The RTSC: Migrating Event-Triggered Systems to Time-Triggered Systems

Fabian Scheler and Wolfgang Schröder-Preikschat
scheler@cs.fau.de,wosch@cs.fau.de

Department of Computer Science 4 - Distributed Systems and Operating Systems
Friedrich–Alexander University Erlangen–Nuremberg

SUMMARY

In this paper we present a prototype of the RTSC – the *Real-Time Systems Compiler*. The RTSC is a compiler-based tool that enables the migration from event-triggered to time-triggered real-time systems. This is achieved by replacing the *real-time systems architecture* of a given real-time system. The real-time systems architecture governs the structural properties of the *white-box view* of a real-time system: how are tasks attached to events and how are dependencies between different tasks implemented. The RTSC uses an abstraction called *Atomic Basic Blocks* (ABBs) to hide the real-time systems architecture and capture all relevant dependencies of an event-triggered system in a global ABB-graph. The RTSC automatically extracts that ABB-graph from an event-triggered real-time system given as source code, transforms that ABB-graph appropriately, and maps it to a statically computed schedule that could be executed by standard time-triggered real-time operating systems. Important temporal properties of the physical environment of the real-time system needed for that transformation are stored in a *system model* provided as additional input to the RTSC. Furthermore, we demonstrate the applicability of our approach and the operation of our prototype by transforming the event-triggered control application into a time-triggered equivalent. Copyright © 0000 John Wiley & Sons, Ltd.

Received . . .

1. INTRODUCTION

At the beginning of any real-time systems project one has the choice either to go for an event-triggered or a time-triggered solution. Event-triggered systems have the advantage of being much more flexible and, thus, are much more handy when dealing with changing requirements or *uncertain knowledge* (aperiodic events for instance as their period is not known). According to Carlow [1] these were the reasons to select an event-triggered approach when building the space shuttle primary avionics system. On the other hand, time-triggered systems can be verified much easier. This is a significant advantage for any kind of dependable system. Therefore, Gagne and Sheppard made a great effort to port the F18 mission computer software to a time-triggered execution environment [2]. A significant part of this porting process had to be done manually. Thus, this was a work-intensive and also a possibly error-prone undertaking. Often the transition from an event-triggered system to a time-triggered system or vice versa is even deemed too cumbersome to be a viable option. In such cases, the redesign of the system tied with the re-implementation of large parts of the system becomes inevitable.

In fact, there are more motivating examples that make a migration between event-triggered and time-triggered systems desirable. One of them is the shift from CAN-based [3] communication to FlexRay-based [4] communication systems in the automotive domain. Here, a purely event-triggered communication system is replaced by a mostly time-triggered one. Although, FlexRay also offers event-triggered communication semantics its real strength is the time-triggered communication mode. So, in order to benefit from FlexRay the affected control units should adopt that time-triggered mode of operation.

This especially holds for so called *X-by-wire* applications like *Steer-by-wire* [5]. Such systems are absolutely safety-critical and are not backed up by mechanical systems. As a consequence, their dependable operation presumably requires some kind of redundancy. A widespread measure for that is *triple module redundancy* (TMR) and it is well known that TMR could much easier be implemented in time-triggered systems than in event-triggered systems [6]. Therefore, adapting the affected control units to a time-triggered execution environment would be very beneficial.

However, the migration between event-triggered and time-triggered systems is very labour intensive and cumbersome. This is mainly owed to the cross-cutting nature of these real-time paradigms. They differ in the way handler functions are activated in response to events and in the mechanisms offered to coordinate multiple simultaneously executed handlers. Event-triggered systems usually offer a variety of mechanisms to implement blocking and non-blocking uni- and multilateral synchronisation. Time-triggered systems, on the other side, impose run-to-completion semantics upon these handlers and order them appropriately in a statically computed schedule table. Thereby, all the places where a particular handler function is activated and all the points where simultaneous handlers interact are tied with the underlying mechanisms offered by employed real-time paradigm. In our work we aim at finding these places in the implementation of event-triggered systems and replace them by the mechanisms available in a time-triggered real-time systems architecture in an automated, tool-based manner, thus, automating that migration process.

1.1. Overview

The approach we pursue to automatically migrate event-triggered to time-triggered systems applies a tool-based source code transformation on the implementation of the given real-time system. We require that this system is "correct" and respects the temporal requirements of its physical surroundings. We assume that for a successful transformation it is sufficient to preserve the following properties:

Internal dependencies among simultaneous event-handlers present in the event-triggered system must also be maintained in the generated time-triggered system.

Temporal requirements imposed by the physical environment must also be met by the generated time-triggered system.

Internal dependencies are covered by a global dependency graph spanned by *Atomic Basic Blocks* (ABBs) [7, 8] that describes the *white-box view* of the real-time system. While the control flow structure of a single event-handler could already be described by an inter-procedural control flow graph (CFG), it is not possible to express dependencies among different event-handlers that way. Thus, the intended purpose of the ABB-based dependency graph is twofold. Firstly, it extends such CFGs to global dependency graphs by incorporating dependencies crossing the borders of event-handlers. Secondly, ABBs partition these CFGs into smaller portions that are inherently independent of the underlying real-time systems architecture. Thereby, ABBs abstract from the real-time paradigm facilitating an automated transition from event-triggered to time-triggered systems.

A *system model* is used to store the temporal requirements of the physical environment, thereby, specifying the *black-box view* of the real-time system. These requirements are then connected to the global dependency graph made up by ABBs. Hereby, these requirements can be taken into account when mapping that dependency graph to a time-triggered execution environment.

In this paper, which is an extended version of our previous paper [9], we introduce the term *real-time systems architecture* to subsume the different mechanisms available to implement dependencies between different event-handlers and the means to attach them to external events. A real-time systems architecture comprises exactly those properties that have to be hidden by ABBs and the system model to represent the system's white-box view independently of the original real-time paradigm.

Furthermore, we present a prototypical implementation of the *Real-Time Systems Compiler* (RTSC) that uses ABB-based dependency graphs to migrate event-triggered to time-triggered systems. The RTSC extracts these graphs from an event-triggered system and finally maps them to statically computed schedule table that is executed by a time-triggered execution environment.

In general, the RTSC is designed to provide a transformation tool between arbitrary real-time systems architectures. However, it currently only supports the migration of event-triggered systems based on OSEK OS [10] to time-triggered systems based on OSEKtime [11]. The transition from time-triggered to event-triggered systems is subject to future work. Furthermore, the RTSC is restricted to single computing nodes with a single processor. It is not capable of handling distributed systems, multi-processor or multi-core systems, so far. Even though, the RTSC implements a scheduling algorithm that also handles distributed systems, it lacks a proper algorithm to do task allocation and to schedule the communication system. There are appropriate algorithms to solve these problems (e.g. the allocation algorithm by Peng [12]) that can be implemented within the RTSC. However, it is very difficult to determine which data elements have to be transferred between the different computing nodes in the general case. Altogether, we believe that both, migrating time-triggered systems to event-triggered systems and supporting distributed systems and multi-core systems are not just a matter of implementation but demand further research. Therefore, we will briefly discuss the challenges coming along with these features later in that paper.

1.2. Contributions

The contributions presented in this paper can be summarised as follows:

- The notion of a *real-time systems architecture* specifying the structural properties of the white-box view of a real-time system.
- A *system model* capturing the black-box view of a real-time system. The main purpose of the system model is to connect the black-box view of a real-time system to its white-box view described by ABBs.
- A prototypical implementation of a compiler-based tool, called *Real-Time Systems Compiler* (RTSC), to migrate event-triggered real-time systems to a time-triggered execution environment.

1.3. Outline

At first, we introduce the term real-time systems architecture in Section 2. In the following Section 3 we briefly sketch the abstraction Atomic Basic Block [8] that is used to capture the white-box view of a real-time system independently of the employed real-time systems architecture. Section 4 presents a description of our system model built on top of ABBs. In Section 5 we revisit the design of the RTSC and describe the transformation used to migrate an event-triggered to a time-triggered system. Section 6 gives an overview over the prototypical implementation of the RTSC and Section 7 presents an evaluation of our prototype. Section 8 discusses the challenges related to migrating time-triggered systems to event-triggered systems as well as distributed systems and multi-core systems. The subsequent Section 9 presents related work, before Section 10 finally concludes this paper.

2. REAL-TIME SYSTEMS ARCHITECTURES

Real-time computing systems are embedded into and interacting with a physical environment. Together, the computing system and the physical environment form the actual real-time system [13]. In the remainder of this paper, we will use the terms real-time system and real-time computing system synonymously, if we address the physical environment of a real-time system we will state this explicitly.

The physical environment not only imposes functional restrictions on a real-time system but also temporal ones. It generates stimuli and expects the real-time system to react to them within a certain

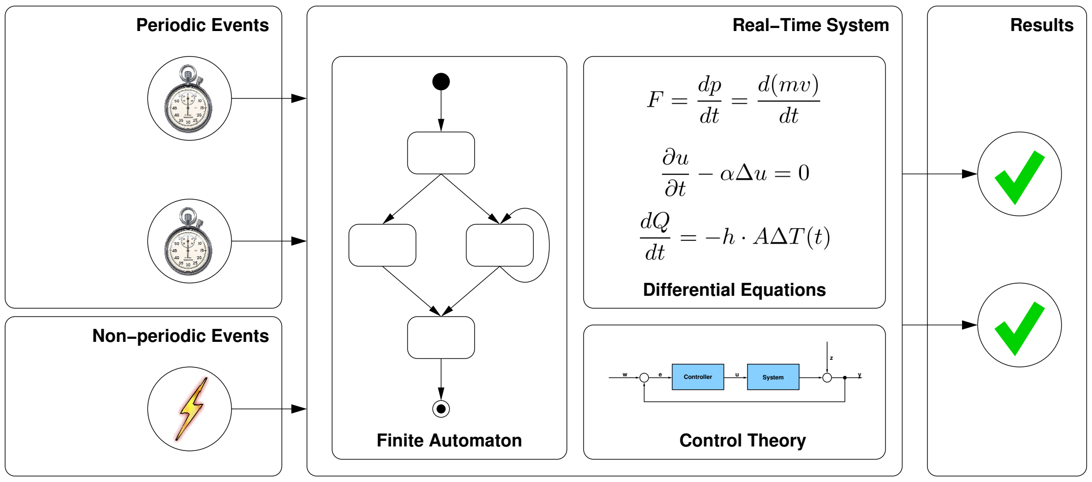


Figure 1. Black-box view of a real-time system

time span. We refer to that interaction between the physical environment and the real-time system as the *black-box view* of the real-time system.

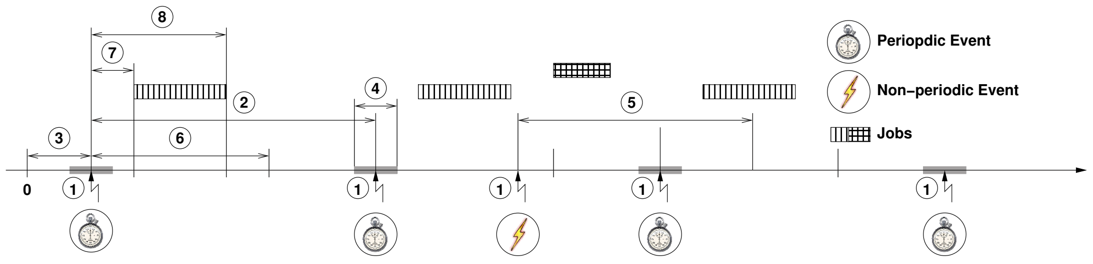
Inside the real-time system event-handlers are activated in response to these stimuli and simultaneous event-handlers potentially have to be coordinated. We refer to this internal structure built by dependencies among different event-handlers as the *white-box view* of the real-time system.

The structural properties of the white-box view are determined by the employed real-time systems architecture. It provides mechanisms to attach event-handlers to events and to implement dependencies between different event-handlers. Thereby, it significantly influences the structure of the white-box view. So, ABBs and the system model must hide exactly these properties while preserving the dependencies among event-handlers and the relation between their representation in the white-box view and the temporal requirements given in the black-box view.

2.1. Black-Box View

The black-box view of a real-time system describing the interaction of the real-time computing system and its physical environment is schematically illustrated in Figure 1. The physical environment generates stimuli and the real-time system reacts upon them computing results. There are various means to describe the functional relation between the inputs sampled in response to the stimuli and the computed results; finite automatons, control theory and differential equations are just a few examples. However, we are only interested in the temporal properties of those stimuli and their related results, so these methods are beyond the scope of this paper and are not covered here.

The temporal dependencies between stimuli and results are sketched in Figure 2. The stimuli generated by the environment are called *events*. According to the temporal specification of these events we can distinguish *periodic events* and *non-periodic events*. Periodic events are characterized by their *period*, a *phase* specifying the temporal offset in comparison to other events and a *jitter*. For non-periodic events only a *minimum interarrival time* is known. Every time an event arrives, a *task* handling that event is triggered and a *job* of that task is released. The occurrence of the event is also referred to as the *release time* of the job. The time interval between the release time of the job and the actual provision of the results is called *response time*. For real-time systems this response time has to be bounded by a *deadline* that could be *hard*, *firm* or *soft* depending on the impact of its miss. Missing a hard deadline could lead to a complete system failure resulting in catastrophic consequences. Thus, this has to be avoided by all means. The miss of a firm deadline causes the cancellation of the associated job and renders the results computed in the meantime unusable. This is similar for soft deadlines, but the results available so far can still be used. In our work we only consider hard deadlines. These deadlines have to be obeyed by the internal structure of the real-time



1	Release Time	5	Minimal Interarrival Time
2	Period	6	Deadline
3	Phase	7	Latency
4	Jitter	8	Response Time

Figure 2. Temporal properties of the black-box view of a real-time system

system. Thus, the white-box view has to guarantee that jobs are released and scheduled properly so that their deadlines can be met.

2.2. *White-Box View*

The white-box view of a real-time system provides insight on the activities that cannot be observed from outside. This mainly applies to jobs of different tasks that are executed simultaneously and how these jobs themselves and dependencies among them are implemented. A real-time systems architecture provides proper mechanisms to trigger tasks, release and schedule jobs and implement the aforementioned dependencies. These mechanisms have a significant impact on the temporal behaviour of the real-time system and, thus, have to be considered carefully.

Triggering Tasks Releasing a job could be accomplished in an event-triggered or a time-triggered fashion. The event-triggered option releases the job as direct reaction to the event itself. The time-triggered version, by contrast, releases the task at a predefined point in time that does not necessarily coincide with the arrival of the event.

Scheduling A scheduling algorithm decides which job is executed next and, thus, also noticeably affects the latencies and the response times of these jobs. So, such a scheduling service has to provide an upper bound for the response time of each job as long as the system is not overloaded. Usually, this holds for deterministic scheduling algorithms. Furthermore, the white-box view of the real-time system could be influenced by the scheduling service as well. In principle, the knowledge about the behaviour of the scheduling algorithm could be exploited to implement dependencies among different jobs.

Dependencies In the most cases, real-time systems service multiple events and simultaneously execute multiple jobs of different tasks. Ideally, all of these tasks are *simple tasks*. In line with Kopetz [13] such tasks do not contain synchronisation points and do not interact with other tasks. Examples for such tasks are T_1 and T_2 triggered by the events E_1 and E_2 in Figure 3. Note that a simple task still might produce more than one result in response to an event. So there is not necessarily a bijective mapping between events and results in real-time systems consisting of simple tasks only.

In real-world real-time systems, however, there normally are also so called *complex tasks* interacting with other tasks in various ways. These interactions can be divided into *directed dependencies* and *undirected dependencies*.

The predecessors of directed dependencies enable their successors. The complex task T_3 in Figure 3, for example, contains a directed dependency as it forks an independent successor producing another result. The tasks T_4 and T_5 in cooperation produce a single result and also form a directed dependency.

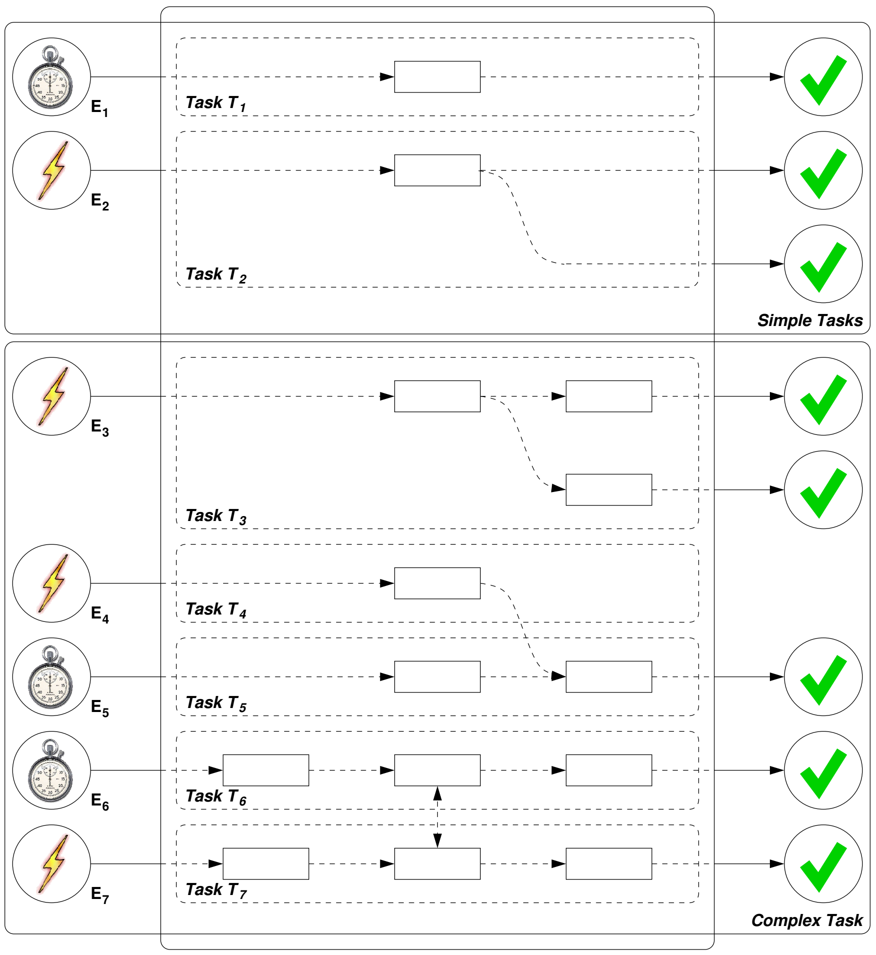


Figure 3. White-box view of a real-time system

Here, through the provision of an intermediate result task T_4 enables its successor T_5. Additionally, such directed dependencies may carry temporal delays giving information about the duration of that dependency. Such temporal delays reflect temporal constraints and have to reproduced as precisely as possible. A typical example is a software-based implementation of pulse-width-modulation (PWM). Here, the signal at an output-pin has to be toggled within certain temporal intervals to produce a given period and duty cycle. As the duty cycle normally is set at run-time the exact time intervals are not known in advance and have to be considered at run-time.

Among undirected dependencies, by contrast, no predecessors or successors can be identified. An undirected dependency denotes the relationship between critical sections concerning the same resource. In Figure 3 the tasks T_6 and T_7 run through critical sections forming an undirected dependency. Besides critical sections also rendezvous-based synchronisation concepts result in undirected dependencies. These dependencies, however, can be mapped to directed dependencies without loss of information which is not the case for critical sections. Therefore, we restrict undirected dependencies to critical sections accordingly.

2.3. *Structural Properties of a Real-Time Systems Architecture*

A real-time systems architecture requires to properly connect the black-box view of a real-time system to its white-box view. As pointed out above, it is necessary to attach event-handlers to events and to provide measures to implement dependencies among simultaneous event-handlers. We assume that the measures listed below are sufficient to accomplish this task. Thus, ABBs and the system model are designed to abstract exactly from the implementation of these measures:

(E1) It must be possible to attach tasks to periodic events. Releasing a corresponding job must be carried out within a bounded delay.

(E2) It must be possible to attach tasks to non-periodic events. Releasing a corresponding job must be carried out within a bounded delay.

(S1) A deterministic scheduling service must be available.

(D1) It must be possible to implement directed dependencies. Directed dependencies are asynchronous on the side of the predecessor and synchronous on the side of the successor. Directed dependencies may furthermore carry an explicit temporal delay.

(D2) It must be possible to implement undirected dependencies. This means, it must be possible to coordinate critical sections properly.

3. ATOMIC BASIC BLOCKS

The main intention of ABBs is to divide the real-time system into small portions of code that are inherently independent of the real-time systems architecture. Furthermore, ABBs should preserve the internal dependencies among event-handlers and should be aware of the temporal requirements of the physical environment. However, ABBs should also be very close to the original implementation fostering the aspired compiler-based transformation tool, because ABBs provide an abstraction playing a key role in the RTSC presented in Section 5.

We decided to use basic blocks widely used in compiler construction as basis to implement ABBs. Basic blocks also inspired the term *Atomic Basic Blocks*. Furthermore, they form graphs describing the control flow of a single event-handler and, thus, are closely coupled to the original implementation. By aggregating one or more basic blocks ABBs segment that CFG and denote fragments of that CFG that contain neither sources nor targets of directed dependencies and also do not cross the boundaries of critical sections.

Therefore, points in the CFG either initiating directed dependencies or entering or leaving critical sections exactly mark the boundaries of ABBs. These points are called *ABB terminations*. Among them we distinguish *joins* that are starting points of inter-function dependencies and *joinpoints* providing targets for these dependencies. Typically, system calls resulting in *joins* are forking threads, setting flags, leaving critical sections or defining global variables. *Joinpoints* are created by waiting for signals, entering critical sections or reading global variables. *Artificial* ABB terminations constitute the last category of ABB terminations. They are used to ensure the proper fragmentation of the CFG, so that the rules given below are met. Within those fragments no interaction with other event-handlers takes place and ABBs can be considered as “atomic” with regard to such interactions. Thus, given a CFG ABBs are extracted from that CFG following these rules:

1. An ABB aggregates one or more connected basic blocks.
2. Every ABB has exactly one distinguished *entry basic block* and at most one distinguished *exit basic block*. Except these basic blocks no other basic blocks have preceding (entry) or succeeding (exit) basic blocks outside that ABB.
3. ABB terminations finalise ABBs and are artificial or mark positions in the CFG that are either origins (*joins*) or targets (*joinpoints*) of inter-function dependencies. Every basic block succeeding a basic block in the CFG that contains an ABB termination marks the beginning of a new ABB.
4. If an ABB termination is located within a basic block the instruction constituting that ABB termination divides it into two subsequent parts. In case of joins and artificial ABB terminations that instruction belongs to the first part, in case of joinpoints the instruction is assigned to the second part.

```
Message *msg;

ISR(SerialByte) {
  unsigned char byte = getByte();
  addTo(msg,byte);

  if(isComplete(msg)) {
    buffer_insert(buffer,msg);
    ActivateTask(MsgHandler);
  }
}
```

```
TASK(MsgHandler) {
  Message *msg = 0;

  msg = buffer_get(buffer);
  handler(currentMsg);

  TerminateTask();
}
```

Figure 4. Receiving messages via to serial wire

If a join or a joinpoint is located within a branch of an alternative statement or within the body of a loop additional ABBs are needed to ensure rule number 2. As there are no appropriate ABB terminations to mark the end of these ABBs, they are closed by artificial ABB terminations.

3.1. *Local, global and undirected ABB-Graphs*

Like basic blocks ABBs form graphs. First of all, ABBs reflect the control flow graphs (CFG) described by basic blocks forming a *local ABB-graph*. Two ABBs $ABB1$ and $ABB2$ are connected by a control flow dependency $ABB1 \rightarrow ABB2$ iff the exit basic block of $ABB1$ is a predecessor of the entry basic block of $ABB2$. The rules stated above also ensure that the local ABB-graph is a coarsening of the CFG formed by basic blocks. Hereby, all analyses and transformations available for CFGs (e.g. dominators and post-dominators [14, chapter 7.3]) can also be applied to local ABB-graphs. So, CFGs can be regarded the formal foundations of ABB-graphs.

Local ABB-graphs are connected via inter-function dependencies and build a *global ABB-graph*. These dependencies are established between matching joins and joinpoints. Joins and a joinpoints match if the underlying operating system calls are compatible (e.g. setting a flag and waiting for it) and apply to related operating system objects (i.e. the same flag). The operating system objects affected by these system calls do not have to be identical, as the operating system could implement a relationship among different objects also resulting in a directed dependency. Additionally, such directed dependencies can be afflicted with a temporal delay.

Finally, critical sections represented by sets of neighbouring ABBs are arranged in an *undirected ABB-graph*. Within that graph there is a dependency between two critical sections if they require access to one or more shared resources.

These ABB-graphs comprise all relevant dependencies of a real-time system and describe its white-box view independently of the employed real-time systems architecture, thus, hiding architecture-specific details. These graphs are finally mapped to the desired real-time systems architecture. Therefore, these ABB-graphs provide an essential abstraction employed by the RTSC to systematically manipulate the real-time systems architecture of a real-time system.

3.2. *Example*

Figure 4 contains an example used to illustrate the concepts introduced above. It contains an interrupt service routine `ISR(SerialByte)` and a thread `TASK(MsgHandler)`. Every time a byte is received at the serial interface `ISR(SerialByte)` is activated and accumulates these bytes in a message. When a message is completed it is stored in a buffer and `TASK(MsgHandler)` is activated to handle that message. The ABB-graph corresponding to that example is depicted in Figure 5. The dashed arrows and boxes show the original basic blocks and the corresponding CFG. The solid arrows and boxes describe the extracted ABBs and the control flow edges mirroring the original CFG. Here, the local ABB-graph obviously mirrors the CFG as every ABB just contains a single basic block.

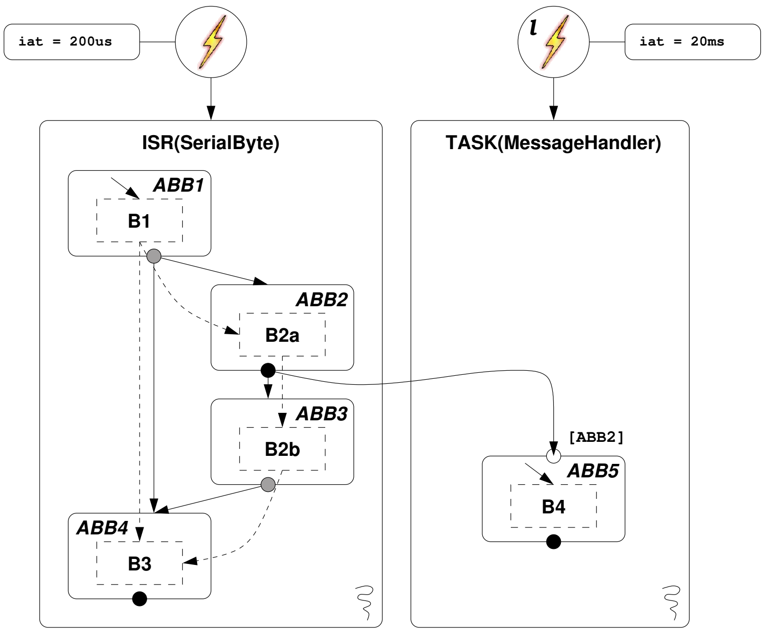


Figure 5. ABB-Graph of the example presented in Figure 4

Joins are illustrated by black circles, white circles mark joinpoints and grey circles denote artificial ABB terminations. By definition thread entry points constitute joinpoints (see $ABB5$) and function return statements are treated as joins (see $ABB4$ and $ABB5$). The join terminating $ABB2$ could easily be mapped to the call of `ActivateTask` in `ISR(SerialByte)`. It also marks the origin of the inter-function dependency $ABB2 \rightarrow ABB5$. $ABB1$ and $ABB3$ are terminated by artificial ABB terminations to ensure rule number 2 as stated above.

4. SYSTEM MODEL

Besides the internal dependencies described the global ABB-graph the temporal properties of the physical environment also provide vital inputs for the RTSC. In the context of time-triggered systems, for instance, it is impossible to compute a static schedule table without that information. In event-triggered systems it is needed to check the feasibility of the generated system. In the RTSC we use a simple system model to capture these properties that are extracted from the black-box view of the real-time system. In this section we describe which elements the system model comprises and how ABB-graphs are correlated to this temporal information.

4.1. *Events*

All activities in a real-time system are initiated by *events*. The main function of events is to carry the temporal properties of tasks. So, such events allow to estimate how often a particular task may arrive. Among events, *periodic* and *non-periodic events* as well as *physical* and *logical events* are further distinguished. The temporal properties of these events are characterised by those parameters already stated in Section 2.1.

Physical events are caused by state changes in the physical environment that are signalled by peripheral hardware components by means of interrupt requests, for instance. Logical events, on the other side, are related to changes in the logical state of the application. Again, this could by exemplified by the program listing shown in Figure 4. The arrival of each byte produces an interrupt; these interrupts are physical events. After a certain amount of bytes has been received the message is complete; this is a logical event. The distinction of physical and logical events enables a much better capturing of the actual temporal properties of an application. Otherwise, one would have to assume that every byte received also entails that activation of the message-handler. However, this is not the case as a single message consists of multiple bytes. Thus, the non-periodic event triggering `TASK(Handler)` has a bigger interarrival time than the non-periodic event triggering `ISR(SerialByte)` fetching those bytes.

4.2. Tasks and Subtasks

All activities that are triggered by events are subsumed under the term *task*. Each task consists of one distinguished *root subtask* and zero or more additional *subtasks*. A subtask encapsulates the actual implementation of the event-handler. The root subtask is executed every time the associated event occurs, the other subtasks are *forked* (see Section 4.3) by the root subtask or one of its successors. Tasks containing multiple subtasks reflect the situation that the same event could trigger different handlers each combined with a different deadline. Each subtask could be assigned a *soft*, a *firm* or a *hard deadline*, indicating its latest possible completion time.

The control flow structure of a subtask is described by a global ABB-graph as presented in Section 3. It starts at the *handler function* implementing that subtask and comprises all functions that are called directly or indirectly by that handler function. Whenever a subtask forks another subtask their global ABB-graphs are connected by a directed inter-function dependency. A task's global ABB-graph is made up by all the interconnected global ABB-graphs belonging to those subtasks being part of that task. As simple tasks are independent of each other their global ABB-graphs are not connected by any directed or undirected dependencies. So the white-box view of a real-time system, in general, is described by forest of ABB-graphs.

As tasks are assigned both, the root of the global ABB-graph and the triggering event, the system model connects the temporal properties of the physical environment denoted by events and the internal structure of the event-handlers as demanded in Section 2.3.

4.3. Triggering and Forking Subtasks

Forking and *triggering* subtasks relate to special directed dependencies targeting the root node of a subtask in the global ABB-graph. A forked subtask is immediately ready for execution. As a consequence, the temporal properties exposed by the forked subtasks are adopted from the forking subtask. In our system model the forked and the forking subtasks always belong to the same task. A triggered subtask, on the other side, does not inherit the temporal properties of its predecessor. These are still described by the event tied to the task enclosing that subtask. Furthermore, in our system model it is only supported to trigger the root subtask of a task. Triggering subtasks is useful when a subtask handling a physical event results in a change of the logical state of the system and, thus, in a logical event.

4.4. Example

The global ABB-graph in Figure 5 also serves as an example for a system model. This system model contains two tasks consisting of a single subtask each. As both subtasks are also implemented by a single handler function, we abstain from depicting tasks and subtasks explicitly.

These tasks are triggered by non-periodic events with an interarrival time of 200 μs (`ISR(SerialByte)`, physical event) and 20 ms (`TASK(MessageHandler)`, logical event). The directed dependency $ABB2 \rightarrow ABB5$ implies that `TASK(MessageHandler)` triggers `ISR(SerialByte)`.

5. THE RTSC

In this section we revisit and refine the design of the RTSC [15]. First, we give an overview of the overall design of the RTSC before we have a closer look on its central components and the transformations taking place in these components. The main purpose of these transformations is to

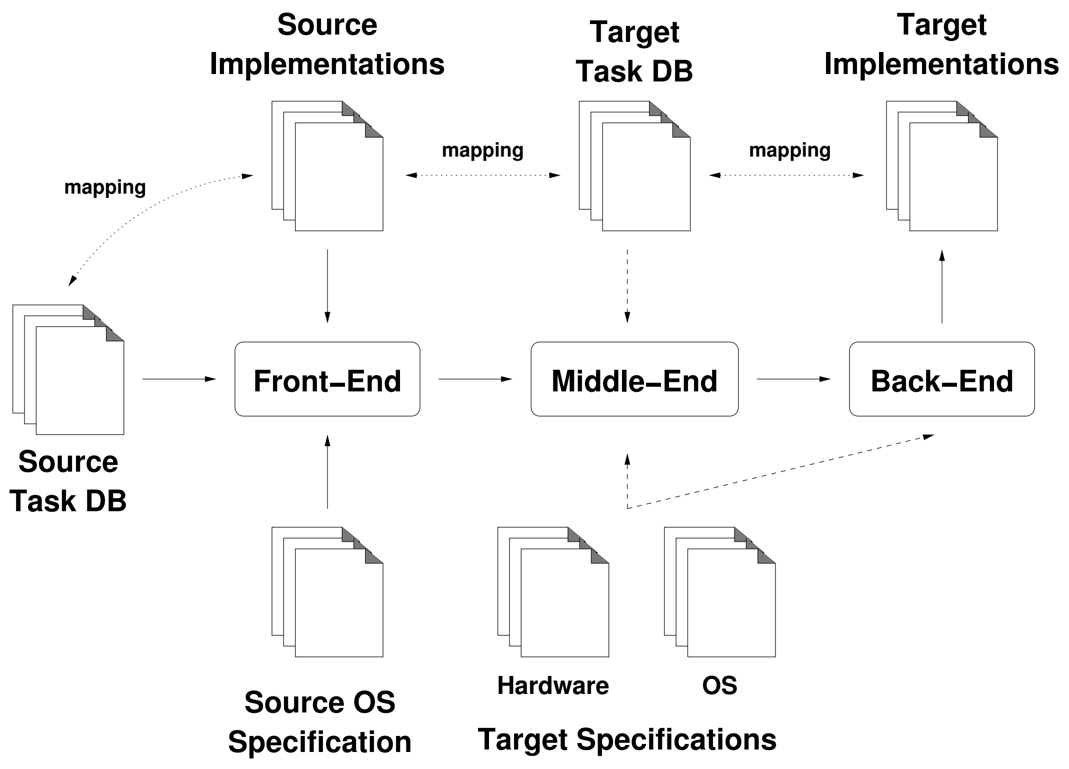


Figure 6. Design of the RTSC

map the global ABB-graph extracted from a given real-time system onto a time-triggered execution environment by means of a static scheduling algorithm.

Although, the RTSC is intended to provide a general operating system (OS) aware compiler tool that aids the generic configuration of the real-time systems architecture, the RTSC currently only supports the migration of event-triggered systems to their time-triggered counterparts. In this paper we, therefore, focus on those steps that are necessary to perform this transition.

5.1. *Overview*

Figure 6 depicts the conceptual design of the RTSC. The main input of the RTSC is a real-time system (*Source Implementations*) using a real-time systems architecture specified by the *Source OS Specification*. The RTSC outputs a modified real-time system (*Target Implementations*) that uses a different real-time systems architecture specified by the *Target OS Specification*.

First, a *front-end* extracts the global ABB-graphs from the implementation of the real-time system. Thereby, it creates an architecture-independent representation of the real-time system's white-box view. In the next step, these ABB-graphs are handed over to the *middle-end*. Here, the global ABB-graph is prepared for the static scheduling algorithm that maps the internal structure of the real-time system to a schedule table. Finally, an OS-dependent *back-end* emits the implementation of the transformed real-time system and also creates the required configuration data for the targeted real-time systems architecture.

The black-box view of the real-time system is specified by *Task Databases* (Task DBs) containing system models as presented in Section 4. There are different Task DBs for the consumed source real-time system (*Source Task DB*) and for the produced target real-time system (*Target Task DB*). It is obvious that the Source Task DB has to be created manually. Deep insight and technical expertise is necessary to precisely extract the temporal properties of the given real-time system. Though, it might be feasible to automatically derive the Target Task DB from the Source Task DB. However, we intentionally refrain from that possibility and the Target Task DB has to be crafted by hand, too.

Within the Target Task DB all non-periodic events of the Source Task DB have to be mapped to corresponding polling periodic events. We do not believe that this could be accomplished by naively applying Nyquist's and Shannon's theorem [16] in general. Furthermore, this manual approach also opens the possibility to fine-tune the temporal parameters for the time-triggered system within certain limits.

5.2. *Front-End*

In order to extract an architecture-independent global ABB-graph the functions marked as subtask-handlers in the Source Task DB are identified by means of their names in the first step. After that, the implementation is scanned for ABB terminations and local ABB-graphs are constructed. These local ABB-graphs are then connected by inter-function dependencies. Therefore, we search for matching pairs of joins and joinpoints and create dependencies between the corresponding ABBs.

These dependencies, however, still are OS dependent, because they carry the explicit semantics of the original system call. Thus, we obtain an OS-independent representation by lowering them to generic directed dependencies. The original semantics of the system call are reflected by logical expressions that are used to *guard* a joinpoint. This expression specifies the set of preceding joins that have to be finished so that the succeeding joinpoint can be executed. In Figure 5, for instance, $ABB5$ is guarded by the boolean expression `[ABB2]` indicating that $ABB2$ has to be executed before $ABB5$. In the last step, we remove the original OS calls from the implementation and gain an OS-independent representation of the source real-time system that still contains all relevant dependencies between its tasks and subtasks.

Our front-end still suffers some restrictions and, thus, implicitly relies on some assumptions. Currently, a flow- and path-insensitive analysis is performed to obtain the global dependency graph. This, of course, limits the use of OS functions within library routines to a certain extent. In order to find inter-function dependencies we precisely need to know which OS object is manipulated at a certain line of code. This information, however, is not necessarily available within library routines as the concerned OS object might be passed to it as a parameter at run-time. Though, we are confident that this problem could be mitigated by a more sophisticated analysis as presented by Mohan [17].

Furthermore we expect the application itself and the OS API to be "well-formed". This mainly refers to a non-ambiguous usage of OS objects and OS functions. On the application level, for instance, we assume that the same instance of a semaphore is not reused for a different purpose at a different location. On the OS level we assume that the same system call is not used for a different purpose. This implies that the semantics of a system call are determined by the system call itself. As an example, we require that semaphores are not used for both, unilateral and multilateral synchronisation. In that case one could only rely on statistical methods to figure out which type of synchronisation actually applies and it is not possible to reliably extract a global dependency graph [18].

These restrictions and assumptions, however, either do not go beyond those that are also present in similar approaches or are mostly technical and could be relaxed by a more sophisticated implementation.

5.3. *Middle-End*

The middle-end's objective is to map the global ABB-graph provided by the front-end to a time-triggered execution environment that solely supports non-preemptive threads using the static scheduling algorithm presented by Abdelzaher and Shin [19]. However, the global ABB-graph has to be pre-processed first, because it contains some patterns that are not handled by standard static scheduling algorithms. This section gives an overview over the transformations performed by the middle-end to eliminate those patterns.

The first step that is not explicitly covered here connects the global ABB-graph to the temporal properties described by the Target Task DB. This is necessary, because the generated static schedule also has to reflect these temporal requirements instead of those noted down in the Source Task DB.

5.3.1. Creating private ABB-graphs In a first step we create a "private" ABB-graph for each task in the system. This step is necessary as the employed scheduling algorithm cannot deal with elements that are activated by a multitude of predecessors. This algorithm rather assumes that all predecessors have to be finished before a common successor could be executed. This pattern typically occurs if a function is called by different callers or if subtasks are forked more than once.

We address that problem by creating "private" copies of the global ABB-graphs representing the control flow structure of a subtask. Starting at the subtask's handler function, we recursively inspect both the local ABB-graphs of all functions that are called within that subtask and the global ABB-graphs of all subtasks that are forked by that subtask. If the function is called more than once or a subtask is forked multiple times we clone that corresponding local or global ABB-graph. Otherwise, we just assign that ABB-graph to the calling or forking subtask. Note that we solely clone the ABB-graphs, not the actual functions. Also note that trigger dependencies are not covered here. Triggered tasks are enabled by a preceding subtask but they are still activated by the assigned event. So, trigger dependencies do not produce the aforementioned pattern and, thus, are handled by another transformation described in Section 5.3.3.

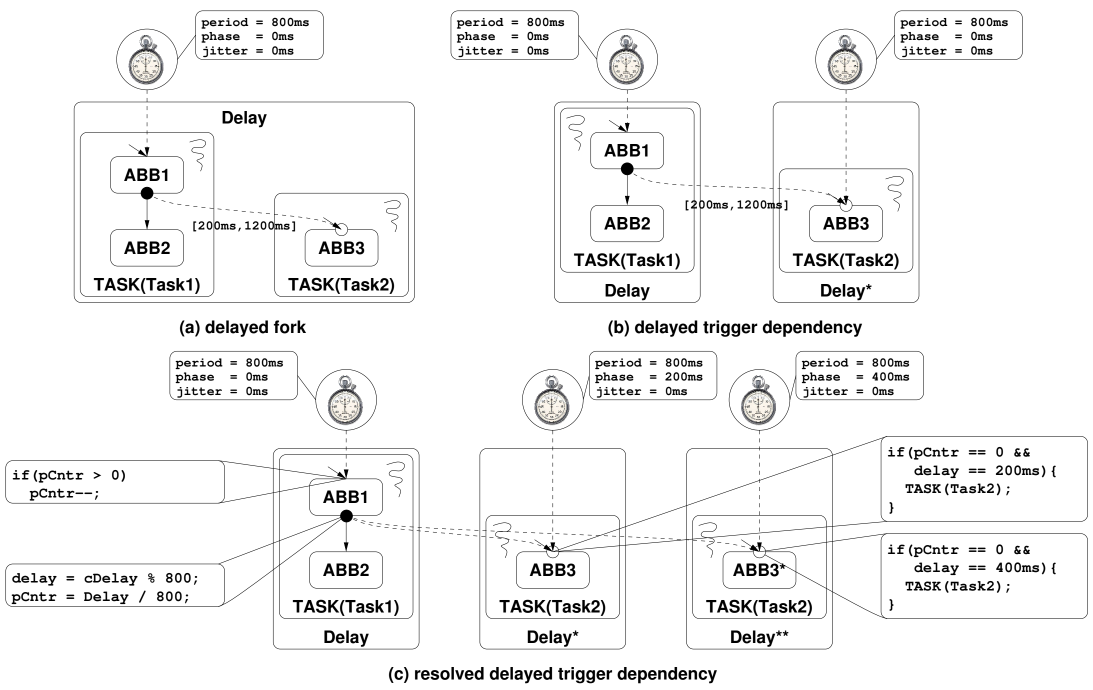
5.3.2. Temporal Delays In Section 4 it was mentioned that directed dependencies may carry additional temporal delays. The durations of these delays usually vary at run-time and have to be reproduced as precisely as possible. As the algorithm by Abdelzaher and Shin already accounts for message passing latencies, it might be worthwhile to treat these temporal delays that way. Message passing latencies, however, only provide a constant upper bound for such delays that could be determined ahead of runtime. Temporal delays, on the other hand, usually cannot be characterised by constant values. Thus, these dependencies have to be eliminated by the RTSC before scheduling the global ABB-graph. Although, the actual delay could be computed at run-time, we demand that it could be specified either as constant value or a list of constant values enumerating all possible delay values.

The RTSC currently supports three different categories of temporal delays. These are *delayed forks*, *delayed trigger dependencies* and *self-trigger dependencies* that are special versions of delayed triggered dependencies. Delayed forks are first mapped to delayed trigger dependencies that are reduced to trigger dependencies handled by the transformation described in Section 5.3.3.

Figure 7 (a) depicts the task *Delay* containing two subtask `TASK(Task1)` and `TASK(Task2)`. Here, `TASK(Task1)` forks `TASK(Task2)` with a delay of either 200 ms or 1200 ms. In the first step, shown in Figure 7 (b), `TASK(Task2)` is removed from the task *Delay* and transferred to the task *Delay**, thus, converting the delayed fork into a delayed trigger dependency. The newly created task *Delay** is triggered by an event identical to that triggering *Delay*.

The delayed trigger dependency finally is resolved by an according phase shift of the triggered task and a period counter. For each possible delay value the RTSC determines a phase shift $phase = delay \bmod period$ and for all resulting phase shifts it creates one copy of the task *Delay**. In Figure 7 (c) this results in two tasks *Delay** and *Delay*** triggered by events with a phase of 200 ms and 400 ms. If the delay exceeds the period of the triggering task a simple phase shift is not sufficient and a period counter explicitly keeping track of the delay is needed. In Figure 7 (c) this period counter is implemented by a global variable `pCntr` introduced by the RTSC. The RTSC also inserts code to initialise it when `TASK(Task1)` triggers `TASK(Task2)` and to decrement it every time `TASK(Task1)` is released. Subtask `TASK(Task2)` only is executed if `pCntr` equals 0 and the delay matches the phase shift. This is achieved by a wrapper function that checks these conditions and that is also depicted within Figure 7 (c). The actual execution of `TASK(Task2)` within that wrapper is adumbrated by a direct call of the subtask.

Self-trigger dependencies are similar to delayed trigger dependencies but the predecessor and the successor belong to the same task. Hence, a phase shift is not really helpful here. Instead, we solely rely on a period counter to reproduce the desired delay. Assuming task *Delay* was a self-triggering task, it would not possible to reproduce the delay of 1200 ms by a period counter and a period of 800 ms. This only would be possible if all delays could be expressed as a product of the period counter and the period of the self-triggering task. We can achieve this by setting the task's period to the greatest



```
period = 800ms
phase  = 0ms
jitter = 0ms
```

```
period = 800ms
phase  = 200ms
jitter = 0ms
```

```
period = 800ms
phase  = 400ms
jitter = 0ms
```

```
if(pCntr > 0)
  pCntr--;
```

```
delay = cDelay % 800;
pCntr = Delay / 800;
```

```
if(pCntr == 0 &&
   delay == 200ms){
  TASK(Task2);
}
```

```
if(pCntr == 0 &&
   delay == 400ms){
  TASK(Task2);
}
```

Figure 7. Resolving temporal dependencies

common divisor of the original period and all possible delay values. In the example given above the new period would be computed as follows: $period = \gcd(800ms, 400ms, 1200ms) = 400ms$. Initialising and decrementing the period counter is carried out identically to delayed trigger dependencies. Though, the wrapper function does not have to check the phase shift, of course, it is sufficient if the period counter equals 0.

5.3.3. Triggering Tasks Here, we take care of trigger dependencies introduced in Section 4. If a task is triggered by another task, it basically maintains the temporal properties specified by its associated event. Nevertheless, the triggering predecessor needs to be completed before the triggered task can be executed. Hence, in a time-triggered execution environment, the predecessor indicates its completion by setting a flag. The triggered task is executed when the flag is set or aborts otherwise.

Here, the RTSC also inserts clones of tasks handling logical events that are triggered by the handler of their related physical event. This is required if the temporal distance between the physical and the logical event is too big and impedes a timely handling of the logical event. By inserting additional logical events this temporal distance can be eliminated to enable a timely execution of the assigned task.

5.3.4. Spanning the Hyperperiod Creating a static schedule often is equivalent to ordering a directed acyclic graph (DAG) with respect to temporal constraints. The linearised DAG then is executed cyclically. As a consequence the DAG comprises all activities of the hyperperiod of the given real-time system. Hence, the RTSC expands the real-time system described in Target Task DB accordingly. This transformation step could be further refined to the following sub-steps:

1. Convert the guards into disjunctive normal form (DNF)
2. Compute the hyperperiod
3. Clone tasks as needed
4. Re-match joins and joinpoints

The first step is just a preparation step for matching joins and joinpoints later on. Here, we compute the DNF for every logical expression guarding a joinpoint that is present in the system. The second

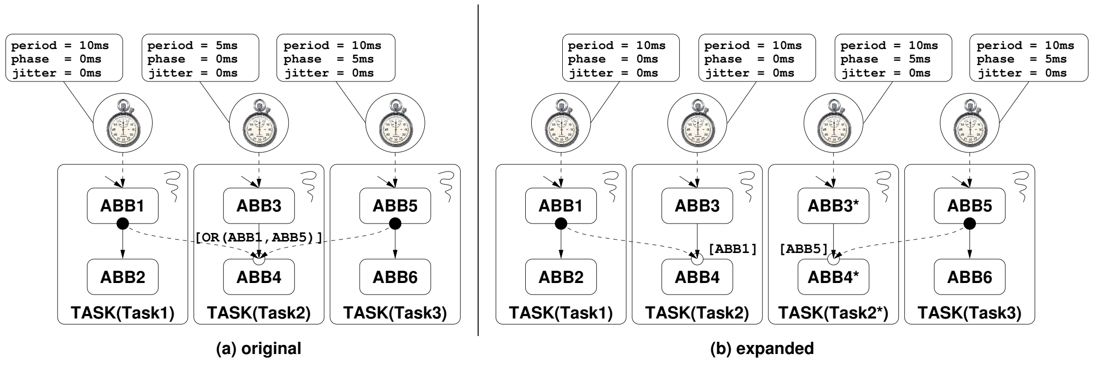


Figure 8. Re-matching Joins and Joinpoints

step computes the hyperperiod hp. This is the least common multiple of all periods p_i of all the events E_i in the Target Task DB: $hp = \operatorname{lcm}_i p_i$.

After that, the actual expansion of the application is carried out. The global ABB-graph of every task whose associated event in the Target Task DB has a period that is smaller than the hyperperiod is cloned as often as needed. The cloned tasks are then triggered by events E_i whose period $p_{i,new}$ is set to the hyperperiod hp and their phase $ph_{i,new}$ is set to the sum of their original phase $ph_{i,old}$ and a multiple of their original period $p_{i,old}$: $ph_{i,new} = ph_{i,old} + kp_{i,old}; \quad k = 0...\lfloor hp/p_{i,old} \rfloor$. After this step, all events of the system have the same period – the hyperperiod – and their original period is mapped to an appropriate phase.

When cloning tasks it is possible that also joins and joinpoints are duplicated. Such a situation is depicted in Figure 8 (a). Here, the task containing the joinpoint $ABB4$ is released with a period of 5 ms and, thus, is executed twice as often as the tasks containing the joins $ABB1$ and $ABB5$ that are triggered every 10 ms. Furthermore, $ABB4$ is guarded by the disjunction `[OR(ABB1,ABB5)]` that could not be handled by the envisaged scheduling algorithm. So, naively duplicating that joinpoint and the incoming dependencies is not very helpful. Instead, we eliminate such guards by re-matching joins and joinpoints.

So, we rip up all dependencies connecting joins and joinpoints of different tasks before cloning the global ABB-graphs. We then try to find a set of joins for each joinpoint that satisfies the guard of that joinpoint. As the guards are given in DNF this is rather easy because it is sufficient to satisfy a single clause of the guard. At this, we assume that each join and joinpoint can be replaced by one of its clones and that each join could be used to satisfy at most one guard. In the end the RTSC chooses that set of joins that permits the earliest possible start time of the joinpoint and minimises the temporal distance between these joins and the joinpoint. Here, the temporal distance is computed using the temporal properties stored within the Target Task DB.

As this search is realised by a relatively simple list-scheduling-like heuristic, it is by no means guaranteed that a suitable matching is found if such a matching exists. Additionally, more sophisticated matching algorithms could be used. An algorithm targeting a similar problem that is based on linear programming is presented by Cuny [20], for instance. In our previous experiments, however, the heuristic we implemented did not impose serious restrictions on the RTSC. So, we abstained from implementing such algorithms for now.

Figure 8 (b) shows the expanded version of Figure 8 (a). Here, the subtask `TASK(Task2)` is duplicated because it is executed twice as often as `TASK(Task1)` and `TASK(Task3)` resulting in the clone `TASK(Task2*)`. Along with that subtask the joinpoint $ABB4$ is cloned, too, producing another joinpoint $ABB4*$. The disjunction `[OR(ABB1,ABB5)]` has been resolved by assigning the join $ABB1$ to the joinpoint $ABB4$ and $ABB5$ to $ABB4*$. This is only possible because the joinpoints $ABB4$ and $ABB4*$ are semantically equivalent and in each case one join is enough to satisfy the original guard.

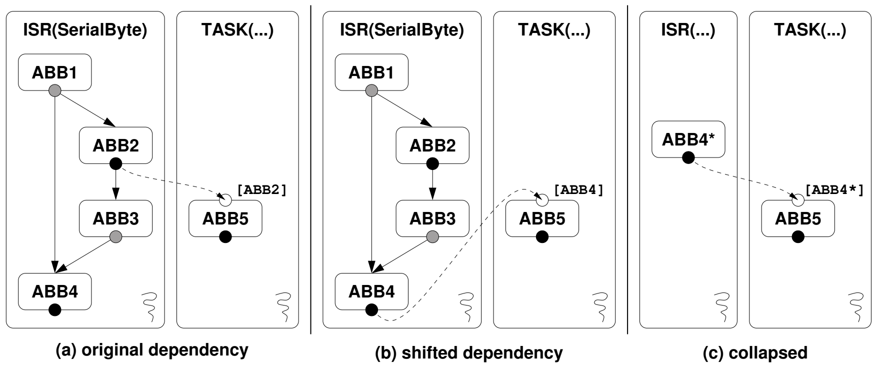


Figure 9. Serialising the ABB-graph

5.3.5. Serialising the ABB-graph Temporally ordering ABBs belonging to alternate branches of if-else-statements is not very helpful because in no case all branches are executed. We try to remove such fragments from the ABB-graph by serialising the ABB-graph as presented by Mohan [17]. This is accomplished by shifting outgoing inter-function dependencies to succeeding ABBs in the ABB-graph and incoming inter-function dependencies to preceding ABBs. When shifting dependencies, of course, we must ensure that we do not create cycles. If we can remove all inter-function dependencies targeting that statement or originating from it we also can collapse all ABBs making up the if-else-statement.

Figure 9 revisits the ABB-graph presented in Figure 4 and exemplifies the effect of serialising the ABB-graph. If the original dependency $ABB2 \rightarrow ABB5$ in Figure 9 (a) could be shifted to the dependency $ABB4 \rightarrow ABB5$ as indicated in Figure 9 (b), then we can also collapse $ABB1$, $ABB2$, $ABB3$, $ABB4$ into $ABB4*$ as shown in Figure 9 (c), thereby, removing that problematic fragment.

In general, we assume that the ABB-graphs are acyclic. So, we neither expect dependencies that reach across the boundary of the hyperperiod nor do we deal with "real cycles" causing deadlocks. However, there still might be patterns that cannot be serialised by naively shifting incoming and outgoing dependencies. Such patterns typically result from alternating producer-consumer interactions [17]. If such patterns should be flattened, either the producer or the consumer would have to be statically split in at least two fragments via some sort of source code transformation. Though, this is not supported by the RTSC, yet.

5.3.6. WCET-Analysis Another necessary input for static scheduling is the WCET of the elements to be scheduled. For this purpose we integrated an automated WCET-analysis in the RTSC. Current WCET-analysis methods consist of two parts: a high-level analysis and a low-level analysis. The high-level analysis examines the CFG of a given function and derives a maximum flow problem that is usually solved via linear programming [21]. The WCET of the basic blocks that serve as input for the high-level analysis are determined by the low-level analysis. The low-level part examines the machine code and performs an extensive hardware-dependent analysis to calculate the WCET of the given basic block.

We implemented the high-level analysis within the RTSC while we use an external tool for the low-level part. For additional input parameters needed for the high-level analysis that cannot be determined from the source code directly (e.g. loop counts or recursion depths), we rely on annotations.

5.3.7. Scheduling The last step in the middle-end is the computation of a static schedule that arranges all ABBs in an absolute temporal order. Successfully scheduling an ABB-graph demands the proper handling of directed order dependencies, undirected mutual exclusion dependencies, as well as release times and deadlines. Hence, we chose the algorithm by Abdelzaher and Shin [19] as it meets all our requirements. Furthermore, it offers the possibility to target multi-processor systems and

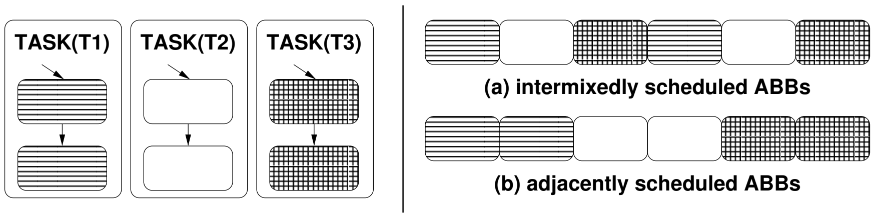

TASK(T1)
TASK(T2)
TASK(T3)
(a) intermixedly scheduled ABBs
(b) adjacently scheduled ABBs

Figure 10. Scheduling ABBs

allows to incorporate an additional message scheduling algorithm later on. Albeit, the RTSC does currently not take advantage of these features.

Whereas, the order and mutual exclusion dependencies needed as input can easily be extracted from the ABB-graph while the release times and deadlines are stored in the Target Task DB. ABBs significantly differ from "modules" that constitute the original unit of scheduling used by the aforementioned algorithm. These modules implicitly enclose complete functions. So, a function is never scattered over several modules. By contrast, a single function could easily comprise multiple ABBs. If all those ABBs were handled equivalently that would lead to sequences of ABBs where ABBs of different functions are intermixed without caution (see Figure 10 (a)). This is caused by the EDF-algorithm that is used inside the algorithm of Abdelzaher and Shin. EDF schedules ABBs according to their deadlines that are normally specified on a per task base. The deadline t_i for ABB_i then is determined recursively with respect to the deadlines t_j of its successors ABB_j and their WCET e_j: $t_i = \min_j(t_j - e_j)$. So, ABBs belonging to the same function may be allotted different deadlines and, thus, also different priorities favouring a intermixture of ABBs belonging to various functions.

As this intermixture also causes non-local jumps between different functions inducing significant runtime overhead this is not desired. For this reason, we try to schedule ABBs belonging to the same function adjacently (see Figure 10 (b)). This is achieved by a slight modification of the EDF-algorithm. All ABBs belonging to the same subtask are assigned the same deadline. As the RTSC explicitly implements order dependencies, it is not necessary to rely on adapted release times and deadlines to ensure those precedence constraints. Additionally, we break ties in the EDF-algorithm in favour of that ABB that belongs to the same function like the ABB that was scheduled at last.

Along with the construction of the schedule table it is also guaranteed that the temporal requirements denoted in the Target Task DB are met. If it is not possible to satisfy these requirements the construction of the schedule table would fail. Thus, it is not necessary to explicitly analyze the timeliness of the generated system.

5.4. Back-End

The task of the back-end is to generate code that can be executed by the targeted RTOS. In the specific case, we generate configuration data and an application skeleton for the targeted operating system. The application skeleton just calls the event-handlers at predefined points in time that are specified by the previously computed static schedule. The original application (i.e. all the implemented event-handlers and libraries) is directly emitted as assembly code for the target system.

6. IMPLEMENTATION

The RTSC is implemented by a set of passes for the LLVM compiler framework [22]. All our transformations work on the virtual instruction set used as intermediate representation within the LLVM that also serves as basis for ABBs. Currently, we use the GCC-based front-end of the LLVM to compile existing real-time applications written in the C programming language into the LLVM representation. We also make use of standard transformations and analyses provided by the LLVM for standard compiler optimisations and the code generation framework to emit assembly code for the target processor.

While we implemented the high-level part of the WCET-analysis on our own, we rely on an external tool for the low-level part. Therefore, we integrated the static WCET-analysis tool aiT[*] into the RTSC. We use it to compute the WCET for every single basic block in the system. Unfortunately, the implemented WCET-analysis is rather pessimistic. That is mainly owed to the following two reasons: Firstly, the hardware-dependent low-level analysis is performed in isolation for each basic block. So, the state of the processor pipeline as left by the preceding basic block is lost leading to a higher estimated WCET. Secondly, we implemented a basic path- and flow-insensitive high-level analysis only. Especially in case of nested loops, the path- and flow-sensitive analysis of the aiT yields much better results, as the bounds of the inner loop often depend on the current state within the outer loop. So far, these shortcomings did not constitute serious restrictions for the RTSC, but we plan to mitigate them by analysing larger pieces of code directly via the aiT.

The RTSC currently encompasses a front-end for an artificial event-triggered OS API and major parts of the OSEK OS API [10]. The artificial API comprises services for forking and triggering subtasks, lock variables, message passing and flags. While working on the RTSC, we first implemented this artificial API and ported the front-end of the RTSC to the API of OSEK OS later on. As the expressiveness of these APIs is very similar porting the front-end was relatively easy and straightforward. On the back-end side, we currently support time-triggered systems only. Here, we target operating systems that comply with OSEKtime specification [11]. Currently we target the TriCore processor [23] by Infineon, as we already have extensive experience using it and there are several RTOS available for that processor. Last but not least, the WCET-analysis tool aiT is available for exactly this processor. As the LLVM did initially not support it we also implemented a LLVM back-end for the TriCore processor.

7. EVALUATION

We have evaluated our the RTSC tool with a real-time system that mimics a highstriker, the well-known attraction on fairs. This system constitutes a fairly challenging evaluation scenario as it comprises non-periodic events only and, thus, could serve as prime example for an event-triggered system. The control application also was developed in an event-triggered fashion and we transformed it into an equivalent time-triggered system. In our quantitative evaluation we mainly paid attention to temporal parameters that are relevant for a successful operation of this system. For a better understanding of the observed parameters we first give a more elaborate explanation of the evaluation scenario and then take a look at the measured parameters.

7.1. Evaluation Scenario

Our highstriker is equipped with a Plexiglas tube that houses an iron projectile. The projectile is controlled by switching coils that are attached to the tube in an equispaced manner. Hereby, the projectile can be guided almost arbitrarily between the different coils. Right above each coil there is a light barrier that is used to track the position and the direction of the projectile.

The control application itself is state machine-based. A step in the state machine is either triggered by the projectile leaving or entering a light barrier or a countdown indicating that the next step must be taken. The countdown is set up within a step of the state machine itself, as some actions require the next step to be carried out within a certain temporal distance. All these occasions are subsumed by the logical event *SMStep*. If several consecutive steps in the state machine are necessary, all these steps are executed in a row within a loop. So, the task handling *SMStep* is a self-triggering task as described in Section 5.3.3. The task is also triggered by the task handling the physical *LightBarrier* event. As it is not necessary to perform a state machine step every time a light barrier is entered or left. So, the task *SMStep* is triggered but not forked by the task *LightBarrier*.

[*]http://www.absint.com/ait

Event	Interarrival time	Task	Deadline
LightBarrier	12.8 ms	`CSYSTEM_P_ISR`	1ms
SMStep	37 ms	`CSYSTEM_FSM_Task`	1ms

Table I. Source Task DB

Event	Period	Task	Deadline
LightBarrier	500 μs	`CSYSTEM_P_ISR`	500 μs
SMStep	37 ms	`CSYSTEM_FSM_Task`	500 μs

Table II. Target Task DB

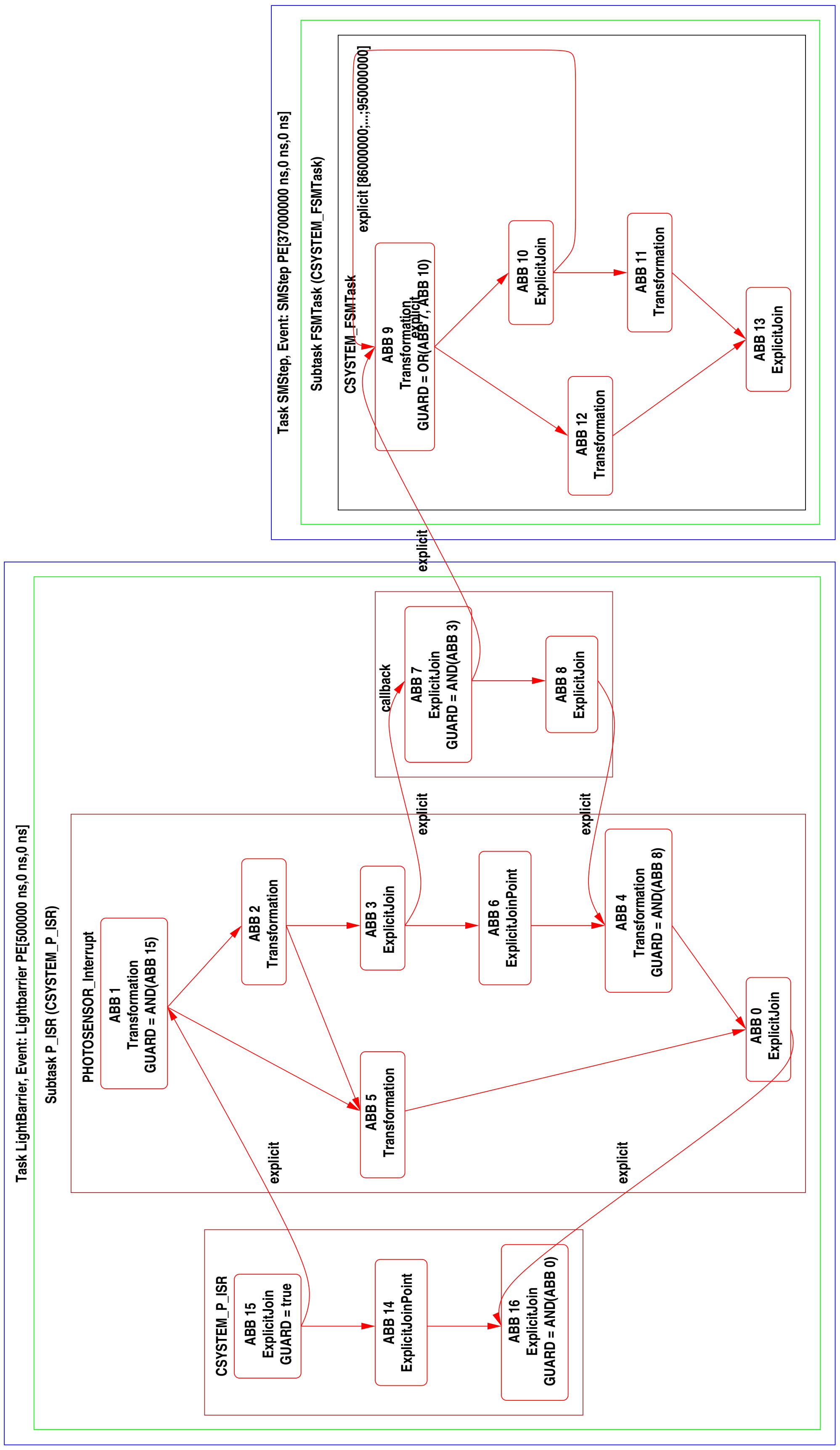
Both events are obviously non-periodic. The minimum interarrival times are achieved when the projectile crosses two subsequent light barriers at its maximum speed of roughly 6.14 m/s. The maximum speed is achieved when the projectile freely falls down from the top of the Plexiglas tube and is continuously accelerated by gravity. The length of the projectile of 8.2 cm leads to a minimum interarrival time of about 12.8 ms for the physical event *LightBarrier* and the distance of 23 cm between two light barriers leads to an minimum interarrival time of 37 ms for the event *SMStep*. The physical event *LightBarrier* is handled by the subtask `CSYSTEM_P_ISR` and the logical event *SMStep* by the subtask `CSYSTEM_FSM_Task`. The deadlines of 1 ms are empirical values we gained from the operation of our highstriker experiment. The deadline of the task `CSYSTEM_FSM_Task` handling the logical event *SMStep* refers to the physical event *LightBarrier*. This information is summarised in the Source Task DB that is depicted in Table I.

The Target Task DB that is derived from the Source Task DB manually is shown in Table II. Here, the non-periodic events of the Source Target Task DB are converted into polling periodical events. According to Briand [24] such events have to be polled cyclically with a period that is at most half as long as the deadline of the corresponding handlers. So, the periods of these events and the corresponding deadlines are adjusted to a value of 500 μs. Note that the period of the event *SMStep* intentionally is not set to 500 μs to demonstrate treatment of self-triggering task within the RTSC. As *SMStep* is triggering itself the RTSC will compute a suitable period so that the temporal delays belonging to that self-trigger dependencies could be reproduced. Both events are handled by the same subtask that is also given in the Source Task DB.

7.2. *Evaluation Results*

The structure of the control application is illustrated by the global ABB-graph presented in Figure 11 that is extracted by the RTSC. Both tasks *LightBarrier* and *SMStep* can easily be identified. The ABB-graph of subtask `CSYSTEM_P_ISR` all in all comprises three functions while the ABB-graph of subtask `CSYSTEM_FSM_Task` consist of a single function, only. The complete implementation of the highstriker encompasses numerous functions, of course. As these functions do not contribute to that global ABB-graph, they are not shown here. Furthermore, the dependencies triggering ($ABB7 \rightarrow ABB9$) and self-triggering ($ABB10 \rightarrow ABB9$) the task *SMStep* are clearly visible.

The schedule table generated by the RTSC is depicted in Table III. It unnecessarily spans a hyperperiod of 1 ms, although 500 μs would have been sufficient. We were able to trace this back to a weakness in the implementation of the RTSC when inserting additional logical events as described in Section 5.3.3. The initial offset of 10 μs is owed to a restriction in the targeted RTOS that does not support threads that have to be started right at the beginning of the schedule table. As mentioned above, the RTSC automatically adjusted the period of the task *SMStep* to 500 μs, so the temporal delays belonging to the self-trigger dependencies can be reproduced. Besides the creation of the Target Task DB the transformation did not require any manual intervention. Particularly, there was no need to explicitly convert the interrupt service routine `CSYSTEM_P_ISR` into a polling variant. The original system suffered from bouncing light barriers, thus, `CSYSTEM_P_ISR` already had to filter relevant interrupts anyway.



Task SMStep, Event: SMStep PE[37000000 ns,0 ns,0 ns]
Subtask FSMTask (CSYSTEM_FSMTask)
CSYSTEM_FSMTask
explicit [86000000;...;950000000]
ABB 9 Transformation GUARD = OR(ABB 7, ABB 10)
explicit
ABB 10 ExplicitJoin
ABB 11 Transformation
ABB 12 Transformation
ABB 13 ExplicitJoin
Task LightBarrier, Event: Lightbarrier PE[500000 ns,0 ns,0 ns]
Subtask P_ISR (CSYSTEM_P_ISR)
PHOTOSENSOR_Interrupt
ABB 1 Transformation GUARD = AND(ABB 15)
ABB 2 Transformation
ABB 3 ExplicitJoin
ABB 5 Transformation
ABB 6 ExplicitJoinPoint
ABB 4 Transformation GUARD = AND(ABB 8)
ABB 0 ExplicitJoin
callback
ABB 7 ExplicitJoin GUARD = AND(ABB 3)
ABB 8 ExplicitJoin
explicit
explicit
explicit
explicit
CSYSTEM_P_ISR
ABB 15 ExplicitJoin GUARD = true
ABB 14 ExplicitJoinPoint
ABB 16 ExplicitJoin GUARD = AND(ABB 0)

Figure 11. Black-box view of a real-time system

Start time	Task	WCET
10 μs	CSYSTEM_P_ISR	19,25 μs
30 μs	CSYSTEM_FSM_Task	75,87 μs
510 μs	CSYSTEM_P_ISR	19,25 μs
530 μs	CSYSTEM_FSM_Task	75,87 μs

Table III. Schedule Table generated by the RTSC

	Latency			Response Time		
	min	avg	max	min	avg	max
Source	6 μs	8 μs	10 μs	9 μs	12 μs	18 μs
Target	13 μs	268 μs	507 μs	24 μs	276 μs	518 μs

Table IV. Event-handler latencies and response times

	Source			Target		
Countdown	min	avg	max	min	avg	max
4	3,05	3,63	4,01	4,00	4,00	4,00
9	8,01	8,46	8,98	9,00	9,00	9,00
14	14,00	14,00	14,00	14,00	14,00	14,00
18	17,05	17,50	17,99	18,00	18,00	18,00
86	86,00	86,00	86,00	86,00	86,00	86,00
950	949,07	949,49	950,00	950,00	950,00	950,00

Table V. Countdown accuracy (in milliseconds)

Using that schedule table and the transformed implementation generated by the RTSC, we were able to operate our highstriker experiment the same way we already did it using the original event-triggered implementation. Besides the successful operation of the highstriker, we also performed a quantitative comparison of the generated and the original system regarding the following aspects: the event handler latencies and their response times (Table IV), the accuracy of the countdowns (Table V) and the overall CPU utilisation. [†]

As shown in Table V the generated target system reproduces the temporal delays of the self-trigger dependency very accurately. Although, the delays are already closely followed by the event-triggered source system, the time-triggered system performs even better. However, it is clearly outperformed by the event-triggered source system regarding event-handler latencies, response times and CPU utilisation. The latencies depicted in Table IV range from the entrance of `CSYSTEM_P_ISR` to the entrance of `CSYSTEM_FSM_Task` while the response times last until the termination of `CSYSTEM_FSM_Task`. The maximum latency and the maximum response time in the target system result from the fact that subtask `CSYSTEM_FSM_Task` is scheduled 20 μs behind subtask `CSYSTEM_P_ISR`, summing up to the maximum possible latency of 520 μs. Due to overhead induced by polling non-periodic events the low overall CPU utilisation of only about 0.5% in the original system increased significantly to 4.1% in the generated time-triggered system.

Although, the quantitative comparison seems not to be very promising, we account this transformation a success. Especially the measured response times indicate that the generated real-time system complies to the temporal requirements of the original real-time system, too. Thus, we were able to automatically transform an inherently event-triggered system servicing only non-periodic events into a functional equivalent time-triggered system. It is well known that time-triggered systems

[†]All benchmarks have been performed on an Infineon TriCore TC1796 processor with a CPU clock speed of 100 MHz and a system clock speed of 50 MHz. Data and code were completely located in the internal RAM of the TC1796. The STM-timer of the TC1796 was used for time measurement. The CPU utilisation was determined by directly inspecting the CPU utilisation of an idle function by the Trace32 debugger by Lauterbach.

systematically suffer higher latencies and a higher CPU utilisation when non-periodic events are mapped to polling tasks [24, 25]. So, we think that primarily the time-triggered mode of execution has to be blamed for the overhead observed within the quantitative comparison. Therefore, we are confident that the RTSC yields better results for real-time systems that service a higher fraction of periodic events.

8. DISCUSSION

Although, the main goal of the RTSC is the generic configuration of the real-time systems architecture it currently only supports the migration of event-triggered to time-triggered systems consisting of single computing node. In this section we discuss the challenges that have to be tackled when the RTSC also should migrate time-triggered to event-triggered systems and target distributed and multi-core systems.

8.1. Migrating Time-Triggered to Event-Triggered Systems

Migrating time-triggered to event-triggered systems significantly differs from the transformation that is currently implemented by the RTSC. Mapping the global ABB-graph to a set of possibly concurrent threads, for instance, is exactly the opposite of linearising it as far as possible using a static scheduling algorithm. The main issue, however, is the extraction of the global ABB-graph from a time-triggered system.

In event-triggered systems directed and undirected dependencies among different subtasks are usually implemented explicitly via special system calls provided by the real-time systems architecture. As no such system calls exist in time-triggered systems directed and undirected dependencies are established implicitly using a static schedule table ordering event-handlers appropriately. So, dependencies are no longer visible within the implementation of the real-time system.

Therefore, the only chance to automatically extract the global ABB-graph from a time-triggered system is to thoroughly analyse the accesses to shared memory locations and correlate these data dependencies with the temporal order enforced by the schedule table. However, this is exacerbated as different subtasks may use shared memory locations by accident without being ordered by a directed dependency. Undirected dependencies, on the other hand, cannot be determined for sure as well. While it is feasible to find possible data races, atomicity violations emerging when correlated memory locations are inconsistently updated or read, cannot be detected reliably [26]. So, these analyses only yield hints to possible directed or undirected dependencies but cannot give any evidence for their existence. Thus, we assume that time-triggered systems have to be enriched by annotations to facilitate the extraction of a global ABB-graph.

8.2. Distributed and Multi-Core Systems

Many real-time systems are distributed computing systems and multi-core systems are becoming more and more popular also among real-time embedded systems. So, automatically mapping ABB-based dependency graphs to such systems seems to be very promising. There are already several considerable results in the area of scheduling theory for time-triggered real-time systems dealing with static scheduling and allocation strategies for that problem. Some examples are the algorithms published by Xu [27] and Peng [12] or the algorithm implemented within the RTSC [19]. But also for event-triggered systems similar work has been done [28]. There are, however, some problems that must be solved before such algorithms can be used to distribute a global ABB-based dependency graph onto a distributed system or a multi-core system.

One of these problems already became evident in Section 5.3.7. ABB-graphs are rather fine-grained and ABBs belonging to the same function should also be allocated to the same computing node, as non-local jumps between different cores or nodes have an adverse impact on the overall performance of the system. Consequences of such non-local jumps are for example invalidated code and data caches and also a higher consumption of network bandwidth as local state has to be transferred to a remote node in some cases. So, these allocation strategies have to be adapted to respect the cohesion of such tightly related ABBs.

Another problem is inter-task communication. Tasks may exchange data explicitly via message passing or implicitly via shared data structures for instance. While message passing could easily be identified within the source code this is very challenging for shared data structures. Without knowing the precise semantics of a queue, for instance, it is hard if not impossible to reason that the values dequeued by the consumer are those enqueued by the producer even if producer and consumer are additionally synchronised via directed dependencies. However, if the different tasks of a real-time system should be on the nodes of a distributed system all these “hidden” communication channels must be mapped to explicit message passing.

The situation is somewhat relaxed for multi-core systems as shared memory normally is available to the different cores of multi-core node. So, this kind of communication does not have to be mapped to message passing at all. In the worst case, the application has to be enriched by means of annotations manually to identify all relevant communication channels. Besides that, current industrial standards like the AUTOSAR RTE specification [29] require that communication between different components is based on message passing. So, these communication channels are already explicitly visible.

Furthermore, the topology and the properties of the distributed system must be available to the RTSC. Otherwise it is, for example, impossible to allocate bandwidth on a communication channel. So the system model has to be enriched by appropriate mechanisms to describe the computing nodes and available communication channels connecting them along with their bandwidth and the communication paradigm they provide. Fortunately, there are also several description methods available for this purpose. Examples are SysML [30] or a modelling language presented by Huber to describe execution platforms [31] in DECOS [32].

9. RELATED WORK

The RTSC is not the first attempt to apply compiler techniques to improve and automate the construction and analysis of real-time systems. Program slicing was employed in by Gerber [33] and Gopinath [34] to improve the schedulability of real-time systems. Kirner [35], for instance, enhances WCET-analysis by transforming flow information within optimising compilers. Thereby, loop annotations are also maintained across transformations like loop unrolling. Other approaches are based on domain-specific languages to describe the temporal structure of the black-box view of real-time systems [36, 37, 38]. Special compilers and code generators exploit the temporal information embedded into these descriptions to generate application skeletons that adhere to the specified temporal constraints and to reason about the feasibility of the whole real-time system. However, none of these approaches attempts to extract the white-box view of a real-time system or to systematically manipulate it in order to alter the real-time systems architecture.

There are also efforts the automatically extract relevant parts of the white-box view of a real-time system in an automated fashion. Albers, for example, presents a method to analyse the schedulability of real-time systems based on hierarchical event streams that allow to compute upper bounds for the number of task activations [39]. Hierarchical event streams are created bottom up starting from simple events streams that are directly extracted by a static source code analysis finding those places where a task is activated. Depending if the task is activated within an alternative statement or a loop the simple event streams are then combined accordingly into hierarchical event streams. Mohan and Helander automatically extract global dependency graphs from the source code of real-time systems also reflecting inter-task dependencies [17]. These graphs are used to analyse and visualise unneeded and potential parallelism that can be exploited to scale the application to smaller or bigger computing nodes. In contrast to the RTSC, these approaches neither cover mutual exclusion nor are these graphs used to exchange the underlying real-time systems architecture.

Considerable work has also been done regarding the provision of model-based integrated tools and tool chains that support the development of software for real-time systems. Among the numerous examples are SysWeaver [40], Simulink [41] or TDL [37] or well-known commercial products like TargetLink. Most of these approaches either start from an abstract model-based input or the input already exposes a very similar real-time systems architecture as the target system. Thus, there is no need to abstract from an underlying real-time systems architecture.

Further approaches that are comparable to ours are implemented by Anvil [42] or PORTOS [43]. However, these approaches make assumptions on the targeted real-time systems architecture – PORTOS, for instance, requires blocking communication – and, thus, are not suited to provide a generic tool-based manipulation of the real-time systems architecture.

As the development of multi-threaded concurrent systems has always been an ambitious undertaking there are also a lot of formal techniques to describe such systems and the dependencies among the different threads. Well-known examples for these techniques are CSP [44], path expressions [45], Petri nets [46] or mathematical logic [47, 48, 49]. These techniques are mainly used to prove higher abstract properties such as the absence of deadlocks or starvation. Although, current efforts even extract such descriptions automatically from a given system [18, 26, 50], these techniques are not suited to be used in transformation tool like the RTSC. The main problem is that these description are rather abstract and are not connected closely enough to the implementation of the considered system. This makes it difficult to develop code transformations based on such descriptions. ABBs, in contrast, are directly derived from control flow graphs and, thus, are exactly reflecting the control flow structure of the implementation.

All in all, the authors of this paper are not aware of any other tool or tool-chain that explicitly aids the migration between different real-time systems architectures. Most of these tools assume some kind of abstract, model-based input and, hereby, automatically gain independence of the employed real-time systems architecture. The RTSC, in contrast, works on the much lower level of an existing implementation. So, the RTSC is also able to deal with existing software.

10. CONCLUSION

In this paper we presented the first prototype of the *Real-Time Systems Compiler* (RTSC), a tool that assists in migrating event-triggered to time-triggered systems. The long-term goal of the RTSC is to achieve the generic configuration of the real-time systems architecture. The key idea is to hide the real-time systems architecture used to implement the internal structure – the white-box view – of a real-time system behind an appropriate abstraction and connect that abstraction to the temporal properties of the physical environment of the real-time system – its black-box view. We propose a global dependency graph made up of *Atomic Basic Blocks* as a proper abstraction for that purpose. We furthermore combined this dependency graph with a *system model* to connect the internal structure of the real-time system to the temporal properties of the physical environment. We also demonstrated the applicability of our approach by implementing a transformation process on the basis of ABBs that converts an event-triggered system into a time-triggered one and, thus, exchanges the real-time systems architecture of a given real-time system. Although, the RTSC still is an early prototype, it can already handle complete real-time systems significantly going beyond mere test-cases. Thus, it provides a profound base for further research and a comfortable alternative to ad-hoc techniques that are still widely used in the development of time-triggered real-time systems. Nonetheless, this prototype was just a first step and there are more challenges that need to be tackled.

REFERENCES

1. Carlow GD. Architecture of the space shuttle primary avionics software system. *Communications of the ACM* 1984; **27**(9):926–936, doi:10.1145/358234.358258.
2. Shepard T, Gagne M. A model of the f18 mission computer software for pre-run-time scheduling. *Proceedings of the 10th International Conference on Distributed Computing Systems (ICDCS '90)*, IEEE Computer Society Press: Washington, DC, USA, 1990; 62–69, doi:10.1109/ICDCS.1990.89285.
3. Robert Bosch GmbH. CAN specification version 2.0. *Technical Report* Sep 1991.
4. FlexRay Consortium. *FlexRay protocol specification 2.1 Revision A*. FlexRay Consortium, 2005. `http://www.flexray.com`.
5. Domaratsky Y, Perevozchikov M, Ingulets A, Alkhovik A. Back-end software for highly dependable real-time control systems. *Proceedings of the 25th IEEE Computer Software and Applications Conference (COMPSAC '01)*, IEEE Computer Society Press: Los Alamitos, CA, USA, 2001; 237–244, doi:10.1109/CMPSAC.2001.960622.

6. Poledna S. Replica determinism in distributed real-time systems: a brief survey. *Real-Time Systems Journal* 1994; **6**(3):289–316, doi:10.1007/BF01088629.
7. Scheler F, Schröder-Preikschat W. Synthesising real-time systems from atomic basic blocks. *12th IEEE International Symposium on Real-Time and Embedded Technology and Applications (RTAS'06)*, Work-in-Progress Session, 2006; 49–52.
8. Scheler F, Schröder-Preikschat W. Time-triggered vs. event-triggered: A matter of configuration? *Proceedings of the GI/ITG Workshop on Non-Functional Properties of Embedded Systems*, VDE Verlag GmbH: Nuremberg, Germany, 2006; 107–112.
9. Scheler F, Schröder-Preikschat W. The RTSC: Leveraging the migration from event-triggered to time-triggered systems. *Proceedings of the 13th IEEE International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC '10)*, IEEE Computer Society Press: Washington, DC, USA, 2010; 34–41, doi:10.1109/ISORC.2010.11.
10. OSEK/VDX Group. Operating system specification 2.2.3. *Technical Report*, OSEK/VDX Group Feb 2005. `http://portal.osek-vdx.org/files/pdf/specs/os223.pdf`, visited 2011-08-17.
11. OSEK/VDX Group. Time triggered operating system specification 1.0. *Technical Report*, OSEK/VDX Group Jul 2001. `http://portal.osek-vdx.org/files/pdf/specs/ttos10.pdf`.
12. Peng DT, Shin KG, Abdelzaher TF. Assignment and scheduling communicating periodic tasks in distributed real-time systems. *IEEE Transactions on Software Engineering* 1997; **23**(12):745–758, doi:10.1109/32.637388.
13. Kopetz H. *Real-Time Systems: Design Principles for Distributed Embedded Applications*. Kluwer Academic Publishers, 1997.
14. Muchnick SS. *Advanced compiler design and implementation*. Morgan Kaufmann Publishers Inc.: San Francisco, CA, USA, 1997.
15. Scheler F, Mitzlaff M, Schröder-Preikschat W, Schirmeier H. Towards a real-time systems compiler. *Proceedings of the 5th International Workshop on Intelligent Solutions in Embedded Systems (WISES '07)*, IEEE Computer Society Press: Washington, DC, USA, 2007; 62–75.
16. Shannon C. Communication in the presence of noise. *Proceedings of the IRE* Jan 1949; **37**(1):10–21, doi: 10.1109/JRPROC.1949.232969.
17. Mohan S, Helander J. Temporal analysis for adapting concurrent applications to embedded systems. *Proceedings of the 20th Euromicro Conference on Real-Time Systems (ECRTS '08)*, IEEE Computer Society Press: Washington, DC, USA, 2008; 71–82, doi:10.1109/ECRTS.2008.38.
18. Engler D, Ashcraft K. Racerx: effective, static detection of race conditions and deadlocks. *Proceedings of the 19th ACM Symposium on Operating Systems Principles (SOSP '03)*, ACM Press: New York, NY, USA, 2003; 237–252, doi:10.1145/945445.945468.
19. Abdelzaher TF, Shin KG. Combined task and message scheduling in distributed real-time systems. *IEEE Transactions on Parallel and Distributed Systems* 1999; **10**(11):1179–1191, doi:10.1109/71.809575.
20. Cuny JE, Snyder L. Conversion from data-driven to synchronous execution in loop programs. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 1987; **9**(4):599–617, doi:10.1145/29873.31334.
21. Puschner P. Zeitanalyse von Echtzeitprogrammen. PhD Thesis, Technische Universität Wien, Institut für Technische Informatik, Treitlstr. 1-3/182-1, 1040 Vienna, Austria 1993.
22. Lattner C, Adve V. Llvm: A compilation framework for lifelong program analysis & transformation. *Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO'04)*, IEEE Computer Society Press: Washington, DC, USA, 2004.
23. Infineon Technologies AG, St.-Martin-Str. 53, 81669 München, Germany. *TriCore 1 User's Manual (V1.3.5), Volume 1: Core Architecture* Feb 2005.
24. Briand LC, Roy DM. *Meeting Deadlines in Hard Real-Time Systems: The Rate Monotonic Approach*. IEEE Computer Society Press: Los Alamitos, CA, USA, 1997.
25. Kopetz H. Event-triggered versus time-triggered real-time systems. *Proceedings of the International Workshop on Operating Systems of the 90s and Beyond*, Springer-Verlag: London, UK, 1991; 87–101.
26. Lu S, Tucek J, Qin F, Zhou Y. AVIO: detecting atomicity violations via access interleaving invariants. *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '06)*, ACM Press: New York, NY, USA, 2006; 37–48, doi:10.1145/1168857.1168864.
27. Xu J. Multiprocessor scheduling of processes with release times, deadlines, precedence, and exclusion relations. *IEEE Transactions on Software Engineering* 1993; **19**(2):139–154, doi:10.1109/32.214831.
28. Zheng W, Zhu Q, Natale MD, Vincentelli AS. Definition of task allocation and priority assignment in hard real-time distributed system. *Proceedings of the 28th IEEE International Symposium on Real-Time Systems (RTSS '07)*, IEEE Computer Society Press: Washington, DC, USA, 2007; 161–170, doi:http://doi.ieeecomputersociety.org/10.1109/RTSS.2007.40.
29. AUTOSAR. Specification of RTE software (version 1.0.1). *Technical Report*, Automotive Open System Architecture GbR Jul 2006.
30. Object Management Group OMG. OMG Systems Modeling Language (OMG SysML), version 1.2. formal/2010-06-02 June 2010.
31. Huber B, Obermaisser R. Model-based development of integrated computer systems: Modeling the execution platform. *Proceedings of the 5th International Workshop on Intelligent Solutions in Embedded Systems (WISES '07)*, IEEE Computer Society Press: Washington, DC, USA, 2007; 151–164.
32. Obermaisser R, Peti P, Tagliabo F. An integrated architecture for future car generations. *Real-Time Systems Journal* Jul 2007; **36**:101–133, doi:10.1007/s11241-007-9015-4. URL `http://portal.acm.org/citation.cfm?id=1265364.1265369`.
33. Gerber R, Hong S. Slicing real-time programs for enhanced schedulability. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 1997; **19**(3):525–555, doi:10.1145/256167.256394.
34. Gopinath P, Gupta R. Applying compiler techniques to scheduling in real-time systems. *Proceedings of the 11th International Conference on Real-Time Systems (RTSS '90)*, IEEE Computer Society Press: Washington, DC, USA,

1990; 247–256, doi:10.1109/REAL.1990.128755.
35. Kirner R, Puschner P, Prantl A. Transforming flow information during code optimization for timing analysis. *Real-Time Systems Journal* June 2010; **45**:72–105, doi:10.1007/s11241-010-9091-8.
36. Henzinger TA, Kirsch CM, Marques ER, Sokolova A. Distributed, modular HTL. *Proceedings of the 30th IEEE International Symposium on Real-Time Systems (RTSS '09)*, IEEE Computer Society Press: Washington, DC, USA, 2009.
37. Farcas E, Farcas C, Pree W, Templ J. Transparent distribution of real-time components based on logical execution time. *Proceedings of the 2005 ACM SIGPLAN/SIGBED Conference on Languages, Compilers and Tools for Embedded Systems (LCTES '05)*, ACM Press: New York, NY, USA, 2005; 31–39, doi:10.1145/1065910.1065915.
38. Wolfe VF, Davidson S, Lee I. RTC: language support for real-time concurrency. *Proceedings of the 12th International Conference on Real-Time Systems (RTSS '91)*, IEEE Computer Society Press: Washington, DC, USA, 1991; 43–52, doi:10.1109/REAL.1991.160357.
39. Albers K, Bodmann F, Slomka F. Hierarchical event streams and event dependency graphs: A new computational model for embedded real-time systems. *Proceedings of the 18th Euromicro Conference on Real-Time Systems (ECRTS '06)*, IEEE Computer Society Press: Washington, DC, USA, 2006; 97–106, doi:10.1109/ECRTS.2006.12.
40. de Niz D, Bhatia G, Rajkumar R. Model-based development of embedded systems: The sysweaver approach. *Proceedings of the 12th IEEE International Symposium on Real-Time and Embedded Technology and Applications (RTAS '06)*, IEEE Computer Society Press, 2006; 231–242.
41. Caspi P, Curic A, Maignan A, Sofronis C, Tripakis S, Niebert P. From Simulink to SCADE/LUSTRE to TTA: a layered approach for distributed embedded applications. *Proceedings of the 2003 Joint Conference on Languages, Compilers and Tools for Embedded Systems & Soft. and Compilers for Embedded Systems (LCTES/SCOPES '03)*, ACM Press: New York, NY, USA, 2003; 153–162, doi:10.1145/780732.780754.
42. Gray I, Audsley NC. Exposing non-standard architectures to embedded software using compile-time virtualisation. *Proceedings of the 2009 International Conference on Compilers, Architectures, and Synthesis for Embedded Systems (CASES '09)*, ACM Press: New York, NY, USA, 2009; 147–156, doi:10.1145/1629395.1629417.
43. Krause M, Bringmann O, Rosenstiel W. Target software generation: an approach for automatic mapping of SystemC specifications onto real-time operating systems. *Design Automation in Embedded Systems* May 2006; **19**(5):229–251.
44. Hoare C. Communicating sequential processes. *Communications of the ACM* Aug 1978; **21**(8):666–677.
45. Campbell RH, Habermann AN. The specification of process synchronization by path expressions. *Operating Systems, Proceedings of an International Symposium*, *Lecture Notes in Computer Science*, vol. 16, Springer-Verlag: London, UK, 1974; 89–102.
46. Petri CA. Fundamentals of a theory of asynchronous information flow. *Proceedings of the IFIP Congress 62*, North Holland Publ. Comp.: Amsterdam, 1963; 386–390.
47. Mackert L. Modellierung, spezifikation und korrekte realisierung von asynchronen systemen Jul 1983.
48. Laventhal MS. A constructive approach to reliable synchronization code. *Proceedings of the 4th International Conference on Software Engineering (ICSE '79)*, IEEE Computer Society Press: Piscataway, NJ, USA, 1979; 194–202.
49. Jahanian F, Mok AK. Modechart: A specification language for real-time systems. *IEEE Transactions on Software Engineering* 1994; **20**(12):933–947, doi:10.1109/32.368134.
50. Lu S, Park S, Hu C, Ma X, Jiang W, Li Z, Popa RA, Zhou Y. MUVI: automatically inferring multi-variable access correlations and detecting related semantic and concurrency bugs. *Proceedings of the 21st ACM Symposium on Operating Systems Principles (SOSP '07)*, ACM Press: New York, NY, USA, 2007; 103–116, doi:10.1145/1294261.1294272.