Synthesising Real-Time Systems from Atomic Basic Blocks

Fabian Scheler, Wolfgang Schröder-Preikschat
Friedrich-Alexander University Erlangen-Nuremberg
Department of Computer Science 4
Martensstrasse 1, 91058 Erlangen, Germany
{scheler,wosch}@informatik.uni-erlangen.de

Abstract

Whether a real-time system is implemented as time-triggered or event-triggered system is constituted quite early in the development process of real-time systems. Unfortunately, different task models are associated inseparably with these real-time architectures. This makes it very hard to migrate from time-triggered to event-triggered systems and vice versa, also the reuse of individual event-handlers of a real-time system is prohibited by this fact. In this paper we point out that there is no need to prefer a certain real-time architecture in many cases. Therefore, we sketch an architecture-independent representation of real-time systems based on so called atomic basic blocks *(ABB). These ABBs allow to describe reusable event-handlers that are composed into the final real-time system by an automated synthesis.*

1. Introduction

An important decision in state-of-the-art real-time (RT) systems development is to settle either for a time-triggered (TT) or an event-triggered (ET) architecture. This decision is crucial for the development of RT systems as both architectures offer very different control flow abstractions. These different control flow abstractions make a later migration from a TT to an ET system or vice versa impracticable. In many cases such a migration is equivalent to a complete redesign and reimplementation of the entire RT system. Moreover, the reuse of individual event-handlers of a RT system is hardly possible when the RT architecture changes. As RT systems definitely can be built in a TT as well as in an ET fashion, it might be valuable to consider the RT architecture as non-functional property. Thereby, the flexibility and the reusability of RT systems on the level of event-handlers would be increased. In this paper we sketch the notion of *atomic basic blocks* that enables treating the RT architecture as non-functional property and give a short overview on their possible applications.

The rest of the paper is structured as follows: Section 2 succinctly subsumes the differences amongst TT and ET systems regarding their non-functional as well as functional properties. Section 3 sketches a method to describe a RT system independent of the used RT architecture and section 4 gives possible applications of such a representation. Section 5 discusses related work, section 6 gives an outlook on future work and section 7 concludes the paper.

2. Time-triggered versus Event-Triggered?

The TT and the ET approach are now compared with respect to their non-functional as well as their functional properties. The focus of this comparison is to figure out which non-functional factors favour either a TT or an ET architecture and which functional properties inhibit an exchange of these two approaches.

2.1. Non-Functional Properties

The non-functional factors influencing the decision for a TT or an ET architecture shall be shortly summed up here. For most of these criterions Kopetz already gave a comparison of TT and ET systems [7]. Nevertheless these criterions are taken into account again, as the prerequisites leading to those appraisals meanwhile may have changed.

Analyzability Deciding the schedulability of a set of tasks and finding an appropriate schedule under the given temporal constraints is crucial for RT systems. For TT systems (statically computed schedules, e.g. [10]) and ET systems (response time analysis, e.g. [8]) suitable methods to ensure deadlines exist. Hence, neither TT systems nor ET systems are to be preferred with respect to analyzability.

Predictability The state of ET systems cannot be predicted exactly, of course, while this is easy for TT systems, as these systems follow a statically computed schedule. However, a RT system does not necessarily have to be predictable. In order to guarantee deadlines, it is sufficient to be deterministic. Hence, neither TT systems nor ET systems are to be preferred with respect to predictability.

Testability Functional testing is similar among TT and ET systems. It is more important how timing constraints are verified. In both kinds of architectures it is sufficient to test each event-handler for its worst case performance. The schedulability of the whole system must be ensured afterwards by formal techniques. As such techniques exist for both, TT and ET systems, neither TT systems nor ET 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, e.g. new event-handlers, to an existing system. In RT systems deadlines also have to be guaranteed within the extended system. That fact is taken care of by recomputing the static schedules in TT systems and repeating the response-time analysis in ET systems. Hence, neither TT systems nor ET 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 RT systems, though active redundancy requires replica determinism. While this is almost for free in TT systems, state synchronism is very hard to achieve in ET systems. So TT system are to be preferred with respect to fault tolerance.

Resource Utilization Even in hard RT systems not all events to be serviced are strictly periodic. In TT systems such aperiodic and sporadic events have to be polled. This imposes a significant run-time overhead, while the average response time for servicing such events is still quite poor. As alternative such events can be serviced in an ET manner in a TT system, but also imposing all other drawbacks of ET systems. So ET systems are to be preferred with respect to resource utilization, when non-periodic events are relevant.

2.2. Functional Properties

The main functional difference between TT and ET systems is the control flow abstraction used in each case. In TT systems jobs are executed according to a statically computed schedule. Synchronisation among different jobs is performed ahead of run-time, so jobs often expose a run-to-completion semantics. Moreover, usage of certain system services like communication often is restricted to certain points in time in order to guarantee predictable behaviour. In ET systems on the other hand schedules unfold dynamically during runtime as per the emergence of events. This entails explicit synchronisation among different jobs, altogether, with all the pitfalls arising with explicit synchronisation mechanisms. However, less restrictions on the usage of system services are imposed and ET systems appear easier to be handled for many developers.

2.3. Conclusion

All in all, the only non-functional criterions having a substantial impact on the selection between a TT or an ET architecture are *fault tolerance* and *resource utilization*. The other factors neither favor the one nor the other approach. Unfortunately this selection cannot be postponed because of the functional difference between these RT architectures, namely, their control flow abstractions.

3. Architecture Independent RT Systems

In order to make a RT system independent of its architecture one has to bypass the functional differences between the TT and the ET approach. As pointed out in the foregoing section the major difference are the control flow abstractions significant for those approaches. In this section we sketch a method that can be used to describe a RT system independent of the RT architecture and how it can be applied in the development of RT systems.

3.1. Atomic Basic Blocks

To bypass the functional difference between TT and ET systems any control flow abstraction has to be omitted within the representation of a RT system. Therefore, a RT system is described by *atomic basic blocks*.

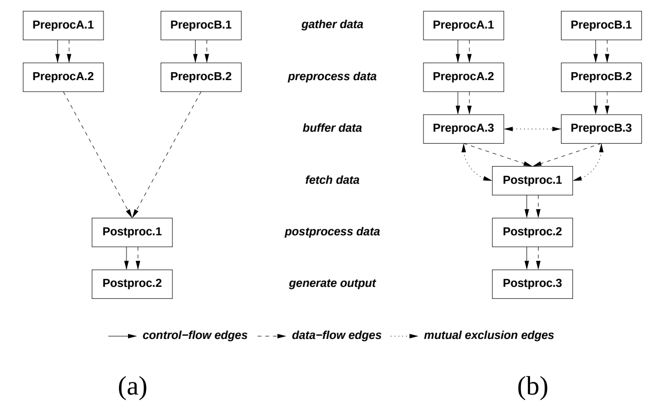
An *atomic basic block* (ABB) is a section of the control flow that ensures the consistency of the data, that is affected within this ABB. Most of these ABBs are identical to minimal basic blocks known from compiler construction, but an ABB can also span a complete critical section. ABBs are basically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

As a short example consider the following scenario: we have two data sources (`sourceA` and `sourceB`). Data is gathered from them 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. Figure 1 shows the representation of such a system by means of ABBs. These ABB-graphs can be mapped to a set of threads for an ET system (see figure 2), but the generation of a static schedule suitable for a TT system is thinkable, too.

3.2. Development Process

An important question is, how these ABB-graphs (i.e. the control-flow graph, the data-flow graph and the mutual-exclusion graph) are provided and how they are mapped to control flow abstractions. To achieve an efficient and flexible mapping ABBs should be as fine grained as possible and, therefore, best be generated automatically.

A possible source for the generation of such ABB-graphs would be to describe an event-handler as simple end-to-end-scenario by means of e.g. a simple procedure. Such



(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*.

(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.

Figure 1. ABB-graphs

event-handlers are most reasonable if they do not depend on other event-handlers, otherwise, these dependencies have to be modeled explicitly by e.g. annotations. Furthermore, it should be feasible to specify such event-handlers e.g. by design and modeling tools like matlab/simulink. Another, really exciting source for ABBs might be provided by already existing RT systems. Thereby, it might be possible to extract event-handlers from existing RT systems. Any information that is needed to build those ABB-graphs is already present in ET systems. For TT systems a given static schedule can additionally be exploited to deduce the control-flow and data-flow information while mutual exclusion will demand manual intervention.

The later mapping to a concrete control flow abstraction, thus, the synthesis of the final RT system should also go on automatically. During the synthesis the ABBs are being connected by some *glue code* taking care of implementing the control-flow, data-flow and mutual exclusion constraints. Depending on the requirements of the RT object a TT or an ET platform can be chosen as target platform and an application for this platform is generated. Such platforms can be already existing RT operating systems, but it is also thinkable to generate or configure such a platform along with the generation of the application.

4. Cases of Application

ABBs are not restricted to facilitate an architecture independent representation of RT systems only. This section gives a short overview on some applications of ABBs for the synthesis of RT systems.

```
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 an OSEK-application [1]. The pre-processing phase is mapped to two independent tasks `PreProcA` and `PreProcB`, 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.

Figure 2. Event-triggered mapping

4.1. Reusable Event-Handlers

Currently, RT systems are only reusable at the level of components, modules or even complete nodes in distributed RT systems. Drivers and libraries, of course, can be easily embedded into different RT systems in most cases, but complete event-handlers are only seldom reused. The reason is that too many assumptions on the particular RT system are hard-coded in such event-handlers. Those assumptions range from a certain RT architecture over the semantics of system services to the capability of peripheral hardware to issue interrupts. ABBs are completely independent of such assumptions and would therefore enable the reuse of complete event-handlers.

4.2. Optimisation

ABB-graphs cary enough information to synthesise the final RT system. Additional ABB-graphs can carry more information about the RT system that can help to optimise the synthesis. The run-time consumption of the ABBs and occurrence patterns of events (given e.g. by period and phase) could be used to optimise synchronisation, for instance. A clever mapping of ABBs to tasks and the selection of a suitable synchronisation protocol, maybe, could secure that some critical sections can never coincide making synchronisation superfluous or, at least, minimize the blocking time. Moreover, all the information embedded into such ABB-graphs can finally be exploited to either configure the target platform, or even better, to synthesise it as well, so the platform suits best the demands of the RT system.

4.3. Scheduling

An ABB-graph equipped with timing information can also be used to check the schedulability of RT systems during their synthesis by e.g. a response time analysis. Conventional response time analysis for ET systems often is too pessimistic due to very complex task models in current RT operating systems. With the help of ABBs a RT system could be synthesised with respect to the adequacy of estimated response times. Should a RT system turn out to be not schedulable anyway, a detailed timing analysis of the RT system is possible on the level of ABBs.

Along with TT systems such graphs could help to generate static schedules. In TT systems static schedules are often divided in minor and major cycles. In the cyclic executive model [2], for instance, the length of minor cycles also imposes restrictions on the WCET of event-handlers: each event-handler has to fit in exactly one minor cycle. Event-handlers that consume more run-time have to be split in junks suitable for the length of one minor cycle. With ABBs this splitting could be done automatically by mapping ABBs appropriately to minor cycles.

5. Related Work

There already exist a number of papers dealing with the synthesis of RT systems. In most cases component frameworks are used to model RT systems, so synthesising results in mapping the different components to a set of tasks and generating some *glue code* to adapt the components to the used task model. Bordin et al [3] and the TimeWeaver framework by de Niz and Rajkumar [5] are two examples. Gu and Shin [6] proceed one step further. Components are mapped to threads according to different multi-threading strategies and a method to analyse the schedulability for a component based multi-threading strategy is presented. Chou and Boriello [4], on the other side, try to increase the retargetability of RT systems by synthesising the run-time system, while a fixed task model is employed. Only few work closing the gap between TT and ET systems has been published so far. Yokoyama [11] presents an approach, that allows him to model the data-flow in a TT, ET or demand-triggered way. However, he does not consider mapping this data-flow to concrete control flow abstractions.

The method sketched in this paper combines and extends the approaches for synthesising RT systems mentioned above. Fine grained components, namely ABBs, are mapped in a flexible way to a tailored run-time system. Thus, facilitating reusable event-handlers that can be mapped to even very different RT systems platforms.

6. Future Work and Evaluation

From this paper it is evident that the work on ABBs is still at a very early stage. As a first step we plan to extend the GNU compiler collection (GCC) by new frontends to evaluate the automatic generation of ABBs from plain C programs, UML models and existing RT systems. As a second step we want to investigate the mapping of ABBs to different TT and ET single-node operating systems like OSEK and OSEKtime [1]. In the third step the synthesised RT system should be optimised and a highly adaptable operating system like CiAO [9] should be targeted and configured as optimal as possible. Finally also distributed systems shall be taken into account. During all phases real-world RT systems (e.g. from the automotive area) will serve as examples to check the applicability of ABBs.

7. Conclusion

In current RT systems the used RT architecture often is hard-coded for some reason, and together with it, also the task model is. This makes it nearly impossible to migrate from TT to ET systems and vice versa. Moreover, the reuse of event-handlers is not viable when the RT architecture changes. This paper shows that there are no reasons to fix the RT architecture in many cases. Thus, ABBs are introduced to describe RT systems independent of their architecture enabling the development or RT architecture independent and reusable event-handlers. The final RT system can then be synthesised from a set of event-handlers described by ABBs.

References

[1] OSEK/VDX standard. http://www.osek-vdx.org/.

[2] T. P. Baker. The cyclic executive model and ada. In *9th IEEE Int. Symp. on Real-Time Systems*. IEEE, 1988.

[3] M. Bordin. Automated model-based generation of ravenscar-compliant source code. In *17th Eurom. Conf. on Real-Time Systems*. IEEE, 2005.

[4] P. Chou. Software architecture synthesis for retargetable real-time embedded systems. In *5th Int. W'shop on HW/SW Co-Design*. IEEE, 1997.

[5] D. de Niz. Time weaver: a software-through-models framework for embedded real-time systems. In *2003 Joint LCTES & SCOPES Conferences*. ACM, 2003.

[6] Z. Gu. Synthesis of real-time implementations from component-based software models. In *26th IEEE Int. Symp. on Real-Time Systems*. IEEE, 2005.

[7] H. Kopetz. Event-triggered versus time-triggered real-time systems. In *Int. W'shop on Operating Systems of the 90s and Beyond*. Springer, 1991.

[8] J. W. S. Liu. *Real-Time Systems*. Prentice Hall, 2000.

[9] D. Lohmann. Architecture-Neutral Operating System Components. *23rd ACM Symp. on OS Principles*, 2003. WiP presentation.

[10] K. Schild. Off-line scheduling of a real-time system. In *ACM Symp. on Applied Computing*. ACM, 1998.

[11] T. Yokoyama. An aspect-oriented development method for embedded control systems with time-triggered and event-triggered processing. In *11th IEEE Int. Symp. on Real-Time and Embedded Technology and Applications*. IEEE, 2005.