Q$, since a Pseudo–FSM may keep track of past events including external data, and base decisions on this history. For example, consider a temperature measurement initiated by the input event `iMeasurement`. The data consists of the temperature value, and the output event – one of `oFalling`, `oConst`, `oRising` – woulld typically depend on some past `iMeasurement` events and their values, probably to perform averaging, or to include a hysteresis.

Output events may also carry data, and a Pseudo–FSM can pass data or calculation results along, which means $\Omega$ can no longer be considered finite. The semantics of the plugin–interface need to account for effects of infinite data, without introducing well-known issues from infinite-state mechanisms. For instance, a state machine given by an infinite branching tree could recognise *any* language [4], effectively surpassing the capabilities of Turing machines. Since, however, such a model of computation could also be fit into our categorical abstraction, we need to ascertain the absence of such troublesome constructs by additional criteria imposed by the plugin–interface.

Recall the structural relationship of two NMealy objects as given by commuting diagrams in Figures 3 and 4: If Pseudo–FSMs satisfy these relationships—most importantly, if they satisfy them even in the presence of infinite external data—, they can still be treated like pure McFSMs. In other words, the diagrams provide criteria that allow McFSMs to enjoy the benefits of extensions that use Turing-complete external code, and interact with arbitrary amount of external data, without sacrificing any of the guarantees that static analysis and reasoning techniques can provide.

$$\begin{array}{ccc} \Sigma^* & \xrightarrow{m} & \mathcal{P}_+(Q) \\ {\scriptstyle f_2^*}\downarrow & & \downarrow{\scriptstyle \mathcal{P}_+(f_1)} \\ \Sigma'^* & \xrightarrow{m'} & \mathcal{P}_+(Q') \end{array} \qquad \begin{array}{ccc} \Delta & \xrightarrow{\omega} & \mathcal{P}(\Omega) \\ {\scriptstyle f_3}\downarrow & & \downarrow{\scriptstyle \mathcal{P}(f_1)} \\ \Delta' & \xrightarrow{\omega'} & \mathcal{P}(\Omega') \end{array}$$

More formally, consider a Pseudo–FSM $P$, its two corresponding NMealy objects $p = (m, \omega)$ and $p' = (m', \omega')$, and their relation $f = \langle f_1, f_2, f_3 \rangle$ as introduced in Figs. 4 and 3—for convenience, a side-by-side presentation is given above.

The top arrows $p$ of the diagrams represent the deterministic *runtime* behaviour of $P$ and the bottom arrows $p'$, represent the non–deterministic *compiletime* model of $P$.

DRAFT

The vertical arrows $f$ constitute a *contract* that $p$ must satisfy to comply with $p'$. From the perspective of our framework (and any static analyses based on it), only $p'$ is visible.

Infinite sets $Q, \Sigma, \Omega$ may appear in $p$, but $p$ must comply to $p'$, which is non–deterministic but based on *finite* sets $Q', \Sigma'$, and $\Omega'$. From the perspective of $P$, the runtime behaviour $p$ of $p'$ is *deterministic*, because it can access the structure of data inside events.

The categorial contract, together with the additional criteria defined above, imply some points that are relevant for the practical implementation. Intuitively speaking, they ensure $P$ does not "stray" from what is expected of an FSM with respect to *time* and *memory* consumption. More concretely, (1) all necessary memory allocations have to happen at initialisation time (no further runtime calls to services the provide dynamic memory must occur in $P$, thus the amount of *memory* the McFSM itself holds does not grow. All other runtime allocations have to happen outside of $P$ and its McFSM), and (2) each state transition must obey strict temporal constraints. Relinquishing the global finiteness requirements for $Q$, $\Sigma$ and $\Omega$ (which concern space complexity) creates the requirement of *pointwise* finiteness of evaluation time ($m(w)$).

## 6.3 Examples

Let's have a look at the plugin–interface by using as example the specification of a Pseudo–FSM `PsFStack` that implements a finite stack with *max* as maximum number of elements.

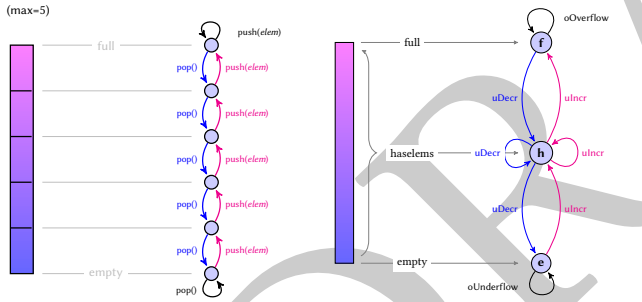


**Figure 7: Deterministic Mealy machine that implements a finite stack (left hand side), and the corresponding non–deterministic plugin-abstraction.**

The left side of Figure 7 visually depicts of the standard understanding of a stack, whereas the right side shows a *non–deterministic* automaton that captures the essential behaviour of the stack, as it is relevant for static analysis. It corresponds to $p' = (m', \omega')$ as discussed in the previous section, and we refer to it as *plugin-abstraction* in a concrete instance. It would, for instance, be identical for a finite queue instead of a finite stack, since both exhibit the same abstract characteristics as far as our reasoning goes.

Output-events with prefix 'o' (`oOverflow`, `oUnderflow`) denote a single event with this name, while the prefix 'u' (`uIncr`, `uDecr`) is used for a name of a set of output–events, but which can nevertheless be used in `sconnect` and `oconnect` calls. The symbol `uIncr` stands for the set $\{\texttt{oIncr}_1, \ldots, \texttt{oIncr}_{max}\}$ and `uDecr` for one of the pushed output-symbols.

Also, these pictures show how the *class* behaves. An *instance* does have a concrete set of output-symbols $O$ to store and use in `uDecr`. Its number of states amounts to $|O|^{max}$, corresponding to the number of different `push(`*elem*`)` call-sequences.

The Pseudo–FSM- and plugin–interface consists of a number of classes and interfaces that must be implemented and obeyed. Firstly, we require an object *ifsm* of class `IFSM`, which describes the behaviour of class `PsFStack`. Fig. 8 shows an example of a finite stack implementation. Function `getPluginDict` returns the non–deterministic automaton corresponding to the right side of Fig. 7. `getSlotList` provides a list of possible slots, that is, methods that are offered to the outside world.

```
1  set ifsm [IFSM new PsFStack]
2  oo::objdefine $ifsm method getSlotList {} { list push pop top }
3  oo::objdefine $ifsm method getPluginDict {} {
4      return {
5          full_full          {push oOverflow  top uSame}
6          full_haselems      {                          pop uDecr}
7          haselems_full      {push uIncr}
8          haselems_haselems  {push uIncr      top uSame   pop uDecr}
9          haselems_empty     {                          pop uDecr}
10         empty_haselems     {push uIncr}
11         empty_empty        {           top oEmpty  pop oUnderflow}
12     }
13 }
```

**Figure 8: Essentials of an implementation of a finite stack that satisfies the plugin abstraction of Figure 7.**

Secondly, we need a class for instances of finite stacks, which we call `InstIFSM(PsFStack)`. it is derived from class `InstIFSM`, so that every PsFStack-instance *DSL:iifsm* receives a so-called describing object—each of these objects describes a *class* given in the DSL, because the McFSM as a whole translates to a class in the target language, which can be instantiated by the user.

Objects of this class implement how instances interpret their DSL-*body* (*i.e.*, the code for each instance between curly braces). This includes setting `config` parameters, refining 'u'-symbols (as discussed, for the example, in Fig. 5), and their plugin-abstraction. Class `InstIFSM(PsFStack)` also provides methods for each *slot*—`slot_push`, `slot_pop`, `slot_top` in case of the finite stack.

The compiler extracts the *signature* of the slot-methods by introspection into the class `InstIFSM(PsFStack)`. This way, each slot may have a different signature and the compiler can still understand this and call the slot-method to interpret eventual slot-calls in the body. Each slot may even invent its own special syntax that is best suited to provide a readable representation for its purpose, because our compiler only passes them along, without interpretation. To give a flavour of how methods in `InstIFSM(PsFStack)` are implemented, consider Fig. 9.

Without discussing implementation details—refer to the documentation on the companion website—, observe that only little code is required. Method `addSC` is a helper methods provided by superclass `InstIFSM` registering the slot-call information with the compiler. `pushSyms` is an instance variable of class `InstIFSM(PsFStack)`, which accumulates the list of symbols that are actually used for this instance. This can be used to refine the actions of the class (details on the companion website).

DRAFT

```
oo::define InstIFSM(PsFStack) method getK3refinement {} {
    set uIncr [list oIncr1 oIncr2 oIncr]
    list uDecr $pushSyms  uIncr $uIncr  uSame $pushSyms
}
oo::define InstIFSM(PsFStack) method slot_pop {nrflag paraLi sfsm ssym} {
    if {[llength $paraLi]} { error "unexpected parameter: $paraLi" }
    my addSC [list pop $paraLi] $nrflag $sfsm $ssym
}
```

**Figure 9: Except of some InstIFSM(PsFStack) methods.**

Thirdly and finally, code needed by a class `PsFStack` in every target language that implements the actual finite stack.

By connecting the McFSM formalism and Turing-complete code, Pseudo–FSMs extend the expressiveness, while retaining advantages of finiteness. A standard library includes Pseudo–FSMs for common algorithmic components.

Conjunctively connecting states of different members of a McFSM is often needed, but hard to accomplish—it involves creating an FSM model of the product, which invites state space explosion. A dedicated pseudo–FSM, `PsAnd`, uses introspection on siblings to simplify the task. It provides logical conjunctions *across* several sibling members. Each FSM provides a number of read-only methods (*ROMs*) that may be called to inspect a FSM without changing its state. Pseudo–FSMs are allowed to use ROMs of siblings to make their decisions. The plugin–interface contains the `getROMs` and `getROMcalls`. Each Pseudo–FSM returns, using function `getROMs`, the names of ROMs that other instances may use. Function `getROMcalls` provides the names of siblings and their ROMs. The compiler can ensure absence of *cycles* appearing in ROM calls, and calculate the corresponding costs $u_i$. This ensures FSM behaviour, even with almost arbitrary function calls among the members.

Each FSM remembers, among other things, the last-output–event (*LOE*) it has returned and requesting this information is such a ROM. The PsAnd has only states `true` and `false` and output–events `oTrue` and `oFalse` and only one input–event `iEval`.

```
PsAnd inst And {
    eval Rot/loe=oLeft Rot/loe=oRight  OnOff/loe=oOn <- iEval
    iconnect iEval  <- /i*
}
```

**Figure 10: Example illustrating the use of the Pseudo–FSM "PsAnd" to compute logical conjunctions.**

Consider Fig. 10 for an exemplary use of a `PsAnd` instance. It computes an output by getting the LOE of the siblings `Rot` and `OnOff`. For the same sibling, the given values form a set whose elements are understood as "OR" connected, while the expressions for different siblings are "AND" connected. The `eval`-expression in Fig. 10 thus returns `oTrue`, iff $LOE_{\texttt{Rot}} \in \{\texttt{oLeft}, \texttt{oRight}\} \wedge LOE_{\texttt{OnOff}} \in \{oOn\}$. This enables any sibling to `sconnect` with `And/oTrue`, and perform its action only when the conditions "`Rot` is rotating" (not stopped) and the switch "`OnOff` is on" (not off) are met.

## 7 RELATED WORK

Finite state machines are one of the earliest theoretical concepts in automated computing, and also form the basis of many modern software engineering mechanisms like UML state machines [48]. Code for (real-time) systems can be synthesized from FSM based descriptions (see, e.g., [24, 34]), and a wide selection of current research explores issues related to FSMs that range from testing [1] Verification techniques, static analysis and software model checking have gained considerable interest in industry [12, 18, 19, 53], and is used from web development [50] via business process models [22] to verifying electronic circuit designs [60]. Countless commercial offerings and academic approaches [16, 17, 43] too numerous to exhaustively mention here (Refs. [2, 8] provide comprehensive reviews) are available. However, recent studies show that there remains ample room for improvement [33], especially for industrially relevant systems where the techniques remain underused [39].

McFSMs share the idea of dividing a system into sub-automata with UML hierarchical state machines [48] The concept of emitting signals from edge transitions can, outside the Mealy formalism, be found in statecharts [35] and related formalisms, although these have received criticism for their lack of consistent theoretical underpinnings (see, *e.g.*, [23]). Using FSMs to control the superordinate behaviour of a composite system has been considered for specific targets like Android [28], classes like safety-critical [47] or reactive [45] systems, and as general design pattern [30].

While category theory has received steadily increasing attention in some fields like database research [7], and is considered a prime candidate for abstractly expressing scientific insights [57], its use in software engineering research is still rare. Ref. [29] provides an introduction to category theory with a focus on issues in software engineering; examples of SWE research often stem from model-driven engineering [3, 10, 26, 27]. We are, to the best of our knowledge, not aware of research that bases the implementation of practical tools on a categorical foundation.

## 8 CONCLUSION

We have presented theory and implementation of multiple coupled finite state machines, a mechanism that combines the many convenient properties of finite automata with the power of orthodox programming. The freedom to incorporate data and computation history into the decision process of FAs brings McFSMs close to the power of Turing–complete code, while avoiding countless intricacies of the latter. Our industry-grade, open source IDE can generate code for numerous target languages and platforms, and is suitable for use in real-time and safety-critical systems.

We have designed our mechanism around an elaborate framework based on mathematical category theory which does ensure a solid foundation for reasoning about McFSM programs. To the best of our knowledge, we are not aware of any other descriptive development mechanism that provides well-defined mathematical semantics based on a consistent categorical treatment. We have shown the benefits of using morphisms to form a safe interface bridging Turing-complete code and finite state machines, which links the arguably most advanced and abstract mathematical theory so far with an important problem in software engineering in both, theory and application.

DRAFT

# REFERENCES


[1] R. Ahmadi and J. Dingel. Concolic testing for models of state-based systems. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 4–15, 2019.

[2] W. Ahrendt, M. Huisman, G. Reger, and K. Y. Rozier. In T. Margaria and B. Steffen, editors, *Leveraging Applications of Formal Methods, Verification and Validation. Verification*, pages 3–7, Cham, 2018. Springer International Publishing.

[3] M. Aiguier, F. Boulanger, and B. Kanso. A formal abstract framework for modelling and testing complex software systems. *Theoretical Computer Science*, 455:66–97, 2012.

[4] S. Arora and B. Barak. *Computational Complexity: A Modern Approach*. Cambridge University Press, USA, 1st edition, 2009.

[5] N. Ayewah, W. Pugh, D. Hovemeyer, J. D. Morgenthaler, and J. Penix. Using static analysis to find bugs. *IEEE software*, 25(5):22–29, 2008.

[6] R. Backhouse, R. Crole, and J. Gibbons, editors. *Algebraic and Coalgebraic Methods in the Mathematics of Program Construction*. Springer-Verlag, Berlin, Heidelberg, 2002.

[7] K. Baclawski, D. Simovici, and W. White. A categorical approach to database semantics. *Mathematical Structures in Computer Science*, 4(2):147–183, 1994.

[8] C. Baier and J.-P. Katoen. *Principles of Model Checking (Representation and Mind Series)*. The MIT Press, 2008.

[9] L. Bass, P. Clements, and R. Kazman. *Software Architecture in Practice*. Addison-Wesley Professional, 3rd edition, 2012.

[10] D. Batory, M. Azanza, and J. Saraiva. The objects and arrows of computational design. In K. Czarnecki, I. Ober, J.-M. Bruel, A. Uhl, and M. Völter, editors, *Model Driven Engineering Languages and Systems*, pages 1–20, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg.

[11] G. Berry and G. Gonthier. The esterel synchronous programming language: Design, semantics, implementation. *Science of computer programming*, 19(2):87–152, 1992.

[12] A. Bessey, K. Block, B. Chelf, A. Chou, B. Fulton, S. Hallem, C. Henri-Gros, A. Kamsky, S. McPeak, and D. Engler. A few billion lines of code later: Using static analysis to find bugs in the real world. *Commun. ACM*, 53(2):66âĂŞ75, Feb. 2010.

[13] E. Börger and R. Stärk. *Abstract State Machines: A Method for High-Level System Design and Analysis*. Springer, New York, 2003.

[14] T. Bray. Ietf rfc 8259 the javascript object notation (json) data interchange format, 2017.

[15] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal. *Pattern-Oriented Software Architecture - Volume 1: A System of Patterns*. Wiley Publishing, 1996.

[16] C. Calcagno and D. Distefano. Infer: An automatic program verifier for memory safety of c programs. In M. Bobaru, K. Havelund, G. J. Holzmann, and R. Joshi, editors, *NASA Formal Methods*, pages 459–465, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg.

[17] C. Calcagno, D. Distefano, J. Dubreil, D. Gabi, P. Hooimeijer, M. Luca, P. O'Hearn, I. Papakonstantinou, J. Purbrick, and D. Rodriguez. Moving fast with software verification. In K. Havelund, G. Holzmann, and R. Joshi, editors, *NASA Formal Methods*, pages 3–11, Cham, 2015. Springer International Publishing.

[18] R. Castano, V. Braberman, D. Garbervetsky, and S. Uchitel. Model checker execution reports. pages 200–205. IEEE, 2017.

[19] M. Christakis and C. Bird. What developers want and need from program analysis: An empirical study. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*, ASE 2016, page 332âĂŞ343, New York, NY, USA, 2016. Association for Computing Machinery.

[20] E. M. Clarke, T. A. Henzinger, H. Veith, and R. Bloem, editors. *Handbook of Model Checking*. Springer, 2018.

[21] E. M. Clarke, W. Klieber, M. Nováček, and P. Zuliani. Model checking and the state explosion problem. In *LASER Summer School on Software Engineering*, pages 1–30. Springer, 2011.

[22] F. Corradini, F. Fornari, A. Polini, B. Re, and F. Tiezzi. A formal approach to modeling and verification of business process collaborations. *Science of Computer Programming*, 166:35–70, 2018.

[23] P. Derler, E. A. Lee, and A. Sangiovanni-Vincentelli. Modeling cyber-physical systems. *Proceedings of the IEEE*, 100(1):13–28, 2012.

[24] C. Dietrich, M. Hoffmann, and D. Lohmann. Back to the roots: Implementing the rtos as a specialized state machine. *11th WS on Op. Sys. Plat. for Emb. RT Applications (OSPERT)*, pages 7–12, 2015.

[25] A. Diewald, S. Voss, and S. Barner. A lightweight design space exploration and optimization language. *Proceedings of the 19th International Workshop on Software and Compilers for Embedded Systems (SCOPES)*, pages 190–193, 2016.

[26] Z. Diskin and T. Maibaum. Category theory and model-driven engineering: From formal semantics to design patterns and beyond. *Model-Driven Engineering of Information Systems: Principles, Techniques, and Practice*, page 173, 2014.

[27] Z. Diskin, T. Maibaum, and K. Czarnecki. Intermodeling, queries, and kleisli categories. In *International Conference on Fundamental Approaches to Software Engineering*, pages 163–177. Springer, 2012.

[28] D. Dominguez Perez and W. Le. Specifying callback control flow of mobile apps using finite automata. *IEEE Transactions on Software Engineering*, pages 1–1, 2019.

[29] J. L. Fiadeiro. *Categories for Software Engineering*. Springer Publishing Company, Incorporated, 1st edition, 2010.

[30] E. Freeman, E. Freeman, B. Bates, and K. Sierra. *Head First Design Patterns*. OâĂŹReilly & Associates, Inc., 2004.

[31] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design patterns: elements of reusable object-oriented software*. Addison Wesley Longman, Boston, MA, USA, 1995.

[32] E. Gamma, R. Helm, R. Johnson, and J. M. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional, 1 edition, 1994.

[33] A. Habib and M. Pradel. How many of all bugs do we find? a study of static bug detectors. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, ASE 2018, page 317âĂŞ328, New York, NY, USA, 2018. Association for Computing Machinery.

[34] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. The synchronous data flow programming language lustre. *Proceedings of the IEEE*, 79(9):1305–1320, 1991.

[35] D. Harel. Statecharts: A visual formalism for complex systems. *Science of computer programming*, 8(3):231–274, 1987.

[36] J. E. Hopcroft, R. Motwani, and J. D. Ullman. Introduction to automata theory, languages, and computation. *Acm Sigact News*, 32(1):60–65, 2001.

[37] J. E. Hopcroft and J. D. Ullman. *Formal Languages and Their Relation to Automata*. Addison-Wesley Longman Publishing Co., Inc., USA, 1969.

[38] M. Huth and M. Ryan. *Logic in computer science - modelling and reasoning about systems*. Cambridge University Press, 01 2000.

[39] B. Johnson, Y. Song, E. Murphy-Hill, and R. Bowdidge. Why don't software developers use static analysis tools to find bugs? In *2013 35th International Conference on Software Engineering (ICSE)*, pages 672–681. IEEE, 2013.

[40] R. Kamal. Protocol buffers v3.11.4, 2020.

[41] S. Lane. *Categories for the Working Mathematician*. Graduate Texts in Mathematics. Springer New York, 1998.

[42] D. Lee and M. Yannakakis. Principles and methods of testing finite state machines-a survey. *Proceedings of the IEEE*, 84(8):1090–1123, 1996.

[43] J. Magee and J. Kramer. *Concurrency: State Models & Java Programs*. John Wiley & Sons, New York, 1999.

[44] I. Maier and M. Odersky. Deprecating the observer pattern with scala.react. page 20, 2012.

[45] Z. Manna and A. Pnueli. *The temporal logic of reactive and concurrent systems: Specification*. Springer Science & Business Media, 2012.

[46] G. H. Mealy. A method for synthesizing sequential circuits. *The Bell System Technical Journal*, 34(5):1045–1079, 1955.

[47] F. Murr and W. Mauerer. Mcfsm: globally taming complex systems. In *Proceedings of the 3rd International Workshop on Software Engineering for Smart Cyber-Physical Systems*, pages 26–29. IEEE Press, 2017.

[48] OMG. Omg unified modeling language superstructure, 02 2009.

[49] J. K. Ousterhout and K. Jones. *Tcl and the Tk Toolkit*. Addison-Wesley Professional Computing Series. Addison-Wesley, Upper Saddle River, NJ, 2 edition, 2009.

[50] C. Park, S. Won, J. Jin, and S. Ryu. Static analysis of javascript web applications in the wild via practical dom modeling (t). In *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 552–562, 2015.

[51] J. C. Reynolds. *Theories of Programming Languages*. Cambridge University Press, 1998.

[52] J. Rumbaugh, I. Jacobson, and G. Booch. *Unified Modeling Language Reference Manual, The (2nd Edition)*. Pearson Higher Education, 2004.

[53] C. Sadowski, E. Aftandilian, A. Eagle, L. Miller-Cushon, and C. Jaspan. Lessons from building static analysis tools at google. *Commun. ACM*, 61(4):58âĂŞ66, Mar. 2018.

[54] M. Samek. *Practical UML statecharts in C/C++*. Newnes/Elsevier, Amsterdam, 2009.

[55] D. Sannella and A. Tarlecki. *Foundations of algebraic specification and formal software development*. Springer Science & Business Media, 2012.

[56] M. Sipser. *Introduction to the Theory of Computation*. International Thomson Publishing, USA, 1996.

[57] D. I. Spivak. *Category theory for the sciences*. MIT Press, 2014.

[58] B. Steffen. Data flow analysis as model checking. In T. Ito and A. R. Meyer, editors, *Theoretical Aspects of Computer Software*, pages 346–364, Berlin, Heidelberg, 1991. Springer Berlin Heidelberg.

[59] M. Vardi. Branching vs. linear time: Final showdown. volume 2031, pages 1–22, 03 2001.

[60] Y. Yang, Y. Jiang, M. Gu, and J. Sun. Verifying simulink stateflow model: Timed automata approach. In *2016 31st IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 852–857, 2016.

[61] I. Zuzak, I. Budiselic, and G. Delac. A finite-state machine approach for modelling and analyzing restful systems. *Journal of Web Engineering*, 10(4):353–390, 2011.


DRAFT