D'Artagnan: An Embedded DSL Framework for Distributed Embedded Systems

ABSTRACT

Programming distributed embedded systems gives rise to a number of challenges. The distributed nodes are typically resource constrained, requiring expert systems programming skills to manage the limited power, communication bandwidth, and memory and computation constraints. The challenge of raising the level of abstraction of programming such systems without incurring too high of an execution performance penalty is thus an important one, and many approaches have been explored in the literature.

In this paper we present a framework and domain specific language, to enable programming of such systems at a global network level. Our framework, D'Artagnan, enables the compilation, analysis, transformation and interpretation of high-level descriptions of stream processing applications in which information is received and processed in real-time. D'Artagnan, is a functional macroprogramming language embedded in Haskell, hiding where possible low-level detail, but allowing the developer to add hints and information to enable more efficient compilation to different target architectures. We present an initial implementation of this framework and show how it can be effectively used to program devices at a high level of abstraction through a case study of a building management system. Despite the high level abstraction, we also show that the loss in performance is minimal, and acceptable for many applications.

CCS Concepts

•Computer systems organization → Embedded systems; •Software and its engineering → Domain specific languages;

Keywords

distributed embedded systems; functional programming; embedded DSL; wireless sensor networks; domain specific languages

ACM ISBN 978-1-4503-2138-9.

DOI: 10.1145/1235

1. INTRODUCTION

Over the past 15 years, there has been a growing trend in embedding sensors and microprocessors in everyday objects so they can communicate information and interact with their environment [30]. This domain, now commonly referred to as the Internet of Things, has experienced great advances in technology such that reductions in the cost and size of sensors has made it possible to measure and sense information at high resolution, opening up a new dimension of applications. Environmentalists can track seabird populations and nesting behaviours in remote areas [25]. Volcanologists can easily deploy hundreds of sensors to detect explosions and volcanic activity, where information is filtered at source such that only interesting information is collected and analysed [33]. Building administrators can place motion, temperature and light sensors in every room in a building, to automatically turn off lights and cooling systems to optimise on energy consumption [8].

Applications in this domain are often seen as stream processing applications — a continuous flow of information is filtered, aggregated and acted upon in real-time. The amount of data and the processing involved may be non-trivial and across a distributed network of heterogeneous resource constrained, unreliable, wireless nodes. Developing applications on a network of such devices is not straightforward and the skills of expert low-level systems programmers are required to implement solutions. Programmers require a good understanding of energy consumption, distributed systems and intra-node communication, a varied range of devices and the heavy resource constraints imposed when using such devices. Radio transmission should be switched on only when needed, nodes need to be synchronised to communicate together and debugging these tiny devices is at times limited to a blinking LED. The need for expert low-level systems programming skills is somewhat slowing down progress and creating a higher barrier to entry. Ideally, we want to make programming of these devices more accessible to application programmers.

One way of addressing this difficulty is through the use of a domain specific language (DSL) [26]. By focusing on the domain, at the expense of general purpose use, DSLs provide a higher level of abstraction than general purpose programming languages and are ideal to make it easier to programme resource constrained devices quickly and effectively. A DSL can be used with less effort and cost, and even less skills. However, building a DSL may require significant initial investment to build the right tools for application development [16]. To overcome this, and reap the benefits of

a DSL early on, one commonly used approach is to embed a DSL within an existing language — creating a domain specific embedded language (DSEL). This is a powerful concept as the features of the host language become available to the embedded language, thereby making it possible to use a fully-fledged programming language to support the domain specific notions in the DSL [9].

Using this technique of embedding a language, we present a framework to describe stream processors. We chose Haskell as the host language as it provides several features — such as higher order functions, polymorphism and a strong type system — to support the embedding of D'Artagnan, a DSEL to analyse, generate, transform and interpret stream processor descriptions. The idea of using Haskell, or other functional languages, as a host language is not new. We take inspiration from the work done in hardware description with Lava [5] and in digital signal processing with Feldspar [2]. Our approach shares similarities to Flask [24] with the embedding of our DSL in Haskell to generate code for resource constrained devices. Our aim is to create a stream processor description that can (i) have different interpretations — simulated or translated (compiled) to low-level code; (ii) be analysed for both functional and non-functional aspects e.g. node placement and (iii) be optimised through transformations, such as alternative energy efficient communication strategies.

This approach can be used effectively in real scenarios, such as in an intelligent and energy-efficient building cooling and lighting systems. With the use of existing C compiler optimisation capabilities, the generated binary is comparable in performance to native C hand-coded versions.

2. BACKGROUND AND RELATED WORK

A wireless sensor node is comprised of a processing unit, a wireless communication interface, a number of sensors and/or actuators, and a limited power source — see Figure 1. A wireless sensor network (WSN) is made up of a number of nodes and can be considered as a distributed system, although with a number of differences to traditional distributed systems — the nodes and the overall network are not as reliable, and node failure and unavailability becomes a normal part of the behaviour of sensor networks. Constrained resources are an accepted fact in wireless sensor nodes. Limited processing capability, limited memory, and limited energy are three important constraints that have influenced how programmers implement applications for WSNs. The typical amount of memory (RAM) is tens of kilobytes, whereas program memory is up to 256KB. A programmer's focus is on writing efficient, tight code that takes advantage of the underlying architecture. This may often result in sacrifices in code structure and readability. Radio transceivers should be switched on only when needed and for short periods of time to reduce power consumption and extend application lifetime. The same applies to sensors and other external peripherals, as well as the microcontroller itself (by utilising low-power sleep modes). Coupled with limited debugging utilities it makes programming of such distributed devices a significant challenge.

2.1 WSN Programming Approaches

Programming wireless sensor nodes is not done in a conventional manner, and several approaches have been proposed in the past two decades. The approaches can be generally grouped into two: a low-level platform-centric approach and a high-level application-centric approach. Low-level, or node-level, programming models focus on abstracting hardware and allowing flexible control of nodes. High-level, or network-level, programming models give a global view of the network and focus on facilitating collaboration among sensors [32].

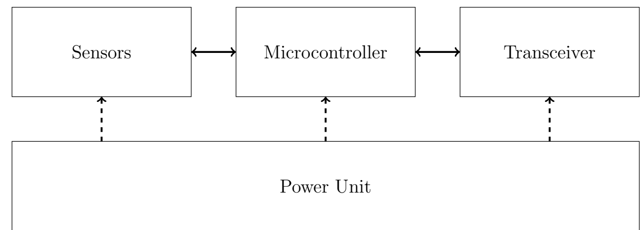

Figure 1: The structure of a wireless sensor node

2.1.1 Node-Level Programming

Low-level programming models are focused on giving fine grain control on the behaviour of each node where components are carefully switched on and off to optimise on energy utilisation. The level of systems programming expertise needed for node-level programming is high. There are two main approaches to node-level programming — operating systems and virtual machines.

Operating Systems

A number of operating systems have emerged in the sensor network community including TinyOS [21], Contiki [11], SOS [15], Mantis OS [4], Nano-RK [12], RETOS [7] and LiteOS [6]. TinyOS and Contiki are by far the most popular operating systems for WSNs, and have quite different characteristics.

TinyOS uses an event-based model to support concurrency. It is a static system so the application structure needs to be defined at design time, offering limited reconfiguration capability. TinyOS is a monolithic system and applications are compiled with the OS as a monolithic binary. Multi-threading is possible using thread libraries such as TinyThread or TOSThreads. Deluge is the standard reprogramming mechanism for TinyOS, which due to TinyOS's monolith approach means that applications are loaded with the OS kernel as a full image. The simulator of choice for TinyOS programs is TOSSIM.

Contiki is a modular dynamic system and is more flexible than TinyOS for reprogramming as only new or changed modules need to be loaded. Contiki has two communication stacks: uIP — allowing the node to communicate over the Internet; Rime — a lightweight communication stack designed for low power radios. Contiki supports multi-threading through the use of protothreads. Contiki programs can be simulated in Cooja before deployed in a real environment.

Virtual Machines

Maté [19] and ASVM [20] are interpreter-based virtual machines that run on top of TinyOS [21]. The main aim is to reduce the amount of data that needs to be transferred to

reprogram the nodes. Levis et al. [19][20] argue that users often do not know what sensor data would look like, and so must be able to reprogram sensor network nodes after deployment. Once a WSN is deployed with thousands of nodes in the field, it is impractical (or sometimes impossible) to reprogram them with physical contact. Therefore, the only option is to reprogram them wirelessly. Conventional approaches of reprogramming involve transmitting the full image to the node which consumes significant amount of energy. Using an application specific virtual machine approach, code is highly condensed reducing RAM requirements, interpretation overhead and propagation cost – making the approach highly beneficial from a reprogrammability point of view.

2.1.2 Network-Level Programming

There are two major approaches for network-level programming. One approach is a database abstraction, where the network is seen as a database from where information is gathered. The other is to provide a macro-programming language which provides a global view of the network and more flexibility for a wider variety of applications.

Database Query-like Languages

TinyDB [22] and Cougar [34] are two examples of a database query style approach. This abstraction allows the user to query the sensor network in a similar way as one would query a database using an SQL-like language.

```
SELECT AVG(volume), room FROM sensors
    WHERE floor=6
    GROUP BY room
    HAVING AVG(volume) > threshold
    SAMPLE PERIOD 30s
```

Queries are entered by the user on the base station, and they are optimised for energy consumption by determining where, when and how often data is sampled. The request is sent to the network where the nodes process the request, gather readings and send back the result.

Macroprogramming Languages

The database abstraction is ideal for a limited range of applications — applications where the primary focus is to collect and aggregate readings from a sensor network. Macroprogramming languages provide more flexibility to allow for a wider range of applications where data may flow between one node and another, and processed in network. Typically, a macroprogram is compiled into different node-level code and then loaded onto the individual nodes.

Pleiades [17] and Kairos [14] are imperative sequential languages where the programmer is provided with a centralised view of the sensor network. The nodes in the network can be addressed individually, and the local state on each node can be accessed. The programmer writes a sequential set of instructions to determine how the nodes are to interact with each other. The higher level of abstraction removes the complications of inter-node communication and node-level resource management. Pleiades is implemented as an extension of the C language, whereas Kairos is an extension of Python. By default, a Pleiades program has a sequential thread of control, but it introduces a language construct that allows concurrency of execution across multiple nodes. The Pleiades compiler analyses the code and determines *node-cuts* — a unit of work that can be executed on a single node. Code is then translated into nesC [13] programs that are executed on the TinyOS system. Kairos provides the programmer with additional constructs to access a node's one-hop neighbours. Using these constructs, the programmer can implicitly express the distributed data flow and distributed control flow. The Kairos model is similar to shared-memory based parallel programming model utilising message passing infrastructures.

COSMOS [1] is another architecture for macroprogramming heterogenous sensor networks. COSMOS is made up of a lean operating system called mOS and an associated programming language called mPL. A programmer can specify the aggregate system behaviour in terms of distributed data flow and processing. Functional components, written in a subset of the C language, are stand-alone modules that can be re-used across applications. Through composition of functional components, COSMOS allows direct specification of aggregate system behaviour. Contracts are used to affect and influence the low-level system behaviour and performance without forfeiting the high level programming interface for the application developer.

Wavescript [29], Regiment [28] and Flask [24] are macroprogramming functional languages. Wavescript is a domain specific language for stream processing applications with focus on asynchronous data streams. It is an ML-like functional language and uses three implementation techniques — it is evaluated in stream dataflow graphs, uses profile-driven compilation to enable optimisations and includes an extensible system for rewrite rules to capture and optimise algebraic properties in specific domains. Regiment, a Haskell-like language, is designed for spatiotemporal macroprogramming sensor networks that translates a global program into node-level event-driven code. The programmer sees the network as a set of spatially distributed time-varying signals, representing individual nodes or regions. Regiment provides constructs for aggregating streams, defining and manipulating regions. Compilation changes the program into an intermediate representation called *token machines* which provides facilities for local computation, sampling and communication with other nodes.

Flask is a stream processing DSL embedded in Haskell. Flask allows a programmer to combine stream operators from a pre-defined and extensible library to define a stream processing application. A Flask program is compiled into low-level nesC code, and allows functions to be defined in Red, a (partially) functional language, or directly in nesC using quasiquoting [3][23]. Low-level code can be safely embedded in the language and included in the compiled output. Flask provides a small number of primitive operations and powerful facilities to combine and create new first class operations — a technique used in several domain specific languages when embedded in a functional language. The power of abstraction means that the programmer does not need to worry about low-level details around how the nodes communicate with each other, or to make efficient use of available energy.

3. THE D'ARTAGNAN FRAMEWORK

Traditionally, sensor networks are programmed by writing a low-level program that is compiled and installed in each individual node. Sensor data is accessed directly and messages are passed to neighbours over radio. In a macro-programming model, the network is programmed as a whole and code is automatically generated for each node in the network. A higher abstraction level can make sensor network programming more accessible to non-expert programmers.

Our aim is to push the level of abstraction in programming such devices using techniques from the field of embedded languages. We embed our language in Haskell — a pure functional language which gives us several features which have been shown to be useful for this purpose, including higher-order functions, polymorphism and a strong type system.

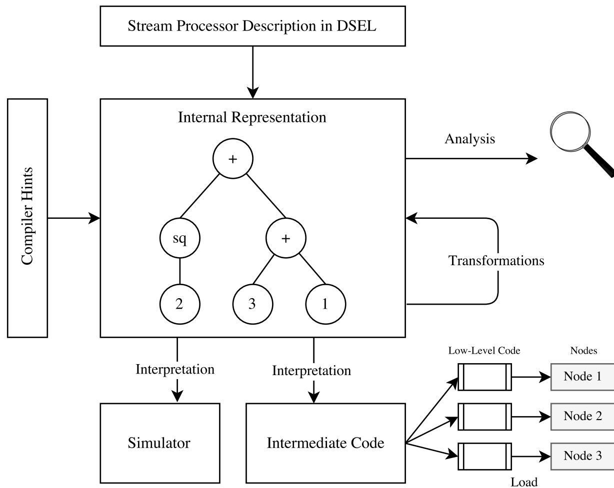

Figure 2: The D'ARTAGNAN Framework.

Figure 2 illustrates our framework. A stream processor description written in our DSEL is internally stored in a form that can be (i) analysed (ii) transformed and (iii) interpreted in different ways. This is the key feature of our approach.

A stream processor can be analysed by traversing the internal representation to determine, as an example, the number of radio messages used, the amount of power used, and to calculate the lifetime of the network in a specific configuration. It can be used to suggest optimal layouts of the distributed wireless network for optimisation on reduced radio messages.

The representation can also be transformed in different ways, possibly by using information from the analysis stage. For example, an automated transformation procedure might be used to move computation across the network to optimise the performance of power consumption by bringing computation closer to where data is sampled thus resulting in less radio transmission. The transformations can take *hints* from the programmer to transform the stream processor in such a way that is more applicable to the application environment. For example, if the network contains a more powerful node with a permanent, replenishable or bigger energy store, the programmer can influence the transformation such that more computation is done on this node.

The framework supports different interpretations of the stream processor internal representation. A simulation interpretation allows the programmer to observe the behaviour of the stream processor under different conditions — a setup which is harder to achieve in a real environment. The same representation can be used to generate node-level code for different platforms, thereby supporting the heterogeneity aspect of distributed embedded systems.

The ultimate goal of D'ARTAGNAN is to allow programmers to write complex stream processing applications using just a few lines of code that only refers to low-level aspects if required.

4. OVERVIEW OF THE SYSTEM

Since D'ARTAGNAN is embedded in Haskell, a stream processor description is defined using plain Haskell combined with a number of stream operators. Operators can be chained together similar to how functions are built. For example, the average temperature reading from three sensors uses the addition ('.+.') and division ('./.') stream operators shown by `average` in Listing 1.

A stream processor works by taking readings from sensors periodically — the period can be set at compile time[1], for example every 5 seconds. We use the term clock cycle to mean the period between one evaluation of the stream processor and the next. If we instantiate a simple `average` stream processor with three input sensors (see `average` in Listing 1), a new average value is calculated by taking new readings from the three sensors with every cycle.

This section presents the features of the system starting with basic building blocks and showing how these can be combined through higher level abstractions to build more complex operators.

4.1 Stream Operators

D'ARTAGNAN contains a number of basic building blocks to read input values from sensors and perform arithmetic and logical operations. The stream processor `moreThan50` (see Listing 1) creates a simple stream processor to determine if a reading from a temperature sensor is higher than 50 degrees Celsius, taking a stream of integers and outputting a stream of booleans. Whenever the input exceeds 50, the output goes high.

Throughout this section we will use a simplified fire alarm system as an example. Such a system would be made up of a number of temperature sensors and an alarm (e.g. a siren). As a first version, let us build a simple fire alarm system that alerts us when the temperature is higher than 50 degrees.

We can instantiate `firealarm50` as follows:

```
> firealarm50 (input (device 1) (sensor 1))
```

This instantiates a fire alarm system using sensor 1 on device 1 specified as parameters of the `input` node. However, `moreThan50` and `firealarm50` are too rigid — they can only detect temperatures higher than 50. If we wanted to have similar systems which trigger at different levels, higher or lower temperatures, then we would need to create similar stream processors such as `moreThan45`, `moreThan55`, etc.,

[1]The current version of D'ARTAGNAN allows the period to be set at compile time. In future versions, we want to add the ability to change during runtime.

Listing 1 System overview: Variants of fire alarm systems

```
average :: (Stream Int, Stream Int, Stream Int) -> Stream Int
average (input1, input2, input3) = (input1 .+. input2 .+. input3) ./. 3

moreThan50 :: Stream Int -> Stream Bool
moreThan50 input1 = input1 .>. 50

firealarm50 :: Stream Int -> Stream Bool
firealarm50 = moreThan50

firealarm :: Int -> Stream Int -> Stream Bool
firealarm threshold sensor1 = sensor1 .>. threshold

firealarm2 :: Int -> (Stream Int, Stream Int) -> Stream Bool
firealarm2 threshold (sensor1, sensor2) = (sensor1 .>. threshold) .||. (sensor2 .>. threshold)

firealarmNotification :: Int -> Stream Int -> Stream Bool
firealarmNotification threshold sensor1 = ((pre 0 sensor1) .<=. threshold) .&&. (sensor1 .>. threshold)

stickyAlarm threshold sensor1 = let x = (pre False x) .||. (sensor1 .>. threshold)
                                in x

firealarmPlus :: Int -> Stream Int -> (Stream Bool, Stream Bool)
firealarmPlus threshold sensor1 = let x = (pre False x) .||. (sensor1 .>. threshold)
                                      y = ((pre 0 sensor1) .<=. threshold) .&&. (sensor1 .>. threshold)
                                  in (x, y)
```

which is not ideal. Through a first higher level of abstraction, thresholds can be passed in as parameters to create a more generic system such that we can set any limit that we want at instantiation stage — see `firealarm` in Listing 1, which is not a stream processor *per se,* but a whole family of stream processors generated by different parameters passed to the function.

Now consider a fire alarm system which uses two sensors, such that the alarm will sound if any of the two sensors has a reading higher than the threshold (see `firealarm2`).

Stream processor descriptions in D'ARTAGNAN are strongly typed. The types of our DSEL are embedded in Haskell's type system such that the compiler does not allow the construction of wrongly typed expressions. For example, if any of two sensors `sensor1` or `sensor2` is not a stream of booleans, the expression `(sensor1 .||. sensor2)` would be rejected at compilation stage due to mismatching types. This is one of the most useful features our DSEL inherits from Haskell. Type errors are detected at compile time, rather than runtime, drastically improving dependability and reliability. Internally (and invisible to the users of the language), we make use of phantom types [18] to ensure strong typing.

4.2 Memory Capabilities

A stream processor is evaluated once with every clock cycle. Without the ability to use readings or calculated outputs from previous cycles, the stream processor can only make use of readings taken during the current cycle. For most applications, this is too restrictive. There are situations where we would want to use a previous sensor reading to compare it to the current. For example, consider an enhanced fire alarm system which sends an alert the moment that the temperature exceeds a specific value, rather than continuously when the reading exceeds the threshold. Such a system requires access to previous readings in order to compare them to new ones. In our DSEL, the `pre` operator allows the use of a value from a previous clock cycle.

A fire alarm notification, using `pre` for memory capability is shown by `firealarmNotification` in Listing 1. A visual representation of the same system is shown in Figure 3.

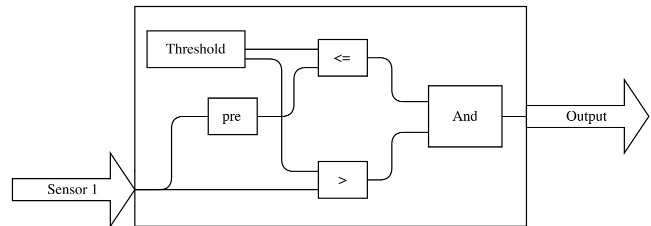

Figure 3: A fire-alarm notification system

Memory capability is particularly useful when combined with feedback loops, where the output of the stream processor is required in the following clock cycle. For instance, a system that outputs whether an input stream has, at any point in the past, exceeded a threshold, will output `False` for as long as the reading is less than the threshold, but outputs `True` from the point the reading is higher than the threshold onwards until the system is reset — even if a new reading is eventually below the threshold. This is illustrated in the definition of `stickyAlarm` in Listing 1 and Figure 4. Note that the apparently unbounded recursion when defining the feedback loop is internally handled using Haskell's lazy evaluation to unroll it only until a cycle is detected using observable sharing [10].

4.3 Communication Operators

Applications deployed on WSNs make use of intra-node radio communication to achieve the desired application goals. Instructions, readings and calculated values may be passed between one node and another. We support two forms of point-to-point communication — `Pull` and `Push` (see Figure 5). `Pull` is based on a request and response pair of messages, and makes use of two radio messages. On the other

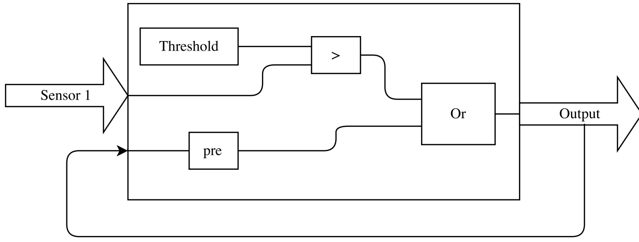

Figure 4: A sticky alarm system — once triggered, remains *ON* until system is reset.

hand, `Push` uses one radio message as there is no request message and information is sent preemptively at regular intervals. The choice between `pull` and `push` depends on the application needs. In situations where information needs to be passed from one node to another continuously on a periodical basis, then `push` is the preferred option as it will use one radio message and therefore less energy is consumed. `Pull` is used in all other situations as it provides fine grain control on triggering radio communication between nodes, and no radio messages are wasted when information may not be required or is discarded.

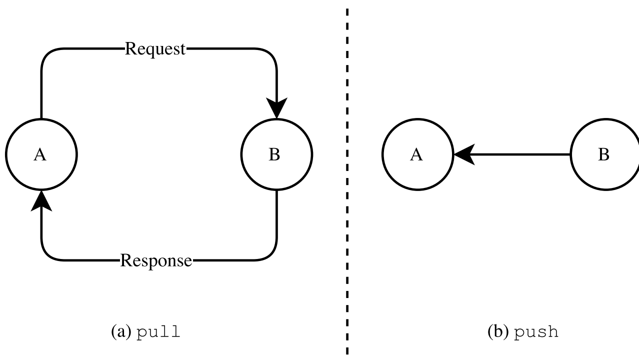

Figure 5: Point-to-point communication models

D'Artagnan allows the programmer to define a stream processor without the need to explicitly specify how communication is to be done between nodes. This approach makes it easier for the programmer to reason about stream processors. By default, as part of an automatic transformation, D'Artagnan introduces `pull` as a communication operator when information is passed between two nodes. Unless explicitly changed, processing is done on the start node and the communication between nodes is left until the latest possible point of interaction. The programmer can give explicit instructions, by using `push` and `pull`, to move computation on to different nodes — possibly closer to where readings are taken so as to reduce communication and therefore reduce energy consumption.

```
input1 = input (device 1) (sensor 1)
input2 = input (device 2) (sensor 1)
input3 = input (device 2) (sensor 2)
sp = input1 .+. (push (device 2) (input2 .*. input3))
```

In the example above for stream processor `sp`, `input2` and `input3` are on the same device — so rather than having readings from the different sensors transmitted in separate messages, the programmer can indicate that the computation `input2 .*. input3` should be processed on `device 2` and only the result is pushed out to `device 1`.

In the future, we would like to introduce different strategies such that communication operators are inserted optimally into the stream processor to minimise communication.

4.4 Stream Tuples

For most stream processing applications, having just one output stream is often too restrictive to build interesting applications. For example, in our fire alarm system earlier on, we needed to choose between one of two actions — either sound an alarm or else send a notification. We would like our system to do both. One way of addressing this is by extending our DSEL with tuples of streams, for example `(stream1, stream2)`.

We can combine the stream processors `firealarmNotification` and `stickyAlarm` into one as shown by `firealarmPlus` in Listing 1.

The output of `firealarmPlus` is a pair of streams. The first element of the pair represents the sound siren and the second is the alert notification. Table 1 shows an example scenario with input and output values.

Time	Sensor1	Output (Siren, Notify)
t	45	(False, False)
t+1	48	(False, False)
t+2	55	(True, True)
t+3	56	(True, False)
t+4	57	(True, False)

Table 1: Example (`firealarmPlus`) with output tuple of streams

5. INTERPRETATIONS

One of the strengths of our approach is that the same stream processor description can have multiple interpretations. These interpretations perform stream processor analyses such as simulation to calculate the output of a stream processor given certain input values, and generation of low-level code to be loaded on devices.

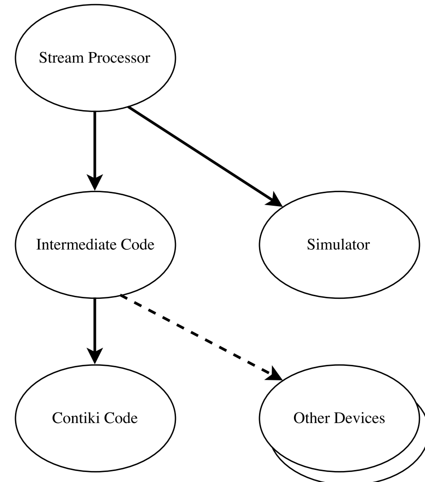

Figure 6: Different interpretations of a stream processor

Listing 2 Code listing

```
doubleUp :: ((Stream Int, Stream Int) -> Stream Int) -> (Stream Int, Stream Int) -> Stream Int
doubleUp f (sensor1, sensor2) = let x = f (sensor1, sensor2)
                                in x .+. x

doubleUp2 :: ((Stream Int, Stream Int) -> Stream Int) -> (Stream Int, Stream Int) -> Stream Int
doubleUp2 f (sensor1, sensor2) = f (sensor1, sensor2) .+. f (sensor1, sensor2)
```

5.1 Simulator

D'Artagnan provides a simulator interpretation that calculates the output of a stream processor given input values whilst still in the Haskell environment. This is useful in that it allows the developer to check and test that the behaviour of the simulator is as intended under certain test conditions — something which is harder to achieve when the application is running in a real environment due to the difficulty of setting environment parameters.

We simulate the behaviour of the `firealarmPlus` stream processor by providing concrete inputs coming from `sensor1`. Such inputs are given in the form of a list of values with values corresponding to sensor readings for every clock tick i.e. $[v_0, v_1, v_2, \ldots]$.

```
> simulate firealarmPlus [45,48,55,56,57]
[(False, False),(False, False),(True, True),
     (True, False),(True, False)]
```

5.2 Intermediate Code

One of the main aims of our system is to be able to generate code that can be readily uploaded onto devices. We do this generation in a 2-stage approach — we first generate intermediate code, effectively an abstract representation of the stream processor, from which we specialise this to device-specific code. This approach has several advantages. First of all, the operating systems of WSNs typically support the C programming language or a WSN-specific dialect (e.g. nesC for TinyOS) [27]. Our intermediate code allows us to have one common language for different devices so that we can separate the syntax from the semantics. Secondly, it makes our approach more extensible, in that generators for other languages can be easily added on. This 2-staged approach has also been used in Feldspar [2] and Regiment [28].

5.3 Device level code: Contiki

Translation from intermediate code, which is already in the form of imperative sequences of assignments, to device specific code is relatively straightforward. It is a matter of getting the correct syntax for the sequence of abstract statements. In our current implementation, we generate code for Contiki as an example. Even with Contiki, different devices may require slightly different syntax — for example, the use of a different type of temperature sensor, or possibly a slightly different version of Contiki. Our current framework generates two types of Contiki; one for the WSN430 nodes at the FIT IoT-LAB[2] test bed and another one for the AdvanticSys CM5000. Other C variants can be added with relative ease.

[2]https://www.iot-lab.info

6. USE CASE: INTELLIGENT COOLING AND LIGHTING SYSTEMS

In order to illustrate the effectiveness of D'Artagnan at higher levels of abstraction, we present an application for smart buildings, building upon the stream operators described in Section 4 to construct higher level components. These components are used at a level of abstraction which omits internal embedded system details altogether.

We present a generic solution for smart building management which can be instantiated for any given room layout plan — supporting automatic switching on and off of lights and cooling systems when motion is detected in a neighbouring room. Further, the lights are only switched on if there is not enough natural light, and cooling systems are only turned on if the room temperature is too high. The solution takes room layout information — which rooms are adjacent to which rooms — and, assuming three types of sensors in every room for motion, light and temperature detection, creates a specific building stream processor tailored to the specified room layout. Using sensors' readings as inputs, the building stream processor can generate code to control lights, cooling systems, etc (see Figure 7). The system can support multiple device and sensors of the same type in the same room, such that more reliable readings are taken. In this example, we use sensor readings from rooms, such that whenever motion detection sensors in a room are triggered, lights are switched on in the room and neighbouring ones. Figure 8 illustrates internal detail of the building stream processor — a room stream processor.

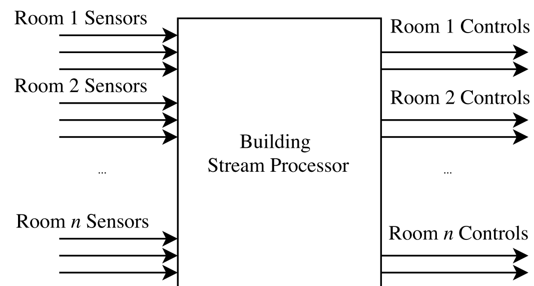

Figure 7: Inputs and outputs of a building stream processor

6.1 Stream Handling Components

Higher level stream handling components are needed to transform sensor readings into application specific meaningful information. Listing 3 shows the definition of four stream handlers that will be used in building the application.

When there is more than one sensor in a room, we use `average` to combine sensors with numeric output (e.g. temperature, light-level sensors) to obtain a more reliable reading for that room. Similarly we use `anyOf` to combine sensors

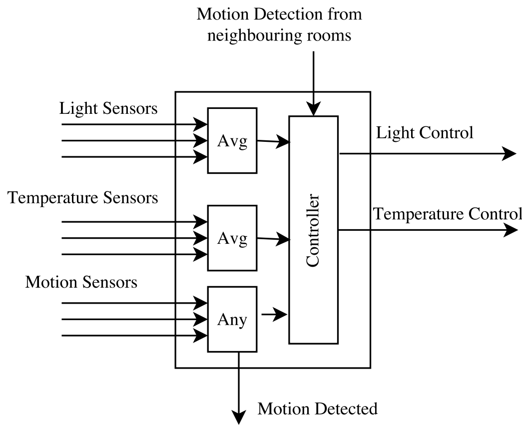

Figure 8: Internals of a room stream processor

Listing 3 Higher level functions

```
average :: [Stream Float] -> Stream Float
average ss = sum ss ./. consStream (length ss)

anyInput :: [Stream Bool] -> Stream Bool
anyInput ss = foldl1 (\x y -> x .||. y) ss

minControl :: Min -> Stream Float -> Stream Bool
minControl min s = s .<=. consStream min

maxControl :: Max -> Stream Float -> Stream Bool
maxControl max s = s .>=. consStream max
```

with boolean readings (e.g. motion detection sensors placed in the same room). Other such agglomeration combinators can also be defined e.g. a median or majority combinator can be more effective if some of the devices are known to fail regularly, hence allowing us to ignore outlier values.

The threshold functions `minControl` and `maxControl` are used to determine whether to switch on or off lighting and cooling systems based on whether the combined values fall below or above certain thresholds — in other words, a behaviour similar to a thermostat.

6.2 Room Layout Representation

In order to describe the particularities of a building to generate the code for all the devices in the different rooms, we provide data structures to represent which sensors are in which rooms and also room adjacency. Consider one particular room layout shown in Figure 9, which includes information about devices with on-board sensors in the rooms. Every device is equipped with three types of sensors — motion, light and temperature — and some rooms have more than one device. Having multiple devices in the same room increases the reliability of the application.

The building topology is represented by a graph using the data types `Plan` and `Room` — see Listing 4. `Plan` is an adjacency list of rooms (represented as a list of pairs of rooms), while `Room` stores information about the room: its name, and references to the motion, light and temperature sensors in that room. The instantiation of the example shown in Figure 9 is given in Listing 5.

In this specific room layout, when motion is detected in Room 1, lights and cooling in Rooms 1 and 2 will be switched

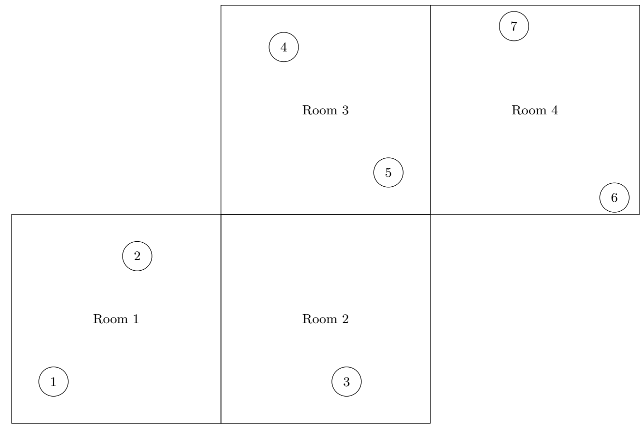

Figure 9: Room layout with device placement.

on — if there is not enough natural light, and/or temperature is too high. As a person walks into Room 2, the lights and cooling for Room 3 will automatically switch on (if light and/or temperature conditions are met) — lights and cooling in Rooms 1 and 2 will remain on, as they have already been switched on. As the person walks into Room 3, lights and cooling in Room 1 are turned off and those in Room 4 are turned on.

Listing 4 Building data types

```
type Plan = [(Room, Room)] -- list of rooms
data Room = Room {
              name    :: String,
              motionS :: [MotionSensor],
              lightS  :: [LightSensor],
              tempS   :: [TempSensor]}
  deriving (Eq, Show)
```

6.3 Application Implementation

Typically, one would program the devices for a particular building, with a particular topology. However, any changes in sensor deployment would require reprogramming from scratch. Similarly, given a new building requires reprogramming the devices for that building from scratch. However, with D'Artagnan we can abstract up, and program a generic solution which works for any given building topology. If a new device is added to a room, one simply changes the building description passed on to the generic solution and automatically obtain code which is to be deployed on the devices. The implementation of such a generic solution can be given in just 10 lines of code — see Listing 6.

The inputs and outputs of the application vary depending on the topology used. The inputs are linked to the number of devices present in the rooms — in the example layout shown in Figure 9 with seven devices in four rooms, the application has 21 inputs — three types of sensors for each device. The number of outputs of the application is determined by the number of rooms in the `Plan` — in the example, the output is made up of eight boolean streams, one for each of light and cooling controllers in each of the four rooms. These outputs need to be connected to the light and cooling controllers for each respective room.

In the simulation interpretation, it is possible to test the

Listing 5 Instantiation of system

```
room1 = Room {name="room1", motionS=[motionSensor1, motionSensor2], lightS=[lightSensor1, lightSensor2],
              tempS=[tempSensor1, tempSensor2]}
room2 = Room {name="room2", motionS=[motionSensor3], lightS=[lightSensor3],
              tempS=[tempSensor3]}
room3 = Room {name="room3", motionS=[motionSensor4, motionSensor5], lightS=[lightSensor4, lightSensor5],
              tempS=[tempSensor4, tempSensor5]}
room4 = Room {name="room4", motionS=[motionSensor6, motionSensor7], lightS=[lightSensor6, lightSensor7],
              tempS=[tempSensor6, tempSensor7]}

plan = [(room1, room2), (room2, room3), (room3, room4)]
```

Listing 6 Main application

```
automation :: Plan -> [(Stream Bool, Stream Bool)]
automation plan = map (roomAutomation plan) (getRooms plan)

roomAutomation :: Plan -> Room -> (Stream Bool, Stream Bool)
roomAutomation plan room = (autoMinControl motionSensors lightSensors 50, autoMaxControl motionSensors tempSensors 25)
  where
    adjacentRooms = adjacent plan room
    motionSensors = msToStream $ getMotionSensors adjacentRooms
    lightSensors  = lsToStream $ getLightSensors adjacentRooms
    tempSensors   = tsToStream $ getTempSensors adjacentRooms
```

behaviour of the application under different input values.

```
> simulate (automation plan) simulatedValues
```

Where `simulatedValues` is a list of input values for all sensors. In simulation mode, the output of the application is a list of tuples for the different lighting and cooling switches in each room. In this example with four rooms, there will be four tuples made up of two boolean streams — reflecting whether lighting and cooling is on or off in the respective room.

```
-- Output format is [(roomN_light, roomN_cooling), ...]
Output at T1 = [(True,True),(True,True),
                (False,False),(False,False)]
Output at T2 = [(True,True),(True,True),
                (True,True),(False,False)]
```

At T1, motion was detected in Room 1. The lights and cooling systems for Room 1 and Room 2 are turned on — output at T1. At T2, motion was detected in Room 2 so the lights and cooling of Room 3 are also turned on — output at T2.

In a Contiki interpretation, the source code is generated uniquely for every device. The generated code takes care of communication between the devices as one device requests information from another. In this example, since a communication type is not explicitly defined, the Pull model is used.

6.4 Discussion

As presented, D'Artagnan is a high level DSL that makes it easier to build applications for IoT-devices. In this section, we evaluate how an application written in D'Artagnan compares to an equivalent application hand-coded in C for Contiki. In order to evaluate the performance of the two variants, we performed experimentation on the FIT IoT-LAB test bed — a platform suitable for testing small wireless devices in a real environment.

Lines of Code: As one would expect, significantly fewer lines of code are required using our framework and DSL, as compared to a hand-coded version — plus we have a more general solution. In this example application, 10 lines of D'Artagnan code[3] are enough to create a generic intelligent and energy efficient cooling and lighting system. The code does not need to change if different room layouts and additional sensors are introduced. The room plan (Listing 5) is updated to reflect the exact layout, and passed in as input to the application and node-level code is generated automatically to reflect the layout. On the other hand, 516 lines of code are required to implement the system directly in C for the current configuration. For different layouts, or additional WSNs, the C code may need to be modified and size will increase linearly with the number of rooms and nodes introduced.

	D'Artagnan	Hand Coded
Lines of Code	10	516
Radio Messages	36	36

Table 2: Comparison of DSL generated versus hand-coded

Radio Messages: The two implementations generate the same amount of radio traffic. This is partly due to following the same design concept, in that point-to-point communication is used with a request/response pattern and a separate message for every reading. It is however possible to reduce the number of messages from 36 to 6, where each node transmits its own three readings (motion, light and temperature) periodically to the master node in one message. Changing from Pull to Push reduces the 36 messages by half, to 18. Sending the three readings in one message will further reduce the 18 messages to 6. This improvement can be implemented for both versions — although the current version of D'Artagnan does not support three readings in one message although this is planned to be added to a future version of the language.

[3]This includes type declarations which can be deduced by the Haskell compiler and are thus unnecessary.

7. PERFORMANCE EVALUATION

In this section we compare different implementations of a processing intensive task to be able to assess performance penalties induced by our approach versus a hand-coded implementation. We use an audio sound soft-clipping algorithm, which for the limitations of wireless sensor nodes can be considered a processing intensive task.

7.1 Clipping

In audio, when the output from an amplifier exceeds the noise levels supported by a speaker, the end result is hard-clipping — the top portion of an audio waveform is clipped. The abrupt juncture between the normal waveform and the horizontal clipped line generates high harmonic frequencies which create a harsh unpleasant sound — see Figure 10.

Soft-clipping is used to reduce the harshness of clipping, by creating a smoother transition from the waveform to the clipped section. The junction between the normal wave and the clipped horizontal line will be slightly curved, rather than at an abrupt angle. Although there is still distortion, less high-pitched harmonics are created causing the output to be much smoother.

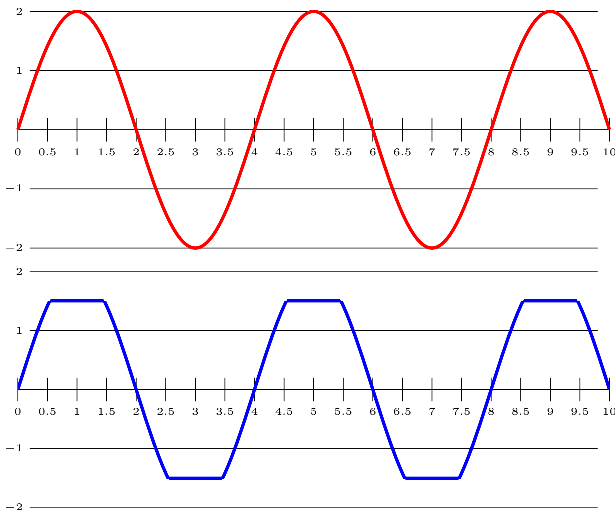

Figure 10: Sine wave (top) and hard-clipped sine wave (bottom)

7.2 Implementation

We have implemented three versions of the soft-clipping algorithm. The first implementation is hard-coded in C and makes use of our small math library with an implementation of an arctan approximation used in soft-clipping calculations. The second implementation makes use of the same math library, but the main algorithm is written in D'Artagnan. The aim of this implementation is to assess any penalties introduced by using the DSEL code generation. The third implementation is written completely in D'Artagnan, including the mathematical functions for arctan approximations. This last approach can be used on platforms that do not have native implementations available, and has the advantage that the implementation can be translated to any platform which D'Artagnan supports without the need to re-implement the math library.

7.3 Evaluation

For our experiments, we use Contiki on an AdvanticSys CM5000, which is equipped with an MSP430F1611 Texas Instruments micro-controller and 48KB of program flash. All three implementations were compiled using the MSP430 GCC (version 4.6.3) with different optimisation levels to study how compiler optimisations interact with our code generation. Optimisation levels range from level 0 (-O0), which disables all code optimisations, to level 3 (-O3), with most code optimisations including function inlining, optimised nonlinear parallelised assembly and global-algorithm register allocation.

	Optimisation Level			
	-O0	-O1	-O2	-O3
Implementation 1: Hand-coded C version				
Bytes Programmed	41522	28570	27928	39084
Avg Duration (s)	114.44	55.94	55.86	55.84
Implementation 2: D'Artagnan *with C Math library*				
Bytes Programmed	41634	28578	27936	39096
Avg Duration (s)	116.56	56.12	56.02	56.10
% Diff with Impl 1	1.85%	0.32%	0.29%	0.47%
Implementation 3: Full D'Artagnan				
Bytes Programmed	44936	28396	27784	38958
Avg Duration (s)	225.76	56.12	56.08	56.18
% Diff with Impl 1	97.27%	0.32%	0.39%	0.61%

Table 3: Results for different implementations

The results in Table 3 indicate that, as expected, with no compiler optimisations (-O0), Implementation 3 — Full D'Artagnan— is significantly more inefficient, with the duration of the test nearly double that of the other two implementations. The footprint (code size) is also the largest of the three implementations.

The results also show that any inefficiencies introduced by D'Artagnan during automatic code generation are completely cancelled out when any level of compiler optimisation is used. This gives us confidence in that our approach, coupled with standard compiler optimisations, will still produce compact and efficient binaries — a much desired outcome when programming resource constrained devices.

One observation that comes to light from this exercise is the need to make the language extensible with native functions — for example, trigonometric functions which may be optimised for the device compared to the DSEL generated code.

8. CONCLUSIONS AND FUTURE WORK

We have described D'Artagnan, an embedded DSL framework that brings functional programming to distributed embedded systems. By using an internal representation of a stream processor description, we can analyse, transform and interpret in different ways. D'Artagnan allows the programmer to use the power of functional programming to build sensor network applications. We have shown through examples that any overheads introduced by D'Artagnan are adequately compensated for by C compiler optimisations and that the framework can be extended to add even more higher layers of abstraction. Libraries can be created for specific application domains to make writing of applications even more easy.

The system which is closest to D'ARTAGNAN is Flask — a stream processing DSL embedded in Haskell. However, Flask makes use of Red, a restricted subset of Haskell which lacks support for type classes, disallows recursive data-types and functions, and closures cannot be allocated. D'ARTAGNAN is different in that it inherits all the features and functionality from Haskell, providing greater expressiveness to the programmer. Also, Flask makes use of quasiquoting to allow code to be written in nesC and used directly in code generation. In a similar manner to Regiment and Feldspar, D'ARTAGNAN makes use of an intermediate representation to convert to low-level device code.

We envisage various directions for D'ARTAGNAN. Firstly, we want to enhance the language with hints, such that the programmer can influence transformations based on his/her expertise and knowledge on how the application is going to be used in a real environment. We also want to make the framework more extensible by allowing easier pluggability of node-native functions, new language constructs and translation to new target platforms. We also intend to explore over-the-air programming and an interpreter-based compilation to minimise footprint for new code transfers. We believe it is also possible to combine different applications, written by different programmers, to be loaded onto the same sensor network. Finally, D'ARTAGNAN raises interesting questions in how high can we raise the abstraction level of programming such systems. From the smart building example, it is evident that there is much to be gained with compositional systems — an observation which coincides with similar languages which have been defined for other domains [31].

9. REFERENCES

[1] A. Awan, S. Jagannathan, and A. Grama. Macroprogramming heterogeneous sensor networks using cosmos. In *ACM SIGOPS Operating Systems Review*, volume 41, pages 159–172. ACM, 2007.

[2] E. Axelsson, K. Claessen, G. Dévai, Z. Horváth, K. Keijzer, B. Lyckegård, A. Persson, M. Sheeran, J. Svenningsson, and A. Vajdax. Feldspar: A domain specific language for digital signal processing algorithms. In *Formal Methods and Models for Codesign (MEMOCODE), 2010 8th IEEE/ACM International Conference on*, pages 169–178, July 2010.

[3] A. Bawden et al. Quasiquotation in lisp. In *PEPM*, pages 4–12. Citeseer, 1999.

[4] S. Bhatti, J. Carlson, H. Dai, J. Deng, J. Rose, A. Sheth, B. Shucker, C. Gruenwald, A. Torgerson, and R. Han. Mantis os: An embedded multithreaded operating system for wireless micro sensor platforms. *Mobile Networks and Applications*, 10(4):563–579, 2005.

[5] P. Bjesse, K. Claessen, M. Sheeran, and S. Singh. Lava: Hardware design in haskell. In *Proceedings of the Third ACM SIGPLAN International Conference on Functional Programming*, ICFP '98, pages 174–184, New York, NY, USA, 1998. ACM.

[6] Q. Cao, T. Abdelzaher, J. Stankovic, and T. He. The liteos operating system: Towards unix-like abstractions for wireless sensor networks. In *Information Processing in Sensor Networks, 2008. IPSN'08. International Conference On*, pages 233–244. IEEE, 2008.

[7] H. Cha, S. Choi, I. Jung, H. Kim, H. Shin, J. Yoo, and C. Yoon. Retos: resilient, expandable, and threaded operating system for wireless sensor networks. In *Information Processing in Sensor Networks, 2007. IPSN 2007. 6th International Symposium on*, pages 148–157. IEEE, 2007.

[8] H. Chen, P. Chou, S. Duri, H. Lei, and J. Reason. The design and implementation of a smart building control system. In *e-Business Engineering, 2009. ICEBE'09. IEEE International Conference on*, pages 255–262. IEEE, 2009.

[9] K. Claessen and G. J. Pace. An embedded language framework for hardware compilation. In *Designing Correct Circuits '02, Grenoble, France*, Apr. 2002.

[10] K. Claessen and D. Sands. *Observable Sharing for Functional Circuit Description*, pages 62–73. Springer Berlin Heidelberg, Berlin, Heidelberg, 1999.

[11] A. Dunkels, B. Gronvall, and T. Voigt. Contiki-a lightweight and flexible operating system for tiny networked sensors. In *Local Computer Networks, 2004. 29th Annual IEEE International Conference on*, pages 455–462. IEEE, 2004.

[12] A. Eswaran, A. Rowe, and R. Rajkumar. Nano-rk: an energy-aware resource-centric rtos for sensor networks. In *Real-Time Systems Symposium, 2005. RTSS 2005. 26th IEEE International*, pages 10–pp. IEEE, 2005.

[13] D. Gay, P. Levis, R. Von Behren, M. Welsh, E. Brewer, and D. Culler. The nesc language: A holistic approach to networked embedded systems. In *Acm Sigplan Notices*, volume 38, pages 1–11. ACM, 2003.

[14] R. Gummadi, O. Gnawali, and R. Govindan. Macro-programming wireless sensor networks using kairos. In *International Conference on Distributed Computing in Sensor Systems*, pages 126–140. Springer, 2005.

[15] C.-C. Han, R. Kumar, R. Shea, E. Kohler, and M. Srivastava. A dynamic operating system for sensor nodes. In *Proceedings of the 3rd international conference on Mobile systems, applications, and services*, pages 163–176. ACM, 2005.

[16] P. Hudak. Modular domain specific languages and tools. In *Software Reuse, 1998. Proceedings. Fifth International Conference on*, pages 134–142, Jun 1998.

[17] N. Kothari, R. Gummadi, T. Millstein, and R. Govindan. Reliable and efficient programming abstractions for wireless sensor networks. In *ACM SIGPLAN Notices*, volume 42, pages 200–210. ACM, 2007.

[18] D. Leijen and E. Meijer. Domain specific embedded compilers. In *ACM Sigplan Notices*, volume 35, pages 109–122. ACM, 1999.

[19] P. Levis and D. Culler. Maté: A tiny virtual machine for sensor networks. *ACM Sigplan Notices*, 37(10):85–95, 2002.

[20] P. Levis, D. Gay, and D. Culler. Active sensor networks. In *Proceedings of the 2nd conference on Symposium on Networked Systems Design & Implementation-Volume 2*, pages 343–356. USENIX Association, 2005.

[21] P. Levis, S. Madden, J. Polastre, R. Szewczyk, K. Whitehouse, A. Woo, D. Gay, J. Hill, M. Welsh, E. Brewer, et al. Tinyos: An operating system for

sensor networks. In *Ambient intelligence*, pages 115–148. Springer, 2005.

[22] S. R. Madden, M. J. Franklin, J. M. Hellerstein, and W. Hong. Tinydb: an acquisitional query processing system for sensor networks. *ACM Transactions on database systems (TODS)*, 30(1):122–173, 2005.

[23] G. Mainland. Why it's nice to be quoted: quasiquoting for haskell. In *Proceedings of the ACM SIGPLAN workshop on Haskell workshop*, pages 73–82. ACM, 2007.

[24] G. Mainland, G. Morrisett, and M. Welsh. Flask: Staged functional programming for sensor networks. In *ACM Sigplan Notices*, volume 43, pages 335–346. ACM, 2008.

[25] A. Mainwaring, D. Culler, J. Polastre, R. Szewczyk, and J. Anderson. Wireless sensor networks for habitat monitoring. In *Proceedings of the 1st ACM international workshop on Wireless sensor networks and applications*, pages 88–97. ACM, 2002.

[26] M. Mernik, J. Heering, and A. M. Sloane. When and how to develop domain-specific languages. *ACM computing surveys (CSUR)*, 37(4):316–344, 2005.

[27] L. Mottola and G. P. Picco. Programming wireless sensor networks: Fundamental concepts and state of the art. *ACM Comput. Surv.*, 43(3):19:1–19:51, Apr. 2011.

[28] R. Newton, G. Morrisett, and M. Welsh. The regiment macroprogramming system. In *Proceedings of the 6th International Conference on Information Processing in Sensor Networks*, IPSN '07, pages 489–498, New York, NY, USA, 2007. ACM.

[29] R. R. Newton, L. D. Girod, M. B. Craig, S. R. Madden, and J. G. Morrisett. Design and evaluation of a compiler for embedded stream programs. *SIGPLAN Not.*, 43(7):131–140, June 2008.

[30] G. J. Pottie and W. J. Kaiser. Wireless integrated network sensors. *Communications of the ACM*, 43(5):51–58, 2000.

[31] M. Sheeran. Hardware design and functional programming: a perfect match. *J. UCS*, 11(7):1135–1158, 2005.

[32] R. Sugihara and R. K. Gupta. Programming models for sensor networks: A survey. *ACM Transactions on Sensor Networks (TOSN)*, 4(2):8, 2008.

[33] G. Werner-Allen, J. Johnson, M. Ruiz, J. Lees, and M. Welsh. Monitoring volcanic eruptions with a wireless sensor network. In *Proceeedings of the Second European Workshop on Wireless Sensor Networks, 2005.*, pages 108–120, Jan 2005.

[34] Y. Yao and J. Gehrke. The cougar approach to in-network query processing in sensor networks. *ACM Sigmod record*, 31(3):9–18, 2002.