Code Generation for Timed Automata System Specifications ConsideringTarget Platform Resource-Restrictions

Daniel Opp*, Mirko Caspar*, and Wolfram Hardt*

Abstract

A transformation concept and the resulting code generator for the automated implementation of timed automata, given as UPPAAL specification, are proposed in this paper. The concept aims to generate code for data memory restricted target platforms, like embedded systems and sensor network nodes. The output of a transformation is a pointer oriented Cimplementation of the state machines that considers the specified time behaviour of the automata. Due to the availability of this code generator, a complete homogeneous and closed work flow can be proposed. It includes specification, verification, and simulation of timed automata in UPPAAL and the automated implementation using the code generator. Hereby, the implementable subsets of timing specification are realised in the compilable code.

The static and dynamic performance of the generated code is checked by comparison of a traditional thread-based implementation of a sensor network application with an according automatically generated system. The results demonstrate a reduction of memory usage of more than 80%, a decreased cyclic execution time at a moderate increasing code size.

Keywords: Code Generatetion, Automata Transformation.

1. Introduction

Thread-based preemptive system development is very popular and approved. Different state machines and algorithms are implemented in independent threads that may be synchronised to working states. A scheduler, usually as part of the operating system, coordinates the resources and execution order of the set of threads. However, the development of applications for target platforms with restricted resources is difficult. Especially the stacks, that are necessary to store the states of all threads, need a lot of RAM. Additionally an operating system is necessary and allocates additional resources.

During the last years, alternative implementation concepts arose for platforms with restricted resources. Some of them base up on state machines (FSM). Hereby, the control flows of algorithms are implemented as sets of FSMs using a defined formalism. The data flows have to be added as functions. The program can be compiled to a static loop evaluating the state changes of each FSM.

This concepts lack in the feasibility to specify temporal behaviour since the semantic of FSMs is not able to express temporal dependencies. In contrast, timed automata extend the concept of FSMs and allow the modelling of a timed behaviour of state based systems. The theory was introduced by Alur and Dill [1]. It defines a timed automaton as the tuple

$$A = (\Sigma, S_0, C, E) \tag{1}$$

Hereby, S is the set of states and $S_0 \subseteq S$ is a subset of initial states. C is the set of clocks that proceed at the same speed. $E \subseteq S \times S \times 2^C \times \Phi(C)$ is a set of transitions from a state $\in$ S to a state $\in$ S where the clock constraints $\Phi(C)$ hold. The binary vector 2^C describes which clocks have to be reseted when the state transition is executed. Σ defines a finite set of input symbols. Timed automata can be non-deterministic.

Several tools for specification and verification of timed automata have been published. Popular examples are UPPAAL [2] and Kronos [3].

UPPAAL is used as formal specification tool for the work flow presented in this paper. It allows graphical specification, user controlled simulation and computation tree logic (CTL) based verification of cooperating timed automata [2]. A typical UPPAAL specification consists of a set of timed automata and some additional elements like synchronisation

* *Department of Computer Science, University of Chemnitz, Chemnitz, Germanny.*

channels and variables. The data flow is declared in C-like functions, called action blocks. Each automata description is realised as a template and different instances are built by parameterisation of the templates.

The main focus of UPPAAL is the simulation and formal verification of system models. Continuing, this paper presents a work flow that uses timed automata system specifications of UPPAAL for the generation of compilable C-code. A code generator has been developed for the transformation. Its special focus is the generation of executable systems that can be hosted on system with restricted resources, especially data memory (RAM).

Accordingly, the work flow is especially suitable and favourable for the development of control-flow dominated embedded systems running parallel algorithms. Common examples are sensor networks, where communication stacks and different applications have to be implemented on each node.

This paper is organised as follows: first a brief outline of related work is given. An overview about the whole development work flow is followed by details about the code generator and the related transformation rules. Furthermore, performance values and the results of implementation experiences of some exemplary implemented systems are presented and discussed.

2. Related Work

Due to the data memory restrictions of wireless sensor network nodes, some work about concepts for resource saving of reactive and parallel systems has been done. Event driven solutions like TinyOS [4] are not able to deal with explicit system states. Hence, the developer is responsible to realise system states as manual implemented state machines or implicit defined variable values. Since event driven systems introduce an event handler for each internal or external event, the implementation of the state transitions is often distributed over several events.

Thread-based systems like MantisOS [5] try to realise context switches with the lowest possible data memory allocation. But they cannot overcome the need to save all registers, the call history and local variables at a several stack area for each thread. Thus, several hundreds of bytes are necessary for the call stacks.

Protothreads [6] are a solution to map cooperative thread based implementations to finite state machines. The threads are programmed by adding some preprocessor macros to the thread functions for thread begin, end and all blocking conditions. The macros insert a switch case structure to the final code. A basic system has to call all protothread functions in a cyclic executive. Protothreads form a state machine for each thread and the states are labelled by the code line numbers of the original code. Control structures, like loops, may be used in Protothreads, but there exist restrictions concerning function calls and switch case statements. Another limitation is the lifetime of local variables. Their values are only stored until the next preemption point. Global variables may create some additional data memory costs and prohibit a Protothread instantiation.

Since state machines are a good way to realise parallel executed algorithms, the Object State Model (OSM) [7] uses automata specifications for the system implementation. The textual definitions are translated by a code generator into Ccode with external data processing functions. Since the OSM design flow uses an Esterel-description as an intermediate language, the generated code is not comprehensible and therefore not suitable for debugging purposes.

Furthermore there is some research about the code generation of timed automata specifications. Times [8] translates UPPAAL specifications into executable real time considering implementations. Due to the offered real time support, Times uses a real time system and accordingly, it allocates a huge amount of data memory for the call stacks. Moreover Times uses transition tables, which require some additional memory.

3. UPPAAL-Based Workflow

UPPAAL allows the formal specification of a system by a set of timed automata. A timed automaton consists of clocks, states (which are called locations) and transitions between the states. Clocks are time variables, which progress in time, all at the same speed. A state may have an invariant parameter, which forces the automata to leave the state until a specified time. Each transition has a guard and an action block. The guard is a condition based on clocks and variables. The transition is called enabled if the guard evaluates to true. An action block contains assignments of variables and clocks and is executed when the according transition is taken to change the state.

A synchronisation concept allows the modelling of dependencies between different state machines. It is realised by two kinds of synchronisation channels. On the one hand, binary channels for the synchronisation of a sender and a receiver transition. On the other hand, broadcast channels synchronise a sender with a various number of receivers.

While binary synchronisation always requires a sender and a receiver, sending transitions at broadcast channels are also feasible if no receiver is involved. In opposite to the theory of timed automata, UPPAAL does not support input symbols. Instead, it uses multiple channel synchronised automata. The synchronisation can be considered as the input symbols for a single automaton.

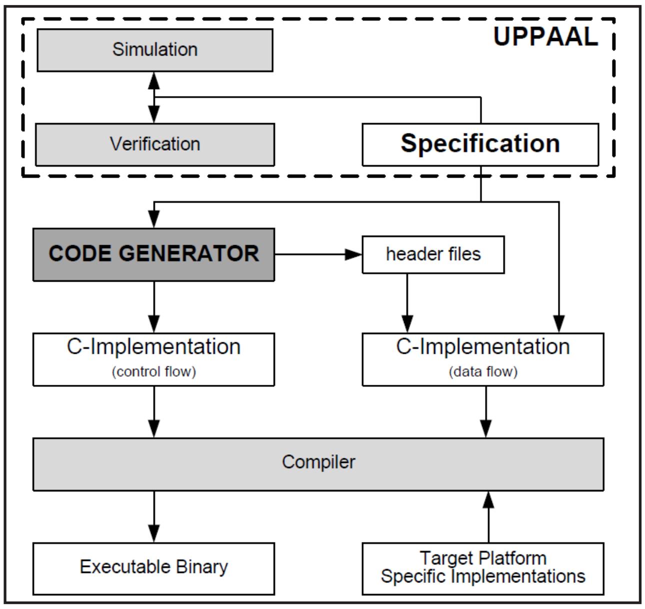

Figure 1 *Timed automata based work flow: the UPPAAL specification is used to generate a C-implementation of the control flow and the data flow (partly).*

Fig. 1 illustrates important parts and relations of the work flow. An UPPAAL specification of a system is the base. It has to be modelled by a developer. He has to consider, that UPPAAL allows the specification of non-deterministic state machines, which cannot be implemented unambiguously on a target platform. Hence, for simulation and verification on the one hand, and implementation on the other hand, the specification has to be separated into two parts. Fig. 2 illustrates this aspect. One system, which defines the behaviour of the final implementation, is called behaviour specification and has to be deterministic. The second part is the testbench system. It generates the input symbols (realised by synchronisation channels) for the behaviour specification and simulates the external data processing. A non-deterministic testbench can be utilised to cover all possible timed input sequences.

The specification can be simulated and verified by the tools of UPPAAL. After finishing this test period, the developer can use a code generator that has been developed as part of the proposed work flow and transforms the behaviour specification to compilable C-code. The state machine itself

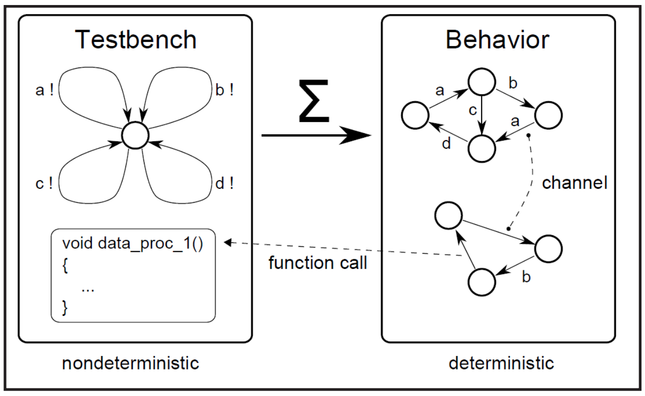

Figure 2 *UPPAAL specification pattern for simulation and verification of an implementable system.*

is transformed to a pointer-based set of C-functions. The data processing that is modeled in the action blocks is exported to header files and the appropriate function bodies. It can be edited by the developer. Additionally, all accesses to dedicated hardware have to be encapsulated in functions, which have to be implemented manually. These functions can be compiled to a library and used for further developments on the same target platform. All parts can be compiled and linked to an executable program by a usual platform aware compiler.

4. Transformation Concept

In order to generate executable C-code from UPPAAL behaviour specifications a mapping of the specification elements into compilable C-code fragments is necessary. The used constructs have to consider the requirement to generate code for memory restricted systems. Naturally, fast execution speed and acceptable machine code size are optimisation aims as well. In the following parts, some basic and exemplary transformation concepts are presented.

A. Automata Transformation

First step is to transform the basic automata behaviour, the states and the transitions between them. Every state s 2 S is mapped to a C-function which implements all guards and action blocks of the outgoing edges. An array of function pointers stores the actual state of each state machine by pointing to the actual state function. The state functions evaluate all outgoing transitions e 2 E and return a function pointer to the new state. A global execution system calls all actual state functions according to the current pointer values and stores the returned pointers as new state. Hence the execution system polls all state machines in a cycle. Due to the evaluation of the outgoing edges, the state functions examine the conditions of the guard for each edge. If a guard evaluates to true, the transition is taken. The function executes all commands in the action block and returns the pointer to the new state. To follow the idea of saving RAM, all variables to calculate the guards are defined globally in the C-program.

Function pointers have been chosen here due to the low execution memory requirements. A typical microcontroller mostly needs 2 bytes to store a function pointer. Thus, only 2 bytes are needed for each state machine to store the actual state. Typical implementation alternatives of the tools mentioned in section II are automata tables or state variable based switchcase constructs. Tables need a lot of RAM or program memory to be stored, which is a conflict to the requirements. State variables generate execution overhead for the state selection.

Special challenges are synchronisations, which are allowed by the UPPAAL specification. They are realised by the assignment of channels to the state machine transitions. Each transition can either receive from a single channel or send to a single channel. There may also be several transitions connected to a channel. If the guards of a sending and a receiving transition are enabled at the same time, both transitions may be executed.

To adopt this concept, a sending transition of a channel needs to check the guards of the receiving transitions before it can be executed. This is realised by the generation of dedicated *eval* () and *action* () functions for every synchronisation receiving transition of every channel.

A state function which evaluates a transition with a sending synchronisation firstly evaluates its own guard. If the guard is enabled all receiver transitions of the channel are evaluated by calling the dedicated *eval* () functions. If an *eval* () function evaluates to true, the synchronisation can be executed and the sending transition executes its own action block. Then it calls the *action* () function for the execution of the receiving transition actions. Additionally, the new state of the according automata has to be set. Hence, receiving transitions are not evaluated from their own state functions.

Sending transitions of broadcast channels do not need an enabled receiver for execution. They are executed, if the guard is enabled. The *eval* () and *action* () functions of receiving broadcast transitions are combined and are called every time a corresponding sender transition is taken.

UPPAAL specifications consist of template descriptions and parameterised instantiations. The instantiation of runtime objects, that are executed by the same code, requires some additional stack memory and function parameters. For the presented approach, a separate state machine and, thus, separated code is generated for each instance. Since the program memory of typical microcontrollers is not as limited as data memory, the usage of additional program memory is passable.

B. Temporal Behaviour

UPPAAL allows the definition of temporal conditions for states and transitions. In principle, each condition has the form *timerX <REL> value*, whereas <REL> can be a typical comparison operation like smaller (<), greater (>), and so on.

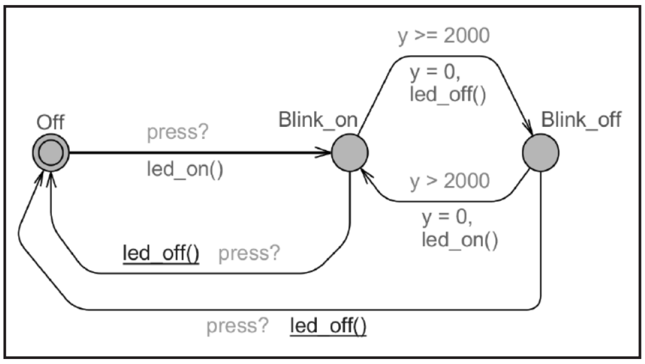

Figure 3 *Example of an UPPAAL specification. The generated C-code isgiven in listing 1.*

```
void*  state_off ( ) {
    if    (press_in ( ) )
    {
          led_on ( ) ;
          return BLINK_ON;
    }
    return NULL; }

void*  state_blinkon ( ) {
    if ( y > 2000)
    {
          y = 0 ;
          led_off ( ) ;
          return BLINK_OFF ;
    }
    if ( press_in ( ) )
    {
          led_off  ( ) ;
          return OFF ;
    } return 0 ; }

void*  state_blink_off ( ) { . . . }
```

Listing 1 *Generated implementation (partially) of the example state machine given in Fig. 3.*

Temporal conditions of state transitions can be interpreted like guards, whereas the *timerX* is an integer variable that changes in time. Hence, this kind of condition can be implemented as guard check too. The resetting operation of a timer variable is implemented as part of the transition evaluation.

The physical time base is taken from system timers. The relation between the given integer values of the UPPAAL specification and the integer timer values can be set by a multiplicator. The standard value of it is 1. Since the access to timers is target platform specific and cannot be generated in a common C-structure, an empty function body is generated. It has to be implemented by the developer manually.

Some of the temporal specification concepts in UPPAAL cannot be transformed - even not manually - to a sequential, time-discrete execution system, like a microcontroller. For instance, state based temporal conditions allow only the relation operation *timerX* < *value*. The semantics of this element is that the according state has definitely to be left before the time *value* is reached. Obviously, it cannot be assured in a sequential system. Even the usage of interrupts cannot guarantee the behaviour since it is possible to specify several state changes at exactly the same time.

Further not implementable concepts are committed and urgent locations or urgent channels. To assure a correct implementation of the specified system, the model is checked for these structures and rejected if they are found. Thus, implementable systems can be modelled by a subset of UPPAAL constructs.

C. System Integration

The timed automata specifications in UPPAAL are useful to describe the control flow of a system. The implementation of data processing is done in external C-functions. These are called by the timed automata in the action blocks.

The implementation of the automata itself, hence the realisation of the control flow, is generated by the presented code generator. It also generates C-header file for each state. If data processing is necessary, it has to be implemented by the developer. Obviously, the developer has to consider thememory restrictions of the platform too. There is no explicit need to generate separate headers for every instance since the data processing is mostly similar for all instances. However, an instance parameter was introduced. It allows different data processing implementations for the same states in various instances of state machines.

The other specification elements of UPPAAL, like arrays, constants, structs, or integer variables, are mapped strait forward to C-code.

Besides the implementation of the automata and the data processing, an execution framework is necessary. A basic execution system (main control loop, interrupt service routines or a thread) must call the automata execution function in a loop. A basic system function must return a current timestamp.

Most real world embedded systems react to environmental events by interrupts. The code generator only generates final state machines that poll internal and external values. The polling of external events is done in side effect free data processing functions called in the guards. These functions have to be implemented manually by the developer since they are usually target platform dependent. Interrupt service routines have to save their event information in variables, which are polled from the timed automata system.

5. Results

A code generator was implemented to prove the transformation concept. It transforms UPPAAL behaviour specifications to executable C-code. UPPAAL uses a XML format to save the specified systems. The code generator is a Java application that reads the XML description and parses the included Clike syntax of guards, action blocks and declaration parts. The generator front-end builds an internal representation of the complete specification. Afterwards, the instances of automata are generated step by step by the back-end. The generation bases upon the presented transformation rules.

In a first step, the characteristics of a generated system and a manual implementation had to be compared. The application is a little sensor network consisting of 2 nodes. A network packet is sent via a wireless transmission from node 1 to node 2. The time until node 1 received an answer packet from node 2 is measured. The complete sensor network node had to be modelled or implemented. It consists mainly of basic wireless network module interfaces, a network middleware, some basic system features (packet queues, memory allocator), a simple and self-organising routing protocol (NCR3), and the small application itself.

On one hand, the manual implementation is a traditional preemptive and thread-based C-program. It was programmed by an experienced developer with a special focus on data memory restrictions. On the other hand, the timed automata specification was modelled in UPPAAL and transformed to Ccode by the mentioned generator. 12 FSMs are necessary to describe the mentioned system.

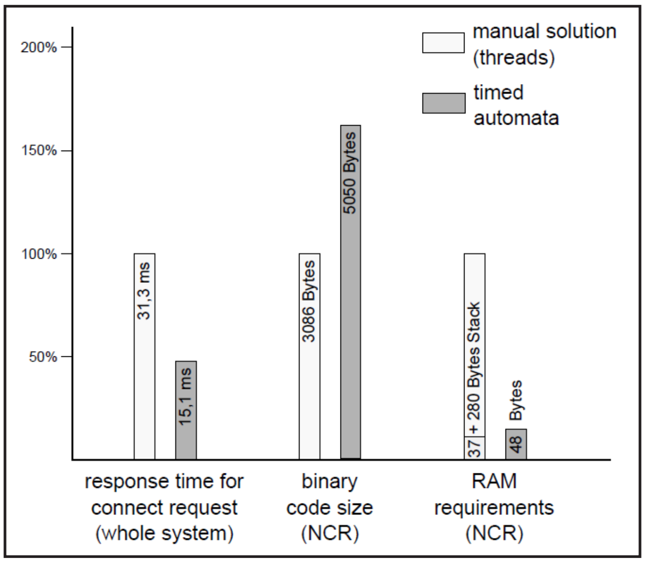

Figure 4 *Comparison of a manual implementation using preemptive threads and a timed automata based, generated system.*

Fig. 4 illustrates the comparison of key parameters of the 2 implementations. The parameters of the manual implementation are normalised to 100%. Obviously, the execution speed of the generated timed automata system is two times faster than the manual implementation. The reason is mainly the execution overhead of the scheduler based task switches and the scheduling order of the preemptive system. It heads to a higher variation of the response times. Another considerable advantage of the generated state based systems is the RAM requirement. Although the directly allocated memory of the preemptive system is a bit smaller, it needs an additional call stack for the runtime management of the threads. In opposite, the generated state based system only needs two bytes for a pointer to the actual state of each FSM and some additional global variables. Hence, the random access memory requirements are reduced by more than 80%. The manually implemented system needs some more requirements, mainly for the operating system. These are ignored and only the data memory usage of the network and routing algorithms is considered for this comparison. Queues and static arrays of the manual implemented data processing functions are ignored too.

In opposite to the data memory usage, the generated system needs 163% of program memory compared to the manual implementation. This downside is tolerable since the program memory of actual microcontrollers is typically not as restricted as the data memory.

While the first example concentrated on the static properties of the generated code, a second example shall deepen the runtime qualities. A comparison with Protothreads is suitable, since the compiler is available and the structures of the generated programs are similar. The analysed application is a packet transmission from a generating process passing 5 data conversion processes to a consuming process. The influences of the data conversion are kept small by only adding some constant integers to the packets.

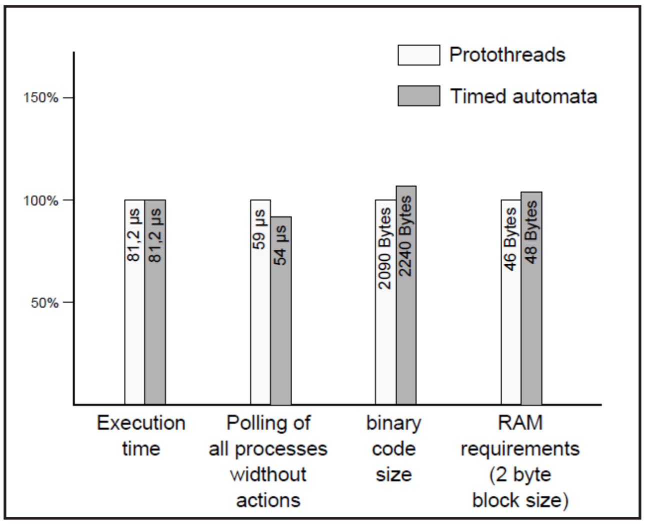

Figure 5 *Comparison of a protothread-based implementation and a timed automata based, generated system.*

In Fig. 5 the comparison between the protothread-based implementation and the timed automata based implementation Besides this comparison of quantifiable key parameters, the discussion of the usability of the presented work flow is necessary. Since the formal system specification by timed automata is not common, the primer question is what kind of algorithms can be described easily by these constructs. Obviously, controlflow driven systems are suitable for automata modelling. Dataflow driven applications can be modelled too but do not benefit from the proposed work flow, since all data manipulations are described in action blocks and not optimised by the code generator.

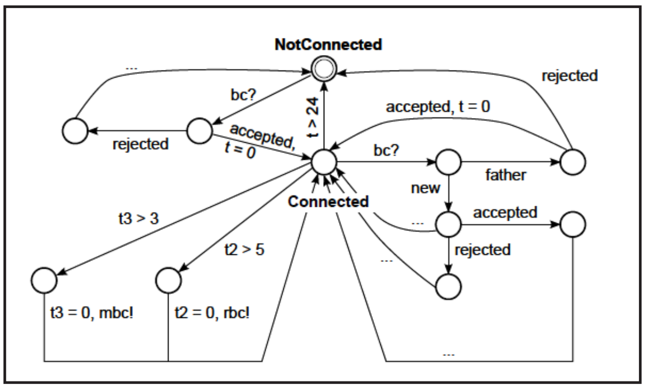

Figure 6 *Timed automata of the network connection and routing algorithm.*

Especially the implementation of networking protocols benefits. State changes and timing behaviour-like timeouts-can be easily modelled, simulated, and verified in UPPAAL and afterwards transferred to a C-based implementation by the code generator. For instance, Fig. 6 illustrates the timed automata of the network connection and routing algorithm presented in the first example of this section. The separation of timed control flow and external data processing led to a well structured system specification that is more intuitive than a pure code-based implementation.

Necessarily, the specification in UPPAAL is target platform independent. Methods to access dedicated hardware of a special target platform can neither be specified nor automatically implemented by the proposed approach. Hence, the implementation of low-level target depending systems, e.g. drivers, is not possible.

Summing up, especially the sensornet development seems to be predestined for the presented workflow. These kind of applications are network protocol driven and are usually executed on target platforms with restricted resources.

A final parameter for the success of the work flow is the developer acceptance. Automata based implementation approaches are known since several years but not widely spread yet. Possibly, the presented work flow with special focus on specification and implementation of temporal behaviour on low-resource devices may help to increase the acceptance.

6. Conclusion

An approach to transform parallel timed state machine specifications into executable source code is presented in this paper. The main application areas are resource constrained platforms, like embedded systems or wireless sensor networks.

The tool UPPAAL is used to define high level system specifications by timed automata. The specifications can be simulated and verified by the integrated tools. Based on its semantics, a code transformation concept to generate executable C-code from an UPPAAL specification is defined. This transformation concept has been implemented as a code generator. The complete work flow allows a closed development cycle starting with the formal specification of a system in UPPAAL and resulting in compilable C-Code. Given that the code generator is implemented error-free and the specified system is deterministic (and hence implementable), the behaviour of the generated systems is equal to the specified system. Independently, restrictions of a sequential execution system (processor) have to be considered.

The quality of the concept and the generator have been tested and compared to other approaches. Therefore, the complete behaviour of a sensor network node has been specified as set of timed automata in UPPAAL. The first experiences are that this modelling paradigm is especially well structured and comprehensible for control-complex algorithms. The presented measurements show that the approach has significant advantages in execution time and memory requirements compared to classical preemptive thread-based solutions. Compared to a very simple parallelism abstraction, called Protothreads, the parameters are nearly similar.

The code generator is implemented as prototype and offers some points for improvement. In the parsing of the C-like parts of the UPPAAL specification, some completions are required, due to recognise the whole syntax of the language. Additionally, work for the generation of cyclic timed variables has to be done. Some optimisations in the synchronisation transformation have to be evaluated, e.g. a sender / receiver exchange at receiver dominated specifications.

Further research has to be done on detecting nondeterministic UPPAAL specifications to guarantee equivalence between verification results and the generated system behaviour. It should also be part of future work to check the implementation alternatives of different temporal concepts, like state based temporal conditions.

7. References

[1] R. Alur and D. L. Dill, “A theory of timed automata,” *Theor. Comput. Sci.*, vol. 126, no. 2, pp. 183–235, 1994.

[2] G. Behrmann, A. David, and K. G. Larsen, “A tutorial on UPPAAL,” *in Formal Methods for the Design of Real-Time Systems: 4th International School on Formal Methods for the Design of Computer, Communication, and Software Systems, SFM-RT* 2004, ser. LNCS, M. Bernardo and F. Corradini, Eds., no. 3185. Springer–Verlag, pp. 200–236, September 2004.

[3] S. Yovine, “Kronos: a verification tool for real-time systems,” *in International Journal on Software Tools for Technology Transfer (STTT)*. Berlin, Heidelberg: Springer-Verlag, pp. 123–133, 1997.

[4] J. Hill, R. Szewczyk, A. Woo, S. Hollar, D. Culler, and K. Pister, “System architecture directions for networked sensors,” *SIGPLAN Not.*, vol. 35,

no. 11, pp. 93–104, 2000.

[5] S. Bhatti, J. Carlson, H. Dai, J. Deng, J. Rose, A. Sheth, B. Shucker, C. Gruenwald, A. Torgerson, and R. Han, “Mantis os: an embedded multithreaded operating system for wireless micro sensor platforms,” *Mob. Netw. Appl.*, vol. 10, no. 4, pp. 563–579, 2005.

[6] A. Dunkels, O. Schmidt, and T. Voigt, “Using Protothreads for Sensor Node Programming,” *in Proceedings of the REALWSN’05 Workshop on Real-World Wireless Sensor Networks*, Stockholm, Sweden, Jun. 2005. [Online]. Available: http://www.sics.se/adam/dunkels05using.pdf

[7] O. Kasten, “A state-based programming model for wireless sensor networks,” Dissertation, Swiss Federal Institute of Technology Zurich (ETH Zurich), 2007.

[8] T. Amnell, E. Fersman, L. Mokrushin, P. Pettersson, and W. Yi, “Times: A tool for modelling and implementation of embedded systems,” *in Proc. of the 8th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, ser. Lecture Notes in Computer Science, J.-P. Katoen and P. Stevens, Eds., no. 2280. Springer–Verlag, pp. 460–464, 2002.