

McFSM: Globally Taming Complex Systems

Florian Murr

Siemens AG, Corporate Research
Otto-Hahn-Ring 6
81739 München
florian.murr@siemens.com

Wolfgang Maurer

Technical University of Applied Sciences Regensburg
Siemens AG, Corporate Research
Regensburg and München
wolfgang.maurer@oth-regensburg.de

Abstract—Industrial computing devices, in particular cyber-physical, real-time and safety-critical systems, focus on reacting to external events and the need to cooperate with other devices to create a functional system. They are often implemented with languages that focus on a simple, local description of how a component reacts to external input data and stimuli. Despite the trend in modern software architectures to structure systems into largely independent components, the remaining interdependencies still create rich behavioural dynamics even for small systems. Standard and industrial programming approaches do usually not model or extensively describe the global properties of an entire system. Although a large number of approaches to solve this dilemma have been suggested, it remains a hard and error-prone task to implement systems with complex interdependencies correctly.

We introduce multiple coupled finite state machines (McFSMs), a novel mechanism that allows us to model and manage such interdependencies. It is based on a consistent, well-structured and simple global description. A sound theoretical foundation is provided, and associated tools allow us to generate efficient low-level code in various programming languages using model-driven techniques. We also present a domain specific language to express McFSMs and their connections to other systems, to model their dynamic behaviour, and to investigate their efficiency and correctness at compile-time.

I. INTRODUCTION

Tackling complex systems usually entails two tasks: Localizing effects to components of a system, and appropriately treating interdependencies between the components. Most theoretical advances in computer science have been focused on the objective of localizing effects, as can be seen in concepts like functional or object-oriented programming that try to limit side-effects to manageable portions of the code, as do common techniques like *information hiding* or *separation of concerns*. Most of these are rooted in a solid theoretical basis. The available options for treating interdependencies in industrial systems are less plentiful.

Such systems are often implemented with mechanisms originating from programmable logic controllers and finite state machines, or by using reactive programming techniques. They rely on languages as defined in the ISO EN 61131-3 standard, or employ patterns similar to these. Finite state machine (FSM)-like approaches, in particular sequential function charts (SFC), are central to algorithms in these domains. FSMs are in widespread use in most areas of computer science: Network protocol dispatchers in operating systems, cryptographic handshakes, formal specifications (like UML) for embedded

systems design [2], formalizations of representational state transfer (REST) based web applications [12], etc. All these and many more are based on FSMs. The theoretical properties of FSMs are extremely well known. Some specialised languages (e.g., [1], [7]), especially reactive languages, are directly based on state machines, as are many other approaches (e.g. recent languages for intuitive robot control [5]). A considerable number of extensions to FSMs, like hierarchical state machines [11], some based on the seminal statechart idea [8], have been suggested.

Albeit libraries to implement FSMs and related approaches are available for most languages, most of them do not provide explicit expressive constructs to describe the global structure of composite systems.¹ Using the state machine pattern [6], FSMs are usually hand-crafted with generic language techniques, for instance using `case` statement dispatchers in C or similar constructs. This way, it is easy to (inadvertently) mix the pure simplicity of the FSM-approach with Turing-complete language features, thus sacrificing compile-time provability. The same observation holds for Turing-complete reactive programming systems.

For distributed systems, the description of the state machine that comprises the system is often spread across multiple physical or virtual machines or at least multiple *local* program components, making it hard to obtain a consistent and coherent picture of the *global* system that is required to guarantee system-wide properties, like overall safe behaviour. The level of diversity in industrial cyber-physical systems is still substantially more pronounced than the diversity in, say, operating systems, tools, programming languages etc. This suggests that despite the previously suggested multitude of approaches, the practical, real-world aspects of the field are far from being solved satisfactorily.

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. The mechanism uses 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

¹We deliberately ignore regular expressions. These are in fact an integral part of many languages and stem from FSM based origins, but are used as acceptors of languages without making the underlying state machines accessible.

interdependencies. As a *low-level* mechanism, it lends itself to an efficient implementation. As a *global superordinate* structure, it avoids the pitfalls of scattered observers acting largely agnostic of their interdependencies. Owing to a *theoretical* foundation, it lends itself well to rigorous scrutiny and formal verification.

McFSMs aim primarily at explicitly describing and handling interdependencies between components at compile-time. They also have a thorough theoretical basis that allows us to prove various system properties. McFSMs can be used to reduce implicit or undesired dependencies between components and foster a consistent and well-structured global description of interdependencies. The approach can also serve as basis for deterministic hard real-time systems, and is therefore applicable to a wide class of industrial systems.

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 [11]. Code for (real-time) systems can be synthesized from FSM based descriptions (see, e.g., [4], [7]). 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, which makes the approach inherently unfeasible for practical software engineering purposes. It creates very dense and confusing diagrams even for systems of moderate complexity, or might use up to much space. Our approach allows to describe industrially relevant coupled systems while avoiding such an exponential “state space explosion”.

II. MODELLING INTERACTING SYSTEMS

Systems based on interacting components, either logical or physical, need to propagate information about state changes between their constituents. When a state change occurs in one part of the system, the change propagates to related components, and may trigger further state changes. This notion is generically captured by the *observer pattern* [6]: An object A (the *subject*) maintains a list of dependent objects (the *observers*) $\{B_1, B_2, \dots, B_n\}$ and notifies them when any state change takes place in object A .

There is growing evidence that the observer pattern is problematic (see, for instance, Ref. [10]). Industrial experience endorses these findings with the observation that modelling even superficially extremely simple systems, for instance displays for multi-function household appliances with a small number of controls, often results in complex and error-prone systems that require substantial implementation and testing efforts when the architecture is based on the observer pattern. The authors are aware of industrial projects where the initial estimated effort of a few weeks resulted in several months worth of implementation effort.

We want to emphasise four major challenges: Firstly, observers promote side-effects since their states are only implicitly available, but not explicitly represented by programming language constructs. Shared states need therefore be made available in a context that is accessible from multiple observers,

or even globally, of course without violating information encapsulation principles. Secondly, observers can execute arbitrary Turing-complete code, which can easily lead to violating the principle of separation of concerns. Thirdly, traditional programming languages make it hard or even impossible to statically analyse and understand the dynamic control flow when chains of dynamically registered observers are used. Finally, any state change apart from the one in subject A triggering the notifications is not part of the observer pattern and is therefore “invisible” from its point of view.

Nonetheless, the observer pattern enjoys wide-spread use in many software systems, and likewise do the very similar publish-subscribe and signal-slot mechanisms. Handling asynchronous interrupts on the system level is also closely related to the described mechanisms.

Possible solutions to the aforementioned problems include the use of reactive programming techniques [10] that require only a specification of dependencies between interacting components, as compared to manual encoding execution flows. However, such techniques often require intensive run-time support and advanced language features that are not available in languages conventionally used for systems programming.

III. MULTIPLE COUPLED FSMS

A. Example and DSL

To facilitate the practical use of McFSMs, we provide a domain-specific language (DSL). An example to illustrate the language is based on an application pattern that occurs frequently in automation and control. Consider a distributed set of binary (on/off) switches that reside at different locations in a plant or a residential building; triggering one of the switches changes the state of one shared dependent entity, perhaps a light bulb (on/off) or a status indicator (green/yellow/red). Even for a system as simple as two binary switches and one ternary status indicator, a straightforward FSM approach leads to $2^2 = 4$ possible combinations of switch settings (off/off, on/off, off/on, and on/on) and three different values for the indicator resulting in $3 \times 4 = 12$ states. For each state, two edges describing the result of flipping the switch are necessary. For the more general case of n switches and an indicator with m levels, the amount of states is $m \times 2^n$. The FSM grows exponentially in the number of switches, which obviously makes it hard to obtain an intuitive understanding of the problem from whatever graphical representation might be chosen.

Now in contrast how our formalism describes the scenario: Each binary switch S_i is modelled with two states, and the ternary indicator I with three states, as shown in Figure 1. All couplings between the switches and the indicator are introduced by edge labels: Labels f_i below the edges of switch i emit event f_i (a flip of switch i) that is consumed by the ternary indicator, as indicated by the labels above the edges. Thus I cyclically switches its state in response to these events. The visual representation of the scenario is as shown in Figure 1.

```
FSM class "HealthSignal" {
  hop green_yellow += xFlip yYellow
  hop yellow_red   += xFlip yRed
```

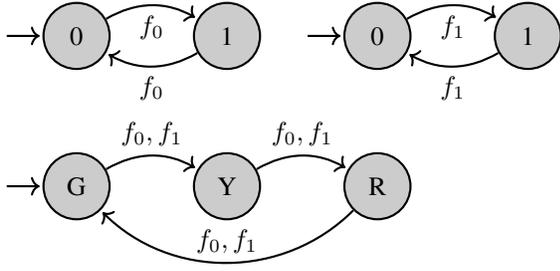


Fig. 1: McFSM visualization of a ternary status indicator controlled by two binary switches. The exponential state space explosion of traditional FSMs is replaced by a linear growth in the number of states (and edges) depending on the (increasing) number of switches.

```

}
hop red_green += xFlip yGreen
}
FSM class "Switch" {
hop up_down += xPress yFlip
hop down_up += xPress yFlip
}
McFSM class "ComboSwitches" {
Switch inst S1 {
Start: up
cap &xPress += ../xPressS1
}
Switch inst S2 {
Start: up
cap &xPress += ../xPressS2
}
HealthSignal inst Lights {
Start: yellow
cap &xFlip += ../S*/yFlip
}
}

```

The DSL distinguishes between classes and instances, which is familiar to most programmers. Classes are very similar to traditional FSMs (as indicated by the keyword `FSM`) in that they are characterised by their states and transitions. However, they do not require an initial state. We distinguish between *incoming* transitions (identifiers prefixed by “x”) triggered by external events, and *outgoing* transitions (identifiers prefixed by “y”).²

Edges correspond to state transitions (hops) constructed e.g. by `hop up_down += xPress yFlip`: This generates a directed edge from state “up” to state “down”, and labels it with `xPress` and `yFlip`. Thus this transition is triggered in state up when event `xFlip` is observed (when the button is pressed), causing the edge-id to be added to an internal list of events that will be processed. The output `yFlip` may be used in the DSL to group or abstract from concrete edge-ids. Note that edge constructions implicitly create states as well.

²The rationale for using prefixes `x` and `y` is the form $y = f(x)$ of algebraic equations, where x is an input and y an output value. In the visual representation, this corresponds to labels above and below the edges.

Classes, instances, states, edges and events can be hierarchically nested. If an object `obj2` is hierarchically below `obj1`, the DLS represents this using the syntax `/obj1/obj2`. We cannot fully describe the concept here, but in the above example, the McFSM class `ComboSwitches` is on level 1 of the global hierarchy, and so are the input events `xPressS1` and `xPressS2`. FSMs instantiated as part of the McFSM reside in level 2, and so on. The syntactic element `../` refers to the previous hierarchy level.

Connecting edges in FSM instances with events requires a generic syntax that can select instances of interest. Consider, for instance, the statement `cap &xPress += ../xPressS1`: It selects every instance of `/ComboSwitch/S1` that already has an event annotation `xPress`, and adds an additional annotation `../xPressS1` (in absolute form: `/ComboSwitch/xPressS1`). This connects the input event `xPressS1` with the relevant transitions in the button instance. Every time a (possibly physical) button, mapped to the external McFSM event by system mechanisms, is pressed, the appropriate transitions are triggered in the class instance that models the switch.

In general, `cap` expects an edge list as parameter, and adds labels to these edges using the `+=` operator. Edge lists and lists of labels may be specified in three ways: 1) As a list of absolute/relative edges/labels, 2) using glob patterns as in `../S*/yFlip` (this selects all `yFlip` labels on switch instances `S1` and `S2`), and 3) as semantic references of the form `&xFlip`, which selects the set of all edges labelled with `xFlip`.

B. Formalism

An FSM F is given by a four-tuple $F = (Q, \Sigma, \delta, q^{(0)})$, where Q is a finite set of states, Σ is a finite set of events, $\delta : Q \times \Sigma \rightarrow Q$ denotes the transition function that determines the next state given the current state and an input event, and $q^{(0)} \in Q$ is the initial state of the machine.

A McFSM $M = (Q, \Sigma, \delta, q^{(0)})$ comprises a set of $n > 1$ interacting FSMs $\{F_1, F_2, \dots, F_n\}$, with state space $Q = Q_1 \times \dots \times Q_n$, event set $\Sigma = \Sigma_{\text{ext}} \cup \Sigma_{\text{int}}$ and initial state $q^{(0)} = (q_1^{(0)}, \dots, q_n^{(0)})$.

The events in Σ_{ext} constitute the “interface” of the McFSM and can be connected, for instance, to inputs of physical sensors as parts of a larger program. Σ_{int} is invisible to the outside. It consists of pairs of states $(q_i^j, q_i^k) \in Q_i$ that create *couplings* between the FSMs F_i , because the *internal event* $e_{i,j,k} = (q_i^j, q_i^k)$ occurs every time the state transition $q_i^j \mapsto q_i^k$ occurs in F_i , but can be associated with any F_x by defining $\delta(q_x, e_{i,j,k}) (q_x \in Q_x)$, thus notifying the observing FSM F_x .

Operationally, events $a \in \Sigma_{\text{ext}}$ can be provided by arbitrary system sources; their processing by a McFSM generates internal coupling events $b \in \Sigma_{\text{int}}$, because *state transitions themselves are considered to be events*.

The transition function δ of the McFSM M combines the transition functions $\delta_1, \dots, \delta_n$ of the constituent FSMs by providing any input event a and the ensuing coupling events b

to the machines F_i in a predefined order that may be event-dependent, but usually just follows the order of the machines F_1, \dots, F_n .

To realise the described event distribution and coupling, we use a data structure called *XQueue* that combines queue- and stack-like features and enforces that 1) all events are treated in the order they occur, as in a queue, and 2) all ensuing *coupling events* are treated before the next event a is processed, as in a stack. Processing is performed as an atomic step before any side effect handlers are called. A precise formulation of the algorithm is given in the accompanying website and the system source code.³

The UI can show an upper bound to the amount of steps required to distribute an external event to the components, and execute their actions. This makes the formalism suitable for real-time systems.

IV. RELATION TO OTHER APPROACHES

McFSMs share many desirable properties with ordinary FSMs, in particular the ability to prove predicates on their state-spaces, which is relevant for safety-critical systems.

In the observer pattern, the observers B_1, B_2, \dots, B_n can again be subjects of further observers. All these objects can be viewed as multiple FSMs that are *coupled* through *notifications*, that is, function calls of *event-handlers*. These cascading notifications can induce substantial hidden complexity that can only be tested at *run-time*. A McFSM combines the state spaces of the constituent FSMs, but makes coupling events (observer *notifications*) explicit at *compile-time*, which provides the basis for debugging and proving static correctness properties. A McFSM handles state transitions as one single *atomic* action and then calls event handlers that can perform any necessary side-effects.

While the available space does not permit to discuss all differences and similarities with previous approaches in detail, we note that McFSMs share the idea of dividing a system into sub-automata with UML hierarchical state machines (our ability to generate instances from classes provides a template-like extension to the idea). Likewise, the concept of emitting signals from edge transitions also appears in statecharts and related formalisms.

As for the differences, we would like to highlight two salient points: Firstly, our pattern matching based interdependency specification does not only lead to considerably increased expressive power, but further economises the number of arrows appearing in charts – our formalism can be seen as hybrid between a reactive programming language (intentionally not Turing-complete) and a visual modelling mechanism where both aspects complement each other. Secondly, McFSMs (in particular using the *XQueue* mechanism) have been designed from the ground up to provide well-defined mathematical semantics, which eliminates one major criticism (see, e.g., [3]) of statechart-like approaches.

Another related approach, the Labelled Transition System Analyser (LTSA, see [9]), aims specifically at modelling concurrency, and proving certain properties of concurrent systems. Although LTSA is also centered around appropriately combining finite state machines, our approach differs in that concurrency is not the core focus, which is also reflected in the substantially different description language that aims on system design, not at describing concurrent processes. The language also avoids mixing any side-effects into the description language, and strives that expressive power is combined with utmost syntactic and conceptual simplicity, which is especially desirable in efforts that require (safety or other types of) certification. Despite the impossibility of making such statements objective, we have tried to leverage decades of industrial experience to achieve the goal to the best of our abilities.

V. TOOLS & OUTLOOK

The approach described in this paper is accompanied by an integrated set of tools and a graphical user interface (GUI) that combines the DSL with a visual representation, and can be used to experiment with our new approach. More information is available on the supplementary website <https://hps.hs-regensburg.de/maw39987/icse/icse.html>.

Future work will focus on testing the efficiency of the specification language and the GUI on practical problems from various domains: Using experts from the fields, we intend to collect typical problems, and implement solutions using the McFSM formalism. By carefully evaluating the results, we will further improve the DSL, the GUI and other tools involved, especially regarding their intuitive usability.

REFERENCES

- [1] G. Berry and G. Gonthier, *The Esterel Synchronous Programming Language*, Science of Computer Programming 19 (2), 87–152, 1992.
- [2] E. Borger and R. F. Stark: *Abstract State Machines: A Method for High-Level System Design and Analysis*, Springer, New York, 2003.
- [3] P. Derler, E. A. Lee, A. Sangiovanni-Vincentelli, *Modeling Cyber-Physical Systems*, Proc. IEEE 100(1):13-28, 2012.
- [4] Ch. 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 (OSPRT), 2015, 7-12.
- [5] A. Diewald, S. Voss, and S. Barner.: *A Lightweight Design Space Exploration and Optimization Language*, Proc. 19th WS on Soft. Comp. Emb. Sys. (SCOPEs), 2016.
- [6] E. Gamma, R. Helm, R. Johnson, and J. Vlissides: *Design patterns: elements of reusable object-oriented software*, Addison-Wesley Longman, Boston, MA, USA, 1995.
- [7] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud *The synchronous data flow programming language LUSTRE*, Proc. IEEE 79(9), 1991.
- [8] D. Harel, *Statecharts: a visual formalism for complex systems*, Science of Computer Programming 8(3), 1987.
- [9] J. Magee and J. Kramer, *Concurrency: State Models & Java Programs*, John Wiley & Sons, New York, 1999.
- [10] I. Maier and M. Odersky, *Deprecating the Observer Pattern with Scala.React*, Technical Report EPFL-REPORT-176887.
- [11] OMG Unified Modeling Language Superstructure, OMG document formal/2009-02-02
- [12] I. Zuzak, I. Budiselic, and G. Delac, *A finite-state machine approach for modelling and analyzing RESTful systems*, J. Web Engineering, 10(4), 353–390, 2011.

³To be published under an open source license.