# ARA: Automatic Instance-Level Analysis in Real-Time Systems

Gerion Entrup, Benedikt Steinmeier, Christian Dietrich
Leibniz University Hannover
{entrup, dietrich}@sra.uni-hannover.de, benedikt.steinmeier@gmail.com

*Abstract*—**Real-time control applications are usually implemented by mapping their real-time model (i.e., tasks, shared resources, and external events) onto software instances of RTOS abstractions, such as threads, locks and ISRs. These instantiated objects and their interactions define what actually happens on the imperative machine; they implement the desired behavior. However, during the lifetime of many projects, the initial real-time model gets lost, becomes outdated, or never existed at all, as all (further) development has been code centric: The source code is all that we have. So developers face a situation where further modifications of the real-time system, but also any attempt of static RTOS tailoring, requires the extraction and the understanding of the employed RTOS instances and their concrete interactions from the source code.**

**We present ARA, a tool capable of automatically retrieving instance-level knowledge (e.g., the instanciated threads, locks, or ISRs) from a given (real-time) application. ARA is an RTOS-aware static analyzer that derives, given the application source, a graph of the employed RTOS abstractions, their concrete instances, and how these instances interact with each other at run time. We describe the design principles behind ARA and validate its implementation with four example applications for OSEK/AUTOSAR and FreeRTOS.**

## I. Introduction

In the domain of real-time systems, application development often begins by mapping tasks, shared resources, and external events onto the underlying real-time operating systems (RTOSs) abstractions, like threads, locks, and interrupt service routines (ISRs). This implementation later executes on the actual machine, leaving the application code as *the* ground truth that defines the system behavior. It consists of concrete *instances* that *interact*, mediated by the RTOS, with each other. For example, an externally-activated ISR activates the data-processing thread after data has been received.

Since developers often use only a small part of the RTOS, leaving a lot of functionality unused, application-specific system specialization bears significant improvements [4], [7], [21]. At OSPERT'18, we presented a taxonomy of specialization levels for (real-time) system software [10]. There, we defined three levels on which the RTOS can be specialized: (1) On the level of abstractions, whole RTOS abstractions can be dropped if applications do not need them. (2) On the level of instances, we can use specialized RTOS data structures and algorithms best suited for the known set of instances. (3) On the level of interactions, we specialize the RTOS with knowledge about individual interactions, like having a single queue writer.

We demonstrated with the example of the GPSLogger[1] application, an real-world real-time application, how instance- and interaction-level knowledge can be used to further specialize the underlying FreeRTOS in order to reduce the memory footprint and startup time. With instance-level knowledge, we were able to initialize all stacks, thread-control blocks, and scheduler data structures statically, which reduced the startup time by 10 percent compared to the unmodified system. With interaction-level knowledge, we could embed a short running thread directly into the activating ISR as it did not interact with any other thread. For this, however, we had to manually analyze the source code of the application and how it exactly utilizes the RTOS; a tedious task. We are convinced that manual specialization is infeasible and that an automatic process to retrieve the used instances and their interactions is needed.

As a follow-up of our previous work, we present ARA (**A**utomatic **R**eal-time system **A**nalyzer), a tool[2] that automatically extracts instance-level knowledge. From the application's source code, ARA extracts all application–operating-system interactions and identifies all instances of RTOS abstractions. Furthermore, ARA retrieves the interactions between the instances and provides further system information, like the number of thread activations. As a result, ARA produces the*instance graph*, a data structure that captures instances as nodes and their interaction as edges. For example, for a queue with only one writer, the instance graph contains only a single write edge with the queue node as target.

Thereby, ARA is not restricted to one specific RTOS. Currently, it is able to analyze applications written against the OSEK/AUTOSAR standard [2], an RTOS API with static instances, and FreeRTOS [3], an RTOS with a POSIX-like interface, where all instances are created dynamically. As both RTOSs name their thread implementation "task", we will use thread and task as interchangeable terms.

The knowledge about instances and interactions cannot only be used for specialization but also for other phases of the development process: When new developers join a project, the instance graph becomes a documentation artifact and provides a fast overview of the code base, easing the introduction phase. Since ARA calculates the instance graph in an automated way, the time-consuming manual extraction and updating of an – often outdated – design document is avoided.

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

[1]https://github.com/grafalex82/GPSLogger
[2]Available at: https://github.com/luhsra/ara

```
ISR i1 {                   .oil   BoundedBuffer bb;        .cpp
  CATEGORY = 2;
  PRIORITY = 101;                 ISR(i1) { // priority: 101
  SOURCE = "PORTA";                 bb.put(readSerial());
}                                   ActivateTask(t1);
                                  }
TASK t1 {
  PRIORITY = 2;                   TASK(t1) { // priority: 2
  SCHEDULE = FULL;                  while(data = bb.get())
}                                     handleSerial(data);
                                  }
TASK t2 {
  PRIORITY = 1;                   TASK(t2) { // priority: 1
  SCHEDULE = FULL;                  while (true)
  AUTOSTART = TRUE;                   handleADC(readADC());
}                                 }
```

List. 1: OSEK example code

```
BoundedBuffer bb;                   task_1 { // priority: 2
TaskHandle_t t1, t2;                  while(1) {
                                         ulTaskNotifyTake();
int main() {                             while(data = bb.get())
  t1 = xTaskCreate(task_1, 2);             handleSerial(data);
  t2 = xTaskCreate(task_2, 1);         }
  vTaskStartScheduler();             }
}
                                     task_2 { // priority: 1
isr_1 { // priority: ∞                 while (true)
  data = readSerial();                   handleADC(readADC());
  bb.put(data);                      }
  vTaskNotifyGiveFromISR(t1);
}
```

List. 2: FreeRTOS example code

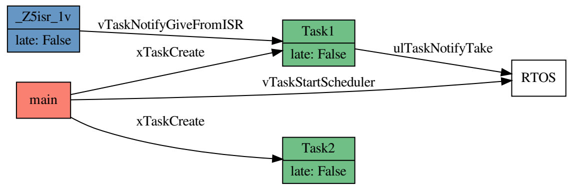

Fig. 1: Instance graph for the given example as generated by ARA. Edge labels always belongs to the edge below them.

Furthermore, the instance graph, which captures the actual usage of RTOS abstractions, can act as a base for further static analyses, like searching for misused RTOS APIs or protocol violations. For example, interaction knowledge makes it possible to check whether calls that take and release a lock occur pairwise. In ARA, we already provide such checks based on the instance graph.

Summarized, the instance graph has three benefits: It serves as a knowledge base for further RTOS specialization. It gives an overview of the application's code base and becomes a living documentation of the program. It provides knowledge to check the application for incorrect or unusual usage of operating system abstractions.

With this paper, we claim the following contributions:

1) We define the *instance graph* as a knowledge base that captures RTOS instances and their interactions.
2) We present automated methods to statically retrieve an instance graph from a given OSEK or FreeRTOS application.
3) We apply our methodology to four real-world applications to validate our approach.

## II. System Model

The input of ARA is a statically configured (real-time) system, so the entire application code is known at compile time. In particular, we have chosen two (real-time) operating-system APIs that meet this requirement: OSEK and FreeRTOS.

### A. Overview of OSEK

The OSEK standard defines an interface for fixed-priority RTOSs and has been the dominant industry standard for automotive applications for the last two decades.

It offers two main control-flow abstractions: *ISRs* and *tasks*. Additionally, primitives for inter-task synchronization are provided. All instances must be declared statically in the domain-specific OSEK Implementation Language (OIL) [18], [19].

Listing 1 provides an example OSEK system. We see two tasks and one ISR: task `t1` waits for a notification from ISR `i1` and consumes its input, while task `t2` runs constantly and handles the analog-digital converter. All instances are statically declared in an OIL file (printed on the left side). The scheduler starts task `t2` automatically at boot, while task `t1` gets activated by ISR `i1`. It is noteworthy that the used bounded buffer `bb` is not an RTOS abstraction, but used as global data structure.

### B. Overview of FreeRTOS

FreeRTOS is an RTOS stewarded by Amazon to use it together with their cloud instances [11]. One of its core features is the high number of ports to different microcontrollers.

FreeRTOS offers *tasks* as a control-flow abstraction and several synchronization primitives like *semaphores* or *queues*. Unlike OSEK, FreeRTOS does not directly offer an ISR abstraction. Instead, it defines a special class of system calls that can be called from an ISR and they can be recognized by their "FromISR" suffix.

In contrast to OSEK, the FreeRTOS API is dynamic: The application creates all OS instances at run time; either in an initialization phase or during the continued operation. Listing 2 shows the running example using the FreeRTOS API. To foster readability, we have left out some system-call arguments, like the stack size or the name of the created thread (`xTaskCreate()`). Compared to Listing 1, this example contains a main function that sets up the system and starts the scheduler. FreeRTOS is not aware of `isr_1` being an ISR, but we can recognize it by the `vTaskNotifyGiveFromISR` system call.

## III. Instance Graph

In this section, we will define the *instance graph* and will present a method to automatically create it. An instance graph describes all instances that will exist in the *whole* application lifetime together with their (flow insensitive) interactions. In Figure 1, we show a simple instance graph that ARA automatically extracted from Listing 2.

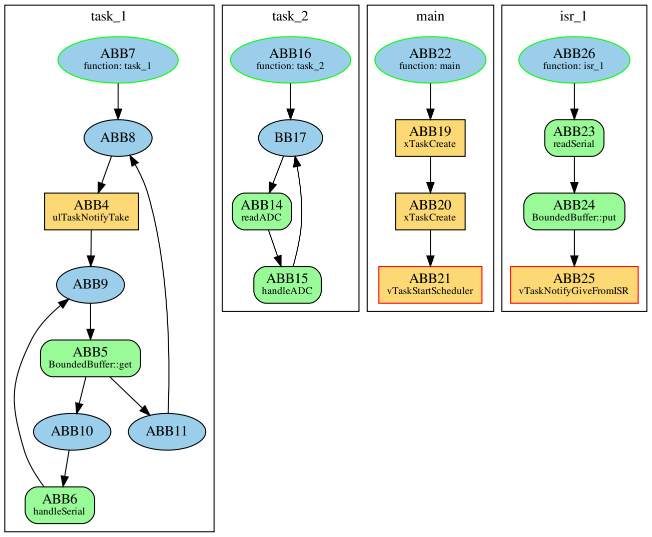

Fig. 2: ICFG for the given example as generated by ARA. System-call blocks are colored in orange (square shape), function-call blocks are colored in green (rounded square shape), computation-blocks are colored in blue (round shape).

The instance graph contains all mentioned instances and their interactions as well as an additional RTOS instance. This pseudo instance collects all interactions that do not take place between two regular instances. Additionally, interactions that originate from the main function reflect the system startup. Since all "late" attributes are set to false, we know that all instances are created before the scheduling begins.

For the construction of the instance graph, two steps are necessary: (1) Building a system-call–aware inter-procedural control flow graph (ICFG). (2) Creating the instance graph based on the ICFG.

## A. *System-Call Aware Inter-Procedural Control Flow Graph*

All interactions between instances originate in system calls; in FreeRTOS also the dynamic instance creation is done via system calls. Therefore, ARA traverses the ICFG of the executed code and then interprets the influence of every system call. For this, it first builds an system-call–centric ICFG that abstracts from the irrelevant code parts.

First, ARA extracts the control-flow graph, which covers the application code with its basic-block nodes. Then, it partitions the control-flow graph into atomic basic blocks (ABBs), a concept introduced by Scheler and Schröder-Preikschat [20], to abstract from the application's micro structure. As an adaptation of the original ABB concept, ARA constructs and connects the ABBs differently, for the whole application at once:

1) Split every basic block (BB) that contains a function or system call. The split is done directly before and after the function or system call. Therefore, all function and system calls reside in their own BB.
2) Each BB gets a type assigned: system-call block, function-call block, or computation block.
3) Merge several computation BBs into a single computation node, if they form a single-entry-single-exit (SE-SE) region, which can only be entered via one distinguished entry BB and left via exactly one exit BB.

Each block constructed with this technique forms an ABB. Afterwards, we have a local ABB-graph for each function within the application code. By assigning a type to every ABBs, we focus on the application logic that is visible to the operating system and all irrelevant computation is subsumed into computation ABBs. Interaction with the kernel is only possible in system-call blocks. Figure 2 shows all application ABBs derived from Listing 2.

Every system call is the intention of an interaction and gets represented by an edge in the instance graph. Usually, the system-call arguments identify the source and the target node together with the exact system-call semantics. Sometimes, the source and target are also defined by the calling instance itself (e.g. the `vTaskDelay` call in FreeRTOS). In order to deduce the system-call arguments, we perform a value analysis: Starting from the call site, we search backwards in the function-local def–use chains and follow the callee–caller relationship if we hit a function beginning. With this interprocedural search, ARA recognizes arguments that have an unambiguous value. In the current implementation, we do not support operations, like an addition with a constant, that change propagated values deterministically.

## B. *Instance Graph Creation*

With the information about the system calls and their arguments, an interpretation of their semantics can be performed. In the first step ARA creates all instances. Here, OSEK and FreeRTOS are handled differently. Since OSEK requires that all instances must be declared in the OIL file, it directly provides information about all instances. As all instances in FreeRTOS are created via system calls, ARA needs to find all instance-creation system calls: It traverses the ICFG, beginning from the system's entry point (usually the main function). Whenever ARA detects an instance-creation system call, it emits a corresponding node in the instance graph. Since it is possible that system calls are invoked in a loop or under a condition, it can happen that the concrete number of actually existing instances cannot be determined ahead of time. This also applies if the system call is contained in another function that is called in a loop or condition. ARA detects such situations, creates exactly one instance, and labels it as being a template for multiple, or an optional, instances.

Additionally, FreeRTOS needs a special handling for ISRs. For all system calls that are recognized as ISR system calls, ARA assumes that they are called within an ISR. To find the actual function that defines the ISR, we traverse the call hierarchy up to a root node.

In FreeRTOS, instance creation can happen anywhere in the application. Therefore, it is important to differentiate whether a creation takes place before or after the start of the scheduler. Since the system entry point is executed exactly once, the code block executed between the entry and the scheduler start gets executed exactly once. For code within a task context, we do

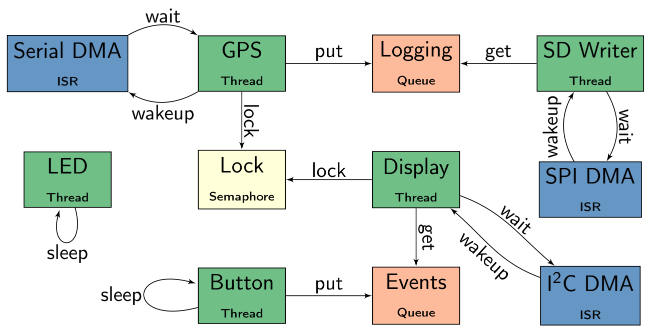

Fig. 3: Manually derived interaction graph for GPSLogger

not have this guarantee, since the task cannot run not at all or it can execute multiple times. Therefore, we analyze all system calls, with respect to their call graphs, and deduce if they are called before or after the start of the scheduler. Also, the call to the scheduler start is a dynamic one. If this call is made in a condition or a loop, no statement can be made. For all instances that are created after the scheduler start, we set the "late" attribute of the instance (see Figure 1).

After the instance creation, we analyze the interactions between them; a step that is equal for FreeRTOS and OSEK. For this, we traverse the ICFG of all tasks and, dependent on the system call, create an edge between the corresponding instances. Since we aim to find all *possible* interactions, the context (loop or condition) of the system call is irrelevant for the edge creation. Interactions whose source or target cannot be determined are assigned to the RTOS instance.

The combination of the system-call–aware ICFG extraction and the subsequent instance and interaction extraction is an automated process and results in the instance graph.

## IV. Program Architecture

ARA uses LLVM [14] and reads multiple files in the LLVM intermediate representation as input format (e.g., clang can create these files easily). The initial control flow graph (CFG) extraction into BBs is performed entirely in LLVM. Additionally, analyses already implemented in LLVM, like dominator analysis or loop information, are used. The LLVM-specific and performance-critical parts of ARA are written in C++, while we default to Python for fast prototyping.

## V. Experimental Validation

To validate the correctness of ARA, we create instance graphs of four real-world applications: The I4Copter with (1) and without (2) events (based on OSEK), the SmartPlug (3, based on FreeRTOS), and the GPSLogger (4, based on FreeRTOS). The generated instance graphs are rather big and can therefore be found in the appendix.

### A. I4Copter

The I4Copter [26] is a safety-critical embedded control system (quadrotor helicopter) developed by the University of Erlangen-Nuremberg in cooperation with Siemens Corporate Technology. We used it as a validation base for OSEK systems. The I4Copter exists in two variants: an implementation for OSEK extended conformance class 1 (ECC1, with events) and another one that runs on the simpler basic conformance class 1 (BCC1, without events). We analyzed both systems with ARA and the instance graphs can be found in Figure 5 and Figure 7. For the event variant, 14 tasks, 4 alarms, 11 events, and 1 resource were identified. From the variant without events, 11 tasks, 4 ISRs, and 1 resource were extracted. We showed the results to an author of the I4Copter who confirmed the results.

### B. SmartPlug

The SmartPlug[3] is a hardware power switch controllable via Wi-Fi. It runs on an ESP8266 and uses FreeRTOS to orchestrate its tasks. The project does not provide any building documentation and depends on several unprovided libraries. We therefore replaced all library calls that do not perform any kind of RTOS interaction with stubs. When analyzing the source code, ARA found 11 tasks, 2 queues, 1 semaphore and 1 ISR, presented in Figure 6. ARA detects 4 tasks that are always created and 7 tasks that are created only if some condition is met (indicated by the question mark at the creation system call). We performed a manual validation which confirmed that these optional tasks are created depending on a configuration option, which is retrieved at run time by reading a file.

### C. GPSLogger

The GPSLogger is a freely available application to collect GPS information.

It runs on a "STM32 Nucleo-F103RB" evaluation board that is equipped with a STM32F103 MCU. It is connected to a graphical display (I$^2$C), a GPS receiver (UART), an SD card (SPI), and two buttons (GPIO). Due to a broken SD card library, we had to replace the SD card operations with a `printf()`. In a previous work [10], we created the instance graph manually as shown in Figure 3. The application consists of 5 tasks, 3 ISRs, 2 blocking queues, and one binary semaphore.

The instance graph as created by ARA is shown in Figure 4. Both graphs are almost isomorph. The automatically-derived graph contains an additional main instance to show all creation system calls and an RTOS instance that captures unassignable interactions. As a main difference, ARA detects the ISR interactions but assigns them to the RTOS instance. ARA does this as a fallback, since the correct instance that the `vTaskNotifyGiveFromISR` call gets as argument is not a global variable but derived dynamically. Also, ARA does not detect one interaction of the "Display Task" with the RTOS (`ulTaskNotifyTake`), since it occurs in a function that ARA's reachability analysis cannot find due to an unresolved function-pointer call.

While we saw some specialties in the analyzed systems, we were able to construct instance graphs from all applications. All instance graphs are providing a compact system overview and can be used as knowledge base for further analysis.

[3]https://github.com/KKoovalsky/Smartplug

## VI. Discussion

In the previous section, we have seen how ARA can extract interaction graphs from different unknown applications, and we validated the results by comparing them with manually extracted graphs and by manual code inspection. While the ARA approach has a great potential to foster application knowledge, its static nature has some limitations; both aspects will be discussed in the following.

### A. Limitations

The main limitation of ARA lays in limitations of its value analysis. On the one hand, this is seen during the extraction of argument values. Values retrieved as result of a function-pointer call or an unambiguous assignment (e.g., in a branch) cannot be retrieved.

On the other hand, ARA does not decide whether to take a branch or how often to execute a loop, so the amount of therein created instances cannot be retrieved. In the current implementation, ARA detects these cases and marks the result appropriately.

In the future, we want improve the recognition by using already implemented compiler techniques such as dead-code elimination and constant folding to remove branches or loop unrolling to determine loop iterations. Another known technique is symbolic execution, which, however, comes with high costs [5]. Nevertheless, we believe that most embedded systems, while programmed against a dynamic API, are rather static in their OS interactions. Mostly, tasks are defined in some kind of main function before the actual scheduler starts and system calls only interpret constant values or global arguments. The analyzed real-world systems are developed this way, except for the SmartPlug, where one task acts as a configuration instance that creates several other tasks. Nevertheless, ARA recognizes this creations but cannot make a statement about the exact amount of instances.

ARA performs a reachability analysis, beginning at the system and task entry points to decide whether an interaction is executed or not. In this analysis, ARA does not resolve any function pointers. In the current implementation, ARA stops the traversal at this point, resulting in possibly unanalyzed system-calls, if they are only reachable via a function pointer. This can lead to unrecognized instances and missing interactions. In the specialization use case unrecognized instances lead to a more generic implementation and thus only to a weaker specialization. However, missing interactions can lead to the selection of the wrong specialization for the corresponding instances and, thus, provoke incorrect system behavior. One way to solve this problem is to retrieve a restricted set of possible call targets by comparing function signatures. A better value analysis will further limit the call-target set of a function pointer. We want to address this limitation in our future work.

The described limitations are inherent for static analysis. Tracing an actual run of the system would circumvent these problems. However, tracing does not detect dynamically *uncreated* instances. This can be seen in the SmartPlug where on an actual run only a subset of all tasks, which are found by static analysis, is created due to dynamic configuration. We plan to extend ARA to additionally support traces.

When we analyzed the real-world applications, we saw different code qualities. Especially the GPSLogger seems like a hobby project that was developed incrementally without a real-time system model. For example, the source base contains two copies of FreeRTOS; both of them are used. Additionally, the analyzed applications are rather small in its code size. We see a threat to validity of ARA's analysis results, that we want to address in our future work with the evaluation of more and larger applications.

ARA considers only interactions that involve the RTOS. For example, ARA does not detect an communication via shared memory like the bounded buffer in Listing 1. Since our main goal is a knowledge gain for RTOS specialization, this is not a limitation. While ARA currently only supports FreeRTOS and OSEK, we plan to extend it to more RTOS APIs.

### B. Advantages

**Application overview** is given by a good visualization of the system composition. This is on the one hand useful to get an overview of the developer's own application as seen by a machine. Often, applications are developed with a program-design model in mind. The instance graph can be used as visualization of this model and prove that it was actually implemented. If the model gets outdated in further development, ARA can serve as a tool to retrieve it in an automated manner. On the other hand, the instance graph is useful as program-design documentation for external developers. Especially for big code bases, it provides the unexperienced developer a compact overview about system composition so she is able to quickly find parts in the source code that are responsible for an observed behavior. ARA is a tool to generate this design document in an automated manner and can, therefore, be integrated into the continuous integration (CI).

**Application verification** is provided by automatic checks. With knowledge about used operating system (OS) abstractions, ARA is able to check for their correct usage. To demonstrate this, we have implemented two verifications in ARA. The first one checks if an ISR in FreeRTOS only uses ISR-enabled system calls. The second one verifies if system calls to enter and exit a critical region always occur pairwise. Automatic lock verification is a topic of ongoing research [13], [16], [15], [9]. Our approach does not try to verify correct lock usage, but it is able to detect lock misuse. Given the already retrieved instance graph, these checks are easy to implement. Again, this functionality of ARA is useful for a CI process.

**Knowledge gain for specialization** is achieved in an automated manner. Instance knowledge at compile time can be used to create a specialized variant of the underlying RTOS. For example, instances can be statically initialized and preallocated at compile time. More efficient data structures (like arrays instead of lists) can be used when the number of instances is known. Algorithms can be improved when the communicating instances are known beforehand. For example, queue synchronization can be reduced if only one producer and one consumer is detected.

## VII. Related Work

There are several other solutions to statically extract the application knowledge that is required to specialize the RTOS. Bertran et al. [4] track – based on the binary application image – those control system that cross the system-call boundary and eliminate dead system- and library calls from Linux and L4. However, they do not extract instance knowledge or try to interpret the system calls. In [7], we built the global control flow graph (GCFG), which we used for excessive system-call specialization. However, the GCFG captures the system only on a flow-sensitive interaction level (instead of the feature and instance level), proved to be computationally more expensive than ARA, and was only implemented for OSEK. Schirmeier et al. [21] transform the CFG in Kripke structures to be able to apply model checking in computational temporal logic (CTL) for patterns that lead to an automatic OS configuration. They apply the method to eCos and its powerful configuration framework. While CTL may be usable to extract instances, the authors aim to use eCos' existing configuration framework and do not try to extract instances or interactions.

A classical approach to document a program structure are UML diagrams. With StarUML [22], BOUML [6], and ArgoUML [1] several tools exist to automate the diagram generation by performing a static analysis on the application source code. Class diagrams are another program-structure visualization, as generated by Structurizr [24], Structure101 [23], NDepend [17], or Doxygen [8] in an automated fashion. However, all these tools extract no instance knowledge, are control-flow agnostic, and do not consider the RTOS.

Especially for the RTOS domain, several tools like Grasp [12] and Tracealyzer [25] exist that retrieve information from the real-time system to show timing behavior of RTOS instances. Therefore, they build an implicit form of an instance graph but with focus on actual execution times. Nevertheless, they use tracing information to retrieve instances and timing behavior and do not perform any form of static analysis. As a result, they only retrieve all actual executed instances. Instances that are defined in the application but not executed in the trace are not retrieved.

## VIII. Conclusion

In this paper, we have presented the instance graph, which is capable of describing all instances of RTOS abstractions together with their interactions. With ARA, we presented a tool to automatically generate an instance graph for applications written against the FreeRTOS or the OSEK API.

We validated the correctness of ARA with four real-world applications and compared the automatically extracted instances graphs to manually extracted knowledge. While having limitations, mainly stemming from the value analysis, ARA was able to recognize all instances and most of its interactions. We have discussed the utility of the instance graph to assist programmers during the application development, to provide an knowledge base for further static analyses, and to foster further RTOS specialization.

## References

[1] ArgoUML - Homepage. http://argouml.tigris.org/.

[2] AUTOSAR. Specification of operating system (version 5.1.0). Technical report, Automotive Open System Architecture GbR, February 2013.

[3] Richard Barry. *Using the FreeRTOS Real Time Kernel*. Real Time Engineers Ltd, 2010.

[4] Ramon Bertran, Marisa Gil, Javier Cabezas, Victor Jimenez, Lluis Vilanova, Enric Morancho, and Nacho Navarro. Building a global system view for optimization purposes. In *WIOSCA'06*, Washington, DC, USA, June 2006. IEEE Computer Society Press.

[5] Armin Biere, Jens Knoop, Laura Kovács, and Jakob Zwirchmayr. The auspicious couple: Symbolic execution and WCET analysis. In *WCET'13*, 2013.

[6] BOUML - a free UML tool box. https://bouml.fr/.

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

[8] Doxygen - Homepage. http://www.doxygen.nl/index.html.

[9] Dawson Engler and Ken Ashcraft. Racerx: effective, static detection of race conditions and deadlocks. In *SOSP'03*, New York, NY, USA, 2003. ACM Press.

[10] Björn Fiedler, Gerion Entrup, Christian Dietrich, and Daniel Lohmann. Levels of specialization in real-time operating systems. In *OSPERT'18*.

[11] FreeRTOS FAQ - What is the difference between FreeRTOS and Amazon FreeRTOS? https://freertos.org/FAQ_Amazon.html.

[12] Mike Holenderski, Reinder J. Bril, and Johan J. Lukkien. Grasp: Tracing, visualizing and measuring the behavior of real-time systems. In *in Workshop on Analysis Tools and Methodologies for Embedded and Real-time Systems*, 2010.

[13] D. Hutchins, A. Ballman, and D. Sutherland. C/c++ thread safety analysis. In *2014 IEEE 14th International Working Conference on Source Code Analysis and Manipulation*, 2014.

[14] Chris Lattner and Vikram Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *CGO'04*, Washington, DC, USA, March 2004. IEEE Computer Society Press.

[15] Alexander Lochmann, Horst Schirmeier, Hendrik Borghorst, and Olaf Spinczyk. LockDoc: Trace-based analysis of locking in the Linux kernel. In *EuroSys'19*, New York, NY, USA, March 2019. ACM Press.

[16] Mayur Naik, Chang-Seo Park, Koushik Sen, and David Gay. Effective static deadlock detection. In *ICSE'09*, ICSE '09, Washington, DC, USA, 2009. IEEE Computer Society.

[17] NDepend - Homepage. https://www.ndepend.com/.

[18] OSEK/VDX Group. OSEK implementation language specification 2.5. Technical report, OSEK/VDX Group, 2004. http://portal.osek-vdx.org/files/pdf/specs/oil25.pdf, visited 2014-09-29.

[19] OSEK/VDX Group. Operating system specification 2.2.3. Technical report, OSEK/VDX Group, February 2005. http://portal.osek-vdx.org/files/pdf/specs/os223.pdf, visited 2014-09-29.

[20] Fabian Scheler and Wolfgang Schröder-Preikschat. The RTSC: Leveraging the migration from event-triggered to time-triggered systems. In *ISORC'10*, Washington, DC, USA, May 2010. IEEE Computer Society Press.

[21] Horst Schirmeier, Matthias Bahne, Jochen Streicher, and Olaf Spinczyk. Towards eCos autoconfiguration by static application analysis. In *ACoTA'10*, Antwerp, Belgium, September 2010. CEUR-WS.org.

[22] StarUML Homepage. http://staruml.io/.

[23] Structure101 - Homepage. https://structure101.com/.

[24] Structurizr - Homepage. https://structurizr.com/help/about.

[25] Percepio Tracealyzr - Homepage. https://percepio.com/tracealyzer/.

[26] Peter Ulbrich, Rüdiger Kapitza, Christian Harkort, Reiner Schmid, and Wolfgang Schröder-Preikschat. I4Copter: An adaptable and modular quadrotor platform. In *SAC'11*, New York, NY, USA, 2011. ACM Press.

## IX. APPENDIX

In the following, the instance graphs of all tested real-world examples are shown. We decided to present them here exactly as generated by ARA. Due to their size, they are probably difficult to read on printed paper but, of course, zoomable in digital form.

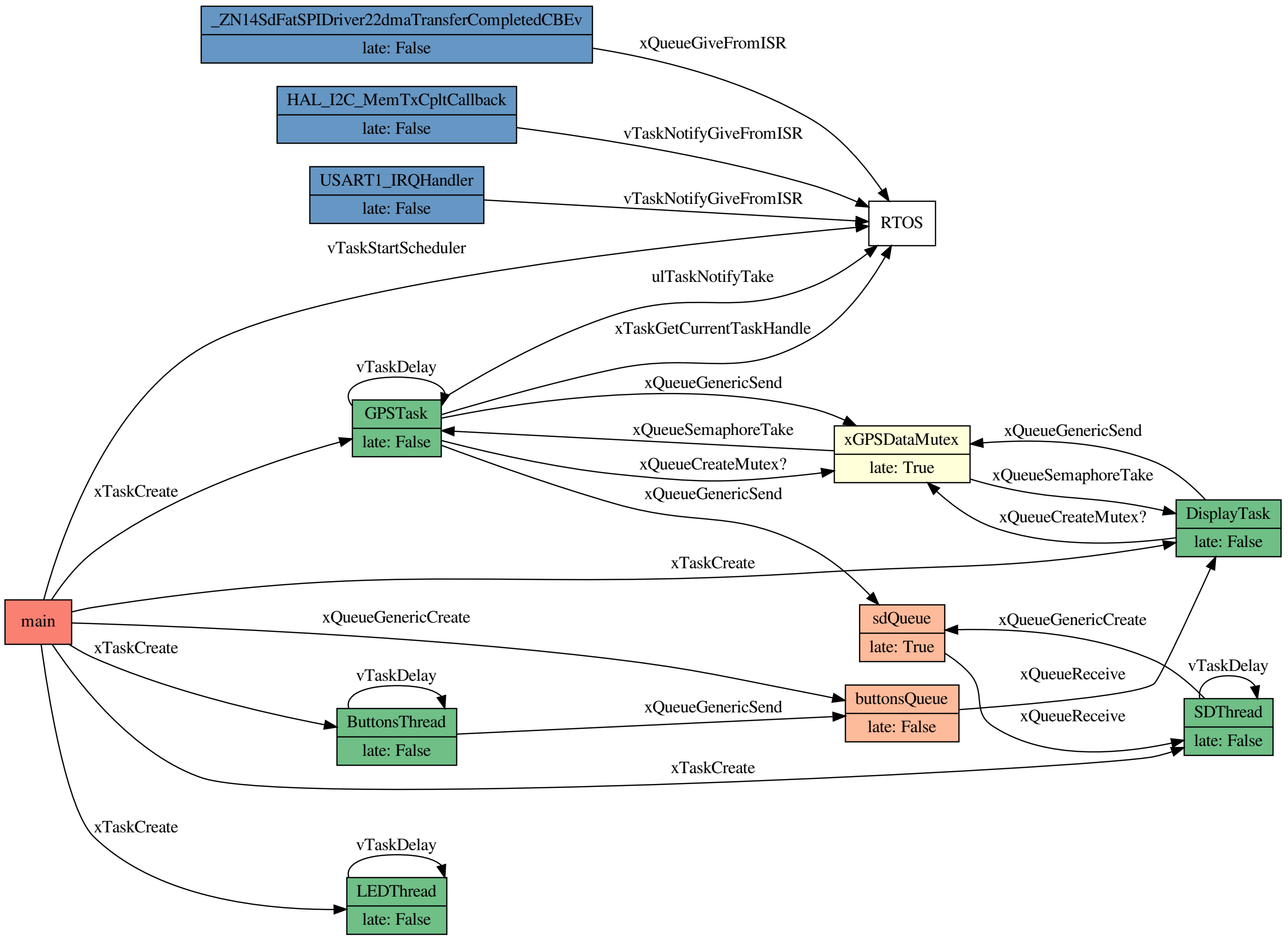

Fig. 4: GPSLogger

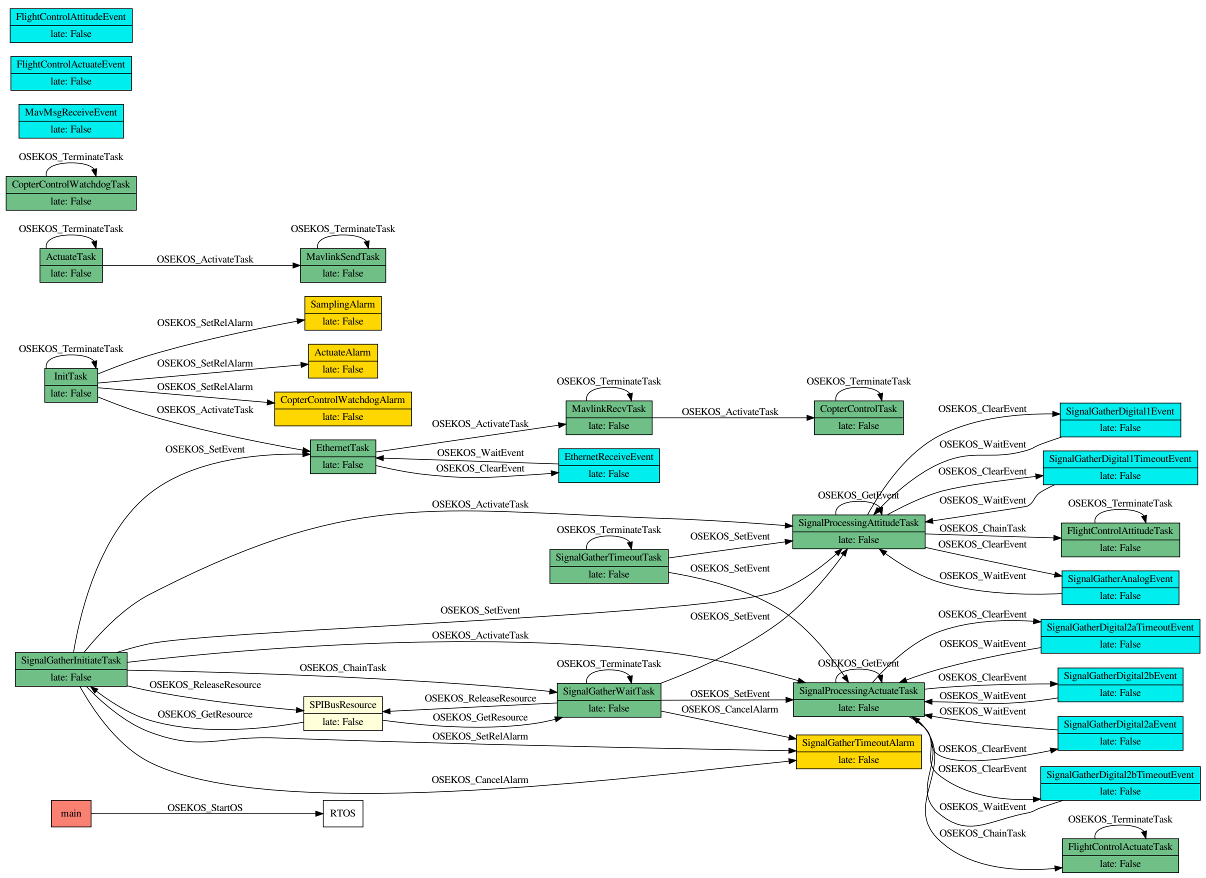

Fig. 5: I4Copter

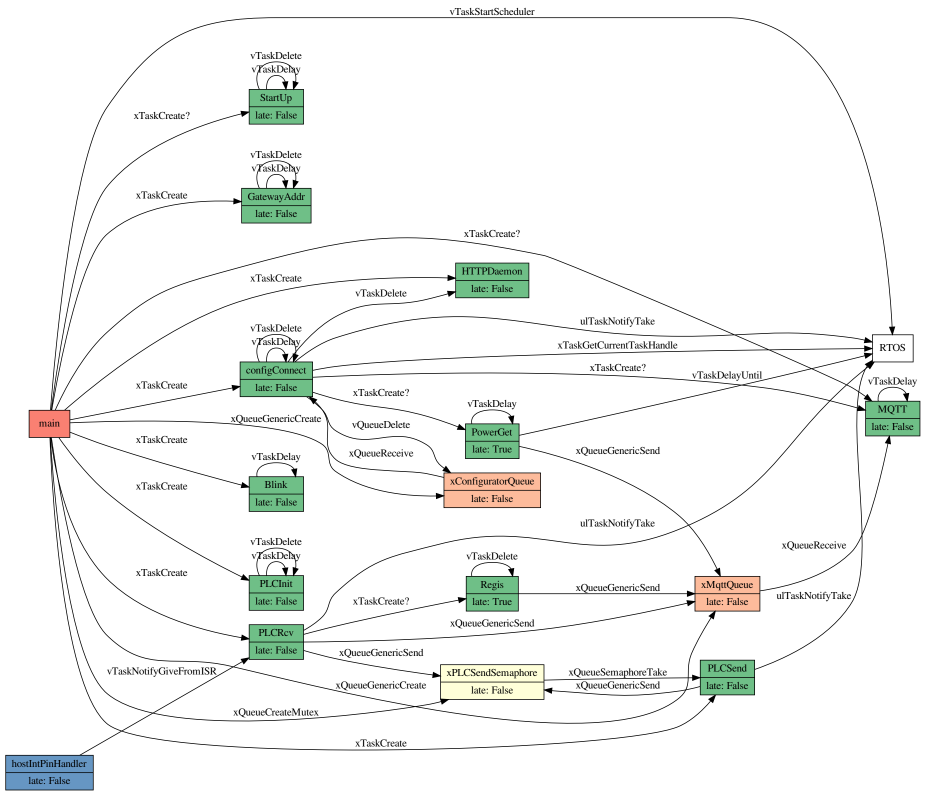

Fig. 6: Smartplug

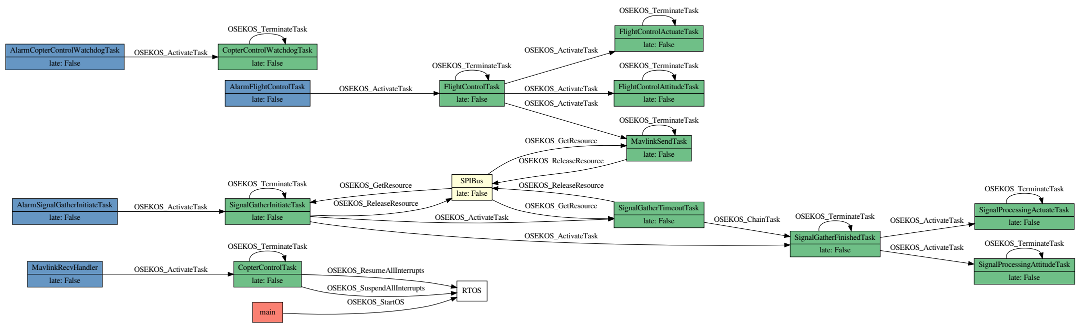

Fig. 7: I4Copter without Events