Timing-Based Communication Refinement for CFSMs

Heloise Hse and Irene Po
{hwawen, ipo}@eecs.berkeley.edu

EE249 Term Project Report
December 10, 1998

Department of Electrical Engineering and Computer Sciences
University of California at U.C. Berkeley

Abstract
In this paper, we present an approach to refine the communication mechanism of CFSMs. Currently, CFSMs communicate via one-place buffer using non-blocking write semantics. If sending events arrive faster than the receiving module can consume, some events may be lost. This is clearly undesirable especially when the events are critical. Using a rendezvous-based mechanism for communication as our starting point, we transform rendezvous points into bounded buffers whenever possible. An algorithm for sizing buffers is presented taking into consideration of the possibility of CPU contention when several processes are sharing a CPU.

1. Introduction
In Polis, a CFSM network currently uses a lossy communication mechanism. Communication takes place via one-place input buffers and uses non-blocking write semantics. The problem with this approach is that if the sending CFSM generates events faster than the receiving CFSM is able to consume, buffers will be overwritten and events will be lost. This is clearly undesirable, especially when the events are critical.

In this project, we attempt to improve this aspect of the CFSM network model by making the communication mechanism lossless. We start with a rendezvous-based specification that is much safer, but is extremely inefficient, and gradually relax this by replacing as many rendezvous points as possible with bounded buffers. Our goal is to find the smallest buffer sizes while maintaining lossless communication. In addition, this transformation should preserve input-output
behavior under timing assumptions on the CFSM network. In the case where the size of the buffer is found to be unbounded, the rendezvous point will be kept, in order to guarantee that events are not lost.

2. Project Evolution

We started by looking into the areas of rate analysis [1] and queuing theory [2]. Our initial idea was that if we could determine the process execution rates, we could then use that information to size buffers. An approach we considered was the following:

Assume that the environmental input rate, \( i_e \), is known. We would size the buffers one at a time. Referring to Fig. 1, the block before the sender of the buffer being sized is an arbitrary sequence of CFSM modules. We considered using a tool, RATAN [3] to perform rate analysis to compute bounds on \( i_e \) and process execution rates of the sending and receiving CFSM modules. Given the upper and lower bounds of the execution rates, we can then find the minimum buffer size by setting up the worst case scenario in which the sender emits events at its fastest rate, while the receiver consumes events at its slowest rate.

However, after studying rate analysis in detail, we realized that rate analysis could not completely solve our problem. Rate analysis only computes the relative frequency that the CFSMs will be executed, in terms of the average upper and lower bounds of the average process execution rate. This information can be used to determine whether it is feasible to transform a rendezvous point into a bounded buffer. For a particular communication channel, if the lower bound of the sender execution rate is greater than the upper bound of that of the receiver, the receiver cannot catch up with the sender on average, therefore the buffer size will be unbounded. In this case, the rendezvous point would be retained. Moreover, the rates obtained from rate analysis may not match all the time due to internal (input) and external (scheduling) burstiness, since the rates are only the bounds on the average rates.

Queuing theory does not help us too much on this problem either. It only evaluates probabilistic measures, such as average throughput, or the blocking probability (probability that events are lost) of a queue (buffer). Most importantly, the queue size is needed as an input. Since our goal is actually to achieve a perfectly lossless communication, this probabilistic evaluation approach is not applicable to our problem. Moreover, this technique is only useful for understanding whether a given buffer size is adequate. In our case, though, we are interested in finding a minimum size of the buffer. Running the algorithm many times and reducing the estimated buffer size every time until we find a satisfactory answer is not a feasible solution.
We have also looked into other ways of solving this buffer sizing problem including Petri Nets. Thus, much of the time in this project was spent brainstorming and discussing ideas with people in an attempt to formulate a reasonable approach.

3. Buffer Sizing Algorithm

Our final decision was to leverage the approach proposed in an existing paper [4] to explore the issues of buffer sizing in a CFSM network. The basic idea in our proposed approach is to translate a CFSM network into communicating FSMs, size the buffers for the FSMs, and use the new buffer size information to adjust the corresponding buffer sizes in the original CFSM network.

3.1 CFSM to FSM translation

Polis provides a tool for translating a CFSM module into an FSM. Each CFSM module takes some time to execute, because in each state, some computations may be performed. For a CFSM module implemented in hardware, the state transition times are all equal to the speed of the processor and hence, every transition takes the same amount of time. However, if the CFSM is implemented in software, the transition times are the execution time of the states which may vary. In FSMs, there is no notion of time associated with transitions. In order to correctly model the timing behavior of the CFSMs during the translation, we need to normalize the transition time by inserting intermediate states into the FSMs using the shortest state transition time as the normalization factor. An example of the normalization process is shown in Fig. 3 and 4.
3.2 Algorithm overview
After translating each CFSM into an FSM, we apply the buffer sizing algorithm based on [4]. Stated in a nutshell, the procedure consists of modeling each buffer as an FSM, computing the set of reachable states of the network, deriving the set of reachable states for each communication buffer, and finally, determining the minimum buffer sizes from the buffer's reachable states.

3.3 Global "Clock"
In order to traverse the state space, we need to know the interaction among processes. Therefore a global "clock" is introduced to serve as a reference for different modules. The frequency of the "clock" is defined as the least common multiple of all frequencies of FSMs in the network. An FSM can make a transition only when it is active based on the global “clock”.

3.4 Buffer Modeling
Each buffer is modeled as a counter using an FSM. An estimated size for each buffer is used as the initial value. When a process writes to a buffer, the buffer counter will be incremented. On the other hand, if a process reads from a buffer, the counter will be decremented. Each buffer FSM has an output function which indicates the overflow condition. When an overflow occurs, the buffer counter will be expanded dynamically.

3.5 Reachable Buffer States
From the transition relations of the FSMs of both the processes and the buffers, we can build the transition relation of the product machine over all the variables in the entire network. We use this to perform an implicit state enumeration in order to obtain the set of reachable states. As mentioned before, during the state space traversal, if a buffer overflows, its counter is dynamically expanded by adding an additional state variable to its FSM.

3.6 Minimum Buffer Size
After the state space traversal, we first derive the set of all reachable buffer states by existentially quantifying over all non-buffer state variables. Then for each buffer, we existentially quantifying over other buffer state variables. Lastly, we determine the minimum buffer size for each buffer from its largest counter value.

4. Mixed HW/SW Implementation with CPU Contention

The algorithm outlined in the previous section does not consider the case when several processes are implemented in software and share a CPU. In this case, there may be contention due to CPU resource sharing. Consider the scenario illustrated in the following figure:

```
|   |   |   |   |   |   |
Sender
|   |   |   |   |   |   |
Receiver
|   |   |   |   |   |
```

Fig. 5

Assume that the sender is implemented in hardware or has a dedicated CPU, and it emits events periodically. If, on the other hand, the receiver is sharing a CPU with some other processes, it may not be able to execute right away when it is ready, because other processes may be occupying the CPU at that time. In this case, the receiver has to wait in the queue until the CPU
becomes available. Thus, it may not be able to catch up with the event-emitting rate of the sender, as depicted in the diagram above. Therefore, we need to increase the corresponding buffer sizes to accommodate this case.

4.1 Modified Buffer Sizing Algorithm

We first simplify the situation by making a few assumptions. Assume that the CFSM network is a DAG and there are no feedback signals so that the rate of the receiving module cannot affect the rate of the sending module. Moreover, the software scheduler is assumed to be FIFO, non-preemptive and non-priority.

To obtain minimum buffer sizes while maintaining a lossless communication, we need to set up the worst case scenario, i.e. the sender of a buffer emits events at its fastest rate while the corresponding receiver consumes events at its slowest rate. However, this situation cannot be achieved in a global sense, since each sender module may also be the receiver with respect to another buffer. Therefore, we can only size the buffers one at a time. We assume that all the modules before and including the sender are sending events at their fastest rates, i.e. either they are implemented in hardware, or they can have the CPU right away whenever they want to execute. The receiver, on the other hand, is assumed to be at the end of the queue every time it wants to execute.

The following algorithm is applied to each buffer whose corresponding receiver shares a CPU with other processes. For each of these buffers, a reachable states table consisting of the processes sharing the same CPU with the receiver (R) being considered is built based on the normalized FSM’s computed in the original algorithm in Section 3.1. A sample table is shown below.

<table>
<thead>
<tr>
<th>Process</th>
<th>P1</th>
<th>P2</th>
<th>P3</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reachable states</td>
<td>S1</td>
<td>S2</td>
<td>S3</td>
</tr>
</tbody>
</table>

Table 1

Since only one process can have the CPU at a time, therefore only one process can be in an intermediate state of its normalized FSM. We can therefore abstract away the set of state combinations that does not satisfy this criterion. For each state (s) in the FSM of R, we examine the set of possible states that other FSMs can be in at the same time. Since these processes are assumed to be in front of R in the queue, we can compute the worst case waiting time when R is in state s. Referring to Table 1, suppose we are examining state S1 of process P1. For all “candidate rows” in Table 1 that have S1 as an entry, we sum up the execution time of the other processes to get the waiting time of P1 before it gets to run. If there is one process which is in an intermediate state (refer to Fig. 4), its executing time is the remaining time that it has on the CPU, i.e. the time it takes to move from the present intermediate state to a “real” state, shown in Fig. 3. For all other processes, the execution times are also the time to move to the next “real” state.

After we have computed the waiting time for all “candidate rows”, we select the worst waiting time and use that to compute the worst case execution time for P1, which is the sum of original execution time and worst case waiting time. This is used for updating the transition time for state S1 in the original “FSM” of P1 (Fig. 3). For each “FSM”, the transition times need to be
normalized again by adding intermediate states to make all state transition times equal. Finally, we apply the buffer sizing algorithm in Section 3 with the updated FSM and compute the minimum size of the buffer in consideration. We repeat this for all the buffers whose corresponding receivers share a CPU with other processes.

5. Unresolved Issues

The following are the interesting challenges that we have encountered during the course of the project. However, due to the time constraint, we were not able to explore the alternatives to solve these issues.

5.1 State Explosion

The use of the implicit state enumeration technique leads to the state explosion problem. The obvious solution of abstracting away all non-buffer related variables during state traversal is not applicable to our control-dominated environment in which events depend on both time and value. Relevant information for buffer sizing will be lost this way. One approach is to abstract away variables in a smart way by examining all data-dependent operations and keep those variables that we need while abstracting away all other unrelated variables.

5.2 Environment Modeling

In the original buffer sizing algorithm in Section 3, no assumption is made about the environment. Rather, all possible input patterns are taken care of during complete state space traversal. This makes the approach very conservative and buffers may be oversized. In general, it is not easy to model the environment. Nonetheless, if we are given a description of all the properties of the environment or a set of test vectors of all possible input patterns, then we will be able to model the environment as an FSM. We can then include the environment FSM when applying the algorithm in Section 3 and obtain a more realistic set of reachable states, and hence, a more accurate minimum buffer sizes.

5.3 Buffer Unboundedness

Our original approach was to start with rendezvous-based communication and in the case where the size of the buffer was found to be unbounded, the rendezvous points would be kept. Our current algorithm, however, does not have the ability to detect that buffer would be unbounded. To achieve this end, we may need to stop the buffer sizing algorithm when the buffer size is dynamically increased to, say, twice the estimated size. Heuristics are needed to find out when to conclude that a buffer is unbounded.

5.4 “Real” Worst Case Situation

In our algorithm to size the buffers in the case of CPU contention, we assume the worst case in which all the modules before and including the sender send at their fastest rate, while the receiver receives at its slowest rate. However, this still may not result in the desired worst case scenario. In some situations, running some modules slower may cause the sender to emit events faster. Referring to Fig. 6 below, if two modules A and B, one implemented in hardware and another shares the CPU with some other processes, both emit events to the sender. Suppose that when B runs at its fastest rate, it emits events faster than A. Otherwise, A emits events faster than B. Suppose state c in the sending module moves to a next state depending on whether the event is from A or B. Suppose also that in either state d or e, an event is emitted to the receiving module of the buffer. If the transition time from state c to state e is much longer than that from state c to
state d, then making module B run at its fastest rate may not actually cause the sender to send at its fastest rate. Though we have identified this special case, we still have not found a solution to handle it.

6. Conclusions

In this project, we have explored various approaches for sizing buffers, although a lot of the techniques explored did not turn out to be useful for solving our research problem. Contrary to what we had originally planned, the focus of the project became purely theoretical and we ran out of time to implement our algorithm. We have proposed an approach to find minimum-sized buffers between CFSM modules while maintaining lossless communication and the case of CPU contention has been considered. Nevertheless, there are still a lot of unresolved issues. This is a challenging problem and it seems that we have concluded with more open questions than we started with.

Acknowledgments

We would like to thank all of the people who helped us along the way in this project. Special thanks go to our mentors, Jonathan Martin, Marco Sgroi, Luciano Lavagno and Ellen Sentovich for their time and guidance. Special thanks also to Harry Hsieh and Bassam Tabbara.

References