FRP IoT Modules as a Scala DSL

Ben Calus, Bob Reynders, Dominique Devriese, Job Noorman, Frank Piessens
imec-DistriNet, KU Leuven, 3001 Leuven, Belgium
{bob.reynders}@cs.kuleuven.be

Abstract

With Internet of Things applications growing in size and popularity, physical sensor networks are more often running multiple complex applications. It becomes increasingly important to maintain these event-driven programs on embedded systems. Traditionally, event-driven applications such as sensor network applications are written using an imperative style of programming where different callback routines are registered to handle events. As the application complexity grows, the inverted control flow and reliance on shared global state makes this style of programming hard to maintain. Furthermore, sensor network applications are inherently distributed and are written by manually managing code-bases of sub-applications that go on all nodes separately. If security is important, the programmer needs to manually interface with low-level security primitives because there is no built-in notion of components.

We propose a more maintainable approach where the developer essentially writes a first-order FRP program, containing code fragments in an embedded subset of C. From this FRP program, we generate efficient C code to be run on every node. Every module of the FRP program is compiled to a separate C module, making it easy to deploy modules to different nodes, and to enhance the security of the application by isolating modules from other software running on the nodes. Our implementation is based on a Scala EDSL that we use to let the user conveniently embed fragments of C code. The annotated C code gets compiled to Sancus, a security architecture for IoT nodes that supports the secure and distributed execution of the generated modules.

CCS Concepts • **Software and its engineering → Domain specific languages**;

Keywords Functional Reactive Programming, DSL, Scala

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
REBLS'17, October 23, 2017, Vancouver, Canada
© 2017 Association for Computing Machinery.
ACM ISBN 978-1-4503-5515-5/17/10...$15.00
https://doi.org/10.1145/3141858.3141861

ACM Reference Format:
Ben Calus, Bob Reynders, Dominique Devriese, Job Noorman, Frank Piessens. 2017. FRP IoT Modules as a Scala DSL. In *Proceedings of 4th ACM SIGPLAN International Workshop on Reactive and Event-Based Languages and Systems (REBLS'17)*. ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/3141858.3141861

1 Introduction

With Internet of Things applications growing in size and popularity, efficiently creating and maintaining event-driven programs on embedded systems becomes more important. Generally, a sensor network as a whole runs one or more applications and it contains a network of collaborating nodes. Each node runs a sub-application that reacts to events from its environment or one of its peer nodes to perform specific actions.

Traditionally, event-driven applications such as sensor network applications are written using an imperative style of programming where different callback routines are registered to handle events. These callbacks react to events by modifying the state of the application. As applications grow, this state becomes more complex and harder to mutate consistently. In short, event-driven programs written in an imperative style become hard to maintain due to an inverted control flow and a reliance on shared mutable state.

Furthermore, scalability problems are not limited to each node's sub-application. When a sensor network application grows, so do the amount of connections between nodes. Unchecked links between nodes become prone to type errors and are prone to subtle errors since the topology is hidden throughout several codebases that are each filled with code to send and receive events.

Functional reactive programming (FRP) [1] (although initially proposed for modeling animations) is an alternative programming model that makes it easier to reason about event-driven code. Instead of using side-effecting callbacks, the program is constructed by composing *behaviors* and *events*: components representing time-dependent values.

Macro-programming [2, 3, 5], not to be confused with Scala's macros, is a programming model that helps a programmer with the inherently distributed model of sensor network applications by enabling *sensor network applications* in *one* codebase instead of having several codebases to represent sub-applications on nodes.

However, both approaches have not been combined together and have problems on their own. Macro-programming languages often lack the ability to reason about event streams as first class values and do not provide a type-safe connection

between nodes. FRP, on the other hand, is often too expensive to implement efficiently on embedded devices due to a runtime cost of the FRP runtime, particularly maintaining a dependency graph.

In this paper we report on a work-in-progress solution for the discussed sensor network problems as a unification between FRP and macro-programming languages in the form of a Scala embedded domain specific language (EDSL). This combination of FRP and macro-programming combines the advantages of both and additionally, we discover that an FRP-first solution to IoT modules is a natural fit for existing protected module architectures which gives a programmer additional security benefits without having to deal with annotations. In summary, we make the following contributions:

- We show a maintainable approach to sensor network applications based on macro-programming techniques and FRP.
- We demonstrate that our first-order FRP API can be used on embedded devices by compiling the program to C modules. We show that a direct one-to-one mapping from FRP modules to C modules can be used to take advantages of existing protected module architectures. The high-level DSL makes is possible to reap the low level security benefits without manually having to annotate the codebase.
- We provide a proof-of-concept implementation in which Scala can be used as a meta-programming language to generate modules for Sancus [6], a protected module architecture.

In section 2, we demonstrate the practical use of our language with a step-by-step solution for a parking sensor application. In section 3, we provide an overview of the compilation pipeline. We discuss our API and how it should be used in section 4. In section 5, we explain how FRP can be implemented to embedded protected modules with a minimal performance cost despite its convenient semantics. We conclude with some future and related work in sections 6 and 7.

2 Parking Lot with FRP

To introduce our approach we focus on the implementation of a parking lot with a simple topology as shown in Figure 1. There are multiple parking spots that are each fitted with a physical node, a node has a processor, memory and I/O devices. Each node has a timer and a sensor to detect whether or not a car has parked and for how long. After a car has been parked for more than a maximum amount of time an alarm module should be notified. In the full scenario all nodes are open to run multiple applications, e.g., simultaneously run a sensor network application to compute how many parking spots are free as well as initiate alarms for cars that are parked for too long. In the following example we only focus on the latter by implementing a sensor and parking module.

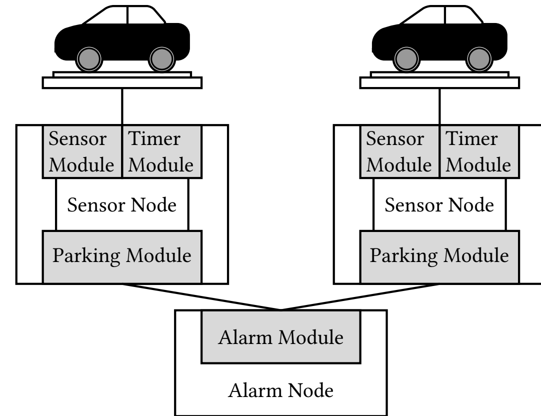

Figure 1. Parking Sensor

Sensor Module The first re-usable module that we implement is the sensor driver module:

```
val sensorMod: Module[Boolean] =
  createModule { implicit n: ModuleName ⇒
    val sensor: Event[Unit] = ButtonEvent(Buttons.button1
     )
    val taken: Behavior[Boolean] =
      sensor.foldp((_, state) ⇒ !state, false)
    out("parkingSwitch", taken.changes)
  }
```

The physical sensors of a parking application are represented as buttons, conceptually they are clicked whenever a car enters or leaves. In FRP, applications are written by composing events and behaviors. Events represent values that exist at discrete points. For example the event: `ButtonEvent(Buttons .button1)` contains all actions on the parking switch. Behaviors represent time varying values such as `taken`, a value that represents the current state of a parking spot. Notice the implicit `ModuleName`, all event operations require an implicit parameter to ensure that their definitions can be linked to a module. The output of a sensor module is defined by returning an `OutputEvent[T]` when creating the module, this in turn defines the resulting module's type: `Module[T]`. A module is typed by its output and output events are those that are explicitly created using `out`. In our example we define the `parkingSwitch` output as a boolean event stream that reflects whether or not a car has taken or left a spot (`taken.changes`).

Parking Module Next we implement the parking module, `parkingMod`:

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

    val sensorB: Behavior[Boolean] =
      sensorE.foldp((_, s) ⇒ s, false)
    val snapSensor = sensorB.snapshot(timer)
```

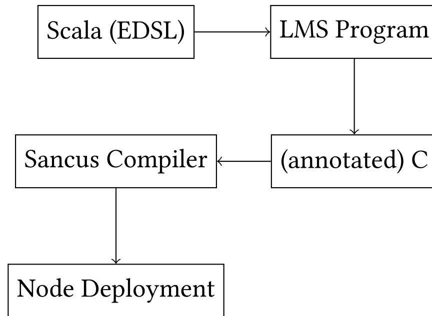

Figure 2. Compilation Pipeline

```
  val timeTaken: Behavior[Int] =
   snapSensor.foldp((taken, s) ⇒ if (taken) s + 1
                                 else 0, 0)
  val violations =
    timeTaken.changes().map(_ >= timeout)
  out("violations", violations)
}
```

Since modules are just Scala values in our EDSL it becomes trivial to abstract over them. We define parkingMod as a function that is parameterised over a timeout. A timeout signals how many ticks from a timer a car is allowed to stand on the parking spot. We use ExternalEvent to access the output of previously defined modules, in our example we do this for both the timer module (of which we assume the existence) and for the sensor module that we defined before. Next, we define a sensor behavior sensorB that starts with false and changes according to our sensor module output. We define sensorB so that we can sample it at the rate of the timer using snapshot giving us snapSensor. With snapSensor we can track how many ticks a sensor has been active. We do this by folding over the event stream and incrementing the counter by one if an event is true (a car is parked) and by resetting if an event is false (a spot is open). The output of the module is defined by mapping the changes of the timeTaken behavior to a timeout check and returning the violation events.

3 Compilation Overview

In fig. 2, a graphical representation of the compilation pipeline is shown. In our approach programmers write sensor network applications in our first-order FRP DSL. The DSL is embedded in Scala by using Lightweight Modular Staging (LMS) [7]. LMS is a modular approach to writing staged applications in Scala. An LMS program contains regular expressions (e.g. String) as well as staged expressions (e.g. Rep[String]). When the LMS program is run, all the normal expressions are executed and the next stage of the program is produced.

LMS allows users to define different compilation targets for staging. In our case, the LMS program maps FRP modules to protected modules in C. The protected module architecture that we target is Sancus [6]. Its compiler expects C programs that are properly annotated to mark, for example, which data is private to a module, how to enter the module, etc. These annotations can be added automatically by the LMS program because the appropriate information is part of the high-level EDSL.

The annotated C program in turn is compiled by the Sancus compiler and produces a hardened deployable module with strong security properties.

4 FRP for IoT

The EDSL provides two FRP abstractions: Events and Behaviors. Note that the Rep[T] types indicate a staged operation, think of them as operations that are compiled to C. We write down the meaning of events and behaviors using a simple mathematical notation.

4.1 Events

Events can be seen as a sequence of discrete timestamped values:

$$Event_\tau = \left\{ \begin{array}{l} e \in \mathcal{P}(Time \times \tau) \mid \\ \wedge \forall (t, v), (t', v') \in e.\, t = t' \Rightarrow v = v' \end{array} \right\}$$

Common examples of events are mouse clicks and button presses. As shown in the semantics, Events cannot fire multiple values at the same time.

Below, we discuss the three core operations for events: map, filter and merge. Note that we use the following notation from now on to show our API:

```
ClassName[A]#methodName[B](p: ParamType): ResultType
```

Keep in mind that for brevity we do not show the implicit argument ModuleName, all the following methods require it:

map applies a function on each value that the source event produced. For example, nbs.map(_ ⇒ 0) fires zeros whenever the nbs fires, since the mapped function throws away the original values.

```
Event[A]#map[B](f: Rep[A] ⇒ Rep[B]): Event[B]
```

filter can ignore values based on a predicate. nbs.filter (x ⇒ x % 2 == 0), for example, will only produce the even values of nbs.

```
Event[A]#filter(p: Rep[A] ⇒ Rep[Boolean]): Event[A]
```

merge takes two events (of the same type) and returns an event that fires whenever one of the original events fire. If both events fire at the same time, the given function combines both values into a single new one. If only one event fires the given function is ignored. For example, if we have events ones (producing 1s) and twos (producing 2s), then ones.merge (twos){ (x, y) ⇒ x + y } produces 1s when just ones fires, 2s when just twos fires and 3s when both fire.

```
Event[A]#merge(b: Event[A])
  (f: (Rep[A], Rep[A]) ⇒ Rep[A]): Event[A]
```

4.2 Behavior

Behaviors can be seen as a value that may change over time. Semantically you can think about behaviors as regular functions:

$$Behavior_\tau = \{b \in Time \longrightarrow \tau\}$$

We go over all core operations in detail:

constant is a method on the `Behavior` singleton object. It creates a constant behavior with a given value.

```
object Behavior#constant[A](a: A): Behavior[A]
```

map2 takes a second behavior and a function for combining the two behaviors' values to the value for the resulting behavior.

```
Behavior[A]#map2[B](fb: Behavior[B],
  f: (Rep[A], Rep[B]) ⇒ Rep[C])): Behavior[C]
```

changes returns all points at which a behavior changes as an event. An important detail is that in this document (contrary to traditional FRP [1]) `Time` is $\mathbb{N}$ and not $\mathbb{R}$. What this conveys is that behaviors can only change discretely so that changes can be observed.

```
Behavior[A]#changes: Event[A]
```

foldp is similar to folding a list, a starting value and an accumulation function is given to calculate a behavior's new 'current' value on each change. `foldp` is the only primitive that provides stateful computations in our language.

```
Event[A]#foldp[B](f: (Rep[A], Rep[B]) ⇒ Rep[B],
                  init: Rep[B]): Behavior[B]
```

4.3 Modules

Modules are the main unit of abstraction in our language. A module can have asynchronous input and output which we model with events. Modules can be deployed with local interactions on one node or distributed on multiple nodes, the configuration of this deployment is considered future work of this EDSL.

A module has a name and output and is created through `createModule`:

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

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

Preferably, all re-usable standalone components should be encapsulated in a module, much like classes. What makes our modules unique is that they can only be connected asynchronously through `ExternalEvent`:

```
case class ExternalEvent[A](oe: OutputEvent[A])
  (implicit n: ModuleName) extends Event[A]
```

Modules are then tied together by combining these two APIs:

```
val sensorMod =
  createModule[Boolean] { implicit n: ModuleName ⇒
    val sensor = ButtonEvent(Buttons.button1)
    val taken: Behavior[Boolean] =
      sensor.foldp((_,state) ⇒ !state, false)
    out("button1", taken.changes)
  }

val ex = createModule[_] { implicit n: ModuleName ⇒
  val sensorE: Event[Boolean] =
    ExternalEvent(sensorMod.output)
  ...
}
```

Asynchronous connections between modules in sensor network applications allow for flexible deployment schemes. Since communication is asynchronous by default, modules can be deployed locally or distributed. While asynchronous programming usually makes code harder to read and maintain due to callbacks, we integrate the communication primitives into FRP to make this native to the language.

5 Compiling FRP & Mapping to Secure Modules

There are several possibilities to compile our proposed API. For example, the application could be compiled and deployed as a whole as usual without maintaining modules. However, a malicious operating system or even other malicious programs on a node could compromise the integrity of the application. Sensor networks are becoming more and more complex and re-use of a similar infrastructure for more than one application would be beneficial.

Protected module architectures are a way to make this re-use safe. These architectures guarantee modules to run isolated from other applications and maintain their security guarantees even if attackers are allowed to deploy their own applications on the infrastructure or even tamper with the OS. To compile a module for such a system, extra information has to be passed to the compiler. These annotations often get tedious and, in a sense, development effort and convenience is traded for stronger security properties. In our approach we wish to provide programmers with a more automated architecture.

The FRP module system that we propose maps well on the interface of such a protected module architecture and for this paper we focus on Sancus [6]. We implement our approach using an EDSL with the LMS framework [7]. We have a working prototype implementation that works on a physical Sancus development board. It contains an example of a local car park system for debugging purposes[1].

LMS The LMS framework helps us to convert the high-level FRP code into low-level constructs in the target language: C. LMS uses types to make a distinction between code to be evaluated now (values of type `T`), in contrast to code to be evaluated later (values of type `Rep[T]`). What this

[1] https://github.com/Tzbob/scala-iot-modules-for-frp/releases/tag/rebls17

```
createModule[Boolean] { implicit n: ModuleName ⇒
  val sensor: Event[_] = ButtonEvent(Buttons.button1)
  val taken: Behavior[Boolean] =
    sensor.foldp((_,state) ⇒ !state, false)
  out("button1", taken.changes)
}
```

Figure 3. A Simple Module

means for us is that we can use the full power of Scala as a meta-language (in values of type `T`) when writing sensor network applications (values of type `Rep[T]`). Modules that differ slightly, for example, modules that require a node-specific key, can be generated through the meta-program. We extended the standard LMS C generations functionality to output specific annotations for the Sancus project.

Secure Modules with Sancus Basic Sancus requires code to be annotated accordingly, programmers are required to: mark a module's specific entry point (`SM_ENTRY`) and mark functions and data internal to a module (`SM_FUNC` and `SM_DATA`). Extensions on Sancus add security primitives to support event-driven distributed applications and to secure control of input and output devices used by these applications (`SM_INPUT` and `SM_OUTPUT`).

Sancus guarantees that specific software modules run isolated from other applications and maintains its security guarantees even if attackers are allowed to deploy their own applications on the infrastructure or even tamper with the OS. However, this all happens through restrictions on the programming model, annotations have to be placed properly and an event-driven manner is forced on the programmer to communicate between distributed modules. The programming model that is proposed in section 4 is fully compatible with Sancus. Our LMS-based implementation generates all annotations where needed and the model is naturally event-driven.

Compiling FRP & Modules We make use of the staged expressions to efficiently compile FRP primitives and modules. For example, Figure 3 shows a simple module that contains a couple of high-level abstractions, a first-class button event is being folded over to build a behavior. Afterwards this behavior is turned into a (named) output event through `changes`. These three lines of code define a module that exposes a boolean event.

In traditional FRP implementations, where performance and overhead is less important, these primitives are implemented similar to the observer pattern or through a propagation runtime. In our case these primitives are compiled more efficiently to function calls with mutable state (i.e., FRP abstractions are compiled away almost entirely). Figure 3 compiles to Sancus and its appropriate annotations (C-macros) in fig. 4. The full functionality of a compiled module in our system is defined by 'top' functions (in this

```
1  SM_DATA(mod1) bool x32;
2  SM_DATA(mod1) int x65;
3  SM_ENTRY(mod1) void init_mod1() {...}
4  SM_FUNC(mod1) void input_button1(uint8_t* x1,int x2,int*
       x3,bool* x4) {...}
5  SM_FUNC(mod1) void fold_function(int x33,bool x34,bool*
       x35) {
6    bool x37 = x34;
7    if (x37) {
8      bool* x38 = x35;
9      *x38 = true;
10     bool x40 = x32;
11     bool x42 = !x40;
12     x32 = x42;
13   } else {
14     bool* x38 = x35;
15     *x38 = false;
16   }
17 }
18 SM_FUNC(mod1) void changes(bool x50,bool* x51,bool* x52)
       {...}
19 SM_OUTPUT(mod1,sm_output);
20 SM_FUNC(mod1) void output(bool x102,bool x103) {...}
21 SM_INPUT(mod1,top_function,x75,x76) { //top1
22   init_mod1();
23   bool x81 = false;
24   int x82;
25   int* x84 = &x82;
26   bool* x86 = &x81;
27   uint8_t* x77 = x75;
28   input_button1(x77,x76,x84,x86);
29   bool x89 = false;
30   bool* x91 = &x89;
31   fold_function(x83,x85,x91);
32   bool x94 = false;
33   bool x95;
34   bool* x97 = &x95;
35   bool x98 = x94;
36   bool* x99 = &x98;
37   changes(x89,x97,x99);
38   output(x96,x98);
39 }
40 DECLARE_SM(mod1, 0x1234);
```

Figure 4. Compiled Foldp

example, `top_function`). All inputs to the module have their own top function. This function contains the unrolled and compiled FRP program related to a specific input event. For example, a new event from `button1` triggers the evaluation of `top_function` which in turn evaluates the `foldp` behavior and sends its changes as output. This static representation of the FRP program stays relevant throughout the execution of the application because of the flavor of FRP that we use. It is first-order, that is, there are no higher-order operations to deal with types like `Event[Event[A]]`. A first-order FRP application has a static dependency graph since it cannot change dependencies between FRP primitives at runtime (yet, it is still powerful enough to write useful applications see [9]).

In general, each event compiles into a function and two variables. Figure 4 shows the `changes` event, the `changes` function determines the event's implementation. The newly

created variable `x94` tracks the propagation, that is, true if the behavior (`foldp`) updated, false if it did not. The other variable holds the latest value of `foldp`. In general, the compiled function executes the event's task (e.g., a mapped event will execute the map function). The status variable keeps track of whether or not an event has actually fired. These statuses are passed around to shortcut the FRP program's computation when needed, that is, when a `filter` operation evaluates to false. The final variable is the result of the computation that happens in an event's function, it is only relevant if the status boolean is true.

In fig. 4, you can see that `foldp` is compiled into one function (`fold_function`) and two variables: `x32` and `x89`. The former represents the `foldps` state, which in this example is a boolean that is flipped every time a new event happens. Behavior state is internal to the module (`SM_DATA`). `x89` tracks whether or not a change in the behavior has occurred, in the case of `foldp`, such a change is always present as long as the folded event updated.

6 Future Work

While our prototype works and shows that this idea is feasible it is not yet mature enough to actually use. An obvious extension would be to make use of the fact that modules are regular Scala objects by extending the EDSL with a deployment EDSL. A Scala EDSL where everything is typechecked and where one does not need to leave the same codebase (not even for deployment!) to write and configure an entire sensor network application would be highly maintainable. Making use of the type system to statically check topology properties seems feasible and desirable.

7 Related Work

For related work we focus on some specific projects related to FRP and macro-programming for the internet of things.

FRP Flask [4] and Regiment [5] are programming languages for sensor networks geared towards streaming data applications. Flask is an EDSL with an FRP interface (this time in Haskell). Regiment is a compiled language which mainly makes use of an FRP-like abstraction. While Regiment and Flask are more mature in performance and usability their communication primitives are limited to a spanning-tree topology.

Macro Programming Macro-programming languages focus on programming sensor network applications as a whole and not just as a collection of nodes. The most extreme form of this can be seen in projects such as Kairos [2] where the system infers the distribution topology completely from code. The other side of the spectrum can be seen in projects such as TinyOS [3] where modules (components) are connected explicitly in an RPC-style. Our approach, much like Regiment and Flask, exposes the communication between nodes as nice as possible using high-level language concepts such as FRP. Object-oriented approaches such as AmbientTalk [8] do not focus on modules and and distributing them but rather focuses on objects. AmbientTalk uses an actor approach and supports peer-to-peer topologies.

Acknowledgments

Bob Reynders holds an SB fellowship of the Research Foundation - Flanders (FWO). Dominique Devriese holds a postdoctoral fellowship of the Research Foundation - Flanders (FWO).

References

[1] C. Elliott and P. Hudak. Functional reactive animation. In *ICFP*, pages 263–273. ACM, 1997.
[2] R. Gummadi, O. Gnawali, and R. Govindan. Macro-programming wireless sensor networks using kairos. In *DCOSS*, pages 126–140. Springer, 2005.
[3] P. Levis, S. Madden, J. Polastre, R. Szewczyk, K. Whitehouse, A. Woo, D. Gay, J. Hill, M. Welsh, E. Brewer, and others. Tinyos: An operating system for sensor networks. In *Ambient intelligence*, pages 115–148. Springer.
[4] G. Mainland, G. Morrisett, and M. Welsh. Flask: Staged functional programming for sensor networks. In *ICFP*, pages 335–346. ACM, 2008.
[5] R. Newton, G. Morrisett, and M. Welsh. The regiment macroprogramming system. In *IPSN*, pages 489–498. ACM, 2007.
[6] J. Noorman, P. Agten, W. Daniels, R. Strackx, A. Van Herrewege, C. Huygens, B. Preneel, I. Verbauwhede, and F. Piessens. Sancus: Low-cost trustworthy extensible networked devices with a zero-software trusted computing base. In *USENIX*, pages 479–498, 2013.
[7] T. Rompf and M. Odersky. Lightweight modular staging: a pragmatic approach to runtime code generation and compiled DSLs. In *GPCE*, volume 46, pages 127–136. ACM, 2010.
[8] T. Van Cutsem, E. G. Boix, C. Scholliers, A. L. Carreton, D. Harnie, K. Pinte, and W. De Meuter. Ambienttalk: programming responsive mobile peer-to-peer applications with actors. *Computer Languages, Systems & Structures*, 40(3):112–136, 2014.
[9] D. Winograd-Cort and P. Hudak. Settable and Non-interfering Signal Functions for FRP: How a First-order Switch is More Than Enough. In *ICFP*, pages 213–225. ACM, 2014.