Doctoral Symposium: An Embedded DSL Framework for Distributed Embedded Systems

Adrian Mizzi
Department of Computer Science
University of Malta, Malta
adrian.mizzi.00@um.edu.mt

Joshua Ellul
Department of Computer Science
University of Malta, Malta
joshua.ellul@um.edu.mt

Gordon Pace
Department of Computer Science
University of Malta, Malta
gordon.pace@um.edu.mt

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.

The work presented in this paper investigates a framework and domain specific language (DSL) to enable programming of such systems at a global network level. The framework 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.

The ongoing research aims at investigating the following directions. Firstly, a framework to allow the manipulation and multiple interpretations of a stream processor description to support the heterogeneity aspect of devices. Secondly, a language to hide the low-level detail from the developer, while allowing the developer to add hints to enable more efficient compilations. Thirdly, through transformations, different stream processing applications can be fused together to run on the same distributed network to make best of use of available resources.

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 Reference format:
Adrian Mizzi, Joshua Ellul, and Gordon Pace. 2017. Doctoral Symposium: An Embedded DSL Framework for Distributed Embedded Systems. In *Proceedings of DEBS '17, Barcelona, Spain, June 19-23, 2017,* 4 pages.
DOI: http://dx.doi.org/10.1145/3093742.3093906

DEBS '17, Barcelona, Spain

DOI: http://dx.doi.org/10.1145/3093742.3093906

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 [18]. 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. 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. 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.

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 at source. 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 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) [15]. 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 program 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 [8]. 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 [5].

This paper outlines ongoing research work to use language embedding techniques to create a framework to describe and automatically generate embedded systems code for stream processor applications. Haskell is used 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 the language, which we call 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 [4] and in digital signal processing with Feldspar [2]. Our approach shares similarities to Flask [14] 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.

2 RESEARCH CHALLENGES

The research questions that motivate our work are as follows:

- **RQ1 — High Level Abstraction** How high can we raise the level of abstraction for developing stream processor applications on distributed embedded systems? What performance penalty, due to automatic code generation, is incurred as the level of abstraction is increased? Can any performance penalty be mitigated in some way?
- **RQ2 — Multiple Interpretations** Considering the heterogeneity aspect of distributed embedded systems, how can we generate code for different architectures? Can the stream processor be also interpreted in a simulation interpretation at the high level of abstraction to observe the behaviour in a simulated environment and thereby simplifying the test and debug cycle?
- **RQ3 — Network Sharing** The cost of setting up and maintaining a sensor network is often a barrier to deploying new applications. Existing solutions for sharing a sensor network focus on providing a run-time system to manage and switch context between different applications. Can we use the high level descriptions to identify ways of fusing together statically at compile time to share the same sensor network infrastructure?
- **RQ4 — Compilation Hints** How can the language embedding be enhanced to take hints from the programmer that influence how the compiler generates more efficient code based on how the application is going to be used in a real environment? How can the hint model be generalised to apply to different contexts?

3 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 typically 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.

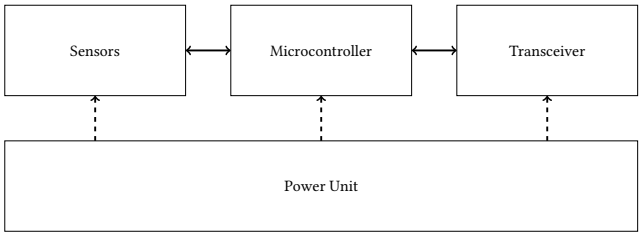

Figure 1: The structure of a wireless sensor node

Several approaches have been proposed for programming wireless sensor nodes. 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 [19].

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.

A number of operating systems have emerged in the sensor network community with TinyOS [12] and Contiki [6] being by far the most popular operating systems for WSNs.

Maté [10] and ASVM [11] are interpreter-based virtual machines that run on top of TinyOS. The main aim of application specific virtual machines is to reduce the amount of data that needs to be transferred to reprogram the nodes.

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.

TinyDB [13] and Cougar [20] 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

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 [9] and Kairos [7] are imperative sequential languages where the programmer is provided with a centralised view of the sensor network. COSMOS [1] 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.

Wavescript [17], Regiment [16] and Flask [14] are macroprogramming functional languages. Wavescript is a domain specific language for stream processing applications with focus on asynchronous data streams. Regiment, a Haskell-like language, is designed for spatiotemporal macroprogramming sensor networks that translates a global program into node-level event-driven code. Regiment provides constructs for aggregating streams, defining and manipulating regions. Flask is a stream processing DSL embedded in Haskell — it 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 node-level functions to be defined in Red, a (partially) functional language, or directly in nesC using quasiquoting [3].

4 RESEARCH APPROACH

Our work is focused on creating a macro-programming model using an embedded domain specific language approach that allows us to generate an internal representation that can be analysed, transformed and interpreted in different ways — both as simulated or compiled code for different target architectures (See Figure 2). In contrast with Flask, our focus is on the high-level abstraction features, which hide away from the programmer low-level details such as managing radio, power and communication between devices. However, we also want to give the ability to the programmer to provide hints to the compiler, such that the compiled code can be more efficient based on the programmer's knowledge.

Firstly, we create and embed in Haskell a DSL that allows us to describe the behaviour of a stream processing application. The stream processor description generates an internal representation such that interactions between devices are identified and node-level code can be generated. A two-stage process allows us to generate code for different architectures in the second stage. The same framework also allows us to simulate the behaviour of the application without the need to generate and upload code on the devices — effectively speeding up the test and deploy cycle.

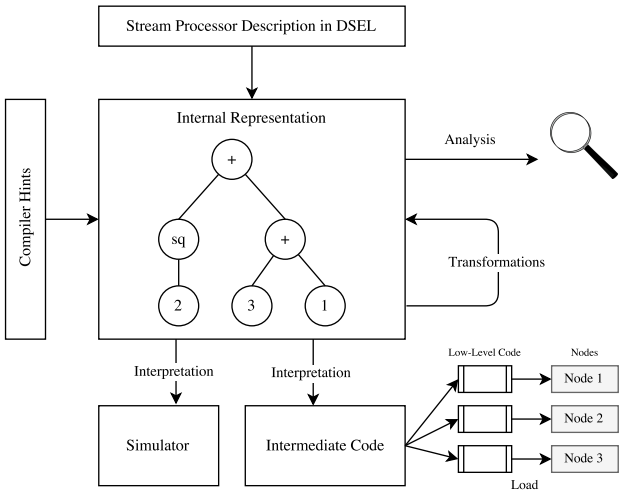

Figure 2: A high-level overview of the D'Artagnan Framework.

Secondly, we want to enhance the framework with a generic hint capability, such that the programmer can influence the transformations and interpretations according to his/her own knowledge. In the absence of hints, the system can still generate good code. However, the programmer's input can lead to systems better suited for the application at hand. For example, the programmer may indicate preference to use a specific node in the WSN more than others due to a renewable energy supply. The compiler can then generate code with this information and combine it with other application restrictions and network information.

Thirdly, through the ability to transform the internal representation, we want to find ways to statically fuse several stream processing applications together at compile-time to run on the same network of devices. The ability to share networks of devices is considered a key enabler for more applications to be developed and deployed as it helps to bring down the cost of ownership. The approach taken by existing solutions is generally to have a run-time environment capable of dynamically loading/unloading applications and context switching between one application and another.

5 CURRENT STATUS

In this section we describe the current status of ongoing research and how the research questions have been addressed so far.

High Level Abstraction To answer the first question **RQ1**, apart from reviewing existing literature, we developed a prototype framework that will serve as the foundation for our work. We embedded our DSL in Haskell and used it to implement a case-study for an *intelligent and energy efficient building cooling and lighting system* which given a layout of a building automatically generates custom code to run on the sensor nodes present in the building. Results have shown that an application written with our DSL that is capable of generating code for any building layout can

be written in 10 lines of code. This is compared to over 500 lines of hand-written low-level C code that is only suitable for a specific layout. We have also evaluated the performance of the automatically generated code in a processing intensive application in comparison to hand-written code with the increase in size and performance being negligible (less than 0.5%), since standard compiler optimisations iron away the overheads.

Multiple Interpretations At a top-level, the framework supports two main interpretations (**RQ2**). The first is a simulation interpretation, such that the behaviour of the stream processing application can be observed in a simulated environment, where sensor readings can be supplied and results analysed. The second type is a two-stage code interpretation — the first stage is intermediate code (an abstraction of C), and the second stage is target-specific code. This approach allows us to generate code for different types of devices. In our work so far, we have generated code for two types of devices, both running the Contiki operating system. The model can be easily extended to support other devices and operating systems.

6 CONCLUSIONS

This paper proposes a framework intended to raise the level of abstraction for programming distributed embedded systems by using techniques from the domain of embedded languages. The framework is capable of generating code for different architectures to run on a network of heterogeneous devices, and the behaviour of a high-level stream processor description can be observed in a simulated interpretation. So far, we have answered almost half of the research questions, therefore in the coming months we will continue along two directions: fusing applications statically (**RQ3**) and compiler hints (**RQ4**). The framework will be enhanced to fuse together different stream processing applications at compile time by transforming and combining the internal representations. Also, we intend to create a generic hint model that allows the programmer to use his/her expertise and knowledge to influence the compiler to generate optimised and efficient code.

7 ACKNOWLEDGMENTS

The research work disclosed in this publication is partially funded by the ENDEAVOUR Scholarships Scheme. "The scholarship may be part-financed by the European Union — European Social Fund".

REFERENCES

[1] Asad Awan, Suresh Jagannathan, and Ananth Grama. 2007. Macroprogramming heterogeneous sensor networks using cosmos. In *ACM SIGOPS Operating Systems Review*, Vol. 41. ACM, 159–172.

[2] E. Axelsson, K. Claessen, G. Dévai, Z. Horváth, K. Keijzer, B. Lyckegård, A. Persson, M. Sheeran, J. Svenningsson, and A. Vajdax. 2010. 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.* https://doi.org/10.1109/MEMCOD.2010.5558637

[3] Alan Bawden et al. 1999. Quasiquotation in Lisp.. In *PEPM*. Citeseer, 4–12.

[4] Per Bjesse, Koen Claessen, Mary Sheeran, and Satnam Singh. 1998. Lava: Hardware Design in Haskell. In *Proceedings of the Third ACM SIGPLAN International Conference on Functional Programming (ICFP '98)*. ACM, New York, NY, USA, 174–184. https://doi.org/10.1145/289423.289440

[5] Koen Claessen and Gordon J. Pace. 2002. An Embedded Language Framework for Hardware Compilation. In *Designing Correct Circuits '02, Grenoble, France.*

[6] Adam Dunkels, Bjorn Gronvall, and Thiemo Voigt. 2004. Contiki-a lightweight and flexible operating system for tiny networked sensors. In *Local Computer Networks, 2004. 29th Annual IEEE International Conference on.* IEEE, 455–462.

[7] Ramakrishna Gummadi, Omprakash Gnawali, and Ramesh Govindan. 2005. Macro-programming wireless sensor networks using Kairos. In *International Conference on Distributed Computing in Sensor Systems*. Springer, 126–140.

[8] P. Hudak. 1998. Modular domain specific languages and tools. In *Software Reuse, 1998. Proceedings. Fifth International Conference on.* 134–142. https://doi.org/10.1109/ICSR.1998.685738

[9] Nupur Kothari, Ramakrishna Gummadi, Todd Millstein, and Ramesh Govindan. 2007. Reliable and efficient programming abstractions for wireless sensor networks. In *ACM SIGPLAN Notices*, Vol. 42. ACM, 200–210.

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

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

[12] Philip Levis, Sam Madden, Joseph Polastre, Robert Szewczyk, Kamin Whitehouse, Alec Woo, David Gay, Jason Hill, Matt Welsh, Eric Brewer, et al. 2005. Tinyos: An operating system for sensor networks. In *Ambient intelligence*. Springer, 115–148.

[13] Samuel R Madden, Michael J Franklin, Joseph M Hellerstein, and Wei Hong. 2005. TinyDB: an acquisitional query processing system for sensor networks. *ACM Transactions on database systems (TODS)* 30, 1 (2005), 122–173.

[14] Geoffrey Mainland, Greg Morrisett, and Matt Welsh. 2008. Flask: Staged functional programming for sensor networks. In *ACM Sigplan Notices*, Vol. 43. ACM, 335–346.

[15] Marjan Mernik, Jan Heering, and Anthony M Sloane. 2005. When and how to develop domain-specific languages. *ACM computing surveys (CSUR)* 37, 4 (2005), 316–344.

[16] Ryan Newton, Greg Morrisett, and Matt Welsh. 2007. The Regiment Macroprogramming System. In *Proceedings of the 6th International Conference on Information Processing in Sensor Networks (IPSN '07)*. ACM, New York, NY, USA, 489–498. https://doi.org/10.1145/1236360.1236422

[17] Ryan R. Newton, Lewis D. Girod, Michael B. Craig, Samuel R. Madden, and John Gregory Morrisett. 2008. Design and Evaluation of a Compiler for Embedded Stream Programs. *SIGPLAN Not.* 43, 7 (June 2008), 131–140. https://doi.org/10.1145/1379023.1375675

[18] Gregory J Pottie and William J Kaiser. 2000. Wireless integrated network sensors. *Commun. ACM* 43, 5 (2000), 51–58.

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

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