RTOS-Independent Interaction Analysis in ARA

Gerion Entrup, Jan Neugebauer, Daniel Lohmann

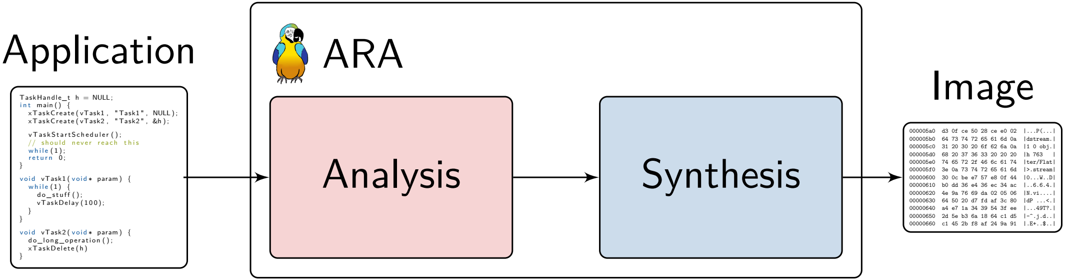
Leibniz Universität Hannover

July 05, 2022

supported by DFG

Building an RTOS-Independent RTOS-Aware Analyzer

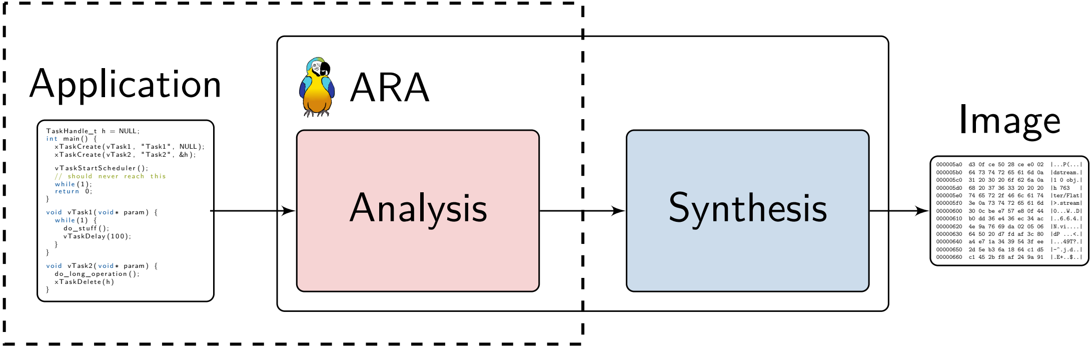
Where do we stand? We have a Whole-System-Compiler:



- Make the system startup faster: Make dynamic OS-object instantiations static.
- Eliminate RTOS-time: Drop unnecessary syscalls or calculations.

Building an RTOS-Independent RTOS-Aware Analyzer

Where do we stand? We have a Whole-System-Compiler:

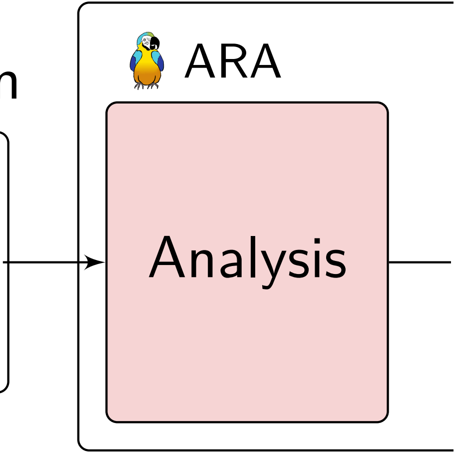


- Make the system startup faster: Make dynamic OS-object instantiations static.
- Eliminate RTOS-time: Drop unnecessary syscalls or calculations.

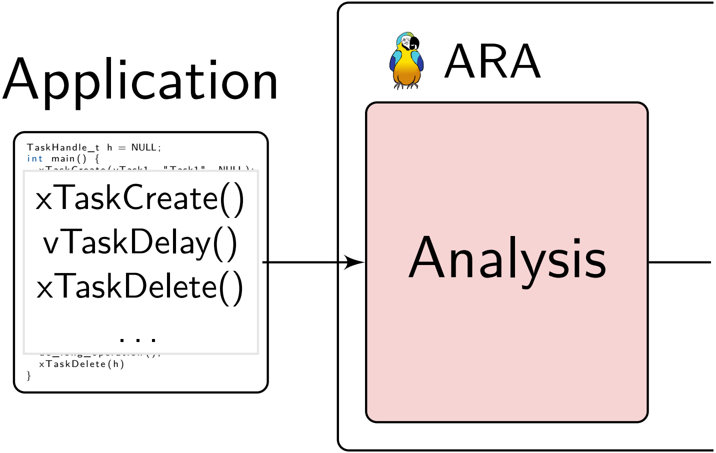
Building an RTOS-Independent RTOS-Aware Analyzer

Application

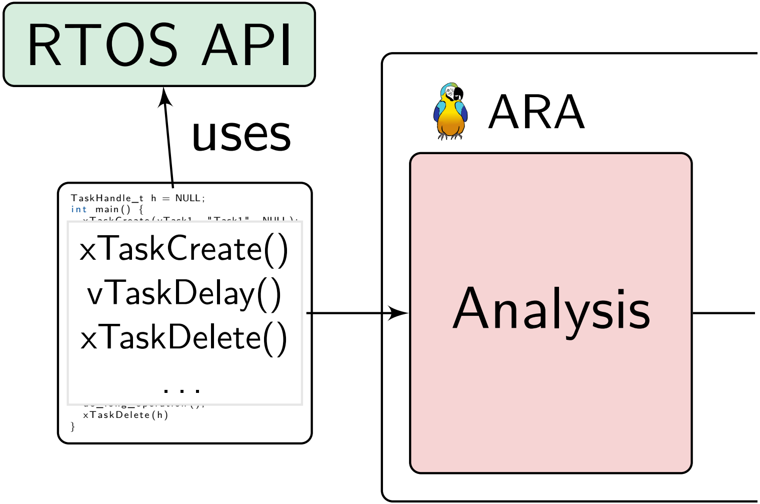
```
TaskHandle_t h = NULL;
int main() {
  xTaskCreate(vTask1, "Task1", NULL);
  xTaskCreate(vTask2, "Task2", &h);

  vTaskStartScheduler();
  // should never reach this
  while(1);
  return 0;
}

void vTask1(void* param) {
  while(1) {
    do_stuff();
    vTaskDelay(100);
  }
}

void vTask2(void* param) {
  do_long_operation();
  xTaskDelete(h)
}
```



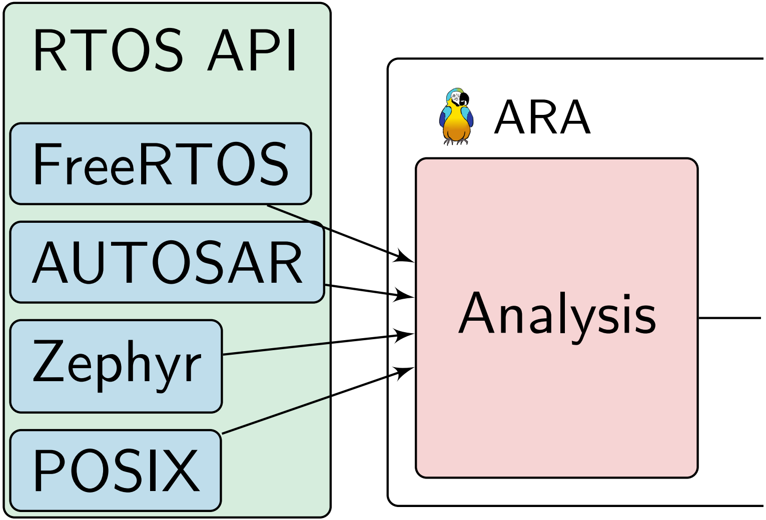
Building an RTOS-Independent RTOS-Aware Analyzer



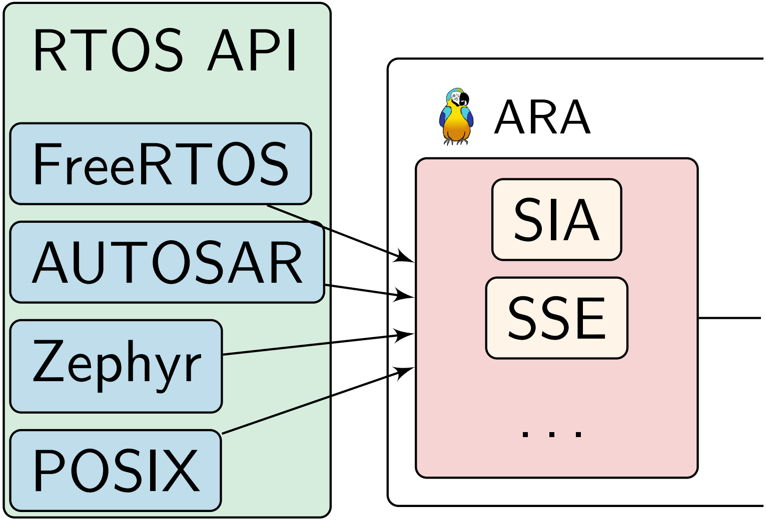
Building an RTOS-Independent RTOS-Aware Analyzer



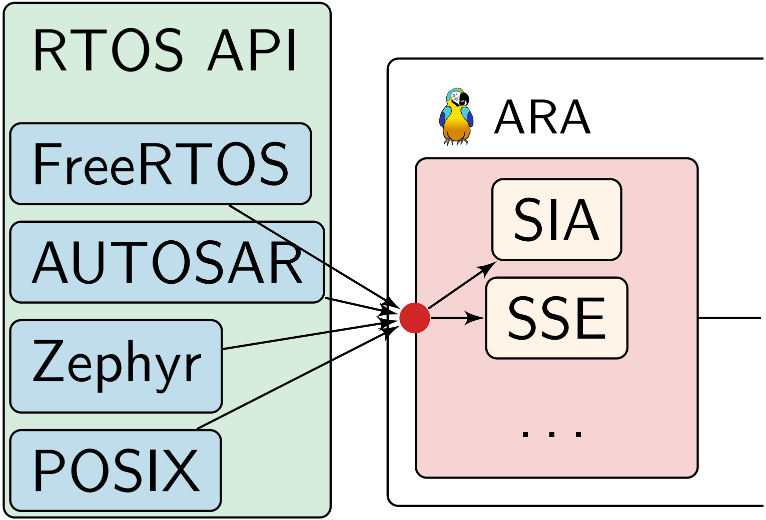
Building an RTOS-Independent RTOS-Aware Analyzer



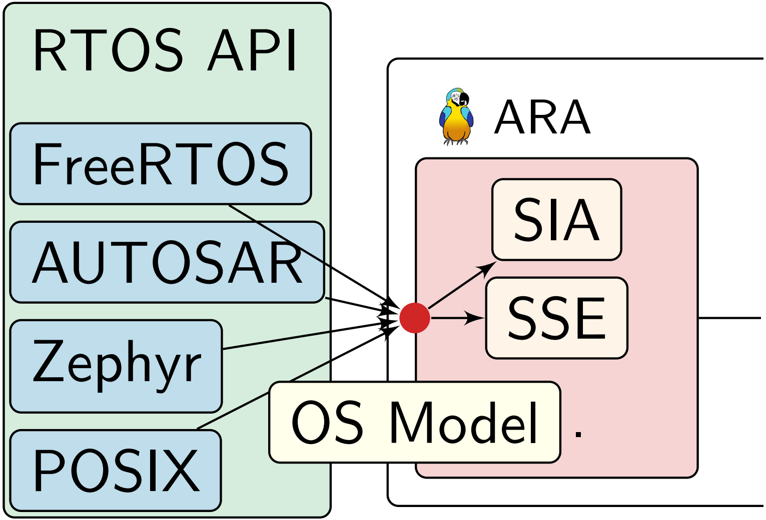
Building an RTOS-Independent RTOS-Aware Analyzer



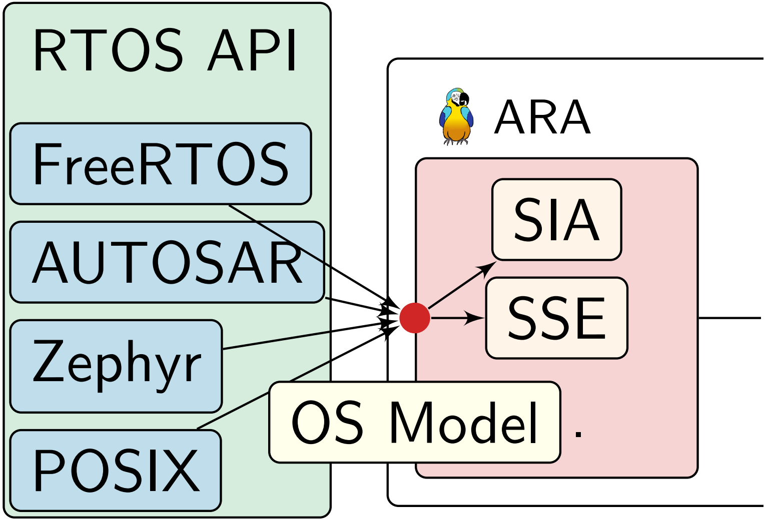
Building an RTOS-Independent RTOS-Aware Analyzer



Building an RTOS-Independent RTOS-Aware Analyzer



Building an RTOS-Independent RTOS-Aware Analyzer



- RTOS API
- Analyses Requirements
- Our Interface
- Case Studies
- Conclusion

Know your RTOS: How dynamic is the System?

	instantiation static	dyn	config
AUTOSAR	✓		DSL
FreeRTOS		✓	CPP-Macros
Zephyr	✓	✓	KConfig
POSIX		✓	-

.cpp

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

void create(Function f, int prio) {
  xTaskCreate("T " f.name(), f, prio);}

int main() {
  t1 = create(task_1, 1);
  t2 = create(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); } }
```

Know your RTOS: How dynamic is the System?

	instantiation		config
	static	dyn	
AUTOSAR	✓		DSL
FreeRTOS		✓	CPP-Macros
Zephyr	✓	✓	KConfig
POSIX		✓	-

.oil

```
TASK T1:
  PRIORITY = 1;
  SCHEDULE = FULL;

TASK T2:
  PRIORITY = 2;
  SCHEDULE = FULL;
  AUTOSTART = TRUE;

EVENT e1:
  TASK = T2;
```

.cpp

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

TASK(T2) {
  ActivateTask(T1);
  WaitEvent();
  consume(m);
}
```

A Generic Interface

Observations

- Common ground of all RTOSs: Syscalls.
- OS interaction happens *only* with syscalls.

Main Idea

- Build an abstract interpreter for syscalls.
- Calculate effects on an abstract state.

What are the exact requirements? Let's look at the algorithms.

Algorithm I – Static Instance Analysis (SIA)

Application Source

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

void create(Function f, int prio) {
  xTaskCreate("T " f.name(), f, prio);}

int main() {
  t1 = create(task_1, 1);
  t2 = create(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); } }
```

Static Instance Analysis (SIA)

Instance Graph

Algorithm I – Static Instance Analysis (SIA)

Application Source

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

void create(Function f, int prio) {
  xTaskCreate("T " f.name(), f, prio);}

int main() {
  t1 = create(task_1, 1);
  t2 = create(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); } }
```

Instance Graph

Static Instance Analysis (SIA)

1. Iterate all syscalls.

Algorithm I – Static Instance Analysis (SIA)

Application Source

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

void create(Function f, int prio) {
  xTaskCreate("T " f.name(), f, prio);}

int main() {
  t1 = create(task_1, 1);
  t2 = create(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); } }
```

freeRTOS

Context

Call Path: main() → create()

Static Instance Analysis (SIA)

1. Iterate all syscalls.
2. Find their context.

Instance Graph

Algorithm I – Static Instance Analysis (SIA)

Application Source

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

void create(Function f, int prio) {
  xTaskCreate("T " f.name(), f, prio);}

int main() {
  t1 = create(task_1, 1);
  t2 = create(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); } }
```

Context

Call Path: main() → create()

Static Instance Analysis (SIA)

1. Iterate all syscalls.
2. Find their context.
3. Calculate the syscall's effect.

Instance Graph

T task_1

Algorithm I – Static Instance Analysis (SIA)

Application Source

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

void create(Function f, int prio) {
  xTaskCreate("T " f.name(), f, prio);}

int main() {
  t1 = create(task_1, 1);
  t2 = create(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); } }
```

Context

Call Path: main() → create()

Static Instance Analysis (SIA)

1. Iterate all syscalls.
2. Find their context.
3. Calculate the syscall's effect.

Instance Graph

- T task_1
- T task_2

Algorithm I – Static Instance Analysis (SIA)

Application Source

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

void create(Function f, int prio) {
  xTaskCreate("T " f.name(), f, prio);}

int main() {
  t1 = create(task_1, 1);
  t2 = create(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); } }
```

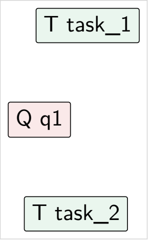
Context

Call Path: main() → create()

Static Instance Analysis (SIA)

1. Iterate all syscalls.
2. Find their context.
3. Calculate the syscall's effect.

Instance Graph



Algorithm I – Static Instance Analysis (SIA)

Application Source

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

void create(Function f, int prio) {
  xTaskCreate("T " f.name(), f, prio);}

int main() {
  t1 = create(task_1, 1);
  t2 = create(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); } }
```

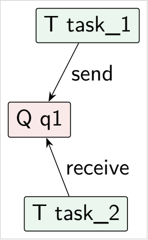
Context

Call Path: main() → create()

Static Instance Analysis (SIA)

1. Iterate all syscalls.
2. Find their context.
3. Calculate the syscall's effect.

Instance Graph



Algorithm I – Static Instance Analysis (SIA)

Application Source

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

void create(Function f, int prio) {
  xTaskCreate("T " f.name(), f, prio);}

int main() {
  t1 = create(task_1, 1);
  t2 = create(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); } }
```

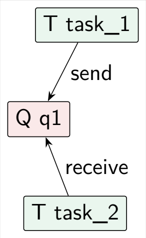
Context

Call Path: `main()` → `create()`

Static Instance Analysis (SIA)

1. Iterate all syscalls.
2. Find their context.
3. Calculate the syscall's effect.

Instance Graph



Requirements:

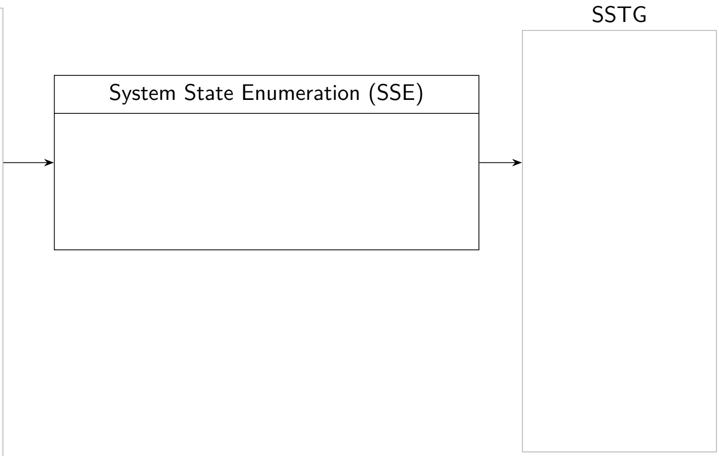
1. Identify the syscalls.
2. Identify the syscalls' effect on the Instance Graph.

Algorithm II – System State Enumeration (SSE)

Application Source

```
TASK T1:                    .oil
  PRIORITY = 1;
  SCHEDULE = FULL;

TASK T2:
  PRIORITY = 2;
  SCHEDULE = FULL;
  AUTOSTART = TRUE;

EVENT e1:
  TASK = T2;
```

```
Message m;                  .cpp

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

TASK(T2) {
  ActivateTask(T1);
  WaitEvent();
  consume(m);
}
```



Algorithm II – System State Enumeration (SSE)

Application Source

.oil

```
TASK T1:
  PRIORITY = 1;
  SCHEDULE = FULL;

TASK T2:
  PRIORITY = 2;
  SCHEDULE = FULL;
  AUTOSTART = TRUE;

EVENT e1:
  TASK = T2;
```

.cpp

```
Message m;

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

TASK(T2) {
  ActivateTask(T1);
  WaitEvent();
  consume(m);
}
```

System State Enumeration (SSE)

1. Follow the control flow (across RTOS boundaries).
2. Maintain a system state.

SSTG

Algorithm II – System State Enumeration (SSE)

Application Source

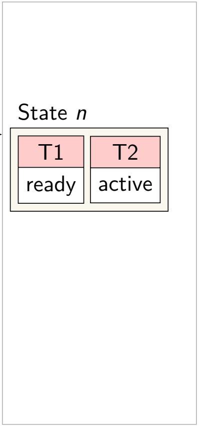
.oil
```
TASK T1:
  PRIORITY = 1;
  SCHEDULE = FULL;

TASK T2:
  PRIORITY = 2;
  SCHEDULE = FULL;
  AUTOSTART = TRUE;

EVENT e1:
  TASK = T2;
```

.cpp
```
Message m;

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

TASK(T2) {
  ActivateTask(T1);
  WaitEvent();
  consume(m);
}
```

System State Enumeration (SSE)

1. Follow the control flow (across RTOS boundaries).
2. Maintain a system state.

SSTG



Algorithm II – System State Enumeration (SSE)

Application Source

AUTOSAR

```
TASK T1:                          .oil
  PRIORITY = 1;
  SCHEDULE = FULL;

TASK T2:
  PRIORITY = 2;
  SCHEDULE = FULL;
  AUTOSTART = TRUE;

EVENT e1:
  TASK = T2;
```

```
Message m;                        .cpp

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

TASK(T2) {
  ActivateTask(T1);
  WaitEvent();
  consume(m);
}
```

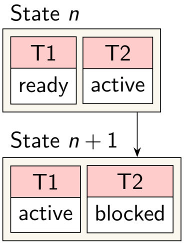
→

System State Enumeration (SSE)

1. Follow the control flow (across RTOS boundaries).
2. Maintain a system state.

→

SSTG



Algorithm II – System State Enumeration (SSE)

Application Source

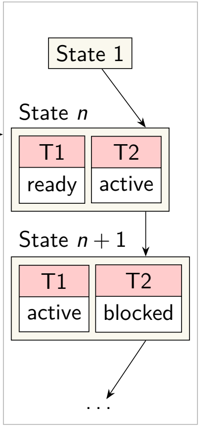
.oil
```
TASK T1:
  PRIORITY = 1;
  SCHEDULE = FULL;

TASK T2:
  PRIORITY = 2;
  SCHEDULE = FULL;
  AUTOSTART = TRUE;

EVENT e1:
  TASK = T2;
```

.cpp
```
Message m;

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

TASK(T2) {
  ActivateTask(T1);
  WaitEvent();
  consume(m);
}
```

System State Enumeration (SSE)

1. Follow the control flow (across RTOS boundaries).
2. Maintain a system state.
3. Capture states in a graph.

SSTG



Algorithm II – System State Enumeration (SSE)

Application Source

.oil — AUTOSAR

```
TASK T1:
  PRIORITY = 1;
  SCHEDULE = FULL;

TASK T2:
  PRIORITY = 2;
  SCHEDULE = FULL;
  AUTOSTART = TRUE;

EVENT e1:
  TASK = T2;
```

.cpp

```
Message m;

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

TASK(T2) {
  ActivateTask(T1);
  WaitEvent();
  consume(m);
}
```

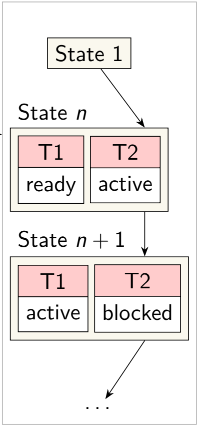
System State Enumeration (SSE)

1. Follow the control flow (across RTOS boundaries).
2. Maintain a system state.
3. Capture states in a graph.

Requirements:

1. Identify syscalls.
2. Identify the syscalls' effect on the system state.

SSTG



Our OS-Model

Algorithmic Requirements:

1. Identify syscalls.
2. Express the syscall's effect
   - on the Instance Graph (SIA).
   - on the System State (SSE).

Our OS-Model

Algorithmic Requirements:

1. Identify syscalls.
2. Express the syscall's effect
   - on the Instance Graph (SIA).
   - on the System State (SSE).

Additional requirements:

3. Support multicore applications.
4. Be greedy (as detailed as possible).

Our OS-Model

Algorithmic Requirements:

1. Identify syscalls.
2. Express the syscall's effect
   - on the Instance Graph (SIA).
   - on the System State (SSE).

Additional requirements:

3. Support multicore applications.
4. Be greedy (as detailed as possible).

Our Approach

Define an OS interpreter on a combined state.

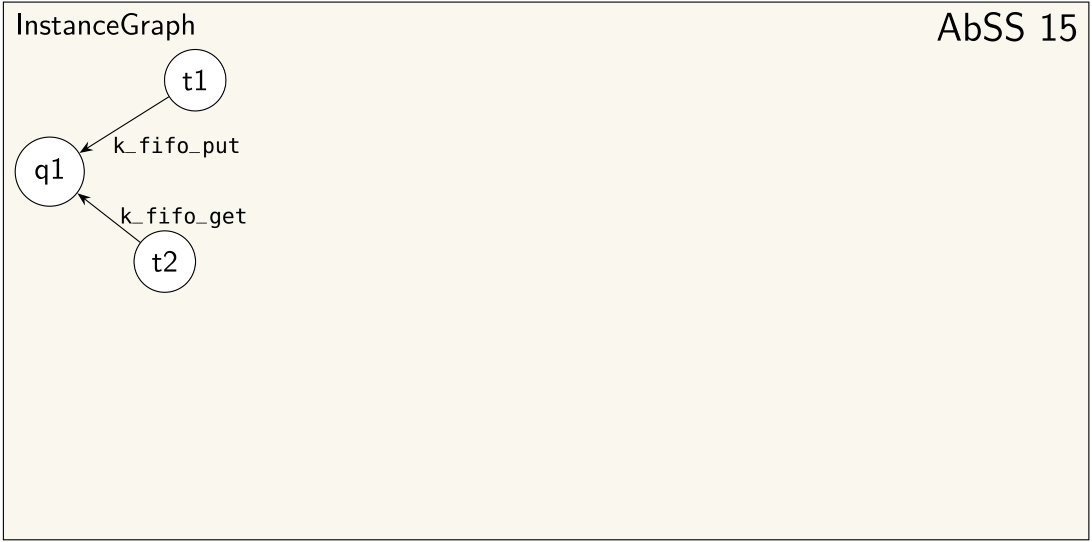
The model needs:

- A list of syscalls
  - Name
  - Arguments
  - An `interpret()` function
- A system state
  - Contains Instance Graph
  - Contains OS-object contexts
  - Contains multiple CPU states

The Abstract System State

AbSS 15

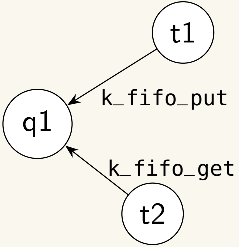
The Abstract System State



The Abstract System State

AbSS 15

InstanceGraph



CPU 0
IRQ: on
Instance: t1
IP: ABB 5 (line 5)
Call Path: *t1_action*
Status: syscall

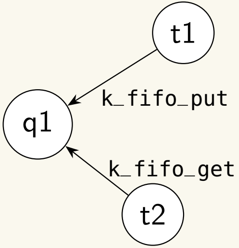
CPU 1
IRQ: off
Instance: t2
IP: ABB 10 (line 10)
Call Path: *t2_action*
Status: computation

...

The Abstract System State

AbSS 15

InstanceGraph



CPU 0
IRQ: on
Instance: t1
IP: ABB 5 (line 5)
Call Path: *t1_action*
Status: syscall

CPU 1
IRQ: off
Instance: t2
IP: ABB 10 (line 10)
Call Path: *t2_action*
Status: computation

...

t1 context
Status: running
IP: ABB 5 (line 5)
Call Path: *t1_action*

t2 context
Status: active
IP: ABB 10 (line 10)
Call Path: *t2_action*

q1 context
Elements: 4

...

SetEvent: An Example

```
@syscall(categories={SyscallCategory.comm},
         signature=(Arg("task", ty=Task, hint=SigType.instance),
                    Arg("event_mask")))
def SetEvent(cfg, state, cpu_id, args, va):
  state = state.copy()
  # - store event in event mask
  # - set other tast ready (wake up), if necessary
  # - add interaction into the instance graph
  return state
```

Validation – Applications

FreeRTOS

- **GPSLogger**, geolocation logging
- **LibrePilot**, quadrocopter firmware

AUTOSAR

- **i4copter**, quadrocopter firmware

POSIX

- **libmicrohttpd**, HTTP server

Zephyr

- **app_kernel**, benchmark application
- **sys_kernel**, benchmark application

Validation – Results

RTOS		SIA Obj	SIA Int	SSE
FreeRTOS	GPSLogger	✔	✔*	
	LibrePilot	✔	✔*	
Zephyr	app_kernel	✔	✔*	
	sys_kernel	✔	✔	
AUTOSAR	i4copter		✔	✔
POSIX	libmicrohttpd	✔*	✔*	

Validation – Results

RTOS		SIA Obj	SIA Int	SSE
FreeRTOS	GPSLogger	✔	✔*	
	LibrePilot	✔	✔*	
Zephyr	app_kernel	✔	✔*	
	sys_kernel	✔	✔	
AUTOSAR	i4copter		✔	✔
POSIX	libmicrohttpd	✔*	✔*	

Found interactions: 12 of 15

Problem value analyzer:
C++ wrapper class prevents instance retrieval

Validation – Results

RTOS		SIA Obj	SIA Int	SSE
FreeRTOS	GPSLogger	✔	✔*	
	LibrePilot	✔	✔*	
Zephyr	app_kernel	✔	✔*	
	sys_kernel	✔	✔	
AUTOSAR	i4copter		✔	✔
POSIX	libmicrohttpd	✔*	✔*	

Found interactions: 60 of 62

Problem value analyzer:
Dynamic assignment to an array

Validation – Results

RTOS		SIA Obj	SIA Int	SSE
FreeRTOS	GPSLogger	✔	✔*	
	LibrePilot	✔	✔*	
Zephyr	app_kernel	✔	✔*	
	sys_kernel	✔	✔	
AUTOSAR	i4copter		✔	✔
POSIX	libmicrohttpd	✔*	✔*	

Found interactions: 60 of 62

Problem value analyzer:
Dynamic assignment to an array

Conclusion

- Results not always complete but sound
- Analyses are incomplete not the OS model

Conclusion

- Problem: Make RTOS-aware analyses RTOS-independent
- Solution: Collect the RTOS specific parts within a model
- Central Design Decision: RTOS interpreter on an abstract state
- Validated for FreeRTOS, AUTOSAR, Zephyr and POSIX with 6 applications

Source: `https://github.com/luhsra/ara`

Thank you! Do you have questions?