Time for Specification of Embedded Systems

Extended Abstract

Felice Balarin
Cadence Berkeley Laboratories
Berkeley, CA 94704, USA
felice@cadence.com

1 Introduction

Embedded electronic systems can be found in almost every aspect of our lives from transportation and industrial production, to health care and entertainment. They make our cars safer and more efficient, and let us communicate with each other whenever and wherever we choose to do so, using voice, images and data. They are very powerful computing machines, but we do not think of them as computers. Rather, we think of them as components of a larger device that we use for a specific purpose. We may not even be aware of their existence, but we certainly appreciate the increased quality of life that they provide.

From the technical point of view, typical embedded system have several computing elements that can operate concurrently. Increasingly, some of these elements are microprocessors controlled by software. They may be either general purpose microprocessors, or specialized for a particular application, such as digital system processing. This trend is driven by increasing cost of designing and manufacturing application specific integrated circuits. Nevertheless, some part of functionality (perhaps small, but often critical) is usually implemented with a combination of custom digital and analog hardware.

Common to all embedded systems is that they must deliver results of their computation in a timely manner, so that the surrounding system can deliver the services that we expect. Therefore, timing behavior is a critical concern in embedded system design, and it should be taken into account throughout the design process. Unfortunately, this runs contrary to the principle of *separation of concerns* that has long served as a basis of many design methodologies. According to this principle, functionality and timing are considered separately. A system is first designed to deliver required functionality. Its timing is then analyzed, and possibly optimized, if it doesn't meet the requirements. Optimizing the timing requires iteratively restructuring functional specification, creating detailed implementation from such a specification, and them analyzing timing behavior of such an implementation. The process is quite long, because meaningful timing optimizations often require changes at the highest level of abstraction, but the impact of these changes can be accurately evaluated only when all implementation details are known.

The separation of concerns is common to both hardware and software design. It has been very successful because it is significantly simpler to model and design system if only functionality is of interest. However as timing requirements become more stringent, many design iterations may be necessary to meet them. This is known as the *timing closure* problem. Timing closure is a major headache for project managers, as the large number of iterations can easily increase time-to-market beyond those required to meet ever shrinking market windows. A delay of a month or two, can cause the latest gadget to be late for the holiday season, which can in turn decimate its total revenue.

In this abstract, we will review the approach which allows timing consideration to be brought in early in the design cycle, while still keeping them orthogonal to functionality. In this way, the advantages of orthogonalization of concerns are preserved, but timing iteration are much shorter. Because timing is modeled at the highest level of abstraction, implementations that do not meet timing requirements can be ruled out fairly quickly, without going through time consuming detailed design. Moreover, modeling alternative implementations is relatively easy, as only the architectural components responsible for timing need to be changed, while functional description remains the same.

More precisely, in Section 2 we will review how functionality and timing are modeled orthogonally in the Metropolis design environment [3]. Then, in Section 3, we will review the Logic of Constraints, a formalism used in Metropolis to express and verify timing and other performance related requirements. Finally, we will conclude in Section 4 with a few thought how these concepts relate to existing UML features, and how they can potentially influence the future of UML.

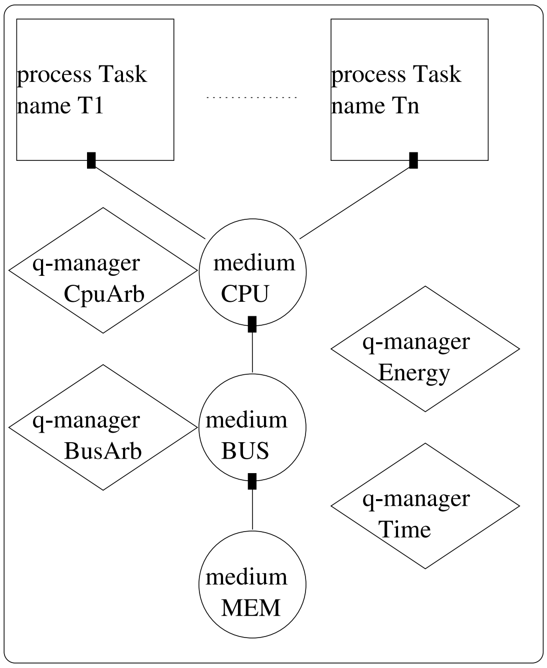
2 Representing Time

In Metropolis, the function of a system is described as a set of objects that concurrently take actions while communicating with each other. We call such an object *process*, and associate a sequential program with it. A process communicates through its *ports*. Ports are connected to *media*. Any medium can be connected to a port if it implements the *interface* of the port. This mechanism, which we borrowed from object-oriented languages such as Java [1], allows the meta-model to separate computation carried out by processes, from communication among them. This separation is essential to facilitate the description of the objects to be reused for other designs. The functional description in Metropolis is un-timed. Timing behavior is available only after the functional description is *mapped* the model of implementation architecture.

Architectures are distinguished by two aspects: the computation that they are capable of implementing, and the efficiency of the implementation. In Metropolis, the former is modeled as a set of *services* offered by an architecture to the functional model. Services are just methods, bundled into interfaces [8]. To represent efficiency of an implementation we need to model the cost of each service. This is done first by decomposing each service into a sequence of events, and then by annotating each event with a value representing the cost of the event.

To decompose the services into sequences of events, we use networks of media and processes, just as in the functional model. These networks often correspond to physical structures of implementation platforms.

For example, Figure 1 shows an architecture consisting of n processes, *T1, . . . ,Tn*, and three media, *CPU*, *BUS*, *MEM*. The architecture in Figure 1 also contains so called *quantity managers*, represented by diamond-shaped symbols, which we will introduce shortly. The processes model software tasks executing on a




```
process Task {
  port CpuService cpu;

  void execute(int n) {
    {$
      ... // make request to CpuArb
      ... // to become CPU owner
    $}
    cpu.execute(n);
  }
  void read() {
    ...
    cpu.read();
  }
  void write() { ... }
  void thread() { ... }
}
```

```
medium MEM implements SlaveService {
  void read {
    {$
      ... // make request to Time
      ... //  for a memory read delay
    $}
  }
  void write { ... }
}
```

```
medium CPU implements CpuService {
  prt BusService bus;

  void execute(int n) {
    {$
      ... // make request to Time
      ... //  for a delay of n clock cycles
    $}
  }
  void read()  { ...   bus.read(); }
  void write() { ...   bus.write(); }
}
```

```
medium BUS implements BusService {
  port SlaveService mem;

  void read()  {

    {$
      ... // make request to BusArb
      ... // to become bus master
    $}
    mem.read();
  }
  void write() { ...   mem.write(); }
}
```

Figure 1. An architectural model

CPU, while the media model the CPU, bus, and memory. The services *offered* by this architecture are the *execute, read* and *write* methods implemented in the *Task* processes. The *thread* function of a *Task* process repeatedly and non-deterministically executes one of the three methods. In this way, we model the fact that the *Tasks* are capable of executing these methods in any order. The actual order will become fixed only after the system function is mapped to the architecture, when each *Task* implements a particular process of the functional model.

While a *Task* process offers its methods to the functional part of the system, the process itself *uses* services offered by the *CPU* medium, which, in turn, uses services of the *BUS* medium. In this way, the top-level services offered by the *Tasks* are decomposed into sequences of events throughout the architecture.

Metropolis supports the notion of *quantity* used to annotate individual events with values measuring cost. For example, in Figure 1 there is a quantity named *energy* used to annotate each event with the energy required to process it. To specify that a given event takes a given amount of energy, we associate with that event a *request* for that amount. These requests are made to the object called *quantity manager*, which collects all requests and fulfills them, if possible.

Quantities can also be used to model shared resources. For example, in Figure 1 quantity *CpuArb* labels every event with the task identifier of the current CPU owner. Assuming that a process can progress only if it is the current CPU owner, the *CpuArb* manager effectively models the CPU scheduling algorithm.

The meta-model has no built-in notion of time, but the time can be modeled as yet another quantity that puts an annotation, in this case a time-stamp, to each event. Managers for common quantities, such as time, are provided as standard libraries with Metropolis, and are understood directly by some tools (e.g. time-driven simulators) for the sake of efficiency. However, quantity managers can also be written by design flow developers, in order to support quantities that are relevant for a specific application domain.

3 Representing Timing Requirements

In this section, we review *Logic of Constraints (LOC)* [2], a formalism designed to express quantitative constraints. LOC is compatible with a wide range of functional specification formalisms that describe a system as a network of components communicating through fixed interconnections. The observed behavior of the system is usually characterized by sequences of values observed at the interconnections. LOC is a formalism designed to reason about traces from the execution of a system. It consists of all the terms and operators allowed in sentential logic, with additions that make it possible to specify system level quantitative functional and performance constraints without compromising the ease of analysis.

LOC can be used to specify very common and useful real-time performance constraints.

- rate, e.g. "$Display$'s are produced every 10 time units":

$$t(Display[i+1]) - t(Display[i]) = 10 \ , \tag{1}$$

- latency, e.g. "$Display$ is generated no more than 25 time units after $Stimuli$":

$$t(Display[i]) - t(Stimuli[i]) \leq 25 \ , \tag{2}$$

- jitter, e.g. "every $Display$ is no more than 4 time units away from the corresponding tick of the real-time clock with period 10":

$$\mid t(Display[i]) - (i+1) * 10 \mid \leq 4 \ , \tag{3}$$

- throughput, e.g. "at least 100 $Display$ events will be produced in any period of 1001 time units":

$$t(Display[i+100]) - t(Display[i]) \leq 1001 \ , \tag{4}$$

- burstiness, e.g. "no more than 1000 $Display$ events will arrive in any period of 9999 time units":

$$t(Display[i + 1000]) - t(Display[i]) > 9999 \quad . \qquad (5)$$

It should be emphasized that time is only one of the possible annotations. Any value that may be associated with an event (e.g. power, area, data value) can be used as an annotation. The basic blocks of LOC formulas are terms, which can be either:

- constants, or
- integer variable i (the only index variable that can appear in an LOC formula), or
- expressions of the form $a(e[n])$, where a is an annotation name, e is an event name, and *index expression* n is an integer-valued term, or
- combination of simpler terms using operators.

We interpret $a(e[n])$ as the value of annotation a of the n-th occurrence of event e. All other terms are interpreted naturally. Terms can be combined using relational operators to create atomic LOC formulas. Finally, LOC formulas are standard Boolean expressions over atomic formulas.

Models of LOC formulas contain a sequence of occurrences for each event name in the formula. We call such structures *annotated behaviors*. Each occurrence may be annotated with some annotation, but we do not require each annotation appearing in the formula to be defined. This feature is important for our design methodology, where performance requirements are specified early in the process, even though they can be evaluated much later, when many implementation details are set.

Given an annotated behavior, we evaluate the formula for each value of index variable i. This is done in quite a standard fashion, except that we need to consider the fact that some terms may not be defined (either because there are only finitely many occurrences of an event, or because an annotation is not defined for an existing event occurrence). To deal with this, we introduce the third logical value $undef$. In general, all operators (including Boolean) return $undef$ if one of their operands are $undef$. The only exceptions are conjunction with $false$ (which is $false$), and disjunction with $true$ (which is $true$). Finally, we say that the annotated behavior satisfies the formula if it does not evaluate to $false$ for any value of i.

Given an LOC formula, it is quite easy to construct a monitor to check whether a particular simulation trace satisfies the formula [5]. In general, it is not possible to automatically and formally prove that whether a system satisfies an LOC formula, because the related decision problem is undecidable. However, there is a significant subset for which the decision problem is decidable. This subset includes the rate and throughput properties discussed above. Standard automatic formal verification techniques can be also applied to other LOC formula, but in this case, the procedure may sometimes terminate without proving or refuting that the system satisfies the formula.

4 Relation to UML

UML is certainly a solid candidate for a language to specify embedded systems at a high level of abstraction, but to fully support a complete design methodology, appropriate profiles and stereotypes need to be defined. These UML extensions need to have sufficiently detailed semantics, so that refinement to a lower level of abstraction can be partially or completely automated. *UML Platform Profile* [4] is one approach in this direction. This approach is essentially a translation of key Metropolis concepts into UML. Since objects in this profile correspond directly to networks of Metropolis processes and media, these networks are used to precisely define semantics of profile objects. However, not all the concepts from Metropolis where translated. In particular, quantity resolution mechanism (and thus also time) is not represented in the profile. It is still an open challenge to extend the Platform Profile with timing information in a way that is precise and complete, but still respects the spirit of UML of being simple and intuitive.

Another approach to embedded system design with UML is represented with *UML Profile for Schedulability, Performance and Time* (SPT Profile, for short) [7]. In the SPT profile, tags are used to attach to behaviors parameters required by various performance modeling techniques. Clearly, tags are reminiscent of annotations used by LOC. The basic difference is that in the SPT profiles relies on separately developed analysis methods, and does not allow the user to create properties, as LOC does. The downside of the LOC approach is that not all of the LOC properties that a user can create, can be verified (at least not formally). LOC is also related to *Object Constraint Language (OCL)* [6]. In fact, the complement each other quite nicely, OCL being much better to expresses static relationship between objects, and LOC being much stronger in expressing properties of execution sequences.

In summary, there is a growing interest in using UML as a starting point of design of embedded real-time systems. However, a lot of work remains to be done to create a complete design methodology, form high-level UML specification all the way to implementation.

Acknowledgments

Under the umbrella of the Metropolis project, many people have contributed to the ideas reviewed here and their implementations. I would like to acknowledge them and thank for collaboration all of them, and particularly the following: prof. Alberto Sangiovanni-Vincentelli, Yosinori Watanabe, Luciano Lavagno, prof. Harry Hsieh, Xi Chen, Grant Martin, and Guang Yang.

References

[1] K. Arnold and J. Gosling. *The Java programming language*. Addison Wesley, 1996.

[2] F. Balarin, Y. Watanabe J. Burch, L. Lavagno, R. Passerone, and A. Sangiovanni-Vincentelli. Constraints specification at higher levels of abstraction. In *Proceedings of the 6th Annual IEEE International Workshop on High Level Design Validation and Test - HLDVT'01*. IEEE Computer Society Press, November 2001.

[3] F. Balarin, Y. Watanabe, H. Hsieh, L. Lavagno, C. Passerone, and A. Sangiovanni-Vincentelli. Metropolis: an integrated electronic system design environment. *Computer*, 36(4):45–52, April 2003.

[4] R. Chen, M. Sgroi, L. Lavagno, G. Martin, A. Sangiovanni-Vincentelli, and J. Rabaey. UML and platform-based design. In L. Lavagno, G. Martin, and B. Selic, editors, *UML for Real: Design of Embedded Real-Time Systems*, pages 107–126. Kluwer Academic Publishers, 2003.

[5] Xi Chen, Harry Hsieh, Felice Balarin, and Yosinori Watanabe. Verifying LOC based functional and performance constraints. In *Proceedings of the 8th Annual IEEE International Workshop on High Level Design Validation and Test - HLDVT'03*. IEEE Computer Society Press, November 2003.

[6] Object Constraint Language specification. available at http://www.omg.org, 1997.

[7] D. C. Petriu and C. M. Woodside. Peformance analysis with UML. In L. Lavagno, G. Martin, and B. Selic, editors, *UML for Real: Design of Embedded Real-Time Systems*, pages 221–240. Kluwer Academic Publishers, 2003.

[8] S. Solden. Architectural services modeling for performance in HW-SW co-design. In *Proceedings of the Workshop on Synthesis And System Integration of MIxed Technologies SASIMI2001, Nara, Japan, October 18-19, 2001*, pages 72–77, 2001.