FRP IoT Modules as a Scala DSL

Ben Calus, **Bob Reynders**, Dominique Devriese, Job Noorman, Frank Piessens

imec - DistriNet - KU Leuven

FRP IoT Modules

- Maintainable sensor network applications

FRP IoT Modules

- Maintainable sensor network applications
- High-level FRP-based API

FRP IoT Modules

- Maintainable sensor network applications

- High-level FRP-based API
- FRP-modules compile to Protected Module Architecture (Sancus)

FRP IoT Modules

- Maintainable sensor network applications

- High-level FRP-based API
- FRP-modules compile to Protected Module Architecture (Sancus)
- As a library in Scala using LMS

FRP: Summary

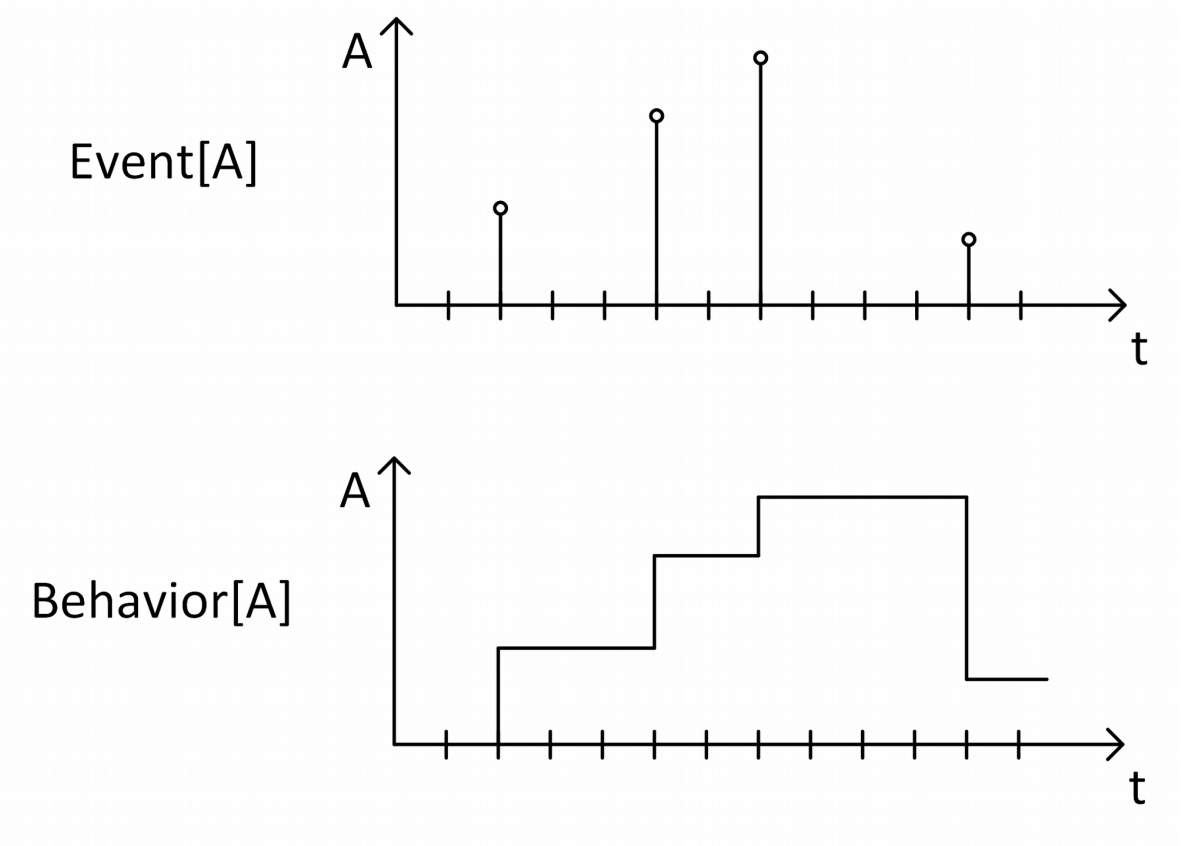
Event sequences of time-stamped values (button presses)

FRP: Summary

Event sequences of time-stamped values (button presses)

Behavior time-varying values (temperature readings)

FRP: Summary



FRP-Modules

- Unit of compilation/deployment

FRP-Modules

- Unit of compilation/deployment
- First-class values

```
def createModule[A](f: ModuleName ⇒ OutputEvent[A]): Module[A]

trait Module[A] {
  val name: ModuleName
  val output: OutputEvent[A]
}
```

FRP-Modules: Interface

- `val input: Event[T] = InputEvent[T]`

FRP-Modules: Interface

- `val input: Event[T] = InputEvent[T]`
- Timer

  `val timer: Event[Unit] = SystemTimerEvent()`

FRP-Modules: Interface

- `val input: Event[T] = InputEvent[T]`
- Timer

  `val timer: Event[Unit] = SystemTimerEvent()`
- Button

  `val button: Event[Unit] = ButtonEvent(Buttons.button1)`

FRP-Modules: Interface

- `val input: Event[T] = InputEvent[T]`
- Timer

  `val timer: Event[Unit] = SystemTimerEvent()`
- Button

  `val button: Event[Unit] = ButtonEvent(Buttons.button1)`
- Modules!

```
val m1 = createModule[Int] { ... }
val m2 = createModule[Unit] {
  val out: Event[Int] = ExternalEvent(m1.output)
}
```

Example

Example: Parking lot

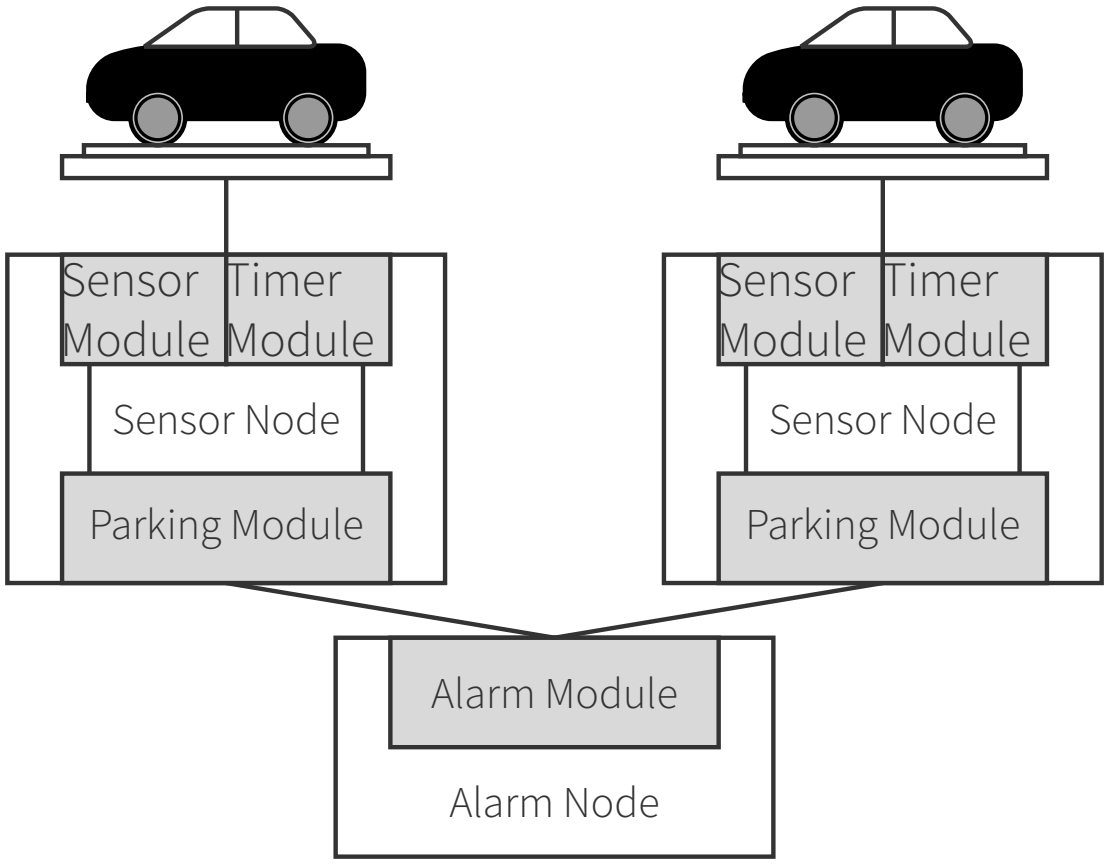


Figure 1: Parking Alarm

Example: Parking module

```
val parkingMod = createModule[Boolean] { implicit n: ModuleName ⇒
    val t = 1000
    val sensorE = ExternalEvent(sensorMod.output)
    val timer = ExternalEvent(timerMod.output)

    val isParked: Behavior[Boolean] = sensorE.startWith(false)
    val isParkedOnTick: Event[Boolean] = isParked.snapshot(timer)
    val ticksParked: Behavior[Int] =
      isParkedOnTick.foldp((taken, s) ⇒ if (taken) s + 1 else 0, 0)

    val violations = ticksParked.changes().map(_ >= t)
    out("violations", violations)
}
```

Example: Parking module as First-class values

- `val parkingMod = createModule[Boolean] { ... }`

Example: Parking module as First-class values

- `val parkingMod = createModule[Boolean] { ... }`
- Code generation

Example: Parking module as First-class values

- `val parkingMod = createModule[Boolean] { ... }`
- Code generation
- `def parkingMod(timeout: Long) = createModule[Boolean] { ... }`

Example: Parking lot

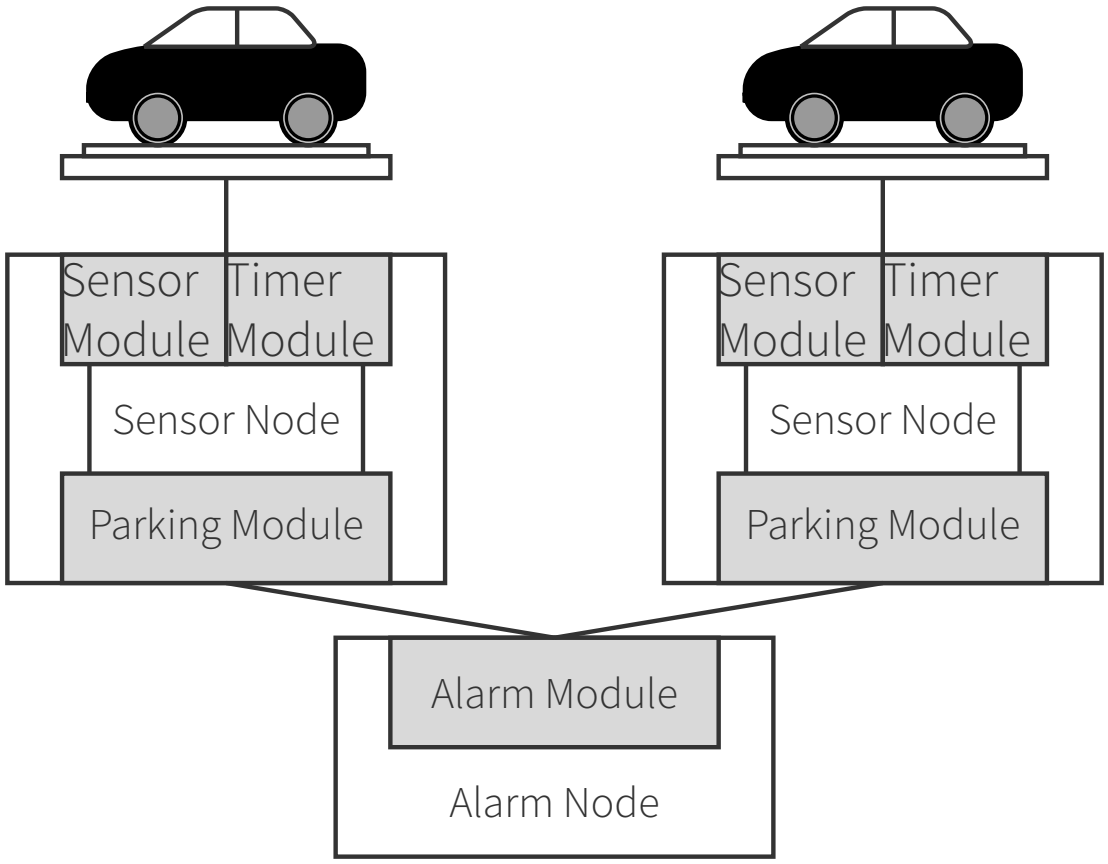


Figure 2: Parking Alarm

Example: Safely reusing the network

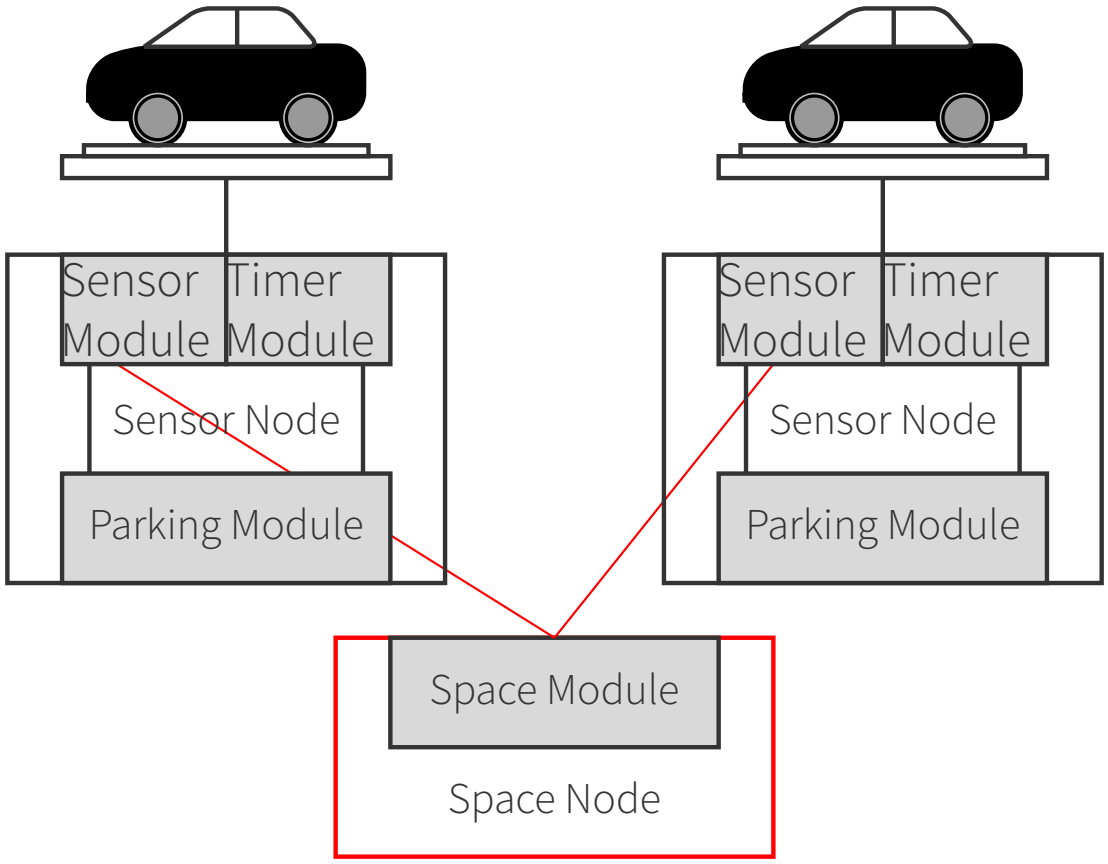


Figure 3: Parking Counter

Example: Safely reusing the network with Sancus

- Spoofed events

Example: Safely reusing the network with Sancus

- Spoofed events
- Tampered execution

Example: Safely reusing the network with Sancus

- Spoofed events
- Tampered execution
- “If the program were to produce an output event, it produces the same event regardless of an attacker.”

Example: Safely reusing the network with Sancus

- Spoofed events
- Tampered execution
- “If the program were to produce an output event, it produces the same event regardless of an attacker.”
- *Authentic Execution of Distributed Event-Driven Applications with a Small TCB* (Noorman, Mühlberg, and Piessens 2017)

Compilation

Compilation Pipeline: Scala (EDSL)

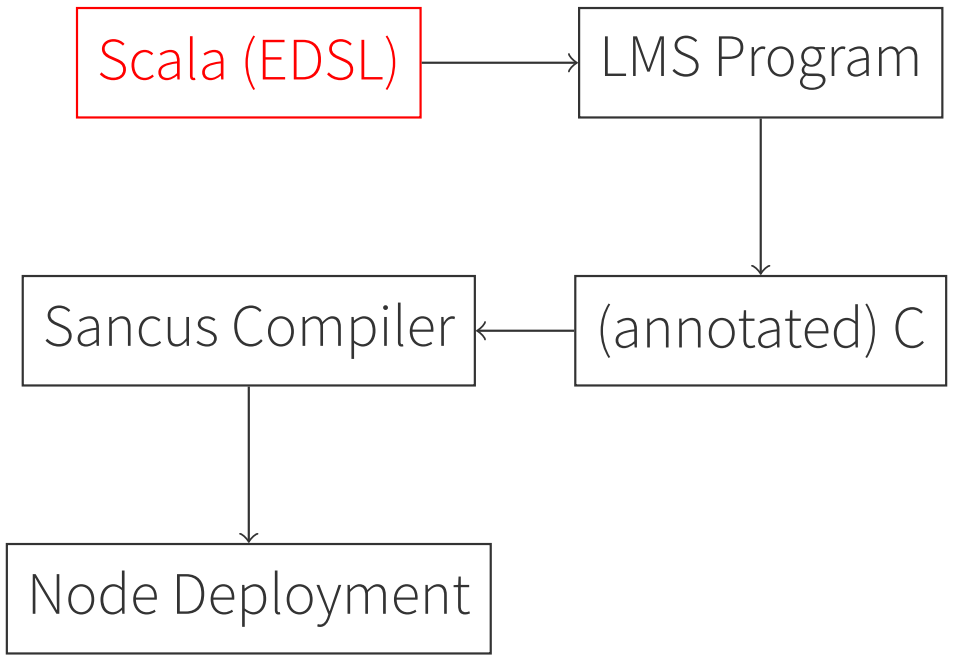
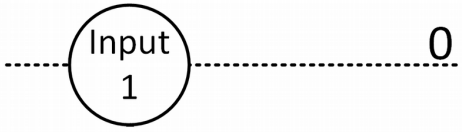


Figure 4: Compilation Pipeline

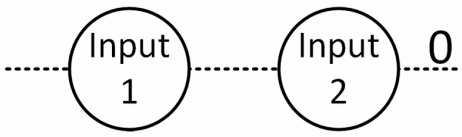
Building the graph

```
1 val input1 = InputEvent[Int]
2 val input2 = InputEvent[Int]
3 val negate2 = input2.map( (i) => 0-i )
4 val merged =
    input1.merge(negate2, (x,y) => x + y)
5 val filtered =
    merged.filter( (x) => abs(x) < 10)
6 val counter =
    filtered.foldp((x,state)=>state + x, 0)
```



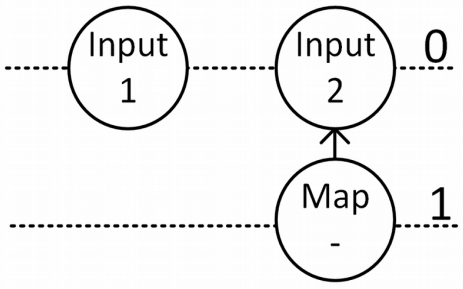
Building the graph

```
1 val input1 = InputEvent[Int]
2 val input2 = InputEvent[Int]
3 val negate2 = input2.map( (i) => 0-i )
4 val merged =
    input1.merge(negate2, (x,y) => x + y)
5 val filtered =
    merged.filter( (x) => abs(x) < 10)
6 val counter =
    filtered.foldp((x,state)=>state + x, 0)
```



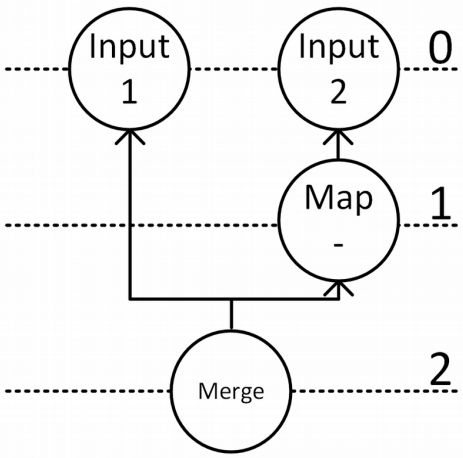
Building the graph

```
1 val input1 = InputEvent[Int]
2 val input2 = InputEvent[Int]
3 val negate2 = input2.map( (i) => 0-i )
4 val merged =
    input1.merge(negate2, (x,y) => x + y)
5 val filtered =
    merged.filter( (x) => abs(x) < 10)
6 val counter =
    filtered.foldp((x,state)=>state + x, 0)
```



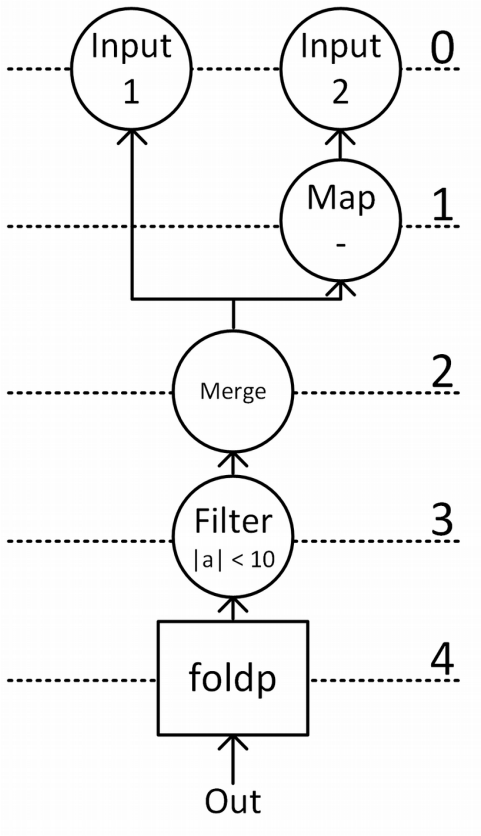
Building the graph

```
1 val input1 = InputEvent[Int]
2 val input2 = InputEvent[Int]
3 val negate2 = input2.map( (i) => 0-i )
4 val merged =
    input1.merge(negate2, (x,y) => x + y)
5 val filtered =
    merged.filter( (x) => abs(x) < 10)
6 val counter =
    filtered.foldp((x,state)=>state + x, 0)
```

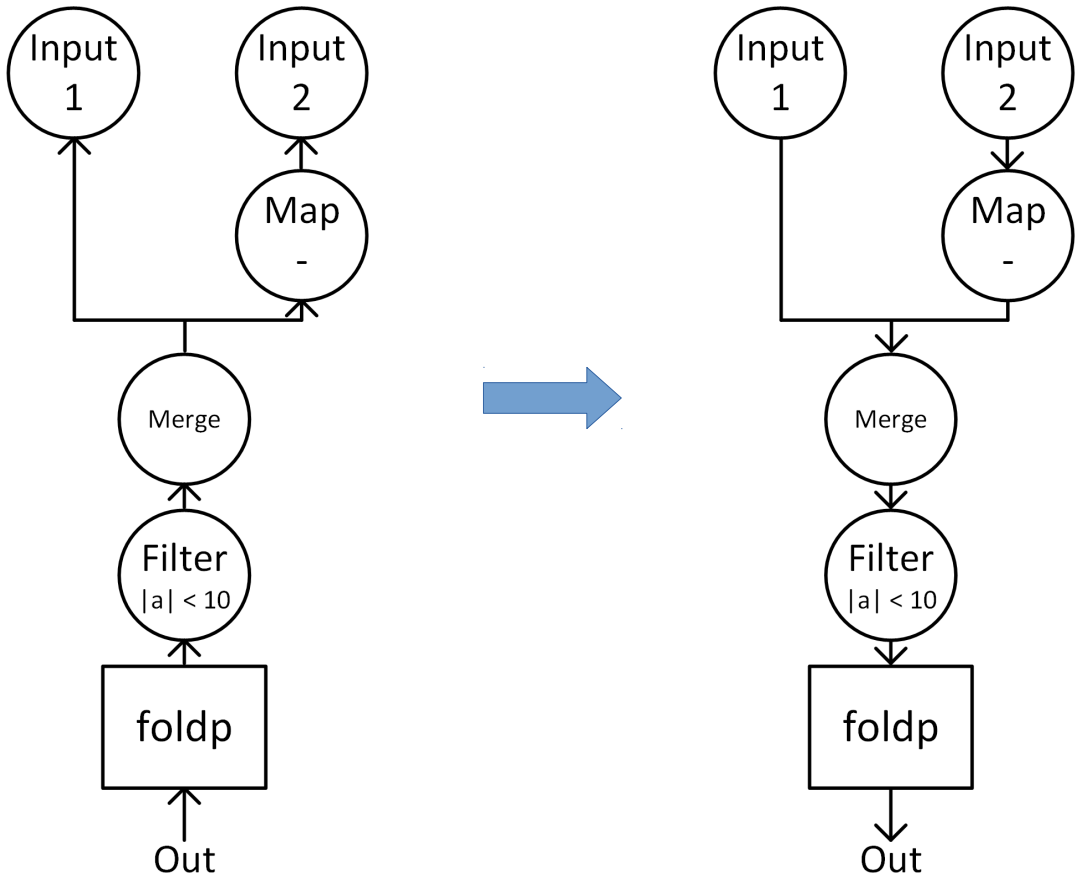


Building the graph

```
1 val input1 = InputEvent[Int]
2 val input2 = InputEvent[Int]
3 val negate2 = input2.map( (i) => 0-i )
4 val merged =
    input1.merge(negate2, (x,y) => x + y)
5 val filtered =
    merged.filter( (x) => abs(x) < 10)
6 val counter =
    filtered.foldp((x,state)=>state + x, 0)
```



Building the graph



Compilation Pipeline: LMS Program

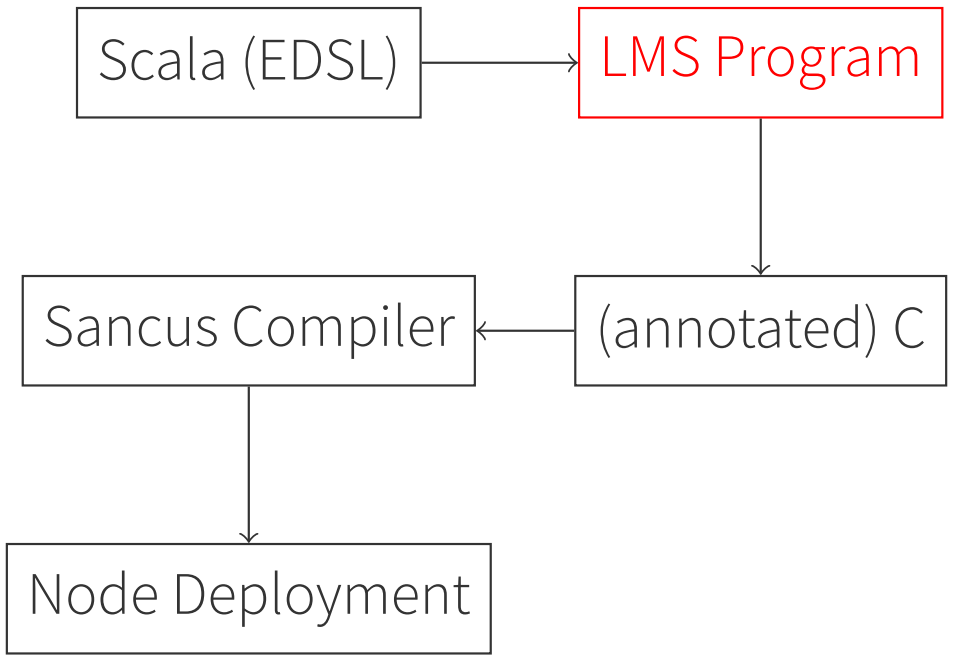
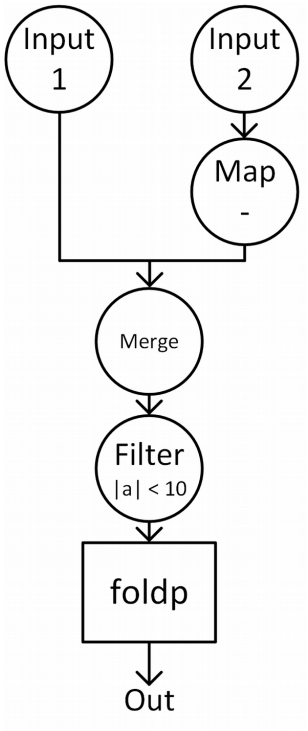
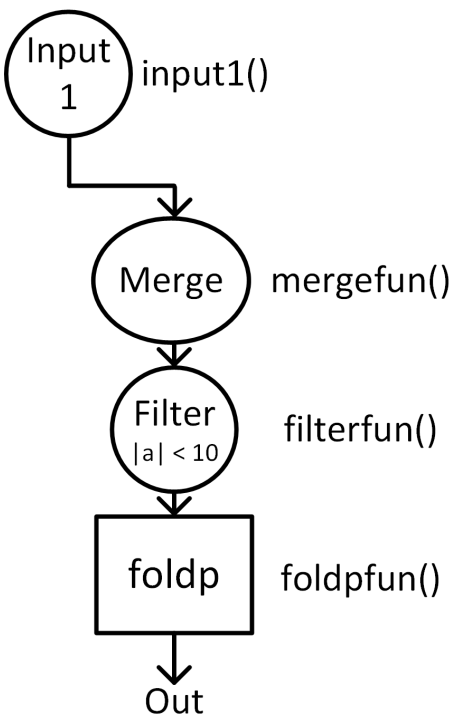


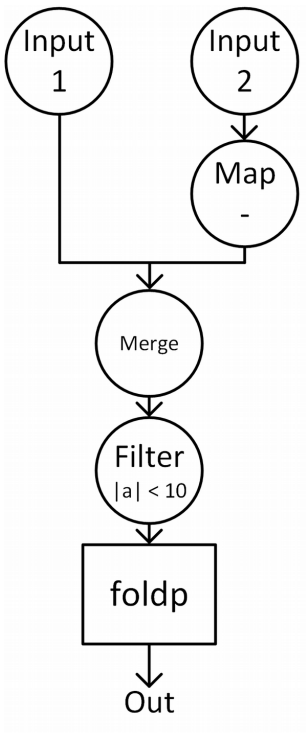
Figure 5: Compilation Pipeline

Compiling to (Sancus) C

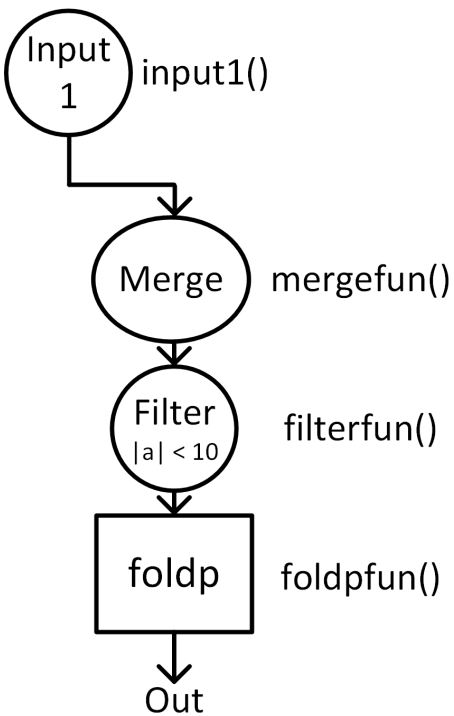


Compiling to (Sancus) C





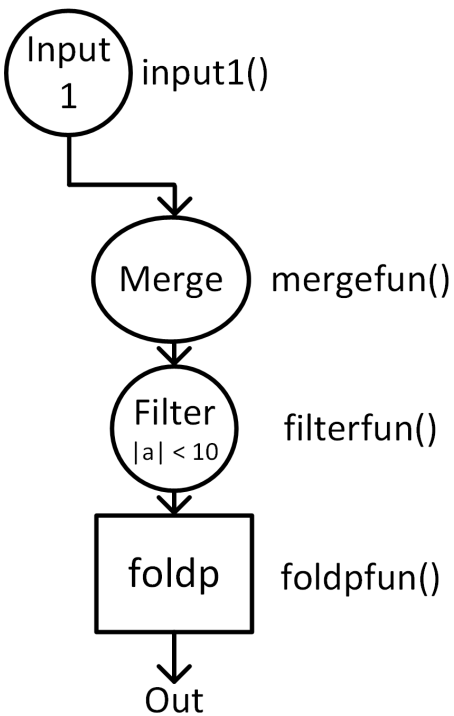
Compiling to (Sancus) C

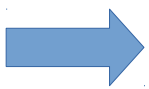



```
void toplevel1( ) {
    input1( );
    mergefun( );
    filterfun( );
    foldpfun( );
}

void foldpfun( ) {
    …
}
```

Compiling to (Sancus) C





```
int foldp_w;

void toplevel1( ) {
    bool input1_p;
    int input1_w;
    input1(&input1_p, &input1_w);
    mergefun( ... );
    filterfun( … );
    bool foldp_p;
    foldpfun(&foldp_p, … );
}
```

Compiling to (Sancus) C

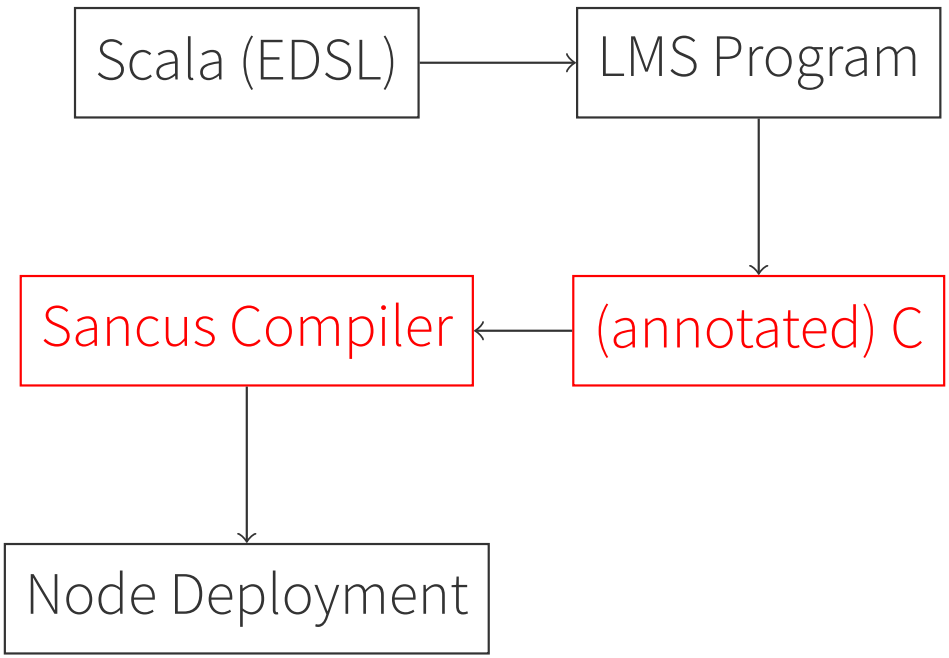
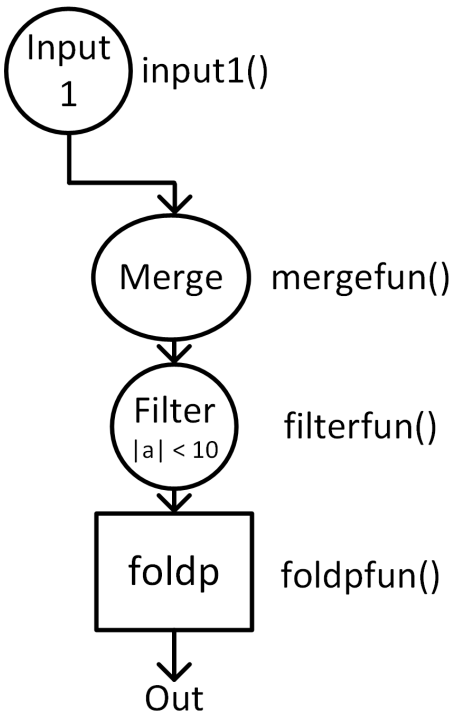


Figure 6: Compilation Pipeline

Compiling to (Sancus) C




```
SM_DATA(mod1) int foldp_w;

SM_DATA(mod1) void toplevel1(
) {
    bool input1_p;
    int input1_w;
    input1(&input1_p, &input1_w);
    mergefun( ... );
    filterfun( … );
    bool foldp_p;
    foldpfun(&foldp_p, … );
}
```

Compiling to (Sancus) C

```
val m1 = createModule[Int] { implicit m ⇒
  val data: Event[Int] = ...
  out(data)
}

val m2 = createModule[Unit] { implicit m ⇒
  ExternalEvent(m1.output)
}
```

Figure 7: Module Communication (EDSL)

Compiling to (Sancus) C

```
SM_DATA(m1) ...
SM_OUTPUT(m1, out);

SM_INPUT(m1) {
  ...
}

SM_FUNC(m1) {
  ...
  out(&data, sizeof(data))
}
```

```
SM_DATA(m2) ...
SM_INPUT(m2) {
 ...
}
```

Figure 8: Module Communication (Sancus)

Deployment

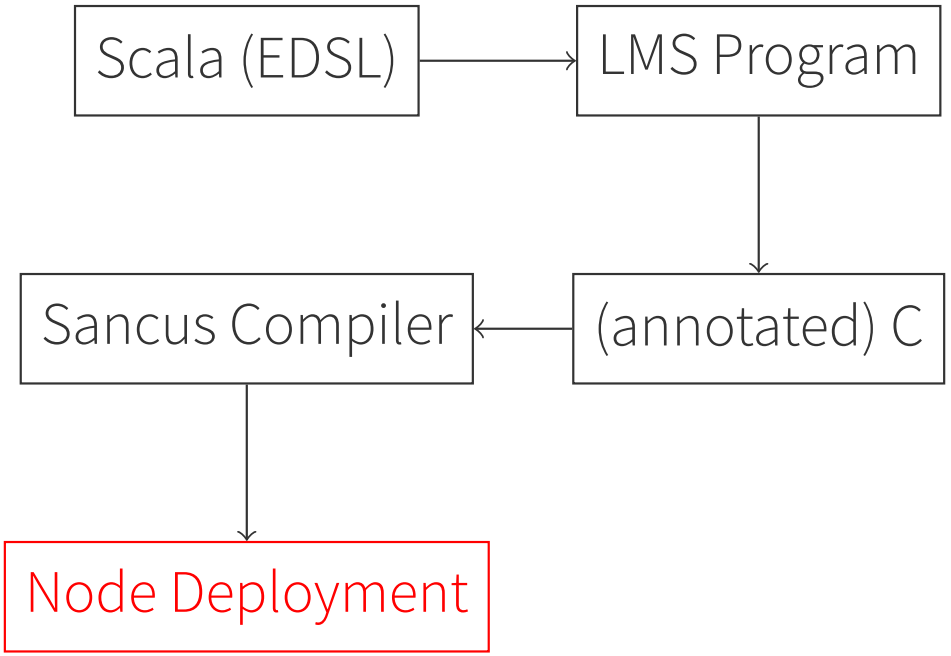


Figure 9: Compilation Pipeline

Deployment

- Sancus configuration file

Deployment

- Sancus configuration file
- Future Work

Deployment

- Sancus configuration file
- Future Work
    - Configuration DSL

Deployment

- Sancus configuration file
- Future Work
  - Configuration DSL
  - Well-typed deployments

Deployment: Future Work

```
val parkDevice = Pressure + Timer
val alarmDevice = Alarm

val parkingMod = ...
val alarmMod = ...

val deploymentScheme =
  parkDevice.deploy(parkingMod, Sancus)
            .and(alarmDevice.deploy(alarmMod, SGX))
```

Summary

Summary

- (step towards) maintainable sensor network applications

Summary

- (step towards) maintainable sensor network applications
- High-level FRP library with (free) low-level guarantees

Summary

- (step towards) maintainable sensor network applications
- High-level FRP library with (free) low-level guarantees
- Scala EDSL with prototype at:
  `github.com/tzbob/scala-iot-modules-for-frp`