Modeling and Simulation of Digital Systems in different Domains

Ivan Paunović, Volker Zerbe

ABSTRACT – A digital system can be specified with the Finite State Machine (FSM) and/or the Statechart approach [1]. A modeled FSM is to be embedded in a Discrete Event (DE) or a Synchronous Data Flow (SDF) domain of a simulator, designtool. It is used MLDesigner, the next generation system designtool [2]. The modelling technique is shown by examples. Simulation results are presented.

KEYWORDS – Finite State Machines, Discrete Event Domain, Synchronous Data Flow Domain, Modeling, Simulation

I. INTRODUCTION

Digital systems are designed on the basis of a functional notation by a structural synthesis or by direct structural notation. The basic concept for specification of digital systems is the finite state machine approach [3], [5]. FSM's also represent a common basis model for both UML (unified modeling language) statecharts and SDL (specification and desicription language) processes.

In the model based design process a tool chain is used. MLDesigner wich is used in the early design phases is a next generation system design tool. The different models are represented graphically in a hierarchy of block diagrams, where the blocks are connceted via Input / Output interfaces. The graphical user interface of the MLDesigner application contains a frame set of different editors, including a multi-document editor to create, edit and save the block diagrams. The storage of the block diagrams is based on the Extended Markup Language (XML). An integrated multi-domain simulator facilitates the analysis and simulation of the different systems and supports for example the following domains:

- Finite State Machine (FSM)
- Discrete Event (DE)
- Synchronous Data Flow (SDF)

Furthermore, MLDesigner [4] embraces an extensive library of base modules and provides animation and plot tools to analyse simulation results.

V. Zerbe is with Faculty of Information and Automatics, Ilmenau University of Technology, Germany,
E-mail: volker.zerbe@tu-ilmenau.de
I. Paunović is with the Faculty of Electronic Engineering, University of Nis, Aleksandra Medvedeva 14, 18000 Nis, Serbia & Montenegro,
E-mail: ivan.paunovic@elfak.rs

In the paper the three domains are shortly described.

Furthermore it is shown the modelling technique by examples.

II. THE FSM DOMAIN

A finite state machine is a conceptual machine with a finite number of states. It can be in only one of the states at any specific time. A state transition is a change in state that is caused by an input event. In response to any input event, the finite state machine might transition to a different state. Alternatively, the event has no effect and the finite state machine remains in the same state. The next depends on the current state as well as on the input event. Optionally, an output action may result from the state transition.

The MLDesigner finite state machine domain includes a graphical editor and an action language for defining and managing states, transitions and interface elements. It supports the UML Statechart semantic, hierarchical states and special events, as well as key MLDesigner features such as data types and data structures, shared memory, and interaction with other design domains.

The FSM semantic provided by MLDesigner supports synchronous and asynchronous behavior, additional events, variables and parameters for various runs of simulations. The FSM mechanism provided by MLDesigner supports all the basic standard elements of a finite state machine. Events are used as triggers to cause a finite state machine to undergo state changes. In doing so, the presence of an event is interpreted as logical true and the absence of an event as logical false. The MLDesigner FSM supports 3 different kinds of events:

Input Ports - The basic events are represented by data parsed into an input port of the FSM model interface.

Special Events - If an FSM model is embedded into a discrete event (DE) environment, MLDesigner Special Event arguments can be used to trigger the FSM model.

Internal Events - In context with the FSMstate slave process mechanism, internal events can be set or reset inside an FSM model, without influence of the outer environment of the finite state machine. These internal events are represented by a boolean flag associated with the name of the event. If a state slave model contains an input port with the name of an internal event, this input port gets the current flag value (true or false) of the associated internal event, before the slave model executes. If a state slave model contains an output port with the name of an internal event and this output port contains new data after

execution of the slave model, the associated internal event flag is set to the integer cast (unequal zero = true, equal zero = false) of the new data. If the output port, associated with an internal event, contains no new data after execution of the slave model, the internal event flag is reset to false.

States represent conditions or periods characterized by the concepts of duration and stability. A finite state machine can have an arbitrary number of states but at any time of execution, the FSM must reside in only one state. In the scope of a single FSM, each state of this FSM must have a unique name. This name is centered at the top of the rounded rectangle in the graphical notation of a state.

Hierarchical States - Each state can have an arbitrary number of sub-states and the sub-states can also be hierarchical. All states on the same level of a hierarchy are sibling states. States up the hierarchy are called ancestor states and states down the hierarchy are called descendant states. Leaf states are states without sub-states. In the graphical representation of hierarchical states, the borders of a sub-state must reside completely inside the boundaries of all ancestor states.

Current State - At any time during simulation, a finite state machine must reside in only one state. This state is called the current state. If the current state is a hierarchical state, then only one of its sub-states must be the current sub-state. This rule goes down the hierarchy until a leaf state is the current sub-state of a level of the hierarchy.

State Actions - Together with each state, it is possible to define two sets of operations. The entry action is executed whenever the state is entered and the exit action is a set of operations performed whenever the state is exited. These actions are defined using the C/C++ like FSM action language.

Slave Process - The MLDesigner FSM provides a slave process associated with leaf states. An MLDesigner module or a different FSM model can be used as a slave process. The slave process of the current state executes if the FSM receives new events and no preemptive transition, possessed by the current state, fires.

Transitions - State changes in finite state machines are described via transitions. Each transition specfies a source state and a target state. The graphical notation is a line or multiple line segments between the source and target state with an arrow on one end, pointing to the target state. If the source and target state of a transition is the same state, the transition is called a **self transition**. Each state possesses all transitions for which it is the transition's source state as well as those possessed by its ancestor states. Latter transitions are called **inherited transitions**. Associated with each transition is a boolean property called **preemptive**. All the preemptive transitions, possessed by the current state, are checked for firing, before the slave process of the current state is performed. In that way, the slave process executes only, if no preemptive transition fires.

Entry Type - In context with the FSMstate slave process mechanism, every transition contains a property called Entry Type. If a transition fires, the entry type of this transition specifies in which way the slave process of the next current state will be executed. The entry type can be either Default or History. For example, a leaf state containing another FSM model as slave process. If this state is entered via a Default entry type transition, the slave process FSM model will always be reset to its initial state, before execution. In the case, this state is entered via a History entry type transition, the slave process FSM model continues the execution in the last current state.

Event Expression - Events can be on Input Ports, Special Events, Internal Events… Complex and nested event expressions can be defined using brackets and the C/C++ logical operators ! (NOT), || (OR) and && (AND). The applicability of the different logical operators within transition event expressions depends on the outer domain of the associated FSM model. MLDesigner provides a special Event Expression Dialog for easy event expression composition. If the event expression evaluates to true, while the current state is a transition's source state or one of its sub-states, the transition is triggered and becomes a candidate for firing. A transition without an event expression is immediately triggered after its source state entry action is executed. These transitions are called synchronous transitions.

Guard Condition - Optionally associated with each transition is a guard condition C, specified by an FSM Action expression. If the guard condition of a triggered transition evaluates to true, the transition fires and the finite state machine's next state becomes the transition's target state. A triggered transition without a guard condition fires immediately. Triggering does not automatically cause transitions to fire, it merely enables firing. If a triggered transition cannot fire, because the guard condition evaluates to false, the transition must be triggered again by a satisfied event expression to become a candidate for firing.

Transition Action - Each transition can also have an action A, which is a set of FSM Action statements. Whenever a transition fires, its action is performed before the transition's target state entry action is executed.

Transition Priority - If more than one transition possessed by the current state is triggered, the inherited transitions up the hierarchy have a higher priority to fire.

Transition Conflict - A transition conflict occurs, when two or more transitions with the same priority are triggered and no transition with a higher priority is candidate for firing. In this case of nondeterminism, the FSM scheduler fires the first one, which guard condition evaluates to true.

Default Entrances - The Default entrance is a special state which indicates the point of entry to that level of the state hierarchy. Each level of the state hierarchy, including the FSM top level, has one default entrance, which is depicted by a small solid circle. A default entrance must not have any incoming transitions and must have only one outgoing transition to designate the default sub-state destination. The top level default entrance designates the initial state of the finite state machine. The Top Level

Default Entrance may be linked to a set of FSM Action statements that initialize the state machine (e.g. initialize memories). Upon simulation startup, the entry action of the initial state is not performed.

Histories - Histories are special states, used to resume the last sub-state of a hierarchical state. An FSM can have an arbitrary number of histories, placed at any level of the state hierarchy. A history must have at least one incoming transition and must not have any outgoing transitions. A **static history** memorizes only the previous sub-state of its hierarchy level. If this sub-state is a hierarchical state, then its default entrance destination becomes the current sub-sub-state and so on, until a leaf state becomes the current state. **Recursive histories** apply to all descendant states and refer to the previous current state of their state hierarchy. So the state, memorized by a recursive history, is always a leaf state on the same or lower level of the state hierarchy. If a transition, pointing to a history fires, actions are performed as if the transition is pointing to the state stored in the history. Trying to enter a hierarchical state via an empty history, when the hierarchical state was never visited before, results in an error being displayed and the simulation aborts.

Arguments - In addition to the basic elements, the MLDesigner FSM semantic supports typical MLDesigner arguments associated with finite state machines.

Memory Arguments - Memories can be used to represent variables in finite state machine. The MLDesigner FSM supports memories of both scopes: internal and external, and all types and data structures as is the case elsewhere in an MLDesigner block. FSM action statements have read as well as write access to the value of a memory argument. In addition to events, represented by the inputs of a finite state machine, the MLDesigner FSM supports **special event arguments** of both scopes, internal and external, and of all types and data structures. These events can also be used in the event expression of transitions and can be accessed via FSM action statements to schedule or cancel an event argument. Events of external scope can generate events in other MLDesigner models, i.e. other FSM models. Using MLDesigner special event arguments in context with finite state machines makes sense only if the FSM model is embedded into a discrete event (DE) environment.

Parameter Arguments - Like memories and special events, parameters are fully supported in MLDesigner finite state machines. FSM action statements have only read access to the value of a parameter.

III. The Discrete Event Domain

The discrete event (DE) domain in MLDesigner provides a general environment for time-oriented simulations of systems such as queuing networks, communication networks, and high-level models of computer architectures. In this domain, each Particle represents an event that corresponds to a change of the system state. The DE schedulers process events in chronological order. Since the time interval between events is generally not fixed, each particle has an associated time stamp. Time stamps are generated by the block producing the particle based on the time stamps of the input particles and the latency of the block.

A DE primitive models part of a system response to a change in the system state. The change of state, which is called an event, is signaled by a particle in the DE domain. Each particle is assigned a time stamp indicating when (in simulated time) it is to be processed. Since events are irregularly spaced in time and system responses are generally very dynamic, all scheduling actions are performed at run-time. At run-time, the DE scheduler processes the events in chronological order until simulated time reaches a global ”stop time”.

Each scheduler maintains a global event queue where particles currently in the system are sorted in accordance with their time stamps; the earliest event in simulated time being at the head of the queue. The difference between the two schedulers is primarily in the management of this event queue. The default DE Scheduler mechanism handles large event queues much more efficiently than the alternative, a more direct DE scheduler, which uses a single sorted list with linear searching. The alternative scheduler can be selected by changing a parameter in the default DE target.

Each scheduler fetches the event at the head of the event queue and sends it to the input ports of its destination block. A DE primitive is executed (fired) whenever there is a new event on any of its input portholes. Before executing the primitive, the scheduler searches the event queue to find out whether there are any simultaneous events at the other input portholes of the same primitive, and fetches those events. Thus, for each firing, a primitive can consume all simultaneous events for its input portholes. After a block is executed it may generate some output events on its output ports. These events are put into the global event queue. Then the scheduler fetches another event and repeats its action until the given stopping condition is met.

It is worth noting that the particle movement is not through Geodesics, as in most other domains, but through the global queue in the DE domain. Since the geodesic is a FIFO queue, we cannot implement the incoming events which do not arrive in chronological order if we put the particles into geodesics. Instead, the particles are managed globally in the event queue.

IV. The Synchronous Data Flow Domain

Synchronous data flow (SDF) is a data-driven, statically scheduled domain in MLDesigner. ”Data-driven” means that the availability of Particles at the inputs of a primitive enables it. Primitives without any inputs are always enabled (including disconnected Xgraphs.) ”Statically scheduled” means that the firing order of the primitives is determined once during the start-up phase. The firing order will be periodic. The SDF domain is one of the most

mature in MLDesigner, having a large library of primitives and demo programs. It is a simulation domain, but the model of computation is the same as that used in most of the code generation domains. A number of different schedulers, including parallel schedulers, have been developed for this model of computation.

SDF is a special case of the data flow model. In the terminology of the data flow literature, primitives are called actors. An invocation of the go() method of a primitive is called a firing. Particles are called tokens. In a digital signal processing system, a sequence of tokens might represent a sequence of samples of a speech signal or a sequence of frames in a video sequence.

When an actor fires, it consumes a number of tokens from its input arcs, and produces a number of output tokens. In synchronous dataflow, these numbers remain constant throughout the execution of the system. It is for this reason that this model of computation is suitable for synchronous signal processing systems, but not for asynchronous systems. The fact that the firing pattern is determined statically is both a strength and a weakness of this domain. It means that long runs can be very efficient, a fact that is heavily exploited in the code generation domains. But it also means that data-dependent flow of control is not allowed. This would require dynamically changing firing patterns.

V. FSM AND CONCURRANCY DOMAINS

In MLDesigner, a finite state machine is always combined with other MLDesigner Models, since an FSM Model is always embedded into a wormhole of a concurrency domain or different MLDesigner Models can be used as a Slave Process inside an FSM Model [6]. This section describes, how FSM Models interact with the Discret Event domain, the Synchronous Data Flow (SDF) domain and the Finite State Machine (FSM) domain, in the case, an FSM Model is used as a Slave Process Model.

A. FSM AND DE

The MLDesigner DE domain uses an event driven Model of computation. Events occur at a point in time. A time stamp, possessed by every event, indicates the time, at which the associated event occurs.

An FSM Model embedded in a DE domain environment behaves like any other DE Model. New data on an FSM Model Input port represent the presence of an new event to trigger the FSM Model and new data on an FSM Model Output port, generated during the execution, are interpreted as new events for the Discret Event environment, whereas the FSM Model reacts to the outer DE domain as a zero delayed system. In this context, Output events, generated by the FSM Model, get the same time stamp as the Input event, which triggered the execution of the FSM Model. A Discret Event outer domain is the only case where a FSM Model uses SpecialEvent arguments to cause State changes inside the finite state machine, because SpecialEvent arguments, possessed by an FSM Model, are scheduled by the outer domain and the DE domain is the only one, which supports SpecialEvent arguments. The Event Expression of non-synchronous Transitions should only consist of a single Event Name, if the associated FSM Model is embedded in a Discret Event environment. The FSM Model reacts every time, either an Input event or SpecialEvent occurs and all the events underlie an OR condition.

B. FSM AND SDF

A SDF system consists of a set of modules or primitives interconnected by directed arcs. MLDesigner SDF Models represent computational functions that map Input data into Output data. Unlike the DE domain, the SDF domain is not event driven and there exist always data on each Input and Output port of the SDF Model.

An FSM Model, embedded into an SDF domain environment, behaves like any other SDF Model. To ensure this behavior, the FSM Model needs an approach to differ between the presence and absence of an event, since there exist always data on each Input port of the FSM Model. In this context, the FSM Model determines the presence and absence of an event via the integer cast of the appropriate Input data. If the integer cast returns zero, the associated event is interpreted absent and in the case of a non zero result, the associated event is interpreted present. If the FSM Model execution produces no data for an associated Output port, a zero valued data is placed on this Output port, to ensure that there are always data available on each FSM Model Output port, as required by the semantic of the outer SDF domain.

VI. EXAMPLE

The first example is to modeling the system that simulates the turning on and off a lamp with just one button (toggle). The purpose of this example is introduction to the FSM model and his work with the DE domain and the SDF domain.

It is necessary to model the FSM, which will be universal for both DE and SDF domain. There are one difference, because in DE domain we need to add another entry, which will be put "clock" signal to the system so it could able to function properly. Systems which runs the FSM for different domains are different, but their principle of operation are the same.

Figure 1. shows the FSM. It is known that the lamp can be placed in the 4 possible states. The first situation is when disabled or when the button is not activated (State0). If the button is activated system cross to other state, "State01" which comes to turning on lamp or light. In this state the lamp remains until dismissal button. When this happens the system is going into the third state in which the

lamp lights and the button is not active (State02). Now, the process will go back to initial state, but through condition number four (State03). Of course in this state when it comes to re-activate the button, it comes to the exclusion off light at the lamps. The release button returns the system to the first condition, the initial state.

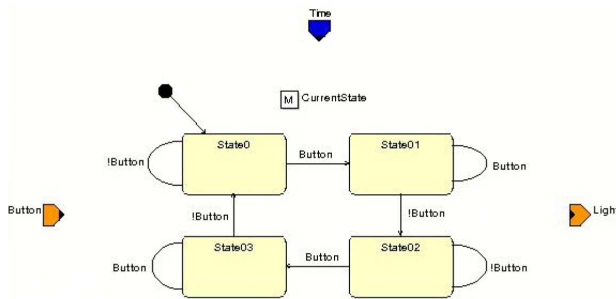

Fig. 1. FSM of Turn ON/OFF light

Framed rectangular shapes symbolize the state in which the FSM is located. Each of the state is necessary to define in the program and set the command Entry Action. In this case we have only to define the output "Light" with:

WriteOutput(Light,0); for first and fourth condition (State0 i State03)
WriteOutput(Light,1); for second and third condition (State01 i State02)

The lines that are between states represent state change in the FSM and called Transitions. Words that are next to these lines denote the defined entrance to the FSM state in the command Event Expression. Which means that the state make changes if a corresponding signal appears on the input of FSM, or FSM will remain in a given state until the input signal is defined (this goes for the line that starts and ends in the same condition and that line is called Self Transition). Of course if you only state the name of the signal then it expects to be or to appear in its logic unit, and if standing in front of the name "!" it's a logical zero.

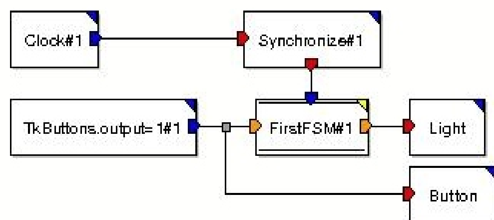

Fig. 2. DE System of Turn ON/OFF light

System for DE domain is shown in Fig. 2. In the FSM model, you need to add models that represent the excitation of system and models that can track signals in the system and all the models in this case are from the DE library in different sublibrary. Block "Clock" is a model of the system clock signal and is in sublibrary Sources. After this block is a block "Synchronize" which serves to synchronize the clock signal with real time (synchronizes simulation time with the system time) and should drag it from sublibrary Timing. Block "TkButtons" is toggle button model and with it signals are specify (sublibrary TclTk). Besides these there are two models "Button" and "Light", which serve to show the given signal and simulation results respectively (sublibrary Sinks / Xgraph).

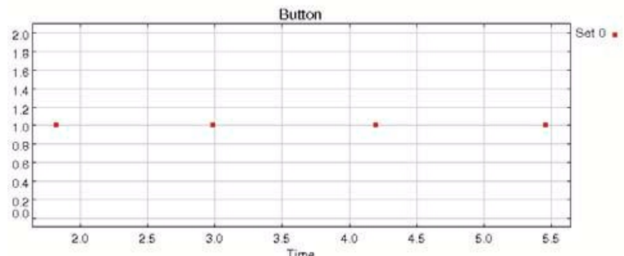

Fig. 3. Input

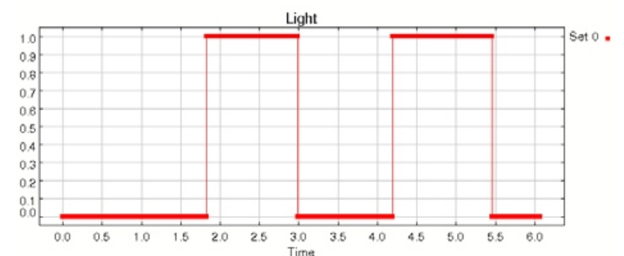

Fig. 4. Output

In Figures 3. and 4. are display the incentive and the simulation results. Comparison of time these two graphics can be seen that the time in which activated the button corresponding to the time of turning on lamps, and after another pressing of the button it comes down (lamp is turning off). This task is fulfilled in the DE domain.

Using exactly the same FSM (except removal TIME entrance), the system were designed in the SDF domain, which is shown in Fig. 5. Models of which is this system made up were used from different sublibrary of SDF library. Models, "Button" and "Light" have the same purpose as in the DE area and were taken from the same sublibrary, but only in the SDF domain (sublibrary Sinks/Xgraph).

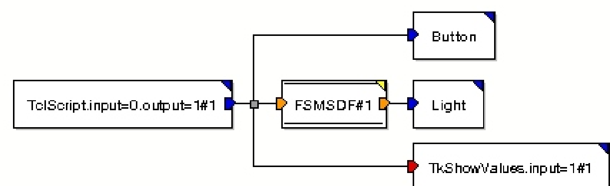

Fig. 5. SDF System of Turn ON/OFF light

For the purposes of seting signals, used models are “TclScript” and “TkShowValues", although we could use

the model of "TkButton". Unlike the previous examples, where we can simply to apply "TkButton", with "TclScript" model we need to write a small program to describe the function of button. Code for this program:

```
set s $ptkControlPanel.middle.button_$starID
if {! [winfo exists $s]} {
  button $s -text "Push Me!!!"
  pack append $ptkControlPanel.middle $s {top}
  bind $s <ButtonPress-1> "setOutputs_$starID 1.0"
  bind $s <ButtonRelease-1> "setOutputs_$starID 0.0"
  setOutputs_$starID 0.0
 }
unset s
```

This code is placed in the file with the extension ".tcl" and then we must connect "TclScript" with him through the function Tcl_File. In addition to this model, the model "TkShowValues" was used also, and serves to, when we run simulations and button appears, below it display the value of a button in real time, ie, when the button is not active "0.0", and when is "1.0".

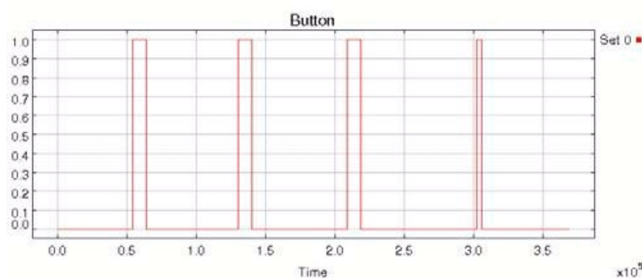

Fig. 6. Input

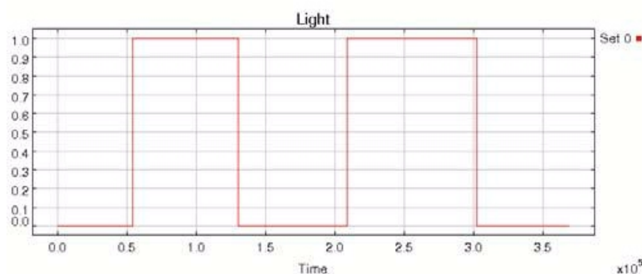

Fig. 7. Output

Figures 6. and 7. represent the incentive system and the simulation results. The first figure shows points when the button is activated and how long was the squeeze, and the other when and how long the lamp was included. Based on this basic insight, is gained some knowledge in SDF domain and in the use of some other models.

VII. Conclusion

This paper presents research results after praxis at the Ilmenau University of Technology. First, the different domains are analysed. Modules for a few examples were developed. It is used the FSM-Domain which could be embedded in the DE-Domain and the SDF-Domain. Later the both solutions, modelling techniques, in these different domains were analysed and compared.

References

[1] D. Harel, "Statecharts: A visual Formalism for Complex Systems", Science of Computer Programming, 8/1987, North Holland, pp. 231-274

[2] http://www.mldesigner.com

[3] V. Zerbe, "Mission Level Design of Complex Autonomous Systems", in Proc. XLVII ETRAN Conference, Herceg Novi (Montenegro), 2003, pp. 55-59

[4] MLDesigner Documentation, version 2.4, 2003.

[5] V. Zerbe, "Systematischer Entwurf paralleler digitaler Systeme", in Proc. Workshop Boolesche Probleme, Freiberg (Germany), 07. Oct. 1994, pp.73-79

[6] H. Rath, „Specification of the MLDesigner Finite State Machine Model, Student Research Project, Ilmenau University of Technology 2002