International Journal of Multimedia and Ubiquitous Engineering
Vol.9, No.1 (2014), pp.289-304
http://dx.doi.org/10.14257/ijmue.2014.9.1.27

A New Approach to Specification of the Behavior of Embedded Systems

Sabah Al-Fedaghi and Abdulrahman R. Alazmi

Computer Engineering Department, Kuwait University

sabah.alfedaghi@ku.edu.kw, abdulrahmanr.alazmi@ku.edu.kw

Abstract

Embedded systems involve a coupling of hardware and software. Modeling of an embedded system is a key step in the requirements phase, when a product's specifications are gathered and modeled. This paper focuses on the specification and modeling phase (the requirements phase) in systems that have been modeled using the FSM approach. The concept of state has been a central notion in this development. While FSM has many diverse and unique properties, the notion is not rich enough to capture complex systems, especially embedded systems with software and hardware parts. The concept of state raises misconceptions, especially in education. To provide an alternative to FSM for system description, this paper introduces a new model as the foundation for a new approach to modeling applications that need state-like methodology. The methodology is based on identifying "things that flow" (e.g., data, numbers, signals) and specifies their streams of flow in terms of no more than six generic stages. The resulting conceptual picture provides a map of diverse flow systems that trigger each other. In this paper, study cases that originally used FSM are re-diagrammed using the new methodology to compare the semantics reflected in each method. Such semantics can be used to understand the behavior of a given system in design and educational contexts. The results point to the viability of the flow-based method for capturing the specification and modeling of systems.

Keywords: *system model, system component, Finite State Machine FSM, Flow Model FM, behavior, specification*

1. Introduction

Current computing hardware and software systems rest on a foundation of discrete models that have been used since the inception of the field. The need to model computing systems for verifying specification and behavior is a well-known challenge [1-3]. Digital systems imply discreteness, and discreteness implies the notion of state. A complex system comprising several functional units can be decomposed into states and state transitions, the basis of the known Finite State Machine (FSM) model. Finite state machines have been the main components in the design of digital sequential circuitry. In addition, FSM has been the dominant tool for modeling of hardware/software systems and is used for synthesis and analysis as well [1]. FSM is especially used in the design of embedded systems.

Embedded systems involve a coupling of hardware and software. Modeling an embedded system is a key step in the requirements phase, in which product specifications are gathered and modeled using FSM. Object oriented and Entity Relationship models are used as well for specifying behavior in the requirements phase [4].

ISSN: 1975-0080 IJMUE

The concept of *state* has been a central notion in this development. States are used to partition all "situations" of a system. Typically, a system is defined as a collection of interacting components functioning as a whole. All non-random combinations available to the system form the states. FSM models can be deterministic, where the states and state transitions are triggered by "predictable" inputs, whereas, in a non-deterministic FSM, states and state transitions are "unpredictable." Here, unpredictability means that combinations of inputs and outside interference trigger state transitions, so they are known to be event driven.

While not overlooking the many diverse and unique properties of the FSM [5], Chomsky [6] criticized these notions as not rich enough to capture the natural language. Nevertheless, FSMs are used to model regular languages, and they are used in many information-processing tasks and form an essential part of system models. In hardware development, FSMs are suitable for designing components and can easily be transformed into actual implementations using flip-flops to represent the states of the machine. For example, if a machine has seven different states, the implementation would use several flips-flops to represent the states.

Gaski *et al.*, [7] demonstrated that FSM is not suitable for describing embedded systems with software and hardware parts because it lacks the ability to reflect programming blocks, concurrent and sequential behavior, and exceptions. Extra rules and constructs are needed, such as hierarchical FSMs for hardware behavior and/or constructs in a programming language for software attributes.

In the software design area, FSMs have some shortcomings that are often misused and misunderstood [8-10].

Much finger-pointing occurred between hardware and software groups when things went wrong, which likely set the foundation for software developers to seek new solutions and methods among their own, as computer scientists and engineers. This separation of hardware and software development was inevitable and necessary to the evolution of the entire embedded industry, but unfortunately it seems, the usefulness of Finite State Machine concepts in software was lost in the aftermath, except to a few people who, to this day, will attest to FSMs as being most beneficial even for software development.

In education, misconceptions exist about the notion of state. According to Herman *et al.*, [11], "In introductory undergraduate courses on digital systems, students have difficulty understanding the state concept. We expect that students have subtle, persistent misconceptions about state because it is an abstract concept and has nuanced meanings in different disciplines as well as colloquial English… First, students had different definitions for a state transition… Second, students disagreed on the definition of the phrase 'leaving each state'."

In this paper, we propose a new model, the Flowthing Model (FM) as an alternative to FSM for describing hardware/software systems. Since a large body of literature exists on FSM and its application, we focus on a few samples to make the point that our proposed methodology introduces a viable approach in the areas where these samples are used; however, this does not limit the general applicability of our approach, and our aim in future research is to apply the new model wherever FSMs are applicable.

The next section presents a brief introduction to the Flowthing Model as it is utilized in diverse areas such as software specification [12], power systems [13], and communications [14]. We then discuss case studies modeled in FSM and also in FM to illustrate the behavior and attributes of the same system represented in both models. Finally, we draw conclusions from the discussion in this paper.

2. Review of some FSM-based specifications

An embedded system with hardware and software components poses difficulties for designing a model that captures its behavior. Often the modeling method is too detailed to hide implementation of the system, such that use of the model to implement the components becomes prone to errors and missing requirements. The critical nature of capture of embedded systems lies in the fact that most of these systems are part of a critical mission and hard real time is required for reliability, fast response, and compliance with requirements other than performance or ease of use.

Typically, design methods are split into multiple stages or steps [7]. In the first step, what the system does and how it behaves are specified (requirements phase) [4]; the system components are then developed, including partitioning of functionality and interconnections (modeling phase). In the implementation phase, the design is transformed from high-level specification to an actual system.

Given the complexity of modern technology, a system can be made from multiple heterogeneous parts. This trend of divergence in implementation has led to the design phase becoming more abstract, in order to hide the details of components and to give system designers the freedom to focus on fulfilling the requirements rather than on dealing with myriad details.

This paper focuses on the specification and modeling phase (the requirements phase) in which the system is modeled using such methodology as the FSM approach.

For example, based on the FSM model, Chiodo *et al.*, [2] discussed a design schema for capturing the behavior of an embedded system with mixed hardware and software attributes. Their approach, called Codedesign Finite State Machine (CFSM), works for small control-dominated systems. It uses a modified FSM model that has code features such as events and data transfer, two features that are not explicit enough in a traditional FSM.

The CFSM model consists of memory elements; events; data flow; and transformations, with triggering of events; storage; and processing. It has high degree of freedom, since an event can be transformed to data, then flow and be transformed yet again. The motivation behind the CFSM is to overcome the *shortcomings* found in FSM models of embedded systems. In addition, concurrent FSM models assume that all FSMs change state synchronously, a fact that does not hold in software implementation.

Edwards *et al.*, [1] discuss the stages needed to move from the requirements phase to the modeling phase for reactive real-time embedded systems. These can range from vehicle controls to electric home appliances to communication systems. Edwards *et al.*, argue that most designs are *ad-hoc* and depend heavily on previous experience or designs. In addition, they believe that formal models and high-level synthesis ensure correct designs and are a prerequisite for a simple model.

According to Edwards *et al.*, [1], a formal model includes the following:

- *Functional specification:* the system's inputs and outputs and possible interactions among its components, which can be a set of heterogeneous groups.
- *A set of properties:* the states, rules applied for inputs, outputs, and possible interactions.
- *Set of performance indexes:* the quality of the design in terms of cost, reliability, and speed.
- Finally, a set of *performance constraints* that rule whether or not a performance is accepted.

These features are incorporated in the design refinement phase. An FSM is used during the first subphase of the stage. This paper deals with this level of description.

Wagner [15] describes a common feature of systems modeled in FSM, including Moore and Mealy FSM machines. He discusses modeling of a home appliance device, an oven, using FSM. The oven's "Run" button initiates the cooking process only when the door is closed; otherwise, the cooking process is halted. A timer specifies the cooking period, with a maximum value of 30 minutes. A lamp is turned on when the door is closed and cooking starts, and it turns off when the door is opened.

First, Moore FSM is used to model the oven, producing seven states of the oven, with some being redundant (*e.g.*, the state of *door opened* is shown multiple times between different sources and destinations). This duplication is justified by the lack of memory in an FSM model; hence, one needs to know which context of the *door opened* state is being shown, and what happens when an interruption occurs during cooking, when idle, or when cooking is done. Second, Mealy FSM is used to model the oven. The Mealy model produces fewer states (five instead of seven). Wagner [15] emphasizes that Mealy FSM produces haziness among states, and that an action within a state determines the next state, not the present state itself. Third, Wagner uses a mixture model comprising both Mealy and Moore FSMs to balance the number of states and their complexity and claims that this produces an optimal model with advantages of both Mealy and Moore FSMs. The model uses state types of both models interchangeably, mixing the inputs and outputs of the two machines, with some states from Mealy FSM, some from Moore FSM, and some mixed.

In this paper, Wagner's oven example will be used as a case study and also analyzed and compared with an FM specification modeling of the same oven system. In the next section, a new model is reviewed that has been used in many applications as a foundation for a new approach to modeling applications.

3. The Flowthing Model

The Flowthing Model (FM, [13]) was inspired by the many types of flows that exist in many fields, such as, for example, supply chain flow, money flow, and data flow in communication models [16]. This model is a diagrammatic schema that uses flowthings to represent a range of items that can be data, information, signals, or objects. FM also provides the modeler the freedom to draw the system using flowsystems that include six stages, as follows:

- Arrive: A flowthing reaches a new flowsystem (*e.g.*, a buffer in a router)
- Accepted: A flowthing is permitted to enter the system (e.g., correct address for a delivery); if arriving flowthings are also accepted, Arrive and Accept can be combined as a Received stage.
- Processed (changed): the flowthing goes into some kind of transformation that changes its form but not its identity (*e.g.*, compressed, colored, packaged).
- Released: a flowthing is marked as ready to be transferred (*e.g.*, passengers waiting to board after clearing a security check).
- Created: a new flowthing is born (created) in the system (a data mining program generates the conclusion *Application is rejected* for use as input data).

- Transferred: the flowthing is transported somewhere outside the flowsystem (*e.g.*, packets reaching ports in a router, but still not in the arrival buffer; goods en route to their destination in the hold of a cargo ship).

These stages are mutually exclusive, *i.e.*, a flowthing in the process stage cannot be in the created stage or the released stage at the same time. An additional stage of Storage can also be added to any FM model to represent the storage of flowthings; however, storage is not a generic stage because there can be, *e.g.*, stored processed flowthings and created stored flowthings. Figure 1 shows the structure of a flowsystem. A *flowthing* is a thing that has the capability of being created, released, transferred, arrived, accepted, or processed while flowing within and between systems. A *flowsystem* depicts the internal flows of a system with the six stages and transactions among them. FM also uses the following notions:

- *Spheres and subspheres*: These are the environments of the flowthing, such as a company and the departments within it, an instrument, a computer, an embedded system, a component, and so forth. A sphere can have multiple flowsystems in its construction if needed.

- *Triggering*: Triggering is a transformation (denoted in FM diagrams by a dashed arrow) from one flow to another, *e.g.*, a flow of electricity triggers a flow of air.

A flowsystem may not need to include all the stages; for example, an archiving system might use only the stages Arrive, Accept, and Release. Multiple systems captured by FM can interact with each other by triggering events related to one another in their spheres and stages.

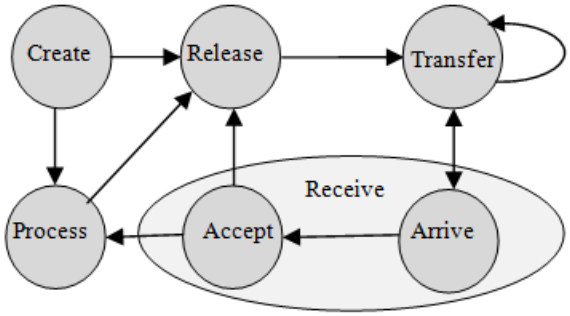

Figure 1. Flowsystem, assuming that no released flowthing is returned

Example: The following example presents an FM description corresponding to an FSM given by Hansen [4], for the purpose of comparing the semantics reflected in each diagram. Such semantics can be used to facilitate understanding of the behavior of a given system, as in an educational context; however, further exploration of the use of FM in the digital design process is beyond the scope of this paper.

In [8], an example is given of a state diagram describing a simple 2-bit counter, as shown in Figure 2. The counter has two inputs (Rst and Clk) and one output (the counter value, Q).

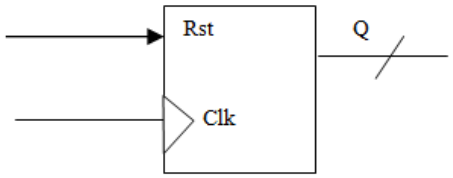

Figure 2. A 2-bit counter (from [8])

The Clk input is the "pulse." Every time the Clk signal moves from low to high (a rising edge), the state machine decides what to do according to its input values. In this case, the machine has one (ordinary) input only: the Rst input. When this signal is held high at the time of a rising edge on the Clk signal, the counter will reset to 00. When the Rst input is held low, the counter will count at each rising clock edge. The numbers we expect to see on the Q output are 00, 01, 10, and finally 11. After this, the counter starts again from 00. This behavior can be described using a state diagram like the one shown in Figure 3. As noted by Hansen [8], "In state diagrams the Clk input is implied ('not shown'), because it in principle has nothing to do with the function of the state machine. It is merely the signal that tells the state machine when to decide to move to the next state" [8].

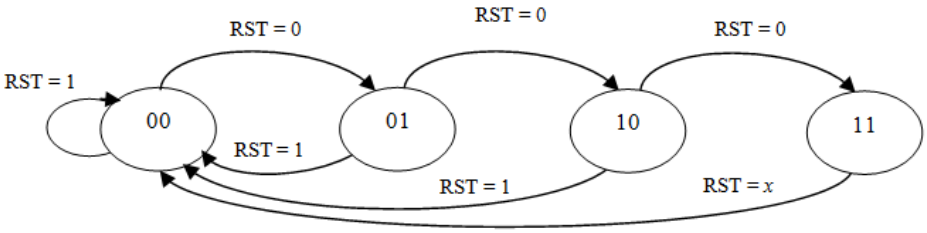

Figure 3. FSM representation of the 2-bit counter (from [8])

The corresponding FM representation is shown in Figure 4. The counter comprises two flowsystems:

1. The Numbers flowsystem handles counting by generating numbers

2. The Reset (Rst) flowsystem represents the following input (numbers in parentheses refer to circled numbers in Figure 4):

If Rst = 1, then the number flowsystem performs a process that creates a number, as follows:
If (true) (1) *create 00* (2),
If (00) (3) *create 01* (4),
If (01) (5) *create 10* (6),
If (10) (7) *create 11* (8),
If (11) (10) *go to If (true)*

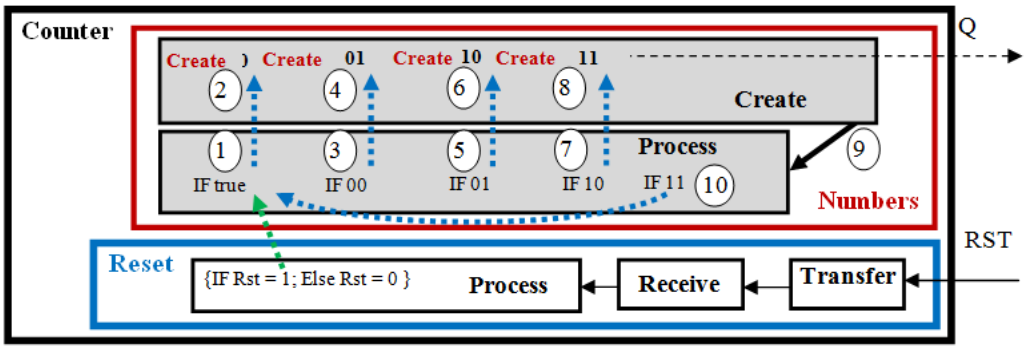

Figure 4. FSM representation of the 2-bit counter

Note that the flow (solid arrow at 9) between the process stage and the create stage is repeated each time a number is created. Also, *if (true)* is used to create uniformity in create/process numbers because it is always true.

The FM looks more complex than the FSM; however, the FSM representation is in fact a sketch of the truth table, as shown partially in Figure 5. The FM representation includes the semantics of the system and includes counting (numbers flowsystem) and a start/re-start mechanism (Rst flowsystem). Counting is performed in two stages: create a number, and then process it to create the successive number.

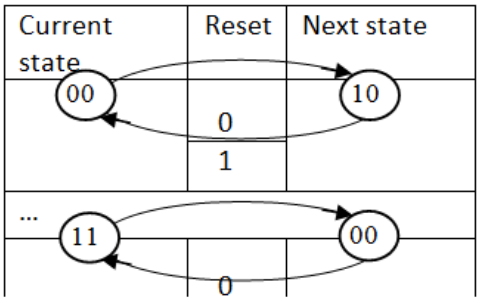

Figure 5. FSM representation is a sketch of a truth table

The presentation is similar to slides in a narrative process that can easily be transformed into a program; for example, we can write it in pseudocode, as follows:

{ Start :
Receive Rst
If Rst == 1 {trigger Numbers, Rst = 0}
Numbers:
{create count == 00
IF count == 00, count = 01, if Rst == 1 go to numbers;
IF count == 01, count = 10, if Rst == 1 go to numbers;
IF count == 10, count = 11, if Rst == 1 go to numbers;
IF count == 11, trigger Q;
Output : Q
IF count == 11, count = 00;
IF Clock == 1, Then Loop Start;

Though, as noted, the Clk input is not shown, it can be incorporated into the original diagram to produce Figure 6. We notice the same sketchy description appears in the FSM diagram. There is discontinuity in the conceptual picture. Are 00, 01, 10, and 11 states of the counter? These values seem more like symptoms of (current) states, analogous to considering a skin rash to be the state of some disease rather than a symptom. A state is the total picture of the system at a given point, as in the state of traffic. Figure 7 depicts the FM conceptual description of the counter.

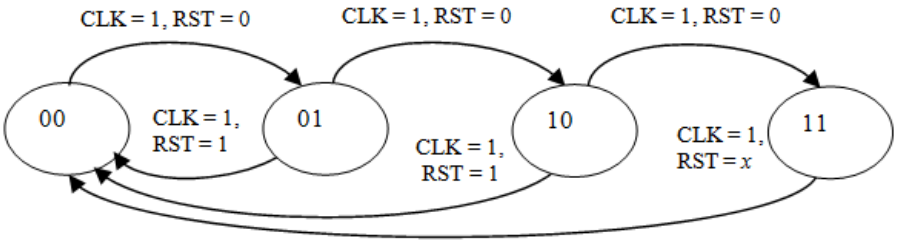

Figure 6. FSM representation of the 2-bit counter

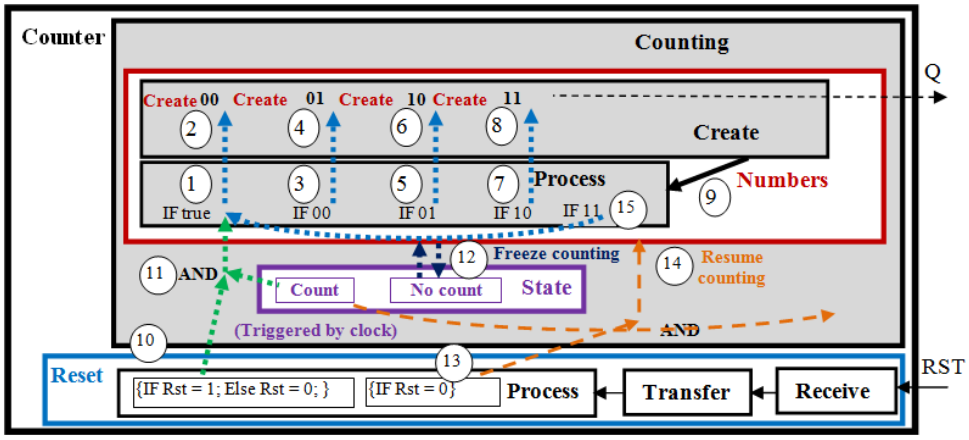

Figure 7. FM representation of the counter

The representation includes a counting subsphere comprising the numbers flowsystem, described previously, and a control mechanism (state flowsystem) that freezes and resumes counting according to

(a) the clock cycle and

(b) completion of creating a number. If Rst = 1 and the state = counting (circles 10 and 11), then the number flowsystem performs a process that creates a number as follows:

If (true) (1 in Figure 7) *create 00* (2), *then freeze by triggering state = No counting* (9)
If (00) (3) *create 01* (4), *then freeze by triggering state = No counting* (9)
If (01) (5) *create 10* (6), *then freeze by triggering state = No counting* (9)
If (10) (7) *create 11* (8), *then freeze by triggering state = No counting* (9)

Figure 8 illustrates the synchronization involved. Note that the flow (solid arrow, circle 9, between the Process and Create stages) is repeated when in the counting state (numbers flowsystem is not frozen). It is assumed that the clock (not shown in the figure) would cause resumption of Process/Create by setting state = Counting.

Rst = 1 (circle 10) and "at the time of a rising edge on the Clk signal," which we call state = counting (11), causes activation of the counter (1) to create 00 (2), the start of counting. Here, create means initialization of the numbers in the sphere of the counter.

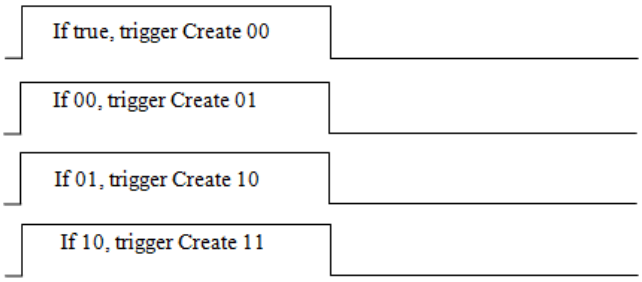

Figure 8. Possible clock synchronization for the example

The number 00 flows to be processed. This flow from Create to Process triggers (9) the low cycle that deactivates the counter (No counting, 12). The counter stays in this condition until being "awakened" by the high clock cycle if Rst = 1 as it triggers the counter to resume its processing of 00 in the Process stage, thus triggering the creation of 01 (13). This procedure continues until the number 11 is reached, as shown in Figure 8, where the high and low cycles of the state (clock) are shown. We assume that reaching 11 will trigger something (14).

It is clear that, conceptually, the FSM representation mixes counting of numbers (changing numbers) with the counting of states (synchronizing with the clock cycle), as in Figure 8.

4. Case Studies

In this section, we examine two cases in which FSM is used as a base for modeling. The first is the FSM of an oven system [15] discussed in Section 2, which we compare to its specification in FM. The second case study is a car seatbelt model presented in [2], which we compare between FSM and FM with circuit implementations. Another case [4] concerns the FSM modeling of an elevator system; FM could also be applied here, but due to space constraints, we shall consider only the first two.

The main idea in this section is that a state is a flowthing. A flowthing is a thing that can be created, processed, released, and transferred, arrive, and be received. "Flow" in FM is not a mere *movement* in space or time; rather, it can mean a change, a transformation in appearance or condition. A state itself can be only created or processed (changed). A *flowthing* flows through its flowsystem. A flowsystem can be viewed as a sphere when it comprises more than one flowsystem. For example, if we define a car as {body, state} then Figure 9 shows a description of a car.

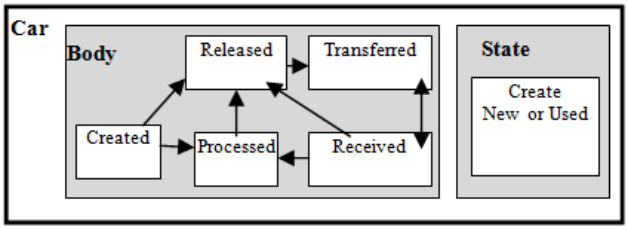

Figure 9. A flowthing with two flowsystems

Similarly, the state of a door can be *Open* or *Closed* when it appears as a component of a sphere. *Creation*, here, is a conceptual phenomenon within a sphere and not necessarily an ontological existence. For example, if a door *appears* in a film, it can appear (is photographed, is generated, is created) as Open or Closed. In another scene, the door may not be included and thus conceptually does not exist, is not "created".

States can be created or processed (*e.g.*, an open door is closed halfway) but cannot directly flow to another sphere in the manner of actions. Consider a human being described as a body and a mind. How can a person *transfer* a personal state of pain to another person, except indirectly through speech, signal, or action? By contrast, an action (a flowthing) is released and transferred (*e.g.*, a hand movement) directly to another person, who receives it (is knocked down).

5. Oven Device Model

As discussed previously in the case of the oven [15] in Section 2, an oven system is represented in FSM three times [15]: as Moore and Mealy FSM machines, and as a mixed model, seen in Figure 10.

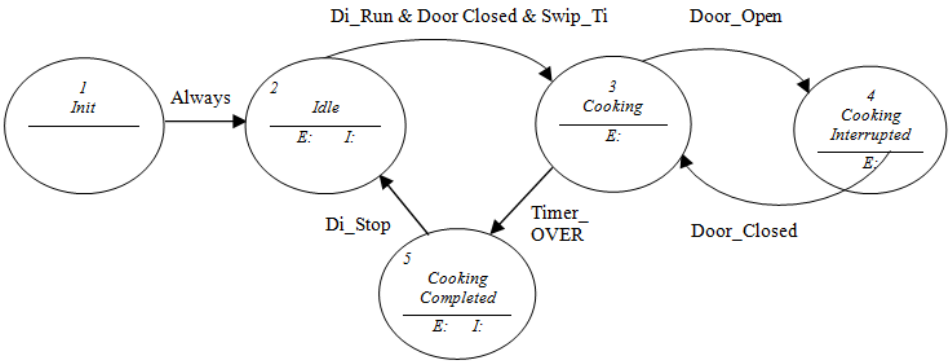

Figure 10. Mixed FSM oven model (from [15])

The states in the mixed model are Init; Idle; Cooking; Cooking Interrupted; and Cooking Complete (see Figures 2 and 3). The Cooking and Cooking Interrupted states are Moore machines; the other states, which are Idle and Cooking Complete, are mixed, and the Init state is Mealy. The model is no less complicated than the previous two models (Mealy and Moore). Transforming a Moore machine into code is relatively easy; the Mealy machine, however, is more difficult, as unintended scenarios may arise in the code. In FM, in contrast, we divide the oven system into the following main components: **Oven**, **Switch**, **Timer**, and **Doo**r, as shown in Figure 11.

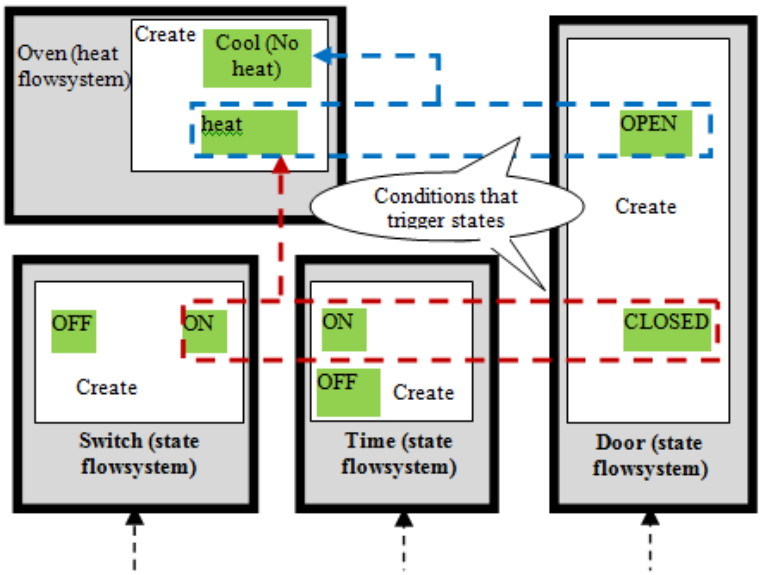

Figure 11. Flow-based description of the oven example

The dotted arrows in the figure show the triggering events. The components have states as follows: Door is either Open or Closed, Time is either On or Off, *and* Switch is either On or Off.

Two conditions are shown in the figures:

1. The **Door** is closed; **Switch** is on and the **Timer** is on; this condition triggers an event in the oven chamber to create heat.

2. The **Door** is open, which triggers creation of cool and thus overrides creation of heat, if oven is in this state. Since the door is either open or closed, then only one such situation exists at a time.

Figure 10 shows the oven device model as described using a mixed FSM, a mixture of both Mealy and Moore machines. States *Idle* and *Cooking Complete* are mixed, while states *Cooking* and *Cooking Interrupted* are Mealy state machines.

As noted, Wagner [15] claims that a mixture of both FSM types yields the optimal modeling. It has five states, in contrast to Mealy alone, which has five, and Moore alone, with seven states. In Mealy, since the output depends on the state and the input in each state, all state transitions leading to it from other states require all actions to be repeated at each state. For example, the Timer has to be started at both states *Idle* and *Cooking Interrupted*. Moore is simpler to understand yet uses more states. For example, *Door_Open* has two versions depending on the status of the Run action. Furthermore, simulating concurrency of this event or a similar event with little difference in action requires different states with entry actions, which are sometimes redundant.

Consider a sample transformation such as Cooking to Cooking Complete when the Timer is Over, which is a state of the Timer itself. Is transformation another state of some items? What is the difference between a state and a transformation? This can be conceptualized as a

change in the state of something (*e.g.*, a door being closed) triggering a change in something else (*e.g.*, cooking). Then why does the model use arrows to represent states? Figure 11 shows the corresponding FM-based modeling of the Oven system. Two types of constraints control the system:

1. The door is OPEN and the oven is producing heat, and
2. The door is CLOSED, Timer is ON, and Switch is ON.

Each constraint triggers changes in the state of the oven.

In such a representation, states are represented conceptually in a uniform way. It seems that such a picture is intuitively simple to understand. FM models can be depicted in a more pictorial description in educational environments, such as by using pictures of a closed/open door, a timer as a clock, *etc*.

6. Seatbelt System Model

A seatbelt system model [2] with FSM is shown in Figure 12. The figure consists of three circles representing the data transformations: Wait, Off, and Alarm. There are four solid lines representing data flows. The seatbelt system starts from the state OFF, then goes into Wait when key is ON and at Start. From Wait, data flows when End is reached, when a timer turns ON the alarm for the seatbelt. From Alarm, data flows to Off when End of time is reached or Belt=On or Key=Off and Alarm=Off. The model represents the behavior of a seatbelt system. This system is set to wait when the key is turned on; it triggers an alarm when a time limit is expired to warn the driver to fasten the seatbelt; the alarm turns off when the driver engages the seatbelt or uses the key to turn off the ignition.

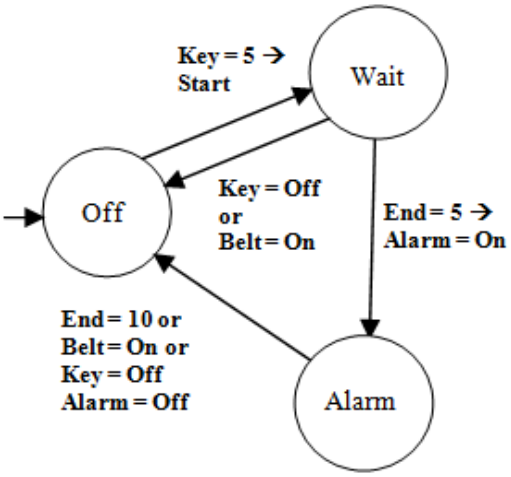

Figure 12. CFSM of the seatbelt system (from [2])

The corresponding FM representation is shown in Figure 13. If the key is turned on (circle 1) and the belt is off (2), then the timer process is triggered (3) to count to 5 seconds. After an interval of 5 seconds (time units) (4), the timer simultaneously activates the alarm (5) and the timer for the alarm itself (6). After 10 seconds (7), the alarm turns itself off (8). Of course, if the key is off (9), the belt is turned on (10) and the alarm is off.

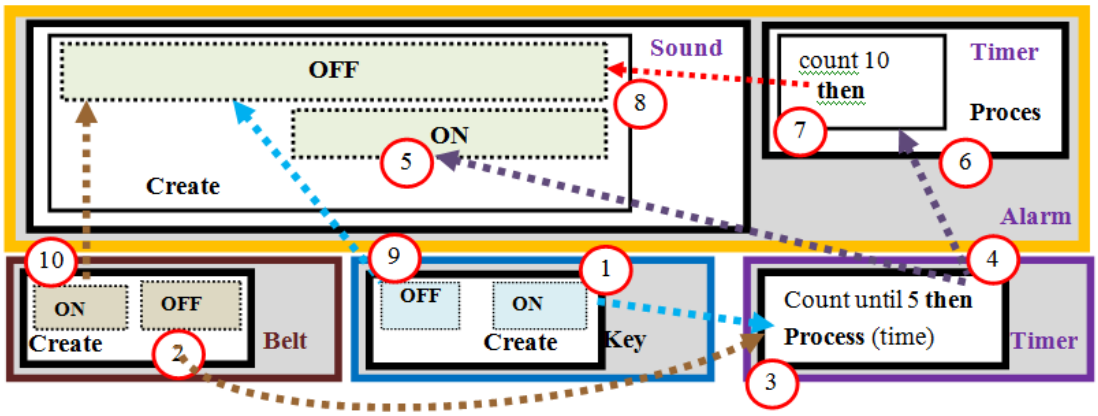

Figure 13. The seatbelt system modeled in FM

In the FM depiction, we can identify the conditions that trigger the event. When the key is on, belt is turned off, and the timer has counted 5 time units, the alarm will sound. When the key is turned off, the alarm is turned off regardless of the other components. When the alarm has sounded for 10 time units, it is turned off. Finally, when the belt is fastened, the alarm is turned off as well.

The FM description starts with the components (spheres) of the system: Seatbelt, Alarm, Key, and Timer. The seatbelt has one flowthing: a thing that changes (in a flowsystem) in the context of the problem. The thing is the state of the belt and focus of creating Off and On. Creation here means that the belt is made to assume a certain manifestation, whether Off or On, in the system; similarly for the key.

The timer is a time processing (measuring) machine. When triggered, it simply delays the activation of the alarm for a certain period. In FSM this is represented by the label "End = 5 → Alarm = ON". This introduces not only END (a variable?) in the conceptual picture unnecessarily, as shown in Figure 12 , but also a mix of values of variables, states (Alarm = OFF), Start (see Figure 12), and logical OR-AND implications, in addition to transformation arrows.

In FM, the conceptual sphere of the alarm includes two phenomena: a sound and a timer for the sound. Each is modeled in its flowsystem, related though triggering to other spheres. The whole FM conceptual picture is intuitive and simple to understand. Figure 14 shows a simplified version of Figure 13 just as a summary.

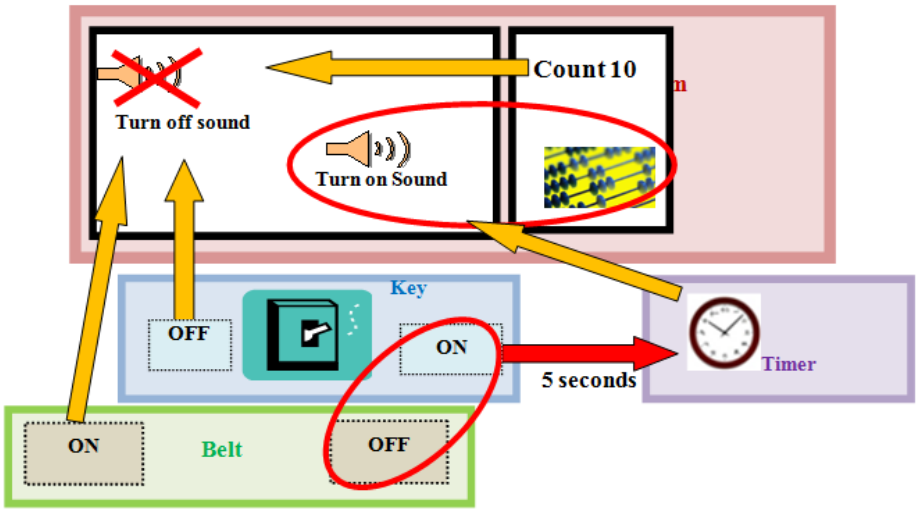

Figure 14. Simplified seatbelt system modeled in FM

7. Concluding Remarks

Modeling of an embedded system has many aspects, including the system's hardware/software components, the interactions between these components, and the components' properties and states that form the system's behavior. Modeling an embedded system is a key step in the requirements phase, in which a product's specifications are gathered and modeled. This paper focuses on a specification and modeling phase (the requirements phase) in which the system was modeled using the FSM approach [17]. The concept of state has been a central notion in this development. While the many diverse and unique properties of FSM cannot be overlooked, its notion of state is not rich enough to fully capture certain systems or, especially, to describe embedded systems with software and hardware parts. The concept of state raises misconceptions, especially in education. An FSM produces scenarios of how the system behaves, but it is not a semantically rich representation, and it represents a type of sketch of truth tables, which are suitable for circuit-like structures.

As an alternative, for a conceptual level representation, we propose the use of the flowthing model (FM) to depict behavior in embedded systems. The methodology is based on identifying "things that flow" (*e.g.*, data, numbers, signals) and specifies their streams of flow in terms of a maximum six stages. The resultant conceptual picture draws a map of different flow systems that trigger each other. Study cases of FSM have been re-diagrammed using the new methodology with the purpose of comparing the semantics reflected in each method's diagrams. Such semantics can be used in understanding the behavior of a given system in design and educational contexts.

The results show the viability of the flow-based model to capture the specification and modeling of systems. It is not a settled issue that FM is used to supplement FSM at the conceptual level, or that a more abstracted version can be used as an alternative to FSM. We are currently extending FM to reach to the digit logic design level, starting by abstracting FM to map it to a truth table-like representation.

References

[1] S. Edwards, L. Lavagno, E. A. Lee and A. Vincentelli, "Design of embedded systems: formal models, validation, and synthesis", Proceedings of the IEEE, **(1997),** ISSN: 0018-9219.
[2] M. Chiodo, P. Giusto, A. Jurecska, H. C. Hsieh, A. S. Vincentelli and L. Lavagno, "Hardware-software codesign of embedded systems", IEEE Micro, vol. 14, no. 4, **(1994)**, pp. 26-36.
[3] A. T. Bouloutas, G. W. Hart and M. Schwartz, "Fault identification using a finite state machine model with unreliable partially observed data sequences", IEEE Transactions on Communications, vol. 41, no. 7, **(1993)**, pp. 1074-1083.
[4] M. Lubars, C. Potts and C. Richter, "A review of the state of the practice in requirements modeling", Proceedings of the First International Symposium on Requirements Engineering, **(1993)**, ISBN: 0-8186-3120-1.
[5] R. M. Kaplan, "Finite state technology", Xerox Palo Alto Research Center.
[6] http://legacy.xrce.xerox.com/competencies/content-analysis/arabic/info/history.html.
[7] J. Brownlee, "Finite State Machines (FSM)", http://ai-depot.com/FiniteStateMachines/FSM.html.
[8] D. Gaski, F. Vahid and S. Narayan, "A system-design methodology: executable-specification refinement", Proceedings of the European Design and Test Conference EDAC, **(1994)**, ISBN: 0-8186-5410-4.
[9] J. R. Hansen, "The practical guide to VHDL", version 2.5, Engineering College of Copenhagen, **(2005)**, March 12, http://svn.ihk-edu.dk/svn/WirelessCenter-Course/SDR-FMT2010/10%20FPGA%20Programming%20&%20Digital%20Filters/The_practical_guide_to_VHDL.pdf.
[10] F. Wagner and P. Wolstenholme, "Misunderstandings about state machines", Computing and Control Engineering, vol. 15, no. 4, **(2004)**, pp. 40- 45.
[11] F. Wagner, "Modeling software with Finite State Machines: a practical approach", Auerbach Publications, **(2006)**.
[12] G. Herman, C. Zilles and M. Loui, "Work in progress: students' misconceptions about state in digital systems", Proceedings of the 39th IEEE International Conference on Frontiers in Education Conference, **(2009)**, E-ISBN: 978-1-4244-4714-5.
[13] S. Al-Fedaghi, "Conceptualizing software life cycle", Proceedings of the 8th International Workshop on Conceptual Modeling Approaches for e-Business, **(2009)**, DOI: 10.1007/978-3-642-01112-2_45.
[14] S. Al-Fedaghi, "Systems of things that flow", Proceedings of the 52nd Annual Meeting of the International Society for Systems Sciences, **(2008)**, ISSN: 1999-6918.
[15] S. Al-Fedaghi, "Integrating services: control vs. flow", Proceedings of the 2009 IEEE International Conference on Information Reuse and Integration, **(2009)**, E-ISBN: 978-1-4244-4116-7.
[16] F. Wagner, "Moore or Mealy model?", http://www.stateworks.com/active/download/TN10-Moore-Or-Mealy-Model.pdf.
[17] S. Al-Fedaghi, "Flow based description of conceptual and design levels", Proceedings of the IEEE International Conference on Computer Engineering and Technology, **(2009)**, ISBN: 978-1-4244-3334-6.
[18] COS 116, The Computational Universe, "How to design a Finite State Machine", http://www.cs.princeton.edu/courses/archive/spr06/cos116/FSM_Tutorial.pdf.

Authors

Sabah Al-Fedaghi

He holds an MS and a PhD in computer science from the Department of Electrical Engineering and Computer Science, Northwestern University, Evanston, Illinois, and a BS in Engineering Science from Arizona State University, Tempe. He has published two books and more than 180 papers in journals and conferences on software engineering, database systems, information systems, computer/information ethics, information privacy, information security and assurance, information warfare, conceptual modeling, system modeling, information seeking, and artificial agents. He is an associate professor in the Computer Engineering Department, Kuwait University. He previously worked as a programmer at the Kuwait Oil Company and headed the Electrical and Computer Engineering Department (1991–1994) and the Computer Engineering Department (2000–2007).

http://cpe.kuniv.edu/images/CVs/sabah.pdf

Abdulrahman R. Alazmi

He has a bachelor's degree in computer engineering from Kuwait University. Currently he is a graduate student in Kuwait University Graduate School majoring in computer engineering. His research interests include mobile ad-hoc networks, data warehousing, wireless sensor networks, and system modeling. Alazmi is also currently working as an engineer at Kuwait University.

