Time-Triggered vs. Event-Triggered: A matter of configuration?

Fabian Scheler, Wolfgang Schröder-Preikschat

Friedrich-Alexander University Erlangen-Nuremberg
Department of Computer Science 4
{scheler,wosch}@informatik.uni-erlangen.de

Abstract. During the development of real-time systems one has either to plump for a time-triggered or an event-triggered architecture. Actually this decision deals with a non-functional property of a real-time system and should therefore be postponed as far as possible. Unfortunately, this property also exhibits functional qualities during the development of real-time systems making this postponement impossible and a subsequent transition very expensive. This paper sketches an approach to specify a real-time system independent of its architecture (time-triggered or event-triggered), thus facilitating to switch between a time-triggered and an event-triggered architecture easily.

1 Introduction

The question, whether to choose a time-triggered or an event-triggered architecture for building a real-time system, is discussed in an extremely controversial way. Actually this is quite surprising: whether a time-triggered or an event-triggered approach is chosen, should not matter as long as all deadlines can be met. Therefore this property can be regarded as a non-functional property of real-time systems. In current real-time systems development, however, the real-time architecture is a functional property. This makes it nearly impossible to migrate from a time-triggered to an event-triggered architecture and vice versa when certain constraints, favouring one of these approaches, change. In many cases, a complete redesign and reimplementation would become necessary. This paper sketches a method that lets one easily switch between these architectures by specifying the real-time system completely independent of them. Furthermore, this method can also be useful for optimizing real-time systems with respect to e.g. context switching and synchronization overhead or generating a carefully tailored runtime system.

The paper is structured as follows: section 2 sums up the major criterions that influence the decision in favor of a time-triggered or an event-triggered architecture today. In section 3 the notion of an *atomic basic block* is introduced and it is suggested how they can be used to real-time systems independent from a certain real-time architecture. Section 4 presents some related work and section 5 concludes the paper.

2 Time-triggered versus Event-Triggered?

The factors influencing the decision for a time-triggered or an event-triggered architecture shall be shortly summed up here. For most of these criterions Kopetz already gave a

comparison of time-triggered and event-triggered systems [1], nevertheless these criterions are taken into account again, as the prerequisites leading to those appraisals meanwhile may have changed. Interestingly, all these criterions are actually non-functional.

Analyzability Deciding the schedulability of a set of tasks and finding an appropriate schedule under the given temporal constraints is crucial for real-time systems and in the general case a NP-hard problem. Hence it is evident that this problem cannot be faced during runtime, the schedulability and the given temporal constraints have to be ensured beforehand. For time-triggered systems this results in statically computed schedules (e.g. [2]), while a thorough response time analysis (e.g. [3]) can guarantee that all deadlines are met in an event-triggered system. Both methods are suitable to ensure deadlines also in peak load scenarios and require detailed knowledge about the timing constraints of the controlled real-time object. Hence, neither time-triggered systems nor event-triggered systems are to be preferred with respect to analyzability.

Predictability Time-triggered systems follow a statically computed schedule, whereas the schedule of an event-triggered system unfolds dynamically during runtime, depending on the occurrence of different events. It is obvious that it is impossible to predict the concrete state of an event-triggered system at a given point in time, because only few assumptions on the occurrence of aperiodic and sporadic events can be made, while this is easy within time-triggered systems. But one should keep in mind, that a real-time system does not have to be predictable by all means. In order to guarantee deadlines, it is sufficient to be deterministic. Hence, neither time-triggered systems nor event-triggered systems are to be preferred with respect to predictability.

Testability Functional testing should not differ so much for time-triggered and event-triggered systems. The important thing is how timing constraints are verified. In both kinds of architectures it is sufficient to test each system task for its worst case performance. The schedulability of the whole system must be ensured afterwards by formal techniques. Such techniques exist for both, time-triggered and event-triggered systems (statically computed schedules, response time analysis). Testing with typical load scenarios is not enough, when hard deadlines have to be kept. Hence, neither time-triggered systems nor event-triggered systems are to be preferred with respect to testability.

Extensibility Extensibility stands for the costs one has to pay when he needs to add new functionality, i.e. new tasks, to an existing system. From a functional point of view those costs mainly depend on the interaction between the new tasks and already existing tasks. From the real-time viewpoint deadlines also have to be guaranteed within the extended system. In case of time-triggered systems the static schedules have to be recomputed. In case of event-triggered systems the response time analysis has to be done again. Hence, neither time-triggered systems nor event-triggered systems are to be preferred with respect to extensibility.

Fault Tolerance For reasons of strict timing constraints fault tolerance is often based on active redundancy within real-time systems, though active redundancy requires replica determinism. Replica determinism means that redundant nodes have to take the same decision at about the same time. While this is for free in time-triggered

systems, state synchronism is very hard to achieve in event-triggered systems. Mechanisms like the leader-follower model [4] are needed in event-triggered systems. These mechanisms require additional communication between the redundant nodes and, thus, cause computational overhead. So time-triggered system are to be preferred with respect to fault tolerance.

Resource Utilization Even in hard real-time systems not all events to be serviced are strictly periodic. Moreover these aperiodic and sporadic events can hardly be mapped to a strictly cyclic schedule in an efficient way. Polling of such events requires to poll them at least with the double rate of their maximum occurrence rate [5], while the average response time for servicing such events is still quite poor. The alternative would be to service such events in an event-triggered manner in a time-triggered system. Polling would no longer be necessary and the average response time would significantly be improved, but this alternative also imposes all other drawbacks of event-triggered systems. So event-triggered system are to be preferred with respect to resource utilization, when aperiodic and sporadic events are relevant.

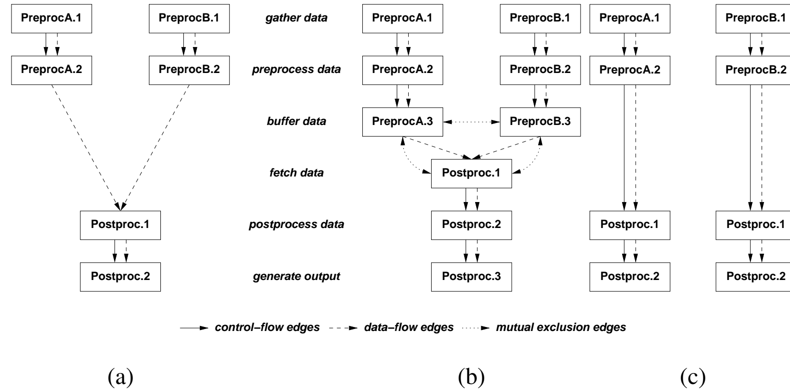
Overall, the only criterions having a substantial impact on the selection between a time-triggered or an even-triggered architecture are *fault tolerance* and *resource utilization*, the other factor neither favor the one nor the other approach. So why is this decision not postponed until it is clear whether a failure tolerant real-time system should be built or not or if aperiodic or sporadic events are relevant?

The reason is: whether a time-triggered or an event-triggered approach is followed, exhibits functional qualities during the development process of a real-time system. Time-triggered and event-triggered systems often provide a completely different programming model in terms of control flow abstractions. Whereas tasks often have run-to-completion semantics within time-triggered systems and also task-synchronization is taken care of ahead of runtime. Tasks in event-triggered systems, however, can be preempted by any other task with a higher priority, so task synchronization, such as scheduling, has also to be done online.

3 Atomic Basic Blocks

In order to postpone the decision on the real-time architecture to be used as far as possible, it is necessary to be independent of any concrete control flow abstraction. As pointed out at the end of section 2 control flow abstractions form the major difference between time-triggered and event-triggered systems. To enable the postponement of this decision, the notion of *atomic basic blocks* is introduced.

An *atomic basic block* (ABB) is a section of the control flow that has to be executed atomically in order to ensure the consistency of data, that is affected within this ABB. Most of these ABBs are identical to minimal basic blocks known from compiler construction, but an ABB can also span a complete critical section. ABBs are basically arranged in three different graphs: a control-flow graph, a data-flow graph and a mutual exclusion graph. The control-flow graph and the data-flow graph are directed graphs



(a) The control- and data-flows among the different ABBs. The fusion of the two branches of the data-flow graph can have either *or-* or *and-semantics. And-semantics* means, that pre-processed data of both sources has to be available before post-processing can take place, with *or-semantics* pre-processed data of only one source is sufficient.

(b) In case of *and-semantics*, the pre-processed data from both sources has to be available before post-processing can start, buffering of the pre-processed data becomes inevitable, therefore additional ABBs are added to the control- and data-flow graph. These ABBs have to be executed mutually exclusive, as each of them accesses a common buffer.

(c) In case of *or-semantics,* the post-processing phase can be *duplicated,* e.g. to avoid context switching overhead or priority violations within event-triggered systems.

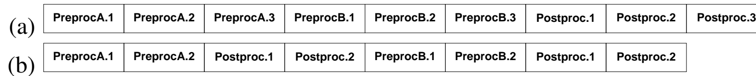
Fig. 1. Control-flow, data-flow and mutual exclusion graphs

depicting the flow of control and data between the different ABBs, while the mutual exclusion graph is undirected and expresses mutual exclusion constraints among different ABBs.

As a short example consider the following scenario: we have two data sources (`sourceA` and `sourceB`). Data is gathered and pre-processed so it can be post-processed by the same algorithm. Finally, some output is generated from the completely processed data. Each of these activities is regarded as an ABB for reasons of simplicity, these ABBs can be arranged in a control- and data-flow graphs shown figure 1(a).

This control-flow graph has now to be mapped to a concrete control-flow abstraction so that the data-dependencies and mutual exclusion constraints specified by the data-flow and the mutual exclusion graphs are satisfied. Possible mappings depend on the semantics of the fusion of the two paths in the data-flow graph. In case of *and-semantics* (figure 1(b)) it is not feasible to modify the control-flow, while in case of *or-semantics* (figure 1(c)) the post-processing phase can be duplicated, in order to e.g. reduce the amount of context switches. For time-triggered systems the non-linear control-flow graph has to be serialized, the immediate result is a possible structure for a static schedule (figure 2). For event-triggered systems the different ABBs are mapped to a set of tasks (figure 3). In both cases some *glue code* has to be generated potentially to implement the data-flow between the different ABBs, in an event-triggered system additional *glue code* for flow control is necessary.

Furthermore it is imaginable to augment these graphs with additional information gathered during the design process of a real-time system, such as timing and other



In case of and-semantics of the fusion of the data-flow graph's paths (figure 1(b)) a static schedule with structure of (a) has to be generated, in case of or-semantics (figure 1(c)) a static schedule with the structure of (b) would also be possible.

Fig. 2. Time-triggered mapping

```
TASK(PreprocA) {
  /* get and pre-process data (source A) */
  GetResource(data_buffer);
  /* buffer pre-processed data */
  ReleaseResource(data_buffer);
  IncrementCounter(PostprocCounter);
}
```

```
TASK(Postproc) {
  GetResource(data_buffer);
  /* get buffered data */
  ReleaseResource(data_buffer);
  /* postprocess data and generate output */
  SetRelAlarm(PostProcAlarm,2,0);
  TerminateTask();
}
```

The listing above shows a possible mapping of figure 1(b) to a set of OSEK-Tasks. The pre-processing phase is mapped to two independent tasks `PreProcA` and `PreProcB` (task `PreProcB` can be implemented analogue to task `PreProcA`), the software counter `PostProcCounter` is incremented when pre-processing is finished. The counter runs an alarm `PostProcAlarm` that activates the post-processing task `PostProc` on expiration.

Fig. 3. Event-triggered mapping

precedence constraints among different ABBs. This additional a-priori knowledge can be exploited to generate an improved mapping to a static schedule or a set of tasks. T.m. the amount of possible context switches could be minimized or a suitable synchronization protocol could be chosen. One can even think of utilizing this knowledge to automatically tailor down the underlying operating or runtime system so it exactly fits the needs of the application described by these graphs.

4 Related Work

The Cluster Compiler [6] by Kopetz and Nossal is an offline planning tool for developing distributed, time-triggered real-time systems. It is used for assigning tasks to distinct nodes within the distributed system and calculating static task and message schedules. It is not intended for tailoring the underlying runtime system or even switching between a time-triggered and an event-triggered architecture.

Yokoyama presents an approach [7], that allows him to model the data-flow in a time-triggered, event-triggered or demand-triggered way by separation of functional and behavioral design with the help of aspects. However, he does not consider mapping this data-flow to concrete control flow abstractions.

The Time Weaver framework [8] by de Niz and Rajkumar is aimed at separating *functional* from *para-functional properties* within real-time systems. The programming model is based on *event processing components* that are connected by so called *couplers* serving mainly the purpose of communication between the different components. These components are finally projected onto a generated runtime system with respect to different dimensions (event flow, deployment, timing, modality), t.m. the different

couplers are filled with an appropriate implementation. This framework is not designed for a real-time architecture independent description of a real-time system. In fact it is intended to model event-triggered systems. Furthermore the notion of components proposed in this paper is much coarser then the notion of ABBs presented here. Mapping of these components to concrete control flow abstractions across component boundaries or tailoring the underlying operating system is also not considered.

5 Conclusion

The decision of either settling for a time-triggered or an event-triggered architecture clearly deals with non-functional properties of a real-time system, and should therefore be postponed as far as possible. Unfortunately these architectures expose functional behaviour during the development of real-time systems, that prevents a postponement and remarkably hamper a later migration from one architecture to the other. This paper sketches a method to describe a real-time system independent of the architecture to be used later, by the notion of atomic basic blocks. A real-time system described on the level of atomic basic blocks can be mapped either to a time-triggered or an event-triggered architecture. The final mapping can take place at a late point in the progress of development, when it is clear which of these approaches is most helpful. Furthermore this method can also be used to optimize the real-time system with respect to e.g. context switching or synchronization overhead and to careful tailor the underlying operating system.

References

1. Kopetz, H.: Event-triggered versus time-triggered real-time systems. In: Proceedings of the International Workshop on Operating Systems of the 90s and Beyond, London, UK, Springer-Verlag (1991) 87–101
2. Schild, K., Wrtz, J.: Off-line scheduling of a real-time system. In: Proceedings of the 13th ACM Symposium on Applied computing (SAC '98), New York, NY, USA, ACM Press (1998) 29–38
3. Liu, J.W.S.: Real-Time Systems. Prentice Hall PTR, Upper Saddle River, NJ, USA (2000)
4. Barrett, P.A., Speirs, N.A.: Towards an integrated approach to fault tolerance in delta-4. Distributed Systems Engineering **1** (1993) 59–66
5. Briand, L.C., Roy, D.M.: Meeting Deadlines in Hard Real-Time Systems. IEEE Computer Society Press, Los Alamitos, CA, USA (1997)
6. Kopetz, H., Nossal, R.: The cluster compiler - a tool for the design of time-triggered real-time systems. In: ACM SIGPLAN Workshop on Languages, Compilers and Tools for Real-Time Systems, New York, NY, USA, ACM Press (1995) 108–116
7. Yokoyama, T.: An aspect-oriented development method for embedded control systems with time-triggered and event-triggered processing. In: Proceedings of the 11th IEEE International Symposium on Real Time and Embedded Technology and Applications (RTAS '05), IEEE Computer Society Press (2005) 302–311
8. de Niz, D., Rajkumar, R.: Time weaver: a software-through-models framework for embedded real-time systems. In: Proceedings of the 2003 Joint Conference on Languages, Compilers and Tools for Embedded Systems & Soft. and Compilers for Embedded Systems (LCTES/SCOPES '03), New York, NY, USA, ACM Press (2003) 133–143