Formal Models for Communication-based Design

Alberto Sangiovanni-Vincentelli[1], Marco Sgroi[1], Luciano Lavagno[2]

[1] University of California at Berkeley, EECS Department, Berkeley, CA 94720
[2] Cadence Berkeley Labs, 2001 Addison Street, Berkeley, CA 94704-1144

Abstract. Concurrency is an essential element of abstract models for embedded systems. Correctness and efficiency of the design depend critically on the way concurrency is formalized and implemented. Concurrency is about communicating processes. We introduce an abstract formal way of representing communication among processes and we show how to refine this representation towards implementation. To this end, we present a formal model, Abstract Co-design Finite State Machines (ACFSM), and its refinement, Extended Co-design Finite State Machines (ECFSM), developed to capture abstract behavior of concurrent processes and derived from a model (Co-design Finite State Machine (CFSM)) we have used in POLIS, a system for the design and verification of embedded systems. The design of communication protocols is presented as an example of the use of these formal models.

1 Introduction

By the year 2002, it is estimated that more information appliances will be sold to consumers than PCs (see *Business Week*, March 1999). This new market includes small, mobile, and ergonomic devices that provide information, entertainment, and communications capabilities to consumer electronics, industrial automation, retail automation, and medical markets. These devices require complex electronic design and system integration, delivered in the short time frames of consumer electronics. The system design challenge of at least the next decade is the dramatic expansion of this spectrum of diversity and the shorter and shorter time-to-market window. Given the complexity and the constraints imposed upon design time and cost, the challenge faced by the electronics industry is insurmountable unless a new design paradigm is developed and deployed that focuses on:

- design re-use at all levels of abstraction;
- "correct-by-construction" transformations.

An essential component of a new system design paradigm is the *orthogonalization of concerns*, i.e., the separation of the various aspects of design to allow more effective exploration of alternative solutions. The pillars of the design methodology that we have proposed over the years are:

– the separation between function (what the system is supposed to do) and architecture (how it does it);
– the separation between computation and communication.

1.1 Function-Architecture Co-Design

The mapping of function to architecture is an essential step from conception to implementation. In the recent past, there has been a significant attention in the research and industrial community to the topic of Hardware-Software Co-design. The problem to be solved here is coordinating the design of the parts of the system to be implemented as software and the parts to be implemented as hardware, avoiding the HW/SW integration problem that has marred the electronics system industry for so long. We actually believe that worrying about hardware-software boundaries without considering higher levels of abstraction is the wrong approach. HW/SW design and verification happens after some essential decisions have been already made, thus making the verification and the synthesis problem so hard. SW is really the form that a given piece of functionality takes if it is "mapped" onto a programmable microprocessor or DSP. The origin of HW and SW is in behavior that the system must implement. The choice of an "architecture", i.e. of a collection of components that can be either software programmable, re-configurable or customized, is the other important

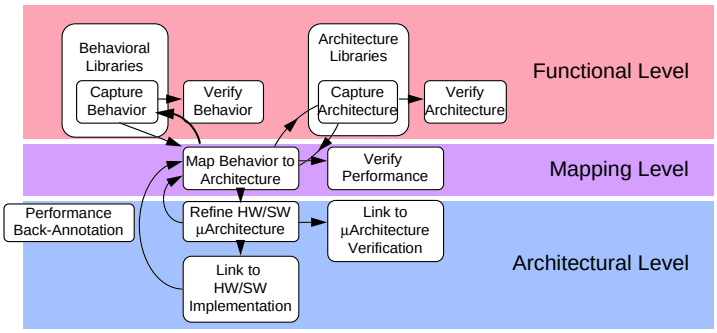


Fig. 1. Proposed design strategy

1.2 Communication-based Design

The implementation of efficient, reliable, and robust approaches to the design, implementation, and programming of concurrent systems is essential. In any large-scale embedded systems design methodology, concurrency must be considered as a first class citizen at all levels of abstraction and in both hardware and software.

Concurrency implies communication among components of the design. Communication is too often intertwined with the behavior of the components of the design so that it is very difficult to separate out the two domains. Separating communication and behavior is essential to dominate system design complexity. In particular, if in a design component behaviors and communications are intertwined, it is very difficult to re-use components since their behavior is tightly dependent on the communication mechanisms with other components of the original design. In addition, communication can be described at various levels of abstraction, thus exposing the potential of implementing communication behavior in many different forms according to the available resources. Today this freedom is often not exploited.

1.3 Formal Models

We have promoted the use of formal models and transformations in system design so that verification and synthesis can be applied to advantage in the design methodology [1]. Further, the concept itself of synthesis can be applied only if the precise mathematical meaning of a description of the design is applied. It is then important to start the design process from a high-level abstraction that can be implemented in a wide variety of ways. The implementation process is a sequence of steps that remove freedom and choice from the formal model. In other words, the abstract representation of the design should "contain" all the correct implementations in the sense that the behavior of the implementation should be consistent with the abstract model behavior. Whenever a behavior is not specified in the abstract model, the implicit assumption is that such behavior is a "don't-care" in the implementation space. In other words, the abstract model is a source of non-deterministic behavior, and the implementation process progresses towards a deterministic system. It is important to underline that way too often system design starts with a system specification that is burdened by unnecessary references to implementations resulting in over-determined representations with respect to designer intent that obviously yield under-optimized designs.

In the domain of formal model of system behavior, it is common to find the term "Model of Computation" (MOC), an informal concept that has its roots in language theory. This term refers more appropriately to mathematical models that specify the semantics of computation and of concurrency. In fact, concurrency models are the most important differentiating factors among models of computation. Ed Lee [2] has very well stressed the importance of allowing one to express designs making use of all models of computation, or at least of the principal ones, thus yielding a so-called heterogeneous environment for system design. In his approach to *simulation and verification*, assembling a system description out of modules represented in different models of computation reduces to the problem of arbitrating communication among the different models. However, the concept of communication among different models of computation still needs to be carefully explored and understood from a *synthesis and refinement* viewpoint.

This difficulty has actually motivated our approach to communication-based design where communication takes the driver seat in system design [8]. In this approach, communication can be specified somewhat independently of the modules that compose the design. In fact, two approaches can be applied here. In the first case, we are interested in communication mechanisms that "work" in any environment, i.e., independently of the formal models and specifications of the behavior of the components. This is a very appealing approach if we look at the ease of component assembly. It is however rather obvious that we may end up with an implementation that is overly wasteful, especially for embedded systems where production cost is very important. A more optimal (but less modular) approach is to specify the communication behavior, and then to refine *jointly* one side of the communication protocol and the behavior that uses it, in order to exploit knowledge of both to improve the efficiency of the implementation. Here, a synthesis approach is most appealing since it reduces the risk of making mistakes and it may use powerful optimization techniques to reduce design cost and time.

Communication and time representation in a Model Of Computation are strictly intertwined. In fact, in a synchronous system, communication can take place only at precise "instants of time" thus reducing the risk of unpredictable behavior. Synchronous systems are notoriously more expensive to implement and often less performing thus opening the door to asynchronous implementations. In this latter case, that is often the choice for large system design, particular care has to be exercised to avoid undesired and unexpected behaviors. The balance between synchronous and asynchronous implementations is possibly the most challenging aspect of system design. We argue that globally asynchronous locally synchronous (GALS) communication mechanisms are probably a good compromise in the implementation space [1]. The research of our group in the last few years has addressed the above problems and allowed us to define a full design methodology and a design framework, called Polis [1], for embedded systems. The methodology that we have proposed is based on the use of a formal and implementation-independent MOC, called CFSMs (Co-design Finite State Machines). CFSMs are Finite State Machines extended with arithmetic data paths, and communicating asynchronously over signals. Signals carry events, which are buffered at the receiving end, and are inherently uni-directional and potentially lossy (in case the receiver is not fast enough to keep up with the sender's speed). The CFSMs model is Globally Asynchronous Locally Synchronous (GALS), since every CFSM locally behaves synchronously following the FSMs semantics while the interaction among CFSMs is asynchronous from a system perspective.

However, the view of communication in CFSMs is still at a level of abstraction that is too low. We would like to be able to specify abstract communication patterns with high-level constraints that are not implying yet a particular model of communication. For example, it is our opinion that an essential aspect of communication is the guarantee of reception of all the information that has been sent. We argue that there must exist a level of abstraction that is high enough to require that communication takes place without loss. The

synchronous-asynchronous mechanism, the protocols used and so on, are just implementation choices that either guarantee no loss or that have a good chance of ensuring that no data is lost where it matters (but that need extensive verification to make sure that this is indeed the case). For example, Kahn process networks [4] guarantee no loss at the highest level of abstraction by assuming an ideal buffering scheme that has unbounded buffer size. Clearly the unbounded buffer size is a "non-implementable" way of guaranteeing no loss. When moving towards implementable designs, this assumption has to be removed. A buffer can be provided to store temporarily data that are exchanged among processes but it must be of finite size. The choice of the size of the buffer is crucial. Unfortunately deciding whether a finite buffer implementation exists that guarantees no loss is not theoretically feasible in the general case, but there are cases for which an optimal buffer size can be found [2]. In other cases, one has to hope for the best for buffer overwrite not to occur or has to provide additional mechanisms that, when composed with the finite buffer implementation, still guarantee that no loss takes place (for example, a request/acknowledge protocol). Note that in this case the refinement process is quite complex and involves the use of composite processes. Today, there is little that is known about a general approach to communication design that has some of the feature that we have exposed.

An essential step to develop communication-based design is the understanding of "communication" semantics. We believe that communication in formal models has not been treated at the correct level of abstraction.

1.4 Outline of the paper

In this paper, using as a guideline the concepts presented above, we discuss "communication semantics" and give a formal model for this semantics. Then we introduce formal models that can be used to design effectively complex systems and we present novel models obtained by "abstracting" Co-design Finite State Machines (CFSMs) to deal with the problems posed by communication-based design. In particular, in Section **??** we review the models of computation that have been most commonly used in system design. We then proceed to present in Section 3, the Abstract Co-design Finite State Machine (ACFSM) globally asynchronous locally synchronous model where communication is lossless. In Section 4, we introduce the Extended CFSM model, that refines the ACFSM model by having finite queues to buffer communication and that contains the CFSM model as a special case. Finally, in Section 6 we show how to use these models to design an application example: a wireless communication protocol.

2 Communication

Large distributed systems are composed of a set of concurrent and interacting components [3].

[3] Following the Tagged Signal Model formalism [3], system components are modeled as functional processes, whose set of behaviors is defined by a mapping from a set

A prerequisite for the interaction between distinct components is the existence of a *connection* between an output port of one component, called the *sender*, and an input port of another component, called the *receiver*. A connection can be modeled as a process whose function is the identity between input and output signals. With respect to the interaction between sender and receiver, a connection imposes the equality of the input signal of the receiver with the output signal of the sender (Figure 2).

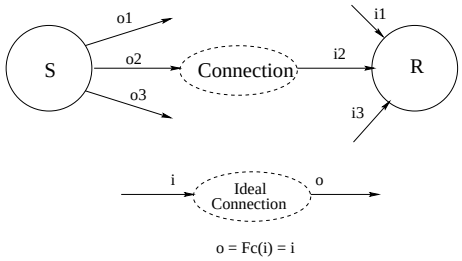


Fig. 2. Connection Process

Consider two components, the *sender* modeled by process S ($F_s : I_s \Rightarrow O_s$) and the *receiver* by process R ($F_r : I_r \Rightarrow O_r$). Connecting S and R as shown in Figure 3 implies that only a signal that is an output of S may be an input of R. As a result, the input space of R is *restricted* to the intersection $O_s \cap I_r$ of its domain I_r with the output space of S (Figure 3). When the domain of R includes a signal $i_r \notin O_s$, the connection results in a restriction of the set of possible behaviors of R that can be optimized by removing the behaviors that correspond to the input signals $i_r \in \overline{O_s} \cap I_r$.

If the set $O_s \cap \overline{I_r}$ is not empty, we say that the behaviors of S and R are not adapted, since the behavior of R is not defined for inputs $o_s^* \in O_s \cap \overline{I_r}$. This mismatch [4] can be solved in one of the following ways:

1. R discards inputs o_s^* and treats them as errors,
2. outputs o_s^* of S and the behaviors originating them are removed from S,
3. signals o_s^* are mapped into signals that can be accepted by R.

In 1) the set of behaviors of R is extended to R' to include the error handling of the undesired input signals that may be received. In 2) the behavior of the sender is optimized and restricted to S' to exclude the production of output signals incompatible with R. In 3) an interface (represented in Figure 4 as a process with function F_{beh_int}) is used to map the signal emitted by S into a

of input signals I to a set of output signals O, $F : I \Rightarrow O$. Unless a definition is explicitly given, terms like process, signal, behavior are used below as in [3].

[4] An example is when the sender is an analog system and the receiver is digital: an A/D converter is needed to allow the receiver to understand the messages of the sender.

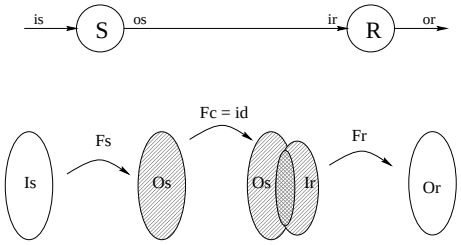


Fig. 3. Behavior mismatch

signal that belongs to the domain of R. Such an interface [9] can be usually split into two processes ($F_{beh\_int} = F_{beh\_sint} \circ F_{beh\_rint}$)), that encapsulate S and R ($F_{s'} = F_s \circ F_{beh\_sint}$ and $F_{r'} = F_{beh\_rint} \circ F_r$) and permit communication between the modified behaviors S' and R' over a connection. We call this type of interface *Behavior Adapter (BA)* (Figure 4).

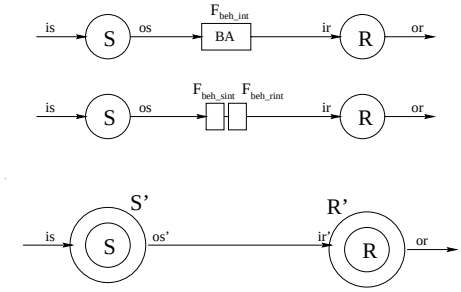


Fig. 4. Behavior Adapter

Connections are implemented using physical channels [5] , whose function ($F_c : I_c \Rightarrow O_c$) in general differs from the identity, e.g. due to noise or interference. As a result, even if the behaviors S' and R' were perfectly adapted, the received signal $F_c(i_c) = F_c(o_s)$ might be out of the domain of R' or trigger an incorrect behavior of R. Therefore, for a safe and correct interaction among system components it is key to select a channel whose behavior approximates that of the ideal connection.

Quality of Service requirements [6] partition the set of behaviors F into two classes, the class of those that satisfy the quality requirements and the class of those that do not satisfy them. Let us introduce a relation $\sim \subseteq F \times F$, such that

$$\sim = (f, f') | f \in F, f' \in F, both f and f' satisfy the quality requirements$$

[5] A connection establishes a relation between signals. A channel is a set of physical objects that implement a connection.

[6] Quality of Service requirements include delay, throughput, maximum number of errors...

Given a connection and a set of requirements on the quality of the received signal, the set of the behaviors $S \circ F_c \circ R$, where F_c is the channel function is partitioned into two classes: the class of *valid* channels and the class of the *invalid* ones. The former includes all the channels that guarantee a quality of the received signal that satisfies the requirements. Since the ideal connection by definition satisfies the quality requirements, the set of valid channels can be defined as:

$$ValidChannels = F_c : F_s \circ F_c \circ F_r \sim F_s \circ id \circ F_r = F_s \circ F_r$$

The first step in designing a valid channel is to select a physical channel whose function is F_c. If the channel is invalid due to its physical limitations, it is necessary to introduce an interface between the behaviors and the channel that matches the undesired effects. We call this type of interface *Channel Adapter (CA)* [7]. be included in the behaviors of sender and receiver... A channel adapter interface is usually symmetric to the channel and is defined by two functions, (F_{cs_int}) implementing the sender-channel and (F_{cr_int}) the channel-receiver interfaces (Figure 5). If $F_{cs\_int} \circ F_c \circ F_{cr\_int} \sim id$ the interface successfully adapts the channel F_c, otherwise it is necessary to iterate the adaptation process and look for two other functions $F_{cs'\_int}$ and $F_{cr'\_int}$ such that $F_{cs\_int} \circ F_{cs'\_int} \circ F_c \circ F_{cr'\_int} \circ F_{cr\_int} \sim id$ [8]. Note also that if channel adapters introduce some mismatch between the range of $F_{cs\_int} \circ F_c$ and the domain of F_{cs_int} a behavior adapter is needed.

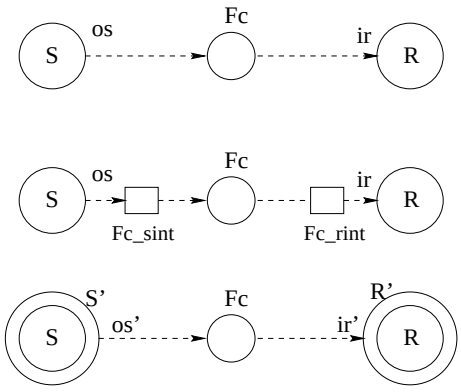


Fig. 5. Channel Adapter

Following the above discussion, we define communication as the mechanism

[7] Example of Channel Adapters functions are: error correction, flow control, medium access control.

[8] The channel adapter interface often consist of a stack of several layers (e.g. ISO-OSI Reference Model), each processing the signal at a different level of abstraction.

that allows the interaction between at least two distinct behaviors. A protocol is the set of interfaces that implement the communication.

Figure 6 describes the flow we propose for designing protocols. Given two components S and R, they are first connected and their behaviors are compared. If there is a need for a behavior adapter BA, this is introduced. The next step is the selction of channel CH. If there is no valid channel available, a channel adapter composed of the two interfaces CRA (channel-receiver adapter) and SCA (sender-channel adapter) is introduced to overcome the limitations of the invalid channel selected. If the behavior of the sender composed with the channel and the behavior of the receiver are not adapted, another behavior adapter is needed. This process is iterated until the behaviors do not need to be adapted and a valid channel is defined.

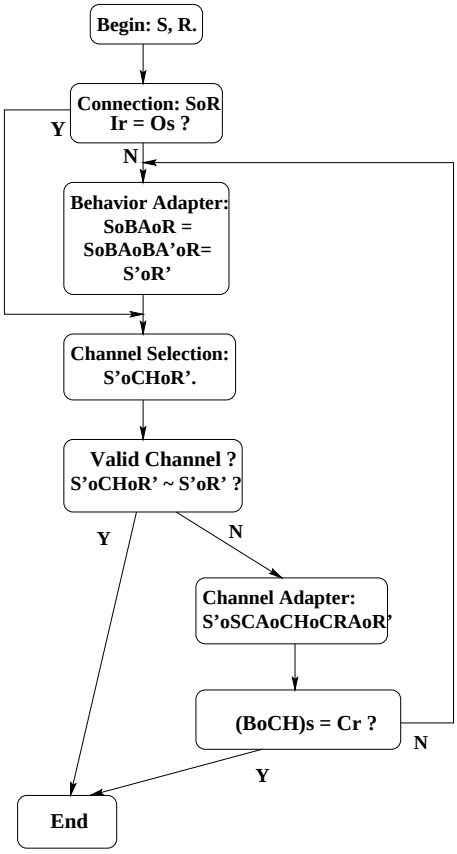


Fig. 6. Design Flow

3 Abstract Codesign Finite State Machines

Abstract Codesign Finite State Machines (ACFSMs) is a formal model that allows one to represent embedded systems specifications, involving both control and dataflow aspects, at a high level of abstraction. It consists of a network of FSMs that communicate asynchronously by means of events (that at the abstract level only denote partial ordering, not time) over lossless signals with FIFO semantics. ACFSMs are a Globally Asynchronous Locally Synchronous (GALS) model:

- the local behavior is synchronous (*from its own perspective*, like the "atomic firing" of Dataflow actors), because each ACFSM executes a transition by producing an output reaction based on a snap-shot set of inputs in zero time,
- the global behavior is asynchronous (*as seen by the rest of the system*) since each ACFSM detects inputs, executes a transition, and emits outputs in an unbounded but finite amount of time.

The asynchronous communication among ACFSMs over an unbounded FIFO channel:

- supports a (possibly very non-deterministic) specification where the execution delay of each ACFSM is unknown a priori and, therefore, is not biased towards a specific hardware/software implementation,
- decouples the behavior of each ACFSM from the communication with other ACFSMs. The communication can then be designed by refinement independently from the functional specification of the ACFSMs, as we show in Section 4.

3.1 Single ACFSM behavior

A single ACFSM describes a finite state control operating on a data flow. It is an extended FSM, where the extensions add support for data handling and asynchronous communication. An ACFSM transition can be executed when a pre-condition on the number of present input events and a boolean expression over the values of those input events is satisfied. During a transition execution, an ACFSM first *atomically* detects and consumes some of the input events, then performs a computation by emitting output events with the value determined by expressions over detected input events. A key feature of ACFSMs is that transitions in general consume multiple events from the same input signal, and produce multiple events to the same output signal (*multi-rate transitions*). We formally define an ACFSM as follows:

Definition 1. An ACFSM is a triple $A = (I, O, T)$:

- $I = \{I_1, I_2, \ldots I_N\}$ is a finite set of inputs. Let i_i^j indicate the event that at a certain instant occupies the j-th position in the FIFO at input I_i.
- $O = \{O_1, O_2, \ldots O_M\}$ is a finite set of outputs. Let o_i^j indicate the j-th event emitted by a transition on output O_i.

– $T \subseteq \{(IR, IB, CR, OR, OB)\}$ is the transition relation, where:
  - IR is the input enabling rate,
    $IR = \{(I_1, ir_1), (I_2, ir_2), \ldots, (I_N, ir_N) \mid$
    $1 \leq n \leq N, I_n \in I, ir_n \in \mathbb{N}\}$
    i.e., ir_n is the number of input events from each input I_n that are required to trigger the transition.
  - IB is a boolean-valued expression over the values of the events
    $\{i_i^j\}, 1 \leq i \leq N, 1 \leq j \leq ir_i$
    that enable the transition.
  - CR is the input consumption rate,
    $CR = \{(I_1, cr_1), (I_2, cr_2), \ldots, (I_N, cr_N) \mid$
    $1 \leq n \leq N, I_n \in I, cr_n \in \mathbb{N},$
    $cr_n \leq ir_n \vee cr_n = I_n^{ALL}\}$
    i.e., cr_n is the number of input events consumed from each input[9].
  - OR is the output production rate,
    $OR = \{(O_1, or_1), (O_2, or_2), \ldots, (O_M, or_M) \mid$
    $1 \leq m \leq M, O_m \in O, or_m \in \mathbb{N}\}$
    i.e., or_n is the number of output events produced on each output O_n during a transition execution.
  - OB is a set of vectors of expressions that determines the values of the output events, one vector per output with $or_n > 0$ and one element per emitted event.
    $\{o_i^j\}, 1 \leq i \leq N, 1 \leq j \leq or_i$

Note that signals that are at the same time input and output, and for which a single event is produced and consumed at each transition act as *implicit* state variables of the ACFSM.

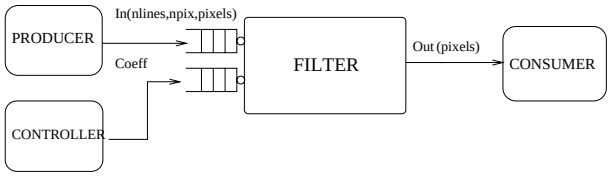
If several transitions can be executed in a given configuration (events and values) of the input signals, the ACFSM is non-deterministic and can execute any one of the matching transitions.

ACFSMs differ from Dataflow networks in that there is no blocking read requirement, i.e. ACFSMs transitions, unlike firings in DF networks, can be conditioned to the *absence* of an event over a signal. Hence, ACFSMs are also not continuous in Kahn's sense [4] (the arrival of two events in different orders may change the behavior). Another difference from DF models is that ACFSMs can "flush" an input signal, in order to model exception handling and reaction to disruptive events (e.g., errors and re-initializations). A DF actor cannot do so, since when the input signal has been emptied, the actor is blocked waiting on the signal, rather than proceeding to execute the recovery action.

[9] The number of events that is consumed should be not greater than the number of events that enabled the transition. It is also possible to specify, by saying that $cr_n = I_n^{ALL}$, that a transition *resets* a given input, i.e., it consumes all the events in the corresponding signal (that must be at least as many as those enabling the transition).

A difference from most FSM-based models (e.g., SDL) is the possibility to use multi-rate transitions to represent production and consumption of events over the same signal at different rates (e.g. images produced line-by-line and consumed pixel-by-pixel).

As an example, consider the filter shown in Figure 7. It filters a sequence of frames, by multiplying all the pixels of a frame by a coefficient. At the beginning of each iteration it receives the number of lines per frame and pixels per line from the input signal *in* and the initial filtering coefficient (used to multiply the pixels) from the input signal *coef*. Then, it receives a frame, i.e. a sequence of lines of pixels, from *in*, possibly interleaved with new coefficient values (if it must be updated) from *coef*. The filtered frame is produced on the output signal *out*. The primitives *read(in,n)* and *write(out,n)* consume and produce n events from signal *in* and *out*, respectively, and *present(coef,n)* returns true if n events are available on signal *coef*.




```
module filter;
input byte in, coef;
output byte out;
int nlines, npix, line, pix;
byte k;
int buffer[];
forever {
  (nlines, npix) = read (in, 2);
  k = read (coef, 1);
  for (line = 1; line <= nlines; line++) {
    if (present(coef, 1)) k = read (coef, 1);
    buffer = read (in, npix);
    for (pix = 1; pix <= npix; pix++)
      buffer[pix] = buffer[pix] * k;
    write (out, buffer, npix);
  }
}
```

Fig. 7. Filter example.

Let *prod(v,s,n)* denote the multiplication of a vector v of n values by a scalar s, and "$\Leftarrow$" a shorthand to represent writing to state feedback signals, which are also omitted for simplicity from IR and OR, where they always have rate 1. The filter can be modeled by an ACFSM as follows:

- $IR = CR = \{(\text{in}, 2), (\text{coef}, 1)\}$, $IB = (\text{state} = 1)$,
  $OR = \{\}$, $OB = \{(\text{nlines}, \text{npix}) \Leftarrow \text{read}(\text{in}, 2),$
  $\text{line} \Leftarrow 1, k \Leftarrow \text{read}(\text{coef}, 1), \text{state} \Leftarrow 1\}$
- $IR = CR = \{(\text{in}, \text{npix})\}$,
  $IB = (\text{state} = 2 \wedge \text{line} < \text{nlines})$,
  $OR = \{(\text{out}, \text{npix})\}$,
  $OB = \{\text{write}(\text{out}, \text{prod}(\text{read}(\text{in}, \text{npix}), k, \text{npix}), \text{npix}),$
  $\text{state} \Leftarrow 2, \text{line} \Leftarrow \text{line} + 1\}$
- $IR = CR = \{(\text{in}, \text{npix})\}$,
  $IB = (\text{state} = 2 \wedge \text{line} = \text{nlines})$, $OR = \{(\text{out}, \text{npix})\}$,
  $OB = \{\text{write}(\text{out}, \text{prod}(\text{read}(\text{in}, \text{npix}), k, \text{npix}), \text{npix}),$
  $\text{state} \Leftarrow 1\}$
- $IR = CR = \{(\text{coef}, 1)\}$, $IB = (\text{state} = 2)$,
  $OR = \{\}$, $OB = \{k \Leftarrow \text{read}(\text{coef}, 1), \text{state} \Leftarrow 2\}$

3.2 Network of ACFSMs

An ACFSMs *network* is a set of ACFSMs and signals. The behavior of the network depends on both the individual behavior of each ACFSM, and that of the global system. In the mathematical model, the system is composed of ACFSMs and a scheduling mechanism coordinating them. The scheduler operates by continually deciding which ACFSMs can be run, and calling them to be executed. Each ACFSM is either *idle* (waiting for input events), or *ready* (waiting to be run by the scheduler), or *executing* a single transition from its transition relation.

The topology of the network, as for dataflow networks, simply specifies a partial order on the execution of the ACFSMs. Initially the time required by an ACFSM to perform a state transition is not specified, hence each ACFSM captures *all its possible hardware* (with or without resource sharing constraints) *and software* (generally with CPU sharing constraints) *implementations*. The ACFSM model is fully implementation-independent but, as a consequence, it is highly non-deterministic.

The non-determinism present in ACFSMs is resolved only after an *architectural mapping* (implementation) is chosen. An architectural mapping in this context means:

- a set of architectural and communication resources (CPUs, ASICs, busses, ...),
- a mapping from ACFSMs to architectural resources and from signals to communication resources,
- a scheduling policy for shared resources.

Once an architectural mapping is selected for a network of ACFSMs, the computation delay for each ACFSM transition can be estimated with a variety of methods, depending on how the trade-off between speed and accuracy is solved [1].

Annotating the ACFSM transitions with such delay estimates defines a global order of execution of the ACFSM network that has now been "refined" into a Discrete Event semantics. At this level the designer can verify, by using a Discrete Event simulator [1], if the mapped ACFSM network satisfies not only functional but also performance and cost requirements.

4 Refining ACFSMs: EFCSMs

Abstract CFSMs communicate asynchronously by means of events, transmitting and receiving data over unbounded communication channels with FIFO semantics. This abstract specification defines for each channel only a partial order between the emission and the reception of events and therefore must be refined to be implemented with a finite amount of resources. To implement an abstract communication it is necessary to design a protocol that satisfies the functional and performance requirements of the communication, and can be implemented at a minimum cost in terms of power, area, delay and throughput.

The amount of data that should be *correctly* received is one requirement. A communication mechanism is called *lossless*, if no data (in the form of events) is lost over the communication channel during any system execution, and *lossy* otherwise. The specification requirements dictate if the communication must be lossless or lossy and, if lossy, how much loss is acceptable. Data can be lost either because of the poor quality of the physical channel, e.g. due to noise or interference, or because of the limited amount of resources in the implementation, e.g. the receiver is slow and does not have enough memory to store the incoming data. In the first case the problem is usually overcome by the definition of a robust protocol that (probabilistically) guarantees correctness of received data by means of re-transmissions or coding techniques for error correction. In the second case the solution, as discussed below, is to use a sufficient amount of resources or an appropriate protocol to meet the requirements. The throughput and the latency in the arrival of the data to the destination are key requirements especially in the design of protocols for real-time applications, e.g. real-time video or audio, that require that incoming video frames and audio samples are received and processed at regular intervals.

Communication protocols in our approach are refined towards implementation through several levels of abstraction, by applying a sequence of refinement steps, such that each step preserves the original behavior and constraints are propagated in a *top-down* fashion. At the same time the use of architectural units, e.g. the physical channel, and functional libraries, e.g. communication primitives, captures also the *bottom-up* aspect of the design process. Therefore, we can say that the overall methodology we propose is really a mix of top-down and bottom-up.

At the Abstract CFSMs level, the specification includes the topology of the network and the functional behavior of each module, while the protocols that implement the communication requirements are still undefined. To optimally design a protocol for each communication channel it is necessary to use a model, or

set of models, that allow one to capture different algorithmic solutions and evaluate their implementation costs. For this reason we introduce the Extended Codesign Finite State Machines (ECFSMs) model that "implements" the ACFSMs model, defined in the previous Section. ECFSMs are obtained from the ACFSMs simply by refining infinite-size queues to implementable finite-size queues. The event-based communication semantics, as well as the rules for the execution of ECFSMs transitions, are the same as in the ACFSMs model.

ECFSMs have input queues of finite size, and write operations can be either blocking or non-blocking (on a channel-by-channel basis). In the latter case, every time a queue is full and new data arrives over the channel, the data previously stored is lost (overwritten)[10]. To avoid this scenario it is often sufficient to use a longer queue. Sometimes, instead, e.g. when the average production rate of the sender is greater than the average consumption rate of the receiver, there exists no finite-queue solution guaranteeing no loss. The problem of checking if there exists a lossless implementation with bounded queues, that has been solved using static or quasi-static scheduling algorithms for Synchronous Data Flow networks [5] and Free-Choice Petri Nets [6], is undecidable in general for the ACFSMs model. Therefore, to implement lossless communication in a given ECFSM network, it is necessary to define an additional mechanism that, when a queue is full, blocks the sender until the queue has again enough space for new incoming data. This can be achieved by means of either a handshake protocol, consisting of explicit events carrying the sender request for an emission and the receiver acknowledgment that new data can be accommodated in the queue, or scheduling constraints that ensure that the sender is scheduled for execution only when the queue at the receiver has enough space. Depending on the requirements on data losses, ACFSM queues are refined into either *lossy* ones, that are eventually overwritten when they are full, or *lossless* ones, where overwriting is prevented by a blocking write protocol or scheduling constraints.

The actual size of the queues should be determined by evaluating the cost of different implementations that satisfy the communication requirements. For example, large-size queues mean high throughput due to the higher level of pipelining that can be achieved by sender and receiver, but are also expensive in terms of area. Small-size queues reduce the area, but decrease the throughput, as the sender is blocked more often, and increase power consumption (more frequent request and acknowledgment messages).

5 Communication over FIFO Channels

The ACFSMs and ECFSMs models follow in the computation part the same Extended Finite State Machines semantics described in Section 3. Their semantics of the communication is instead different: both models use FIFO channels, but while ACFSMs communicate over channels with unbounded queues, ECFSMs communicate over channels with queues of finite size.

[10] Traditional CFSMs [1] are a special case of ECFSMs, where the queues have length one.

A FIFO channel can be seen as an adapter between behaviors that operate at a different rate. If the sender produces outputs at a faster rate than the receiver, a FIFO allows to handle their rate difference and prevent from losses.

An unbounded queue is an ideal adapter, since it allows to accumulate an infinite number of tokens and therefore match any rate difference. However, as discussed above, ACFSMs communication is to be implemented using ECFSMs whose channels have finite queues. Unfortunately, bounded FIFO channels do not prevent from overflow, i.e. losses, when the FIFO is full. For this reason, a bounded FIFO channel is in many cases not be a valid channel, especially for systems where the rate difference between sender and receiver is large. In this case it is necessary to introduce a channel adapter interface that here may take the form either of a scheduling policy or an explicit Request/Acknowledgment protocol that blocks the sender when the FIFO is full. In particular the Req/Ack protocol restricts the behavior of S to S' excluding all the behaviors, considered illegal, where the number of consecutive output events exceeds the capacity of the queue 8.

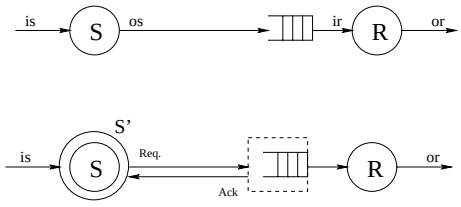


Fig. 8. Fifo Channel

6 Example: a wireless protocol

Intercom [7] is a single-cell wireless network supporting voice communication among a number of mobile terminals. The network operation is coordinated by a unit, called base station, that handles user service requests (e.g. request to establish a connection), and solves the shared wireless medium access problem using a TDMA (Time Division Multiple Access) policy and assigning the slots to communicating users.

Figure 9 describes the Intercom protocol stack, that is composed of the following layers. The *User Interface Layer* interacts with the users and forwards to lower layers user service requests and (logarithmically quantized) voice samples. The *Transport Layer* takes care of message retransmission until acknowledgment reception. The *MAC Layer* implements the TDMA scheme, keeping within internal tables the information on which action (transmit, receive or standby) the terminal should take at each slot. The *Error Control Layer* applies the Cyclic Redundancy Check algorithm to the incoming and outgoing streams of data and,

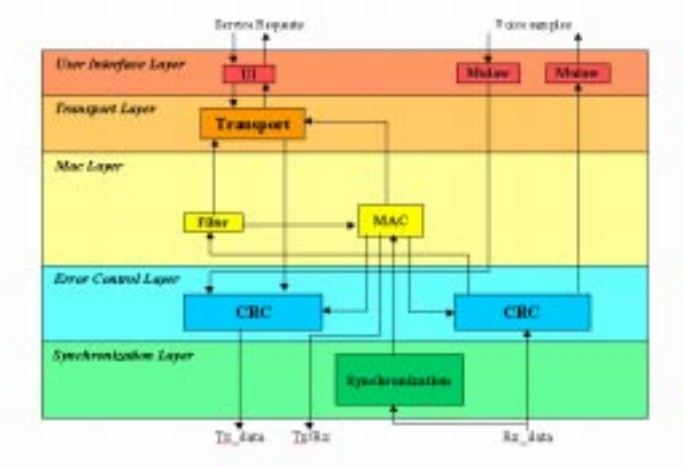


Fig. 9. Intercom protocol stack

if detects an error, discards the incorrect packet. The *Synchronization Layer* extracts from the received bit stream the frame and slot synchronization patterns, and notifies the MAC of the beginning of a new slot. The *Physical Layer* includes computation-intensive bit-level data processing functions, e.g. modulation, timing recovery. The Intercom protocol specification includes both data processing and control functions. It includes computation intensive functions (e.g. error control, logarithmic quantization) that are applied to the flows of (a) data samples and (b) service request/acknowledgment messages both in transmission and reception . Control functions include time-dependent and data-dependent control. The first type occurs at the MAC layer, where the processing and the transmission (or reception) of data flows is enabled using a time-based (TDMA) mechanism. Error control is instead a data-dependent control function since it enables some actions (e.g. discarding a packet) depending on the result of data computation.

To model the Intercom protocol specification we have initially used ACFSMs. Then, starting from the communication requirements and taking into account the assumptions on the behavior of the environment, e.g. on the rates and patterns of the input events, we have refined the ACFSMs into implementable ECFSMs and selected a communication protocol for each channel (currently this step is manual, but we are exploring ways of automating it). Since high-quality voice communication requires that no data is lost within the protocol stack due to buffer overflow, we have refined ACFSMs into *lossless* ECFSMs. Additional timing requirement are imposed by the TDMA policy: data transmission can occur only within TDMA slots. Environmental assumptions concern the occurrence of the following input signals: *Voice samples* is periodic at 64 kbps, *Rx_data* is a periodic bit stream at 1.6 Mbps. Let us consider in detail the behavior of a protocol stack fragment. Incoming voice samples are first processed by the Mulaw

and sent to the CRC module that at each transmission slot is enabled by the MAC. The CRC input FIFO shapes the data stream to make it fit the TDMA slot allocation pattern and, since the read and write operations from this queue occur at regular times, in this case, the FIFO size can be simply computed from input/output rates and slotsize, and chosen to be 500 bytes. A smaller size would require to block the Mulaw module and introduce another queue at the input of Mulaw. A greater size does not give any advantage since 500 bytes is the maximum amount of data arriving during a frame.

If we used the classical CFSMs model instead of ACFSMs, we would be forced to represent each FIFO as a CFSM. A CFSM of type FIFO calls, upon occurrence of external events, internal read and write functions that access an "internal" memory in FIFO manner. This implies that the CFSM at the receiver end of the channel (in this case CRC) has to explicitly make a request (in the form of an event) for new data as soon as it can process it. This results in an unnecessary overhead that is not present in the ACFSM model where FIFOs are part of the model and can be directly accessed by simple communication primitives.

Finally, we also compared the design of the Intercom using our ACFSM-based methodology with a previous design [7] of the same specification done following an SDL-based approach. The overall design process turned out to be easier and more effective using ACFSMs due to their capability of modeling control and dataflow components independently from the final implementation. The previous approach, instead, due to SDL's purely asynchronous nature, required to partition into hardware and software components from the beginning and model the hardware components directly in VHDL. This greatly reduced the design space and prevented from some system optimizations.

7 Conclusions

In this paper, we presented a new way of formalizing communication among processes. We then introduced a new Model Of Computation called Abstract Codesign Finite State Machines, in which a finite state control coordinates multi-rate transitions and data-intensive computations. We described how a specification using ACFSMs can be refined, by queue sizing and static scheduling, until communication becomes implementable. We used a real-life example, a wireless communication protocol, to illustrate how the abstract communication can be refined to a concrete, implementable one, and we discussed the main trade-offs involved.

8 Acknowledgements

We gratefully acknowledge the discussions about Models of Computation with Ken McMillan and Roberto Passerone and about communication protocols with

Jan Rabaey. This research is sponsored in part by the Giga-scale Silicon Research Center (GSRC) and the Consiglio Nazionale delle Ricerche, Progetto MADESSII.

References

1. F. Balarin and al. *Hardware-Software Co-Design of Embedded Systems: The POLIS Approach.* Kluwer Academic Publishers, 1997.
2. J. Buck, S. Ha, E. Lee, and D. Messerschmitt. Ptolemy: a framework for simulating and prototyping heterogeneous systems. *International Journal of Computer Simulation*, Special issue on Simulation Software Development, Jan. 1990.
3. E.A.Lee and A.L.Sangiovanni-Vincentelli. A framework for comparing models of computation. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 17(12):1217–29, December 1998.
4. G. Kahn. The semantics of a simple language for parallel programming. In *Proceedings IFIP Congress*, Aug. 1974.
5. E. A. Lee and D. G. Messerschmitt. Synchronous data flow. *IEEE Proceedings*, Sept. 1987.
6. M.Sgroi, L.Lavagno, Y.Watanabe, and A.Sangiovanni-Vincentelli. Synthesis of embedded software using free-choice petri nets. In *Proceedings of the Design Automation Conference*, June 1999.
7. J. Rabaey and et al. Intercom project. `http://bwrc.eecs.berkeley.edu/Research/Intercom_group/`.
8. J. Rowson and A. Sangiovanni-Vincentelli. Interface-based design. In *Proceedings of the Design Automation Conference*, pages 178–183, 1997.
9. R.Passerone, J.Rowson, and A.Sangiovanni-Vincentelli. Automatic synthesis of interfaces between incompatible protocols. In *Proceedings of the Design Automation Conference*, pages 8–13, June 1998.