System Level Specification for Multimedia Applications

Dohyung Kim
Department of Computer Engineering
Seoul National University
Seoul, Korea 151-742
dhkim@iris.snu.ac.kr

Soonhoi Ha
Department of Computer Engineering
Seoul National University
Seoul, Korea 151-742
sha@iris.snu.ac.kr

Abstract

This paper presents a system level specification for codesign of control modules and function modules in multimedia applications which have significant computational requirements as well as complex control systems. In the proposed compositional approach, control modules and function modules are separately described with extended FSM, called fFSM, and dataflow respectively, and integrated in the top-level codesign backplane. The codesign backplane supports interaction requirements between control and function modules, enables cosimulation of their composition, and will provide cosynthesis. Since fFSM and dataflow models are formal models of computation, we can analyze the specification of each module. Comparison with previous approaches and preliminary experiments show novelty and practicality of the proposed technique.

1 Introduction

As the system complexity of electronic digital systems grows, a new systematic design methodology, called codesign, has been sought for targeting diverse architectures from system-on-chip to distributed heterogeneous systems [1]. The common feature of codesign procedure is to co-specify, co-simulate, and co-synthesize modules with different characteristics, among which the hardware and the software modules have gained most attention, in the same framework. Recently, codesign of analog modules and digital modules, or codesign of processing engines and memory modules also attracts attention. In this paper, we are interested in codesign of control modules and function modules in an embedded system.

Codesign is a system level design, which evaluates the influence of a design decision for one part of the system on the other parts to make a system-wide optimal decision. Using a formal model of computation as a specification model, therefore, is very advantageous in any useful codesign framework since the formal computation model makes both evaluation and automatic synthesis easier.

The specification languages used in the codesign researches heavily depend on the application areas. For computation-oriented applications such as digital signal processing (DSP) applications, many researchers use dataflow graphs as specification languages since the dataflow model is well suited to describe the algorithmic function flow of such applications. Among various dataflow models, some models such as synchronous dataflow (SDF)[2] and cyclo-static dataflow (CSDF)[3] are used in many DSP system design environments since they furnish better analytical properties due to restricted semantics.

On the other hand, control-oriented applications, such as automatic control systems, make the Finite State Machine (FSM) or its variants favored as the specification language. Because of their finite nature, FSMs yield better to analysis and synthesis than alternative control models, such as a sequential program with if-then-else and goto. For example, a designer can enumerate the set of reachable states to ascertain that a particular dangerous state can not be reached. The same question may be undecidable in a richer language. In spite of these advantages, FSM has a big weakness. If the system has concurrency or memory, the number of states would explode. Many researches try to combine concurrency with FSM to overcome the state explosion problem. The POLIS approach [4] specifies the system as the network of Codesign FSMs (CFSM).

Starting from different specification models, two groups of codesign researchers have developed separate codesign environments for different application areas. But there emerges a new class of applications that contain significant amounts of computation modules and control modules simultaneously. For example, a cellular phone includes a substantial amount of embedded control logic for call processing and multiple access protocol. It also performs major signal processing tasks such as coding and modulation. Ignoring the analog subsystem of a cellular phone, we believe that any system-level design tool targeting such system should support specification models for dataflow and control in the same framework.

In this paper, we present the system level specification methodology which combines control and function modules on the *codesign backplane* [5]. The proposed methodology enables us to design control modules and function modules separately and provides communication protocols that support control modules to change the status of function modules. And we introduce an FSM extension which is very flexible and does not loose formality. *Flexible* FSM supports concurrency, hierarchy and internal event as Statechart does. And it includes a method to express memory in FSM. STATEMATE of i-Logix inc.[6] and the Ptolemy [7] approaches are previous works closely related with our proposed compositional approach. In a compositional approach, each subsystem can be designed, validated, and synthesized independently of one another, and can then be composed in a way that the composition can also be analyzed, validated and synthesized.

In the next section, we overview the related works and highlight the key features of the proposed approach compared with previous approaches. Section 3 introduces the codesign backplane which is the heart of our codesign environment. It includes subsections to describe a formal dataflow model for function modules, the extended FSM model for control modules, and their interaction. We will show the novelty and practicality of our work through experiments and conclude the paper in sections 4 and 5 respectively.

2 Previous Work

In this section, we overview the existent approaches to support control and dataflow specifications in the same framework. They are STATEMATE based on Statechart [8], Ptolemy, COSY [9].

The STATEMATE [6] consists of three charts: statechart, activity chart, and module chart. The module chart represents the system architecture, in which a block represents a system component and an arc represents physical connectivity between components. The statechart, an innovative extension of FSM, is the major module that depicts reactive behavior over time. The activity chart, corresponding to the codesign backplane in our context that will be discussed in the next section, describes the functional decomposition and the information flow of the system. Since the STATEMATE was originally developed for control-oriented reactive applications, the activity chart is not an independent specification of the statechart. Also, the lack of formality in both the activity chart and the statechart lose all advantages of formal representations for analysis and automatic synthesis.

The Ptolemy software environment [7] supports multiple models of computation including dataflow and FSM models. Using the object-oriented principle of polymorphism, the block diagram representation is assigned a specific model of computation according to the will of the application designer. The block diagram specification with a specific model of computation is called a *domain* in Ptolemy. The key innovative idea of Ptolemy is to use a hierarchical block diagram as the means of inter-domain interactions. A hierarchical block contains another block diagram inside. If the domain of the inside block diagram is different from the outside domain, the hierarchical block is called a *wormhole*.

Inter-domain interactions occur on the wormhole boundary. If a dataflow graph contains a wormhole of FSM domain, an execution of the wormhole consumes input samples from the neighbor dataflow nodes and sends them to the inside FSM. The inside FSM makes a state transition and sends output samples to the outside dataflow graph. If an FSM contains a wormhole of SDF domain, the state transition to the wormhole invokes the inside dataflow graph like a function call. Thus, synchronous interaction between a dataflow graph and an FSM occurs at the wormhole boundary.

Ptolemy FSM model does not support concurrency since other domains can already represent the system concurrency. If there is a need of concurrent FSMs, they make a dataflow graph to specify two concurrent wormholes of FSM domain inside. To preserve formality in the composition of dataflow and FSM models, however, they define the interaction mechanism very carefully as discussed in [10]. Another limitation of the Ptolemy approach is that they do not support asynchronous interaction, which will be described later section.

COSY [9] is developed for IP-based real-time design methodology. The goal of COSY is to specify the mixture of control and dataflow processing and to re-use parts of existing design. COSY uses an extension of Kahn process network, called Y-chart API (YAPI) as the specification language. YAPI model consists of a network of processes connected by FIFOs. A process has *read*, *write* and *select* port operations. Read and write operations are the same as those in a Kahn process network. Select operation is added to model the interference of the control path and data path. It takes two input ports, a data port and a control port for an instance, and returns a port ID. If both input ports have data available, “select” chooses one of them non-deterministically.

COSY approach is similar to our codesign backplane. Both provide a coordination method to compose systems with different components. But COSY does not specify any formal model inside a component process. An IP or a user-defined process is used as an atomic design entity.

In short, no previous approach satisfies all requirements of the formal composition of dataflow and control. The STATEMATE lacks formal properties and the Ptolemy approach does not consider asynchronous interactions. COSY supports only interactions between component processes without formal specification inside a process. The proposed technique is to use the codesign backplane to solve all problems at the same time.

3 Codesign Backplane

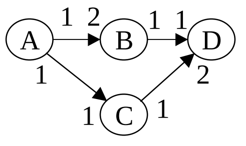
The codesign backplane is a central backbone for intermodel communications. All components communicate through the backplane that has all controls over the components. While the codesign backplane supports cospecification, cosimulation and cosynthesis, we focus on specification only in this paper.

We divide the system design into control specification and function specification. Then the communication protocol is defined to support interactions between them. We propose asynchronous communication, which allows the control module to change the status of function modules.

Subsection 3.1 and 3.2 describe the specification languages for function modules and control modules respectively. Then subsection 3.3 explores communication requirements between them. Finally subsection 3.4 shows the implementation of the defined communication protocols.

3.1 Specification of Function Modules

We specify function modules using dataflow graphs. In a dataflow graph, a node represents a function block and a directed arc between two nodes represents the flow of data samples. When a node is executed (or fired), it consumes a certain number of samples from the input arcs and produces some samples to the output arcs. If we fix the number of samples produced or consumed on each arc for each node firing, the model becomes the synchronous dataflow (SDF), a restricted dataflow model of computation. Figure 1 shows an example SDF graph and a possible scheduling result as well as buffer requirements. In figure 1(a), numbers on an arc represent the numbers of samples produced and consumed per each firing of source and destination nodes. Nodes are executed repeatedly and their repetition ratios are determined not to accumulate samples on arcs. With such restrictions, we can determine the execution order of nodes and the buffer requirements on arcs through compile-time analysis, which is very desirable for embedded system design.



(a)

Schedule: A-C-A-C-B-D
Buffer size: AB(2), AC(1),
CD(2), BD(1)

(b)

Figure 1: (a) An example SDF graph and (b) its schedule and buffer requirements on arcs. "AB(2)" means that the required buffer size for arc AB is 2.

However, the SDF model loses the expressive power of general dataflow: for example, it can not model a dynamic behavior where a node produces samples on an output arc conditionally. To overcome this limitation, we use graphical dynamic constructs (or, well-behaved schema) such as if-then-else, while, and for constructs without sacrificing analytical properties [11]. In this paper, we concentrate on the SDF model for brevity.

3.2 Specification of Control Modules

The pure FSM representation suffers from *state explosion problem* to represent a system with concurrent modules or memory. To avoid this problem, Harel introduced Statechart model [9] where FSMs can be combined in an "*or*" fashion with hierarchical FSMs or in an "*and*" fashion. He also introduces the notion of internal event that supports communication between concurrent states and/or hierarchical states. Since the Statechart does not rigorously specify the dynamic behavior of "and" states, a number of variations have been fostered [12].

There are other FSM extensions, mainly focusing on how to express concurrency. Codesign FSM (CFSM) [13] describes the system with a network of concurrent FSM modules. Hierarchical FSM (HFSM) [10] expresses the concurrency with other models of computation in which a concurrent entity embeds each FSM module inside.

We propose another FSM extension, called flexible FSM (fFSM), which is as expressive as Statechart but preserves the formal properties like CFSM. Moreover, it has a special syntax to express memory in a compact form. In a formal FSM model, if a system has a memory element, the number of states explodes as the number of possible values. We introduce a new concept of "variable states" to overcome this difficulty. Since the proposed fFSM is embedded in the codesign backplane that is a concurrent model of computation, it also resembles HFSM to a limited extent. Thus the fFSM model is flexible to take the benefits of previous approaches: expressive power from Statechart, formal properties from CFSM, and inter-domain interaction from HFSM.

Concurrency

fFSM expresses concurrency as illustrated in figure 2. Bold lined states indicate initial states and a doted-line divides two concurrent FSMs. Two states A and C are concurrently active at the same time. And transitions may occur simultaneously if both inputs "a" and "c" are active. No direct transition is allowed between concurrent states. However, two concurrent states can affect each other by an internal event to be explained below. This constraint is for formality and simplicity.

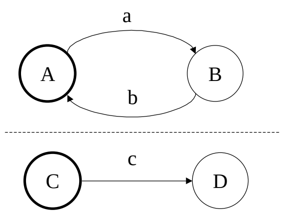


Figure 2: Concurrency in fFSM

Hierarchy

Hierarchy means that a state can have another FSM inside. When the outer state is active, the internal FSM becomes active and initial state is ready to execute. And as soon as the outer state is inactive, the internal FSM also becomes inactive. We avoid the use of inter-hierarchy transitions, unlike Statechart, to provide the abstraction between hierarchies. Figure 3 shows an example fFSM graph that shows hierarchy.

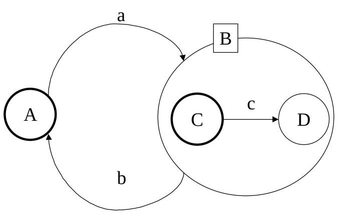


Figure 3: Hierarchy in fFSM

If the current state is A and input event “a” occurs, the next state becomes the state B and internal FSM is active, which makes the state C active. And input event “b” occurs, the internal FSM exits to go to state A.

But it has an ambiguity problem. Suppose that the current states are B and C and input events “b” and “c” occur simultaneously. Which transition should be executed first? There are a number of methods to determine the priority between transitions. We give an outer transition higher priority.

Internal event

Internal event is used for communication between concurrent states and/or hierarchies like Statechart. In CFSM, it is not allowed because it may cause ambiguity. We remove ambiguity by adopting the *delta-delay* model as the VHDL simulation does. All input events exist during delta-delay phase only. If there is any new internal event, it becomes valid at the next phase after delta-delay and all previous events set to be invalid. Harel also solved this problem in his recent paper [14], which is similar to our approach.

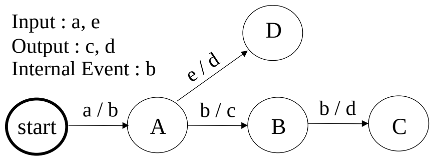


Figure 4: Internal event example in fFSM

Figure 4 shows how we can avoid ambiguity by applying the delta-delay model of execution. If a current state is *start* and input event “a” and “e” occur, an internal event “b” becomes valid and the current state is changed to a state *A*. We can not determine which transition is executed if the current state is A and events “e” and “b” are active. But we assume that event “e” is deactivated after delta-delay and the next state will become B.

Variable state

A variable state provides a *memory* to FSM model. It is considered as a local variable and expressed as a concurrent FSM. The usage of a variable state is similar to an internal event. While an internal event disappears after delta-delay, the variable state always exists. A variable state is formally equivalent to a concurrent FSM except that it can be examined and updated by other concurrent FSMs, not by itself. If it is used as a condition, it is called a *state reference*. If it is used as a output, it implies a *state transition*.

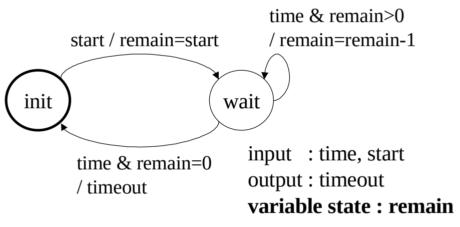


Figure 5: Timeout example using variable state

Figure 5 implements a time-out FSM using a variable state. An input event “time” is an external clock and an output signal ‘timeout’ is the result output which indicates that actual timeout occurs. If an outer block wants to set a timeout, it sends “start” signal to the timeout FSM with a required timeout value. After the timeout FSM gets the input, it sets the variable state “remain” as the value of the input event “start”, changes the state to “wait”. It decreases the “remain” state at each “time” input until the “remain” value becomes zero. Then a transition from “wait” to “init” occurs and a “timeout” signal is supplied to the outer block. If we specify this example by using previous FSM models, we must create different FSMs with different time-out values.

We can prove the formality of fFSM by translating fFSM to FSM. But it is beyond the scope of this paper thus omitted.

3.3 Composition of FSM and Dataflow

To define the communication requirements between dataflow and FSM models, we examine the interaction patterns between control modules and function modules in typical embedded systems. We do not confine to any specific implementation. A dataflow graph and an FSM may be integrated as a single process, or synthesized as separate processes interacting with each other. A dataflow graph itself can be divided into multiple communicating processes (or threads). Regardless of implementations, interaction patterns between control modules and function modules can be divided into two categories: one is *synchronous interaction* and the other is *asynchronous interaction*.

In a synchronous interaction, a control module and a function module communicate with each other by exchanging data samples. The function module may send a sample to set a flag of a control register and the controller becomes active on the arrival of the flag. Or, the control module may send a data sample which activates a function module to process the sample. A simple implementation of this scenario is calling a procedure with the FSM output sample as an argument data like figure 6(a). Since the receiving module waits for the arrival of samples at a certain state, the interaction is synchronized with sample exchange.

Using asynchronous interactions, a control module plays the role of a supervisory module to manage the state of a dataflow module. We define three states of a dataflow module by its current activity: *active, suspend* and *stop* as illustrated at figure 6(b)-(i). If there is no synchronous input interaction from the FSM module to a dataflow graph, the dataflow module goes into the active state from the start by default. When the control module enters into a certain state, it may want to suspend the dataflow module and resume it later. When the dataflow module goes into the suspend state, it stops its execution and just discards the incoming samples from the outside environment.

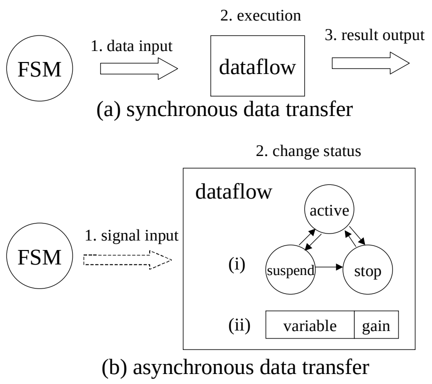


Figure 6: communication requirements between FSM and dataflow

Another situation of asynchronous interaction occurs when the control module wants to change parameters of dataflow nodes asynchronously. Suppose a dataflow module plays an encoded audio. If the user wants to lower the volume, that action is delivered to the control module and finally to the dataflow module by asynchronously changing the "gain" parameter of the dataflow node that amplifies the sound samples (figure 6(b)-(ii)).

Depending on the role of the control module, either interaction is prevailed. If an FSM describes the control structure inside a function block, the interaction pattern is likely to be synchronous. On the other hand, a supervisory control module will have asynchronous interactions with function modules. Therefore, composition of FSM and dataflow modules should support both synchronous and asynchronous interactions.

3.4 Cospecification Implementation on Codesign Backplane

The codesign backplane manages all interactions between different models of computation. Therefore, a model of computation needs not concern about direct interaction with other models but only with the backplane. We adopt the discrete-event model of computation [15] for the codesign backplane. In the discrete-event model, there is a notion of global time and all data samples carry time-stamps to indicate when they are produced. Therefore the simulation engine in the codesign backplane collects, sorts all output samples from the source blocks into a global queue, and transfers the samples to the destination blocks in the chronological order.

We implement the proposed technique in our codesign environment PeaCE(Ptolemy extension as a Codesign Environment) that is based on the Ptolemy software environment. We make a new "backplane" domain for the codesign backplane. Dataflow graphs and FSMs are encapsulated as wormholes in the backplane domain. The interactions between dataflow graphs and FSMs should go through the top-level backplane domain.

The synchronous interaction is represented by directed arcs between two hierarchical blocks in the codesign backplane, which contains dataflow and FSM models inside as shown in figure 7. The input arc to the dataflow block is connected to the inner dataflow node A since the input data sample will be delivered to that node. However, the arcs connected to the FSM block do not penetrate the block boundary to the inside. Instead, a sample on the input arc (“flag”) is interpreted as an input event to the FSM triggering the state transition from Y state to X state. On the other hand, the state transition from X to Y generates an output sample (“out”) which is delivered to the SDF block via a direct arc.

Figure 7 also shows two atomic blocks in the backplane domain: *Event source* and *Display*. The *Event source* block generates input events or the test vectors for system simulation. It can be regarded as the input interface module of the system to the environment. On the other hand, the *Display* module displays the simulation outputs from the dataflow module. In short, the backplane domain facilitates simulation-aid blocks for event generation and result processing in the same framework, which is a side benefit of using the backplane domain.

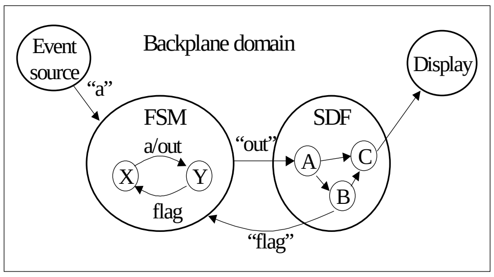


Figure 7: Codesign backplane and an illustration of a synchronous interaction of an FSM and a dataflow graph.

Though there are many possible methods to describe asynchronous interactions, we choose to use a "script" language inside a state node in an FSM. Table 1 shows the syntax of the scripts. The first three manage the states of the dataflow modules and the last script describes the asynchronous parameter updates of dataflow nodes. The difference between "suspend" and "stop" command is that the suspended block resumes its action starting from where it was suspended while the stopped block resumes from the beginning when resumed.

How to accomplish an asynchronous interaction is shown in figure 8. The codesign backplane receives a script command from the FSM and interprets it to send a special control signal to the specified dataflow block whose "node_name" property is equal to the *n_node* field of the script. However this control signal path is not drawn explicitly in the graphic editor. There are hidden connections between the simulation engine of the codesign backplane and all blocks in the backplane. The inside SDF domain checks the arrival of the special control tokens regularly. If a special token is received, it changes the graph activity as indicated. Otherwise, it continues to execute the graph.

Table 1: Scripts for asynchronous interaction

Scripts	Actions
Run *n_name*	Resume the *n_name* block
Suspend *n_name*	Suspend the *n_name* block
stop *n_name*	Stop the *n_name* block.
Set *n_name parameter value*	Update the *parameter* with *value* in the *n_name* block

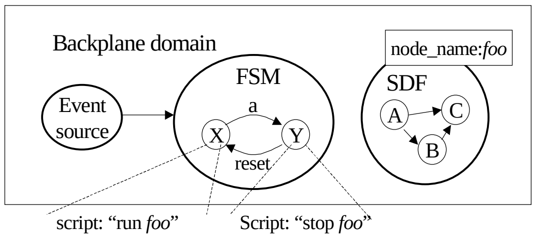


Figure 8: Asynchronous interaction in the backplane domain.

4 Examples

In this section, we use two examples. A traffic light example shows that our fFSM successfully can specify complex reactive systems. Our second example is MPEG 1 audio (layer 3) player which has significant computation requirements and a supervisory module. Codesign backplane not only supports the co-specification of control modules and function modules but also enables co-simulation of software C process and the codesign backplane.

4.1 Traffic Light

The traffic light example is composed of an fFSM block, user interface blocks and a clock generator. Top level specification on the codesign backplane is shown in figure 9.

The FSM block gets user inputs and the clock. It changes the light display. The traffic light for the car changes the value as yellow, red and green in sequence. When the traffic light for the car is red, the traffic light for the pedestrians becomes green and shows blinked green to indicate the end of green-light. And it changes to red when the traffic light for the car goes green. And it can be manually controlled or operated automatically by the buttons.

The signal FSM has four fFSMs inside as illustrated figure 10. Top level FSM has two concurrent FSMs. One determines the operation between the automatic state and the manual state. Each has an internal fFSM. The other provides time-out signal by the internal event. "AutoFSM" state provides automatic signal change by using time-out signal. It has another internal fFSM (blinkFSM) in the red state, which provides green light blinking for pedestrians. "ManualFSM" state changes its state by the user input.

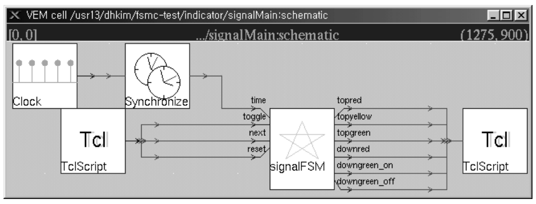


Figure 9: Top level specification of a traffic light example on the codesign backplane

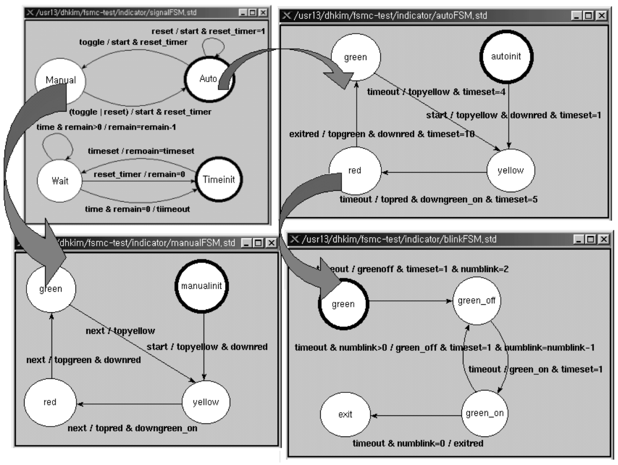

Manual
Auto
Wait
Timeinit
reset / start & reset_timer=1
toggle / start & reset_timer
(toggle | reset) / start & reset_timer
time & remain>0 / remain=remain-1
timeset / remoain=timeset
reset_timer / remain=0
time & remain=0 / tiimeout
green
autoinit
red
yellow
timeout / topyellow & timeset=4
start / topyellow & downred & timeset=1
exitred / topgreen & downred & timeset=10
timeout / topred & downgreen_on & timeset=5
manualinit
next / topyellow
start / topyellow & downred
next / topgreen & downred
next / topred & downgreen_on
green_off
green_on
exit
timeout / greenoff & timeset=1 & numblink=2
timeout & numblink>0 / green_off & timeset=1 & numblink=numblink-1
timeout / green_on & timeset=1
timeout & numblink=0 / exitred

Figure 10: fFSM modules of traffic light example

This example uses concurrency, hierarchy, internal event and variable state altogether to specify a rather complex control system with only 16 states and 19 transitions.

4.2 MPEG 1 Audio (Layer 3) Player

In this example, we implement the MPEG 1 Layer 3 audio player (MP3 player)[16] in which the decoding algorithm is depicted as a dataflow graph and synthesized into a C-code at compile-time (figure 11). The FSM control module resides in the “controlFSM” block and the TclScript block generates user control events to the FSM block. The MP3 decoder reads an MP3 file and decodes the encoded samples through a series of Huffman decoder, Dequantizer, and other function blocks. The FSM block has three operations. One is to control the execution of the MP3 decoder. Figure 11 shows that MP3 decoder has “start”, “stop” and “suspend” execution states. Another is to control the “volume” value in the MP3 decoder asynchronously. Final operation is to display the current volume at the user interface.

An MP3 decoder is synthesized and compiled as a separate C process at compile-time, from the initial dataflow specification in the codesign framework. At run-time, the C process and the codesign framework communicate with each other via TCP/IP sockets. From this communication channel, the special control signal samples are delivered to the C process to suspend or resume the decoding. In this experiment, we play the MP3 file in the Sun Ultra1 machine. When we generated an event to the dateflow module to suspend the decoding, we noticed that the play was stopped a few seconds later. This was because the decoding speed is fast enough to accumulate the music samples in the output buffer to the speaker.

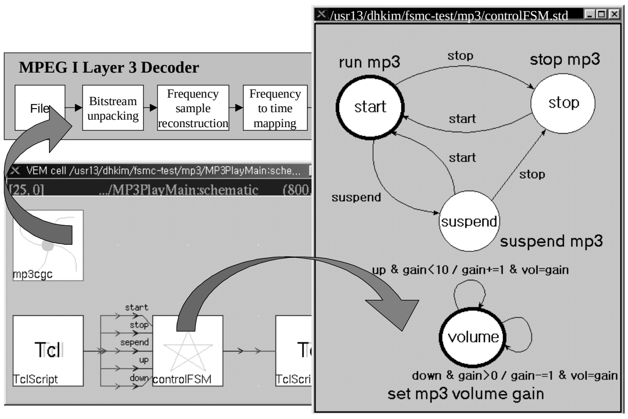


Figure 11: MPEG 1 Layer 3 (MP3) Audio Player example. In this example, the MP3 decoding algorithm is automatically synthesized into a C code. This example, thus, demonstrates a cosimulation between the C process and the codesign framework PeaCE.

4.3 Comparison with Other Methodologies

Because STATEMATE is very expressive, two examples can be specified in that environment. But the differences come from formality. In our approach, all components on the codesign backplane are formally specified. So we can apply formal verification methods on the specification at the early stage of design. Moreover, we do not impose any implementation restriction on the specification and the target environment. Each component can be implemented in software, hardware or both.

If we implement the traffic light example using Ptolemy, every timeout signal must be implemented by a separate dataflow graph inside a state which uses a timeout signal. And the scheduling control in MP3 player is basically impossible in Ptolemy. It may only start or stop MP3 player if the decoding dataflow graph is specified inside a state of HFSM. The volume control can be expressed by using peek/poke stars [17] with block modification.

Select operation in COSY can specify the MP3 player with the volume control and the scheduling control. But it does not provide formal specification languages for internal MP3 decoding algorithm.

5 Conclusion

We present a system level design specification for codesign of control modules and function modules in multimedia applications. In the proposed compositional approach, control modules and function modules are separately described with FSMs and dataflow graphs respectively, and integrated in the top-level codesign backplane. The codesign backplane implements both synchronous and asynchronous interaction requirements between control and function modules.

And we extend FSM model and propose the fFSM model. The model supports concurrency, hierarchy, internal event and variable state. It successfully overcomes the state explosion problem but does not lose formality.

The proposed technique is implemented in our codesign framework, PeaCE, based on the Ptolemy environment. Through comparison with the related works and experiments, the novelty and practicality of our technique is demonstrated. Since we preserve the formality of individual specifications, we envision that the cosynthesis of control and dataflow will be achieved in a systematic way, which is the on-going research topic. The tough problem we have to attack is not to make the cosynthesis task doable but to perform it as efficiently as manual synthesis.

References

[1] Chiodo, M.; Giusto, P.; Hsieh, H.; Jurecska, A.; Lavagno, L. Sangiovanni-Vincentelli, A.; *A formal methodology for hardware/software codesign of embedded systems.* IEEE Micro, August 1994.

[2] Lee, E.; Messerschmitt, D.; *Synchoronous data flow*. Proceedings of IEEE, Vol. 75, No. 9, pp. 1235-1245, 1987.

[3] Bilson G.; Engels, M.; Lauwereins, R.; *Cyclo-static dataflow.* IEEE Trans. on Signal Processing, Vol. 44, No. 2, pp. 397-408, February. 1996.

[4] Balarin, F.; Chiodo, M.; Giusto, P.; Hsieh, H.; Jurecska, A.; Lavagno, L.; Passerone, C.; Sangiovanni-Vincentelli, A.; Sentovich, E.; Suzuki, K.; Tabbara, B.; *Hardware-Software Co-Design of Embedded Systems: The Polis Approach.* Kluwer Academic Press, June 1997.

[5] Sung, W.; Ha, S.; *Efficient and flexible cosimulation environment for DSP applications.* IEICE Transactions on Fundamentals of Electronics, Communications and Computer Sciences, Special Issue on VLSI Design and CAD algorithms, Vol.E81-A, No.12, pp. 2605-2611, December, 1998

[6] Harel, D.; Lachover, H.; Naamad, A.; Pnueli, A.; Politi, A.; Sherman, R.; Shtull-Trauring, A.; Trakhtenbrot, M.; *Statemate: a working environment for the development of complex reactive systems.* IEEE Trans. on Software Engineering, Vol. 16, No. 4, April 1990.

[7] Buck, J.; Ha, S.; Lee, E.; Messerschmitt, D.; *Ptolemy: a framework for simulating and prototyping heterogeneous systems.* International Journal of Computer Simulation, Vol. 4, pp. 155-182, 1994.

[8] Harel, D.; *Statecharts: a visual formalism for complex systems.* Sci. Comput. Program, Vol. 8, pp. 231-274, 1987.

[9] Brunel, J-Y. et al.; *COSY: a methodology for system design based on reusable hardware & software IP's,* in J-Y. Roger (ed.), Technologies for the Information Society, pp. 709-716, 1998.

[10] Girault, A.; Lee, B.; Lee, E.; *Hierarchical finite state machines with multiple concurrency models.*, IEEE Transactions on CAD, Vol. 18, No. 6, pp. 742-760, June 1999.

[11] Ha, S.; Lee, A.; *Compile-time scheduling of dynamic constructs in dataflow program graphs.* IEEE Trans. on Computers, Vol. 46, No. 7, pp. 768-778, July 1997.

[12] von der Beeck, M.; *A comparison of statecharts variants.* Proceedings of Formal Techniques in Real Time and Fault Tolerant Systems, LNCS 863, pp. 128-148, Springer-Verlag, Berlin, 1994.

[13] Chiodo, M.; Giusto, P.; Hsieh, H.; Jurecska, A.; Lavagno, L.; Sangiovanni-Vincentelli, A.; *A Formal Specification Model for Hardware/Software Codesign*, In Proceeding of International Workshop on Hardware-Software Codesign, October 1993.

[14] Harel, D.; Naamad, A; *The statemate semantics of statechart*. ACM Trans. on Software Engineering Method, Oct. 1996

[15] Cassandras, C.; *Discrete event systems, modeling and performance analysis.* Irwin, Homewood IL, 1993.

[16] Shlien, S.; *Guide to MPEG-1 Audio Standard*, IEEE Trans. on Broadcasting, Vol. 40, No. 4, pp. 206-218, December 1994.

[17] Pino, J.; Parks, T.; Lee, E.; *Mapping Multiple Independent Synchronous Dataflow Graphs onto Heterogeneous Multiprocessors*, Proc. of IEEE Asilomar Conf. on Signals, Systems, and Computers, Pacific Grove, CA, Oct. 31 - Nov. 2, 1994.