# RTOS-Independent Interaction Analysis in ARA

Gerion Entrup, Jan Neugebauer, Daniel Lohmann
Leibniz Universität Hannover
{entrup, lohmann}@sra.uni-hannover.de, jan.neugebauer@stud.uni-hannover.de

***Abstract*—ARA is an RTOS-aware whole-system compiler for embedded applications that takes RTOS semantics into account for interprocedural analysis and optimization. To be applicable for a multitude of RTOS interfaces and semantics, ARAs analysis steps shall operate on an abstract RTOS model as far as possible, while still providing means to exploit OS-specific particularities. In this paper, we describe the design of such a model and its utilization with two static analysis algorithms for AUTOSAR, FreeRTOS, Zephyr and a subset of POSIX.**

## I. Introduction

Embedded systems typically come as whole systems: All code that will eventually run on the device is known in advance. For the compilation process, this lays the foundation for interprocedural whole-system optimization, which is well-explored on the language level [15], [23], [18]. Taking also the real-time operating system (OS) into account [21], [2], [8] enables further aggressive optimizations by tailoring the OS to the actual application implementation.

One approach to achieve whole-system optimization is model-based generation, that is, generating the system including the application from an abstract language description [29], [1]. However, in practice, embedded applications are written against a classical system-call interface, employing OSs such as FreeRTOS or Zephyr mainly as a helper library or markup-language to describe event- and control-flow interactions at run time.

With the Automatic Real-time System Analyzer (ARA)[1], we are building a whole-system compiler (based on LLVM) for embedded systems written against such common OS interfaces. ARA is then able to compile the application with additional optimizations based on the actual interactions between the OS and the application code. Examples include the transformation of dynamic into static initialization [13], folding of pre-known scheduling decisions [9], or elision of never taken locks in multi-core settings (not yet published).

For genericity, ARA shall support multiple OSs without having to change the underlying analysis algorithms. However, these analyses need, by design, OS-specific knowledge in some parts. We, therefore, split them into an OS-agnostic core and summarize all OS-specific parts in an analysis independent *OS model* that serves as a unified interface between all algorithms and OSs. Figure 1 visualizes the separation and an overview of the ARA toolchain (we present details in Section III).

This work was partly supported by the German Research Foundation (DFG) under grant no. LO 1719/4-1

[1]https://github.com/luhsra/ara

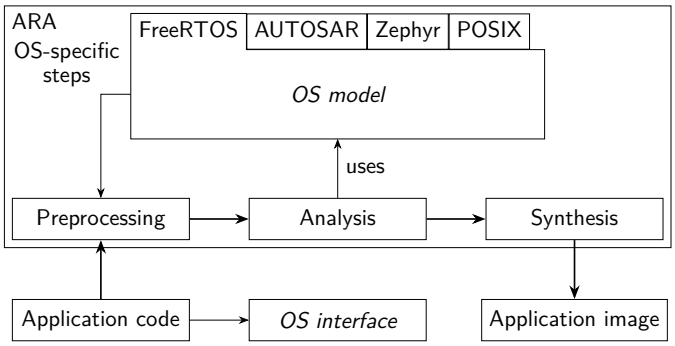


Fig. 1: Overview over the ARA toolchain. ARA processes the application code with various steps and finally emits an image. The model encapsulate all OS specific knowledge. It (optionally) triggers OS-specific preprocessing steps and is used by the main analyses for syscall interpretation.

In this paper, we describe our findings from designing this model and implementing it for four very different syscall interfaces: AUTOSAR, Zephyr, FreeRTOS, and a subset of POSIX. In particular, we claim the following contributions:

- An abstract OS model with implementations for FreeRTOS, AUTOSAR, Zephyr, and POSIX.
- The possibility to analyze and (partly) optimize real-world systems for all these OSs.

## II. System Model and Implementation

For our OS model, we target embedded real-time systems. The concrete OS semantics and types of system objects have no further constraints. However, the model requires that all communication between application and OS takes place via explicit syscalls or interrupts.

The current implementation in ARA imposes some further constraints: As a toolset for static analysis and optimization, its algorithms rely on a closed-world assumption, that is, all code is known in advance. Late binding via function pointers is supported and sound, but excessive use may impact the strictness of analysis results. We furthermore assume a defined application starting point, which, however, can also be given by the OS model according to the OS-defined scheduling strategy. Technically, ARA operates on the LLVM intermediate representation (IR) and expects a single file in this format. We implemented the model for FreeRTOS, AUTOSAR, Zephyr, and POSIX; Figure 2 shows a minimal application example for each of them. While semantically equivalent, the system mostly differs in the way system objects (threads, events, ...) are instantiated: AUTOSAR is completely static, all instances of OS

```cpp
TaskHandle_t t1, t2;
QueueHandle_t q1;
struct Message {...};

int main() {
  t1 = xTaskCreate(task_1, 1);
  t2 = xTaskCreate(task_2, 2);
  q1 = xQueueCreate(5,
    sizeof(Message));
  vTaskStartScheduler(); }

task_1 {
  while(true) {
    Message m = produce();
    xQueueSend(q1, m); } }
task_2 {
  Message m;
  while(true) {
    xQueueReceive(q1, &m);
    consume(m); } }
```

(a) FreeRTOS

```cpp
char* Message = "{...}";
int pipe_fds[2];
pthread_t t1;
pthread_t t2;

thread_1() {
    write(pipe_fds[WRITE_FD], Message);
}

thread_2() {
    read(pipe_fds[READ_FD], received_msg);
}

int main() {
    pipe(pipe_fds);
    pthread_create(t1, thread_1);
    pthread_create(t2, thread_2);
    pthread_join(t1);
    pthread_join(t2);
}
```

(b) POSIX

```cpp
struct Message {...};
K_FIFO_DEFINE(q1);

t1_action() {
  Message m = produce();
  k_fifo_put(&q1, &m);
}
t2_action() {
  Message* m =
    k_fifo_get(&q1, K_FOREVER);
  consume(m);
}

K_THREAD_DEFINE(t1, t1_action, 1);
k_thread t2;
int main() {
  k_thread_create(t2, t2_action, 2);
}
```

(c) Zephyr

```oil
TASK T1:
  CPU = 1;
  PRIORITY = 2;
  SCHEDULE = FULL;
  AUTOSTART = TRUE;

TASK T2:
  CPU = 2;
  PRIORITY = 1;
  SCHEDULE = FULL;

EVENT e1:
  TASK = T2;
```

```cpp
Message m;


TASK(T1) {
  m = produce();
  SetEvent(T2);
}


TASK(T2) {
  WaitEvent();
  consume(m);
}
```

(d) AUTOSAR

Fig. 2: Examples for OS interfaces: Two threads implement a producer–consumer scheme. In FreeRTOS, POSIX, and Zephyr via a queue; in AUTOSAR, which misses a queue abstraction, an event (condition variable) is employed. AUTOSAR specifies its OS objects in an extra configuration file (*.oil*).

Listing 1: The SIA algorithm (sketched)

```
def SIA(entry) -> InstanceGraph:
  instance_graph = InstanceGraph()
  for call in CFG:
    if model.is_syscall(call):
      for call_context in all_call_contexts(entry, call)
        instance_graph.update(model.interpret(call,
                                              call_context))
  return instance_graph
```

objects are specified in a configuration file and typically created at compile time. In contrast, FreeRTOS and POSIX (except static mutexes) require dynamic OS object creation by syscalls at run time. Zephyr supports both, static (via preprocessor macros) and dynamic (via syscalls) instantiation.

## III. OS Model Design

Our model is based on two fundamental ideas: (1) The least common ground of all operating systems are syscalls, which modify the state of OS objects (such as threads or mutexes). (2) The model shall always serve the most detailed information possible about a specific syscall and its resulting state changes. Thereby, the model supports the most detailed analysis while others can just throw away the unneeded details.

For the initial design of the OS model, we target mainly two analyses, the static instance analysis (SIA) [13], which is a flow-insensitive analysis, and the system-state enumeration (SSE) [7], as an example of a flow-sensitive analysis. To better understand the underlying requirements for the model design, we briefly introduce them here.

Listing 2: The SSE algorithm (sketched)

```
def system_semantic(state) -> List[State]:
  if model.is_syscall(state.abb):
    new_states = model.interpret(state)
    return model.schedule(new_states)
  else:
    return follow_control_flow(state)

def SSE(entry) -> SSTG:
  sstg = SSTG()
  stack = model.get_initial_os_state()
  while stack:
    state = stack.pop()
    new_states = system_semantic(state)
    sstg.add_nodes(new_states)
    sstg.connect(new_states, state)
    stack.push(new_states)
```

### A. SIA

The SIA retrieves all system-object instances (and interactions) that are created over the whole lifetime of the system and captures them in the instance graph. Its nodes represent the instances, its edges the interactions. Listing 1 sketches the algorithm. First, it iterates all syscalls. After that, the analysis calculates the call context of each syscall to enable a call-context–aware analysis of the argument values. The call together with its context is then given to the model which calculates the OS-specific effect on the instance graph. From the model point of view, the analysis mainly needs this information:

- Which call is a syscall and what is its category?
- What is the effect of the syscall on the instance graph?

### B. SSE

The SSE at its core is designed as a symbolic execution on the OS level. It defines an abstract system state (the OS relevant state of the whole system), extracts the starting state of the system, and traverses the control flow from this point while capturing the effect of each instruction as a new state. Listing 2 sketches the SSE algorithm. The analysis starts by retrieving an OS-specific initial state that it pushes onto a stack. Then, for each state, it first retrieves the effects of the current control flow onto the state (it interprets the semantics of the system), which it captures in a set of new states. After that, it connects the new states with the old one thus forming a graph, the static state-transition graph (SSTG). The *system semantic* function is divided into two parts: If the state represents a syscall, the model needs to interpret and schedule it. Otherwise, the analysis calculates the new states by following the normal control flow. As part of this, it also triggers all currently active interrupts whose handling is part of the model again (not sketched).

Listing 3: The model interface

```
class OSBase:
  public:
    get_special_steps() -> List[Step]
    get_initial_state(cfg, instances: Graph) -> State

    get_interrupts(instances: Graph) -> List[int]
    handle_irq(state, cpu_id: int, irq: int) -> State
    handle_exit(state, cpu_id: int) -> List[State]

    interpret(state, cpu_id: int,
              categories=All) -> List[State]
    schedule(state, cpus=None) -> List[State]

  private:
    List[Syscall] syscalls
```

From the model point of view, the SSE needs the following information:

- Which call is a syscall?
- What is the effect of the syscall on the abstract state?
- In which abstract state does the system start?
- The possibility to schedule an abstract state.
- Which interrupts can occur in which state and how they are handled?

### C. A generic OS model

Additionally, to be more generic, our model should fulfill also the following requirements: (1) It should be able to support multi-core applications. In particular, this means, that the model must be able to calculate the effect of a syscall on a specific CPU. (2) Furthermore, it should not restrict the OS initialization and setup process. The presented OS all have different configuration mechanisms, which shall be supported. (3) Finally, the model should allow other future analyses of different precision.

All this results in the definition of an OS interpreter that acts on abstract system states (AbSSs), that is, the model implements a function for each syscall that takes an AbSS, interprets the effect of a syscall on this state, and outputs one or multiple follow-up states:

$$\mathrm{AbSS}_{a,n+1}, \mathrm{AbSS}_{b,n+1}, \cdots = \mathrm{interpret}(\mathrm{AbSS}_n)$$

Conceptually, this is pretty close to the SSE algorithm. The AbSS, however, is extended. Figure 3 presents such an AbSS. First, it includes a reference to the instance graph, a data structure whose elements are immutable and to which only can be added. Furthermore, it holds all system-object instance contexts, which represents all changeable parts of an OS object instantiation, for example, the current thread status. The exact content of the context is OS specific and therefore not part of the generic interface. Finally, it holds the current execution context for each CPU, that is, each execution unit in the system. This consists of the current instruction pointer, a call path to specify the calling context, the current interrupt state, and the currently executed instance. To summarize, the state consists of OS-specific parts, the objects and their contexts, and hardware-specific parts, the execution contexts.

With that, each OS model implements a generic interface that uses the AbSSs. Listing 3 shows the (simplified) interface. The list of syscalls contains most of the information. Each syscall is an object with four properties: the name, its signature, a category and an interpret function. The name serves as unique identifier to dispatch to the correct syscall interpretation function. The category is used to fasten the analysis when used as a filter. The signature is necessary for the extraction of the syscall arguments, which in turn is necessary for the correct interpretation of the syscall. Finally, the interpret function gets an abstract state as input and outputs a list of new states which represents the effects of this specific syscall.

To ease the development, we use a Python decorator that turns a function into a syscall object and adds the necessary value-analysis code to each interpret function. Each argument in the signature can be annotated with extra information for the value analysis. The interpret function gets the results of the value analysis via the `args` argument. The syscall name is extracted from the function name. With that, for example, the implementation for *SetEvent* in AUTOSAR looks as follows:

```
@syscall(categories={SyscallCategory.comm},
         signature=(Arg("task", ty=Task, hint=SigType.instance),
                    Arg("event_mask")))
def SetEvent(cfg, state, cpu_id, args, va):
  task_ctx = state.context[args.task]
  # set the task ready if it already waits
  if task_ctx.status == TaskStatus.blocked and \
      event_mask & task_ctx.waited_events != 0:
    task_ctx.status = TaskStatus.ready
    task_ctx.waited_events = 0

  # set the event
  if task_ctx.status != TaskStatus.suspended:
    task_ctx.received_events |= event_mask

  # update the instance graph
  cur_task = state.cpus[cpu_id].instance
  for event in get_events(args.event_mask):
    state.instances.add_edge(cur_task, event)

  return state
```

All the effects of *SetEvent* are captured: First, it wakes up the potentially waiting task. Then, it updates the task's event mask and finally marks the interaction within the instance graph.

The rest of the interface provides the necessary functions for initialization, interrupt handling, and interpretation:

- `get_special_steps` gives the OS-specific preprocessing steps and `get_initial_state` returns the first abstract system state.
- `get_interrupts` returns a list of interrupts that are triggered by the analysis if needed and can be handled via `handle_irq` on which the `irq` argument describes the interrupt to be handled. `handle_exit` outputs a new state which represents all effects that result from an interrupt exit.
- `interpret` and `schedule` are the actual transition functions for abstract states to simulate a syscall interpretation and a reschedule. Both functions return a list of follow-up states.

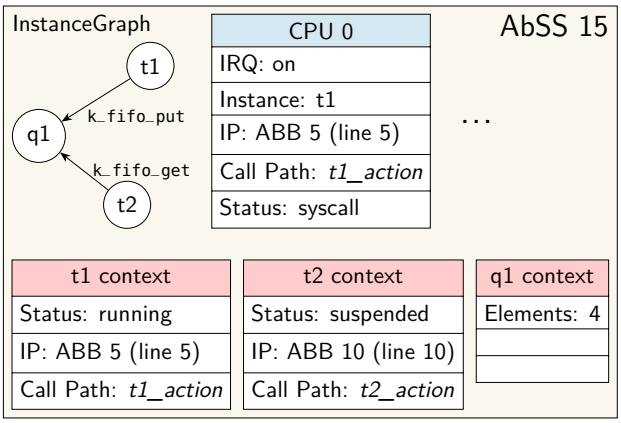


Fig. 3: Representation of the AbSS. It contains a reference to the instance graph, a list of OS-object-specific contexts, and a list of CPUs. The values match the Zephyr example application (Figure 2c).

To support multi-core systems, most functions also get an additional `cpu_id` argument to specify the (abstract) CPU on which the action should take place. The analysis has to take care of invoking the in reality happening parallel actions in a sequential manner.

Figure 1 gives an overview of embedding the model into ARA. The application code that is written against a specific OS interface is preprocessed by ARA to extract control flow and data flow. At this stage, the model can request additional steps, like the parsing of extra system configuration files. After that, the main analyses run, which use the model interface for all OS-specific parts.

Mapping the SSE onto the generic model is trivial. The model interface basically provides all necessary functions for direct SSE support.

For using the model with the SIA, we have to slightly modify the algorithm. Since the model applies the instance-graph specific effects only as part of an overall state change, the SIA has to craft a fake state to fit the model functions. For that, it combines the current instance graph, constructs a fake CPU and empty OS-object contexts. The fake CPU contains the current instruction pointer, call context and active OS-object. The model interprets this state and returns an updated state. From that, the SIA can extract the updated instance graph and continue.

## IV. Experimental Validation

To validate the model, we apply the SIA and the SSE to several applications (see Table I for details). The SSE, which enumerates *all* system states, depends on a strictly bounded set of system objects and interactions. Currently, only AUTOSAR ensures this. On the other OSs, the application might, for instance, create system objects in an unbounded loop.

### A. FreeRTOS

For FreeRTOS, we verified the model with the GPSLogger[2], an embedded application for logging GPS positional data on an handheld device and the LibrePilot CopterControl[3] firmware as a safety-critical real-time application for the flight controller of a quadcopter.

Doing so for the GPSLogger results in 6 tasks, 3 queues and 1 mutex which a manual verification proofs as complete. The SIA fails to resolve 3 interactions between tasks and mutexes (out of 15 interactions in total) due to restrictions in the value analysis, which fails the correct syscall argument retrieval to get the involved mutex instance. Especially for mutexes, the GPSLogger uses a C++ wrapper class which forces the value analyzer to resolve two indirections.

For the LibrePilot the SIA finds 17 tasks, 15 queues and 9 mutexes which are correct. Additionally, it can determine 12 interactions, while it ignores 22 invocations of interaction syscalls due to restrictions in the value analyzer.

### B. Zephyr

For Zephyr,[4] we were not able to obtain any implemented real-world application. Hence, we decided to choose two of the bigger benchmarks from their test suite as applications: sys_kernel and app_kernel.

The app_kernel application creates all OS objects statically, for which we use a special preprocessing step. It detects 2 threads, 6 kernel semaphores, 4 message queues, 3 pipes and 1 mutex, which a manual check verifies as correct.

The OS objects are connected with 60 interactions. ARA fails to determine the arguments of 2 interactions. They belong to a pipe interaction in which the pipes are stored dynamically in an array and therefore are not found by the value analyzer.

The sys_kernel benchmark creates its instances dynamically of which ARA detects 22 threads, 14 queues, 6 kernel semaphores, and 6 stacks. Additionally, ARA finds 274 interactions. A manual check confirms these results.

The sys_kernel application reassigns its OS objects to the same memory location. This makes it impossible for a flow-insensitive analysis like the SIA to retrieve a correct mapping between OS object creation and its usage. Our model, therefore, marks these objects as duplicated and cannot distinguish interactions that lead to them. We plan to extend the SIA in the future to become flow-sensitive for exactly those parts.

### C. AUTOSAR

For AUTOSAR, we use the *I4Copter* [25], a safety-critical embedded real-time control system (quadrotor helicopter), as a test application. Due to its static nature, both the SIA and SSE work with AUTOSAR.

Applying the SIA to the *I4Copter* results in 78 interactions. They happen on 30 OS objects which are defined statically in the configuration file that ARA parses in a preprocessing step.

Applying the SSE to the *I4Copter* results in an SSTG with 648 479 states and 2 032 326 transitions. We also ported the SSE unit tests from dOSEK [9] to ARA and manually verified all resulting SSTGs.

[2]https://github.com/grafalex82/GPSLogger, Git commit: `8808b922`

[3]https://www.librepilot.org/, Version 16.0.9

[4]https://www.zephyrproject.org/ Commit: c2a0b0f50b

| | FreeRTOS | | Zephyr | | AUTOSAR | POSIX |
|---|---|---|---|---|---|---|
| | GPSLogger | LibrePilot | app_kernel | sys_kernel | i4copter | libmicrohttpd |
| Lines of code | 79 573 | 78 787 | 1603 | 1206 | 591 | 45 322 |
| Number of basic blocks | 11 268 | 19 974 | 1 152 | 688 | 148 | 41 698 |
| Number of functions | 1 311 | 3 028 | 212 | 95 | 30 | 2 755 |
| Number of calls | 118 | 919 | 49 | 34 | 0 | 129 |
| Number of syscalls | 37 | 187 | 56 | 91 | 48 | 88 |
| Maximal call path depth | 16 | 5 | 5 | 3 | 1 | 8 |

TABLE I: Code statistics of the benchmark applications.

### D. POSIX

POSIX defines more than a thousand syscalls. A huge part of them (e.g. `strcmp`) does not need the operating system or defines concepts that are unusual in embedded systems (e.g. `fork`). We therefore restricted our POSIX model to the subset of calls[5] that is likely to be employed in an embedded context.

We evaluated the model with libmicrohttpd,[6] an HTTP server library, which is well suited for embedded controllers due to its small memory footprint. To make use of the library, we decided for `fileserver_example_dirs` as an application, which libmicrohttpd includes as an example. We built and analyzed the library in conjunction with the musl libc[7] as implementation of the POSIX standard for the user space.

We found that libmicrohttpd on its own was too dynamic to be useful for ARA. Therefore, we modified libmicrohttpd to reduce its complexity, make the data more static, and focus on the features that are supported by the model. The demo application is runnable on our modified version of libmicrohttpd.

The generated instance graph consists of 4 threads, 1 pipe, 19 files, and 64 mutexes. ARA was able to find 123 interactions. For 46 invocations of interaction syscalls, it fails to determine the belonging OS object due to dynamic calculations or a complicated data flow. While ARA detects nearly all created objects correctly one static mutex is missing since libmicrohttpd `typedef` the standard mutex type which our preprocessing step cannot resolve.

ARA finds 18 different call contexts for `fopen` and one call to `opendir` which it tracks as files. However, this does not reflect all loaded files at runtime since libmicrohttpd opens all files in the current directory, which is inherently dynamic information.

Overall, while we were not able to always find all OS object instantiations and interactions this is not a restriction of the model but of the value analyzer and the analysis algorithms.

## V. Discussion

Implementing the model for the four OSs shows the general applicability of the approach, but also uncovers some practical challenges:

**Value analysis.** The semantics of a concrete syscall is also determined via its arguments. To statically extract their values, we leverage a sophisticated value analyzer based on the Static Value-Flow (SVF) [24] framework, which, however, is still not able to find all value-flows in real-world C/C++ code. Especially constant values that are passed around via (nested) structs appear to be difficult to resolve.

**Dynamic object creation.** A general problem of static analysis is the possibility to create new system objects at run time. While most real-time applications behave well in this respect in that they create and initialize all system objects before entering the application's main loop, the OS interface does not enforce this. Luckily, the SIA is able to detect this in a reliable manner [13].

The most analysis friendly system in these respects is AUTOSAR, as all system-object instances and also some of their possible interactions (e.g., which task may take which resource) must be declared ahead of time in the configuration file. FreeRTOS, Zephyr, and POSIX are less nicely. While Zephyr, at least, supports static system object definition, POSIX and FreeRTOS basically rely on dynamic object creation only.

**Scheduling determinism.** For the state change, the scheduling policy is taken into account. Again, AUTOSAR specifies a fixed priority scheduling (partitioned on multi core) which requires no additional information in the state. FreeRTOS allows tasks with the same priority that are scheduled in a round-robin fashion via a timer interrupt which has to be tracked in the state. Zephyr supports multi core but does not really specify how tasks are scheduled on different cores.

**Semantically uncompleted syscalls.** Especially POSIX defines syscalls that are not conclusive with respect to their effect on the system state. Thread creation is a good example here: To specify the exact behavior of threads, the developer may optionally provide thread attributes, which need to be created and modified by a sequence of syscalls before the actual `pthread_create` call that puts the new thread into existence. This may result in complex dependency chains, defined by the control flow, rendering a flow-insensitive analysis like the SIA impossible. In the future, we plan to extend the SIA to switch to a flow-sensitive analysis for exactly these parts of the code.

**OS/library interface ambiguities.** The POSIX standard does not distinguish between syscalls (like `read()`) and utility library functions (e. g. `strcmp()`). Both of them are implemented within the same libc. We tackle this by splitting the libc conceptually in syscalls and user functions and analyzing only the former. This can also be implementation-defined, so it might be required to adopt the POSIX model for the concrete OS.

[5] pthread_create, pthread_mutex_init, PTHREAD_MUTEX_INITIALIZER, sem_init, pthread_cond_init, PTHREAD_COND_INITIALIZER, pipe, pause, nanosleep, read, readv, sigaction, open, pthread_join, pthread_detach, pthread_cancel, pthread_mutex_lock, pthread_mutex_unlock, sem_wait, sem_post, pthread_cond_wait, pthread_cond_signal, pthread_cond_broadcast, write, writev

[6] https://www.gnu.org/software/libmicrohttpd/ v0.9.73, Commit: 64e91ef6

[7] http://musl.libc.org/

Overall, our findings show that ARA is able to extract OS-interaction knowledge regardless of the specific OS. The remaining implementation challenges mostly result from idiomatic anachronisms of the respective OS interfaces and programming models (e.g., POSIX, FreeRTOS) and underspecification of its semantics (e.g., multi-core scheduling), which naturally limits static analysis. Note, however, that ARA still behaves sound in these cases, even though it yields less tight analysis results.

## VI. Related Work

To the best of our knowledge, no other compiler exists that tailors OSs while supporting multiple of them. However, there do exist other usages of operating system models and compilers for different purposes.

First, dOSEK [16] is a whole system compiler and able to tailor applications written for the OSEK OS standard, the predecessor of AUTOSAR. As part of its implementation, it contains an abstract OSEK interpreter. In contrast to the OS model of ARA, this interpreter is for OSEK only which includes the restriction to single-core applications. However, ARA is influenced by dOSEK and shares concepts and code with it.

SWAN [22] is a whole-system WCET analyzer that also builds an SSTG to calculate tighter bounds for a better WCET. While SWAN also operates on FreeRTOS, it does not use a model of it but analyzes the internal syscall implementation to calculate a tighter WCET bound.

With the RTSC, a whole system compiler exists that was written to automatically convert event-driven real-time systems (written for OSEK) into time-driven real-time systems (written for OSEKTime) [20]. As an extension, it supports the mapping on multi core and into a POSIX program [14]. The RTSC, therefore, uses a system model for OSEKTime and POSIX but aims for system generation only.

Another common use of operating system models is formal verification or conformance checking. Brekling et. al. define a precise operating system specification based on timed automata [3]. With the help of UPAAL, the automata can both be converted to running code and theoretically verified. The model here, however, can be seen as another OS implementation and does not try to unify and abstract existing OSs.

For the OSEK and AUTOSAR standard, several works exist to convert the specification into a formal model to verify the system like checking for schedulability or generating conformance tests [4], [17], [27], [28], [26], [10], [11]. Similar works try to formalize parts of FreeRTOS [5], [19], [6] or Zephyr [12]. All of these works use models that are not used to optimize the system but proof the real-time capabilities. Furthermore, in contrast to ARA, no unifying model is formulated.

## VII. Conclusion

In this paper, we presented a generic interface to model OS behavior for usage in static analysis. Thereby, we were able to reduce the constraints for an OS to a minimum: The system must communicate with a syscall interface.

We implemented the interface for the four OSs FreeRTOS, AUTOSAR, Zephyr, and POSIX and evaluated all targets with at least one test application. The results prove the working of the model. While experiencing limitations, we can assign them either to the analysis algorithm or inaccuracies in the value analysis, not the OS model itself. We discussed the characteristics of different OSs and their potential for optimization.

## References


[1] Tobias Amnell, Elena Fersman, Leonid Mokrushin, Paul Pettersson, and Wang Yi. TIMES: A tool for schedulability analysis and code generation of real-time systems. In *Formal Modeling and Analysis of Timed Systems*. Springer Berlin Heidelberg, 2004.

[2] Ramon Bertran, Marisa Gil, Javier Cabezas, Victor Jimenez, Lluis Vilanova, Enric Morancho, and Nacho Navarro. Building a global system view for optimization purposes. In *2nd Work. on the Interaction between Operating Systems and Computer Architecture (WIOSCA '06)*. IEEE Computer Society Press, 2006.

[3] Aske Brekling, Michael R. Hansen, and Jan Madsen. Models and formal verification of multiprocessor system-on-chips. *The Journal of Logic and Algebraic Programming*, 77(1), 2008. The 16th Nordic Work. on the Prgramming Theory (NWPT 2006).

[4] Jiang Chen and Toshiaki Aoki. Conformance testing for OSEK/VDX operating system using model checking. In *18th Asia-Pacific Software Engineering Conf. (APSEC 2011)*. IEEE Computer Society Press, 2011.

[5] Nathan Chong and Bart Jacobs. Formally verifying FreeRTOS' interprocess communication mechanism. In *Embedded World Exhibition and Conf.*, 2021.

[6] David Déharbe, Stephenson Galvao, and Anamaria Martins Moreira. Formalizing FreeRTOS: First steps. In *Brazilian Symp. on Formal Methods*. Springer, 2009.

[7] 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)*. ACM Press, 2015.

[8] Christian Dietrich, Martin Hoffmann, and Daniel Lohmann. Global optimization of fixed-priority real-time systems by RTOS-aware control-flow analysis. *ACM Trans. on Embedded Computing Systems*, 16(2), 2017.

[9] Christian Dietrich and Daniel Lohmann. OSEK-V: Application-specific RTOS instantiation in hardware. In *2017 ACM SIGPLAN/SIGBED Conf. on Languages, Compilers and Tools for Embedded Systems (LCTES '17)*. ACM Press, 2017.

[10] Timothee Durand, Katalin Fazekas, Georg Weissenbacher, and Jakob Zwirchmayr. Model checking AUTOSAR components with CBMC. In *2021 Formal Methods in Computer Aided Design (FMCAD)*. IEEE, 2021.

[11] Ling Fang, Takashi Kitamura, Thi Bich Ngoc Do, and Hitoshi Ohsaki. Formal model-based test for AUTOSAR multicore RTOS. In *2012 IEEE Fifth Intl. Conf. on Software Testing, Verification and Validation*. IEEE, 2012.

[12] Zhang Feng, Zhao Yongwang, Ma Dianfu, and Niu Wensheng. Fine-grained formal specification and analysis of buddy memory allocation in Zephyr RTOS. In *2019 IEEE 22nd Intl. Symp. on Real-Time Distributed Computing (ISORC)*, 2019.

[13] Björn Fiedler, Gerion Entrup, Christian Dietrich, and Daniel Lohmann. ARA: Static initialization of dynamically-created system objects. In *27th IEEE Real-Time and Embedded Technology and Applications Symp. (RTAS'21)*, 2021.

[14] Florian Franzmann, Tobias Klaus, Peter Ulbrich, Patrick Deinhardt, Benjamin Steffes, Fabian Scheler, and Wolfgang Schröder-Preikschat. From intent to effect: Tool-based generation of time-triggered real-time systems on multi-core processors. In *19th IEEE Intl. Symp. on Object-Oriented Real-Time Distributed Computing (ISORC '16)*. IEEE Computer Society Press, 2016.

[15] T. Glek and Jan Hubicka. Optimizing real world applications with GCC link time optimization. *CoRR*, abs/1010.2196, 2010.

[16] Martin Hoffmann, Christian Dietrich, and Daniel Lohmann. dOSEK: A dependable RTOS for automotive applications. In *19th Intl. Symp. on Dependable Computing (PRDC '13)*. IEEE Computer Society Press, 2013. Fast abstract.

[17] Yanhong Huang, Yongxin Zhao, Longfei Zhu, Qin Li, Huibiao Zhu, and Jianqi Shi. Modeling and verifying the code-level osek/vdx operating system with csp. In *5th Intl. Symp. on Theoretical Aspects of Software Engineering (TASE'11)*. IEEE Computer Society Press, 2011.

[18] Maksim Panchenko, Rafael Auler, Bill Nell, and Guilherme Ottoni. Bolt: a practical binary optimizer for data centers and beyond. In *2019 IEEE/ACM Intl. Symp. on Code Generation and Optimization (CGO)*. IEEE, 2019.

[19] David Sanán, Liu Yang, Zhao Yongwang, Xing Zhenchang, and Mike Hinchey. Verifying FreeRTOS' cyclic doubly linked list implementation: From abstract specification to machine code. In *2015 20th Intl. Conf. on Engineering of Complex Computer Systems (ICECCS)*. IEEE, 2015.

[20] Fabian Scheler and Wolfgang Schröder-Preikschat. The RTSC: Leveraging the migration from event-triggered to time-triggered systems. In *13th IEEE Intl. Symp. on Object-Oriented Real-Time Distributed Computing (ISORC '10)*. IEEE Computer Society Press, 2010.

[21] Horst Schirmeier, Matthias Bahne, Jochen Streicher, and Olaf Spinczyk. Towards eCos autoconfiguration by static application analysis. In *1st Intl. Work. on Automated Configuration and Tailoring of Applications (ACoTA '10)*, CEUR Work. Proceedings. CEUR-WS.org, 2010.

[22] Simon Schuster, Peter Wägemann, Peter Ulbrich, and Wolfgang Schröder-Preikschat. Proving real-time capability of generic operating systems by system-aware timing analysis. In *2019 IEEE Real-Time and Embedded Technology and Applications Symp. (RTAS)*, 2019.

[23] Amitabh Srivastava and David W. Wall. Link-time optimization of address calculation on a 64-bit architecture. 29(6), 1994.

[24] Yulei Sui and Jingling Xue. SVF: Interprocedural static value-flow analysis in LLVM. In *25th Intl. Conf. on Compiler Construction*, CC 2016. Association for Computing Machinery, 2016.

[25] Peter Ulbrich, Rüdiger Kapitza, Christian Harkort, Reiner Schmid, and Wolfgang Schröder-Preikschat. I4Copter: An adaptable and modular quadrotor platform. In *26th ACM Symp. on Applied Computing (SAC '11)*. ACM Press, 2011.

[26] Dieu-Huong Vu, Yuki Chiba, Kenro Yatake, and Toshiaki Aoki. Verifying OSEK/VDX OS design using its formal specification. In *Proc. TASE'16*. IEEE Computer Society, 2016.

[27] Haitao Zhang, Toshiaki Aoki, and Yuki Chiba. Yes! you can use your model checker to verify OSEK/VDX applications. In *8th IEEE Intl. Conf. on Software Testing, Verification and Validation, ICST 2015, Graz, Austria, April 13-17, 2015*, 2015.

[28] Min Zhang, Yunja Choi, and Kazuhiro Ogata. A formal semantics of the OSEK/VDX standard in K framework and its applications. In *Proc. WRLA'14*. Springer, 2014.

[29] Ming-Yuan Zhu, Lei Luo, and Guang-Ze Xiong. The minimal model of operating systems. *SIGOPS Oper. Syst. Rev.*, 2001.