Towards a Real-Time Systems Compiler

Fabian Scheler[1], Martin Mitzlaff[1],
Wolfgang Schröder-Preikschat[1], Horst Schirmeier[1]

[1]Department of Computer Science 4
Friedrich-Alexander University Erlangen-Nuremberg
Martensstrasse 1, 91058 Erlangen, Germany
{scheler,mitzlaff,wosch,schirmeier}@informatik.uni-erlangen.de

Abstract — *Model driven development gains more and more relevance for the development of hard real-time systems as it eases subsequent certification. Whereas generating the application source code from such models no longer is uncommon thanks to the research of the last years, targeting a specific operating system with an application still is done by hand. This significantly restrains flexibility and reuse within the development of real-time systems, as often assumptions on the underlying operating system are hard-coded within the application. Consequently we come up with a compiler-based approach to automatically map real-time applications coded for a specific operating system to other operating systems offering different abstractions (e.g. time-triggered vs. event-triggered execution). This compiler-based approach is enabled by an operating system independent intermediate representation, which allows to flexibly combine different operating system specific front- and back-ends.*

1 Introduction

In hard real-time systems not only the logical correctness of the results produced is of importance, but also the accomplishment of timing constraints is demanded for their successful operation. Missed deadlines can provoke consequences having an impact equivalent to that of incorrect results. In combination with automatic code generation model driven development eases the development of such systems a lot, as constraints assured on the level of the model are also kept by the source code generated from these models. Therefore, a lot of effort has been put into the research of model driven development processes. Due to the rising hardware complexity of real-time systems, so called real-time operating systems gain more and more popularity. While generating source code from abstract models no longer is uncommon, mapping the application to such real-time operating system is done manually most of the time. This paper explains, why this is a serious drawback for the development of real-time systems.

Thus, we suggest to automate this mapping by a compiler-based tool instead of applying a hand-crafted mapping. In other words, such a tool expects the source code of a real-time application (a simple operating system independent program or a program targeting

an abstract operating system API or even a specific operating system API e.g. OSEK [1] or OSEKtime [2]) as input and generates a real-time application targeting a different operating system or even programming language. Interesting combinations are e.g. operating systems supporting the event-triggered or the time-triggered control paradigm. A tool like this would result in decoupling the application from the employed system software as a conventional compiler decouples a programming language and the targeted instruction set architecture, thus, gaining a similar amount of portability and reusability of real-time applications across different operating systems as of programming languages across different processors. Some existing real-time system CASE tools[1] indeed already support generating code from abstract models targeting a specific operating system. However, these tools normally support only a single specific operating system and do not benefit from the opportunities a compiler-based approach offers.

The rest of this paper is organised as follows: In section 2 we present a succinct overview of the abstraction levels employed in the development of real-time systems, pointing out the importance of automatically lowering abstraction levels. After that, Atomic Basic Blocks (ABBs) introduced in [3] are revisited in section 3, because ABBs are perfectly qualified to constitute an intermediate representation for a compiler-based tool as outlined above. Section 4 picks up some benefits of such a compiler-based tool and section 5 discussed its preliminary design. In section 6 we illustrate a possible application of this tool within an automotive use case. Finally, section 7 sums up the paper and gives some prospects to future work.

2 Levels of Abstractions in the Development of Real-Time Systems

In this section abstractions employed in the state-of-the-art development process for real-time systems are examined. Firstly, the objectives of each abstraction are outlined and work that has been done on the level of this abstraction is exemplarily cited. Subsequently, the impact of the availability of transformations lowering abstractions in an automated fashion is explained and the benefits of a compiler-based transformation are sketched.

Figure 1 gives an overview on the abstraction levels encountered when engineering a real-time system. The actual level of abstraction decreases from the modelling level down to the hardware level. The boxes on the right side represent the lowering transformations mentioned above. Transformations annotated with a check mark can already be carried out in an automated and generic, i.e. compiler like, fashion, while those marked with a cross either have to be crafted manually or are only covered by fixed, i.e. non-generic, transformations.

2.1 Levels of Abstraction

Modelling The main purpose of modelling is to hide unnecessary details so the developer can focus on a single aspect of the real-time system, e.g. deployment, functional or temporal behaviour. Due to the rising complexity of software-based real-time systems it is very helpful to incorporate modelling approaches into the development of such systems. Moreover, modelling approaches often offer formal or semi-formal specified syntax and

[1] e.g. Matlab RTW (http://www.mathworks.com), TargetLink (http://www.dspace.com) or Rhapsody (http://www.ilogix.com)

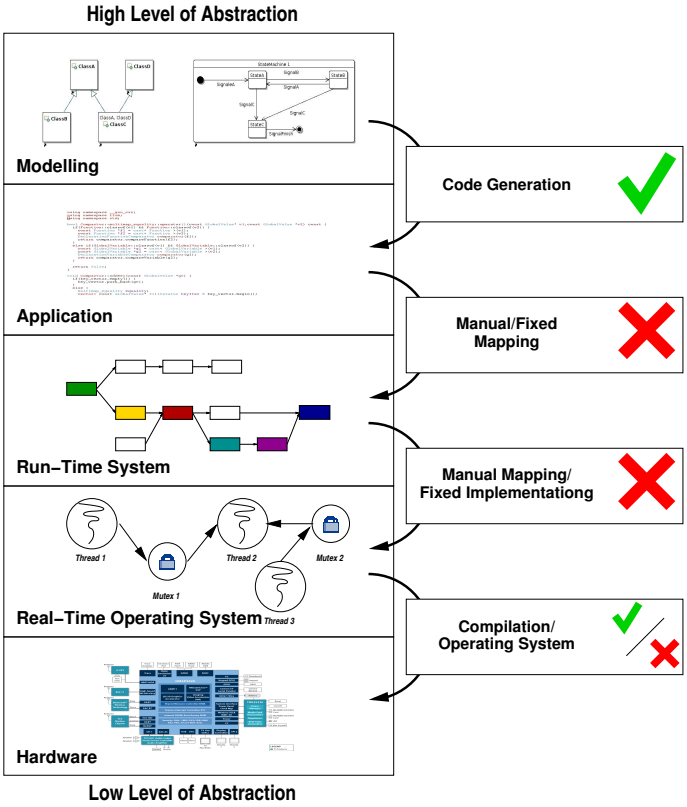

Figure 1: Abstraction Levels in the Process of Real-Time Systems Development

semantics, therefore allowing for complex analyses to verify the functional and the temporal correctness of a real-time system. Many of these analyses can hardly be conducted on lower abstraction levels like the application level. This property becomes more and more important due to an increasing demand of software-based hard real-time systems also in safety-critical environments. Lowering models to the application level means implementing these models. This can either be done manually or automatically by means of code generators.

A widely used kind of model to specify the behaviour of a system are statecharts or state machines. These models are part of UML [4] and are also used in other modelling tools like STATEMATE [5], Charon [6], AutoFocus [7], or Matlab/Simulink[2]. The deployment of software components to the nodes of a distributed system or replication can be modelled using e.g. SysWeaver [8] or AutoFocus [7], whereas the real-time extension of UML [9] can be used to explicitly model a real-time system's timing constraints.

Application The application provides an event-handler, i.e. a task implementation, for each event to be serviced in a real-time system. The structure of the application is mainly determined by techniques provided by conventional software engineering and the deployed programming language. The programming paradigm favoured for real-time systems development is the procedural paradigm (e.g. C, Ada), but object orientation (e.g. C++, Ada, Java) has gained more attention during the last few years as this paradigm often is more suitable for complex software systems. Especially for real-time systems it

[2]http://mathworks.com

is interesting to provide an explicit notion of time to express timing constraints on the programming language level. This is particular popular amongst data-flow languages like LUSTRE [10], the data-flow language described by Faustini et al [11], or time-triggered programming languages like Giotto [12]. The application either targets the bare hardware or an underlying run-time or operating system depending on the complexity of the application. There already exist some CASE tools, as TargetLink or AutoFocus [7], which provide source code generators for behavioural and deployment models targeting specific operating systems. In many cases however, this mapping is hand-crafted, i.e. the developer splits up the application in several tasks and embeds calls to run-time or operating system into the application.

Run-Time System A run-time system provides a software infrastructure to implement tasks, connect them to events and have them executed when the corresponding events occur, either in a time-triggered or in an event-triggered manner. Such run-time systems must not to be intermixed with operating systems. In most cases, these run-time systems provide a much higher level of abstraction than operating systems and are manually implemented on top of an operating system. So, the term *middleware* might in many cases be quite appropriate for such systems. Examples are the run-time system that executes the time-triggered Giotto programming language [12] running on top of RTLinux [13] or TMO [14] running on top of e.g. eCos [15].

Operating System Operating systems are the interface between the application and the hardware. The main tasks of an operating system are making available and multiplexing the resources offered by the hardware. Lowering an operating system to the hardware level is twofold. On the one hand an operating system is implemented using a programming language like C to an extent as great as possible. The lowering of a C program to a specific instruction set architecture can be performed by a compiler and, thus, is automated. On the other hand, the purpose of an operating system is just to hide the complexity of the hardware. So, designing and implementing an operating system also means dealing with all the peculiarities of the underlying hardware. It is very unlikely that this ever can be automated in a reasonable way.

Hardware Real-time systems not only comprise software encapsulating the functional logic of the application and the run-time or the operating system, respectively, but also the hardware the system runs on. Additionally, customised hardware can be utilised for special purposes, e.g. implementing special algorithms using a FPGA. This paper, however, focuses on software-based real-time systems. Hence, hardware is beyond the scope of this paper and will not be discussed any further.

2.2 The Impact of Automated and Generic Transformations

Transforming higher levels of abstraction into lower ones primarily in an automated fashion means that a manual step in the development process is replaced by a machine-based one. Naturally, this results in saving time and often also in more efficient transformation outputs, but maybe most important, the elimination of an additional source of errors. Moreover, two classes of automated transformation have to be distinguished: *generic* and *special purpose* transformations. Generic transformations decouple different levels of abstraction by providing specific front-ends and back-ends that can be combined in a flexible

way through a common intermediate representation. Special purpose transformations on the other hand propagate requirements of the front-end down to the back-end and vice versa because the intermediate representation is omitted. So such transformations cannot effectively decouple different levels of abstraction. Those transformations provide nothing more than a fixed mapping between inputs and outputs fulfilling these requirements. For all transformations, however, the targeted level of abstraction has to be expressive enough to implement all constructs of the higher level of abstraction.

Code generators and compilers provide generic transformations to lower models to the application level and source code to the hardware level, respectively. The existence of these tools have a serious impact on the development of real-time systems. The absence of code generators in model driven development on the one hand would e.g. imply that it has to be assured manually that the hand-crafted implementation really implements the model. Otherwise, analyses and proofs performed on those models were completely useless. Both, manually implementing the model and manually evidencing the equivalence of model and implementation, are very time-consuming and also error-prone tasks. High-level programming languages on the other hand would be completely unusable without appropriate compilers. If the high-level language constructs had to be lowered to assembly by hand there would be no benefit in using such programming languages at all. The existence of an appropriate compiler is absolutely vital for the success of a programming language. So code generators decouple the modelling and the application level with respect to the employed programming language whereas compilers decouple the application and the hardware level with respect to the employed programming language and instruction set architecture.

The major obstacle when decoupling the application from the run-time or operating system level is to deal with the control paradigm employed by the run-time or operating system. This can either be a time-triggered or an event-triggered one. These control paradigms are accompanied by fundamentally different control flow abstractions. While threads in time-triggered systems expose run-to-completion semantics they may block due to explicit synchronisation in event-triggered systems. This hampers porting real-time systems between run-time environments implementing different control-paradigms a lot and often only a complete redesign and reimplementation is viable. So software reuse on the level of complete event handlers across different control paradigms on the application level is very labour intensive due to the complex manual porting.

There are approaches offering a closed tool chain starting at the modelling level down to the hardware. These tool chains, however, use special purpose transformations when mapping the application to the run-time environment. The SysWeaver [8], for instance, explicitly specifies event-triggered semantics already on the modelling level and propagates this property down to the execution environment by targeting an event-triggered operating system. The time-triggered counterpart, AutoFocus [7], employs a “formal time-synchronous operational semantics” on the modelling level and, therefore, also targets a combination of time-triggered operating system and bus system, namely OSEKtime [2] and FlexRay [16]. Giotto [12] and TDL [17], on the other side, are based on a time-triggered programming language and demand for a time-triggered execution of the real-time application. These tool chains require a certain control paradigm on the input level (this is either the modelling or the application level) and propagate this control paradigm

to the execution environment. They do not effectively decouple the application level from the run-time or operating system level with respect to the employed control paradigm.

2.3 Conclusion

In order to alleviate the deficiencies of existing tool chains as outlined above we propose a compiler-based solution to map applications to run-time environments and thereby decouple these abstraction levels. The compiler therefore has to dispose an intermediate representation of a real-time systems that is

1. independent of any particular control paradigm and
2. expressive enough to describe all relevant properties of an implementation of a real-time system.

A proper intermediate representation called Atomic Basic Blocks is described in the subsequent section 3.

3 ABBs: An Intermediate Representation for Real-Time Systems

A widely used intermediate representation utilised by compilers are control flow graphs composed of basic blocks. While such a representation clearly is independent of a particular control flow abstraction offered by an operating system, it is also not expressive enough to represent complete real-time applications. Usually the description of a complete real-time system comprises a forest of control flow graphs connected by data dependencies, other explicitly enforced dependencies, and mutual exclusion constraints marking critical sections. These dependencies cannot be expressed within conventional control flow graphs.

The prerequisite to capitalising on such an intermediate representation is, that a real-time application can be mapped to this representation. So it is not very promising to support arbitrary applications, and we demand for real-time applications having a structure as illustrated in figure 2. In short, we assume that real-time systems are assembled from one or more tasks. The execution of a task always is initiated by an event, whether the event should be serviced in an event-triggered or a time-triggered fashion does not matter here. A task consists of at least one sub-task and may *fork* further sub-tasks (1). Each sub-task constitutes an own event handler for a particular event, i.e. a single event can be handled by different sub-tasks each of them computing its own results. Deadlines can be specified for complete tasks, thus applying for all sub-tasks forked by the task, for single sub-tasks, or combinations of tasks and sub-tasks. Furthermore, in order to capture directed dependencies, two or more sub-tasks can join into one common sub-task. These joins can either have *or* (2) or *and* (3) semantics.

In [3] we already came up with an extension to conventional basic blocks called atomic basic blocks (ABBs) that exactly allow for specifying the dependencies listed at the beginning of this section. As conventional basic blocks, ABBs can be distinguished in minimal and maximal ABBs. Minimal ABBs are identical to minimal basic blocks while maximal ABBs aggregate one or more basic blocks. The criteria for grouping one or more basic blocks into one maximal ABBs are as follows:

1. A maximal ABB always starts at the end of the preceding ABB in the control flow

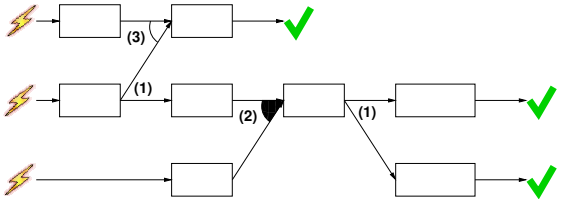

Figure 2: Expected Structure of Real-Time Systems

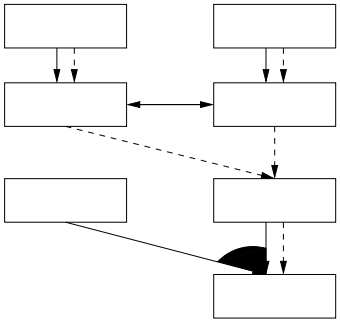

Figure 3: Atomic Basic Blocks

graph and continues to an appropriate termination of the ABB.

2. Appropriate terminations of ABBs are:
   - a sub-task forks another sub-task
   - a sub-task joins another sub-task due to data dependencies (sub-task T_1 defines a variable sub-task T_2 reads) or other explicitly modelled dependencies (e.g. unilateral synchronisation or temporal dependencies)
   - a sub-task is joined by another sub-task for the same reasons as stated above
3. Critical sections form an exception regarding these rules: a critical section always is enclosed by one maximal ABB.

Maximal ABBs can be split into non-maximal or minimal ABBs except those ABBs that enclose critical sections. This is also the origin of the term *Atomic Basic Blocks* – ABBs enclosing critical section cannot be split and are therefore atomic on the level of ABBs. Figure 3 illustrates some possible relations among different ABBs. The white boxes emblematise ABBs, the different types of edges depict different relations among ABBs: control flow (solid edges), data flow (dashed edges) and explicit dependency edges (control flow edges connected with an arc). Mutual exclusion relations among ABBs enclosing critical section are captured using bidirectional edges.

With respect to the given definition of ABBs and the required structure of real-time systems stated above, a real-time system now is described as a forest of control flow graphs consisting of ABBs. Dependencies among different sub-tasks can be modelled using joins either having *and* or *or* semantics, mutual exclusion constraints are covered by ABBs enclosing critical sections that are related by an undirected graph.

We are quite confident that real-time systems can be described by the means stated above, as such systems fulfil some assumptions that are not kept by arbitrary programs:

1. For successfully building and running a real-time system lots of a-priori knowledge has to be gathered to validate the timing constraints in such systems. This a-priori knowledge comprises e.g. detailed characterisation of the events to be serviced or dependencies between the sub-tasks implementing the handlers for these events.
2. For the sake of being statically analysable, often certain programming language constructs are abandoned especially in safety-critical real-time system, as these constructs might destroy this property. An example for such programming language constructs are virtual function calls.

These assumptions ease the static analysis a lot and usually do not apply to arbitrary programs. It is e.g. rather unlikely that arbitrary functions are called through a function pointer in safety critical real-time systems, whereas this is not unusual in *normal* programs. Therefore, we are not convinced that arbitrary programs can be transformed into ABBs as described in this section. Beyond that, we doubt that it is possible to automatically map arbitrary programs to arbitrary run-time or operating systems. So the idea presented here has not to be intermixed with the various approaches of automatically generating multi-threaded programs by the help of an optimising compiler.

4 Benefits of a Compiler-Based Approach

The major benefit of a tool employing an intermediate representation as sketched above definitely is that it decouples the application from the run-time environment with respect to the particular control paradigm. ABBs are an extension of conventional basic blocks and depend neither on a time-triggered nor an event-triggered execution environment, therefore, ABBs can be mapped to control flow abstractions of both control paradigms. The decision for one of these real-time architectures can be postponed until it is clear what kind of events have to be serviced in the particular real-time system, thus enabling the reuse of complete event-handlers across different real-time systems also employing different real-time architectures.

Besides that, a compiler-based approach has more advantages. Some of these advantages already haven been discussed in [18], a few of them should be picked up here exemplarily:

Improved construction of time-triggered schedules Most conventional tool chains map rather coarse grained components to the run-time environment. The mapping in a compiler, by contrast, can take place on the level of the control flow graph. This is very helpful when e.g. computing static schedules for the cyclic executive model [19]. Such a scheduler demands the execution of a sub-task to fit in a slot of predefined temporal length. In the worst case, i.e. a sub-task does not fit into such a slot, a sub-task has to be split up in several pieces that are distributed to appropriate slots manually. A control flow graph, on the other hand, can be split up and mapped to these slots by a compiler in such a way that these slots are entirely occupied by ABBs without manual intervention.

Global Optimisation Optimisation techniques like constant, copy or value range propagation offered by conventional compilers can be supported by the a-priori knowledge about the events a real-time system has to service. Thereby, these techniques can be exploited to perform global optimisation much more aggressive than it is possible for existing tools that just offer a fixed mapping between the application and the run-time environment. Examples for such optimisations at global scope are e.g. elimination of unnecessary thread synchronisation or context switches.

Support for Legacy Software A compiler-base approach could also provide support for existing software by providing an appropriate front-end for transforming these implementations into ABBs. This is not possible for tool chains that only accept input at the modelling level.

5 Design of the Real-Time Systems Compiler

In this section we present and discuss the preliminary design of a tool based on the intermediate representation provided by ABBs. Due to the disposed intermediate representation and the ability to combine different front-ends and back-ends such tool is very close to a traditional compiler, hence we call it the *Real-Time Systems Compiler* (RTSC). First we have a look on the overall structure of the real-time systems compiler, then the inputs expected, the outputs generated by this tool and its building blocks are discussed.

In earlier work [3] we had in mind to extend the GCC for this purpose, but it soon became evident that the *Low-Level Virtual Machine* (LLVM) [20] is much better suited for this project. The LLVM provides a generic and modular compiler framework that uses the *Low-level Virtual Instruction Set Architecture* (LLVA) [21] as intermediate representation which will serve as base for the implementation of ABBs. The main design objectives of LLVM/LLVA are to provide an intermediate language expressive enough to allow high-level analysis and optimisation in combination with easy lowering to machine code, thus enabling efficient global analysis and optimisation. A tool like the RTSC could benefit from these features. Mapping a real-time application to a run-time or operating system yields several problems that have to be tackled at a global scope like thread synchronisation or inter-thread communication.

5.1 Overall Design

Figure 4 gives an overview over the structure of the RTSC tool. First the tasks of the real-time system are imported from some kind of implementation by a *Front-End* that always is programming language specific and run-time or operating system aware if necessary. The Front-End generates the intermediate ABB-representation from these tasks. In the following step the WCETs of the single ABBs are determined and provided to a combined *Analyser/Composer*. Tasks are analysed with respect to logical (explicitly defined dependencies or data dependencies) and temporal dependencies and are subsequently mapped to the control flow abstractions offered by the targeted operating system preserving all these dependencies. The generated real-time system then is fed into a *Checker* performing schedulability analysis along with the WCETs of the various ABBs. If all the timing constraints are satisfied, the *Back-End* emits the final implementation of the real-time system. In case the *Checker* fails to verify one or more deadlines, the *Composer/Analyser* is informed to search for a more accurate mapping with respect to the hints given by the *Checker* (e.g. a deadline is missed and the corresponding task has been blocked due to a resource conflict).

5.2 Inputs

The description of the relevant events are stored in task databases. These databases characterise the relevant events (e.g. periodic vs. aperiodic vs. sporadic events, period, deadline, ...) and establish a mapping between these events and the corresponding event handlers (i.e. implementations of sub-tasks). For every event stored in the *Source Task Database* an appropriate event handler exists in the *Source Implementations*. A subset of these events is selected and stored in the *Target Task Database* together with the mapping to the event handlers provided by the *Source Implementations*. These events are those

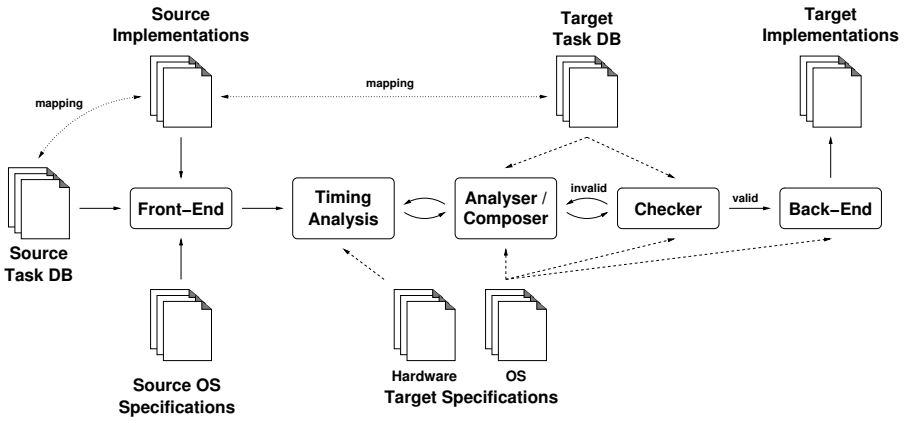

Figure 4: Overall RTSC Design

to be serviced in the target real-time system. The properties of the underlying run-time or operating system employed in the source and the target real-time system are encapsulated in *OS Specifications*. Such a specification e.g. comprises the syntactic and semantic specification of the operating system API or the employed scheduler. Furthermore, a specification of the target hardware (instruction set, processor pipeline and caches, ...) is needed for the timing analysis.

5.3 Outputs

The output of the RTSC tool is a real-time application providing event handlers for all events specified in the *Target Task Database*. The application targets a run-time or operating system specified by the *Target OS Specification* (cf. 5.2) that is supposed to provide the necessary infrastructure to execute these tasks, i.e. the generation of driver code is not treated so far. Nevertheless, it is possible that the source and the target run-time system offer very different semantics with respect to their real-time architecture (e.g. polling vs. callbacks). In this case references of such drivers have to be identified and properly mapped to the corresponding driver of the target run-time or operating system. Yet, we assume that the semantics of the driver can be adequately hidden by the source and the target run-time system so no such conversion is needed.

5.4 Components

Front-End The *Front-End* is a compiler front-end that is programming language specific and, if necessary, operating system aware. Its main purpose is to transform the Source Implementation into the intermediate ABB representation.

Timing Analysis The *Timing Analysis* is of crucial importance for the whole RTSC tool, as neither the *Analyser/Composer* nor the *Checker* can do without timing information. It is not surprising that schedulability analysis is not possible without the necessary timing information, but this information is also essential for the *Analyser/Composer*. A time-triggered target operating system, for example, requires the provision of pre-computed schedule tables. Without knowing the WCET of the implementation such a schedule table could not be constructed.

While the retrieval of flow-facts (e.g. loop bounds, cf. [22]) should not be more difficult than for the LLVA than other instruction set architectures, the provision of a proper execution time model will be very difficult in the RTSC tool. The result of the RTSC is a real-time application targeting some kind of run-time or operating system, i.e. the RTSC is a source code transformation system producing e.g. C code. This C code is then compiled to machine code by a C-compiler. For a given basic block on the level of the LLVA, therefore, it is hard to predict what the generated machine code finally will look like. A possible solution to compute the WCET of an ABB would be to look up the WCET of every LLVA instruction for the specific target architecture in a database and enforcing a static mapping of LLVA instructions to machine code similar to RT-SYN [23]. This, however, would prohibit any optimisation performed during the machine code generation and yield very inefficient code. Another solution would be to emit machine code for the entire real-time system and to determine the WCET of the basic blocks directly on the machine code level. This, however, demands for a difficult transformation of the flow-facts, due to the complex translation process, as the optimisations performed by the C-compiler might also result in an altered control flow graph. So it might be a good compromise to emit machine code for a single ABB and to determine the WCET of this ABB either by measurement (i.e. executing the ABB on a physical processor) or static analysis. In combination with the gathered flow-facts the overall WCET could be computed. The outcome is a WCET less pessimistic than the WCET yielded by the method using a database that is computed in a less complex way than proposed in the second approach.

Analyser/Composer The *Analyser* component examines data dependencies and dependencies resulting from mutual exclusion. Exploiting the information from the *Target Task Database* and the timing information provided by the timing analyses, the Analyser tries to ensure analytically that these dependencies are always preserved. A task T1, for instance, is activated by the occurrence of an event E1 and produces data that another task T2, activated by an event E2, consumes. There is no need to establish this dependency explicitly, if it can be assured that the producer T1 produces a piece of data no later than 100ms after the occurrence of E1, and E2 always occurs at least 500ms after E1.

Mapping the task implementations to abstractions offered by the target operating system is the duty of the *Composer*. Mutual exclusion constraints as well as explicit or implicit dependencies among different task have to be ensured by appropriate synchronisation mechanisms deployed by the target operating system as long as they cannot be ensured analytically. It is worth mentioning that this composition is a lot more than just aggregating procedure calls into or mapping software components to a single thread, as the composition takes place at global scope using the ABB-based representation yielded by the RTSC front-end. The mapping generated by the *Composer* might also demand transformations of the control flow graph and consequently also influences the WCET of the emitted code. Therefore, timing analysis and composition have to be accomplished in an incremental and closely related manner.

Checker The *Checker* finally performs a schedulability analysis on the real-time system generated by the Composer. The algorithm to be used for the analysis is determined by the targeted operating system. If the *Checker* fails to verify that all deadlines are met within the generated real-time system, the *Composer* is informed to generate a different

variant of the system. Thereby, the *Composer* is guided by hints of the *Checker*, e.g. a certain deadline might be missed due to blocking a higher priority task resulting from a resource conflict.

Back-End Finally, the *Back-End* translates the real-time system generated and verified in the foregoing steps into an application targeting the desired run-time or operating system.

6 An Automotive Use Case

A sector, with demand of both time-triggered and event-driven systems is the automotive industry. With the introduction of the FlexRay bus [16] network and communication architecture of cars has a new time-triggered element. FlexRay is a relatively new TDMA-based bus system and should solve the demand for a faster and dependable communication bus between the Electronic Control Units (ECU) in a car. Another well established bus system is the event-driven, priority-based CAN bus.

The first FlexRay-ECUs are existing, well-engineered ECUs, that were formerly connected via CAN. So ECUs (and the applications running on them) formerly communicating through a priority-based network are now, if the ECU is part of a FlexRay cluster, connected with a time-triggered bus system. This migration should, of course, take place with as less as possible new development.

The simplest way of migrating an application to this new bus system is just to adjust the calls to the communication module and to put some kind of buffering adaptor between the application and the FlexRay communication system. The FlexRay driver thus reads the message from that adaptor, when it is time for sending. A task servicing periodic events can finish with filling the send-buffer and triggering the communication-system to send the message. The problem is, that it is not known how much time it will take till the message is really sent. This can be nearly immediate, if the slot for sending this message is ahead. But it can also take a multiple of the FlexRay cycle-time, depending on the FlexRay schedule, the use of slot-multiplexing and the point in time, when the transmission is requested. So the originally periodic events now come with a big jitter.

One way to lower the jitter is to use oversampling when planning the FlexRay scheduling. Another, bandwidth-saving way is to migrate the whole system to a time-triggered system. Here the timing of the event, the threads servicing it and the FlexRay bus can be synchronised. Among other things the RTSC can help doing this in an automatic way. On the one hand it can map the applications to the time-triggered run-time system. Just generating schedule tables might work with other tools, too. However, if no valid schedule can be found, maybe because the WCET of one thread is too long and that very (or some other) thread has to be split up manually? The RTSC will handle these conflicts automatically. The timing information gathered by the timing analysis component enables the RTSC tool to map the ABBs to threads in such a way that the threads exactly fit into the pre-allocated time slots of the schedule table (cf. section 4).

7 Summary

At first we revealed two chain links that are missing to provide a closed tool chain for the development of real-time systems that is capable of automatically lowering high-level

models down to low-level implementations in a generic way by effectively decoupling the employed abstraction levels. The first one is the mapping of real-time applications to appropriate run-time or operating systems. The second one is the development of appropriate run-time and operating systems for different hardware targets itself.

After that, we introduced and discussed the preliminary design of the RTSC tool that could provide the first missing chain link mentioned above. The RTSC tool exploits an intermediate representation called ABBs, that are based on control flow graphs. This intermediate representation is independent of the control flow abstractions employed by a run-time or operating system and therefore allows for a flexible combination of front-ends and back-ends importing and targeting different kinds of run-time and operating systems. We identified the timing analysis as the most challenging component of the RTSC. On the one hand the timing analysis is the requirement for many analyses, optimisations and the generation of time-triggered real-time applications. On the other hand it is very difficult to provide an accurate execution time model for the LLVA within the RTSC tool due to the complex translation process.

Subsequently, we illustrated how such a tool can support the migration from event-triggered CAN-based applications to time-triggered FlexRay-based applications in the automotive area.

Currently, an early prototype of a front-end transforming plain C programs into the ABB-based intermediate representation exists. The next steps will be to finish the implementation of this front-end, tackle the timing analysis problem and develop a back-end for the OSEK [1] as well as the OSEKtime [2] operating system.

References

[1] OSEK/VDX Group. *Operating System Specification 2.2.3*. OSEK/VDX Group, February 2005. http://www.osek-vdx.org/.

[2] OSEK/VDX Group. *Time Triggered Operating System Specification 1.0*. OSEK/VDX Group, July 2001. http://www.osek-vdx.org/.

[3] Fabian Scheler and Wolfgang Schröder-Preikschat. Time-triggered vs. event-triggered: A matter of configuration? In *Proceedings of the GI/ITG Workshop on Non-Functional Properties of Embedded Systems*, pages 107–112, Nuremberg, Germany, March 2006. VDE Verlag GmbH.

[4] Grady Booch, James Rumbaugh, and Ivar Jacobson. *The Unified Modeling Language User Guide*. Object Technology Series. Addison-Wesley, Reading/MA, 1999.

[5] D. Harel, H. Lachover, A. Naamad, A. Pnueli, M. Politi, R. Sherman, and a. Shtul-Trauring. STATEMATE: a working environment for the development of complex reactive systems. In *Proceedings of the 10th International Conference on Software Engineering (ICSE '88)*, pages 396–406, Los Alamitos, CA, USA, 1988. IEEE Computer Society Press.

[6] Rajeev Alur, Thao Dang, Joel M. Esposito, Yerang Hur, Franjo Ivancic, Vijay Kumar, Insup Lee, Pradyumna Mishra, George J. Pappas, and Oleg Sokolsky. Hierarchical modeling and analysis of embedded systems. *Proceedings of the IEEE*, 91(1):11–28, 2003.

[7] Jewgenij Botaschanjan, Alexander Gruler, Alexander Harhurin, Leonid Kof, Maria Spichkova, and David Trachtenherz. Towards modularized verification of distributed time-triggered systems. In Jayadev Misra, Tobias Nipkow, and Emil Sekerinski, editors, *Proceedings of the 14th International Symposium on Formal Methods (FM '06)*, volume 4085 of *Lecture Notes in Computer Science*, pages 163–178. Springer-Verlag, August 2006.

[8] Dionisio de Niz, Gaurav Bhatia, and Raj Rajkumar. Model-based development of embedded systems: The sysweaver approach. In *Proceedings of the 12th IEEE International Symposium on Real Time*

and Embedded Technology and Applications (RTAS '06), pages 231–242. IEEE Computer Society Press, 2006.

[9] Bruce Powel Douglass. *Real-Time UML*. Addison-Wesley, Reading, USA, 1998.

[10] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. The synchronous data-flow programming language LUSTRE. *Proceedings of the IEEE*, 79(9):1305–1320, September 1991.

[11] Antony A. Faustini and Edgar B. Lewis. Toward a real-time dataflow language. *IEEE Software*, 3(1):29–35, January 1986.

[12] T.A. Henzinger, B. Horowitz, and C.M. Kirsch. Giotto: a time-triggered language for embedded programming. *Proceedings of the IEEE*, 91(1):84–99, 2003.

[13] Thomas A. Henzinger, Christoph M. Kirsch, and Slobodan Matic. Composable code generation for distributed giotto. In *Proceedings of the 2005 Workshop on Languages, Compilers and Tools for Embedded Systems (LCTES '05)*, pages 21–30, New York, NY, USA, 2005. ACM Press.

[14] K. H. Kim and C. Subbaraman. The TMO structuring approach and its potential for telecommunication applications. In *Proceedings of the 1st IEEE Workshop on Application-Specific Software Engineering and Technology (ASSET '98)*, Washington, DC, USA, 1998. IEEE Computer Society Press.

[15] Jung-Guk Kim, Moon Hae Kim, Kwang Kim, and Shin Heu. TMO-eCos: An ecos-based real-time micro operating system supporting execution of a TMO structured program. In *Proceedings of the 8th IEEE International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC '05)*, pages 182–189, Washington, DC, USA, 2005. IEEE Computer Society Press.

[16] FlexRay Consortium. *FlexRay protocol specification 2.1 Revision A*. FlexRay Consortium, December 2005. http://www.flexray.com.

[17] Emilia Farcas, Claudiu Farcas, Wolfgang Pree, and Josef Templ. Transparent distribution of real-time components based on logical execution time. In *Proceedings of the 2005 Workshop on Languages, Compilers and Tools for Embedded Systems (LCTES '05)*, pages 31–39, New York, NY, USA, 2005. ACM Press.

[18] Fabian Scheler and Wolfgang Schröder-Preikschat. Synthesising real-time systems from atomic basic blocks. *12th IEEE International Symposium on Real Time and Embedded Technology and Applications (RTAS'06)*, April 2006. WiP presentation.

[19] Theodore P. Baker and Alan C. Shaw. The cyclic executive model and ada. In *Proceedings of the 9th International Conference on Real-Time Systems (RTSS '88)*, pages 120–129. IEEE Computer Society Press, 1988.

[20] Chris Lattner and Vikram Adve. Llvm: A compilation framework for lifelong program analysis & transformation. In *Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO'04)*, Mar 2004.

[21] Vikram S. Adve, Chris Lattner, Michael Brukman, Anand Shukla, and Brian Gaeke. LLVA: A low-level virtual instruction set architecture. In *Proceedings of the 36th annual IEEE/ACMInternational Symposium on Microarchitecture (MICRO 36)*, pages 205–216. IEEE Computer Society Press, 2003.

[22] Raimund Kirner and Peter Puschner. Classification of WCET analysis techniques. In *Proceedings of the 8th IEEE International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC '05)*, pages 190–199, Washington, DC, USA, 2005. IEEE Computer Society Press.

[23] T.E. Smith and D.E. Setliff. Towards an automatic synthesis system for real-time software. In *Proceedings of the 12th International Conference on Real-Time Systems (RTSS '91)*, pages 34–42. IEEE Computer Society Press, 1991.