Back to the Roots: Implementing the RTOS as a Specialized State Machine

Christian Dietrich, Martin Hoffmann, Daniel Lohmann
Department of Computer Science 4 - Distributed Systems and Operating Systems
Friedrich-Alexander University Erlangen-Nuremberg
{dietrich,hoffmann,lohmann}@cs.fau.de

***Abstract*— Real-time control systems, originally arisen from simple, state-machine–based discrete elements, nowadays comprise sophisticated and manifold software-based algorithms consolidated with different applications on single, yet powerful microcontrollers. Real-time operating systems were introduced to handle this complexity by providing APIs to describe the desired system behavior, however, at the cost of losing the clarity and explicitness of state-machine–based representations.**
This paper presents an approach to bring the RTOS back to the roots of a hardware-implementable finite state machine. The concept is based on a detailed static analysis of the application–kernel interaction to distill the real-time operating system behavior and find a FSM-based representation of the expected OS states and transitions. We apply our idea to a realistic control application based on an OSEK operating system, which results in a feasibly sized programmable logic array implementation. Having such a representation at hand might further leverage thorough system verification and validation based on existing and mature FSM analysis tools.

I. Introduction

Up to twenty-five years ago, embedded real-time control systems were typically designed by electrical engineers as *finite state machines* (FSMs) out of discrete elements. With the advent of cheap 4-bit and 8-bit microcontrollers, software has begun to take over the role of wiring discrete elements, but the paradigm of implementing control systems as FSMs remained. In comparison, the employment of a full-blown *real-time operating system* (RTOS) as underlying system software is a relatively young trend, triggered by the increasing complexity of control applications and the necessity of hardware consolidation. This is not always warmly welcomed by control-system engineers [18, 15], which is understandable, as the simple FSM paradigm has had some clear advantages: It is well understood (especially by certification authorities) and there is a large body of formal methods, heuristics, and tools available for optimization and validation, which leads to highly specialized, efficient implementations with low hardware requirements. On the other hand, employing an RTOS and its concepts (e.g., tasks, events, resources) can significantly ease the development of more complex control applications.

In this paper, we explore the possibility to get the best of both worlds: The idea is, to keep the RTOS interface for application development, but implement the *RTOS itself* (or more precisely: its concrete instance) as a FSM. Thereby, it becomes possible to use existing FSM-based analysis and validation tools (also) on the RTOS – or to push the RTOS completely "back into hardware" for perfect isolation.

This work was partly supported by the German Research Foundation (DFG) under grants no. LO 1719/1-3 (SPP 1500) and SCHR 603/9-1

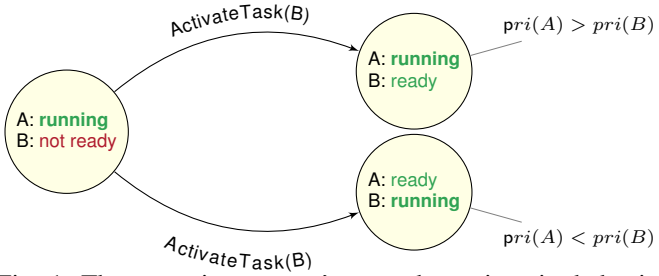

Fig. 1: The operating system's state determines its behavior. On system-call events, the OS changes this internal state.

A. Our Idea in a Nutshell

In theory, every computing system could be modelled as a FSM. This also holds for the RTOS: Every syscall, triggered synchronously by the application or asynchronously by an interrupt, can be considered as a transition on the OS-internal state (such as the ready list). The problem, however, is state explosion, caused by complex states and indeterminism in the control flow: Every syscall is a potential point of rescheduling at which, depending on the dynamic state of the ready list, some other task may be selected to continue execution.

The core idea of our approach is to reduce such indeterminism as far as possible at compile time: We exploit static knowledge about the RTOS configuration and *its semantics* in combination with a whole-system analysis across all control flows of the application to figure how the RTOS is actually used. Thereby, we derive a model on how the concrete application interacts with the kernel. We replace parts of the traditional OS implementation by an implementation of the derived model and (partially) specialize each syscall in the application at caller side to interact with the model.

The possible transitions on the kernel's state (such as the outcome of a scheduling decision) can thereby be greatly reduced at compile time, in many cases even to exactly one: If for instance, some task `A` triggers another task `B` for execution (`ActivateTask(B)`), this is a potential point of rescheduling. In a strictly priority-based system, however, the result can be reduced (by considering the scheduler semantics) to exactly two possible follow-up states: Depending on the relative priorities of `A` and `B`, either `A` is running and `B` is set ready (as shown in Figure 1) or vice versa. If we can further determine their priorities by static analysis, the effective result of this concrete syscall invocation can be reduced to exactly one follow-up state.

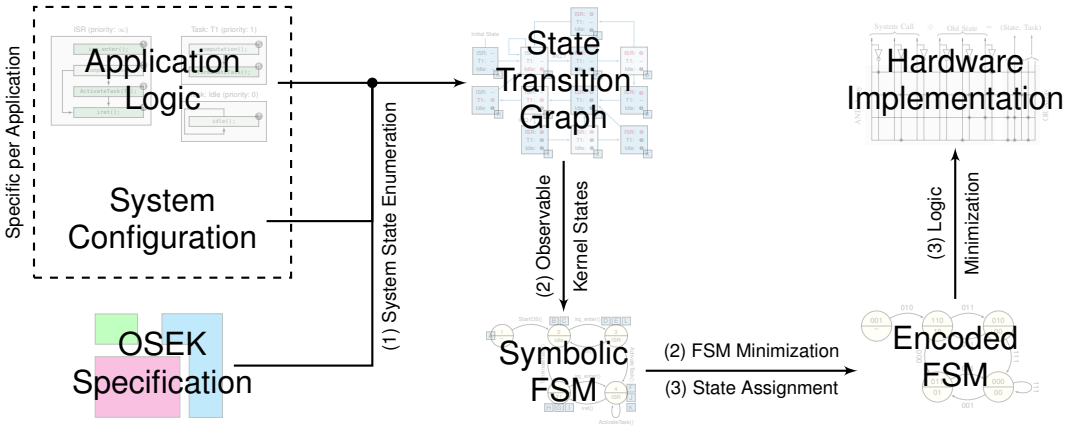

Fig. 2: Methodic Overview. From the general OSEK specification, and one concrete application, we generate a specialized OS implementation in several steps.

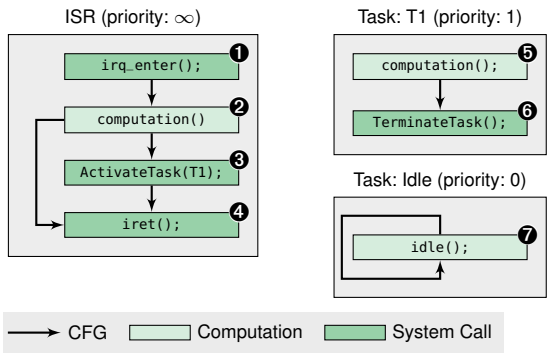

Fig. 3: Application Logic of a small (complete) OSEK System

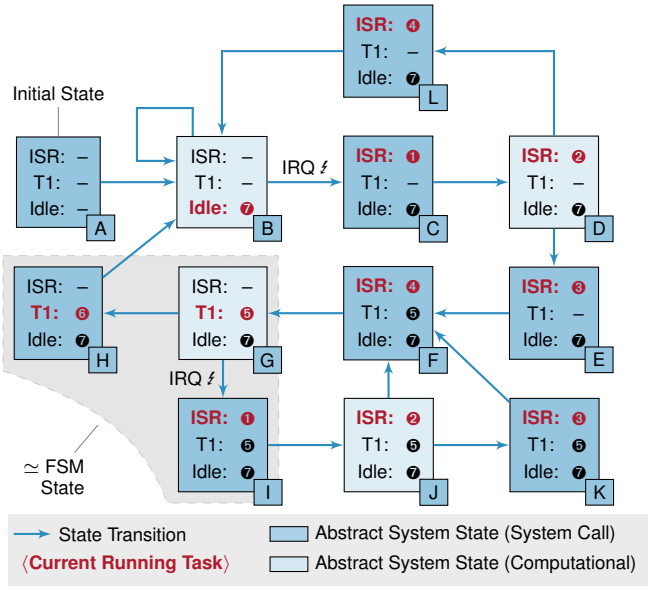

Fig. 4: State-Transition Graph for Figure 3. Each node is an abstract representation of the system at a one point in time.

Of course, in real-world systems, not all kernel interactions can be reduced that easily – especially interrupt-based alarms are a significant source of indeterminism. Nevertheless, our results show that the resulting state reduction makes it still feasible to generate the RTOS instance as a simple FSM.

B. Structure of the Paper

We apply our idea to the OSEK [13] / AUTOSAR [1] standards employed in the automotive industry. The RTOS included in these standards is an event-triggered, priority-driven, preemptive kernel. Its static configuration includes the number of tasks, their priority, the events they can wait for, and the resources they synchronize on using a static stack-based priority ceiling protocol. Without loss of generality, we choose OSEK as the running example throughout the paper.

In Figure 3, an example OSEK application is shown. It consists of one ISR, one normal task, and the idle loop. On an *interrupt request* (IRQ), the ISR may or may not activate the task. After the task finished its execution, it terminates and the OS executes the idle loop until the next IRQ occurs. Based on this example, the following Section II presents the static analysis and FSM construction. Finally, we provide first preliminary results on applying our concept to a realistic application scenario, and discuss further possible use cases.

II. Implementation

We divide our approach into three distinguishable parts: (1) The extraction of fine-grained interaction knowledge from the application. (2) The transformation to an executable model of the operating system. (3) The concrete implementation of the executable model. Figure 2 depicts the information flow of all three stages. With the *system-state enumeration* (SSE), we extract the interaction as a *state-transition graph* (STG) that enumerates all possible system states and their execution sequences. We identify all visible kernel states and construct a (minimized) FSM. As one possible implementation, we assign binary vectors for inputs, states, and outputs of the FSM and encode the minimized truth table as *programmable logic array* (PLA) simulation in software. In the following, we will investigate these steps in a greater detail.

A. System-State Enumeration

In the first step, we statically analyze the interaction of a given application with an abstract OSEK operating system. We already described this extraction step in previous work [5]. Therefore, we outline the *system-state enumeration* (SSE) mechanism only briefly and focus on the extracted fine-grained interaction knowledge, which is expressed as a STG.

The system-state enumeration combines three different sources of information in a forward simulation of the system: First, the *system semantics*, as defined by the OSEK specification [13]. Second, the *system configuration*, as declared in a domain-specific configuration language (OIL). And, third, the application logic, which is extracted from the control-flow graphs of the compiled application. The configuration already contains coarse-grained information about the system, like the set of tasks and their priorities. Together with the system semantics, we calculate fine-grained knowledge to predict the operating system's decisions in presence of the given application logic.

The SSE discriminates two block archetypes in the application: *computation* and *system-call blocks*. In computation blocks, the application does not issue system calls and therefore the OS state cannot be changed synchronously. Nevertheless, IRQs can *only* occur in computation blocks, and are modeled as asynchronous activation of ISR proxy tasks. The other block archetype contains system calls, which interact with the kernel synchronously and modify its state.

The central data structure for the SSE is the *abstract system state* (AbSS), which captures information about a system at a given point in time. For each task, an AbSS includes the ready flag, the current priority, and which block should be executed next in a task's context. Except the initial state, each AbSS has one task marked as the currently running task. In Figure 4, each node represents a simplified AbSS for the example system from Figure 3. For each task (interrupt-service routines and idle task included), the node contains the blocks to be executed next, while the currently active task is highlighted.

The SSE discovers all possible AbSSs for the given application, by repeated application of a `systemSemantic()` function on already discovered states until no new states appear. This transition function evaluates the block of the currently running task, calculates the block's influence on the current system state, and emits one or more follow-up states. For example, in Figure 4 only AbSS H executes block ❻ next. Since block ❻ contains a `TerminateTask()` system call, the transition function emits one follow-up state B with `T1` marked as *not-ready*. Furthermore, the transition function applies the OSEK scheduling rules and marks the idle task as *running*. All discovered AbSSs and their follow-up states are connected in the *state-transition graph* (STG).

interrupt-service routines (ISRs) are modeled with proxy tasks, which are assigned the highest possible priority and are executed under interrupt blockade. They are activated by the transition function within computation blocks. In Figure 4, the idle state B has two follow-up states: first, a self loop, since it is its own CFG successor. Secondly, the idle state can proceed to state C. This transition is the result of a virtual IRQ and the ISR entry block ❶ will be executed next.

The STG contains all possible state–state transitions for the given application. Depending on the application and its structure, it can become very large, but remains always finite. It is important to note, that each AbSS in the graph represents the system immediately *before* a block is executed. For a more detailed discussion on the SSE and mechanisms to ease the state explosion we refer to our previous work [5].

B. Kernel-Visible System States

As desired, the STG subsumes the application's control flow, as well as the kernel's scheduling decisions. We aim to implement only the OS' behavior. Therefore, we have to separate state transitions into application transitions and OS transitions. The application transitions are implemented by the application itself, in terms of branches, loops, and function calls. They are executed directly by the processor. Our specialized kernel should only implement the OS transitions, since only those are dictated by the OSEK specification.

As already said, each state represents the system right before a certain block is executed. Some states execute a computation block next, some a system-call block. Only the latter ones, *system-call states*, will ever be visible to an OS implementation. Therefore we partition all AbSSs in the STG into *regions of states* which are indistinguishable from the kernel's perspective. These regions are connected subgraphs within the STG; system-call states can only occur as leaf nodes in a region. In Figure 4, the states G, H, and I form such an region. This region cannot be extended to AbSS F, since F is a system-call state and must, therefore, be a leaf node in a different region.

These regions are constructed by repeated merging of initial minimal regions: Initially, each AbSS is located in its own region. For each state in a region, we merge the successor regions into the region, if the originating state is computational. Furthermore, we merge a predecessor region, if the preceeding state is a computational state. This process is repeated until no further changes happen.

With this construction, all states with a successor outside their region are system-call states. Since the OS state is only modified at the region's border, all inner states, which are computational, have the same task marked as running.

From these regions, we construct the initial *finite state machine* (FSM) for the kernel: Each region corresponds to a state in the FSM. An FSM transition from state A to state B is present, if a system-call state in region A can proceed to region B. The input event for this transition is the execution of the system-call block. Each FSM state exposes the currently running task as an output. It is noteworthy, that each system-call block results in a different FSM input signal, even if they invoke the same system service.

Figure 5 shows the resulting state machine with the AbSS regions drawn next to each FSM state. The constructed FSM matches the observation that an OS is a FSM with system calls as inputs and the currently scheduled task as output. In our construction, the resulting FSM is a Moore machine.

C. State-Machine Minimization

The resulting FSM already exhibits the required kernel transitions when triggered by external events and system calls.

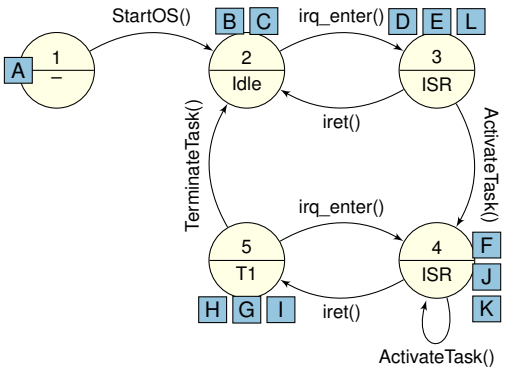

Fig. 5: Symbolic Finite State Machine with abstract system state next to each state.

Nevertheless, the number of states and transition edges is not minimal yet. Minimization of state machines is a well covered and long standing topic [12, 8]. Therefore, we will only investigate on the specifics for our operating-system FSM.

For the minimization of FSMs, states are grouped into *equivalence classes* (ECs), where each state within exposes the same observable behavior. From each equivalence class, a new state in the minimized FSM is generated, and transitions are added accordingly to the EC connections.

Our FSM is not an acceptor for a formal language. Furthermore, we are allowed to remove triggers from the system by wiping out system-call sites. We only have to ensure that the scheduling sequence remains the same. Therefore, we adapt the EC construction to fit these requirements.

First, we demand that each state in an EC results in the same *current running task*. Furthermore, the set of possible follow-up ECs must be equal for all states within an EC. The follow-up ECs of an state are those ECs which are reachable in the FSM when following the transitions. We used an adapted Moore algorithm [12] to find the most coarse EC partition of the FSM which fits both requirements.

In the minimized FSM, many transitions are self loops. If all transitions that are triggered by one system-call block are self loops, we wipe out the system-call site. The specific system-call signal never transfers the system into an observable different state; it is useless for our implementation. In the example (see Figure 5), the FSM is already minimal after its construction, but in general the size of FSM decreases significantly. With the FSM minimization, we have completed the construction of the executable model.

D. State Assignment and Logic Minimization

The last step is the implementation of the executable model and its linking to the application. The possibilities to implement the calculated FSM are endless. We chose to present an approach directed towards an OS implementation that fully resides in hardware. This would result in a specialized OSEK implemented as a processor extension.

However, while this is still a topic of further research, we currently provide a software simulated programmable logic array implementation of the generated FSM. While dispatching,

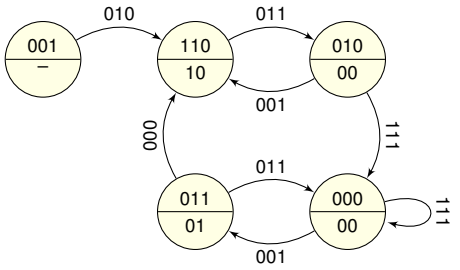

Fig. 6: Finite State Machine with Assigned Binary Vectors for Inputs, Outputs, and State Encodings

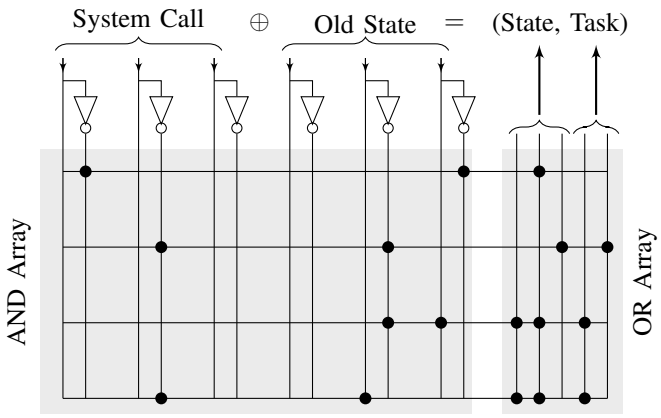

Fig. 7: Implementation as Programmable Logic Array

interrupt handling, and timer control is still implemented traditionally, the OS logic is already suited for a hardware implementation.

One main challenge of implementing a FSM in hardware is the selection of bit vectors for inputs, states, and output signals. This encoding largely influences the minimal required complexity of the hardware implementation. Luckily, many methods were already proposed to solve this problem for different hardware designs [17, 4, 16].

We decided to use the NOVA program [17] to choose the encoding of our FSMs. The driving factor of this decision was the availability of the NOVA source code. NOVA targets optimal encoding for two-level logic implementations. NOVA chooses input and state encoding for our FSM, while we choose the output encoding arbitrarily. The result of the assignment process is shown in Figure 6.

From the FSM and the encoding of inputs, states, and outputs, we generate a truth table with one line for each transition. Each line consists of the input word, the current state, the next state, and the desired output. To achieve an efficient implementation of this truth table in hardware, we use the ESPRESSO [2] heuristic logic minimizer.

From the minimization result, a PLA implementation can be derived in hardware. Figure 7 shows the final OS execution model for our running example. The resulting component takes the current system-call–block number and the saved system state as inputs. Each line in the AND array checks a certain

```
TerminateTask() called from T1
disable_interrupts();                                    ❹
OS_state, task = fsm_step(0b000, OS_state);
switch_to(task);
// Never returns, IRQ enable in next task
```

Fig. 8: `TerminateTask()` implementation called from T1.

TABLE I: Size of graphs, machines, and implementation after each step for the *I4Copter* task setup (11 tasks, 3 alarms, 1 ISR, 1 Resource).

Step		w/o Ann.	w/ Ann.
State-Transition Graph	[S(T)]	1,563,169 (2,098,236)	20,063 (23,876)
Symbolic FSM	[S(T)]	407,530 (942,597)	6,242 (10,055)
Minimized FSM	[S(T)]	2,938 (8,822)	667 (1,212)
Two-Level Logic	[AND Terms]	5,144	728
Software PLA Table	[Bytes]	35,798	4,566

bit pattern and emits a logic `1`, if the pattern matches. The OR array decides which outputs of the AND array will enable a bit in the output word. In our case, the output word consists of a new FSM state and the currently running task.

In our current implementation, we simulate this PLA in software by iterating over all lines in the ESPRESSO output. We use the task output word as an input for the dispatcher.

We replace every *system-call site* with a specialized code fragment that calls the FSM. Figure 8 exemplifies the implementation of the system-call block ❹. The `fsm_step()` function contains the PLA simulation, while the bitstring `000` identifies the call location exactly.

III. Preliminary Results

Currently, we do *not* produce hardware components from the execution model, but use a (slow) PLA software simulation. Therefore, we will only show some preliminary results for a realistic scenario to give an impression of the general feasibility.

We implemented the presented approach for the *d*OSEK [7] system generator[1]. As evaluation scenario, we use a realistic real-time workload. We revive a setup, already presented in previous work [6], resembling a real-world safety-critical embedded system in terms of a quadrotor helicopter control application. The scenario consists of 11 tasks, which are activated either periodically or sporadically by an interrupt. In total, 4 asynchronous events can trigger within computation blocks. Inter-task synchronization is done with OSEK resources and a watchdog task observes the remote control communication.

In the first column of Table I, the sizes of the system at different steps is given. While the STG has more than 1.5 million states and 2 million transitions, the (unminimized) FSM already reduces the size significantly. The minimization of the FSM removes 99.28 percent of the internal states. The state assignment and the logic minimization achieve a implementation of the execution model with 5,144 AND terms (rows in the PLA). In our software implementation, the minimized truth table occupies 35,798 bytes of read-only memory, while the implemented FSM requires 4 bytes of volatile memory for storing the current state.

In the second column of Table I, we show the results for the same system, but with additional annotations for the SSE analysis. We declared four task groups. Each group handles a different job in the system, which is released through an external signal (alarm or IRQ). The annotation forbids the retrigger of the signal while not all tasks of a group have finished their execution. This annotation is a qualitative statement that the deadline of the job is smaller than its period. This qualitative statement, which has to be supplied by the real-time developer, was already described in previous work [5].

With the annotation, the system has a 98.72 percent smaller STG, which, of course, was the intention of the annotation in the first place. Surprisingly, the state count of the minimized FSM shrinks only by 77.3 percent with annotations. This smaller decrease factor indicates an unnecessary edge redundancy in the STG without annotations.

IV. Discussion

In this paper, we derive an OS instance specifically tailored towards a given application. We used the OSEK API as a markup language to annotate the desired task orchestration and interaction. When we perceive the system configuration and placement of system calls as the abstract intentions of the real-time engineer, we can switch our focus from the traditional way of implementing the specification, to realizing only the developer's intended behavior. Encoding the minimized FSM in hardware is only one of many possible options. More importantly, this demonstrates the expressive power of the STG and the various FSMs as immediate representations of the system. Furthermore, pushing the OS logic fully into the hardware, we achieve perfect isolation. Not a single instruction would be needed for the OS execution. Only special opcodes would be reserved for giving inputs to the hard-coded FSM.

Apart from that, a FSM representation is not only useful for implementing the desired OS logic, but can also be used as watchdog for an off-the-shelf OSEK system. Fed with the same inputs, the actual OS must expose the same behavior. Combined with a WCET-based intrusion detection [19], an effective security scheme could be derived from static analysis of the system behavior.

Besides implementing the system behavior, the immediate representations make the actual kernel behavior accessible to other tools: The minimized FSM representation can be used to test whether the behavior of one real-time system is equivalent to or partially embedded in another system.

Our immediate representations may also assist the verification of tailored OS implementations: If we prove the equivalence of STG (or FSMs) to the OSEK standard for a certain application, and furthermore show the equivalence of the actual implementation to the STG, we get an OSEK implementation that is verified for a certain application; even in the presence of extensive system tailoring.

V. Related Work

The RTSC [14] that significantly inspired this work also uses the OSEK API as markup language to annotate the desired real-

[1]Code is released as free software at https://github.com/danceos/dosek

time behavior. It translates the system from an event-triggered to a table-driven, time-triggered system. Unlike our approach, their immediate representation is flow insensitive.

Chen and Aoki [3] use a formal model of OSEK and model checking techniques to automatically generate test cases for OSEK/OS. Their approach does not incorporate information about the configuration or the inner structure of a specific application, but emits whole applications as test-cases. Our application specific FSM could be used to generate application-specific event sequences to test the application, as well as the kernel.

In the sensor-network community, state machines are recognized as mean to compactly implement application and control logic. Kim and Hong [9] proposed state machines as well-suited paradigm for sensor nodes. Their *SenOS* kernel is an executor for transition tables, where each task comes with its own table. In contrast to our approach, the tables are not derived automatically.

Kothari, Millstein, and Govindan [10] proposed an automatic derivation of FSMs from TinyOS applications through symbolic execution. They derived "user-readable FSMs" in order to make the application logic more comprehensible to developers. As they state, their interrupt semantic is incomplete. Additionally, TinyOS has a simpler execution model than OSEK, since tasks have no wait states and only execute in a run-to-completion manner. Also, all their inferred FSMs do not exceed 16 states.

There are many projects implementing parts of the (or the whole) operating system in hardware. As one example, the ReconOS project [11] extends the multithreaded programming model across the hardware/software boundary. ReconOS provides a unified synchronization and communication API for hardware, which is executed on an FPGA, and software threads. Nevertheless, ReconOS is not tailored explicitly to fine-grained application logic, but mimics a generic, POSIX-like, interface.

VI. Conclusion

Many years of embedded real-time control engineering piled more and more abstraction layers on top of each other to ease the development process at the cost of complex software stacks and operating systems. In this paper, we presented an approach to descend these layers from an abstract RTOS-based control application back to the roots of an FSM-based PLA. Preliminary results already show the feasibility of our approach on the example of a realistic real-time application. Distilling the RTOS behavior not only allows to push it back into hardware, but might also leverage profound verification and validation of the system as a whole.

References

[1] AUTOSAR. *Specification of Operating System (Version 5.0.0)*. Tech. rep. Automotive Open System Architecture GbR, Nov. 2011.

[2] Robert King Brayton, Alberto L. Sangiovanni-Vincentelli, Curtis T. McMullen, and Gary D. Hachtel. *Logic Minimization Algorithms for VLSI Synthesis*. Norwell, MA, USA: Kluwer Academic Publishers, 1984. ISBN: 0898381649.

[3] Jiang Chen and Toshiaki Aoki. "Conformance Testing for OSEK/VDX Operating System Using Model Checking". In: *18th Asia-Pacific Software Engineering Conference (APSEC 2011)*. (Ho Chi Minh). Los Alamitos, CA, USA: IEEE, Dec. 2011, pp. 274 –281. ISBN: 978-1-4577-2199-1. DOI: 10.1109/APSEC.2011.26.

[4] S. Devadas, Hi-Keung Ma, A.R. Newton, and A. Sangiovanni-Vincentelli. "MUSTANG: state assignment of finite state machines targeting multilevel logic implementations". In: *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on* 7.12 (Dec. 1988), pp. 1290–1300. ISSN: 0278-0070. DOI: 10.1109/43.16807.

[5] Christian Dietrich, Martin Hoffmann, and Daniel Lohmann. "Cross-Kernel Control-Flow-Graph Analysis for Event-Driven Real-Time Systems". In: *2015 ACM SIGPLAN/SIGBED Conf. on Languages, Compilers and Tools for Embedded Systems (LCTES '15)*. (Portland, Oregon, USA). New York, NY, USA: ACM, June 2015. DOI: 10.1145/2670529.2754963.

[6] Martin Hoffmann, Christoph Borchert, Christian Dietrich, Horst Schirmeier, Rüdiger Kapitza, Olaf Spinczyk, and Daniel Lohmann. "Effectiveness of Fault Detection Mechanisms in Static and Dynamic Operating System Designs". In: *17th IEEE Int. Symp. on OO Real-Time Distributed Computing (ISORC '14)*. (Reno, Nevada, USA). IEEE, 2014, pp. 230–237. DOI: 10.1109/ISORC.2014.26.

[7] Martin Hoffmann, Florian Lukas, Christian Dietrich, and Daniel Lohmann. "dOSEK: The Design and Implementation of a Dependability-Oriented Static Embedded Kernel". In: *21st IEEE Int. Symp. on Real-Time and Embedded Technology and Applications (RTAS '15)*. Washington, DC, USA: IEEE, 2015.

[8] John Hopcroft. *An $n \log n$ algorithm for minimizing states in a finite automaton*. Tech. rep. Computer Science Department, University of California, 1971.

[9] Tae-Hyung Kim and Seongsoo Hong. "State Machine Based Operating System Architecture for Wireless Sensor Networks". In: *Parallel and Distributed Computing: Applications and Technologies*. Ed. by Kim-Meow Liew, Hong Shen, Simon See, Wentong Cai, Pingzhi Fan, and Susumu Horiguchi. Vol. 3320. LNCS. Springer Berlin Heidelberg, 2005, pp. 803–806. ISBN: 978-3-540-24013-6. DOI: 10.1007/978-3-540-30501-9_158.

[10] Nupur Kothari, Todd Millstein, and Ramesh Govindan. "Deriving State Machines from TinyOS Programs Using Symbolic Execution". In: *IPSN '08: 7th Int. Conf. on Information Processing in Sensor Networks*. Washington, DC, USA: IEEE, 2008, pp. 271–282. ISBN: 978-0-7695-3157-1. DOI: 10.1109/IPSN.2008.62.

[11] Enno Lübbers and Marco Platzner. "ReconOS: Multithreaded Programming for Reconfigurable Computers". In: *ACM Trans. on Embedded Computing Systems (TECS)* 9.1 (Oct. 2009), 8:1–8:33. ISSN: 1539-9087. DOI: 10.1145/1596532.1596540.

[12] Edward F. Moore. "Gedanken-experiments on sequential machines". In: *Automata studies*. Annals of mathematics studies, no. 34. Princeton University Press, Princeton, N. J., 1956, pp. 129–153.

[13] OSEK/VDX Group. *Operating System Specification 2.2.3*. Tech. rep. http://portal.osek-vdx.org/files/pdf/specs/os223.pdf, visited 2014-09-29. OSEK/VDX Group, Feb. 2005.

[14] Fabian Scheler and Wolfgang Schröder-Preikschat. "The RTSC: Leveraging the Migration from Event-Triggered to Time-Triggered Systems". In: *13th IEEE Int. Symp. on OO Real-Time Distributed Computing (ISORC '10)*. (Carmona, Spain). Washington, DC, USA: IEEE, May 2010, pp. 34–41. ISBN: 978-0-7695-4037-5. DOI: 10.1109/ISORC.2010.11.

[15] Jim Turley. "Operating Systems on the Rise". In: *embedded.com* (June 2006). http://www.eetimes.com/author.asp?section_id=36&doc_id=1287524. URL: http://www.eetimes.com/author.asp?section_id=36\&doc_id=1287524.

[16] D. Varma and E.A. Trachtenberg. "A fast algorithm for the optimal state assignment of large finite state machines". In: *Computer-Aided Design, 1988. ICCAD-88. Digest of Technical Papers., IEEE International Conference on*. Nov. 1988, pp. 152–155. DOI: 10.1109/ICCAD.1988.122483.

[17] T. Villa and A. Sangiovanni-Vincentelli. "NOVA: State Assignment of Finite State Machines for Optimal Two-level Logic Implementations". In: *26th ACM/IEEE Design Automation Conference*. (Las Vegas, Nevada, USA). DAC '89. New York, NY, USA: ACM, 1989, pp. 327–332. ISBN: 0-89791-310-8. DOI: 10.1145/74382.74437.

[18] Collin Walls. *The Perfect RTOS*. Keynote at embedded world '04, Nuremberg, Germany. 2004.

[19] Christopher Zimmer, Balasubramanya Bhat, Frank Mueller, and Sibin Mohan. "Time-based Intrusion Detection in Cyber-physical Systems". In: *1st ACM/IEEEInt. Conf. on Cyber-Physical Systems*. ICCPS '10. Stockholm, Sweden: ACM, 2010, pp. 109–118. ISBN: 978-1-4503-0066-7. DOI: 10.1145/1795194.1795210.