PLOS17

Annotations in Operating Systems with Custom AspectC++ Attributes

Birte Friesel, Markus Buschhoff, Olaf Spinczyk

TU Dortmund, Computer Science XII
Embedded System Software group

Oct 28th, 2017

Annotations in C/C++

- Annotations add data or functions to code segments

Example (Align array to 16-Byte boundary)

```
__attribute__((align(16))) int x[] = { ... };
```

Annotations in C/C++

- Annotations add data or functions to code segments

Example (Align array to 16-Byte boundary)

```
__attribute__((align(16))) int x[] = { ... };
```

- Not included in C and (pre-2011) C++ standards
  - Many compilers still support attributes
  - Syntax and set of attributes depend on compiler and architecture

Annotations in C/C++

- Annotations add data or functions to code segments

Example (Align array to 16-Byte boundary)

```
__attribute__((align(16))) int x[] = { ... };
```

- Not included in C and (pre-2011) C++ standards
  - Many compilers still support attributes
  - Syntax and set of attributes depend on compiler and architecture
- C++11 defines attribute syntax and two attributes (seven in C++17)
  - `[[noreturn]]` → function does not return
  - `[[carries_depenency]]` → memory access ordering control

Annotations in C/C++

- Annotations add data or functions to code segments

Example (Align array to 16-Byte boundary)

```
__attribute__((align(16))) int x[] = { ... };
```

- Not included in C and (pre-2011) C++ standards
  - Many compilers still support attributes
  - Syntax and set of attributes depend on compiler and architecture
- C++11 defines attribute syntax and two attributes (seven in C++17)
  - `[[noreturn]]` $\rightarrow$ function does not return
  - `[[carries_depenency]]` $\rightarrow$ memory access ordering control
- $\approx$ 100 compiler-specific attributes

Annotations in C/C++

- Annotations add data or functions to code segments

Example (Align array to 16-Byte boundary)

```
__attribute__((align(16))) int x[] = { ... };
```

- Not included in C and (pre-2011) C++ standards
  - Many compilers still support attributes
  - Syntax and set of attributes depend on compiler and architecture
- C++11 defines attribute syntax and two attributes (seven in C++17)
  - `[[noreturn]]` $\rightarrow$ function does not return
  - `[[carries_depenency]]` $\rightarrow$ memory access ordering control
- $\approx$ 100 compiler-specific attributes
- Standard does not specify custom annotations or custom attribute behaviour

Custom annotations

- Not widely supported in C/C++ compilers
  - Support present in MSVC++, but not gcc/g++/clang
  - Available in Java, Python, C#

Custom annotations

- Not widely supported in C/C++ compilers
  - Support present in MSVC++, but not gcc/g++/clang
  - Available in Java, Python, C#

Example (Custom annotations for function and variable tracing)

```
[[tracing::call]] void some_func();      →  "[DEBUG] Running void some_func()"
[[tracing::write]] int num_threads;      →  "[DEBUG] num_threads set to 123"
```

Custom annotations

- Not widely supported in C/C++ compilers
  - Support present in MSVC++, but not gcc/g++/clang
  - Available in Java, Python, C#

Example (Custom annotations for function and variable tracing)

```
[[tracing::call]] void some_func();      →  "[DEBUG] Running void some_func()"
[[tracing::write]] int num_threads;      →  "[DEBUG] num_threads set to 123"
```

- AspectC++ 2.2 introduces support for custom annotations
  - Transforms AspectC++ to C++
  - supports arbitrary C++-compatible backend compilers, including clang and g++

Custom annotations

- Not widely supported in C/C++ compilers
  - Support present in MSVC++, but not gcc/g++/clang
  - Available in Java, Python, C#

Example (Custom annotations for function and variable tracing)

```
[[tracing::call]] void some_func();      →  "[DEBUG] Running void some_func()"
[[tracing::write]] int num_threads;      →  "[DEBUG] num_threads set to 123"
```

- AspectC++ 2.2 introduces support for custom annotations
  - Transforms AspectC++ to C++
  - supports arbitrary C++-compatible backend compilers, including clang and g++

→ Discussion of use cases of custom annotations in OS development

Contents

Aspect-Oriented Programming (AOP)

- AspectC++ is a C++ extension for AOP [SL07]
- Aspect: piece of code affecting (many) other system modules

Aspect-Oriented Programming (AOP)

- AspectC++ is a C++ extension for AOP [SL07]
- Aspect: piece of code affecting (many) other system modules

Example (Function call tracing)

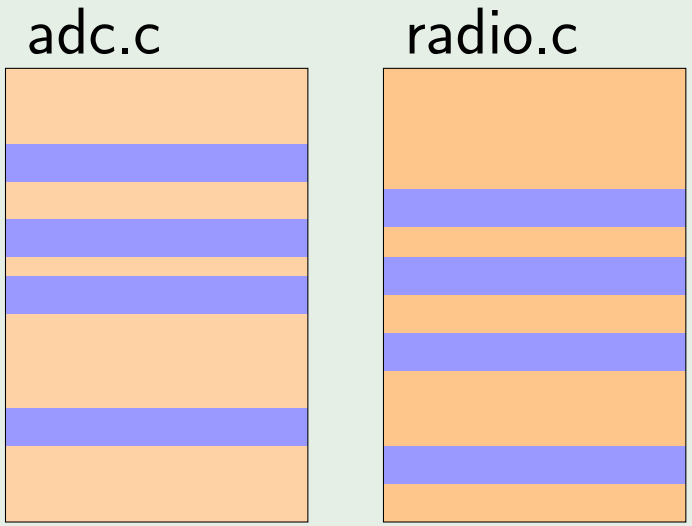
Components:

Aspect-Oriented Programming (AOP)

- AspectC++ is a C++ extension for AOP [SL07]
- Aspect: piece of code affecting (many) other system modules

Example (Function call tracing)

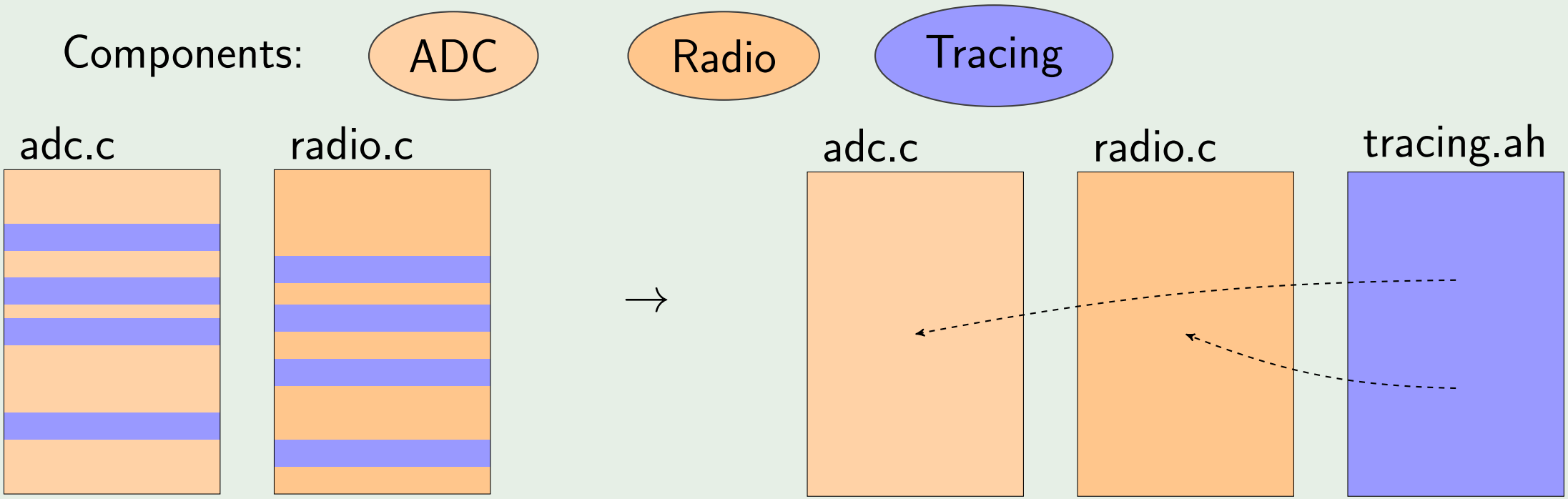
Components: ADC, Radio, Tracing



Aspect-Oriented Programming (AOP)

- AspectC++ is a C++ extension for AOP [SL07]
- Aspect: piece of code affecting (many) other system modules

Example (Function call tracing)



- Tracing code is **woven** into ADC and radio

AspectC++

- Uses compile-time weaving $\rightarrow$ suitable for OS development
  - Runtime weaving would cause runtime overhead

AspectC++

- Uses compile-time weaving → suitable for OS development
  - Runtime weaving would cause runtime overhead
- Allows definition of custom C++-style attributes

Example (Tracing attribute: definition)

```
// attributes.ah
namespace debug { attribute log(); }
```

```
// driver.h
[[tracing::call]] void
foo();
```

AspectC++

- Uses compile-time weaving → suitable for OS development
  - Runtime weaving would cause runtime overhead
- Allows definition of custom C++-style attributes

Example (Tracing attribute: definition)

```
// attributes.ah
namespace debug { attribute log(); }
```

```
// driver.h
[[tracing::call]] void
foo();
```

- ... and custom attribute behaviour
  - Using developer-provided advice on attribute joinpoints

Example (Tracing attribute: implementation in aspect)

```
advice execution(tracing::call()) :  before() {
  uart << "Running " << tjp->signature() << endl;
}
```

AspectC++

- Uses compile-time weaving → suitable for OS development
  - Runtime weaving would cause runtime overhead
- Allows definition of custom C++-style attributes

Example (Tracing attribute: definition)

```
// attributes.ah
namespace debug { attribute log(); }
```

```
// driver.h
[[tracing::call]] void
foo();
```

- ... and custom attribute behaviour
  - Using developer-provided advice on attribute joinpoints

Example (Tracing attribute: implementation in aspect)

```
advice execution(tracing::call()) :  before() {
  uart << "Running " << tjp->signature() << endl;
}
```

AspectC++

- Uses compile-time weaving → suitable for OS development
  - Runtime weaving would cause runtime overhead
- Allows definition of custom C++-style attributes

Example (Tracing attribute: definition)

```
// attributes.ah
namespace debug { attribute log(); }
```

```
// driver.h
[[tracing::call]] void
foo();
```

- ... and custom attribute behaviour
  - Using developer-provided advice on attribute joinpoints

Example (Tracing attribute: implementation in aspect)

```
advice execution(tracing::call()) :  before() {
  uart << "Running " << tjp->signature() << endl;
}
```

Contents

Portable Compiler Attributes

- Developer-provided aspects control behaviour of custom attributes
  → independent of backend compiler
- simple example: `[[tracing::call]]` → tracing of function calls

Portable Compiler Attributes

- Developer-provided aspects control behaviour of custom attributes
  → independent of backend compiler
- simple example: `[[tracing::call]]` → tracing of function calls
- Standard attribute features can be extended
- "`[[deprecated]]` indicates that the use [...] is allowed, but discouraged"
  - Commonly implemented as compile-time warning

Portable Compiler Attributes

- Developer-provided aspects control behaviour of custom attributes
  → independent of backend compiler
- simple example: `[[tracing::call]]` → tracing of function calls
- Standard attribute features can be extended
- "`[[deprecated]]` indicates that the use [...] is allowed, but discouraged"
  - Commonly implemented as compile-time warning
  - No support for runtime warnings
  - Cannot silence warnings in legacy code

Portable Compiler Attributes

- Developer-provided aspects control behaviour of custom attributes
  → independent of backend compiler
- simple example: `[[tracing::call]]` → tracing of function calls
- Standard attribute features can be extended
- "`[[deprecated]]` indicates that the use [...] is allowed, but discouraged"
  - Commonly implemented as compile-time warning
  - No support for runtime warnings
  - Cannot silence warnings in legacy code

Example (Flexible function deprecation)

```
[[Attr::deprecated]] void f(void *data);
[[Attr::legacy]] void old_impl();

void old_impl()   { /* ... */ f(data); /* ... */ }
void new_helper() { /* ... */ f(data); /* ... */ }
```

Portable Compiler Attributes

- Developer-provided aspects control behaviour of custom attributes
  → independent of backend compiler
- simple example: `[[tracing::call]]` → tracing of function calls
- Standard attribute features can be extended
- "`[[deprecated]]` indicates that the use [...] is allowed, but discouraged"
  - Commonly implemented as compile-time warning
  - No support for runtime warnings
  - Cannot silence warnings in legacy code

Example (Flexible function deprecation)

```
[[Attr::deprecated]] void f(void *data);
[[Attr::legacy]] void old_impl();

void old_impl()   { /* ... */ f(data); /* ... */ } // OK
void new_helper() { /* ... */ f(data); /* ... */ }
```

Portable Compiler Attributes

- Developer-provided aspects control behaviour of custom attributes
  → independent of backend compiler
- simple example: `[[tracing::call]]` → tracing of function calls
- Standard attribute features can be extended
- "`[[deprecated]]` indicates that the use [...] is allowed, but discouraged"
  - Commonly implemented as compile-time warning
  - No support for runtime warnings
  - Cannot silence warnings in legacy code

Example (Flexible function deprecation)

```
[[Attr::deprecated]] void f(void *data);
[[Attr::legacy]] void old_impl();

void old_impl()   { /* ... */ f(data); /* ... */ } // OK
void new_helper() { /* ... */ f(data); /* ... */ } // not OK
```

Portable Compiler Attributes: Implementation

- AspectC++ provides checks to determine calling function

Example (Runtime warning for deprecated functions)

```
advice call(Attr::deprecated()) && !within(Attr::legacy())
    :  before() {
  uart << tjp->signature() << " is deprecated" << endl;
}
```

Portable Compiler Attributes: Implementation

- AspectC++ provides checks to determine calling function

Example (Runtime warning for deprecated functions)

```
advice call(Attr::deprecated()) && !within(Attr::legacy())
    :  before() {
  uart << tjp->signature() << " is deprecated" << endl;
}
```

- Other possibility: `[[OS::serialize]]` → prohibit concurrent execution
  - Behaviour selected by aspect as appropriate (disable interrupts, lock bottom half, . . .)

Portable Compiler Attributes: Implementation

- AspectC++ provides checks to determine calling function

Example (Runtime warning for deprecated functions)

```
advice call(Attr::deprecated()) && !within(Attr::legacy())
    :  before() {
  uart << tjp->signature() << " is deprecated" << endl;
}
```

- Other possibility: `[[OS::serialize]]` → prohibit concurrent execution
  - Behaviour selected by aspect as appropriate (disable interrupts, lock bottom half, . . .)

→ Developers **express intentions** by annotating code fragments

- Aspects implement them in a portable way

Portable Compiler Attributes: Implementation

- AspectC++ provides checks to determine calling function

Example (Runtime warning for deprecated functions)

```
advice call(Attr::deprecated()) && !within(Attr::legacy())
    :  before() {
  uart << tjp->signature() << " is deprecated" << endl;
}
```

- Other possibility: `[[OS::serialize]]` → prohibit concurrent execution
  - Behaviour selected by aspect as appropriate (disable interrupts, lock bottom half, . . .)

→ Developers **express intentions** by annotating code fragments

  - Aspects implement them in a portable way
- Limitation: Cannot affect code generation by compiler
  - Custom attributes cannot set alignment, binary sections, . . .

Operating System APIs

- Annotations can also **describe behaviour** and serve as API

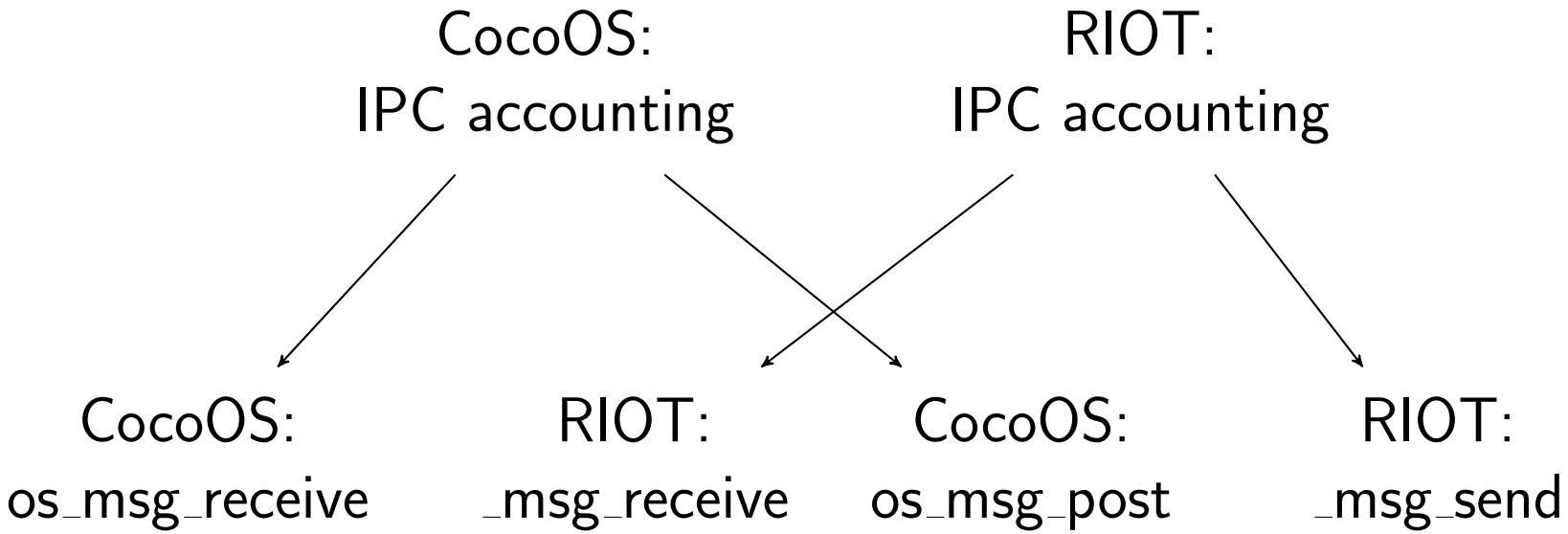


Figure: Interfacing with system components

Operating System APIs

- Annotations can also **describe behaviour** and serve as API

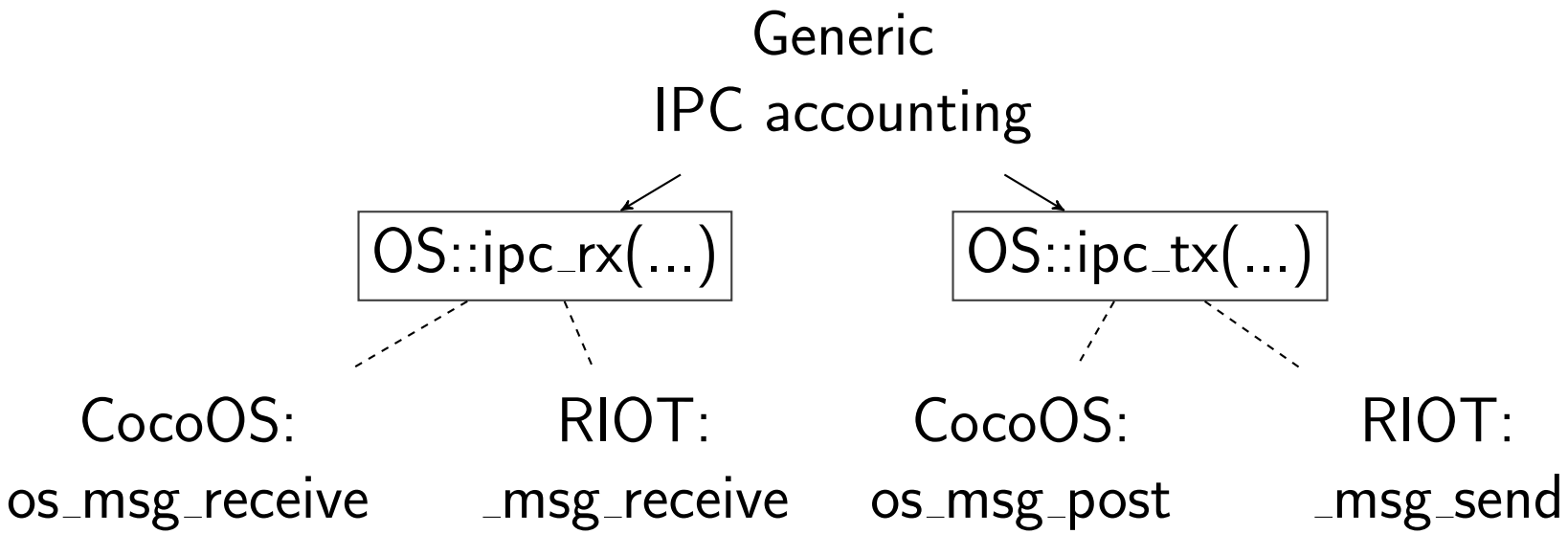


Figure: Interfacing with system components

Operating System APIs

- Annotations can also **describe behaviour** and serve as API

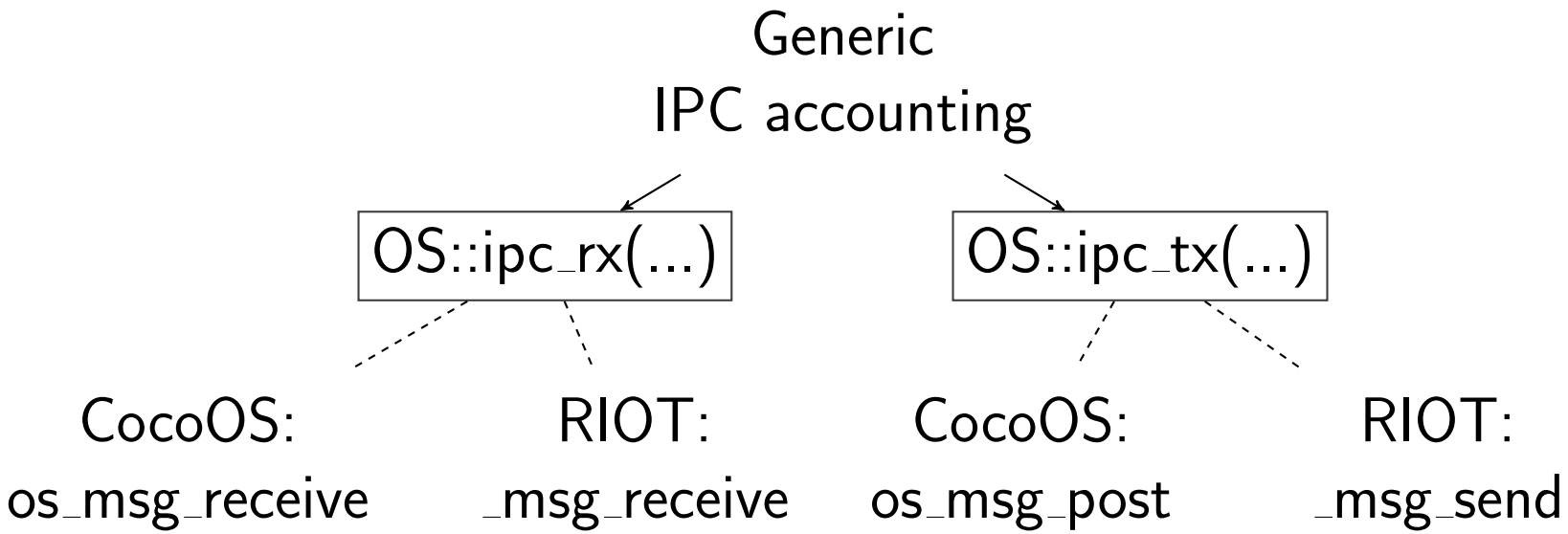


Figure: Interfacing with system components

→ Re-usable aspects across annotated systems
  - Details (parameter holding IPC message length, . . .) can be specified via attribute parameters

Operating System APIs

- Annotations can also **describe behaviour** and serve as API

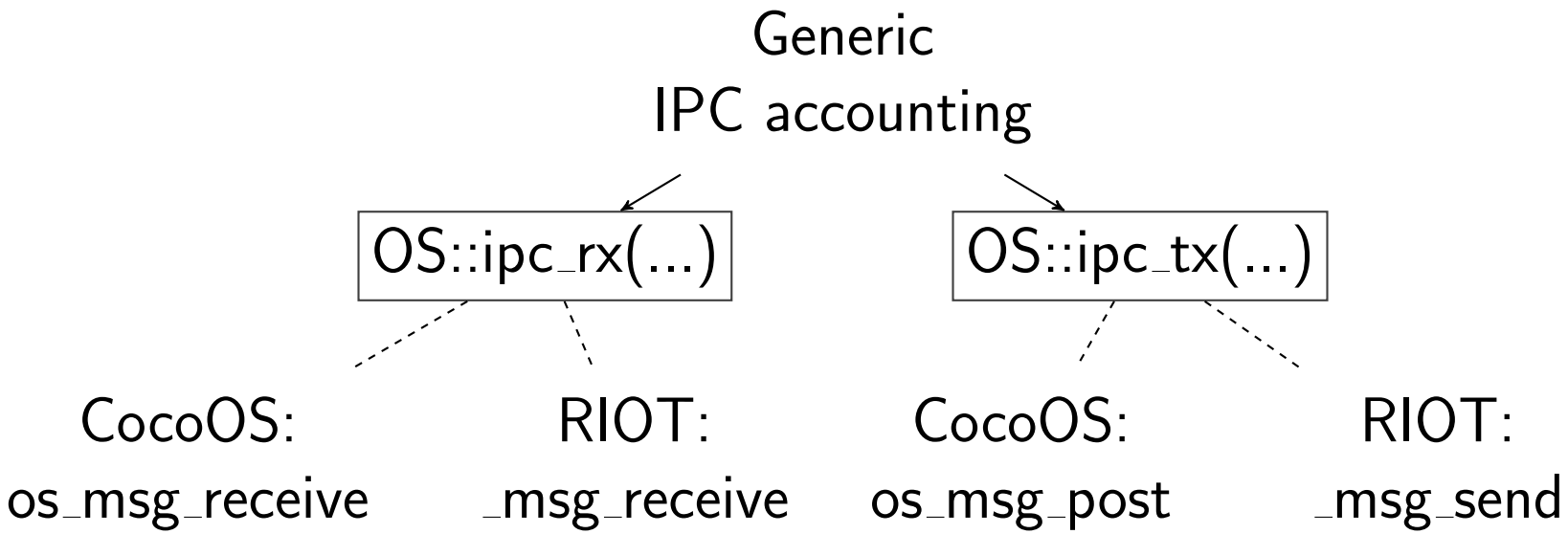


Figure: Interfacing with system components

→ Re-usable aspects across annotated systems
  - Details (parameter holding IPC message length, . . .) can be specified via attribute parameters
- Additional machine-readable documentation layer

Plugins for System APIs

- Modules can hook into annotation API
  - System provides annotated function stubs or default implementation
  - Modules use attribute advices to replace them

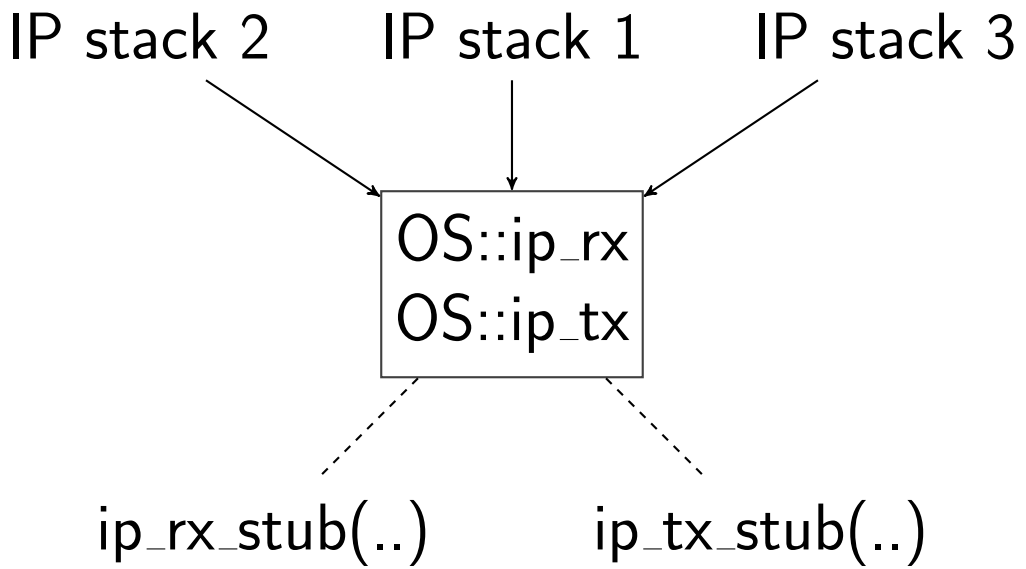


Figure: Plugin API for IP implementations

Plugins for System APIs

- Modules can hook into annotation API
  - System provides annotated function stubs or default implementation
  - Modules use attribute advices to replace them
- Easy to evaluate different modules
  - No system changes required once stub functions are present

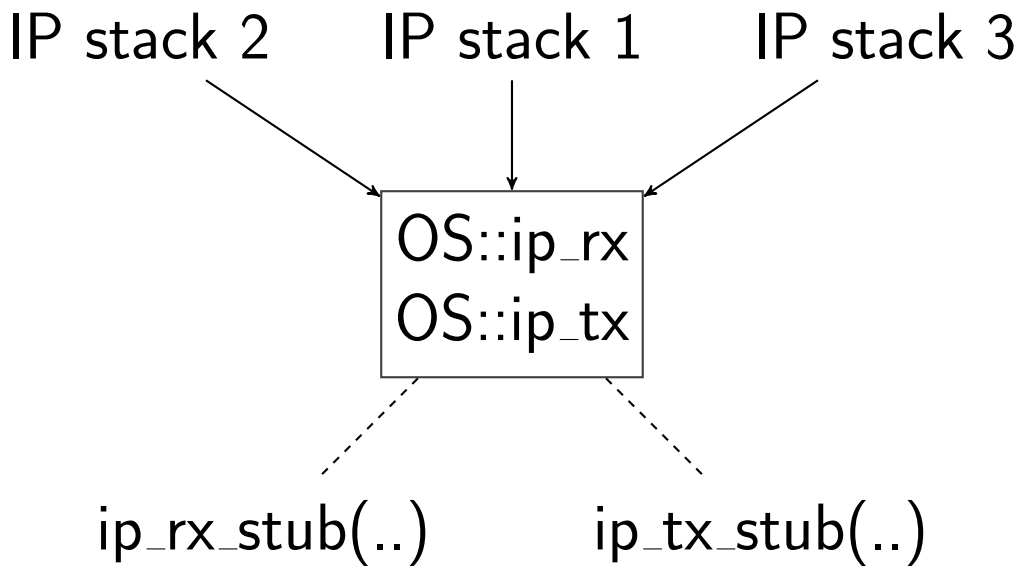


Figure: Plugin API for IP implementations

Plugins for System APIs

- Modules can hook into annotation API
  - System provides annotated function stubs or default implementation
  - Modules use attribute advices to replace them
- Easy to evaluate different modules
  - No system changes required once stub functions are present
- Modules need to match annotations
  - Moves adaption responsibility from system to modules
  - Helps writing clean system code

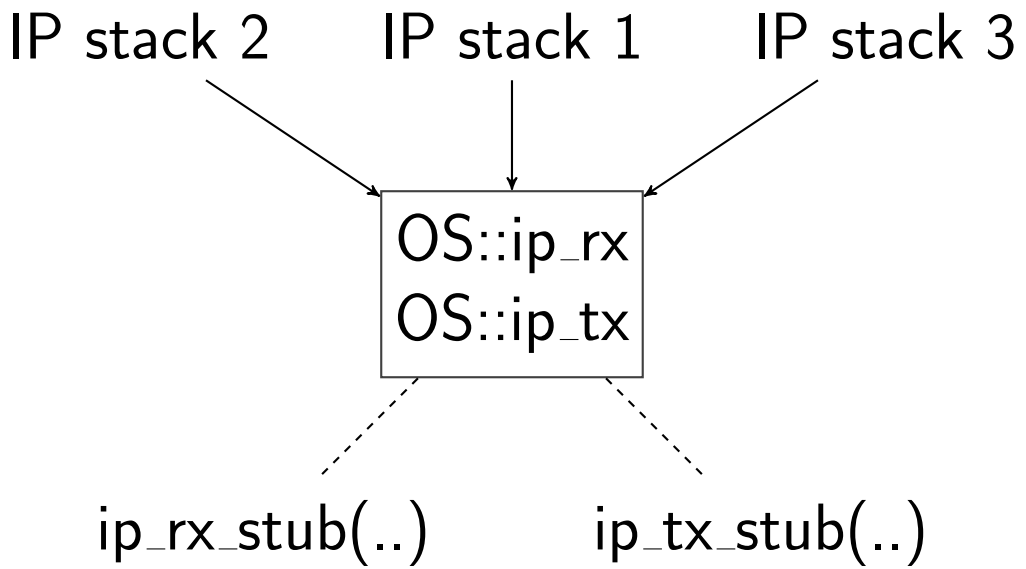


Figure: Plugin API for IP implementations

Modelling

```
void disable(); void enable();
int send(char *data, u8 len);
void txDone();
```

$\leftrightarrow$

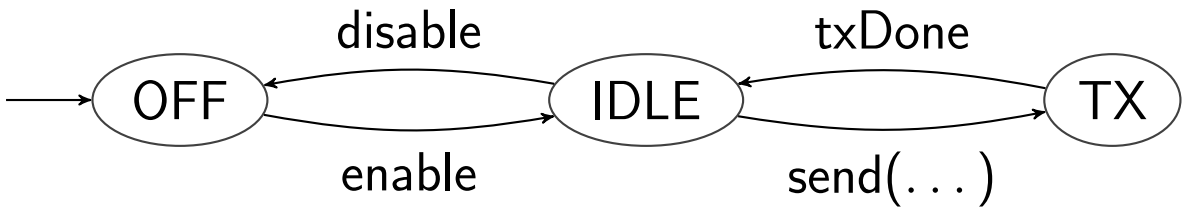


Figure: Simplified model for a radio driver

- System components frequently correspond to models
  - Task state transitions in scheduler, hardware states and energy consumption, ...
  - Model may be needed at run-time
- function call $\hat{=}$ model transition

Modelling

```
void disable(); void enable();
int send(char *data, u8 len);
void txDone();
```

$\leftrightarrow$

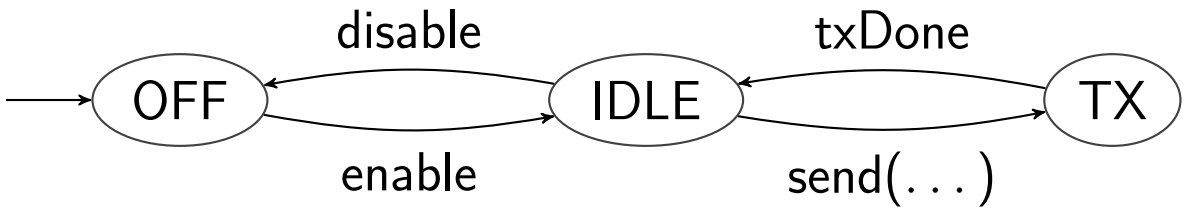


Figure: Simplified model for a radio driver

- System components frequently correspond to models
  - Task state transitions in scheduler, hardware states and energy consumption, ...
  - Model may be needed at run-time
- function call $\hat{=}$ model transition
- Challenge: synchronization model $\leftrightarrow$ code
  - Model driven development addresses model adjustment $\rightarrow$ implementation update

Modelling

```
void disable(); void enable();
int send(char *data, u8 len);
void txDone();
```

$\leftrightarrow$

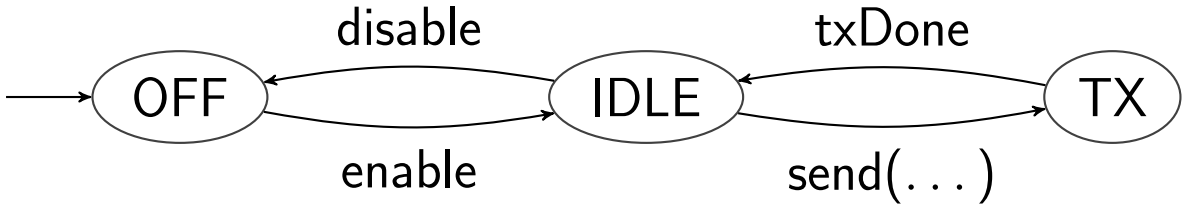


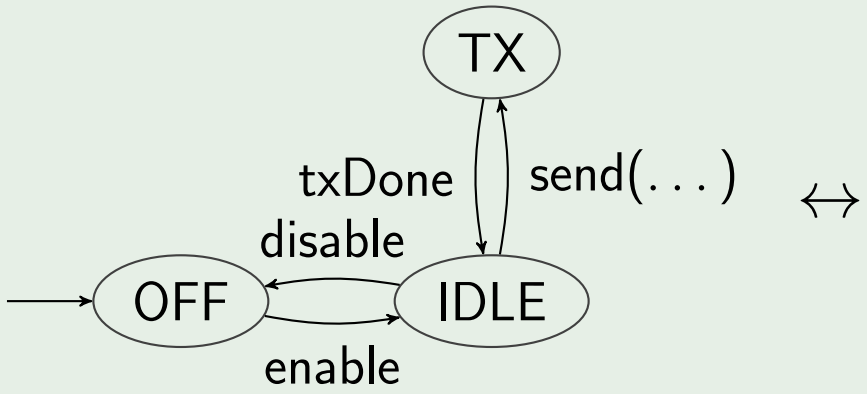
Figure: Simplified model for a radio driver

- System components frequently correspond to models
  - Task state transitions in scheduler, hardware states and energy consumption, ...
  - Model may be needed at run-time
- function call $\hat{=}$ model transition
- Challenge: synchronization model $\leftrightarrow$ code
  - Model driven development addresses model adjustment $\rightarrow$ implementation update
  - But: implementation may evolve $\rightarrow$ model needs update
  - model inference from source code rarely captures entire model [GSB09]

Source Code and Model Co-Development

- Attributes allow embedding models into source code

Example



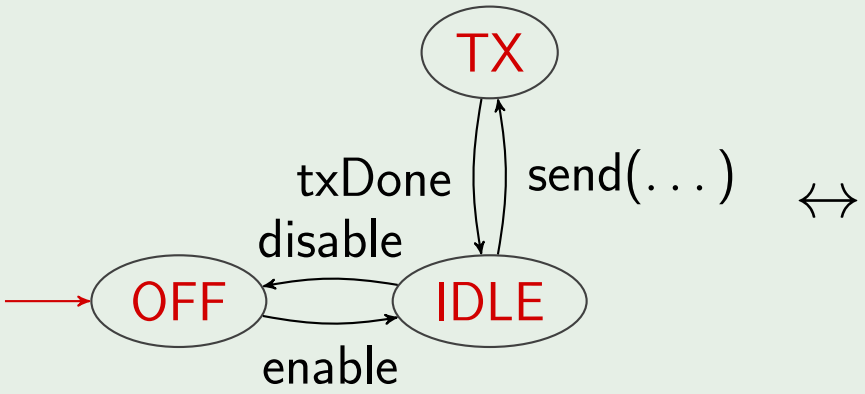
$\leftrightarrow$

```
void enable();

int send(char *data, uint8_t len);
```

Source Code and Model Co-Development

- Attributes allow embedding models into source code

Example



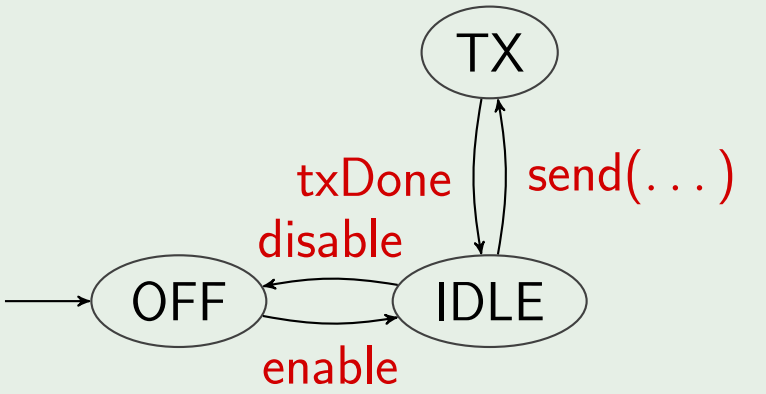
$\leftrightarrow$

```
enum states = {OFF, IDLE, TX};

void enable();

int send(char *data, uint8_t len);
```

Source Code and Model Co-Development

- Attributes allow embedding models into source code

Example



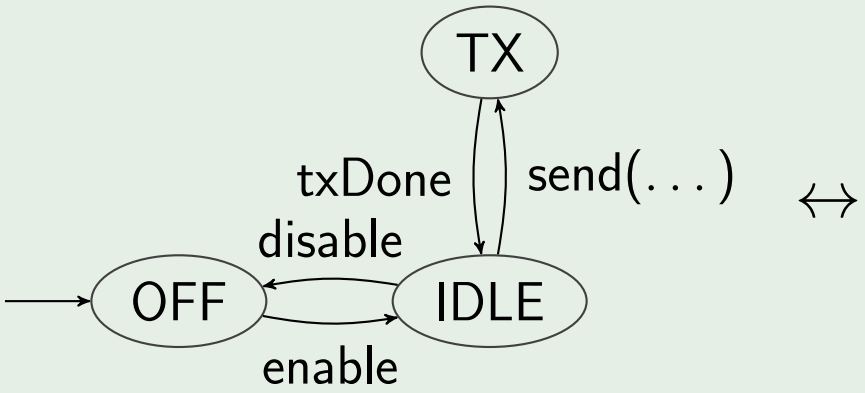
$\leftrightarrow$

```
enum states = {OFF, IDLE, TX};
[[Model::transition(OFF, IDLE)]]
  void enable();


[[Model::transition(IDLE, TX)]]
  int send(char *data, uint8_t len);
```

Source Code and Model Co-Development

- Attributes allow embedding models into source code

Example



$\leftrightarrow$

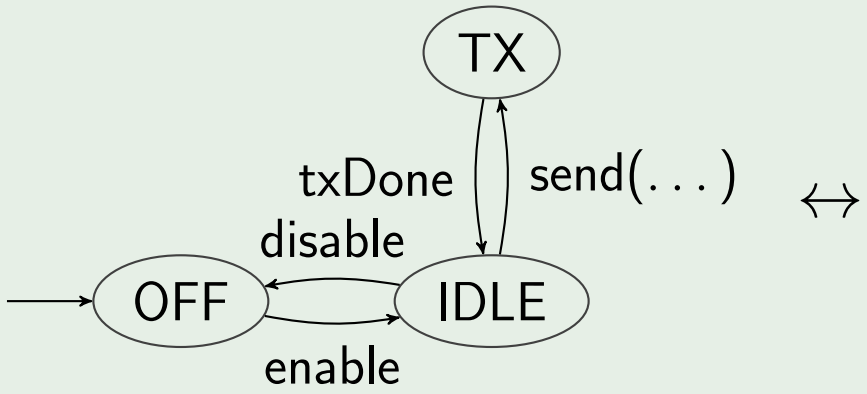
```
enum states = {OFF, IDLE, TX};
[[Model::transition(OFF, IDLE)]]
  void enable();


[[Model::transition(IDLE, TX)]]
  int send(char *data, uint8_t len);
```

- Works for arbitrary DFA-based models

Source Code and Model Co-Development

- Attributes allow embedding models into source code

Example



$\leftrightarrow$

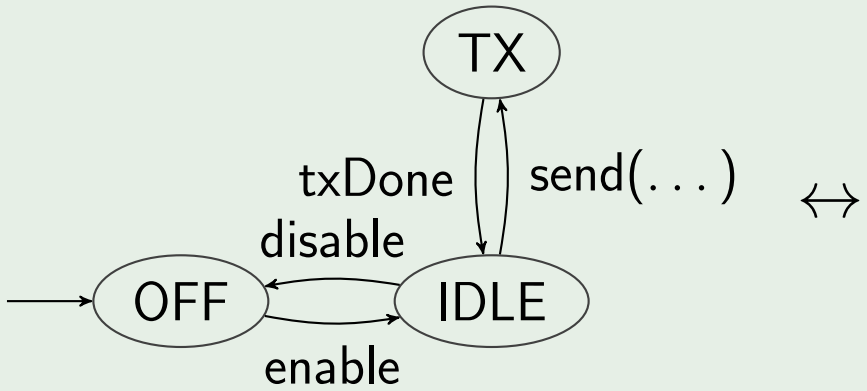
```
enum states = {OFF, IDLE, TX};
[[Model::transition(OFF, IDLE)]]
  void enable();


[[Model::transition(IDLE, TX)]]
  int send(char *data, uint8_t len);
```

- Works for arbitrary DFA-based models
- Model next to implementation $\rightarrow$ easy to update
  - Pre-existing code can be extended with models and re-used

Source Code and Model Co-Development

- Attributes allow embedding models into source code

Example



$\leftrightarrow$

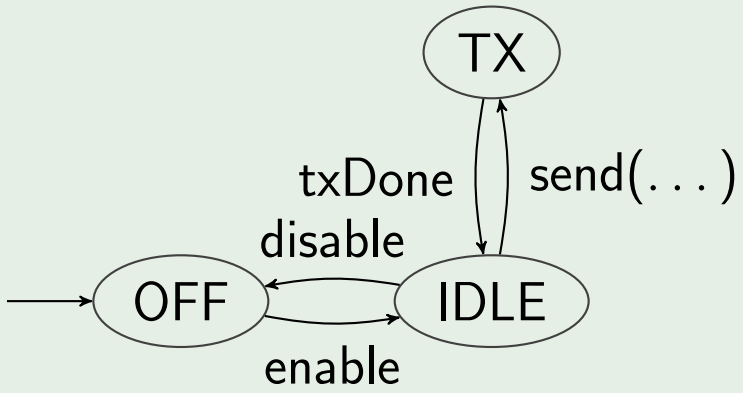
```
enum states = {OFF, IDLE, TX};
[[Model::transition(OFF, IDLE)]]
  void enable();


[[Model::transition(IDLE, TX)]]
  int send(char *data, uint8_t len);
```

- Works for arbitrary DFA-based models
- Model next to implementation $\rightarrow$ easy to update
  - Pre-existing code can be extended with models and re-used
- Can be made (partially) available at runtime by aspects

Source Code and Model Co-Development

- Attributes allow embedding models into source code

Example



$\leftrightarrow$

```
enum states = {OFF, IDLE, TX};
[[Model::transition(OFF, IDLE)]]
  void enable();
[[Model::testval_str(0, "Hello, World!")]]
[[Model::testval_int(1, 13)]]
[[Model::transition(IDLE, TX)]]
  int send(char *data, uint8_t len);
```

- Works for arbitrary DFA-based models
- Model next to implementation $\rightarrow$ easy to update
  - Pre-existing code can be extended with models and re-used
- Can be made (partially) available at runtime by aspects
- Zero overhead when unused
  - Annotations can also be parsed by external tools

Contents

Application

- Most (embedded) operating systems use C / C++

Application

- Most (embedded) operating systems use C / C++
- C++ source is compatible with AspectC++
  - Can be annotated and used with AspectC++ compiler
  - C++ backwards-compatibility possible with preprocessor macros

Application

- Most (embedded) operating systems use C / C++
- C++ source is compatible with AspectC++
  - Can be annotated and used with AspectC++ compiler
  - C++ backwards-compatibility possible with preprocessor macros
- C source needs to be adapted to C++
  - Tedious, but usually feasible
  - Depends on project size and amount of non-C++-compatible C code

Evaluation

- Examples used on three embedded OSes: CocoOS (C), RIOT (C) and Kratos (AspectC++)
  - Adaption to C++ took < 1 hour for CocoOS and one day for most of RIOT x86
  - Retained full C compatibility in CocoOS

Evaluation

- Examples used on three embedded OSes: CocoOS (C), RIOT (C) and Kratos (AspectC++)
  - Adaption to C++ took < 1 hour for CocoOS and one day for most of RIOT x86
  - Retained full C compatibility in CocoOS
- Implementation of:
  - Function call tracing, deprecation handling and interrupt control
  - IPC accounting via system API
  - Energy modelling of peripherals

Evaluation

- Examples used on three embedded OSes: CocoOS (C), RIOT (C) and Kratos (AspectC++)
  - Adaption to C++ took < 1 hour for CocoOS and one day for most of RIOT x86
  - Retained full C compatibility in CocoOS
- Implementation of:
  - Function call tracing, deprecation handling and interrupt control
  - IPC accounting via system API
  - Energy modelling of peripherals
- Compilation with `-Os`

Evaluation

- Examples used on three embedded OSes: CocoOS (C), RIOT (C) and Kratos (AspectC++)
  - Adaption to C++ took < 1 hour for CocoOS and one day for most of RIOT x86
  - Retained full C compatibility in CocoOS
- Implementation of:
  - Function call tracing, deprecation handling and interrupt control
  - IPC accounting via system API
  - Energy modelling of peripherals
- Compilation with `-Os`

$\rightarrow$ Annotations without aspects cause zero runtime overhead

Evaluation

- Examples used on three embedded OSes: CocoOS (C), RIOT (C) and Kratos (AspectC++)
  - Adaption to C++ took < 1 hour for CocoOS and one day for most of RIOT x86
  - Retained full C compatibility in CocoOS
- Implementation of:
  - Function call tracing, deprecation handling and interrupt control
  - IPC accounting via system API
  - Energy modelling of peripherals
- Compilation with `-Os`

$\rightarrow$ Annotations without aspects cause zero runtime overhead

$\rightarrow$ Negligible difference between manual and annotation-based implementations

  - less than ± 0.1 % runtime size variation (assumed to be optimization corner cases)
  - No runtime aspect resolution

Conclusion

- AspectC++ 2.2 supports C++-style annotations with custom behaviour

Conclusion

- AspectC++ 2.2 supports C++-style annotations with custom behaviour
- Custom annotations allow developers to express intentions for annotated code
  - Customizable tracing, synchronization or interrupt handling

Conclusion

- AspectC++ 2.2 supports C++-style annotations with custom behaviour
- Custom annotations allow developers to express intentions for annotated code
  - Customizable tracing, synchronization or interrupt handling
- ... can provide System APIs for plugins

Conclusion

- AspectC++ 2.2 supports C++-style annotations with custom behaviour
- Custom annotations allow developers to express intentions for annotated code
  - Customizable tracing, synchronization or interrupt handling
- ... can provide System APIs for plugins
- ... support source code and model co-development

Conclusion

- AspectC++ 2.2 supports C++-style annotations with custom behaviour
- Custom annotations allow developers to express intentions for annotated code
  - Customizable tracing, synchronization or interrupt handling
- ... can provide System APIs for plugins
- ... support source code and model co-development
- ... do not cause significant code size or runtime penalties

Conclusion

- AspectC++ 2.2 supports C++-style annotations with custom behaviour
- Custom annotations allow developers to express intentions for annotated code
  - Customizable tracing, synchronization or interrupt handling
- ... can provide System APIs for plugins
- ... support source code and model co-development
- ... do not cause significant code size or runtime penalties
- Similar possibilities with Java/AspectJ, Python and C#
  - Resolved at runtime $\rightarrow$ execution time overhead
  - C/C++ more common for OS development
  - AspectC++ usable with any C++-compatible backend compiler

References I

[GSB09] Michael Goedicke, Michael Striewe, and Moritz Balz. "Support for Evolution of Software Systems using Embedded Models". In: Design for Future–Langlebige Softwaresysteme. 2009.

[SL07] Olaf Spinczyk and Daniel Lohmann. "The design and implementation of AspectC++". In: Knowledge-Based Systems 20.7 (2007), pp. 636–651. ISSN: 0950-7051. DOI: http://dx.doi.org/10.1016/j.knosys.2007.05.004. URL: http://www.sciencedirect.com/science/article/pii/S0950705107000524.