Modeling and Simulating Discrete Event Systems in Metropolis

Guang Yang

EECS 290N Report
December 15, 2004

University of California at Berkeley
Berkeley, CA, 94720, USA

guyang@eecs.berkeley.edu

Abstract

This paper describes modeling discrete event systems in Metropolis using one of the key concepts employed by Metropolis, orthogonalization of design aspects, which in this particular case, is the orthogonalization between capability and cost. To support the orthogonalization, quantity annotation mechanism is introduced. This paper formally analyzes simulation strategies for quantity annotation using tagged signal models. To improve simulation efficiency, taking advantages of built-in LOC constraints is discussed. Finally, a little thoughts are made for co-simulation between Metropolis MetaModel discrete event systems and external continuous time or hybrid systems.

1 Introduction

To deal with constantly increasing complexity and time-to-market pressure, a paradigm called *platform based design* [1] has been proposed. In this paradigm, design reuse is the key, but it is often hard to do so because modifying a design or a part of the description of a model usually requires extensive changes of other models in the rest of the design.

A solution to the re-use problem is to orthogonalize concerns and keep various aspects of a design separate. There are several concerns in embedded system design that could be orthogonalized. In particular,

- *Behavior versus Architecture*
- *Processes versus Coordination*
- *Capability versus Cost*

Changing any of the aspects would not affect the orthogonal aspect, which does increase the reusability of the model. However, the orthogonalization has big influence on the models of computations and on analysis tools. So far, we have observed the influence in Metropolis, which is based on *platform based design* methodology.

This paper will focus on the orthogonalization of capability and cost and its influence on models of computation and analysis tools. We pick time as the cost, because it is the most common factor in practical models. In section 2, preliminary concepts of orthogonalization and quantity annotation are explained. Section 3 discusses discrete event system modeling and simulation in Metropolis. Finally, in section 4, we try to come up with interfaces to make co-simulation easy between Metropolis discrete event systems and external continuous or hybrid systems.

2 Preliminaries

2.1 Capability vs Cost

Conceptually, within a model, typically an architecture model, there are two aspects that could be represented separately: capability, i.e. the behavior it can implement versus the cost it bears when it implements a given behavior. For example, in modeling a CPU in terms of the instructions it supports, the model would capture the behavior of each instruction such as addition or data move. On the other hand, the cost of the instruction such as the number of required clock cycles or latency in the time defined for the CPU can be specified separately. To do this, both behavior and cost models can be reused without any changes. For instance, when switching from Intel Pentium 4 to AMD Atholon microprocessor, the instruction sets are the same. Just compose it with different cost models.

Now, let's get down to more details. In Metropolis MetaModel, the Metropolis design language, systems are modeled as concurrent processes communicating with common media. Each process p will generate a sequence of m-events $< me >_p$[1]. An m-event is a three tuple, $(process, action, tag)$, consisting of which process the m-event belongs to, an action in the specification and a partial order tag[2]. The process is the one which generates this m-event. An action corresponds to a statement or a part of the statement. For example, 'a=b+c' is an action, the right hand side 'b+c' is also an action. In the specification, a block of sequential code defines a sequence of actions. It is safe to just consider the overall execution of each process to be a sequence of m-events. The partial order for the sequence of the m-events in each process is manifested by the tags in the m-events. Note that these partial order tags have nothing to do with actual timing information. M-event sequences from any two different processes can be arbitrarily interleaved with one another, unless there are communications between the processes, which add order constraints on the two m-event sequences.[2]

The capabilities of a model can be defined over m-event sequences. A continuous sequence of m-events demonstrates a particular behavior. In Metropolis, such sequences can be encapsulated, named and referred to by using either labels or functions. The cost associated with such capabilities is captured by additional tags associated with m-events. In this paper, we restrict cost to time. But in general, tags could be any quantities that of interests, such as power, priority, etc. When annotating an m-event with a time tag t, an m-event v becomes a regular event (v, t)[3]. To simplify the notations, if there is no annotation to an m-event v, we denote the event in the form of $(v, \perp)$. In the rest of the paper, we also use m-event to refer to an m-event with $\perp$ tag. When we talk about tags, we mean the actual time stamps, NOT the partial order tag inside m-events.

2.2 Quantity Managers

Due to the orthogonalization idea, in Metropolis, cost is modeled by a separate object, which is called quantity manager. Figure 1 shows the quantity annotation flow. The cost annotation flow is initiated by an m-event, who sends an annotation request of time t to the time quantity manger first. If there exist multiple processes, each one will generate an m-event and could send an annotation request to quantity manager. But note that not every m-event makes quantity annotation requests, only those designer are interested in do. Then, time quantity manager resolves all the requests. Because of the non-decreasing nature of time, time quantity manager will disable all events except the ones with the smallest time requests. Finally, the time is annotated to the m-event. Had one event not been granted, it will make an updated request again in the next round of quantity annotation. With this quantity annotation mechanism, a process ends up with a mixed sequence of m-events and regular events. Or $< e >_p = < (v, t) | t = \perp$ or $t =$some value$>_p$.

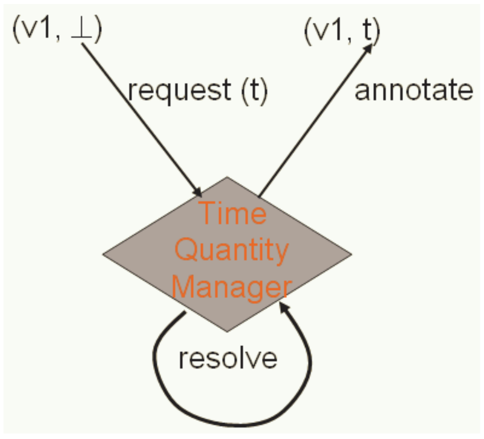

Figure 1. Quantity Annotation Flow

3 Discrete Event Systems in Metropolis

3.1 Discrete Event Systems Modeling

As described in the previous section, each process generates a sequence of m-events. Designers could add quantity annotation code to associate time tags to some m-events[4]. These time tags define a total order among the events (possibly from different processes) in the systems. However, not every m-event is required to have a time tag, which results in nondeterminism making design specification easier and leaving optimization space.

3.2 Interleaving of Events

In discrete event (DE) systems, tags define a total order among all events. This can be generalized to Metropolis MetaModel using the time quantity annotation mechanism. Suppose there are n process $p_1, p_2, ..., p_n$. Each

[1]Note that in the first Metropolis public release, all documents use the name event. But in this paper, to remove possible confusion to later discussion, we refer to Metropolis events by m-events.

[2]There are other ways to constrain m-events, such as using declarative constraints. But they are not the focus of this paper.

[3]We use the name event, because in most other DE systems, such as VHDL and Verilog, an event does have a value and a tag for actual timing information.

[4]In this sense, modeling discrete event system in Metropolis is similar to that in VHDL or Verilog. Time advance is made explicit by the designer. The ones that do not have explicit time stamps take delta time.

process generates a m-event sequence $me_1, me_2, ..., me_n$. Denote the ith ($i \in N$) m-event in me_j with $me_j(i)$, the tag of the ith m-event in me_j with $T[me_j(i)]$. Then, all $T[me_j(i)] \neq \perp$ define a total order among $me_j(i)$ themselves, as a consequence, they also restrict the possible interleaving among segments of me_j. More precisely, define

$$\lfloor me_j(i) \rfloor = \begin{cases} 0, & \text{if } i = 0. \\ i, & \text{if } T[me_j(i)] \neq \perp; \\ \lfloor me_j(i-1) \rfloor, & \text{otherwise.} \end{cases} \quad (1)$$

as the most recent annotated m-event in me_j and prior to $me_j(i)$. Similarly, define

$$\lceil me_j(i) \rceil = \begin{cases} \infty, & \text{if } i > \text{length}(me_j). \\ i, & \text{if } T[me_j(i)] \neq \perp; \\ \lceil me_j(i+1) \rceil, & \text{otherwise.} \end{cases} \quad (2)$$

as the next annotated m-event in me_j but after $me_j(i)$.

The range specified by ($\lfloor me_j(i) \rfloor$, $\lceil me_j(i) \rceil$) denotes the segment of the m-events in $me_j(i)$ that have $\perp$ tags therefore can be interleaved with other segments from other processes. In order for two segments, such as ($\lfloor me_1(i) \rfloor$, $\lceil me_1(i) \rceil$) and ($\lfloor me_2(i) \rfloor$, $\lceil me_2(i) \rceil$), to be arbitrarily interleaved, another condition must be satisfied

$$\begin{aligned} &(T[me_1(\lfloor me_1(i) \rfloor)], T[me_1(\lceil me_1(i) \rceil)]) \\ &\cap (T[me_2(\lfloor me_2(i) \rfloor)], T[me_2(\lceil me_2(i) \rceil)]) \quad \neq \quad \varnothing \end{aligned}$$

Because otherwise the two segments occur one after the other, no interleaving is possible. In fact, if the end point of a segment e.g. $T[me_j(\lfloor me_j(i) \rfloor)]$ falls in the time span of the other segment, the end point itself can join in the interleaving.

3.3 Challenges in DE Simulation and Solutions

Based on segment interleaving reasoning, simulation seems to be very straightforward, because simulator can just follow the increasing time segments and interleave them if possible. However, in reality, there is a big limitation, i.e. simulator just knows and processes the current m-events, it does not save the event history. This implies that no roll back mechanism is possible. If simulator can not decide at a certain time point, just postpone the decision until it has enough information to decide. A typical example is the unknown future end point of the segments.

Suppose at some point i, each process has a $me_j(i)$ m-event. Some of them may request time annotations from time quantity manager, others may not, therefore they have tags of $\perp$. Since we cannot predict the end point of the segments of those processes whose $T[me_j(i)] = \perp$, we postpone the decision on the time annotation requests [3]. This implies that whenever there is a m-event without time annotation requests, process that m-event, until all processes request time annotation for their m-event at the same point, where time quantity manager begins to resolve annotation requests. Based on the annotation requests, some of the m-events become events and begin to run, other m-events stay and request time annotation again. Repeat this procedure, simulation then proceeds gradually.

In above algorithm, we can foresee inefficiency. This is because at each round, simulator needs to find out whether there are m-events without time annotation requests. Also, there are other overheads in resolving communication constraints among processes[5]. It is obvious that to finish the same simulation, the less the number of rounds, the less the overhead, therefore the faster the simulation speed. In order for that, it is necessary to predict future time annotation requests. This can be partially achieved by statically analyzing the system. However, this technique is rather complex. On the other hand, system designers can usually give helpful hints. For example, in the system, there may exist test pattern generation processes, which have regular events generation rates; a process models a MPEG decoder, which has a certain latency, etc. If this kind of information can be conveyed to simulator, time quantity resolution can begin even though there exist m-events from those processes, because future time annotation requests are known. Towards this direction, built-in logic of constraints (LOC) based on quantities are introduced into Metropolis. [6] They include maxrate, minrate, maxdelta, mindelta and period. These built-in LOCs are resolved in the resolve phase of time quantity manager in figure 1. Built-in LOCs can be classified into two categories. One includes period, maxdelta and minrate. They say that an event will occur no later than a time point. Time quantity manager could do resolution as long as it does not progress beyond that point, otherwise the event would be blocked forever. The other category includes mindelta and maxrate. They set a lower bound for time only after which certain events can occur. For time quantity manager, that means it can resolve time and ignore those events until after a time point. The real value for both categories is that time quantity manager can rely on the built-in LOCs to predict future, start resolution sooner and therefore decrease the number of simulation rounds.

Take period as a concrete example. Suppose there are two processes. Process p_1 has one event with period of 10. Process p_2 has several events with random time stamps less than 10. They both generate m-events with equal rates. Before introducing LOC, it can be observed that p_1 keeps running its m-events until arrives at the event with time an-

[5]Such constraints are either imposed by the design language itself or by the designers who write declarative constraints over events

[6]This is also a methodology decision. Having declarative constraints in system specification usually simplifies the model of the system significantly, because it just states the property not the realization of the property.

notation period 10. However, p_2 usually blocks at the time annotation less than 10. Once p_1 gets to 10, p_2 can catch up and run its m-events and events with annotation less than 10. So, two processes take turn to proceed. The number of rounds is almost the sum of the number of (m-)events generated by the two processes. Now, adding the LOC period constraint, at each round, since simulator knows 10 is going to be the next time requested by p_1, even thought it is currently having an m-event, it can still resolve the time quantity for p_2. This way two processes overlap their rounds a lot, therefore speed up simulation.

3.4 Experiments

In order to verify and compare the simulation strategies described in the previous section, a producer-consumer example is built. Figure 2 shows the block diagram of the example. There are two producers p0 and p1. They keep writing integers into the FIFO M. The write operation takes 10 time units, which is ensured by saying the beginning and end time of the operation. The consumer C reads integers out of the FIFO M. The read operation takes 15 time units. They are all governed by the same time quantity manager. Both simulation results show the correct behavior of the system. The following table shows the comparison between postponed time resolution strategy and the built-in LOC strategy. It can be seen that built-in LOC helps to decrease the number of simulation rounds and thus simulation time dramatically. Note that the products of the number of simulation rounds and the average number of events running in each rounds for the two strategies are not equal. This is because the different event interleaving can affect other language-related constraints like mutual exclusion.

	Postpone Res.	Built-in LOC
# of rounds	2120	1168
Avg. # of events run in each round	1.25283	1.49914
total simulation time	0.28s	0.19s

4 Discussion: Co-simulation between Metropolis Discrete Event Systems and External CT/Hybrid Systems

Although in principle, it is possible to model continuous or hybrid systems in Metropolis with quantity managers, Metropolis does not aim to include all of them and develop its own continuous time or hybrid simulator. Instead, it defines an interchanging format such that a continuous time or hybrid model can be easily ported to other existing tools and simulate it. Then, one issue becomes important: how to interface external tools and Metropolis simulator to co-simulate external continuous time or hybrid model with

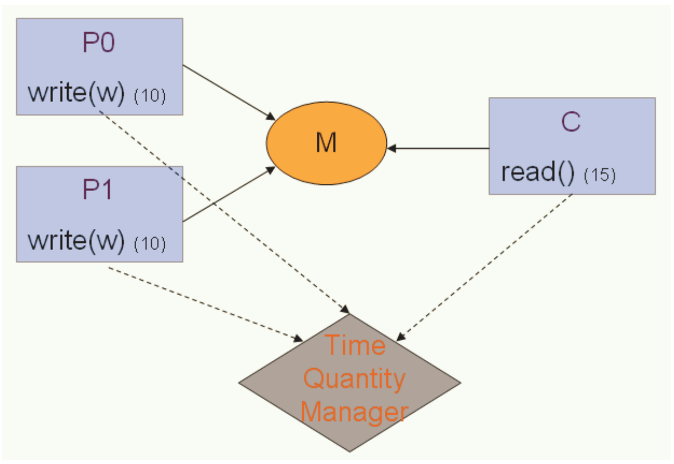

Figure 2. Producer-Consumer Example

Metropolis MetaModel? I have attempted to build an interface for this purpose. The simplest form of the interface is that

- Let Metropolis MetaModel simulator be the master
- Before time quantity manager commits the time resolution result t_2, it tells the external tool to proceed simulation time from t_1 to t_2
- External tool does the job. If it generates an event which is prior to t_2, say t_3, sends it back to Metropolis time quantity manager, and time stops at t_3.
- Metropolis MetaModel simulator processes the new event from external. During the processing, new events may be generated. Find the one with the least time stamp t_4. (In general, t_4 should be greater than t_3, but could be greater or less than t_2). Assign $t_1 = t_3$ and $t_2 = t_4$, then repeat the whole process.

5 Conclusion

This paper talks about modeling and simulating discrete event systems in Metropolis. A formal analysis framework based on tagged signal model is built to demonstrate the correctness of simulation strategy and of the simulation speed up algorithm based on built-in LOC. Experiments are made to demonstrate the effectiveness of the speed up algorithm. There are still a lot of on-going work, e.g. what happens if a process has mixed built-in LOC constraints and non-regular time annotation requests? Can continuous time systems be simulated together with discrete event systems inside Metropolis? What does fix-point iteration mean then? etc.

References

[1] K. Keutzer, A. Newton, J. Rabaey, and A. Sangiovanni-Vincentelli. System-level design: orthogonalization of concerns and platform-based design. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 19(12):1523–1543, Dec. 2000.

[2] E. Lee and A. Sangiovanni-Vincentelli. A framework for comparing models of computation. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 17:1217–1229, Dec. 1998.

[3] J. Misra. Distributed discrete-event simulation. *ACM Comput. Surv.*, 18(1):39–65, 1986.