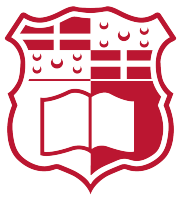

Macroprogramming using an Embedded DSL approach

ADRIAN MIZZI

Supervised by
Dr Joshua Ellul and Prof. Gordon Pace

Department of Computer Science
Faculty of ICT
University of Malta

September, 2019

A dissertation submitted in partial fulfilment of the requirements for the degree of Ph.D. Computer Science.

Statement of Originality

I, the undersigned, declare that this is my own work unless where otherwise acknowledged and referenced.

Candidate Adrian Mizzi

Signed _____

Date April 26, 2020

The research work disclosed in this publication is partially funded by the Endeavour Scholarship Scheme (Malta). Scholarships are part-financed by the European Union - European Social Fund (ESF) - Operational Programme II – Cohesion Policy 2014-2020 "*Investing in human capital to create more opportunities and promote the well-being of society*".

European Union – European Structural and Investment Funds
Operational Programme II – Cohesion Policy 2014-2020
"Investing in human capital to create more opportunities and promote the well-being of society"
Scholarships are part-financed by the European Union - European Social Funds (ESF)
Co-financing rate: 80% EU Funds;20% National Funds

Acknowledgements

I would sincerely like to thank my supervisors, Dr Joshua Ellul and Prof. Gordon Pace for their patience, commitment and support throughout the duration of my studies. Apart from the part they played in introducing me to the area of Internet of Things, functional programming and blockchain technology, they were also available to provide guidance and steer me in the right direction when things became unclear. They pushed me beyond my limits, and in doing so I feel I have grown in my capabilities as a researcher and as a computer scientist.

I also thank my close friend Dr Jean Paul Ebejer, who was not only an inspiration for starting this journey, but also the person whom I turned to many times for guidance and support. I am indebted to him for being there during the difficult times that come with such programmes, but also for taking personal time and interest to discuss, review and suggest ideas around my research work.

I must also thank my immediate family for the unwavering support they have provided through the duration of my work. This work would not have been possible without the support from my wife Ruth, who not only returned back to employment to support the family but also spent many hours with our daughter Michaela, so I could focus on my studies. I also thank my parents and in-laws for the long hours they dedicated to our daughter so I could complete my research.

Abstract

Software applications were traditionally developed using a monolithic approach and developed as a single instance. As distributed systems emerged, these traditional methods were no longer suitable. In the domain of wireless sensor networks, an application is developed to run across multiple nodes, and devices must communicate and collaborate together. The general trend is for a software developer to write a single program which is loaded on all the devices. In the case of heterogeneous networks where the systems making up the network vary in architecture, capabilities and characteristics a different approach is used — different programs are written and loaded on each different system. Such an approach requires expertise programming different systems, and the interactions between disparate systems need to be explicitly handled by the programmer.

In this thesis, we propose a model for programming heterogeneous systems using a single macroprogram, thereby achieving a higher level of abstraction and enabling applications to be described at the macro-level. We combine techniques from macroprogramming and multi-target compilation, using an embedded DSL approach to generate target-specific code for different domains on different ends of the spectrum. On one end of the spectrum, we apply the model to wireless sensor networks where challenges exist around optimising code for execution on heavily resource constrained devices. At the other end of the spectrum, we propose a framework for writing smart contracts spanning multiple diverse blockchain systems. Each domain brings its' own challenges, however the model is shown to be applicable to different domains.

Contents

1

Introduction

The first computers were programmed using low-level machine code, but it did not take long for scientists to realise that this approach was very error-prone and requiring expert programming skills. To address this and make programming more accessible to programmers, a higher level of abstraction was needed — a number of higher-level languages soon emerged making it easier for programmers to write software applications.

As software applications grew in complexity and spanned across multiple systems, software programs were split in smaller parts, where each part gets executed on a different system. A typical scenario is that of a client-server application, where the code of the client is written separately from that of the server, and both parts need to tackle the communication between them (see Figure 1.1(a)). Writing applications in this fashion is not straightforward, and the programmer needs to be fully aware of the implications of distributed systems — such as synchronisation of processes, communication between systems and handling of partial failures. Such an approach is still commonly used today in many domains, primarily because alternative approaches are lacking. A possible solution lies in achieving an even higher level of abstraction such that the complexity of distributed applications is hidden away from the programmer.

A technique that has emerged from the domain of wireless sensor networks is *macroprogramming* — a single program is written to define the be-

haviour of tens, hundreds or even thousands of small, heavily resource-constrained devices. In this domain, several approaches have been proposed. In one approach (see Figure 1.1(b)), a software program is written from the perspective of the device and this is then uploaded to all the devices in the wireless sensor network. The applications that can be implemented in this manner are somewhat restricted to those where all devices in the system behave in, more or less, the same way — for example, applications where sensors detect environmental data (such as devices capable of sensing seismic activity around a volcano) and forward the information to a central server.

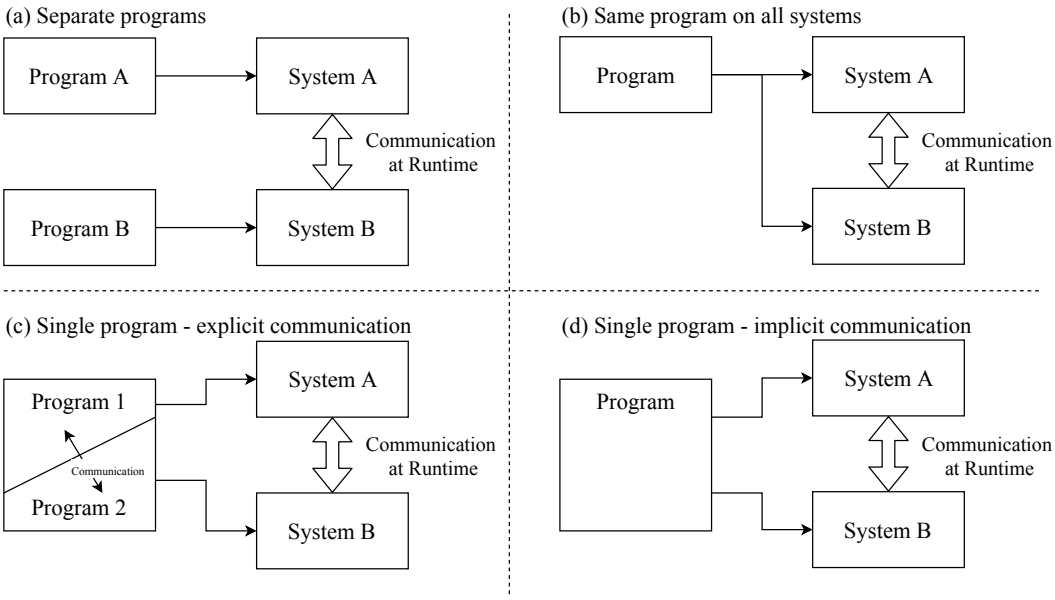

Figure 1.1: Programming models

Another macroprogramming approach that has been proposed, and which can be applied to a wider range of applications, is one where the program is written from a network-perspective and the behaviour of each system is determined in a centralised manner (refer to Figure 1.1(c)). In this approach, the communication between different systems is explicitly defined by the programmer, and communication-handling code is then automatically generated for the devices. Although such an approach removes the need for the

programmer to write the actual code which handles communication, there is still the need for the programmer to determine how application logic is placed on which system as well as define when and how interaction between systems is made.

A potentially better solution lies in extending the macroprogramming network-perspective approach in such a way that the communication between devices does not need to be explicitly defined by the programmer. By using a higher level of abstraction the programmer does not need to specify which device must perform which action, but rather, the function of splitting application logic across different systems and the communication between them is handled under the bonnet.

A number of challenges exist with such an approach — for example, determining how application logic is split and placed on the different available systems is not straightforward. If certain application logic makes use of functionality which is only available or possible on a specific system, then that logic must be bound to execute on that system — putting the logic on a different system would clearly cause the application to fail. Also, other application logic may exist which is not bound to a single system and this logic can be freely placed in different locations. In such situations, other factors may influence where the application logic is placed — for example, the cost of computation or the performance of execution are important factors that influence a placement strategy.

1.1 Aims and Achievements

In the domain of wireless sensor networks, macroprogramming is an effective technique for programming a network of wireless sensor nodes where the devices are, in general, of the same type (homogeneous). Existing approaches rely on the programmer to define how the interactions between devices are made. Whether this technique can be extended further to reduce, or remove, the dependency on the programmer to provide placement and communication directives remains largely unexplored. Also, whether

such a technique can be successfully applied to heterogeneous systems, both in the domain of wireless sensor networks and also beyond, is still largely unknown. Our aim is to explore how macroprogramming can be taken further:

- One of the drawbacks of existing macroprogramming techniques is the general assumption that the systems onto which an application is implemented are of the same type. Aside from specific wireless sensor network applications, this is rarely the case and probably the main reason why the technique has not been widely adopted in other domains. An aim of this work is to extend the macroprogramming approach for heterogeneous systems such that the technique can be applied to new and different domains.
- In existing macroprogramming techniques, the programmer determines which application logic is placed on which device, and how the interaction between the different systems should be done. This is not straightforward, and the programmer must have a good understanding of the underlying systems and the functionality available, as well as how and when these need to communicate with each other. Having a solution which removes the need for the programmer to give placement and communication directives is preferred as it simplifies the writing of such programs.

To address the above aims, we propose techniques for defining a multi-system application using a single program description without the need for explicit information about placement or communication. For heterogeneous systems, we propose a two-stage compilation process where code is first generated into existing high-level languages using multi-target compilation techniques, and then these generated programs are further compiled down to the target systems using standard compilers. To write programs which span multiple systems, we identify the need to limit the application domain and create domain specific languages. Our main contributions are given below:

Macroprogramming We propose a technique where a single software program generates an intermediate representation which can be analysed, transformed and used for different interpretations, including simulation and the generation of code for different architectures. A framework takes as input a single program written by a programmer which determines the behaviour of a number of systems. This program generates an internal representation which can be analysed by the framework itself — for example, to calculate the cost or duration of executing a function on a specific system. The framework can transform the internal representation to place logic on different systems, and together with the analysis capability, determine an optimal placement strategy. Finally, once a placement strategy has been chosen, the application logic generates custom target code for each device. We also show that such an approach does not come at the expense of excessive computation power.

Stream processing domain specific language To illustrate the technique of macroprogramming, we design and implement a domain specific language for macroprogramming stream-processing applications on heterogeneous wireless sensor networks. We show how the language and framework are expressive enough to successfully implement a range of stream-processing applications — applications where data is continuously flowing through the system as data is sensed on the sensors and processed inside the network. We also evaluate how generated code compares on performance with hand-written code.

Commitment-based smart contract language To further illustrate the versatility of the technique and framework, we also create a language for writing smart contracts which span across multiple blockchain systems. We define a simple, yet expressive, language for commitment-based smart contracts where the interactions between participants and smart contracts are limited to commitments.

To validate the above contributions, we have developed a number of use-cases across very different domains:

Intelligent Cooling and Lighting System The first use-case is set in the domain of wireless sensor networks where a single-program can be used to generate code for any building layout such that cooling and lighting systems are turned on and off according to motion detectors. The use-case has been successfully implemented and as part of the evaluation, the performance of the resulting code has been compared to a hand-written version.

Smart Rent Management system The second use-case extends the language used in the first case study to go beyond the boundaries of wireless sensor networks and into blockchain systems. The use case shows how a single-description program can be split to run across wireless sensor nodes, edge devices and a blockchain system to calculate consumption of electricity and usage of appliances.

Property Sale This use case shifts the domain focus to blockchain systems where the proposed commitment-based smart contract language is used to implement a property sale involving three participants (buyer, seller and notary) and three assets (currency, property and notary vote). The case study is fully implemented to show how the proposed framework can be used in a completely different setting with blockchain systems and using a commitment-based smart contract model.

Single-Shot DAO We show how a single-shot decentralised autonomous organisation (DAO) can be implemented using the commitment-based smart contract language and the proposed framework. This use-case shows how the use of a macroprogramming language, the framework and the underlying network of interconnected blockchain systems, a DAO can grow beyond the boundaries of a single blockchain system as participants can interact via different blockchain systems.

1.2 Overview of Subsequent Chapters

The thesis has been divided into three main parts. The first part provides background, and the remaining parts deal with macroprogramming across different domains.

Part I This part mainly serves to set the scene and provide the background necessary in later parts.

Chapter 2 gives an overview on macroprogramming, how it originated and what the main challenges are.

Chapter 3 describes the technique of embedding a domain specific language in a host language. It starts by describing the different embedding techniques, and then it is followed by a description of the respective challenges.

Part II In this part we focus on macroprogramming in the domain of wireless sensor networks.

Chapter 4 provides background on the domain of wireless sensor networks, starting with a description of the main characteristics of a wireless sensor node. This is followed by an overview of the key challenges in this domain, to motivate the use of macroprogramming for wireless sensor networks. Finally, a variety of existing programming techniques is described to give the reader a broader understanding of the various possible approaches.

Chapter 5 describes an embedded domain specific language framework for macroprogramming heterogeneous wireless sensor networks. The language is ideal for stream-processing applications — a number of simple examples are defined to illustrate the use of the language. Finally, a couple of use-cases for smart buildings are used to illustrate the effectiveness of the language at a higher level of abstraction.

Part III In this part, we shift focus on to the domain of blockchain technology and smart contracts.

Chapter 6 provides background on blockchain technology and smart contracts. It starts with an overview of a number of different blockchain systems and the key characteristics of the smart contract languages used on these platforms.

Chapter 7 extends the heterogeneity aspect of the framework from Chapter 5 to reach beyond the domain of wireless sensor networks by including blockchain and edge systems.

Chapter 8 introduces a new embedded domain specific language and framework for describing commitment-based smart contracts that span multiple blockchain systems. Two use-cases (a property sale and a single-shot decentralised autonomous organisation) with assets residing on different blockchain platforms, is used to illustrate the effectiveness of the language.

Lastly, the ninth and final chapter gives a short summary and critical analysis of the main contributions given in this thesis and investigates possible extensions which may prove to be fruitful.

Part I

Background

This part of the thesis sets the scene for the rest of the work by introducing the background and techniques used. Chapter 2 introduces the notion of macroprogramming. Chapter 3 describes the techniques of embedding domain specific languages.

2

Macroprogramming

2.1 Introduction

In the early days of computing, the first computers had limitations in computational capability and memory capacity so programmers wrote hand-tuned machine code to program them. It did not take long for scientists to realise that it required a significant amount of intellectual effort to write programs in this way, so a number of higher-level programming languages were soon born. These languages were easier for programmers to reason with because they were at a higher level of abstraction. The software programs were then compiled down to the low-level machine code to be executed on a computer system.

As software applications grew in complexity, it became increasingly common for applications to grow beyond the boundaries of a single-system and to make use of several systems to achieve an application goal. This may have been needed for a number of possible different reasons — the need for more computational power than is available on a single system; the need for specialised functionality only available on specific systems; or simply due to the nature of the application which may require systems to be physically placed in different locations.

Developing such multi-system applications meant writing several software programs (one for each system, in possibly different languages) and

the communication between the systems must be explicitly defined by the programmer — communications channels must be managed, messages are sent and received over the network, synchronisation between systems must be handled and so on. This approach significantly increases the level of complexity and is more error-prone — programmers of these systems must have expert programming skills.

To alleviate and address this problem, a new technique emerged in the nineties in the domain of hardware/software codesign — hardware and software components of electronic systems were designed and described together using the same language. The use of a common language for both hardware and software components meant that a single programmer was able to fulfil the role of both a hardware engineer and a software architect with the compiler handling the translations to the different components [Page, 1996].

A decade later, as the domain of wireless sensor networks gained interest, the term macroprogramming was coined by Newton and Welsh [Newton and Welsh, 2004] to describe a technique to write a single program for a network of tens, hundreds or even thousands of connected devices. Figure 2.1 illustrates the concept of writing a single program (macroprogram) to generate programs in a higher level language, which are then compiled down to machine code to be executed directly on systems.

Throughout this thesis, we use the term *node* to refer to a single individual component of a set of systems being programmed and the term *network* to refer to the ensemble of the systems. Depending on the domain, a node may refer to a wireless sensor device in wireless sensor networks or the hardware component in the domain of chip design. A node is defined as an entity which has computational capability and which may communicate with other nodes to exchange information as part of an application.

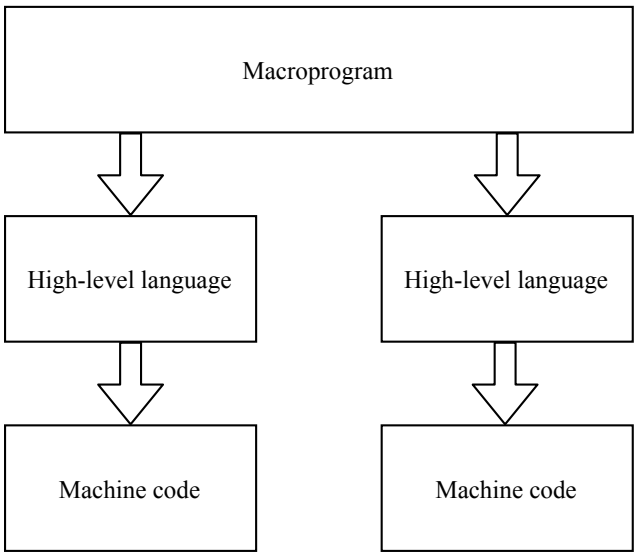

Figure 2.1: Macroprogramming of systems

2.1.1 Hardware/Software Codesign

Hardware/software codesign surfaced in the early 1990s for the concurrent design of hardware and software components of electronic systems. In this discipline, the optimisation and synergy of hardware and software components is sought. The goal is to optimise on constraints such as cost, performance and power utilisation. Codesign refers to the coordination between interdisciplinary design teams — firmware, operating systems and application developers on the software side; chip designers on the hardware side.

Early work in two projects sought to address the challenges from two complementary approaches. In the Vulcan project [De Michell and Gupta, 1997], solution design started from a hardware-only approach and tasks were migrated to software to reduce costs as long as performance constraints were satisfied. On the other hand, the Cosyma system [Henkel and Ernst, 1997], which was developed over the same period as Vulcan, started from a software-only approach and tasks were migrated to hardware to meet performance requirements. One of the major shortcomings of these early projects, was that both assumed that implementation was single-threaded. Nevertheless, this early work inspired others to investigate new

approaches to design and partition across hardware and software components.

To address these requirements, researchers looked into ways to raise the level of abstraction at which system designers specify their systems. Languages for defining both the hardware and software parts were needed. Hardware description languages (HDLs) were suitable for the hardware component, whereas software designers were more used to programming languages such as C and C++ instead. Handel [Page, 1996] attempted to bridge the gap between the two parts, and provided ways to deal with the concurrency element. Other languages, such as SystemC [Grötker et al., 2007] and SpecC [Dömer et al., 2008], emerged as extensions of C and C++ to address both the hardware and software elements.

Handel-C

Handel-C is a high-level language designed to compile programs into synchronous hardware and software components. Handel-C has C-like syntax and is inspired by CSP and Occam. It is ideal for software engineers, with no or limited hardware background, to implement and translate an algorithm into hardware and software components. The different components (hardware and software) are written separately by the programmer using the same language, and a netlist is generated for the hardware component with which an FPGA can be programmed.

The hardware and software components of a simple Handel-C program [Page, 1996] are shown in Listings 2.1 and 2.2.

Listing 2.1: Handel-C program for hardware implementation

```
PROC main_hw (
  CPC_Tputer.PAIR.IN   : CHAN OF INT16 FromT805,
  CPC_Tputer.PAIR.OUT  : CHAN OF INT16 ToT805,
  CPC_SRam.RAM.ADDR    : CHAN OF INT15 SRH_Addr,
  CPC_SRam.RAM.IN      : CHAN OF INT16 SRH_Din,
  CPC_SRam.RAM.OUT     : CHAN OF INT16 SRH_Dout)
  INT16 a, d:
  SEQ
```

```
    UNTIL a = 32768(INT16)
      PAR
        SRH_Addr ! a <- 15
        SRH_Dout ! 0 (INT16) - a
        a := a + 1(INT16)
    WHILE TRUE
      SEQ
        FromT805 ? a
        PAR
          SRH_Addr ! a <- 15
          SRH_Din ? d
        ToT805 ! d
:
```

The example shown is a trivial one, with little usefulness other than to show the reader how the structure of a program is made. The hardware part of the system shown in Listing 2.1 first fills the RAM with numbers, and then listens for requests from the software process (the T805 microprocessor) to lookup and return a value to the software component. The hardware and software components communicate over channels and follow the occam model.

Listing 2.2: Handel-C program for software implementation

```
PROC main_sw()
  INT s :
  INT16 r :
  SEQ
    s := 0(INT)
    WHILE s <= 32768(INT)
      SEQ
        DPGA.write (INT s)
        DPGA.read.16 (r)
        so.write.string (fs, ts, "*n result = ")
        so.write.int (fs, ts, INT s, 12)
        so.write.int (fs, ts, INT r, 12)
        s := s + 1 (INT)
:
```

The software component of the example is shown in Listing 2.2, where a loop iterates through all the RAM locations and requests the value from the hardware component.

The application is constructed by combining the hardware and software components. The programmer is explicitly defining which part of the algorithm should be placed on the hardware and which on the software using the same programming language. The top-level structure, with suppressed details, is shown here:

```
PAR
  Hardware_Process (in_chan, out_chan)
  Software_Process (out_chan, in_chan)
```

The advantage of using a single framework to define both the hardware and software components is that less specialised skills (for either software or hardware) are required. Hardware designers need to become more literate in programming, and software programmers need to become more familiar with parallelism concepts.

2.1.2 Wireless Sensor Networks

The term macroprogramming emerged in the domain of Wireless Sensor Networks (WSNs) and refers to the technique of programming sensor networks as a whole, rather than at the individual node level. Figure 2.2 shows a taxonomy of the different programming models for WSNs — these are divided between node-level and network-level models. In the domain of hardware/software codesign (as described in the previous section) programmers had to deal with two components — the hardware and software components. In WSNs, it is normal to have tens, hundreds and even thousands of devices, so the challenge of writing a single program for the network of nodes is even more desirable, but also challenging.

In this section, we will focus on the network-level programming model and more specifically on macroprogramming.

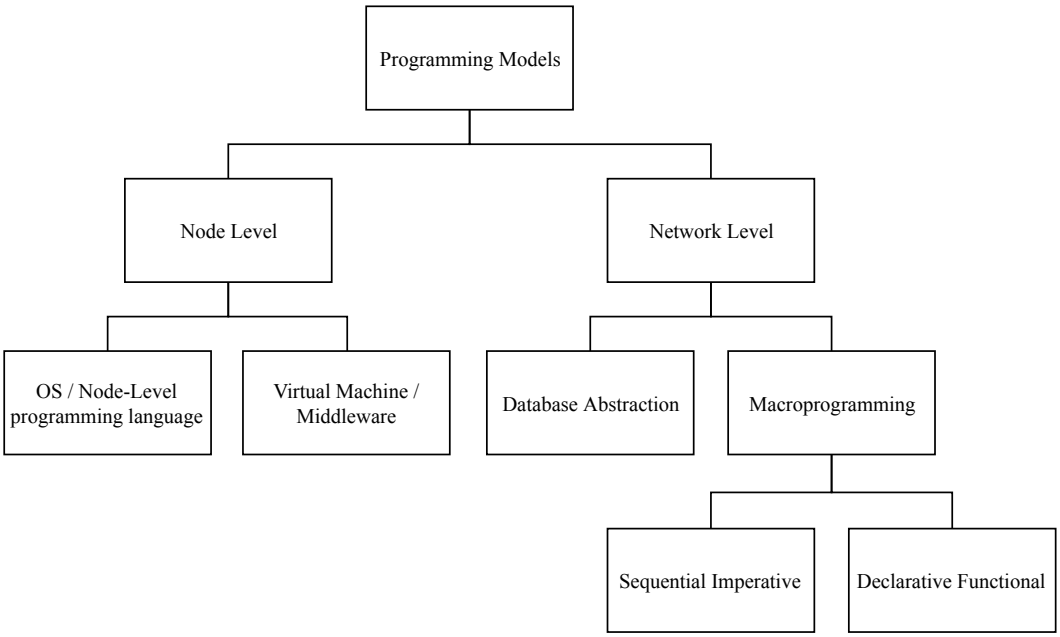

Figure 2.2: A taxonomy of programming models for wireless sensor networks

Network-level programming of WSNs can be classified into two main categories — the first is a database abstraction and the second is a macroprogramming technique. In a database abstraction, the network of nodes is viewed in a similar manner as one would view and query a database for information. 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. The second approach is to use a macroprogramming language which provides a global view of the network and more flexibility, than the database abstraction, for a wider variety of applications. Macroprogramming languages allow for a range of applications where data may flow between one node and another, and processed inside the network.

Macroprogramming languages can be further classified into two broad categories. The first category is for sequential imperative programming models, which includes the languages Pleiades [Kothari et al., 2007], Kairos [Gummadi et al., 2005] and COSMOS [Awan et al., 2007]. Pleiades and

Kairos are very similar in that they allow the programmer to have a centralised view of the sensor network and programming is done by giving instructions in a step-by-step fashion. On the other hand, COSMOS allows functional components to be wired together to determine the data flow. The resulting program is then distributed across the network as the runtime environment instantiates the functional components on the devices. The second category is a declarative functional approach and includes the languages Regiment [Newton et al., 2007b], Flask [Mainland et al., 2008] and Wavescript [Newton et al., 2008]. The programmer describes the application at a high level of abstraction, and the compiler breaks it down into node low-level behaviour — hiding away the details of intra-node communication or how computation is done across the sensor network. This approach provides the highest level of abstraction, while supporting the implementation of a wide variety of applications.

A macroprogram typically involves a two-stage compilation process — the macroprogram is first compiled into node-level code, and then this is further compiled into object code to be loaded and executed onto the individual nodes.

Macroprogramming Languages

In this section we shall have a closer look at some of the macroprogramming languages in the domain of wireless sensor networks.

Pleiades [Kothari et al., 2007] provides the programmer with a centralised view of the sensor network. Implemented as an extension of the C language, new constructs allows the programmer to address nodes in the network and to access local state on the individual nodes. The higher level of abstraction removes complications of inter-node communication and node-level resource management. 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 *nodecuts* —

a unit of work that can be executed on a single node. The code is then translated into nesC programs that are executed on the TinyOS system.

The sample code below, taken from Kothari et al. [2007] is an excerpt from a street parking application written in Pleiades. The `cfor` construct is used to run code in parallel over a set of nodes.

```
while(!reserved && !empty(nToExamine)){
    cfor(nodeIter=get_first(nToExamine);nodeIter!=NULL;
            nodeIter = get_next(nToExamine)){
        neighbors@nodeIter=get_neighbors(nodeIter);
        for(neighborIter@nodeIter=get_first(neighbors@nodeIter);
            neighborIter@nodeIter!=NULL;
            neighborIter@nodeIter=get_next(neighbors@nodeIter)) {
                if(!member(neighborIter@nodeIter,nExamined))
                    add_node(neighborIter@nodeIter,nToExamine);
            }

        if(isfree@nodeIter){
            if(!reserved){
                reserved=TRUE; reservedNode=nodeIter;
                isfree@nodeIter=FALSE;
                break;
            }
        }

        remove_node(nodeIter,nToExamine);
        add_node(nodeIter,nExamined);
    }
}
```

Kairos [Gummadi et al., 2005] uses a centralised approach for specifying the global behaviour of a sensor network. In a similar manner to Pleiades, Kairos provides the programmer with constructs to access local state on the nodes and to address arbitrary nodes. Kairos also provides the programmer with the ability to iterate through the node's one-hop neighbours. Using these constructs, the programmer can im-

plicitly express the distributed data flow and distributed control flow. Code generation is implemented as a preprocessor add-on to the compiler that generates the binary code. The Kairos model is similar to a shared-memory based parallel programming model. Shared node state is kept consistent across the different nodes using a message-passing approach. Kairos is implemented as an extension of Python.

The following code shows a complete Kairos program for building a routing tree from a given root node. The program is written in a centralised fashion, and the use of the function `get_neighbours` acquires the one-hop neighbours at each node for the construction of the routing tree.

```
void buildtree(node root)
  node parent, self;
  unsigned short dist_from_root;
  node_list neighboring_nodes, full_node_set;
  unsigned int sleep_interval=1000;
  //Initialization
  full_node_set=get_available_nodes();
  for (node temp=get_first(full_node_set); temp!=NULL;
        temp=get_next(full_node_set))
    self=get_local_node_id();
    if (temp==root)
      dist_from_root=0; parent=self;
    else dist_from_root=INF;
    neighboring_nodes=create_node_list(get_neighbors(temp));
  full_node_set=get_available_nodes();
  for (node iter1=get_first(full_node_set); iter1!=NULL;
        iter1=get_next(full_node_set))
    for(;;) //Event Loop
    sleep(sleep_interval);
    for (node iter2=get_first(neighboring_nodes); iter2!=NULL;
          iter2=get_next(neighboring_nodes))
      if (dist_from_root@iter2+1<dist_from_root)
        dist_from_root=dist_from_root@iter2+1;
        parent=iter2;
```

Wavescript [Newton et al., 2008] is a domain specific language for stream processing applications with focus on asynchronous data streams. It is a functional language with support for type inference, polymorphism and higher-order functions. Wavescript uses three implementation techniques. First, a Wavescript program is partially evaluated into stream dataflow graphs. Second, it uses profile-driven compilation to enable optimisations and finally includes an extensible system for rewrite rules to capture and optimise algebraic properties in specific domains. The code below is an excerpt from a case study for localizing yellow-bellied marmots taken from Newton et al. [2008].

The same code runs on every node in the network. Each node is equipped with four sound sensors aimed at different directions for calculating the direction of arrival (DOA) of the sound of a marmot.

```
// Node-local streams, run on every node:
NODE "*" {
  (ch1,ch2,ch3,ch4) = ENSBoxAudioAllChans(44100);
  // Perform event detection on ch1 only:
  scores :: Stream Float;
  scores = marmotScores(ch1);
  events :: Stream (Time, Time, Bool);
  events = temporalDetector(scores);
  // Use events to select audio segments from all:
  detections = syncProject(events, [ch1,ch2,ch3,ch4]);
  // In this config, perform DOA computation on the ENSBox:
  doas = DOA(detections);
}
```

Information is then forwarded to the server where data from all nodes is fused together to determine the likely location of the marmots.

```
SERVER {
  // Once on the base station, we fuse DOAs:
  clusters = temporalCluster(doas);
  grid = fuseDOAs(clusters);
  // We return these likelihood maps to the user:
  RETURN grid
```

```
}
```

Regiment [Newton et al., 2007b] is a high-level functional language designed for spatiotemporal macroprogramming sensor networks, that translates global programs 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, and defining and manipulating regions. Compilation changes the program into an intermediate language called *token machines* which provides for local computation, sampling and communication with other nodes. The Regiment syntax is similar to Haskell, and a complete program to calculate the average temperature across a sensor network is shown here:

```
dosum :: float, (float, int) -> (float, int)
fun dosum(temp, (sumtemp, count)) {
    (sumtemp+temp, count+1)
}
tempreg = rmap(fun(nd){sense("temp",nd)}, world);
sumsig = rfold(dosum, (0,0), tempreg);
avgsig = smap( fun((sum,cnt)) {sum / cnt}, sumsig);
BASE <- avgsig
```

The example above illustrates a complete Regiment program. The program starts with a number of function and variable bindings, followed by a single line of the form "`BASE <- <expression>`". The last line in the program determines what information will be returned to the base station from the sensor network. The program makes use of a number of language operations such as `rmap` (map on regions), `smap` (map on signals) and `rfold` (fold on regions with an initial value and combining function).

Flask [Mainland et al., 2008] is a stream processing DSL embedded in Haskell. Flask provides a staging mechanism that maintains a clean separation between node-level code and the metalanguage used to

generate node-level code fragments. Node-level code can be written in a language called Red, which inherits Haskell syntax but places some restrictions such as disallowing closures and recursive data types to ensure that the code can be translated efficiently into node-level code. Flask places low-level code nesC on equal footing as Red, in that it uses a technique called quasiquoting [Mainland, 2007] to allow the interchangeable use of nesC and Red. Quasiquoting is a technique for the inclusion of code in another language as part of programs written in the host language — in this case, the use of nesC and Red can be included in Haskell programs. Flask uses the term *signals* for streams and provides a set of signal combinators that can be used to merge, filter and aggregate signals. The following is an example program written in Flask that attempts to detect earthquakes using a ratio of two *low-pass* filters approach on a seismometer signal.

```
detect :: Double -> Double -> Double -> S Double -> S ()
detect low high thresh =
  (\sig -> ewma high sig &&& ewma low sig)
  >>>
  map [$exp | \(hi, lo) -> hi / lo > $flo:thresh |]
  >>>
  edge
```

The program takes periodic samples from a seismic sensor and processes the data through two exponentially-weighted moving average (`ewma`) filters with different gain settings. If the ratio of the filters exceeds a threshold, this indicates that the seismic signal is significantly larger than the background noise, and a message is transmitted to the base station. The program shows the use of combinators for splitting and merging signals, and the use of quasiquoting for mapping an expression onto an incoming signal.

COSMOS [Awan et al., 2007] is another architecture for macroprogramming heterogenous sensor networks. COSMOS is made up of a lean operating system called mOS and an associated programming lan-

guage called mPL. A programmer can specify the aggregate system behaviour in terms of distributed dataflow 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. A sample macroprogram for a structural monitoring example is shown below:

```
//logical instances
accel_x   : accel(12);
disp      : disp1, disp2;
cpress_fc : cpress;
thresh_fc : thresh(250);
max_fc    : max;
fft_fc    : fft;
ctrl_fc   : ctrl;

// refining capability constraints
@ on_mote = MCAP_ACCEL_SENSOR : thresh, cpress;
@ on_srv = MCAP_UNIQUE_SERVER : ctrl;

IA {
  timer(30) -> accel;
  accel     -> cpress[0];
  cpress[0] -> thresh[0], max[0];
  thresh[0] -> fft[0];
  fft[0]    -> disp1;
  max[0]    -> ctrl[0], disp2 | max[1];
  ctrl[0]   -> thresh[1];
}
```

Functional components, such as `accel`, `cpress` and `thresh` above are defined in C code elsewhere. The program above creates instances of functional components and the interaction assignments (shown under

IA above) describe the dataflow from one component to another.

2.2 Discussion and Challenges

In this section we discuss the key challenges of macroprogramming. Code slicing determines how application logic is split across the available nodes, whereas interoperability refers to how nodes communicate with each other during runtime. The heterogeneity challenge refers to the difficulties of programming nodes with different architectures and characteristics.

2.2.1 Code Slicing

A declarative macroprogram defines the behaviour of an application at the network-level without specifying how the logic should be mapped across the nodes. This approach raises the level of abstraction and hides away the detail from the programmer.

The mapping of tasks across a set of heterogeneous nodes was shown to be an NP-complete problem in Blickle et al. [1998]. The problem is made up of three parts: (1) selection of the right type of nodes from the available set; (2) mapping from the application description onto the selected nodes; and (3) discovery of different placements and implementations that satisfy a number of constraints. For example, consider a smart building equipped with hundreds of wireless sensors capable of sensing the environment and communicating with others. To build an application which uses the temperature in the kitchen, a device with temperature sensing capabilities and physically located in the kitchen must be used. If that same application uses readings from several rooms to calculate the average temperature in the building, the computation for calculating the average can be placed across the hundreds of devices connected to the network in many different ways. A placement strategy may be used to optimise the energy consumption of the network. This may involve reducing the number of radio messages be-

tween devices, perform calculations in-network, or utilising devices which have access to a permanent or renewable power source.

The example described is intentionally simplistic to easily explain this challenge. Real-world applications have computations which are several orders of magnitude more complex than the 'average temperature' example. A placement strategy takes into consideration the trade-off between speed and accuracy, by using approximation functions to calculate a result quickly, and deferring to a more powerful node for a more accurate result only when this is needed. For example, audio-related applications are well known to make use of computationally intensive functions. In many cases, approximation functions can be used by nodes placed close to the source to detect an event, and only information of interest is sent to be processed centrally (and more accurately) by a more powerful server. In this way, the energy of the nodes will be preserved as much as possible.

2.2.2 Interoperability

The second key challenge in macroprogramming is the need for a runtime communication layer. Nodes in the network need the ability to communicate with each other during runtime. The communication layer depends on the type of network, but can be broadly classified under two categories. The first is a message-passing model, where information is shared between nodes through messages passed over a medium (see Figure 2.3). The communication layer takes different forms, depending on the domain at hand. In hardware/software codesign, the communication between chips and processor is held over the system bus, whereas in wireless sensor networks, nodes communicate with each other over the radio network and messages from one node to another may be either a singe-hop, if within radio signal range, or multi-hop via intermediaries. Traditional software applications may communicate with each other using message queues. At a high level of abstraction, every node is 'assumed' to have connectivity with all other nodes of the network that it needs to interact with.

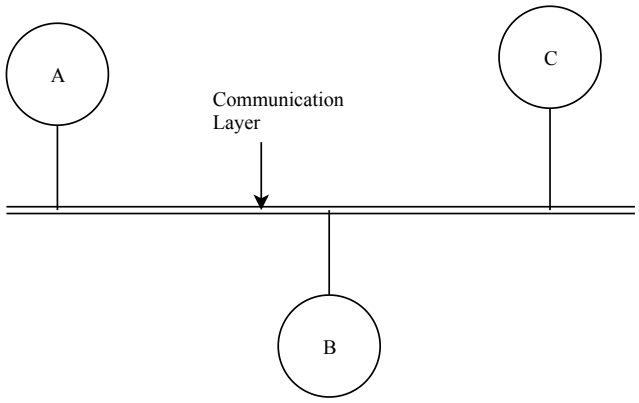

Figure 2.3: Communication layer between nodes

The second model is based on a shared memory approach, where the nodes in the network conceptually share a part of the memory with other nodes. A value, flag or instruction can be stored to the local memory which is then replicated to other nodes. A shared memory approach is a higher level of abstraction, but still implemented using message-passing.

2.2.3 Heterogeneity

Networks of nodes may be classified in two categories: homogeneous and heterogeneous. When writing a macroprogramme for homogeneous networks, it may be possible to have the same generated code used by all the nodes of the network. In a case-study for Wavescript [Newton et al., 2008], the authors investigated the use of a sensor network to localise marmots. The same generated code was loaded on all the devices in the network, giving the devices the ability to use sound sensors to detect and identify marmots and to relay messages to the base station. In a similar manner, the authors of Flask [Mainland et al., 2008] deployed a sensor network to detect earthquakes. The devices in the network, equipped with seismic sensors, were programmed with identical code to detect seismic activity and forward messages to the central server.

The second category involves heterogeneous networks — the nodes in the network are different from each other. This may include hardware

differences (i.e. architecture), software differences (i.e. programming languages) and differences in capability and functionality. A macroprogramming approach for heterogeneous networks must be aware of the differences in the target nodes, such that the code slicing and code generation process is aware of these differences.

2.3 Conclusions

In this chapter we have described macroprogramming — a technique used to program a network of nodes using a single program description. We have also outlined the three key technical challenges of the technique. Code slicing determines how the application logic is split and distributed across the available nodes, interoperability refers to how nodes communicate together and the heterogeneity challenge relates to the handling of architectural and functional differences of nodes in the network.

3

Embedded Domain Specific Languages

A domain specific language (DSL) is a programming language intended for a particular application domain. In contrast to general-purpose programming languages, an effective DSL allows a programmer to develop applications quickly and easily. It is easier to reason about and modify when compared to programs written in general-purpose programming languages, making it possible for non-expert programmers to use it [Hudak, 1998]. A good DSL covers precisely the semantics of the application domain — no more and no less, making it a better abstraction [Hudak, 1996]. DSLs are intended to be used by domain experts, rather than expert programmers.

However, it is often argued that the initial investment cost for creating the programming environment for a DSL outweighs the benefits reaped [Hudak, 1998]. A solution lies in embedding the DSL in another language, thus creating a domain specific embedded language (DSEL). The DSEL inherits the infrastructure of the host language, tailoring it in ways applicable to the domain of interest. Functional languages are often used as a host language due to their features which include pattern matching, algebraic data-types, lazy evaluation, higher-order functions, strong typing, polymorphism and overloading [Gibbons, 2015; Kamin, 1998]. Haskell, a pure functional programming language, is often the language of choice

and has been used successfully in various domains including hardware description [Bjesse et al., 1998; Pace, 2007], animation [Elliott and Hudak, 1997], digital signal processing [Axelsson et al., 2010a], origami-based geometry [Caruana and Pace, 2007; Hudak, 1998], graphics [Yates and Yorgey, 2015] and stream processing [Aronsson et al., 2014]. The syntax of Haskell is lightweight and flexible allowing the creation of a DSEL with syntax which is close to what one would provide in a corresponding standalone DSL [Elliott et al., 2003].

3.1 Shallow versus Deep Embedding

There are two main approaches to embedded DSLs — commonly called shallow and deep embedding [Gibbons, 2015]. In a shallow embedding, the terms of the DSL are implemented directly as the values to which they evaluate. With a deep embedding, the terms in the DSL construct an internal representation — such as an abstract syntax tree. The result of a computation inside a deeply embedded DSL is a structure that can be interpreted in different ways. The structure can be analysed, it can be transformed as part of an optimisation process and it can be evaluated into a value or cross-compiled into a low-level language [Gill, 2014] (see Figure 3.1).

Shallow Embedding

Historically, DSELs have been shallow embedded – which is good enough when there is only one interpretation of a program description. In figure 3.1, an arithmetic expression is evaluated into a value. No semantic information is stored about the expression, and the result is calculated using the host language's functions. Here we will illustrate using a simple function of how a simple shallow embedded DSL can be constructed in Haskell.

```
foo :: Float -> Float -> Float -> Float
foo x y z = (x * y) + z
```

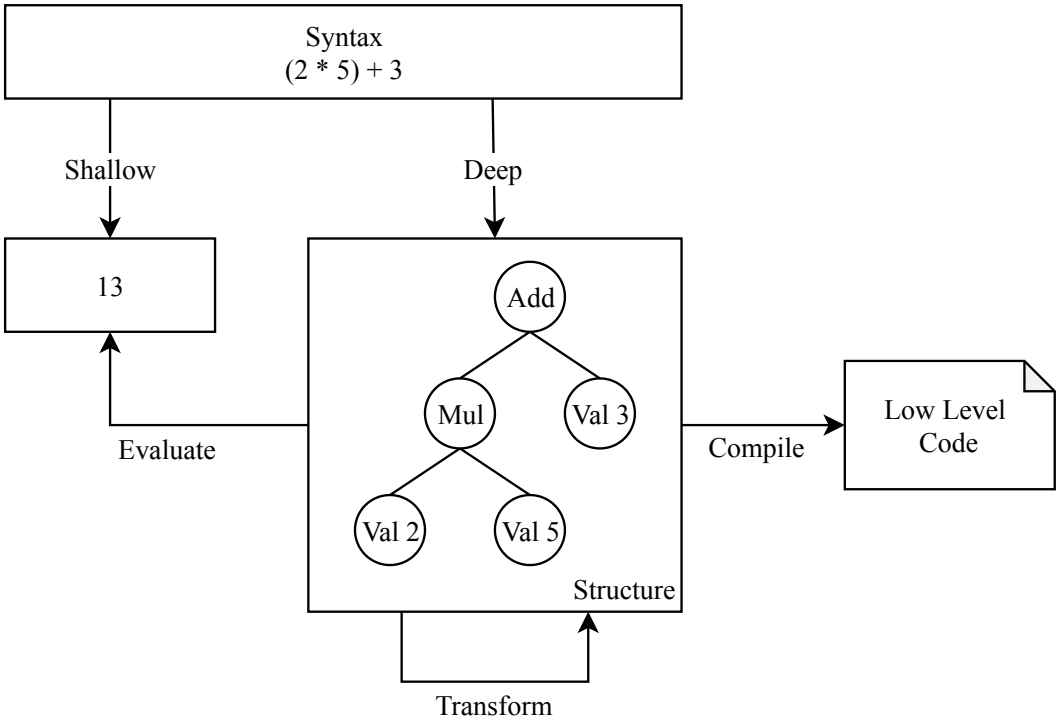

Figure 3.1: Shallow and deep embedding

The function `foo`, used to define a construct in a DSL, multiplies two numbers and adds the third one using the host language's native functions to evaluate to a result. The evaluation of the expression `foo 2 5 3` returns the value `13`.

Once evaluated, the structure of the original expression is lost. Shallow embedding is ideal when only one interpretation is needed. It has the added benefits of being fast to evaluate, as it makes use of the host language functions, and is quick to build up.

Deep Embedding

Deep embedding captures the semantics of the expression in a structure which can be transformed, analysed and interpreted in different ways. There are a few downsides for deep embedding – evaluation functions are more complex, and in general more code is needed. For a simple arithmetic DSL,

we would implement deep embedding in this way:

```
data Exp = Plus Exp Exp
         | Mult Exp Exp
         | Val Integer
  deriving (Show)

(.+.) :: Exp -> Exp -> Exp
x .+. y = Plus x y

(.*.) :: Exp -> Exp -> Exp
x .*. y = Mult x y
```

A function which is similar to `foo` defined earlier is `fooDeep`, defined as follows using a deep embedding:

```
fooDeep :: Float -> Float -> Float -> Exp
fooDeep x y z = (Val x .*. Val y) .+. Val z
```

An expression can be evaluated in different ways. The arithmetic evaluation is implemented as follows:

```
eval :: Exp -> Integer
eval (Plus a b) = (eval a) + (eval b)
eval (Mult a b) = (eval a) * (eval b)
eval (Val x) = x
```

The main advantage of deep embedding is that we have the structure of expression in an abstract syntax tree. We can analyse the expression or do transformations – for example, how many `Plus` operators are present in `(4 .+. 4 .+. 4)` or to convert the same expression into a `(4 .*. 3)`. The deep representation allows us to evaluate the expression in a different way – for example, to generate low-level highly optimised code to run on another device. Lava uses a deep embedding approach for hardware description [Bjesse et al., 1998].

Shallow and Deep Embedding

Shallow embedding is ideal when there is one interpretation of an expression and a DSL needs to be built and extended quickly. Deep embedding allows analysis, transformations and multiple interpretations of the same expression — it significantly improves the ability to manipulate the input expression. With deep embedding, new interpretations can be added easily, at the price of having a fixed set of constructs. Adding a new construct to the language means updating all existing interpretation functions. Feldspar combines the two approaches, deep and shallow, to attempt to gain the best of both worlds [Axelsson et al., 2010b]. However, when a shallow embedded construct cannot be translated into a deeply embedded one, the core language still needs to be extended [Svenningsson and Axelsson, 2012].

The following code, taken from Dévai et al. [2010], gives the reader a taste of Feldspar for a simple function to calculate the sum of squares.

```
sumSq :: Data Int -> Data Int
sumSq n = sum (map square (1 ... n))
  where
    square x = x*x
```

The code is very similar to a Haskell program. Depending on how it is interpreted, different outputs can be generated. Feldspar can generate low-level efficient C code by optimising the function structure before generating the code. The core program for `sumSq` is shown below:

```
*Main> printCore sumSq
program v0 = v11_1
  where
    v2 = v0 - 1
    v3 = v2 + 1
    v4 = v3 - 1
    (v11_0,v11_1) = while cont body (0,0)
      where
        cont (v1_0,v1_1) = v5
          where
```

```
    v5 = v1_0 <= v4
body (v6_0,v6_1) = (v7,v10)
  where
    v7 = v6_0 + 1
    v8 = v6_0 + 1
    v9 = v8 * v8
    v10 = v6_1 + v9
```

3.2 Challenges of DSELs in Functional Languages

Functional languages are often used for the embedding of DSLs. The clean syntax, lazy evaluation, pattern-matching, higher-order functions, polymorphism, and strong typing are features which come in useful when creating a DSL. However, a few challenges still exist which are described in this section.

3.2.1 Sharing and Feedback

One main feature of functional languages is *referential transparency*, for which the output of an expression is always the same for the same arguments. Referential transparency is a result of lambda beta reduction, where evaluation is simply argument replacement [Cordina and Pace, 2006]. Unfortunately this means that we are unable to detect when a component is used multiple times leading to multiple evaluations of the same component. Even more problematic is when feedback loops exist since with lazy evaluation this translates into a recursive and endless loop. For several applications, including hardware circuits [Claessen and Sands, 1999] and stream processing [Aronsson et al., 2014], detecting component sharing is a necessity.

Let's have a closer look at these problems. Consider a simple function, called `nextSquared`, which returns the square of the next number.

```
addI :: Integer -> Integer -> Integer
addI i1 i2 = i1 + i2
```

```
nextSquared :: Integer -> Integer
nextSquared inp = let x = addI inp 1
                  in x * x
```

In typical functional language fashion, the expression `x` is evaluated twice, even though the result will be the same. If this (simple) function had to be translated into a circuit, the resulting circuit would not be very efficient, as the same calculation is performed twice. Due to referential transparency, the translation process is unable to detect that the two components (`addI`) are identical — see Figure 3.2. A more efficient translation detects the reuse of the output from `addI` as shown in Figure 3.3.

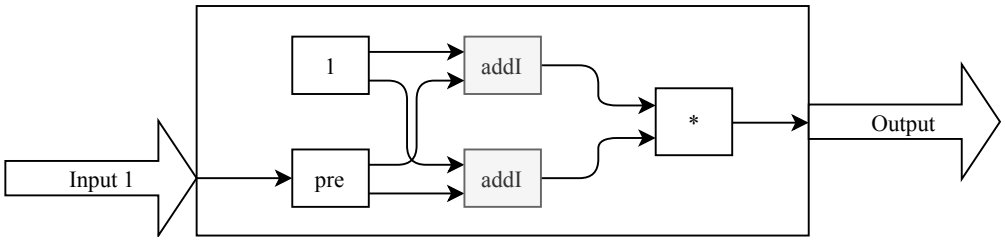

Figure 3.2: `nextSquared` (1) – the same calculation (`addI`) is done twice.

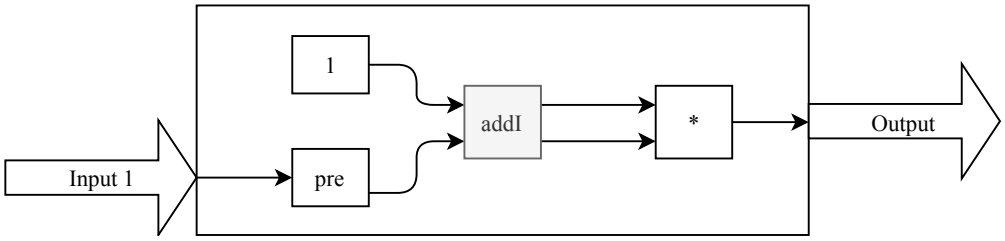

Figure 3.3: `nextSquared` (2) – `addI` is done once and the output used twice.

The problem is magnified in the case of feedback loops, since this will create an infinite loop with no terminating condition as the expression is evaluated to generate a circuit or a C program.

A number of methods have been used to address this problem [Cordina and Pace, 2006]:

Wire Forking involves the explicit use of a circuit that represents the forking, or duplication, of an input wire into two or more output wires. While this addresses the sharing problem, it does not address feedback loops and is therefore a partial solution.

Explicit Naming is a better solution where every component is given an explicit name by the programmer. During the translation phase, the names are stored and when the same name is encountered the evaluation stops to avoid endless recursion loops. This approach was used in Hydra for circuit design [O'Donnell, 2002]. The major drawback is that it relies on the programmer to keep track of names and ensure that the same label is not used more than once.

Monadic State solves the problem of having the user keeping track of names, as data is stored during evaluation and to ensure the same component is not evaluated a second time. The first implementation of Lava used this approach [Bjesse et al., 1998]. On the downside, it changes the way the user uses the DSL and introduces special operators that affect the readability of the code.

Observable Sharing is a technique proposed by Claessen which uses immutable references and a reference equality test [Claessen and Sands, 1999]. The approach is similar to explicit naming, but the naming (or references) are given to the components in an implicit manner and hidden from the user. The DSL remains clean and intuitive for the user.

3.2.2 Type Safety

When a DSL that handles multiple types is embedded in a statically typed functional language, the resulting DSL is not statically typed. A data declaration is unable to capture type information to ensure type safety and a DSL may fail horribly – with an incomprehensible error message. This is a common problem when embedding a DSL in a functional language. One way

of addressing this problem is by using a type-checker, however types are only checked at runtime, rather than at compile time, removing the benefits of static typing for the DSL.

```
data Expr = ValI Integer
          | ValB Bool
          | And Expr Expr
          | Plus Expr Expr
  deriving (Eq, Show)

iVal x = ValI x
bVal x = ValB x
myAnd x y = And x y
myAdd x y = Plus x y
```

The data declaration `Expr` and the constructors `iVal`, `bVal`, `myAnd` and `myAdd` can be used as a simple DSL that handles integers and booleans. The data structure and associated constructors do not restrict the user from creating ill-typed expressions.

```
>> myAnd (iVal 5) (bVal True)
And (ValI 5) (ValB True)

>> myAdd (iVal 5) (bVal True)
Plus (ValI 5) (ValB True)
```

The DSL has allowed us to create incorrectly typed expressions for `And` and `Plus` with an integer and a boolean — `And` needs two booleans and `Plus` needs two integers. At runtime, a type checker must be used to validate the types of the expression but this defeats the purpose of static typing.

The solution to these problems is to use Phantom Types which allow static typing in an embedded DSL [Hinze et al., 2003; Leijen and Meijer, 1999] and avoid type safety problems. A phantom type wraps around the existing data declaration becoming invisible from the user's perspective.

```
newtype ExprP a = ExprP Expr
  deriving (Eq, Show)
```

```
iVal :: Integer -> ExprP Integer
iVal x = ExprP (ValI x)

bVal :: Bool -> ExprP Bool
bVal x = ExprP (ValB x)

myAnd :: ExprP Bool -> ExprP Bool -> ExprP Bool
myAnd (ExprP x) (ExprP y) = ExprP (And x y)

myAdd :: ExprP Integer -> ExprP Integer -> ExprP Integer
myAdd (ExprP x) (ExprP y) = ExprP (Plus x y)
```

`ExprP` is introduced and wrapped around `Expr`, and constructors are modified to use the phantom type. The changes do not modify how the DSL is seen by the user, but attempting to create an expression with incorrect types (for example `myAnd` with two integers) will cause the compiler to report an error.

3.3 Conclusions

In this chapter, we have described embedded domain specific languages. We outlined the differences between shallow and deep embedding, as well as described the two key challenges when embedding in a functional language — that is, sharing and feedback, and type safety of the DSLs. We also described techniques to solve these challenges.

Part II

Macroprogramming for Wireless Sensor Networks

Now that the background for macroprogramming and embedded DSLs has been laid out, we propose a framework for macroprogramming heterogeneous wireless sensor networks. Chapter 4 provides background about the domain of Wireless Sensor Networks. Chapter 5 introduces a framework and language, called D'ARTAGNAN, for macroprogramming Wireless Sensor Networks.

4

Background: Wireless Sensor Networks

4.1 Introduction

Wireless sensor networks emerged at the beginning of the 2000s, when everyday objects started becoming equipped with sensors and microprocessors. This domain later became known as the Internet of Things domain, and a new wave of applications appeared for smart homes and buildings, environmental monitoring, and many others.

In this chapter, we will start by looking at the structure of a wireless sensor node. We will then describe the different challenges that exist in using and programming these nodes, before looking at the various programming models that exist.

4.1.1 Structure of a Wireless Sensor Node

A wireless sensor node is typically made up of a processing unit, a wireless communication interface, a number of sensors and/or actuators, and a limited power source (e.g. battery) — see figure 4.1. A wireless sensor network (WSN) is made up of a number of nodes and can be considered as a distributed system with important 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. Many WSN deployments use devices which have constrained resources. Limited processing capability, limited memory, and limited energy are three important constraints that have influenced how programmers implement applications on top of WSNs. The typical amount of memory (RAM) ranges between 2KB and 256KB, whereas program memory varies between 32KB and 128KB. Application logic is often written in low-level C or Assembly code making it quite hard for other programmers to read and understand code written by someone else, as well as making it difficult for the author of the same code to detect and fix a bug. Coupled with limited debugging utilities, which often are limited to a blinking LED, it makes programming of such devices a significant challenge.

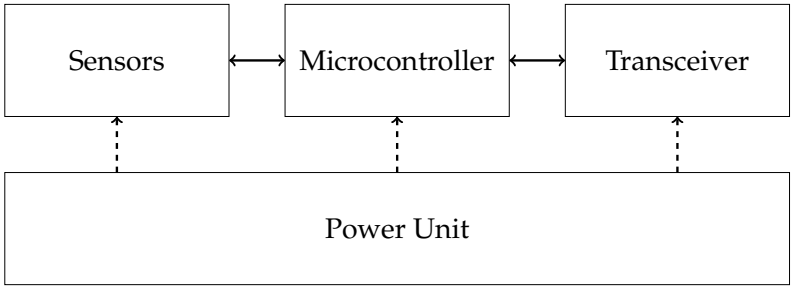

Figure 4.1: The structure of a wireless sensor node

In this chapter we describe the challenges for programming WSNs (Section 4.2) and different programming approaches used (Section 4.3).

4.2 Challenges of WSN

4.2.1 Hardware Constraints

Wireless sensor nodes have very limited resources — memory, processing and power. The limitations are primarily due to the small size of the devices, as well as to keep production costs low. Typical applications for wireless sensor networks often involve the deployment of hundreds (or thousands) of these devices in the area under observation. For example, in a

smart building, one or more devices with on-board sensors might be placed in each room of a building. The limited resources influence the design decisions of the application programmer on how these sensor nodes can be programmed to work together. In this section, we describe the different limitations of wireless sensor nodes and how they influence program design.

Power Limitations

Energy consumption is the most important factor that determines the lifetime of a wireless sensor network. A sensor node has limited power supply from a battery, and in some cases a node may be able to harvest energy from the environment through seismic, photovoltaic or thermal conversion. Careful and optimised use of energy determines whether the lifetime of a WSN is measured in days, months or even years. Several architectural and algorithmic approaches have been designed to optimise energy consumption. There are three main subsystems in a wireless sensor node which consume energy — radio, processing and sensing.

Energy optimisation can be done on two levels — node-level energy optimisation and network-wide energy optimisation [Raghunathan et al., 2002].

Several node-level energy optimisation strategies exist, such as: idle components are automatically put into sleep mode by power management schemes; energy aware software can help reduce consumption by being selective on computation; reduced precision when calculating a result can translate in energy savings; intelligent radio hardware that forwards messages received intended for other nodes without waking up other system components.

For network-wide energy optimisation, there are three main techniques [Anastasi et al., 2009]:

Duty cycling scheme — nodes alternate between active and sleep modes depending on network activity. A scheduling algorithm is used to de-

termine when and which nodes should sleep and wake-up at which time. In a simplistic scenario, the algorithm may determine that all nodes switch on the radio receiver/transmitter for a few milliseconds every minute on the minute for data transfer to be made. If the radio transceiver is on for just one second in every minute, the lifetime of the network will be roughly 60 times longer[1]. Routing techniques are used to distribute traffic evenly across the network, to avoid the quick depletion of energy of heavily used sensor nodes. Through topology management, nodes which are in close proximity take turns being active to prolong the longevity of the network. However, duty-cycling introduces a number of challenges. Firstly, the different devices may need to have a synchronised time to ensure that wake up and sleep times are done at the same time for the communication to be successful. Several algorithms and protocols have been designed to specifically address this challenge [Sundararaman et al., 2005]. Secondly, the reduced window of radio communication increases the likelihood of message collision — that is, several nodes may attempt to transmit at the same time. One solution for collision avoidance is the introduction of a random delay before communication is attempted again.

Data-driven approach — a trade-off is made between computation and communication. By utilising the computation capability of the sensor node, the collected information is reduced in size so that less data needs to be transmitted. Nodes may combine the readings from other nodes through aggregation, although this may mean less precision in data. For some applications it may be acceptable to receive aggregated data and suffer some precision loss as raw data is lost in the process.

Mobility-based schemes — reduction in energy utilisation through the use of a *mobile sink*. Depending on the type of application, it may be possible to reduce radio communication by utilising mobility — such as

[1]Work in this field typically makes the assumption that the energy consumption of the device excluding radio is negligible.

attaching a device to a moving object (e.g. a bus). As a *mobile sink* moves around the environment hosting the WSN, the sensor nodes can transmit information when the *sink* is close by in short-hop radio communication. This can help reduce long range or multi-hop communication, and therefore reduces energy consumption. On the other hand, this increases the latency of data collection as it creates a dependency on the movement of the *collector*. This approach is typically limited to a restricted set of applications and can be considered as less mainstream.

The lifetime of a wireless sensor network, and any application(s) deployed onto it, can be significantly extended once techniques, as described above, are used when compared to an implementation that does not take into consideration power limitations.

Memory Constraints

Memory is very scarce on a wireless sensor node. The typical amount of memory (RAM) is tens of kilobytes, whereas program memory is typically up to 256KB [Akyildiz et al., 2002]. Programmers who develop applications for wireless sensor nodes need to be extremely careful on the allocation and utilisation of memory — in terms of both program space as well as volatile space used at runtime. An *out-of-memory* scenario is more likely to happen than on a normal system as the available space is much smaller. Some WSN operating systems have developed techniques whereby different threads use the same limited memory space. For example, data used during computation of a thread is volatile and lost when the context is switched between one thread and another.

An *out-of-memory* error may cause an application running on a wireless sensor network to fail. Programmers need to be aware of the capacity of different nodes and tailor their programs accordingly.

Processing Limitations

Although higher computational power is being made available in smaller processors (such as microcontrollers), processing remains a scarce resource in a wireless sensor node. In general, the more powerful the processor, the more energy will be consumed. Since energy is a finite resource it is better to have a slow processor that lasts several months, rather than a fast processor which lasts just days[2].

The processing capability on a wireless sensor node can be used to process data coming in from the sensors such that only data of interest is transmitted. It is also possible to aggregate information inside the network as it is collected from the sensors to reduce the amount of information communicated. The processing power available is just enough to do limited amount of computation.

The microcontroller unit provides the processing capability of a wireless sensor node. Different microcontrollers have different power consumption levels. A high-end processor such as the StrongARM may consume around 400mW of power whereas an ATmega103L AVR microcontroller consumes around 16.5mW with significantly lower performance [Raghunathan et al., 2002]. The processing requirements of an application, whether heavy or light on processing, determine what type of microcontroller should be deployed. Tradeoffs need to be made between fast and slow processors. A fast processor may be more energy efficient if the computation is done faster allowing the microcontroller to sleep for longer periods of time and thereby saving on energy utilisation.

Sensors and Actuators

Depending on the nature of the application, and the type of sensors used, the sensing subsystem of a device could also be a significant consumer of energy. Passive sensors, such as temperature, seismic and humidity consume negligible energy compared to other components of a sensor node.

[2]Some studies are showing that this trade-off may soon disappear [Ko et al., 2012]

On the other hand, active sensors such as sonar rangers or radar sensors, can be large consumers of power [Raghavendra et al., 2006]. Sensors and actuators must be carefully switched on and off according to need such that the power consumption is reduced and the lifetime of the network is extended.

Radio Transceiver

The radio transceiver of a sensor node is a major energy consumer of the device. The energy cost for transmitting 1Kb a distance of 100 meters is estimated to be approximately the same as executing 3 million instructions on a general-purpose processor [Pottie and Kaiser, 2000]. In general, the radio operates in one of four modes – transmit, receive, idle and sleep. For short-hop transmission, the energy consumption for transmitting is equivalent to receiving. Also, radios consume roughly the same amount of energy when in listening mode as compared to during the receiving state [Raghunathan et al., 2002]. Therefore, the radio should be put in sleep mode when not being used to reduce the amount of energy consumed, while keeping in mind that significant power may be consumed during the wake-up phase as the radio transients from one state to another.

4.2.2 Programming Challenges

Hardware limitations and the large variety of platforms present a number of programming challenges. Applications are not implemented in a manner similar to conventional systems, as the dependencies on the underlying hardware and the lack of abstraction lead to different programming techniques. In this section we describe the two main aspects that make programming of wireless sensor nodes a challenge — heterogeneity and unreliability.

Heterogeneity

IoT applications are currently deployed across multiple architectures and platforms. There are over 150 WSN platforms from several suppliers — AdvanticSys[3], Shimmer[4], Zolertia[5], Libelium[6] and others, used for both commercial and research projects creating a highly fragmented landscape[7]. Even though the same components are used on different platforms, for example most platforms use either a 16-bit Texas Instruments MSP430 microcontroller or an 8/16-bit Atmel ATMega family microcontroller, the binary program code for one platform will generally not work on another platform.

Two popular sensor network operating systems, TinyOS [Levis et al., 2005b] and Contiki [Dunkels et al., 2004], have somewhat managed to improve on the fragmentation problem by being adopted by several platforms. Still, TinyOS, considered as the leader in this space, is only available, at best, on half the platforms. Contiki, is even less widespread. Code written for one platform can theoretically be re-compiled for a different platform without code changes. In practice, the use of different components, such as a different type of temperature sensor, may require the use of a different set of drivers and libraries which leads to some minor code changes and is therefore not completely platform independent.

Unreliability

Sensor nodes may fail when power runs out or if they become physically damaged. It is also possible that changes in the environment cause a sensor node to become disconnected from the rest of the network. The failure of a node, or a number of nodes, should ideally not impact the overall sensor network. An adequate level of fault tolerance is needed when using wireless sensor nodes. The fault tolerance level may vary depending on where

[3]https://www.advanticsys.com/
[4]http://www.shimmersensing.com/
[5]http://zolertia.io/
[6]http://www.libelium.com/
[7]https://en.wikipedia.org/wiki/List_of_wireless_sensor_nodes

the WSN is being deployed. For example, a sensor network deployed inside a house requires a different level of fault tolerance when compared to a network deployed in a battlefield.

4.2.3 Economic Challenges

Traditionally, WSNs are tailor built for specific applications, with little or no possibility of using them for additional applications. This approach is inefficient and often a major financial obstacle when designing a new application. It may also lead to redundant deployments which are costly to implement and maintain. In recent years, a number of studies have been made to address this restriction, by allowing a WSN to be shared between multiple applications.

Virtualisation is a concept that allows the abstraction of the underlying physical layer into logical units that can be used by independent users. This technique has been adapted to WSNs to become multi-purpose, and is broadly classified into two categories: node-level and network-level [Khan et al., 2016][Farias et al., 2016][Leontiadis et al., 2012].

Node-Level Sharing

A wireless sensor node can be used by multiple applications by having tasks (from different applications) run concurrently. There are different ways of achieving this via existing operating system models — an event-based model triggers a callback function to handle an event, whereas a thread-based model allows context switching between one application and another to run several threads concurrently. The former is a simpler implementation but may have limitations in that applications may block each other during the handling of an event. In the thread-based model, the operating system is more complex to implement due to the limited resources of the devices. Another approach is the use of a hypervisor on top of which different virtual machines are running [Levis and Culler, 2002]. Each application runs

in its own virtual machine and the underlying system takes care of the concurrent execution of multiple applications.

Network-Level Sharing

Through network-level sharing the same WSN can be used by different applications. Virtual networks are formed on top of a network of sensor nodes, or virtual sensor nodes. Several virtual networks may exist in the same WSN with different applications making use of different virtual networks. This is achieved in a similar manner as logical networks are formed over physical networks. Routing of messages between physical devices takes into consideration the virtual networks such that the devices can handle messages coming in from different applications and route them accordingly to other nodes or to the appropriate thread or virtual machine on the same node.

When the same WSN is shared between multiple applications, a separation of duties and responsibilities is created. A WSN administrator owns the network infrastructure and performs maintenance of the network and devices. Application developers develop and deploy their applications in virtual networks. Application developers would typically pay a usage fee to the WSN owner/administrator which goes towards the initial investment of setting up the WSN and the costs for maintaining it. Since the network is shared between several application developers, the fees paid would typically be lower than if a dedicated WSN was created and thereby help to lower the barrier to entry.

4.3 Programming Approaches

Programming wireless sensor nodes is done in a different manner to traditional applications, 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 abstract-

ing 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 [Sugihara and Gupta, 2008].

In this section we look at a number of the more common different programming models.

4.3.1 Node-Level Programming

Node-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.

Operating Systems

Operating systems provide a low-level programming model, where the programmer determines how a node is to behave. Operating systems for wireless sensor nodes need to take into consideration a number of factors [Chien et al., 2011]. The operating system, together with the program, has to fit on limited memory and must have a small footprint. Code may need to be upgraded through reprogrammability capabilities to be able to adjust the behaviour of individual sensor nodes from time to time. Good process and memory management mechanisms are needed to allocate processor time and the limited memory in a fair way, or according to priority. Operating systems need to be energy aware and have good power management such that components which are idle are turned off and turned on only when needed. Access to the underlying hardware is provided to the applications via a program interface, for example to read a sensor or to change the transmission power of the radio transceiver. Finally, the operating systems need to be reliable as they need to function well for long periods of time of unattended operation.

A number of operating systems have emerged in the sensor network community including TinyOS, Contiki, SOS, Mantis OS, Nano-RK, RETOS

and LiteOS [Chien et al., 2011].

TinyOS [Levis et al., 2005b] and Contiki [Dunkels et al., 2004] 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 one monolith binary. Multi-threading is possible using OS extensions in the form of thread libraries. TinyOS has a reprogramming capability, which due to the monolith approach means that applications are loaded with the OS kernel as a full image. Applications can be run in a simulated environment (such as TOSSIM) to observe behaviour.

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 different 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

Levis et al. [Levis and Culler, 2002; Levis et al., 2005a] 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. The main aim is to reduce the amount of data that needs to be transferred to reprogram the nodes. Maté [Levis and Culler, 2002] and ASVM [Levis et al., 2005a] are interpreter-based virtual machines that run on top of TinyOS [Levis et al., 2005b]. Darjeeling [Brouwers et al., 2009] and TakaTuka [Aslam et al., 2010] are small Java virtual machines that are optimised to run on resource constrained wireless sensor nodes. IBM Mote Runner demonstrates that VMs supporting various languages is doable [Caracas et al., 2009]. Interpreter based systems suffer from extensive overheads and work from Ellul and Reijers shows that ahead-of-time compilation techniques can reduce overheads [Ellul, 2012; Ellul and Martinez, 2010; Reijers and Shih, 2017]. Using virtual machines in the wild has proven to be a challenge, however recent work, proposing changes for real-world applications, has shown that it is possible to have a successful outcome [Reijers et al., 2018].

4.3.2 Network-Level Programming

There are two major approaches for network-level programming. One approach is a database abstraction, where the network is viewed in a similar manner as one would view and query a database for information. The other is to provide a macroprogramming language which provides a global view of the network and more flexibility for a wider variety of applications.

Database Query-like Languages

TinyDB [Madden et al., 2005] and Cougar [Yao and Gehrke, 2002] 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.

```
SELECT AVG(temperature), room FROM sensors
    WHERE floor=6
```

```
GROUP BY room
SAMPLE PERIOD 30s
```

In the example above, a query written in TinyDB, the average temperature from every room on the sixth floor is retrieved every thirty seconds. 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

Macroprogramming languages can be used for a wider range of applications than the database abstraction — applications where data may flow between one node and another, and can be processed inside the network.

Macroprogramming languages can be classified into two categories — the first category is for sequential imperative programming models and the second category is for a declarative functional approach. The languages Pleiades [Kothari et al., 2007], Kairos [Gummadi et al., 2005] and COSMOS [Awan et al., 2007] are examples of sequential imperative programming models. Regiment [Newton et al., 2007b], Flask [Mainland et al., 2008] and Wavescript [Newton et al., 2008] are examples of declarative functional programming models.

These programming languages have been described earlier and the reader is referred to Section 2.1.2 for a more detailed description.

4.4 Conclusions

In this chapter we have described wireless sensor nodes and networks — resource constrained devices which have limited computational power and can communicate wirelessly. We have described the structure of a wireless sensor node, as well as the various challenges encountered when program-

ming and using these devices in practice. We have also outlined different programming approaches used for these devices.

5

D'Artagnan

5.1 Introduction

Over the past 15 years, there has been a growing trend of embedding sensors and microprocessors in everyday objects so they can communicate information and interact with their environment [Pottie and Kaiser, 2000]. 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 [Mainwaring et al., 2002]. 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 [Werner-Allen et al., 2005]. 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 [Chen et al., 2009].

Applications in this domain can be 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 substantial and spread across a distributed network of heterogeneous

resource constrained, unreliable, wireless nodes. Developing applications for 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) [Mernik et al., 2005]. 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 time, and even less skills. However, building a DSL may require significant initial investment to build the right tools for application development [Hudak, 1998]. 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 [Claessen and Pace, 2002].

As the level of abstraction is increased through the use of a DSL, the ability to optimise code may be lost — less low-level control. Good programmers use knowledge of the environment and how the application is going to be used to optimise code, however at this high-level, optimisation is impossible. The use of annotations may help overcome this problem as the programmer can provide hints to the compiler to be used for optimisa-

tions.

Using the technique of embedding a language, we present a macroprogramming framework to describe stream processors. The D'ARTAGNAN DSL, embedded in Haskell, can be used to analyse, generate, transform and interpret stream processor descriptions. We take inspiration from the work done in hardware description with Lava [Bjesse et al., 1998] and in digital signal processing with Feldspar [Axelsson et al., 2010a]. Our approach shares similarities with Flask [Mainland et al., 2008] in the generation of code for resource constrained devices. Our aim is to create a single stream processor description that can (i) have different interpretations — simulated or translated (compiled) to low-level code; (ii) be analysed for both functional (e.g. number of times sampling is done) and non-functional aspects (e.g. memory requirements) and (iii) be optimised through transformations, such as alternative energy efficient communication strategies.

The research questions that motivate this work are:

- 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 such performance penalties be mitigated?

- Distributed embedded systems are known to be heterogeneous. Can the same stream processor description be used to generate code for different architectures? Can the stream processor be also simulated at the high level of abstraction to observe the behaviour in a simulated environment and thereby simplifying the test and debug cycle?

- 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 hints or annotations be generalised to apply to different contexts?

This chapter describes an overview of our framework and language. Two use-cases are used to illustrate the proposed model — the first use case deals with smart rent management, and a second use case example is used for an intelligent and energy-efficient building cooling and lighting systems to validate whether the proposed approach can be used effectively in real scenarios.

5.2 A Framework for Macroprogramming of WSNs

Sensor networks are traditionally programmed using a low-level program that is compiled and installed on each individual node. Environment conditions are collected using on-board sensors, data is processed at source and information of interest is passed on to neighbouring nodes via radio messages. The programmer defines the behaviour of the nodes under each possible scenario, taking care of maximising available power by implementing a synchronised radio duty-cycle across all nodes so that they can communicate with each other in a power efficient manner. In a macroprogramming 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.

We propose D'ARTAGNAN— a macro-programming model using an embedded domain specific language (DSEL) approach. Our aim is to increase 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. We use a data type approach to achieve deep embedding.

In D'ARTAGNAN, the key feature is a stream processor description that generates an internal representation that can be (i) analysed (ii) transformed

and (iii) interpreted in different ways. Figure 5.1 illustrates the framework. The internal representation is in the form of an abstract syntax tree built from the deep embedding of the D'ARTAGNAN language in Haskell, the host language.

The programmer can also provide compiler *hints* to influence how the internal representation is interpreted for improved code generation.

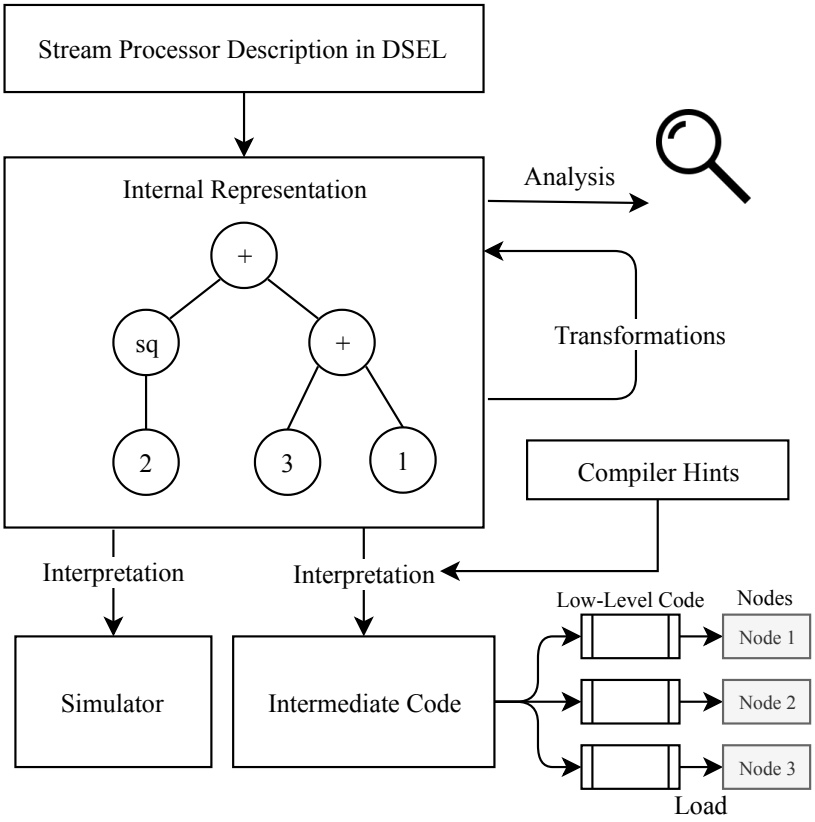

Figure 5.1: A high-level overview of the D'ARTAGNAN framework.

The analysis of a stream processor can be done by the traversal of the internal representation. The result of the analysis can be used for optimisation of the stream processor or to calculate and report on metrics. For example, the number of radio messages that will be used under a certain configuration; the amount of power used; to calculate the lifetime of the network in a specific configuration. Through analysis, different layouts may be evalu-

ated to optimise and reduce radio messages.

The internal representation can be transformed in different ways, possibly by using information gathered through analysis. For example, a transformation may move computation across the network to reduce power utilisation 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 (Section 5.4.3). For example, if the network contains a more powerful node with a permanent, renewable 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 not easy to achieve in a real environment. Further, 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.

5.3 Interpretations

One of the key strengths of our approach is the ability to have multiple interpretations for the same stream processor. At the top-most level there are two main types of interpretations for a stream processor — see Figure 5.2. The first type is a simulator, where the stream processor is evaluated according to specific input values. As an example, consider the following simple stream processor — `sum`:

```
sum :: (Stream Int, Stream Int, Stream Int) -> Stream Int
sum (input1, input2, input3) = input1 .+. input2 .+. input3
```

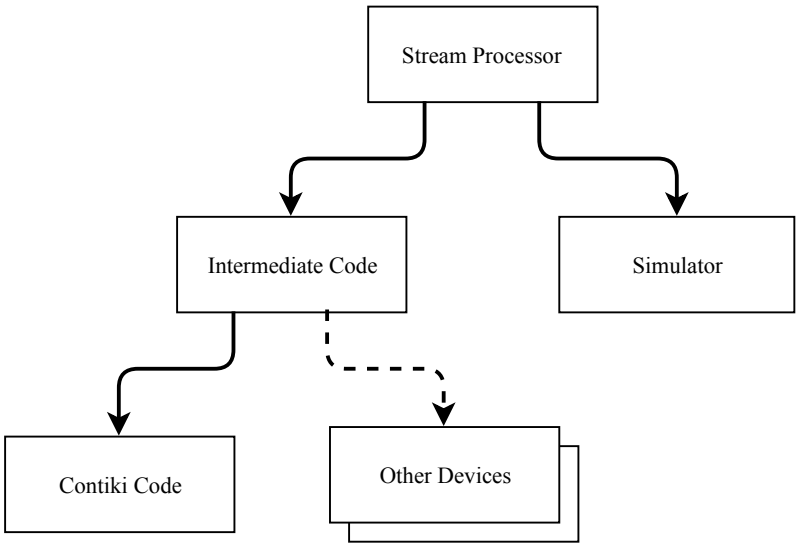

Figure 5.2: Different interpretations of a stream processor

`sum` takes a tuple of three integer streams and returns their sum in a new integer stream. The results of this stream processor can be observed under different conditions by using the simulator interpretation.

```
> simulate sum [(2,3,1), (4,1,6), (1,3,3)]
[6,11,7]
```

The second type of interpretation is intermediate code, an abstract representation of code, that can then be further translated into code for different devices and operating systems. The intermediate code is used to generate device-specific statements, dependent on the actual hardware it is being deployed on. This approach allows the possibility to add new platforms as and when required without changing the core of the framework.

It is also possible to introduce new interpretation types which are different from simulation or generated code. The approach of multiple interpretations has been used successfully in other fields, such as hardware description languages, where the output of an internal representation can generate several different outputs that feed into other tools.

5.4 D'ARTAGNAN as a language

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.

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

A stream processor works by taking readings from sensors periodically — the period can be set at compile time, 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, a new average value is calculated by taking new readings from the three sensors with every clock 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.

5.4.1 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` 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 is `True`.

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

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 sen-

sors 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 `moreThan50` as follows:

```
>> moreThan50 (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` is too rigid — it 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., 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, which is not a stream processor *per se,* but a whole family of stream processors generated by different parameters passed to the function.

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

Extending the fire alarm system to use two sensors is quite straightforward. The alarm will sound if any of the two sensors has a reading higher than the threshold.

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

By defining custom stream handler operators (such as `orList` below), a more generic fire alarm system can be built which takes any number of sensors.

```
orList :: [Stream Bool] -> Stream Bool
orList ss = foldl1 (.||.) ss

firealarm3 :: Int -> [Stream Int] -> Stream Bool
firealarm3 threshold ss =  orList (map (.>. threshold) ss)
```

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.

The structure of the language is shown below in pseudo BNF.

```
StreamProcessor = input <deviceLocation> <sensorNumber> |
                  constant <value> |
                  StreamProcessor InfixOp StreamProcessor |
                  max (StreamProcessor, StreamProcessor) |
                  min (StreamProcessor, StreamProcessor) |
                  if <boolean> then StreamProcessor else StreamProcessor |
                  pre <init> StreamProcessor |
                  pull <device> StreamProcessor |
                  push <device> StreamProcessor

InfixOp = .+. | .-. | .*. | ./. | .||. | .&&. |
          .==. | .>. | .<. | .>=. | .<=. |
```

A stream processor is constructed by combining stream operators together, including the reading from sensors, as well as mathematical or boolean operators.

5.4.2 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 5.1. A visual representation of the same system is shown in Figure 5.3.

Listing 5.1: Fire alarm system with notification

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

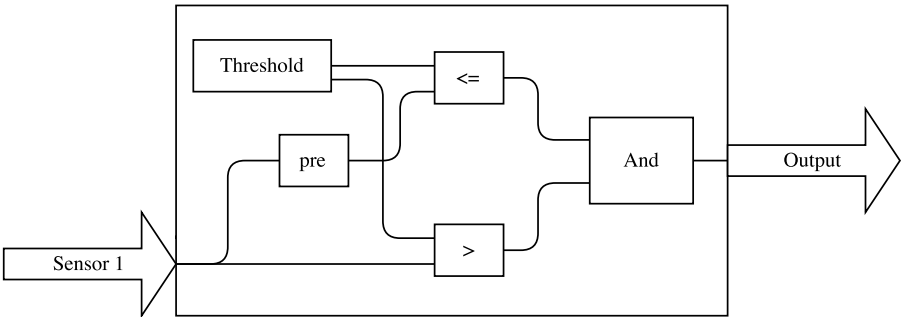

Figure 5.3: A fire-alarm notification system

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 5.2 and visually in Figure 5.4. 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 [Claessen and Sands, 1999].

Listing 5.2: Fire alarm system - Sticky Alarm

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

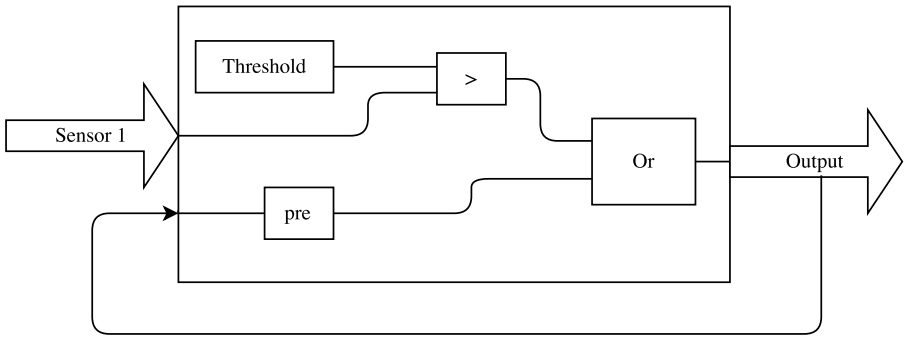

Figure 5.4: A sticky alarm system — once triggered, remains *ON* until system is reset.

5.4.3 Compiler Hints

When a stream processor description is compiled to generate code to run on different devices, application logic is split and assigned to specific nodes. By the nature of the devices, certain logic is bound to a specific node — for example, the code for reading a specific sensor must be placed on the node where that sensor is located. However, other logic is not bound and this may be placed on any node in the network.

One way of improving application logic placement is by getting information (or hints) from the programmer who has application knowledge and can guide the compiler to work out an improved placement.

Communication Operators

Applications deployed on WSNs make use of inter-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`. `Pull` is based on a request and response pair of messages, and makes use of two radio messages. On the other 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.

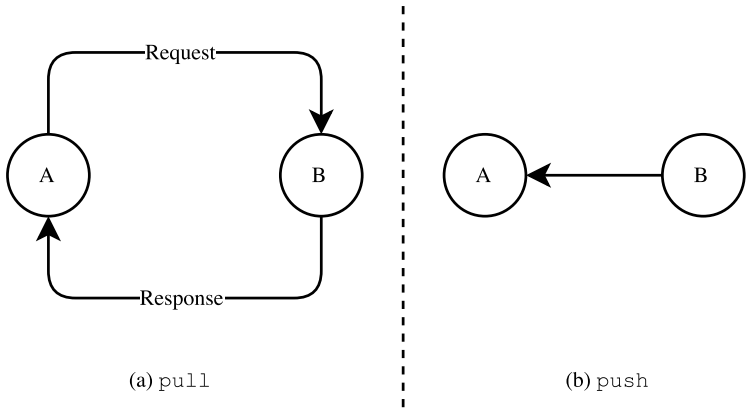

Figure 5.5: Point-to-point communication models

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 use

```
push (device 2) (input2 .*. input3)
```

to indicate that the computation `(input2 .*. input3)` should be processed on `device 2`. The result of this is then sent to `device1` to be added to `input1` — `input1` is located on `device1`.

Placement Strategies

The communication operators `push` and `pull` allow the programmer to define how communication between nodes should occur and also to influence, to some degree, on which nodes application logic should be placed. However, in D'ARTAGNAN, we want to provide the programmer with a higher level of abstraction for placement of application logic through the use of compiler hints. The use of placement strategies is optional (hence the term *hints*) — in the absence of a placement strategy, the default placement algorithm is used. D'ARTAGNAN supports two placement strategies: (i) `LatestPossible`, the default placement strategy, uses communication operators at the last possible moment when data is required from a node (ii) `EarliestPossible` instructs the compiler to use communication operators the earliest possible for the added advantage of in-node data processing so as to reduce the amount of data transmitted.

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

5.4.4 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 two stream processors `firealarmNotification` and `stickyAlarm` into one description as shown by `firealarmPlus` in Listing 5.3.

Listing 5.3: Fire alarm system with notification and sticky alarm using tuples

```
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)
```

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 5.1 shows an example scenario with input and output values.

Time	**Sensor Reading**	**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 5.1: Example (`firealarmPlus`) with output tuple of streams

5.4.5 Simulator

D'ARTAGNAN provides a simulator interpretation that calculates the output of a stream processor given input values whilst still in the Haskell envi-

ronment. 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 `firealarmNotification` 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. $[v_0, v_1, v_2, \dots]$.

```
>> simulate firealarmNotification [45,48,55,56,57]
[False,False,True,False,False]
```

5.4.6 Intermediate Code / Device 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 two-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, applications for WSNs are written in the C programming language or a WSN-specific dialect (e.g. nesC for TinyOS) [Mottola and Picco, 2011]. 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 two-staged approach has also been used in Feldspar [Axelsson et al., 2010a] and Regiment [Newton et al., 2007b].

5.4.7 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[1] test bed and another one for the AdvanticSys CM5000. Other C variants can be added with relative ease. Listing 5.4 shows the automatically generated code that can be used in a Contiki implementation.

Listing 5.4: Automatically generated C code

```
static bool mem0 = False;
static int mem1 = 0;
...
static int x1,x6,x5,x13,x7,x14,x15,x8,x16,
          x17,x10,x18,x19,x20 = 0;
static bool x4,x3,x21,x12,x11,x9 = false;
static bool x22[10];
static bool x2[10];
x3 = mem0;
x5 = (int) readTemperature();
x6 = 50;
x10 = mem1;
x4 = x5 > x6;
x20 = x6;
x19 = x5;
x17 = x10;
x16 = x6;
x15 = x5;
x14 = x6;
x1 = x3 || x4;
x9 = x20 > x10;
mem1 = x19;
x11 = x17 == x16;
x12 = x15 > x14;
x13 = x1;
x8 = x9 || x11;
mem0 = x13;
x7 = x8 && x12;
```

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

```
x2[0] = x1;
x2[1] = x7;
```

5.4.8 Implementation Details

D'ARTAGNAN is deeply embedded in Haskell, the host language. A stream processor description is processed by the framework in a number of steps.

- A pre-processor is first used to detect any recursion in the stream processor description. The stream processor is traversed, and each node is labelled. If a labelled node is encountered a second time in the graph, then this is replaced with a new type of node (called duplicate) which links to the original node. This approach ensures the compiler does not end in an infinite loop when generating target device code or during analysis.

- Each node is temporarily assigned to be executed on one of the available devices. Sensor readings are, by nature, bound to a specific device, while other operators can be placed on any available node. In general, nodes are placed on the same device as neighbour operators, such that the number of radio messages is minimised. This assignment is only an initial placement and can be modified as part of the transformation phase.

- The transformation phase may optionally take input from the programmer, in the form of compiler hints, to follow a specific placement strategy. At this stage, the operators may be re-assigned to other nodes. Analysis is used to find preferred placement options by using specific metrics (e.g. less radio messages).

- Once placement is finalised, the stream processor is traversed once again to introduce communication operators where the computation has to move from one device to another. Where two neighbour oper-

ators are on different nodes, a communication operator is introduced. This is an instruction for the compiler to create communication code.

- During the generation of intermediate code, the stream processor is traversed and connected operators running on the same device are translated to an abstract C-language equivalent. Communication operators determine the start and end of a C-language 'method'. At the end of a sequence of operators on the same device, a call to a method on another node is introduced to fetch the result of the computation of a sub-tree.
- In the final stage, the Abstract C is converted to device specific code. In general, most devices run some variant of the C-language and this stage involves generating the correct C dialect.

5.4.9 Discussion

D'ARTAGNAN was designed as a high-level language for describing stream-processing applications. An application is defined by connecting stream operators which take a number of inputs to generate one or more data stream outputs. The execution of an application generates an output for that instant. A computation unit is a sequence of operators that are executed together on the same device. The computation of the application includes the firing of several computation units, and is considered complete when all computation units have fired and an output has been generated. If any computation unit fails, the application computation is considered to have failed. The result of a computation can be used in the next computation through the use of memory, thereby enabling a richer set of applications.

In the current version of D'ARTAGNAN, the execution of an application is triggered by a clock on a root node firing at regular intervals. A message is sent to all devices to prime the computation units and execute as soon as all inputs are received. This approach is suitable for a category of stream processing applications used for monitoring purposes, where readings from the environment are recorded at regular intervals. The approach

is less suitable for applications where the execution trigger is determined from an incoming data source (for example, a push-button). One way of addressing this is by introducing triggers (from sensors or other timers) as first-class citizens of the language such that the execution of an application starts when a sensor detects an event (e.g. a switch is pressed), or when a timer fires. We consider the addition of triggers as a suitable enhancement to the language to address a broader range of applications.

5.5 Use-case: Smart Rent Management

As a first use-case to illustrate the use of D'ARTAGNAN, we present an application for smart-rent management. Short-term rental sites typically allow home-owners to rent out their property using a fixed day-rate model. Determining an optimal day rate is one of the biggest challenges for home-owners to maximise on profits. A high day-rate may translate in less booked nights, whereas a low day-rate could possibly lead to low margins, or even losses, if tenants are high consumers of commodities. An alternative solution may be a fixed low daily rate to attract budget travellers and increase booked nights, combined with a variable pay-per-use rate for commodities — electricity and water consumption is charged at a pre-agreed rate and the use of appliances, such as a washing machine, dish-washer and air conditioning attract an additional charge to compensate for the wear and tear of these appliances.

To illustrate the basic concepts of the D'ARTAGNAN framework, we show a single-description smart-rent application that calculates meterage of the usage of appliances inside a home. The same description, in our case the code describing the stream processor, can be used to generate different target code depending on utility rates and the devices the application will run on. The generality of this approach allows the same application to be used at different homes where the appliances available may vary, and is by far easier to manage than an equivalent written directly in low-level code.

We identify two primary ways for charging the use of commodities, but

other creative ways may easily be defined too. The first method is a simple pay-per-cycle, where a fixed fee is charged for each use of an appliance — for example, a 2 euro charge is imposed for every washing machine cycle. The second charging method is a pay-by-consumption model, where a pre-defined rate is used for every unit consumed and can be applied to electricity, water, soap, fuel and other consumables. Listing 5.5 shows the definition of these two methods. The function `payPerCycle` takes a stream of boolean values to indicate when, for example, a new washing machine cycle has started and outputs the fee charged in a stream of integers. When the appliance is not used, a stream of zero values is output, and when the appliance is used the fee is included in the output feed. On the other hand, `payByConsumption` simply multiplies the fee with input units to generate a stream of fees to be charged. A third function (`meter`) uses the `pre` operator to act as an accumulator for an input stream of integers, by keeping track of total consumption — this is the equivalent of a metering device.

Listing 5.5: Stream Handlers

```
payPerCycle :: (Int, Stream Bool) -> Stream Int
payPerCycle (fee, inUse) = ifThenElse (inUse, (liftS fee, liftS 0))

payByConsumption :: (Int, Stream Int) -> Stream Int
payByConsumption (fee, usage) = liftS fee .*. usage

meter :: Stream Int -> Stream Int
meter feed = let x = pre 0 x .+. feed
             in x
```

The D'ARTAGNAN simulator can be used to assist in the writing and testing of stream processors, including helper handlers as shown in Listing 5.5. As part of the simulation, we provide a list of input boolean values (to represent readings taken from devices), so that we can observe the output for specific inputs and confirm that the behaviour is as expected.

```
» simulate (payPerCycle 2 [F, F, T, F, T, F, F])
[0, 0, 2, 0, 2, 0, 0]
```

Using `meter` as an accumulator, we can keep track of a running value for consumption — the last value of the stream is the total consumption since the initialisation of the system.

```
» simulate (meter (payPerCycle 2 [F, F, T, F, T, F, F]))
[0, 0, 2, 2, 4, 4, 4]
```

Now that we have described the utility stream handlers in Listing 5.5, we can easily construct our smart rent application. The stream processor `smartMeter` in Listing 5.6 takes a number of inputs and calculates the fees to be charged depending on whether a pay-per-cycle or pay-per-usage model is applicable. Since our language is embedded in a host language (Haskell), the use of `map` and `fold` can be used to increase the expressiveness of the language — in this example, we apply the same rate to a list of appliances.

Listing 5.6: Smart Rent

```
smartMeter :: (Int, Int, Int, Int, Int, Int)
              -> (Stream Int, Stream Int, Stream Bool, Stream Bool,
                        Stream Bool, [Stream Bool])
              -> Stream Int
smartMeter (elecRate, waterRate, wmRate, dwRate, tdRate, acRate)
        (elec, water, washingmachine, dishwasher, tumbledryer, airconList)
                = meter inputs
  where
    inputs = payByConsumption (elecRate, elec) .+.
             payByConsumption (waterRate, water) .+.
             payPerCycle (wmRate, washingmachine) .+.
             payPerCycle (dwRate, dishwasher) .+.
             payPerCycle (tdRate, tumbledryer) .+.
             foldl (.+.) (liftS 0)
                (map (\x -> payPerCycle (acRate, x)) airconList)
```

A more generic, yet more concise, version of the smart rent application is shown in Listing 5.7. Any number of appliances can be passed as parameters, together with the applicable fee rates, into the compilation stage divided into lists according to the charging model which applies. The use of Haskell's `zip` function takes two lists and creates a list of tuples, which we can then pass into the stream-handlers `payPerCycle` and `payByConsumption`.

Listing 5.7: Smart Rent

```
smartMeter2 :: ([Int], [Int]) -> ([Stream Int], [Stream Bool]) -> Stream Int
smartMeter2 (rateUsage, rateCycle) (appUsage, appCycle) = meter inputs
  where
    inputs = foldl (.+.) (liftS 0)
                   (map payPerCycle (zip rateCycle appCycle)) .+.
             foldl (.+.) (liftS 0)
                   (map payByConsumption (zip rateUsage appUsage))
```

A specific instance of this application is created with fee amounts and real sensors passed as input parameters such as:

```
» generateCode (smartMeter2 ([2,1], [10,7,4,3,3]) ([elecMeter,
waterMeter], [washingmachine, dish_washer, tumbledryer, aircon_1,
aircon_2])
```

Behind the scenes, D'ARTAGNAN creates code that will run on each of the devices to measure unit consumption or to indicate that an appliance is being used. The placement of in-network computation and communication between the devices is determined and handled by D'ARTAGNAN—thereby hiding away all the complexity from the programmer. Since the D'ARTAGNAN language is embedded in a host language, functions in the host language become available in our language to create a richer and more expressive language for programmers to define stream processors in a concise manner.

5.6 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 5.4.1 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 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 systems are then switched off when no motion is detected in the room or neighbouring rooms. 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 5.6). 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 5.7 illustrates internal detail of the building stream processor — a room stream processor.

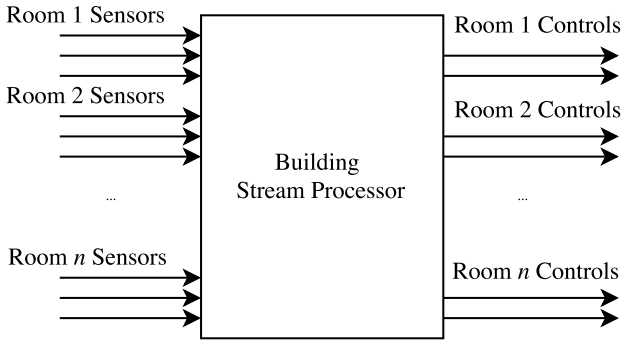

Figure 5.6: Inputs and outputs of a building stream processor

5.6.1 Stream Handling Components

Higher level stream handling components are needed to transform sensor readings into application specific meaningful information. Listing 5.8

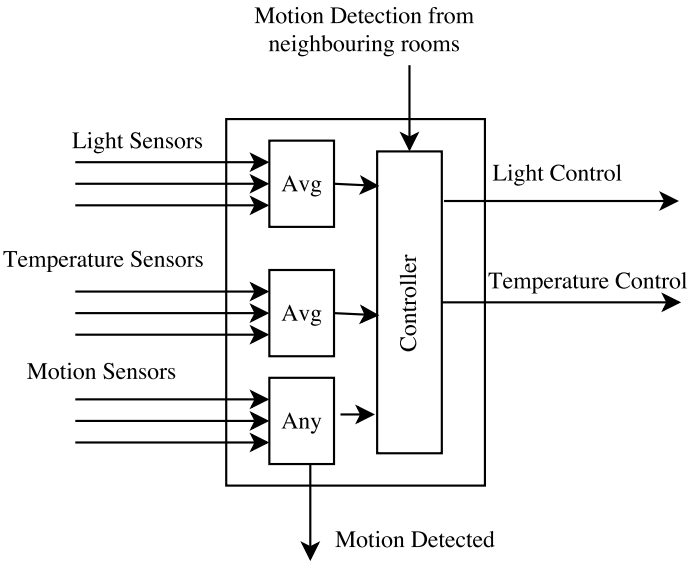

Figure 5.7: Internals of a room stream processor

shows custom stream handlers that will be used in building the application.

Listing 5.8: Custom stream handlers

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

minControl min s = s .<=. consStream min

maxControl max s = s .>=. consStream max
```

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 `orList` (defined earlier) to combine sensors 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 deter-

mine 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.

5.6.2 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 5.8, 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.

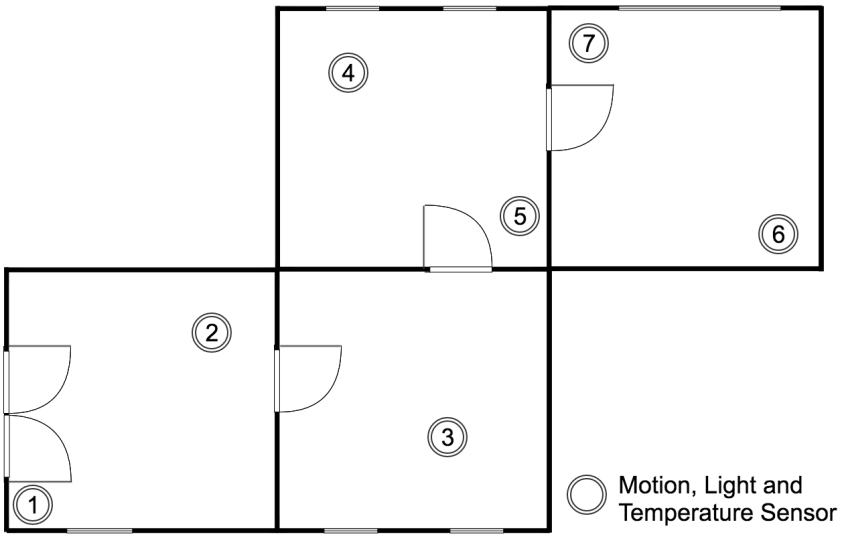

Figure 5.8: Room layout with device placement.

The building topology is represented by a graph using the data types `Plan` and `Room` — see Listing 5.9. `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 5.8 is given in Listing 5.10.

In this specific room layout, when motion is detected in Room 1, lights and cooling in Rooms 1 and 2 will be switched 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 5.9: Building data types

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

Listing 5.10: 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)]
```

5.6.3 Application Implementation

Typically, one would program the devices for a particular topology. Any changes in sensor deployment requires reprogramming from scratch. Similarly, given a new building, devices for that building need to be programmed from scratch. 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:

```
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
    adjRooms = adjacent plan room
    motionSensors = msToStream (getMotionSensors adjRooms)
    lightSensors  = lsToStream (getLightSensors adjRooms)
    tempSensors   = tsToStream (getTempSensors adjRooms)
```

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 5.8 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 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),...]
```

At T1, motion was detected in the first room. The lights and cooling systems for the first and second rooms are turned on.

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.

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[2] test bed — a platform suitable for testing small wireless devices in a real environment.

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

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 are enough to create a generic intelligent

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

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 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. In the specific room layout example with four rooms and seven devices, the system generates 980 lines of C code. 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 devices, the C code may need to be modified and size will increase linearly with the number of rooms and nodes introduced.

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 without receiving a request (push). This improvement can be implemented for both versions.

5.7 Performance Evaluation

To assess the performance of our approach versus a hand-coded version in C, we have implemented the same process intensive task using both approaches. We use an audio sound soft-clipping algorithm, which for the limitations of wireless sensor nodes can be considered a processing intensive task. The hand-coded version makes use of a small math library for arctan approximation used in soft-clipping calculations. The second implementation is written completely in D'ARTAGNAN, including mathematical functions for arctan approximations.

For evaluation, we use Contiki on an AdvanticSys CM500 equipped with an MSP430F1611 Texas Instruments micro-controller, 48KB of program flash and on-board sensors for temperature, humidity and light. Both im-

plementations were compiled using the MSP430 GCC (v4.6.3) with different optimisation levels to study how compiler optimisations interact with our code generation. Optimisation levels range from level 0 (-O0) to level 3 (-O3).

	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: 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%

The results in the table above indicate that, as expected, with no compiler optimisations (-O0), Implementation 2 — 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 bigger. For this experiment, we did not measure energy consumption since the task was a computationally intensive one with no use of external peripherals or communication, therefore the energy consumed is directly proportional to the time taken by the node to finish the audio clipping task.

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.

5.8 Related approaches

Over the past decade, a number of DSELs for resource constrained devices have emerged. In this section we describe such frameworks, all of which have been embedded in Haskell.

Flask [Mainland et al., 2008], already introduced in section 2.1.2, 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 Haskell-like language) or nesC using quasiquoting. 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. Unlike D'ARTAGNAN, Flask is intended to execute on homogeneous networks, where the nodes of a network are programmed using the same generated code. D'ARTAGNAN does not have the notion of quasiquoting, as programmes are defined using the high level language.

Ivory and Tower [Hickey et al., 2014] are two complementary DSELs designed for building applications to run on embedded systems. Both languages are embedded in Haskell — Ivory compiles to restricted C code suitable for embedded programming, whereas Tower is an extension to Ivory designed to deal with the concerns of multithreaded software architectures. The authors of Ivory claim that one of the main advantages of their approach is that by embedding the domain-specific type checker into Haskell's type system, type safety is guar-

anteed throughout the generated code and thereby reducing common errors made during programming of embedded systems. The overall productivity is increased drastically. Ivory includes quasiquoting allowing C code to be embedded directly. Tower was created to address needs that were outside Ivory's domain – to act as "glue code" to implement inter-process communication, initialise data-structures, read system clock, lock the processor, etc. One of the main benefits of Tower is that it supports multiple interpreters allowing code to be generated for different embedded systems operating systems, as well as system descriptions and visualisation graphs with Graphviz. Unlike D'ARTAGNAN, Ivory and Tower are intended to provide a higher level of abstraction for writing safe programmes on singular embedded systems. Tower defines inter-process communication, whereas in D'ARTAGNAN, a single stream processor description is automatically sliced to run on several nodes as required — no explicit definition is needed, although the use of hints can influence the compiler. D'ARTAGNAN is a more restricted language, and does not allow the use of low-level code such that code can be generated according to the target architecture.

Copilot [Pike et al., 2010] is a DSL embedded in Haskell. It is intended for use in runtime monitoring (runtime verification) of hard real-time systems. The DSL is a stream-based data-flow language. Copilot samples global variables to check that the state of the application is correct. The DSL can be used to create, as an example, a monitor to ensure that the temperature reading does not increase too quickly. The variable sampling technique is suitable since hard real-time systems have a strict schedule that is adhered to, and therefore removes the risk of false positives and false negatives arising from incorrect sampling. Copilot does not change the source program although may run on a different process on the same device. Since it is constant-time and constant-space, it can guarantee on finishing on time and not to interfere with the main application. The DSL is statically and

strongly typed — which means that incorrect (badly typed) expressions are caught at compile-time. The types are embedded in Haskell's type system. Types are lifted into streams, and arithmetic and logical operators can be used on streams. A monitor specification needs to be *well-defined* and *well-formed* — meaning that circular dependencies are not allowed and there must be no dependencies on future values. D'ARTAGNAN is also a statically and strongly type language for stream-based applications, however circular dependencies are allowed through the use of memory (`pre`). D'ARTAGNAN is focused on writing stream-processing applications, whereas Copilot's main aim is to monitor other applications. D'ARTAGNAN is also intended to generate code for heterogeneous networks, whereas Copilot generated C code to run on a single device.

Feldspar [Axelsson et al., 2010a] is a DSL embedded in Haskell intended for digital signal processing algorithm design. Unlike the other languages in this section, it is not intended for resource constrained devices, but rather generates performant C code that manipulates an incoming stream of data. The aim of Feldspar is to raise the level of abstraction at which a programmer works with algorithms to reduce the development time, and the writing of code is very close to the mathematical way of writing algorithms. Feldspar uses a deep embedding approach with a two-stage compilation process to generate C code. The language is extensible through the use of shallow embedding where new language constructs and combinators can be created with the use of existing ones. D'ARTAGNAN uses a similar approach as Feldspar with deep embedding and a two-stage compilation to generate target specific code. Unlike Feldspar, in D'ARTAGNAN we slice application logic into code to run on multiple devices at the same time and including intra-device communication.

There are also languages designed with the notion of synchronous dataflow computation, in the same spirit as D'ARTAGNAN. Lucid [Wadge and

Ashcroft, 1985] is a general-purpose functional language using this style, where a programmer would write a program where an endless stream of values are passed through operators to generate an output. More importantly, Lustre [Halbwachs et al., 1991] and Signal [Gautier et al., 1987] are languages which follow in the nature of Lucid but were specifically designed for the control of real-time systems. Unlike D'ARTAGNAN, Lustre and Signal are not embedded in another language and the main focus is on reactive systems rather than taking a macroprogramming approach.

5.9 Conclusions

We have proposed D'ARTAGNAN, an embedded DSL framework that brings functional programming to distributed embedded systems. By using an internal representation, we can analyse, transform and interpret a stream processor in different ways. D'ARTAGNAN allows the programmer to use the power of functional programming to build sensor network applications. For the use-cases evaluated in this work, we have observed 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 higher layers of abstraction. Libraries can be created for specific application domains to make writing of applications even easier.

The approach 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.

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 [Sheeran, 2005].

Part III

Macroprogramming for Blockchain Systems

Part II has shown how the technique of macroprogramming can be applied to stream processing applications on heterogeneous wireless sensor networks. We now shift focus onto blockchain systems. First, we extend the D'ARTAGNAN framework and language to extend beyond wireless sensor networks by including blockchain for stream processing applications. Then we propose a new framework and language called PORTHOS for macroprogramming commitment-based smart contracts.

6

Background: Blockchain and Smart Contracts

6.1 Introduction

The recent rise in popularity of cryptocurrencies has attracted widespread interest to blockchain technology, a type of Distributed Ledger Technology (DLT). A blockchain is a shared, distributed ledger made up of a log of immutable and verifiable list of transactions. Transactions are cryptographically signed instructions by users of a blockchain system. In a cryptocurrency application, the user's instructions indicate the transfer of cryptocurrency from one user to another. However, blockchain platforms can be used beyond transfer of cryptocurrency, as code (*smart contracts*), can be executed to perform computation by the blockchain system.

This chapter gives an overview of blockchain technology and smart contracts. This account is intended to equip the reader with the necessary background information for the other chapters in this Part. Chapter 7 describes the extension of the D'ARTAGNAN framework to include a blockchain element. In chapter 8 we shift focus entirely onto blockchain systems and smart contracts, as we propose a macroprogramming framework to write applications that span different blockchain systems.

6.1.1 Overview

Blockchain technology originated in Bitcoin [Nakamoto, 2008] in 2008 as the underlying technology for a cryptocurrency application where transactions are stored in a ledger shared between all participants and which is not controlled by any single entity. The approach is considered revolutionary in that it resolves the age-old problem of trust between trustless parties, without the involvement of a central institution. Transactions are stored in a list of blocks, and cryptographic techniques are used to ensure that the information stored is not altered. The chain grows continuously as so-called miners mine new blocks when new transactions happen on the network. To ensure that only valid transactions are added to the chain, a consensus algorithm is used where peers agree to the validity of blocks and transactions.

Since blockchain can be used to perform transactions without a bank or intermediary, blockchain fits quite well in the domain of financial services. It also has uses in other fields and applications where immutability, distribution and reliability of data is required. Businesses may use blockchain technology to remove a single point of failure, or to share data with other businesses without the need of a central authority to act as intermediary.

Blockchain technology is nowadays widely used for smart contracts — an agreement between two parties, written in code, which is executed on the blockchain when certain conditions are satisfied. Smart contracts help overcome the lack of trust that exists between two or more parties engaged in a trade — it enables business transactions that would otherwise not happen without the assistance of a trusted intermediary.

Smart contracts originate in Bitcoin scripts, written in a minimal non-Turing complete bytecode language. These scripts are mostly too restricted to be considered full smart contract languages. On the other hand, second generation blockchain systems such as Ethereum, offer a fully-fledged virtual machine with a Turing-complete instruction set.

Today's smart contracts are intended to execute on a single blockchain system. It is expected that the need for multi-chain distributed applications (DApps) spanning across multiple blockchain systems is going to increase

as blockchain technology continues to gain popularity. Different blockchain systems will co-exist to offer different features, or to provide different benefits. Interactions between blockchains (interoperability) will be needed to implement new types of applications where assets may be exchanged between participants across different blockchain systems.

6.2 Blockchain Technology

In a blockchain system, a number of nodes form a peer-to-peer network, and a consensus algorithm is used to ensure that all nodes have the same shared ledger information. In this section we describe the architecture of a blockchain system and we also briefly describe smart contracts. A variety of smart contract languages are covered in more detail in Section 6.4.

6.2.1 Blockchain Architecture

Blocks in a blockchain are linked together through cryptography — every block includes the hash of the previous block, such that a chain of blocks is created. The very first block, called the genesis block, does not have a parent block and this is usually hardcoded into the software of a blockchain application. If a transaction in any block is modified, the hash for that block is modified, affecting all subsequent blocks. This allows tampering with data to be immediately detected by the peers of the blockchain system. Figure 6.1 illustrates an example of a blockchain, which is made up of a block header and a block body.

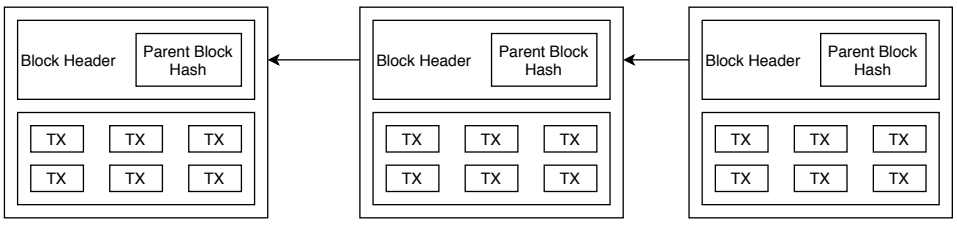

Figure 6.1: Blockchain architecture

The header of a block is typically made up of a number of elements, including:

- A hash of the parent block, creating a chain of connected blocks.
- A merkle tree root hash — a hash value of all the transactions in the block.
- The block timestamp.
- A version number indicating which blockchain consensus rules are being followed. The rules of a blockchain may be altered over time, so validators need to know which rules were applied for appending a new block to the chain.
- A nonce — a number adjusted by miners such that when included with all the other information in the block header and hashed, will generate a hash for the block which is equal to or less than a target hash value.

A number of different blockchain platforms exist and these differ by a number of key properties — the consensus protocol, the access policy and the validation policy.

Consensus Protocol — determines how trust is created among the peers of a blockchain system. A number of protocols exist, with the primary two being *proof of work* and *proof of stake*. In proof of work, miners spend a significant amount of energy to find a nonce that when hashed with the block information creates a hash value which satisfies certain criteria. This protocol was first proposed and used in the Bitcoin [Nakamoto, 2008] blockchain platform. Due to the nature of the protocol, this creates performance issues due to the limited throughput in terms of transactions processed by the blockchain system. Proof of stake requires nodes to put their own cryptocurrency at stake as a guarantee against bad behaviour. Proof of stake is used in the Cardano [car, 2018] blockchain platform. Ethereum [Wood, 2014], probably the

most popular blockchain platform, is currently using proof of work, but is reviewing a proposal to move to proof of stake. Other protocols include *proof of elapsed time*, *proof of importance*, *proof of state*, *raft-based consensus* and *stream-processing ordering service*.

Access Policy — determines which nodes can participate in a blockchain network. A *public* blockchain allows anyone to join and access the information stored in the blockchain, whereas a *private* blockchain can only be accessed by selected nodes.

Validation Policy — used to determine which nodes can participate in the consensus protocol and to initiate transactions. *Permissionless* blockchains allow any node to participate, whereas *permissioned* blockchains restrict these activities to selected nodes only.

6.2.2 Smart Contracts

A smart contract is a program that runs on a blockchain platform. The correct execution of a smart contract is enforced by the peers of the blockchain platform by using the agreed consensus protocol. Each blockchain platform typically supports one or more programming languages that can be used to write contracts. Rules and events of the contract are encoded in the programming language to implement a wide variety of applications.

The code of a smart contract is stored on the blockchain, and it can be identified and invoked with an address. Users can interact with a smart contract by invoking functions. Depending on the type of platform, it may be necessary to send cryptocurrency to pay for the execution of the smart contract function. This concept is typically referred to as gas, where the participant invoking the function has to pay for gas needed to execute that function.

Smart contract languages differ from one another in terms of expressivity. Some languages are intentionally restricted to reduce the risk of errors or non-terminating programmes. Other languages are Turing complete, allowing any application to be coded at the expense of increased complexity.

6.3 Blockchain Systems

In this section we outline a number of popular blockchain systems including Bitcoin, Ethereum and Hyperledger Fabric and highlight their key characteristics.

6.3.1 Bitcoin

Bitcoin [Nakamoto, 2008], launched in January 2009, was the first application to use blockchain technology. Bitcoin's main purpose is to serve as a digital currency, and the technology behind it prevents double-spending by allowing anyone to have a copy of the shared ledger and to validate all past transactions.

Although still popular today, the Bitcoin blockchain has a number of limitations. The transaction throughput is low (due to restricted block size) and the consensus algorithm is expensive based on proof-of-work causing transactions to take long to be confirmed.

We included Bitcoin in this section because we felt the review would not be complete without it. However, Bitcoin is only a digital currency application and the underlying network cannot be used for other applications as can be done with other blockchain platforms.

6.3.2 Ethereum

To overcome the limitations in Bitcoin (some of which were intended by design), a new generation of blockchain systems emerged. The Ethereum Virtual Machine (EVM) [Wood, 2014] is a simple but powerful Turing-complete virtual machine on which EVM byte code can be executed. Ethereum is a smart contract platform which supports stateful contracts where values persist on the blockchain to be used in multiple invocations. To avoid non-terminating computation, Ethereum introduced the notion of gas (a unit of consumption), where participants pay for computation performed on the

network. If gas runs out, the transaction is aborted, but fees are retained by the miners of the network.

As a result of the richer set of operations, Ethereum is much more powerful than Bitcoin and supports a wider range of applications. Ethereum is a permissionless blockchain system — anyone with access to the network can validate and submit transactions. A public instance of the Ethereum blockchain exists, and anyone can connect to it. The same technology can also be deployed in a private setup (restricted access) such that the information in the ledger is only accessible to a private consortium.

6.3.3 Hyperledger Fabric

Hyperledger Fabric [Cachin, 2016] is an implementation of a distributed ledger platform onto which smart contracts can be executed. It is a permissioned blockchain system with immediate finality — validating peers in the network are responsible for running consensus, validating transactions and maintaining the ledger.

Participants interact with Hyperledger through three types of transactions. The *deploy transaction* installs chaincode (a smart contract) on peers and makes it ready to be invoked. The *invoke transaction* invokes previously deployed chaincode with specific parameters and the result of the execution is returned. The third type of transaction is a *query transaction* where the state and result of a transaction can be retrieved from a peer's persistent state.

6.3.4 Others

Several other blockchain platforms exist or are currently in development stage, and the list continuously grows longer. Some of the more popular platforms on the list include Cardano, EOS, Tezos, Ripple, R3 Corda, Lisk, NEO and Stellar. The distinguishing factors between one platform and another may in some cases be quite radical while for others these are minimal — one blockchain platform soon becomes the predecessor of another as a

team decides to take the technology in a slightly different direction. For example, æternity[1] claims to improve a number of shortcomings on Ethereum including the introduction of state channels, improvement on governance, a hybrid proof-of-work/proof-of-stake consensus algorithm and the incorporation of oracles[2].

6.4 Smart Contract Programming Languages

To make it easier and familiar for programmers, the more popular smart contract languages (such as Solidity[3] for Ethereum) use an imperative-style programming paradigm where contract intermediate state is managed explicitly by the programmer. Solidity is an unrestricted language and computation is only limited by gas — a unit of consumption, where participants pay for computation performed on the network. The benefits of using such a paradigm (i.e. ease of use) are far outweighed by the risk of bugs unknowingly introduced by the programmer — in particular, unhandled scenarios which the programmer did not expect to happen. This is a risk that exists also for normal systems, but with smart contracts such risks can have much bigger impact — the DAO hack and other high-profile heists are evidence of this [Luu et al., 2016].

Different programming paradigms have emerged to address the risks associated with unrestricted languages such as Solidity. Explicit state transition languages, including Scilla [Sergey et al., 2018], Rholang [Meredith et al., 2018], Bamboo [Hirai, 2018] and Obsidian [Coblenz, 2017], use concepts from finite state machines and automata. Transactions either change the state of a contract or fail with an error. Re-entrancy is not allowed, and external calls that change the state are not possible except for tail-calls — thereby reducing the risk of attacks. Functional programming paradigms are used in Vyper [Buterin, 2018], Simplicity [O'Connor, 2017], Bamboo [Hi-

[1] https://aeternity.com/

[2] An oracle is a third party agent that feeds verified real-world information into a blockchain system to be used by smart contracts

[3] https://solidity.readthedocs.io/en/latest/index.html

rai, 2018] and Pact [Popejoy, 2016]. Functions are designed to be atomic (execute in their entirety or revert completely) and can call other pure functions (i.e. state is not changed).

Other techniques introduced to address the risks of unrestricted languages include the use of DSLs — a high-level language designed to work in a specific field or domain. Two main approaches exist (i) an intrepreter-type approach such as Findel [Biryukov et al., 2017] where an Ethereum smart contract is used to execute Findel contracts and (ii) a compiler-type approach, where a contract written in a DSL generates code in existing smart contract languages as used by Pettersson and Edström [Pettersson and Edström, 2016], Frantz and Nowostawski [Frantz and Nowostawski, 2016] and in Marlowe [Seijas and Thompson, 2018].

In this section, we will look at a selection of smart contract languages spanning across a range of different programming paradigms. Since not all languages offer the same functionality, we use different examples to illustrate capabilities.

6.4.1 Bitcoin Script

Bitcoin Script, the programming language used to write smart contracts[4] for Bitcoin, has a limited set of operations and was intentionally designed as non-Turing complete. In Bitcoin, a smart contract is implemented by defining a set of rules that must be satisfied for a value to be spent — this is referred to as the Unspent Transaction Output (UTxO) model. For example, a hash of the spender's public key must be provided to spend the value stored. Due to the restricted nature of Bitcoin Script, the applications that can be implemented on Bitcoin are limited.

Bitcoin Script is a stack-based language similar to Forth [Moore and Leach, 1970]. The code below illustrates a Bitcoin Script example to pay to a public key hash (P2PKH) address:

[4]We use the term *smart contract* loosely for Bitcoin, due to the restricted set of operators available in Bitcoin Script.

Listing 6.1: Pay-To-Public-Key-Hash (P2PKH) in Bitcoin Script

```
<Sig> <PubKey> OP_DUP OP_HASH160 <PubKeyHash>
OP_EQUALVERIFY OP_CHECKSIG
```

The code is processed from left to right, with operators added to the stack in a last-in first-out manner. The signature is first moved to the stack, followed by the public key. The `OP_DUP` operator duplicates the top item on the stack (i.e. the public key) and then the `OP_HASH160` function hashes the same top item on the stack (i.e. hashing the public key). Next, the public key hash required to unlock the funds is pushed to the top of the stack and the `OP_EQUAL` operator checks whether the top two elements of the stack (i.e. the two hashes) are the same. Finally, `OP_CHECKSIG` verifies whether the signature is correct. If the remaining element on the stack is `true`, then the funds can be spent.

The example shown for Bitcoin Script is relatively simple due to the restricted nature of the language. While in the other sections we show examples of auctions or escrow agreements, these type of applications cannot be expressed with Bitcoin Script.

6.4.2 Solidity

Solidity [sol, 2019], the most popular language on Ethereum, is a high-level language for writing smart contracts. Similar to object-oriented programming, contracts are like classes — contracts have functions which can call other functions in the same contract or in other contracts; contracts can be abstract or can inherit from other contracts; the contract intermediate state is managed explicitly by the programmer. Solidity compiles to EVM bytecode — a Turing-complete programming language with a stack, random access memory and persistent storage. Infinite loops are prevented through the use of gas, paid for by the participant invoking a function to the miner who processes the transaction in return for the computation. If gas runs out, the transaction is discarded, but the miner keeps the gas payment. The following code shows an auction example written in Solidity.

Listing 6.2: An auction example in Solidity

```
contract SimpleAuction {
    address public beneficiary;
    uint public auctionEnd;
    address public highestBidder;
    uint public highestBid;
    mapping(address => uint) pendingReturns;
    bool ended;
    event HighestBidIncreased(address bidder, uint amount);
    event AuctionEnded(address winner, uint amount);

    constructor(uint _biddingTime, address _beneficiary) public {
        beneficiary = _beneficiary;
        auctionEnd = now + _biddingTime;
    }

    function bid() public payable {
        require(now <= auctionEnd, "Auction already ended.");
        require(msg.value > highestBid, "There already is a higher bid.");
        if (highestBid != 0) {
            pendingReturns[highestBidder] += highestBid;
        }
        highestBidder = msg.sender;
        highestBid = msg.value;
        emit HighestBidIncreased(msg.sender, msg.value);
    }

    function auctionEnd() public {
        require(now >= auctionEnd, "Auction not yet ended.");
        require(!ended, "auctionEnd has already been called.");
        ended = true;
        emit AuctionEnded(highestBidder, highestBid);
        beneficiary.transfer(highestBid);
    }
}
```

Solidity gained widespread attention in its relatively short history as it was intentionally designed to look like JavaScript, and was therefore very

familiar to mainstream developers. On the other hand, the unrestricted nature and ambiguities of the language created opportunity for weaknesses in smart contracts to be exploited. The language is continuously evolving with changes being applied to improve the security and auditability aspects of the language.

6.4.3 Marlowe

Marlowe [Seijas and Thompson, 2018] is a domain specific language embedded in Haskell, intended for the Cardano blockchain platform but can also be implemented for other blockchain systems including both UTxO or account-based blockchains. The language is made up of a small set of basic contract constructs (seven basic constructs) which can be combined with Haskell primitives for additional expressivity. The language is compositional, in that complex contracts are built by connecting simpler contracts. Marlowe can be used to create financial contracts, in the style of Peyton Jones [Peyton Jones et al., 2000].

The following code sample shows the implementation of an escrow agreement in Marlowe.

Listing 6.3: Escrow agreement in Marlowe

```
escrow :: Contract
escrow = CommitCash iCC1 1 (ConstMoney 450) 10 100
                    (When (OrObs (two_chose alice bob carol 0)
                                 (two_chose alice bob carol 1))
                        90
                        (Choice (two_chose alice bob carol 1)
                            (Pay iP1 alice bob (AvailableMoney iCC1) 100
                                redeem_original)
                            redeem_original)
                        redeem_original)
                    Null
```

In the example above, the primitive `CommitCash` allows a user to commit a cryptocurrency amount of 450 (Cardano blockchain uses the ADA cur-

rency) before block 10, with the promise that money will be released on block 100 if it is not claimed before that. The next primitive, `When`, waits for two out of three participants (`alice`, `bob` and `carol`) to agree on the outcome — whether the money should be claimed by `alice` or by `bob`. If agreement is not reached by block 90, the money is refunded.

6.4.4 Others

Ivy is a high-level language [ivy, 2017] that allows a programmer to write smart contracts for the Bitcoin platform. An Ivy programme gives better form to a smart contract with structured clauses for unlocking value. A contract written in Ivy can be compiled to Bitcoin Script, to run on Bitcoin. In Section 6.4.1 we showed the Bitcoin Script code for paying to a public key hash address. The equivalent contract written in Ivy is shown here.

Listing 6.4: Pay-To-Public-Key-Hash (P2PKH) example in Ivy

```
contract LockWithPKH(pubKeyHash: Sha256(PublicKey), val: Value) {
  clause spend(pubKey: PublicKey, sig: Signature) {
    verify sha256(pubKey) == pubKeyHash
    verify checkSig(pubKey, sig)
    unlock val
  }
}
```

Simplicity [O'Connor, 2017] is a typed, combinator-based, functional language for building smart contracts. It is designed as a low-level language to be interpreted by blockchain software. The language is made up of just nine combinators to build expressions: `unit` returns the singular value of the unit type; `injl` and `injr` create tagged values while `case` is a branching operation; `pair`, `take` and `drop` create and manage pairs; `iden` is the identity function and `comp` is used for functional composition. Simplicity is purely functional and has no state so as to facilitate equational reasoning about the semantics of the expression.

The language has no bound variables and no function types, hence no higher-order functions. The language is Turing incomplete, and has no recursion or loops making it amenable to static analysis to determine the upper bounds of computation resources required.

Listing 6.5: Basic signature verification in Simplicity

```
basicSigVerify b c := comp (pair(witness b)
        (pair pubKey (comp (witness c)sighash)))
                        (comp (pair checkSig unit) (case fail unit))
```

Liquidity [liq] is a high-level language for writing smart contracts on the Tezos platform. The language is a fully-typed functional language and compiles to Michelson — a low-level, strongly-typed, stack-based language. Values can be stored in local variables, instead of more complex stack manipulations and the language supports high-level types like sum-types and record-types. The code below shows an example of an auction smart contract written in Liquidity.

Listing 6.6: An auction example in Liquidity

```
type storage = {
  auction_end : timestamp;
  highest_bid : tez;
  bidder : key_hash;
}

let%entry main
    (parameter : key_hash)
    (storage : storage) =

  (* Check if auction has ended *)
  if Current.time () > storage.auction_end then Current.failwith ();

  let new_bid = Current.amount () in
  let new_bidder = parameter in
  (* Check if new bid is higher that the last *)
  if new_bid <= storage.highest_bid then Current.failwith ();
```

```
  let previous_bidder = storage.bidder in
  let previous_bid = storage.highest_bid in

  (* Set new highest bid in storage *)
  let storage = storage.highest_bid <- new_bid in
  let storage = storage.bidder <- new_bidder in

  (* refund previous bid to previous bidder *)
  let refund_to = Account.default previous_bidder in
  let op = Contract.call refund_to previous_bid () in
  ([op], storage)
```

Sophia is a smart contract language for the æternity blockchain. It uses the functional programming paradigm and is a strongly-typed language. Sophia has restricted mutable state for each contract instance — state is enclosed within the `state` type. Anything outside this state is not persisted between contract invocations. A contract initialisation function is pure, in that it returns the initial state as the return value and then on, state is accessible through an implicitly bound variable `state`. Sophia contracts can call other contracts, and the structure of a contract is similar to classes in an object oriented language. Contract functions can be abstract, and contracts can inherit from other contracts.

Listing 6.7: An auction example in Sophia

```
contract Auction =
  type state = { start_amount : int, start_height : int,
                 dec : int, beneficiary : address, sold : bool }

  private function abort(err) = abort(err)
  private function spend(to, amount) =
    let total = Contract.balance
    raw_spend(to, amount)
    total - amount

  private function require(b : bool, err : string) =
```

```
  if( !b ) abort(err)

public function init(beneficiary, start, decrease) : state =
  require(start > 0 && decrease > 0, "bad args")
  { start_amount = start, start_height = Chain.block_height,
    beneficiary = beneficiary, dec = decrease, sold = false }

public stateful function bid() =
  require( !(state.sold), "sold")
  let cost =
    state.start_amount - (Chain.block_height - state.start_height)
    * state.dec
  require( Contract.balance >= cost, "no money")
  spend(state.beneficiary, cost)
  spend(Call.caller, Contract.balance)
  put(state{sold = true})
```

Vyper [Buterin, 2018] is a high-level smart contract language for Ethereum, and like Solidity compiles down to EVM bytecode. Vyper is syntactically similar to Python, and like Solidity, is contract-oriented. However, Vyper was designed to address specific goals of security and auditability. A number of features were intentionally left out, including recursive calling and infinite loops such that gas limit attacks are not possible. Also, modifiers, class inheritance and function overloading are not possible in order to improve the readability, and hence auditability, of a smart contract. An example of an auction is shown below.

Listing 6.8: An auction contract example in Vyper

```
# Open Auction

# Auction params
# Beneficiary receives money from the highest bidder
beneficiary: public(address)
auctionStart: public(timestamp)
auctionEnd: public(timestamp)
```

```
# Current state of auction
highestBidder: public(address)
highestBid: public(wei_value)

# Set to true at the end, disallows any change
ended: public(bool)

# Keep track of refunded bids so we can follow the withdraw pattern
pendingReturns: public(map(address, wei_value))

# Create a simple auction with `_bidding_time`
# seconds bidding time on behalf of the
# beneficiary address `_beneficiary`.
@public
def __init__(_beneficiary: address, _bidding_time: timedelta):
    self.beneficiary = _beneficiary
    self.auctionStart = block.timestamp
    self.auctionEnd = self.auctionStart + _bidding_time

# Bid on the auction with the value sent
# together with this transaction.
# The value will only be refunded if the
# auction is not won.
@public
@payable
def bid():
    # Check if bidding period is over.
    assert block.timestamp < self.auctionEnd
    # Check if bid is high enough
    assert msg.value > self.highestBid
    # Track the refund for the previous high bidder
    self.pendingReturns[self.highestBidder] += self.highestBid
    # Track new high bid
    self.highestBidder = msg.sender
    self.highestBid = msg.value
```

6.4.5 Discussion

The primary differentiator between different smart contract languages is whether the language is Turing-complete or not. A Turing-complete language has more expressivity to define a wider variety of smart contracts, at the cost of sacrificing readability and thereby security of contracts. This is an ongoing debate as some languages (such as Solidity) are considered to be more error-prone and security flaws could lead to big financial losses, when compared to other languages which are more restricted. In Jansen et al. [2019], the authors find that only around a third of smart contracts on the Ethereum blockchain system make use of while-loops, for-loops and recursion. This indicates that the majority of contracts do not need the expressivity of a Turing-complete language, but a more restricted language should suffice.

Other differences between languages include the choice of programming paradigm — whether an imperative sequential paradigm is used, or a declarative one. The former provides better familiarity with developers experienced with popular languages such as Javascript, whereas the latter may be more appropriate for domain-specific areas. For smart contracts, declarative languages without side-effects (or with controlled side-effects) may be preferred as these can reduce security risks.

6.5 Chain Interoperability

Today's smart contracts are intended to execute on a single blockchain system, however it is expected that with the proliferation of blockchain systems, the need for multi-chain distributed applications (applications that span across multiple blockchain systems) is going to increase. Different blockchain systems will co-exist to offer different features, or to provide different benefits. Interactions between blockchains (interoperability) will be needed to implement new types of applications where assets may be exchanged between participants across different blockchain systems. A sin-

gle application may handle payments on the Bitcoin network, Ethereum for public interactions and Hyperledger Fabric for specific private point-to-point interactions.

Blockchain interoperability is not straightforward. Vitalik Buterin [Buterin, 2016] identifies three strategies for chain interoperability — atomic swaps using hashed time-locks, relay chains and centralised or multisig notary schemes.

Hashed time-locks are ideal for the swapping of assets across different blockchain systems. Operations on two chains use the same trigger — a hash of a secret, which is then revealed and used to unlock the transactions in sequence. Strategies involving relays or notaries both rely on the presence of a trusted entity, or group of entities. With relays, blocks are copied from one blockchain system to another and the receiving blockchain has the capability of validating and inspecting incoming blocks to trigger actions as needed. In notary schemes, a trusted entity triggers an operation on a blockchain when an event is detected on another blockchain.

Blocknet is a solution for interoperability which makes use of atomic swaps. The XBridge component enables an exchange functionality between two blockchains, and the XRouter component enables communication functionality. Solutions based on atomic swaps are not suitable for causality — for example, when an event on Blockchain A triggers an action on Blockchain B.

Cosmos [Kwon and Buchman, 2018] and Polkadot [Wood, 2018] are two of the top contenders in the blockchain interoperability space, aiming to create a network of blockchains by using a relay-based strategy — blocks from one chain can be read and verified by another chain. The advantage of a relay-based approach is that the system is completely trustless — the relay contract is publicly auditable and anyone can relay blocks. However, the weakness of the approach is that the amount of data required on the destination chain may be quite significant and with block size limitations this may become problematic over the long term.

Aion's Transwarp-Conduit [Shidokht Hejazi-Sepehr and Sharif, 2019] is

a notary-based approach for connecting distinct public blockchain systems. Notary schemes offer better versatility than relay-based schemes, however the presence of an external entity (or entities) is required and therefore trust concerns may be introduced.

None of the frameworks mentioned here offer the perfect solution for interoperability, and this remains an active area of research. One of the more advanced interoperability projects is the Interledger [Thomas and Schwartz, 2015] protocol, which is intended to enable the exchange of value between different systems. The protocol does not rely on a single system for processing payments, and anyone with accounts on two or more ledgers can act as a connector.

The scope of blockchain interoperability in this thesis is limited to the requirement of a network layer for communication between different blockchain systems. We do not attempt to provide a solution to blockchain interoperability, but simply highlight the challenges which are currently present in this domain and which currently remains an open research question.

6.6 Conclusions

In this chapter we have described blockchain technology and smart contracts. We outlined a number of popular blockchain systems, including Bitcoin, Ethereum and Hyperledger Fabric, and their key characteristics. We have also shown a variety of smart contract languages, including short snippets of smart contracts in different languages. Finally, we described the challenges and potential solutions for chain interoperability. This chapter intended to provide the reader with information about blockchain technology and smart contracts as background for the next chapters (Chapters 7 and 8).

7

Macroprogramming the Blockchain of Things

In Chapter 5 we proposed D'ARTAGNAN, an embedded DSL framework for macroprogramming distributed embedded systems. In this chapter we extend the heterogeneity aspect of that framework to be able to write stream processor descriptions including blockchain and edge systems, in addition to embedded systems.

Blockchain and smart contract technology provide a means of decentralised computational agreements that are trusted and automated. By integrating Internet of Things (IoT) devices with blockchain systems and smart contracts, agreements can not only be confined to in-blockchain manipulation of state, however can enable agreements to interact on the physical world. This integration is non-trivial due to the limited resources on IoT devices and the heterogeneity of such an architecture. Such blockchain connected IoT devices typically require programming of smart contracts, edge blockchain nodes and the IoT devices.

IoT embedded systems require expertise in low level development. Similarly, smart contract programming requires expertise with an extensive attention to detail, as even minor bugs can have catastrophic consequences. In this chapter, we propose a macroprogramming approach, as an extension of the D'ARTAGNAN framework, for developing the different system components required for blockchain connected IoT devices including smart

contracts, edge nodes and IoT devices from a monolithic description. In this manner, one can use a higher level of abstraction to develop an application, while still being able to generate code automatically which can be deployed on different nodes.

7.1 Introduction

The rise of smart contracts on blockchain or other distributed ledger technologies have enabled the possibility of regulated interaction and resource exchange between parties without the need of a trusted entity. With smart contract technologies such as Ethereum [Wood, 2014], which provide a Turing complete programming language for the specification of executable contracts, one can encode any complex behaviour between parties within the contract. A major challenge is, however, that of reaching beyond the confines of the smart contract itself, and interacting with real world systems. This challenge is particularly acute in cases where external systems are Internet of Things (IoT) devices which are often limited in resources.

One of the hurdles is the fact that programming models (and virtual machines) developed for smart contracts are not designed to be executed on systems with limited resources. For instance, the Ethereum Virtual Machine uses 256-bit instructions and associated stack, which cannot be easily deployed effectively on most limited resource devices without major overheads of space and time. Other work has looked at providing means to explicitly switch word-size in order to have virtual machine-level code executable across blockchain and IoT devices transparently [Ellul and Pace, 2018].

However, this does not address the other major challenge of developing applications at a high level of abstraction across the two domains. Particularly due to the fact that smart contracts regulate transfer of digital assets (and particularly frequently used to move cryptocurrency), system correctness is critical, as has been shown in cases where the equivalent of mil-

lions of US dollars were lost due to bugs in smart contracts[1]. The need for a unified view of the system across the different levels of abstraction and different locations of deployment (one of possibly many IoT devices, smart contract or an intermediate blockchain edge node) is crucial, since programming the different layers separately and gluing things together with custom communication is not straightforward and may easily lead to unforeseen situations.

On IoT devices, this challenge is typically addressed through the use of macroprogramming [Newton et al., 2007a] — in which programmers can focus on the top-level, global view and goal of the application being developed, hiding low-level details such as deployment location, communication, etc. In this chapter, we propose to extend such an approach to reach beyond the computation and sensor data acquisition on IoT devices, thus enabling parts of the macroprogrammed system to be deployed on edge devices and beyond — in our case as smart contracts on the blockchain.

A number of application areas have been identified in the literature as good use-cases for combining blockchain technologies with Internet of Things devices, including transportation (smart vehicle systems, vehicle-to-vehicle networking and intelligent transportation systems), transactive energy (microgrids and energy markets), smart cities and smart homes, communication between drones and robots, and manufacturing and supply-chain models [Abadi et al., 2018].

7.1.1 D'ARTAGNAN for Blockchain of Things

D'ARTAGNAN (as proposed in Chapter 5) is a macroprogramming language aimed at enabling the programming of stream processing systems to be deployed on heterogeneous devices, primarily targeting low-level devices. However, nothing prevents it from being extended to high-level systems. In this chapter, we discuss how D'ARTAGNAN can be extended to push the limits of heterogeneity to edge devices and even onto smart contracts on the

[1] See [Atzei et al., 2017] for a list of cases, although since its publication many other high profile, high loss cases have happened

blockchain. We identify a number of challenges which need to be addressed to make such an approach possible, namely (i) enabling the deployment of stream processors beyond IoT devices, particularly to enable in-blockchain computation (in the form of smart contracts), and (ii) incorporating communication between the different levels of abstraction, invisible to the user. Solutions for D'ARTAGNAN are proposed and a prototype enables us to evaluate how an end-to-end solution can be programmed for a smart building rent management use-case.

The rest of this chapter is organised as follows. In Section 7.2 we describe the architecture and workflow of our proposed solution. Then, in Section 7.3 we present D'ARTAGNAN and how it is extended to enable the framework proposed. We then present a use-case to illustrate the use of D'ARTAGNAN in such a context in Section 7.4, an evaluation in Section 7.5 and conclude in Section 7.6.

7.2 Proposed Framework

Blockchain connected edge IoT devices typically interact with the blockchain by either having their own local copy of the blockchain or using an intermediary node (such as a cloud-based service or edge blockchain node). Writing code for systems made up of such combinations of devices typically involves development of: (i) a smart contract, (ii) code to be deployed on blockchain edge nodes and (iii) code for the IoT devices (which may each use different technologies due to the heterogeneity of the devices). In addition, each of these devices has to handle communication with the others in an explicit manner. In order to reduce the complexity required to develop such systems, we propose the use of macroprogramming which enables the use of a single high-level application description (using a domain specific language approach) ranging over the whole system. Figure 7.1 depicts the system architecture.

A single macroprogram is written by the system implementer which is passed through transformations to generate the smart contract, blockchain

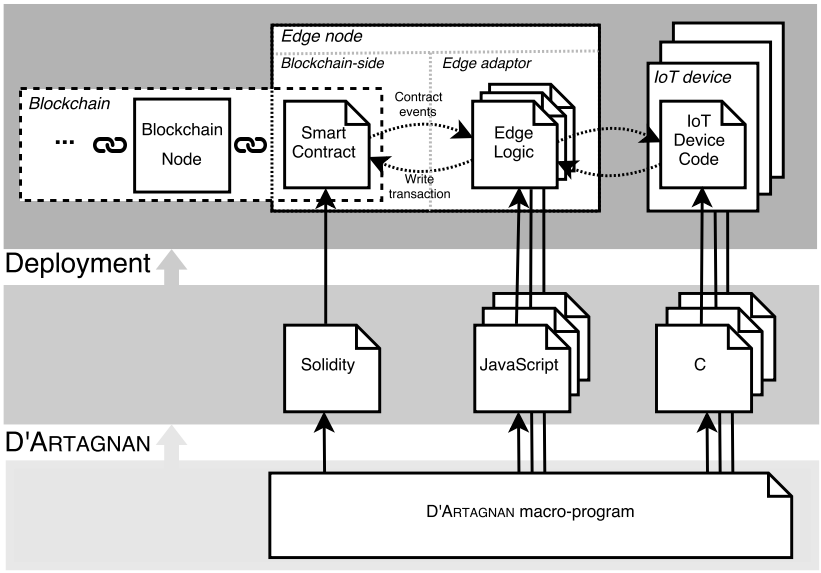

Figure 7.1: Proposed macroprogramming blockchain of things architecture

edge nodes and IoT device code. Every participating blockchain node requires a copy of the same blockchain data and is updated when new transactions occur. When a smart contract is to be deployed within the blockchain, each node will gain a copy of the smart contract. Actions can be initiated by monitoring smart contract events of interest within the blockchain edge node (this involves nothing more than monitoring the local state of the blockchain). Thereafter, the blockchain edge node can perform any required tasks and propagate messages throughout the different system components (be it IoT devices, other edge nodes or to the smart contract itself). Similarly, connected IoT devices can perform actions based upon the logic that is required of them. Such actions may involve propagating data back to the blockchain edge node in order to update the blockchain state.

7.3 D'ARTAGNAN: A Macroprogramming Framework

As we have shown in Chapter 5, D'ARTAGNAN is a framework for programming stream processing applications using a high-level domain specific language (DSL). The framework automatically translates a stream processor description into target code that can be run on a network of heterogeneous devices. The generated code is specific to each device in the network depending on both the intended behaviour and also the target architecture.

The D'ARTAGNAN language is embedded in Haskell — a domain specific embedded language, effectively a domain specific library developed in a style such that the use of the library results in parts of a system written in the host language (Haskell in our case) to resemble programs written in a DSL for the target domain.

In the case of D'ARTAGNAN, the library allows for stream processors to be defined as part of (and using) Haskell programs. Haskell primitives can be used with the DSL to raise the level of abstraction. Consider the following code snippet:

```
result = foldl1 combine (map (applyRate 10) eSensors)
```

A rate is applied to each sensor stream with `map`, and `fold` is used to aggregate results by using a specific `combine` function. For instance, in wireless sensor networks, where power utilisation is a precious resource, information can be aggregated before sent wirelessly to neighbouring nodes in order to reduce the amount of data traffic.

Internally, the description of a stream processor results in a representation which can be (i) analysed; (ii) transformed; and (iii) interpreted in different forms. These are the three key features of the D'ARTAGNAN framework (refer to Figure 7.2).

A stream processor can be analysed by traversing the internal representation to look for relevant and interesting information. For example, to determine how computation is distributed across the network and whether

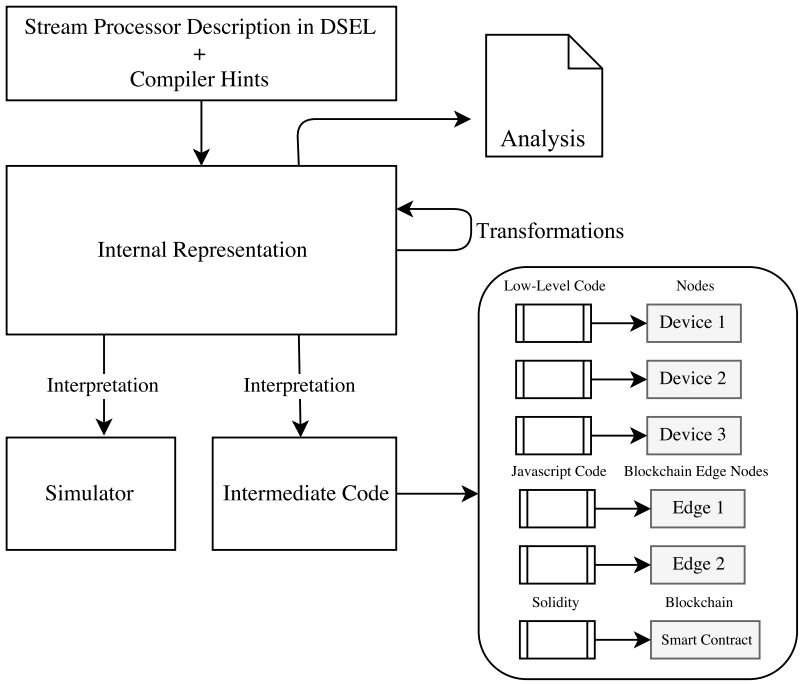

Figure 7.2: The D'ARTAGNAN framework.

one device is more loaded than others. For embedded devices, an even distribution is desirable as it typically increases the longevity of the application on these resource constrained systems.

The internal representation can also be transformed in different ways — for example to evenly distribute the computation across the network (as a result of the analysis phase) or perhaps to replace a computationally intensive mathematical function with an approximation function. The framework also supports compiler hints — tips supplied by the programmer to the compiler such that generated code is optimised for the given application environment. For example, if one of the devices in the network has a more powerful processor, the compiler attempts to shift computation on to this device.

The same internal representation can be interpreted in different forms. A simulator interpretation can be used to observe the behaviour of the stream processing application in a simulated environment. Perhaps more impor-

tantly, another interpretation automatically generates low-level code which can be loaded onto target devices — the generated code can be specific to different types of devices.

7.3.1 D'ARTAGNAN for IoT

D'ARTAGNAN was initially designed for IoT devices with limited capabilities and resources. The computation for a stream processor would thus be spread across different devices based on sensors they possess and computational power. The approach is that the developer typically partially tags which parts have to be deployed on which devices (e.g. due to the features, or the positioning, of a device — if we require the temperature of the kitchen, then we can only read it from a device which has a temperature sensor and which lies in the kitchen). The locality of the rest of the computation can be left up to D'ARTAGNAN, or directed using code transformation libraries which allow the developer to request certain constraints or compilation strategies (e.g. to minimise communication, or to put as much computation as possible on devices connected to a permanent power source).

The IoT device spectrum is very fragmented with hundreds of hardware platforms and several operating systems. Code which runs on a specific device cannot be executed on another device running the same operating system because the underlying hardware differences cannot be ignored. Because of these differences D'ARTAGNAN was designed with heterogeneous networks in mind, where the same logic can be translated into different target code depending on the target device.

7.3.2 Extending D'ARTAGNAN

The ability to generate code for different targets (heterogeneity) and the ability to transform and statically move computation logic across the devices in a network is what enables us to extend the D'ARTAGNAN framework to be applicable for applications that span across IoT devices and

smart contracts technology. Whereas the main concern with IoT devices is the preservation of energy and load balancing of computation, applications involved with let's say the Ethereum blockchain are typically concerned about gas[2] utilisation. The mix of these two concerns together with the need to have business logic visibility in smart contracts provide an interesting challenge for which we believe D'ARTAGNAN can contribute. Our framework allows the application logic to be placed in an improved manner according to information provided by the programmer (i.e. hints) and the target code generated accordingly.

Listing 7.1: D'ARTAGNAN language extensions

```
transaction :: Stream Int -> Stream Bool -> Stream Bool

native :: StreamType -> String -> Stream a

native1 :: StreamType -> String -> Stream a -> Stream b
```

In order to adapt D'ARTAGNAN to a wider range of applications, we have extended the language with new functionality (see Listing 7.1). To be able to implement blockchain applications, we have added `transaction` for the execution of blockchain transactions. Given a condition (in the form of a boolean stream) and an amount, a transaction is executed on a smart contract with the amount deducted from a balance. The result of the transaction function shows whether the transaction has been successful or not (in the form of a boolean stream). As an example, consider `app` below, which shows the listing of a simple app which given a sensor (e.g. luminosity sensor) will attempt to execute a transaction worth 1 coin[3] when the luminosity is below 150. The result of transaction is then used to turn on a light bulb.

```
app :: Stream Int -> Stream Bool
app input1 = transaction 1 (input1 .<. 150)
```

[2]The cost of running a transaction on the Ethereum blockchain varies according to the computational resources needed and is referred to as 'gas'.

[3]We use coin to mean the quantity or cost associated with the transaction.

With the introduction of a wider class of devices, some of which are not necessarily resource-constrained, new functionality has been introduced to reflect the new capabilities. For instance, native functions allow the application to get access to new functionality which now becomes available as a result of a wider class of devices. For example, a native function can be used to retrieve information from a database or a third party service. Native functions are defined directly inside a stream processor description and are executed directly on the devices where they will be deployed, an approach borrowed from a similar one used in Flask [Mainland et al., 2008]. To allow for a wide range of services, the content of the native function is not evaluated by D'ARTAGNAN. This means that errors may not be caught early and will be detected during the compilation to target, or possibly even at runtime. This approach is a compromise to allow for new services to be added easily without changing the D'ARTAGNAN core.

In addition, we have extended the capability of heterogeneity further by adding support in D'ARTAGNAN for compiling to Javascript for edge devices and to Solidity for deployment of parts of the system as smart contracts. These additions further require support for communication between the different types of devices e.g. through a serial port between an IoT device and a blockchain edge node, wireless communication between IoT devices running different operating systems, etc.

As mentioned at the beginning of this section and depicted in Figure 7.2, the D'ARTAGNAN framework allows transformations which also include placement of logic on specific targets. To illustrate the effect of this, we will use Listing 7.2 which extends our previous example app with two sensors. This example is trivial, but sufficient enough to illustrate the concepts of how different placements work.

Listing 7.2: app2

```
app2 :: Stream Int -> Stream Int -> Stream Bool
app2 x y = transaction 1 (sAvg (x, y) .<. 150)

>> generateCode (app2 (inputI 1) (inputI 2))
```

The application app2 is made up of six steps:

1. Read Sensor 1
2. Read Sensor 2
3. Add the values from the previous two steps
4. Divide by 2 the result from the previous step for average value
5. If the result is less than 150, execute a transaction
6. Transaction to deduct an amount from a balance

The code for reading a sensor is naturally bound to the device where the sensor is located — Steps 1 and 2 are placed on Devices 1 and 2 respectively. Step 6, which is used to deduct coin from a balance, is also location-bound and has to be placed in a smart contract. There is more flexibility in the placement of the remaining three steps and what goes where may depend on the application needs and the layout of the devices. We highlight 3 possible placement options with the respective advantages/disadvantages:

- **Placement Option 1 – IoT-focus**: In this configuration the code is placed (wherever possible) on IoT devices to provide in-network processing and filtering. In the example above, the code for steps 3, 4 and 5 would be placed on Device 1, such that the device determines whether to trigger a transaction or not. This approach requires less communication in the form of radio messages, and gas is only used when the condition is met. However, trust level is low as application logic is off-chain. [low comms, low gas, low trust]

- **Placement Option 2 – Edge-focus**: In this placement option, code is placed on the edge node where possible. Every time the sensors are polled, the readings are passed from the IoT devices to the edge node (2 wireless messages). The edge node then determines whether a transaction should be executed. The number of radio messages increase as a result, but this approach allows the edge node to do more

complex computation that may not be possible or accessible on resource constrained devices (e.g. record the information in a database or convert using a rate-conversion service). [high comms, low gas, low trust]

- **Placement Option 3 – Blockchain-focus**: With this option, the application logic (steps 3, 4 and 5) is placed in the smart contract for transparency. This setup also utilises two radio messages every time the sensors are polled. The information is recorded in the smart contract, but a transaction is only performed when the condition is satisfied. [high comms, high gas, high trust]

Since different applications may have different requirements, the framework allows the programmer to use placement directives (such as `onDevice`) to influence how business logic is placed across the network (see Listing 7.3), including the smart contract, as shown in the different placement options above. The different types of communication needed (IoT to IoT, IoT to Edge, Edge to Blockchain) to connect the logic in a coherent manner is handled under the bonnet.

Listing 7.3: Different placement directives

```
app2 :: DeviceNum -> Stream Int -> Stream Int
    -> Stream Bool
app2 placement input1 input2 = onDevice(placement,
    transaction 1 (sAvg input1 input2 .<. 150))

>> app2 IOT (inputI 1) (inputI 2)
```

In the future, compiler directives may be added, such as *low-comms*, *low-gas* or *high-trust*, such that the programmer does not need to explicitly indicate where application logic should be placed, but such placement is handled automatically by the framework.

7.4 Use Case: Smart Rent Management

In Section 5.5, we introduced the smart rent use-case where home-owners wanting to rent out their apartment can calculate a consumption charge for commodities including water, electricity and home appliances. This approach allows home-owners to lower daily rental rates to attract more bookings without being exposed to high utility bills or maintenance costs. In that Section, we showed how a metering application can be described to measure and calculate the cost from several home appliances. In this Chapter, we extend and complete that use-case to include a billing mechanism by deducting funds directly from a wallet residing on a blockchain system. If funds run out, appliances can be disabled and an automatic lock may possibly lock out the tenants until funds are replenished.

To illustrate the basic concepts of the D'ARTAGNAN framework we show a simple smart-rent application that calculates electricity consumption cost (see Listing 7.4). The same description, in our case the code describing the stream processor, can be used to generate different target code depending on utility rates and the devices the application will run on. The generality of this approach allows the same application to be used at different premises and where different rates may apply, and is by far easier to manage than an equivalent version written directly in low-level code.

Listing 7.4: A simple smart-rent application

```
consumption :: Int -> Stream Int -> Stream Bool
consumption eRate usage =
    transaction (liftS eRate .*. usage) true

rate :: Stream Int -> Stream Bool
rate input = consumption 10 input
```

A specific instance of this application is created with real sensors passed as input parameters such as:

```
» generateCode (rate (inputI 1))
```

Behind the scenes, D'ARTAGNAN creates low-level C code that will run

on device 1 for sensing the electricity consumption, generates code that will run on the blockchain edge node and Solidity for the smart contract. The complexity of communication between the IoT device, the edge node and the smart contract, and the placement of in-network computation is determined and handled by D'ARTAGNAN– thereby hiding away all the complexity of communication and placement from the programmer.

Listing 7.5: Snippet for blockchain edge node

```
port.on('data', function processData(data) {
  var usage = data;
  var rate = 10;
  var costConsumed = usage * rate;

  contract.methods.transaction(costConsumed).
    send({from:caller}).on('receipt',
      function (receipt) {
        if (receipt.events.TxEvent.returnValues.tx)
          console.log("Transaction successful");
      });
});
```

Using the same idea, the application can be further enhanced to also capture information directly from appliances and apply a charge to every washing-machine cycle, per-hour use of air-conditioning and so on. Listing 7.6 illustrates how such an application would be written. Since the D'ARTAGNAN DSL is embedded in Haskell, the programmer can use Haskell constructs, such as `fold` and `map`, as part of the stream processor description. The application can then be instantiated as follows (definitions for sensors have been omitted for conciseness):

Listing 7.6: Smart rent application with appliance dependent fees

```
consumption2 :: (Int, Int, Int, Int, Int, Int, Int)
    -> (Stream Int, Stream Int, Stream Int, Stream Int, Stream Int,
        [Stream Int], [Stream Int])
    -> Stream Int
consumption2
    (elecRate, waterRate, wmRate, dwRate, tdRate, acRate, tvRate)
```

```
    (elec, water, wm, dw, td, acList, tvList) = transaction totalCost true
  where
    totalCost = simpleRateCost elecRate elec .+.
                simpleRateCost waterRate water .+.
                simpleRateCost wmRate wm .+.
                simpleRateCost dwRate dw .+.
                simpleRateCost tdRate td .+.
                foldl1 (.+.) (map (simpleRateCost acRate) acList) .+.
                foldl1 (.+.) (map (simpleRateCost tvRate) tvList)

    simpleRateCost :: Int -> Stream Int -> Stream Int
    simpleRateCost rate usage = liftS rate .*. usage

rate2 = consumption2 (10, 5, 25, 25, 25, 15, 3)
```

```
>> toBlockChain (rate2 (elecMeter, waterMeter, washingmachine,
    dishwasher, tumbledryer, [aircon1, aircon2, aircon3], [tv]))
```

Our example can be enhanced further by applying rates which vary according to the time of day. The rates can be stored in a database attached to one of the blockchain edge nodes, and a conversion service can be used to convert units to actual cost (e.g. Euros). The sample code below shows how a database query can be executed on the edge node connected to the database, and then the conversion service is used to change the units into a common base currency.

```
dbResult = onDevice (edgeNodeDB, Pull,
    native1 dbQueryType "DBQuery:
    SELECT elecRate, currency FROM ElecRates
    WHERE timeOfDay=$1" t1)

(elecRateForeign, curr1) = unbundle dbResult

elecRate = elecRateForeign .*.
    onDevice (edgeNodeForex, Pull,
    native1 IType "Service: forex($1, 'EUR')" (curr1 :: Stream Int))
```

Consumption rates can be stored in the database and updated by an external party, thus producing a dynamic system which does not require to recompile and upload new code to the devices every time a rate changes. Database queries can be defined inside a `native` function. Similarly, external services such as forex rates follow the same pattern.

The function `native1` indicates that a native function with one parameter is being defined. Since it is not possible for the macroprogram to statically determine what the result of a native function is, the return type needs to be defined in the function definition. In the database example above, the result of the query is a stream tuple of two integers.

7.5 Evaluation

To assess the performance of D'ARTAGNAN, we compare code generated automatically against an equivalent version which is manually coded. We use the application defined in Listing 7.3, which behaves like a thermostat using two temperature sensors — an amount is deducted from a balance stored in a smart contract to switch on heating/cooling. We compared the code generated for each of the three placement options (1) IoT-focus; (2) Edge-focus; and (3) Blockchain-focus. Our aim is to answer two questions: (i) how the lines of code compare between hand-coded and automatically generated versions; (ii) how consumption is affected by the different placement options — gas for smart contracts; and radio messages and clock cycles for IoT devices, which is indicative of power consumption (we do not measure Edge devices as these are typically computers connected to a permanent power supply and computation is not restricted by battery-power or gas).

One line of D'ARTAGNAN code (specifically `app2`) generates between 200–250 lines of code (depending on placement) — C for IoT devices, Javascript for the edge node and Solidity for the smart contract. Table 7.1, as well as a visual inspection of the generated code, confirms that the automatically generated version is more verbose than a hand-coded one. However, for IoT

	IoT	Edge	Blockchain
Placement Option 1: IoT-focus			
Manually-Coded	150	25	35
Auto-Generated	200	25	35
Placement Option 2: Edge-focus			
Manually-Coded	143	44	35
Auto-Generated	191	65	35
Placement Option 3: Blockchain-focus			
Manually-Coded	143	25	48
Auto-Generated	191	25	60

Table 7.1: Lines of code comparison

devices, our evaluation in Chapter 5 had shown that basic GCC compiler optimisations almost completely eliminate any inefficiencies introduced by the automatic generation for IoT devices — so this should result in minimal overhead. On the other hand, at this time, Solidity does almost no optimisation on the generated code which leads to higher gas consumption.

Placement Focus	Blockchain Gas	IoT Devices	
		Radio Messages	Clock Cycles
IoT	1.114M	150	21,869K
Edge	1.114M	200	21,813K
Blockchain	1.240M	200	21,813K

Table 7.2: Consumption comparison for different placement options

To calculate consumption (Table 7.2), the experiment was designed with 100 iterations of the application for which 50% trigger a smart contract transaction. For situations where high trust is needed (more transparency via smart contract), a Blockchain-focus placement moves more code to the smart contract (see Section 7.3.2) and therefore more gas is consumed — partly due to more code being executed, as well as some code being executed even when the transaction condition is not triggered. Therefore, higher trust results in higher consumption for both gas and energy.

On the other hand, for lower trust scenarios where application logic can be placed off-chain, deciding between IoT-focus or Edge-focus depends on the combined energy utilisation of both radio messages and clock cycles. As expected, as application logic is placed away from IoT devices, more radio messages (50) and less clock cycles (roughly 56K) occur. One radio message consumes as much energy as 3 million instructions [Pottie and Kaiser, 2000]. In this example, an IoT-focus placement has lower consumption since application logic is simple. In the case of computationally intensive tasks (requiring millions of clock cycles per iteration), an Edge-focus placement will have an overall lower consumption than an IoT-focus — the complex computation moved away from the IoT devices makes up for the extra radio messages. In the future, improved placement of application logic across both IoT and Edge devices (for low-trust scenarios) can be introduced by taking into consideration both radio messages and computation complexity.

7.6 Discussion and Conclusions

In this chapter we have presented a macroprogramming approach to describe blockchain connected IoT devices and their interaction with smart contracts. This is to our best knowledge the first approach to attempt to provide a single macroprogramming description for such blockchain connected IoT devices. The closest work to that being presented here includes programmability of blockchain connected edge nodes using a control systems approach [Stanciu, 2017]. Other related work includes: IoT devices making use of the blockchain as a means of storing data [Huh et al., 2017]; defining virtual resources within IoT device firmware that can be instructed to execute a sequence of function invocations [Samaniego and Deters, 2016]; and making use of the blockchain to store IoT firmware [Boudguiga et al., 2017].

Several macroprogramming solutions for wireless sensor networks have been proposed over the past decade. Regiment [Newton et al., 2007a], Wave-

script [Newton et al., 2008] and Flask [Mainland et al., 2008] are closest to D'ARTAGNAN in that they use a functional programming approach. Our native functions are inspired from Flask's quasi-quoting, and Flask, like D'ARTAGNAN, is a DSL embedded in Haskell. Both Flask and Regiment are different from D'ARTAGNAN and Wavescript in that a macroprogram is written from the perspective of an individual node, rather than the network as a whole. In Wavescript, generated code is the same for all the devices and suitable for homogeneous networks. In contrast, in D'ARTAGNAN, code is generated specifically for the target devices according to behaviour and architecture. This capability is what enables us to generate code for the blockchain and blockchain edge nodes, in addition to the IoT devices.

Macroprogramming has long been proposed as a solution to programming heterogeneous systems. However, as the degree of heterogeneity increases, being restricted to the least common subset of the devices in the programming language can be too limiting. In our extension of D'ARTAGNAN, the adoption of native code allows access to device-specific capabilities.

D'ARTAGNAN is designed to allow programmers to quickly and easily build stream processing applications — applications where information is flowing continuously through the system, as is common in many wireless sensor network use cases. However, since D'ARTAGNAN is *domain specific* to stream processing applications, it may not be suitable to implement other types of applications. In the next chapter we shift focus to blockchain systems — an area where typically applications involve transfer of assets from one participant to another. In macroprogramming such systems, assets may be traded across different blockchain systems. In this domain, D'ARTAGNAN is not suitable, so we propose a new model and framework which has a better fit with smart contracts and the trading of assets across multiple blockchain systems.

8

Porthos

So far we have shown a macroprogramming framework and language for stream-processing applications where code is generated for a heterogeneous network of devices — devices that may be resource-constrained (Chapter 5), and also extending into edge and blockchain systems (Chapter 7). In this chapter we propose a macroprogramming framework and language for a different type of application which spans across multiple systems — commitment-based smart contracts spanning multiple blockchain systems.

8.1 Introduction

Blockchain technology, a type of Distributed Ledger Technology (DLT), has attracted widespread interest in recent years — different blockchain systems will co-exist and will be used for different purposes by individuals, businesses and institutions. A smart contract, which executes on a blockchain system, helps overcome the lack of trust that exists between two or more parties engaged in a trade — it enables business transactions that would otherwise not happen without the assistance of a trusted intermediary. Even though the use of smart contracts is becoming more mainstream, some limitations still remain. Today's smart contracts are intended to execute on a single blockchain system. The need for multi-chain distributed applications (DApps) spanning across multiple blockchain systems is going to in-

crease as blockchain technology continues to gain popularity, and different blockchain systems will co-exist to offer different features, or to provide different benefits.

Interactions between blockchains (interoperability) will be needed to implement new types of applications where assets may be exchanged between participants across different blockchain systems. A single application may handle payments and public interactions on Ethereum [Wood, 2014] and use Hyperledger Fabric [Cachin, 2016] for specific private point-to-point interactions. Implementing such multi-chain DApps is non-trivial. Good knowledge of at least one smart contract language on each of the target underlying blockchain systems is required, together with a good understanding of features and characteristics. Further, blockchain interoperability is not straightforward and the use of relays, notaries or atomic swaps [Buterin, 2016] is required.

What is currently missing and desirable, is the ability to write a single smart contract which spans across multiple blockchain systems. This would replace the need to write several smart contracts (one for each blockchain system) and the handling of the communication between them.

We propose a way of addressing this gap through the use of macroprogramming — a technique often used in the domain of IoT and sensor networks [Gummadi et al., 2005; Kothari et al., 2007; Mainland et al., 2008; Mizzi et al., 2018; Newton et al., 2007b]. With macroprogramming, the level of abstraction is increased and the network is programmed as a whole rather than each component individually. The higher level of abstraction allows the programmer to focus on the logic, rather than the details of communication between components. We propose a domain specific language (DSL) for defining commitment-based smart contracts [Mernik et al., 2005]. DSLs provide a higher level of abstraction than general purpose programming languages and are ideal to make it possible to write secure smart contracts in a quick and efficient way.

Using a technique of embedding a domain specific language, we present

a framework called PORTHOS[1], to define and execute multi-chain smart contracts. In our proof of concept, we show Ethereum and Hyperledger Fabric as two diverse and interacting blockchain systems — a technique that can be extended to other systems. The Turing-incomplete language allows a programmer to describe commitment-based smart contracts that may span multiple blockchain systems. Our work is inspired from financial contracts of Peyton Jones et al. [2000] and the work done with Marlowe [Seijas and Thompson, 2018], but we extend the compilation of contracts to span across multiple chains.

Our contribution is to provide a model in which a commitment-based smart contract can be translated to execute safely on one or more interacting blockchain systems. Our aim is to (i) provide a mechanism to split contract logic on different blockchain systems according to asset location (ii) design a safe and restricted DSL for composing commitment-based smart contracts (iii) define an extensible mechanism to generate code in different target smart contract languages (iv) propose a simple runtime framework to enable chain interoperability.

PORTHOS follows an approach which is similar to D'ARTAGNAN (as described in Chapter 5), with support for heterogeneous networks. The generated code is different for every blockchain system and depends on how assets are used in the application. Unlike D'ARTAGNAN, the communication between nodes (in our case, blockchain systems) requires an off-chain framework to enable a communication medium between nodes — this is due to the nature of existing DLTs which are unable to react to events happening in other systems. Also, in D'ARTAGNAN placement of code depends on the control flow and sensor node capabilities, whereas in PORTHOS placement is asset-based.

A requirement that is becoming increasingly important is the ability to execute DApps across multiple blockchain systems. Different blockchain systems will co-exist to offer a variety of features and to hold different assets. Some blockchain systems can be used for payments, while others as

[1] Available at https://github.com/adrianmizzi/porthos-1

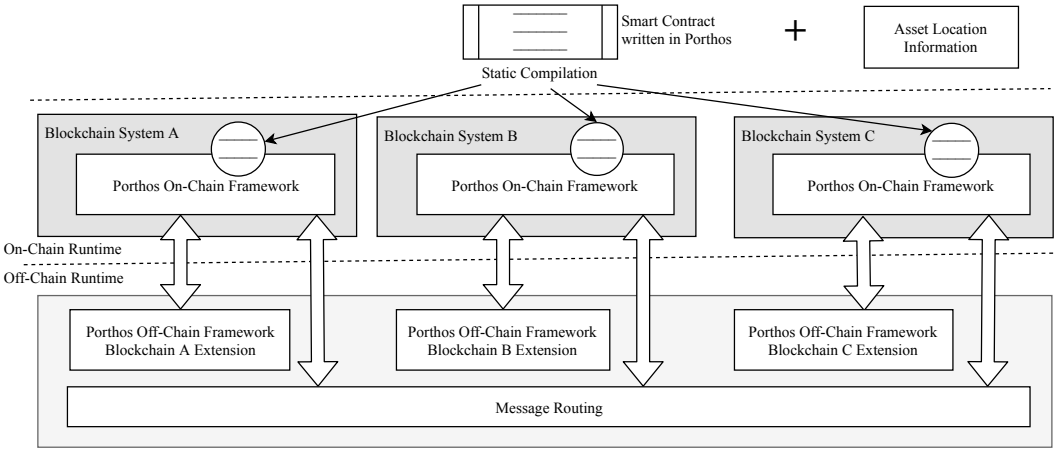

Figure 8.1: Proposed architecture

a register of assets. Vitalik Buterin [Buterin, 2016] identifies three strategies for chain interoperability — atomic swaps using hashed time-locks, relay chains and centralised or multisig notary schemes (refer to Section 6.5).

In PORTHOS, we provide a notary scheme type of interoperability where a trusted group of entities react to events by triggering other smart contracts. In the future, we expect technologies such as Cosmos and Polkadot to replace our off-chain runtime environment, as these become available.

8.2 Porthos Framework

Smart contracts are traditionally written to be executed on a single specific blockchain system. Interactions between smart contracts located on different blockchain systems require complex mechanisms to be implemented — including atomic swaps, notary schemes and relays [Buterin, 2016]. Using a macroprogramming model, we propose to program a network of blockchain systems as a whole, where code (in the form of smart contracts) is automatically generated to be executed on each blockchain system. A higher level of abstraction ensures that the programmer needs only focus on the overall logic of the smart contract using only one programming language.

Similar to D'ARTAGNAN, our aim is to use techniques from the field

of embedded languages to define an embedded domain specific language. Our language is embedded in Haskell — a pure functional language which gives us several useful features, such as polymorphism, higher-order functions and a strong type system. Our model supports both the macroprogramming aspect of writing smart contracts that run across multiple diverse blockchain systems, and also inherently the ability to generate code for different target smart contract languages.

Figure 8.1 illustrates our framework. A macro smart contract is written in our DSEL in a form that can be analysed, translated and deployed to different blockchain systems. The smart contract description first generates an internal representation of the intended contract. With additional information about asset location mapping, the internal representation can be transformed into chunks that need to be placed on the individual blockchain systems according to the assets being used. A first-stage compilation process generates code for each of these chunks into a smart contract language supported by the underlying blockchain system. For example, for Ethereum, the first-stage compilation process generates Solidity and for Hyperledger, Go Chaincode is generated. During a second stage process, the standard compilation and deployment tools for each of the target languages are used to deploy the generated code to the intended blockchain systems. In our example, Solidity is compiled to EVM bytecode and then deployed to an Ethereum blockchain instance, and Go Chaincode is deployed and instantiated on a Hyperledger Fabric instance.

The ultimate goal of PORTHOS is to allow programmers to safely write multi-chain smart contracts that are easy to read and hide away the complexities of blockchain interoperability.

8.2.1 Multi-chain Support

The proposed macroprogramming approach highlights two key challenges. Blockchain systems are heterogeneous — they have different characteristics and not all required functionality may be available on all systems. Secondly,

most of the blockchain systems that we are interested in are passive — they are unable to react to external events. In this section, we describe how these challenges can be addressed.

Blockchain Requirements and Extensions

A blockchain system can be supported in PORTHOS if a minimal set of requirements is satisfied:

- Smart contracts must have an address and must be capable of 'holding' assets transferred to them by users
- Participants must be able to interact with a blockchain system through smart contract functions
- Asset registers must be supported and implementable to track fungible or non-fungible assets

Blockchain systems such as Bitcoin are not supported in PORTHOS. Bitcoin follows the unspent transaction output (UTxO) model and a smart contract capable of holding assets is not supported. Blockchain systems which satisfy the requirements described above can be supported, however, since different systems have different features a solution is needed to harmonise these differences. To address this in PORTHOS, we use blockchain extensions — a solution made up of on-chain and off-chain components which addresses gaps or differences in the required functionality.

The PORTHOS abstraction model requires a callback-on-timeout mechanism to be able to resume execution in case an expected user interaction is not performed in time. This feature is not natively available on Ethereum and other target blockchain systems that we are interested in. For some blockchain systems, third party extensions are readily available to provide this functionality — for example, on the Ethereum blockchain, the Ethereum Alarm Clock [eth, 2019] or Oraclize [ora, 2019] provide this functionality. In our proof-of-concept, we develop our own blockchain-specific extension to complement natively available existing functionality.

Message Routing

Blockchain systems are unable to communicate in the traditional way as normal systems do. Due to the nature of being passive, a common characteristic of current DLTs, the blockchain systems that we are interested in are unable to actively react to events from other systems. The use of an external party is therefore needed to provide a communication layer between blockchain systems. The PORTHOS framework makes use of an external message router to relay messages between one blockchain system and another. The message router is in the spirit of a notary scheme, where events of interest are captured and actioned upon. The communication layer is lightweight in the sense that there is no knowledge of the smart contract logic being executed — the router listens for events generated by the blockchain systems and triggers other contract functions as instructed by these events. The order of the messages is implicit in the contract logic — messages cannot be received out of order as the progression of the contract from one stage to another depends on the delivery of such messages. Duplicate messages are not processed multiple times, as on receipt of the first message, the contract progresses to the next state. This mechanism removes the dependency on the off-chain framework, in that the routing mechanism can be performed by any intermediary or interested party.

In our implementation we trust the intermediaries relaying messages from one blockchain system to another. As long as one honest router exists, then messages cannot be withheld. However, there is a risk that messages can be forged. To overcome this risk, the current implementation can be enhanced such that messages are signed by the originating blockchain system, and validated by the receiving blockchain system before being processed.

8.2.2 Code Cuts

The ultimate aim of the PORTHOS language is to enable writing a single contract to describe DApps which make use of blockchain assets, including both fungible (such as cryptocurrency) and non-fungible assets, residing on

different blockchain systems. Code is sliced into smart contracts, and placed on one of the underlying blockchain systems.

It may be possible to use different strategies to slice code: (i) an execution-cost optimised strategy — executing code on some blockchain systems may be more expensive than others (ii) a location-based strategy — contract logic is placed on the same blockchain system according to where the asset being handled is located (iii) a programmer tag-based strategy, where the programmer instructs which logic should be placed on which blockchain.

In our proposal, we use a location-based placement strategy as this avoids the added burden on the programmer for tagging code. In the future, we envisage enhancing the strategy to consider execution cost too and to possibly allow the user to add compiler hints for optimisation.

8.2.3 Coordination Model

Coordination between interacting smart contracts can be either orchestration or choreography. In an orchestration model, a centralised entity coordinates the execution of the individual parts, whereas in a choreography model, the interactions of each contract are spontaneous on cue.

From a higher level of abstraction, our model follows a choreography model — the message router is enabling a communication medium and makes it possible for blockchain systems to react to events happening on other blockchains. As the execution on one blockchain system is completed, execution starts on cue on another blockchain.

8.3 PORTHOS as a smart contract language

PORTHOS is a domain specific language for composing commitment-based smart contracts [de Kruijff and Weigand, 2017]. In commitment-based smart contracts, a contract is viewed as a business exchange of commitments which are released or cancelled depending on contract criteria. The abstraction model includes the following concepts:

- When a participant *makes a commitment* of an asset towards another participant, the ownership of that asset is transferred to a smart contract and held temporarily.

- A commitment is said to be *released* when the contract transfers the ownership of the held asset to the intended recipient. This is typically done when certain contract conditions are satisfied and the commitment is delivered to the intended recipient.

- A commitment is said to be *cancelled* when the contract returns a committed asset back to the original owner — that is, the participant who made the commitment.

These three concepts are depicted visually in Figure 8.2.

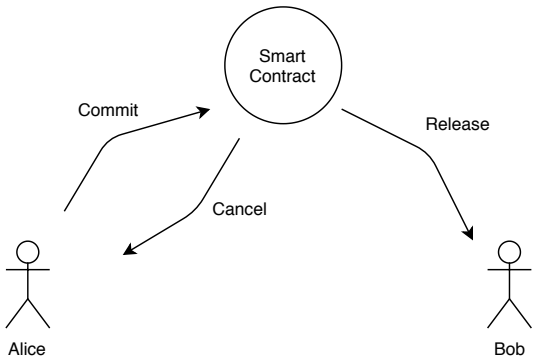

Figure 8.2: Assets are committed by participants, and eventually released or cancelled by the smart contract

PORTHOS is a continuation-based language embedded in Haskell. Basic language constructs are connected together to form a contract. Haskell's strong type system ensures that only valid contracts can be constructed. Contracts are made up of other contracts in a compositional manner. The structure of the language is shown below in pseudo BNF.

```
Contract = onUserCommit <actionName> (<assetType>, FilterExpr)
                Contract (onTimeout <blocktime> Contract) |
           repeatCommit <actionName> (<assetType>, FilterExpr)
                (onTimeout <blocktime> Contract)|
```

```
            releaseAll Contract |
            cancelAll Contract |
            Contract ; Contract |
            if <boolean> then Contract else Contract |
            fireEvent <message> Contract |
            Contract || Contract |
            Contract && Contract |
            sendAssets <participant> Contract |
            end

Commitment = allCommitments |
              whereCommitterIs (<participant>, Commitment) |
              whereReceiverIs (<participant>, Commitment) |
              whereAssetTypeIs (<assetType>, Commitment) |
              orderByParticipant Commitment

FilterExpr = isCommitTo <participant> |
              isCommitBy <participant> |
              isAsset <asset> |
              isAssetType <assetType> |
              FilterExpr & FilterExpr |
              FilterExpr | FilterExpr
```

As a simple example to introduce the PORTHOS language we show how a simple savings-plan contract is implemented by composing constructs together. Such a contract allows an individual to put away assets over a period of time in a time-locked savings account. The committed assets are released after a specific amount of time, thereby helping the individual reach a savings target.

Listing 8.1: A time-locked savings plan

```
savings :: Participant -> Time -> Contract
savings recipient expiryTime =
    repeatCommit "save" (ETH, isCommitTo recipient)
        (onTimeout expiryTime (releaseAll end))
```

The implementation of the time-locked savings plan (see Listing 8.1) is made up by combining three basic constructs: `repeatCommit` followed by

`releaseAll` and finally `end`. Figure 8.3 shows a visual representation of the savings contract. Since PORTHOS is embedded in Haskell, contracts look like Haskell programs. The construct `repeatCommit` causes the progression of the contract to suspend to allow contract participants to make commitments. A filter is used to determine which commitments are accepted by the contract. In this example, valid commitments are in `ETH` (Ether is the native currency on the Ethereum blockchain) and must be in favour of a *recipient* — the identity of the recipient is provided at compilation stage. The same contract can be recompiled with different parameters (i.e. recipient) to generate different contracts. Once the defined timeout period elapses, no more commitments are accepted and the contract execution continues spontaneously — in this example, the contract continues with `releaseAll`, that is, all commitments held in the contract at that point in time are released. The contract then ends.

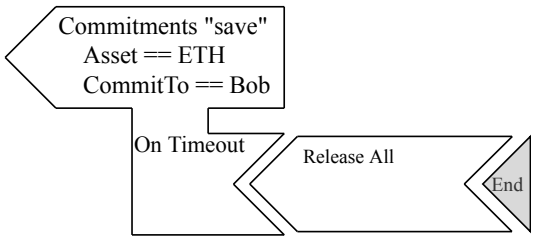

Figure 8.3: A visual representation of a time-locked savings contract

The language provides two distinct basic constructs for accepting commitments. The first, `repeatCommit` described earlier, accepts any number of commitments (zero or more) in a given time-window, and the second expects one-and-only-one commitment (`onUserCommit`). In the latter, when a valid commitment is received, the contract continues execution immediately, or if no commitment is made in time, then the contract resumes with a time-out continuation. An atomic swap contract (see Listing 8.2 and Figure 8.4) allows two participants to swap assets safely — the assets are released to both participants once both commitments have been made. Should any of the participants fail to make a valid commitment in time, then the contract ends and any commitments made are cancelled. Commitments

are accepted by the contract if they match the filter — in the example, the commitment from Participant 1 (`p1`) must be in ether (`ETH`), the intended recipient needs to be Participant 2 (`p2`) and the asset quantity committed must be as specified by `a1`. Since PORTHOS is embedded in Haskell, operators at the contract level must be surrounded by dots (‘.’) as in ‘`.&.`’ to make a distinction from Haskell’s own operators.

Listing 8.2: An asset swap contract

```
swap :: (Participant, Asset Currency) ->
            (Participant, Asset Currency) -> Contract
swap (p1, a1) (p2, a2) =
        onUserCommit "p1Commit"
            (ETH, (isCommitTo p2 .&. isAsset a1))
            doP2Commit
            (onTimeout 10 end)
  where
    doP2Commit =
        onUserCommit "p2Commit"
            (XYZ, (isCommitTo p1 .&. isAsset a2))
            (releaseAll end)
            (onTimeout 20 (cancelAll end))
```

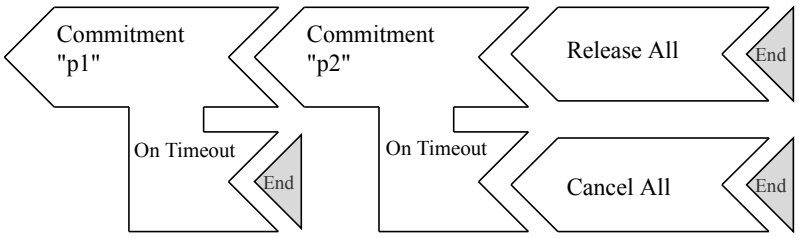

Figure 8.4: A visual representation of an asset swap agreement

Commitments are stored in the smart contract state and can only be accessed via a small SQL-like DSL. Commitments can be filtered, counted and summed to determine whether enough assets have been committed. Specific commitments can be cancelled or released — for example, by specific asset type or for a quantity which is smaller than a specific amount. In a single-asset crowd funding campaign (see Listing 8.3) we sum up all the

assets before deciding whether the campaign has been successful or not. If the campaign is successful, then commitments are released otherwise cancelled and refunded to the crowd funding participants. In a multi-asset crowd funding campaign, where deposits are made with assets located on different blockchain systems (see Listing 8.4), an exchange rate is used to determine if the overall total meets the declared target. Contracts can be composed sequentially — the '.>>>.' (followed-by) operator is equivalent to the ';' (semi-colon) in sequential programming languages. We use this symbol due to the embedding of our language in Haskell.

Listing 8.3: A single-asset crowd funding campaign contract

```
crowdFunding :: Participant -> Asset Currency
                    -> Contract
crowdFunding p target = repeatCommit "fund"
                            (ETH, isCommitTo p)
                            (onTimeout 100 closeC)
  where
    aType = assetType target
    closeC = ifThenElse (sumCommit .>. target)
               (releaseAll (fireEvent "Success" end),
               cancelAll (fireEvent "Fail" end))
    sumCommit = sumC (aType,allCommitments)
```

Listing 8.4: Crowd funding across multiple assets

```
crowdFunding :: Participant -> (Currency, Currency, Float) -> Asset Currency
                    -> Contract
crowdFunding recipient (x, y, f) targetY =
                    both (campaignX, campaignY) .>>>. closeCampaign
  where
    campaignX = repeatCommit "fundX" (x, isCommitTo recipient)
                    (onTimeout 100 end)
    campaignY = repeatCommit "fundY" (y, isCommitTo recipient)
                    (onTimeout 100 end)
    closeCampaign = ifThenElse (totalY .>. targetY)
                      (releaseAll (fireEvent "Campaign Successful" end),
                       cancelAll (fireEvent "Campaign Failed" end))
    sumCommitX = sumC (x, allCommitments)
```

```
    sumCommitY = sumC (y, allCommitments)
    totalY = sumCommitY .+. exchange (x, y, f, sumCommitX)
```

One of the key benefits of embedding the DSL in Haskell, our host language, is that a smart contract can make use of standard Haskell combined with our contract constructs. In a group pay contract (see Listing 8.5), a group of participants agree to transfer an agreed amount to one participant. The funds are released to the recipient only once all commitments have been made. In the example, Haskell's map and lambda expressions are used to concisely build more complex contracts — the use of these techniques is optional, but more experienced programmers will find that the language's expressivity increases even further.

Listing 8.5: Group pay

```
groupPay :: [(Participant, Asset Currency)] -> Participant -> Contract
groupPay yy recipient =
        allOf (userCommits yy) .>>>.
          ifThenElse (countC(allCommitments) .==. liftN (length yy))
            (releaseAll end,
            cancelAll end)
  where
    userCommits = map (\x -> onUserCommit (name (fst x))
                    (ETH, txFilter x) end (onTimeout 100 end))
    txFilter (a, b) = isCommitTo recipient .&. isCommitBy a .&. isAsset b

    allOf :: [Contract] -> Contract
    allOf []     = Null
    allOf [c]    = c
    allOf (c:cc) = both (c, allOf cc)
```

8.3.1 Implementation Details

The implementation of PORTHOS follows an approach which is similar as that of D'ARTAGNAN described in Section 5.4.8 with the main difference being that placement is according to the type of asset. In contrast, with

D'ARTAGNAN, operators are bound to an underlying system depending on the location of the sensor and to minimise radio communication.

PORTHOS is embedded in Haskell as a host language. Haskell's strong type system ensures that only valid contracts can be constructed. Contracts are made up of other contracts in a compositional manner. During the construction of the internal representation, the contract is decomposed of parts to run on different blockchain systems. Communication between blockchain systems is handled through events, where an off-chain router detects events from the blockchain systems. Each event indicates which blockchain system should continue computation, and the off-chain router simply triggers the next contract method with the parameters included in the event.

PORTHOS handles an abstract version of the contracts that need be generated on the target blockchain systems. A single smart contract is generated for every blockchain system in a language supported by that target system. Supporting a new blockchain system involves creating a thin translation layer.

8.4 Use Cases

8.4.1 Property Sale

To illustrate the effectiveness of PORTHOS as a multi-chain smart contract language, we present a property sale agreement where a buyer and a seller make an agreement to transfer property in exchange for payment. As described earlier in Section 8.2.1, in PORTHOS, information about asset location is kept separate from the smart contract. This means that contract logic is clearer to the reader, and simpler to write for the programmer. The high level of abstraction completely omits the details of inter-chain communication that may be necessary to manage the assets.

In our use-case, the property sale is a two-stage process. During the first stage, a buyer and a seller engage in a promise-of-sale agreement —

the buyer confirms interest by committing a deposit amount, and the seller promises to sell the property to the buyer. Within a few weeks or months, the buyer must obtain funds (in the form of a bank-loan or otherwise) to be able to pay the remaining balance. If the buyer is unable to make the payment, the deposit amount is forfeited in favour of the seller and the seller is freed to find a new buyer. However, before the deed is completed, a public notary must also conduct a title search and ensure that all is in order before submitting his approval. Should the notary reject the deed, then the promise-of-sale is cancelled and the buyer receives back his deposit.

Listing 8.6: A property sale agreement

```
propSale :: (Participant, Asset Property) -> Participant -> Asset Currency
            -> Asset Currency -> Participant -> Contract
propSale (seller, property) buyer deposit balance notary =
        onUserCommit "commitProperty"
            (getAssetType property, isCommitTo buyer .&.
                isCommitBy seller .&. isAsset property)
            doBuyerCommit
            (onTimeout 10 end)
  where
    doBuyerCommit   = onUserCommit "payDeposit"
                        (ETH, isCommitBy buyer .&. isCommitTo seller .&.
                            isAsset deposit)
                        doBalanceCommit
                        (onTimeout 20 (cancelAll end))
    doBalanceCommit = onUserCommit "payBalance"
                        (ETH, isCommitTo seller .&. isAsset balance)
                        (oneOf (notaryApproval, notaryRejection))
                        (onTimeout 100 sellerTakesAll)
    sellerTakesAll  = release (whereReceiverIs(seller, allCommitments))
                        (cancel (whereCommitterIs(seller, allCommitments))
                            end)
    notaryApproval  = onUserCommit "approved"
                        (ApprovedByNotary, isCommitBy notary .&.
                            isCommitTo notary)
                        (releaseAll end)
                        (onTimeout 200 (cancelAll end))
```

```
    notaryRejection = onUserCommit "rejected"
                        (RejectedByNotary, isCommitBy notary .&.
                            isCommitTo notary)
                        (cancelAll end)
                        (onTimeout 200 (cancelAll end))
```

We identify three participants: the buyer, the seller and the public notary; and three types of assets: (i) a currency asset used for making payments from the buyer to the seller, (ii) a property asset to represent the asset being transferred from the seller to the buyer, and (iii) the public notary's decision of approval or rejection is also modeled as an asset. A complete implementation of the property-sale smart contract is shown in Listing 8.6. Asset location information is provided during the compilation stage such that the contract logic is placed in the generated code according to which blockchain an asset type is located (see Listing 8.7). Assets may all be located on the same blockchain (in that case, only one smart contract is generated), or alternatively located on different blockchains.

Listing 8.7: Asset location information

```
data Property = Property
instance AssetType Property where
  chainOf _ = "Hyperledger"

data Currency = ETH | XYZ
instance AssetType Currency where
  chainOf ETH = "Ethereum_1"
  chainOf XYZ = "Ethereum_2"
```

The contract is instantiated by providing input values for participants (seller, buyer and notary addresses) and assets traded (property, deposit and balance). Code generated after first-stage compilation is specific to those participants taking part and agreed assets. The generated code, in the form of smart contracts, can then be deployed and instantiated on the respective blockchain systems. If the same contract is to be reused for another property sale between other participants, then the PORTHOS contract is re-compiled from first stage with new input values.

During runtime, the smart contract progresses through different states — the seller commits the property, then the buyer pays the deposit and then the balance, and finally the notary approves or rejects the transfer. The commitments cannot be made out of sequence or at the wrong time, and once a commitment is made, it cannot be cancelled by any participant. Timeout continuations remind the contract programmer to define actions in case a user commitment is not made in time. In some cases, commitments are cancelled but in others (e.g. `sellerTakesAll`) all commitments may be sent to one participant as a result of the inaction of another participant.

8.4.2 Single-shot DAO

The idea behind a Decentralised Autonomous Organisation (DAO) stemmed from the desire to enable stakeholders to make decisions about their organisation directly without the involvement of an agent or manager. In a DAO, a set of rules is encoded in a smart contract to enforce how an organisation is run. This replaces the need for traditional executive-led organisations where decisions are typically made centrally by a management team. Participants can vote on proposals to determine how held assets are invested by the organisation.

Once the rules of the DAO are defined, an initial funding period allows participants to purchase DAO tokens which represent ownership in the organisation. This period is usually called a crowd-sale or Initial Coin Offering (ICO). Once the funding period is over, the owners of the DAO tokens can start making proposals on how funds can be spent. All token holders can vote for a proposal, and if a proposal is accepted, the rules encoded within it (in the form of a smart contract) must be adhered to.

Participation models are determined by the rules of the DAO. For example, token holders may sell some or all of their tokens on an exchange to new persons so they too can get voting powers on the DAO. Alternatively, new voting tokens may be minted and given to persons who are approved by others (through voting) to join the DAO. Such a model is similar

to how companies raise investment through the issue of new shares to new investors.

Since a DAO is encoded as a smart contract, and smart contracts are executed on a single blockchain instance, a DAO is confined to the boundaries of a single blockchain instance — for example, to run on the Ethereum blockchain.

In this use-case, we illustrate how a simplified DAO can be implemented using PORTHOS to be executed in a fully decentralised manner such that voting and assets may be committed by participants on different blockchain systems. Our use-case is restricted to a single-fund-single-vote scenario — which we call a *single-shot DAO*. The phases in a single-shot DAO are shown in Figure 8.5.

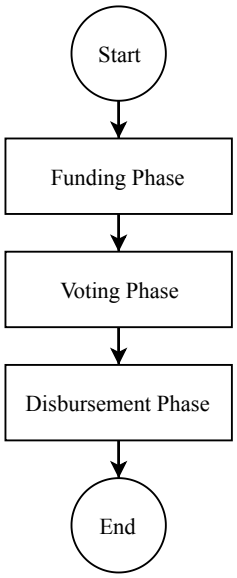

Figure 8.5: Phases of a single-shot DAO

A single-shot DAO has three phases:

1. **Funding**: Participants deposit assets on one of the underlying blockchain instances linked to the DAO
2. **Voting**: Once funding is completed, participants can vote how the deposited assets should be spent

3. **Disbursement**: As voting concludes, votes are counted and the funds or assets are transferred to the most-voted recipient

A complete implementation of a single-shot DAO contract is shown in Listing 8.8.

Listing 8.8: A single-shot DAO

```
andL = foldl1 (.&&.)

ssDAO :: [(Participant, Currency, DaoVote)] -> Contract
ssDAO pp = (andL (map (\(x, y, _) -> doPayment x y) pp)) .>>>.
           (andL (map (\(x, _, z) -> doVote x z) pp)) .>>>.
           releaseAll (sendAssets winner end)
  where
    doPayment x y = onUserCommit ("pay" ++ show x)
                      (y, (isCommitBy x .&. isCommitTo this)) end
                      (onTimeout 20 (cancelAll end))
    doVote x y = onUserCommit ("vote" ++ show x) (y, (isCommitBy x)) end
                   (onTimeout 20 end)
    winner = maxOf (groupByReceiver (whereAssetTypeIs allCommitments (vote)))
```

A single-shot DAO can be instantiated by providing a list of participants and the type of asset that will be used to interact with the system. An example instantiation of the contract is shown here:

```
ssDAO [(alice, Currency ETH1, Vote Ethereum_1),
       (bob, Currency ETH2, Vote Ethereum_2),
       (charlie, Currency ETH1, Vote Ethereum_2)]
```

In the example, three participants (Alice, Bob and Charlie) form a DAO across two blockchain systems (Ethereum_1 and Ethereum_2). Alice and Charlie will commit funds in a fictitious currency ETH1, located on blockchain Ethereum_1, and Bob will deposit funds in ETH2 located on Ethereum_2. The commitments are made in favour of the DAO since the ultimate beneficiary is not yet known. Deposit commitments are accepted in any order, and if any of the deposit commitments is not made within a specified time

period, then the agreement aborts and any commitments already made are cancelled. The agreement progresses to the voting phase only once all three deposits have been received. The three participants vote by making a commitment in favour of their preferred proposal (an address) — to vote, Alice and Bob use the same blockchain they used for deposit (Ethereum_1 and Ethereum_2 respectively), whereas Charlie uses Ethereum_2 to vote rather than Ethereum_1 used for his deposit. Once the voting period elapses, any votes received are counted and the proposal with the highest votes receives the funds (ETH1 and ETH2) stored in the DAO.

The PORTHOS language is expressive enough to define our DAO use-case with a single-funding stage followed by a single-voting stage. It can also be used with additional stages (more funding stages and/or more voting stages), however the number of stages must be known in advance. Since PORTHOS does not support loops (intentionally by design), the applications which can be expressed must have a predefined number of stages. To be able to implement a more enhanced DAO, new concepts would need to be introduced — the ability to re-initialise a contract from within a contract, and the ability to create (mint) new tokens. Figure 8.6 shows how an enhanced DAO would be built once such new concepts are made available.

This use-case has also shown how participants can commit assets (cryptocurrency in this example) on different blockchain systems. Since assets are not traded from one blockchain system to another (for example, 1 ETH1 converted to 3 ETH2), recipients, or winning proposals, must have an address on each blockchain system to be able to receive funds committed on that blockchain system. The introduction of an exchange service to the runtime framework would allow certain assets to be traded, such that recipients can receive funds in their preferred currency.

8.5 Evaluation

We evaluate PORTHOS in three parts. We first evaluate the programming abstraction by showing that it is expressive enough to implement a vari-

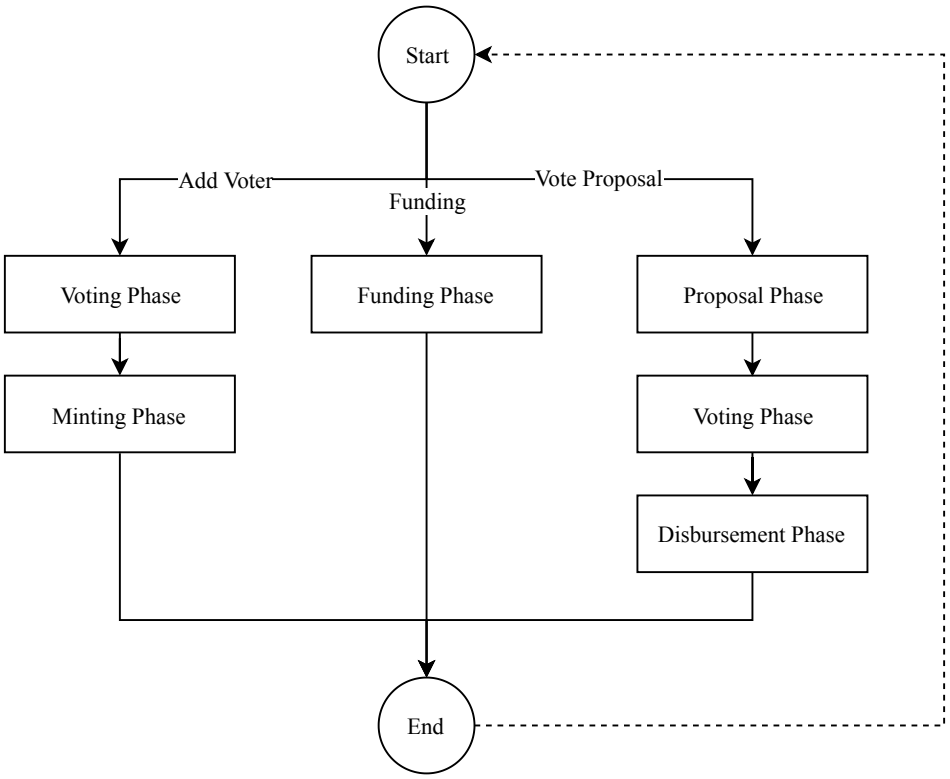

Figure 8.6: Enhanced DAO

ety of applications. Then we evaluate the security aspects of the proposed model and finally, we evaluate the framework's extensibility.

8.5.1 Expressiveness of Abstraction

We evaluate the expressiveness of the abstraction model by showing that the language can be used to implement a variety of commitment-based smart contracts.

In this chapter, we have shown two detailed use-cases and a number of smaller examples applications: a property sale, a single-shot DAO, group payments, asset swapping and crowd-funding. The model is suitable for process-oriented applications with a finite number of steps. Applications which include a voting element, such as elections, can be implemented by treating votes in the same way as assets — a participant votes by committing

a token in favour of a candidate.

In PORTHOS, user interactions with smart contracts are limited to commitments. Other interactions, such as cancelling or redeeming a commitment, are not currently possible. Other languages (such as Marlowe) support the notion of soft commits — commitments can be cancelled by the participant directly. We believe our limitation on user interactions is not too restrictive on the variety of smart contracts that can be described. The language is intentionally Turing incomplete, so applications involving loops are not describable. For example, it is not possible to write an application that accepts commitments until a specific condition is met, and thereby potentially creating a non-terminating contract. We do not consider this to be a shortcoming of the model, but rather an intentional design decision to ensure contracts terminate.

8.5.2 Security Analysis

Blockchain systems have a high level of security due to their decentralised and immutable characteristics — the consensus algorithm ensures that any data added to the blockchain is verified by multiple parties, and immutability ensures that data is not changed. However, blockchain systems are still not completely immune to attacks or weaknesses. For example, in an attack where malicious miners obtain 51%+ of the hashing power of the network may create the possibility that stored data is altered and leading to situations such as double spending of funds. It is also possible for smart contracts to behave incorrectly where interaction with the external world is compromised, in the form of oracles providing false information.

Applications that span multiple blockchain systems may introduce new attack vectors and security weaknesses — these weaknesses are applicable to any multi-chain application, however, we felt that this evaluation would not be complete without mentioning these.

Single Point of Failure — One of the strengths of blockchain systems is that due to the decentralised nature, no single point of failure (SPOF)

exists. However, when working with multiple blockchain systems and relying on an intermediary to route and relay messages between one blockchain system and another, a SPOF can be introduced. To mitigate this, the communication layer itself must be decentralised with multiple copies running concurrently. As this may cause the same smart contract function to be triggered multiple times by different intermediaries, a mechanism for filtering duplicate messages (e.g. nonce) must be used.

Third-Party Interference — Intermediaries route messages between one blockchain system and another. It may be possible for intermediaries to withhold, modify or forge messages between blockchain systems. As long as one honest intermediary is available to relay messages, then messages cannot be withheld. To mitigate the risk of modifying or forging messages, all communication is signed by the originating system. During initiation stage, cryptographic keys are exchanged such that messages can be verified.

Distributed Transactions — Transactions spanning multiple systems are susceptible to partial failure, causing other parts of the same transaction to be reversed on other systems. If the transaction is not reversed correctly, the systems may end up with inconsistent state. This scenario also applies for multi-chain apps. Our framework does not currently handle such errors in a graceful manner, and this is an area to be developed further in future enhancements.

8.5.3 Extensibility

The PORTHOS framework is extensible in different directions. Different asset types are supported as long as a contract interface is implemented. The model can also be extended to support new systems by adding code generation into the target language, and extending the on-chain/off-chain framework. The framework currently generates Solidity code to be executed on

multiple Ethereum instances, as well as Go Chaincode for Hyperledger Fabric. Blockchain systems can be added to PORTHOS as long as the requirements described in Section 8.2.1 are satisfied.

8.6 Conclusions

In this chapter we presented PORTHOS, an embedded DSL framework for describing commitment-based smart contracts spanning multiple blockchain systems. This is to our knowledge the first attempt at providing a macro-level approach for specifying multi-chain smart contracts in a single specification. The closest work to that being presented in this chapter include: our other work with D'ARTAGNAN for programming IoT devices at a network level (see Chapter 5) and for writing a single macroprogram for blockchain connected IoT devices (see Chapter 7); Marlowe for specifying financial contracts on Cardano [Seijas and Thompson, 2018].

We have shown through examples that the PORTHOS language is expressive enough to be used to implement a variety of commitment-based smart contracts across heterogeneous blockchain systems such as Ethereum and Hyperledger Fabric. The language is designed with safety in mind such that the smart contract programmer is aware of timeout scenarios and must define what happens in these situations. Although the language cannot avoid all types of 'bugs', it does help the programmer to significantly reduce easy-to-forget cases.

PORTHOS has shown that by raising the abstraction level, it is possible to separate the complexities of placement and communication from the contract logic such that the programmer needs only to focus on the contract. Through the use of composition there is much to be gained as complex contracts can be made up of simpler contracts.

Part IV

Conclusions

9

Conclusions and Future Work

The main goal of this thesis was to study how the combined techniques of macroprogramming, multi-target compilation and embedded DSLs, can be used in heterogeneous networks for domain-specific applications. Our approach involved the use of an embedded domain specific language technique to design and implement programming languages with their respective frameworks, for different application domains. The use of a deep embedding approach allows for the analysis, transformation and multiple interpretations of an internal representation of the application description.

We designed a language and framework (D'ARTAGNAN in chapter 5) for stream-processing applications in heterogeneous wireless sensor networks. A stream processor description written in our language generates code to run on multiple, potentially different, devices. The framework automatically splits application logic to run on the nodes making up the network, although the programmer may optionally provide *hints* to influence the code generation. We learnt that code generated by the framework is significantly less efficient than the hand-written equivalent — however, existing compilers practically eliminate the inefficiencies through optimisation to produce a version which is at par with the hand-written version on both size and speed aspects.

We then extended D'ARTAGNAN for applications that extend beyond wireless sensor networks, into edge systems and blockchain platforms (chap-

ter 7). We found that the framework and techniques can be easily extended beyond the domain of wireless sensor networks for a truly heterogeneous network. As blockchain and other concepts were introduced to the framework, the language constructs were easily adapted and enhanced to include the new functionality.

To further illustrate the flexibility of the proposed approach, we also targeted a radically different application domain — smart contracts that span multiple blockchain systems. In chapter 8, we proposed a language and framework (PORTHOS) for a different paradigm, commitment-based smart contracts — smart contracts between participants were the interactions are limited to commitments of assets. Although the core concepts around the framework are similar to D'ARTAGNAN a number of interesting challenges were encountered. Blockchain interoperability (or the lack of) creates challenges as the passive nature of popular blockchain systems makes communication harder to achieve. Also, since existing smart contract compilers are relatively new and less mature than other domains, generated code is not optimised to the same levels experienced with D'ARTAGNAN. With PORTHOS we learnt that by restricting the application domain to commitment-based smart contracts, the language is still expressive enough to write and implement a wide variety of examples and applications. For safety reasons, PORTHOS was intentionally designed as non-Turing complete making it ideal to describe applications with a predefined and finite number of steps.

9.1 Future Work

The work done in this thesis can be further developed in different directions. The implementation of additional use-cases for each framework can help identify gaps and features that can be used to extend and strengthen the languages. In addition, in this section we identify other areas that from our experience can be researched further.

Language evaluation framework — One of the difficulties that we faced during the evaluation of our work is the lack of a standard approach for evaluating languages. Different approaches have been undertaken in the language domain ranging from a questionnaire approach as user-acceptance with a group of programmers (usually a class of students) who assess and compare the language to other languages in a guided session, to arbitrary quantitative measurements around arising inefficiencies of the resulting code. A standard framework for evaluating languages would provide a benchmark and ranking mechanism of a language against other languages on different aspects such as expressivity, performance and safety. It may be possible to define use-case categories that can be implemented to illustrate the expressivity of a language, as well as guidelines for benchmarking the performance on a set of predefined applications.

Interoperability — As soon as we ventured outside the boundaries of a wireless sensor network we realised that the communication between different systems and devices was one of the most significant challenges to overcome. To communicate between wireless sensor nodes and a blockchain system, we needed to communicate over three media: the radio network for wireless sensor nodes; a serial port on a server where one wireless sensor node acted as a gateway to the wireless sensor network; and the blockchain node for communication with the rest of the blockchain network. When we shifted our attention to blockchain systems, we found no suitable ready-to-use solution so we developed our own *minimum viable solution* to be able to illustrate our framework and language. There are several gaps in interoperability, in particular a lack of a standard communication model between disparate systems.

Generalisation of macroprogramming approach — In our work we have shown how our model can be used effectively in domain specific applications through the use of a restricted Turing-incomplete language.

We question whether the model can be used in a more generic setup by applying it to a general-purpose programming language and to see how effective that would be. How far can the idea be taken with an unrestricted language and what difficulties exist to adopting such a macroprogramming approach?

9.2 Concluding Thoughts

The main aim of this thesis was to study the use of the macroprogramming technique on heterogeneous networks using an embedded DSL approach. Through this work, it has been shown that the proposed model can be successfully applied to diverse application domains and paradigms, and much is to be gained by raising the level of abstraction for multi-system applications. We have shown how the same model can be applied using different languages to be better suited for different applications.

This work has also shown that when applying the technique to domains where existing compilers are well matured, the inefficiencies created in generated code are practically eliminated thereby making the model a serious challenger to traditional programming methods. We hope that in the future these results lead to increased adoption of macroprogramming techniques for the development of multi-system applications.

A

Publications

The work presented in this thesis includes the following published papers.

Published Papers

1. Doctoral Symposium: An Embedded DSL Framework for Distributed Embedded Systems. Adrian Mizzi, Joshua Ellul and Gordon Pace, in Proceedings of the 11th ACM International Conference on Distributed and Event-based Systems, DEBS '17, pages 374-377. ACM, 2017.

2. D'Artagnan: An Embedded DSL Framework for Distributed Embedded Systems. Adrian Mizzi, Joshua Ellul and Gordon Pace, in Proceedings of the Real World Domain Specific Languages Workshop 2018, page 2. ACM, 2018b.

3. Macroprogramming the Blockchain of Things. Adrian Mizzi, Joshua Ellul and Gordon Pace, in Proceedings of The 1st International Workshop on Blockchain for the Internet of Things, pages 1673–1678.

4. Porthos: Macroprogramming Blockchain Systems. Adrian Mizzi, Joshua Ellul and Gordon Pace, in Proceedings of the 10th IFIP International Conference on New Technologies, Mobility & Security.

References

Liquidity language. URL `http://www.liquidity-lang.org/doc/`. [Online; accessed March-2019].

Ivy language. `https://blog.chain.com/announcing-ivy-playground-395364675d0a`, 2017. [Online; accessed February-2019].

Cardano. `https://www.cardano.org/en/home/`, 2018. [Online; accessed October-2018].

Ethereum alarm clock. `https://www.ethereum-alarm-clock.com/`, 2019. [Online; accessed January-2019].

Oraclize. `https://docs.oraclize.it/`, 2019. [Online; accessed January-2019].

Solidity. `https://solidity.readthedocs.io/en/latest/`, 2019. [Online; accessed March-2019].

Fthi Arefayne Abadi, Joshua Ellul, and George Azzopardi. The blockchain of things, beyond bitcoin: A systematic review. In *2018 IEEE International Conference on Internet of Things (iThings) and IEEE Green Computing and Communications (GreenCom) and IEEE Cyber, Physical and Social Computing (CPSCom) and IEEE Smart Data (SmartData)*, pages 1666–1672. IEEE, 2018.

Ian F Akyildiz, Weilian Su, Yogesh Sankarasubramaniam, and Erdal Cayirci. Wireless sensor networks: a survey. *Computer networks*, 38(4):393–422, 2002.

Giuseppe Anastasi, Marco Conti, Mario Di Francesco, and Andrea Passarella. Energy conservation in wireless sensor networks: A survey. *Ad hoc networks*, 7(3):537–568, 2009.

Markus Aronsson, Emil Axelsson, and Mary Sheeran. Stream processing for embedded domain specific languages. In *Proceedings of the 26nd 2014 International Symposium on Implementation and Application of Functional Languages*, page 8. ACM, 2014.

Faisal Aslam, Luminous Fennell, Christian Schindelhauer, Peter Thiemann, Gidon Ernst, Elmar Haussmann, Stefan Rührup, and Zastash A Uzmi. Optimized java binary and virtual machine for tiny motes. In *International Conference on Distributed Computing in Sensor Systems*, pages 15–30. Springer, 2010.

Nicola Atzei, Massimo Bartoletti, and Tiziana Cimoli. A survey of attacks on ethereum smart contracts (sok). In *POST*, volume 10204 of *Lecture Notes in Computer Science*, pages 164–186. Springer, 2017.

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

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 2010a. doi: 10.1109/MEMCOD.2010.5558637.

Emil Axelsson, Koen Claessen, Mary Sheeran, Josef Svenningsson, David Engdal, and Anders Persson. The design and implementation of feldspar. In *Symposium on Implementation and Application of Functional Languages*, pages 121–136. Springer, 2010b.

Alex Biryukov, Dmitry Khovratovich, and Sergei Tikhomirov. Findel: secure derivative contracts for ethereum. In *International Conference on Financial Cryptography and Data Security*, pages 453–467. Springer, 2017.

Per Bjesse, Koen Claessen, Mary Sheeran, and Satnam 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. ISBN 1-58113-024-4. doi: 10.1145/289423.289440. URL `http://doi.acm.org/10.1145/289423.289440`.

Tobias Blickle, Jürgen Teich, and Lothar Thiele. System-level synthesis using evolutionary algorithms. *Design Automation for Embedded Systems*, 3(1):23–58, 1998.

Aymen Boudguiga, Nabil Bouzerna, Louis Granboulan, Alexis Olivereau, Flavien Quesnel, Anthony Roger, and Renaud Sirdey. Towards better availability and accountability for

iot updates by means of a blockchain. In *IEEE Security & Privacy on the Blockchain (IEEE S&B 2017)*, 2017.

Niels Brouwers, Koen Langendoen, and Peter Corke. Darjeeling, a feature-rich vm for the resource poor. In *Proceedings of the 7th ACM Conference on Embedded Networked Sensor Systems*, pages 169–182. ACM, 2009.

Vitalik Buterin. Chain interoperability. *R3 Research Paper*, 2016.

Vitalik Buterin. Vyper, 2018. URL `https://vyper.readthedocs.io/en/latest/`.

Christian Cachin. Architecture of the hyperledger blockchain fabric. In *Workshop on Distributed Cryptocurrencies and Consensus Ledgers*, volume 310, 2016.

Alexandru Caracas, Thorsten Kramp, Michael Baentsch, Marcus Oestreicher, Thomas Eirich, and Ivan Romanov. Mote runner: A multi-language virtual machine for small embedded devices. In *2009 Third International Conference on Sensor Technologies and Applications*, pages 117–125. IEEE, 2009.

Gaetano Caruana and Gordon J Pace. Embedded languages for origami-based geometry. *Proceedings of CSAW'07*, page 99, 2007.

Han Chen, Paul Chou, Sastry Duri, Hui Lei, and Johnathan 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.

Thang Vu Chien, Hung Nguyen Chan, and Thanh Nguyen Huu. A comparative study on operating system for wireless sensor networks. In *Advanced Computer Science and Information System (ICACSIS), 2011 International Conference on*, pages 73–78. IEEE, 2011.

Koen Claessen and Gordon J. Pace. An embedded language framework for hardware compilation. In *Designing Correct Circuits '02, Grenoble, France*, April 2002.

Koen Claessen and David Sands. *Observable Sharing for Functional Circuit Description*, pages 62–73. Springer Berlin Heidelberg, Berlin, Heidelberg, 1999. ISBN 978-3-540-46674-1. doi: 10.1007/3-540-46674-6_7. URL `http://dx.doi.org/10.1007/3-540-46674-6_7`.

Michael Coblenz. Obsidian: a safer blockchain programming language. In *Proceedings of the 39th International Conference on Software Engineering Companion*, pages 97–99. IEEE Press, 2017.

Joseph Cordina and Gordon J Pace. Functional hdls: A historical overview. In *Proceedings of Computer Science Annual Workshop*, 2006.

Joost de Kruijff and Hans Weigand. Ontologies for commitment-based smart contracts. In *OTM Confederated International Conferences" On the Move to Meaningful Internet Systems"*, pages 383–398. Springer, 2017.

G De Michell and Rajesh K Gupta. Hardware/software co-design. *Proceedings of the IEEE*, 85(3):349–365, 1997.

Gergely Dévai, Máté Tejfel, Zoltán Gera, Gábor Páli, Gyula Nagy, Zoltán Horváth, Emil Axelsson, Mary Sheeran, András Vajda, Bo Lyckegård, et al. Efficient code generation from the high-level domain-specific language feldspar for dsps. In *Proc. ODES-8: 8th Workshop on Optimizations for DSP and Embedded Systems, workshop associated with IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, 2010.

Rainer Dömer, Andreas Gerstlauer, Junyu Peng, Dongwan Shin, Lukai Cai, Haobo Yu, Samar Abdi, and Daniel D Gajski. System-on-chip environment: A specc-based framework for heterogeneous mpsoc design. *EURASIP Journal on Embedded Systems*, 2008:5, 2008.

Adam Dunkels, Bjorn Gronvall, and Thiemo 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.

Conal Elliott and Paul Hudak. Functional reactive animation. In *ACM SIGPLAN Notices*, volume 32, pages 263–273. ACM, 1997.

Conal Elliott, Sigbjørn Finne, and Oege De Moor. Compiling embedded languages. *Journal of functional programming*, 13(03):455–481, 2003.

Joshua Ellul. *Run-time compilation techniques for wireless sensor networks*. PhD thesis, University of Southampton, 2012.

Joshua Ellul and Kirk Martinez. Run-time compilation of bytecode in sensor networks. In *2010 Fourth International Conference on Sensor Technologies and Applications*, pages 133–138. IEEE, 2010.

Joshua Ellul and Gordon J Pace. Alkylvm: A virtual machine for smart contract blockchain connected internet of things. In *New Technologies, Mobility and Security (NTMS), 2018 9th IFIP International Conference on*, pages 1–4. IEEE, 2018.

Claudio M De Farias, Wei Li, Flávia C Delicato, Luci Pirmez, Albert Y Zomaya, Paulo F Pires, and José N De Souza. A systematic review of shared sensor networks. *ACM Computing Surveys (CSUR)*, 48(4):51, 2016.

Christopher K Frantz and Mariusz Nowostawski. From institutions to code: Towards automated generation of smart contracts. In *Foundations and Applications of Self* Systems, IEEE International Workshops on*, pages 210–215. IEEE, 2016.

Thierry Gautier, Paul Le Guernic, and Loic Besnard. Signal: A declarative language for synchronous programming of real-time systems. In *Conference on Functional Programming Languages and Computer Architecture*, pages 257–277. Springer, 1987.

Jeremy Gibbons. Functional programming for domain-specific languages. In *Central European Functional Programming School*, pages 1–28. Springer, 2015.

Andy Gill. Domain-specific languages and code synthesis using haskell. *Queue*, 12(4):30, 2014.

Thorsten Grötker, Stan Liao, Grant Martin, and Stuart Swan. *System Design with SystemCTM*. Springer Science & Business Media, 2007.

Ramakrishna Gummadi, Omprakash Gnawali, and Ramesh Govindan. Macroprogramming wireless sensor networks using kairos. In *International Conference on Distributed Computing in Sensor Systems*, pages 126–140. Springer, 2005.

Nicholas Halbwachs, Paul Caspi, Pascal Raymond, and Daniel Pilaud. The synchronous data flow programming language lustre. *Proceedings of the IEEE*, 79(9):1305–1320, 1991.

Jörg Henkel and Rolf Ernst. A hardware/software partitioner using a dynamically determined granularity. In *Proceedings of the 34th annual Design Automation Conference*, pages 691–696. ACM, 1997.

Patrick C Hickey, Lee Pike, Trevor Elliott, James Bielman, and John Launchbury. Building embedded systems with embedded dsls. In *ACM SIGPLAN Notices*, volume 49, pages 3–9. ACM, 2014.

Ralf Hinze et al. Fun with phantom types. *The fun of programming*, pages 245–262, 2003.

Yoichi Hirai. Bamboo, Nov 2018. URL `https://github.com/pirapira/bamboo`.

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

Paul Hudak. Building domain-specific embedded languages. *ACM Computing Surveys (CSUR)*, 28(4es):196, 1996.

S. Huh, S. Cho, and S. Kim. Managing iot devices using blockchain platform. In *2017 19th International Conference on Advanced Communication Technology (ICACT)*, pages 464–467, Feb 2017. doi: 10.23919/ICACT.2017.7890132.

Marc Jansen, Farouk Hdhili, Ramy Gouiaa, and Ziyaad Qasem. Do smart contract languages need to be turing complete? 03 2019.

Samuel N Kamin. Research on domain-specific embedded languages and program generators. *Electronic Notes in Theoretical Computer Science*, 14:149–168, 1998.

Imran Khan, Fatna Belqasmi, Roch Glitho, Noel Crespi, Monique Morrow, and Paul Polakos. Wireless sensor network virtualization: A survey. *IEEE Communications Surveys & Tutorials*, 18(1):553–576, 2016.

JeongGil Ko, Kevin Klues, Christian Richter, Wanja Hofer, Branislav Kusy, Michael Bruenig, Thomas Schmid, Qiang Wang, Prabal Dutta, and Andreas Terzis. *Low Power or High Performance? A Tradeoff Whose Time Has Come (and Nearly Gone)*, pages 98–114. Springer Berlin Heidelberg, Berlin, Heidelberg, 2012. ISBN 978-3-642-28169-3. doi: 10.1007/978-3-642-28169-3_7. URL `http://dx.doi.org/10.1007/978-3-642-28169-3_7`.

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

Jae Kwon and Ethan Buchman. Cosmos White Paper. `https://cosmos.network/resources/whitepaper`, 2018. [Online; accessed 30-January-2019].

Daan Leijen and Erik Meijer. Domain specific embedded compilers. In *ACM Sigplan Notices*, volume 35, pages 109–122. ACM, 1999.

Ilias Leontiadis, Christos Efstratiou, Cecilia Mascolo, and Jon Crowcroft. Senshare: transforming sensor networks into multi-application sensing infrastructures. In *European Conference on Wireless Sensor Networks*, pages 65–81. Springer, 2012.

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

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

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

Loi Luu, Duc-Hiep Chu, Hrishi Olickel, Prateek Saxena, and Aquinas Hobor. Making smart contracts smarter. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, pages 254–269. ACM, 2016.

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

Geoffrey 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.

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

Alan Mainwaring, David Culler, Joseph Polastre, Robert Szewczyk, and John 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.

Lucius Gregory Meredith, Gary Pettersson, Jack Stephenson, Michael Stay, Kent Shikama, and Joseph Denman. Contracts, composition, and scaling: The rholang specification 0.2., 2018.

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

Adrian Mizzi, Joshua Ellul, and Gordon Pace. D'artagnan: An embedded dsl framework for distributed embedded systems. In *Proceedings of the Real World Domain Specific Languages Workshop 2018*, page 2. ACM, 2018.

Charles H Moore and Geoffrey C Leach. Forth–a language for interactive computing. *Amsterdam: Mohasco Industries Inc*, 1970.

Luca Mottola and Gian Pietro Picco. Programming wireless sensor networks: Fundamental concepts and state of the art. *ACM Comput. Surv.*, 43(3):19:1–19:51, April 2011. ISSN 0360-0300. doi: 10.1145/1922649.1922656.

Satoshi Nakamoto. Bitcoin: A peer-to-peer electronic cash system, 2008.

R. Newton, G. Morrisett, and M. Welsh. The regiment macroprogramming system. In *2007 6th International Symposium on Information Processing in Sensor Networks*, pages 489–498, April 2007a. doi: 10.1109/IPSN.2007.4379709.

Ryan Newton and Matt Welsh. Region streams: Functional macroprogramming for sensor networks. In *Proceeedings of the 1st international workshop on Data management for sensor networks: in conjunction with VLDB 2004*, pages 78–87. ACM, 2004.

Ryan Newton, Greg Morrisett, and Matt 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, 2007b. ACM. ISBN 978-1-59593-638-7. doi: 10.1145/1236360.1236422. URL `http://doi.acm.org/10.1145/1236360.1236422`.

Ryan R. Newton, Lewis D. Girod, Michael B. Craig, Samuel R. Madden, and John Gregory Morrisett. Design and evaluation of a compiler for embedded stream programs. *SIGPLAN Not.*, 43(7):131–140, June 2008. ISSN 0362-1340. doi: 10.1145/1379023.1375675.

Russell O'Connor. Simplicity: A new language for blockchains, 2017. URL `https://blockstream.com/simplicity.pdf`.

John O'Donnell. Overview of hydra: A concurrent language for synchronous digital circuit design. In *ipdps*, volume 2, page 240, 2002.

Gordon J Pace. Hedla: A strongly typed, component-based embedded hardware description language. *Proceedings of CSAW'07*, page 192, 2007.

Ian Page. Constructing hardware-software systems from a single description. *Journal of VLSI signal processing systems for signal, image and video technology*, 12(1):87–107, 1996.

Jack Pettersson and Robert Edström. *Safer smart contracts through type-driven development*. PhD thesis, Master's thesis, Chalmers University, Department of Computer Science and Engineering, Sweden, 2016.

Simon Peyton Jones, Jean-Marc Eber, and Julian Seward. Composing contracts: an adventure in financial engineering (functional pearl). In *ACM SIGPLAN Notices*, volume 35, pages 280–292. ACM, 2000.

Lee Pike, Alwyn Goodloe, Robin Morisset, and Sebastian Niller. Copilot: a hard real-time runtime monitor. In *International Conference on Runtime Verification*, pages 345–359. Springer, 2010.

S Popejoy. The pact smart contract language (2016), 2016. URL `http://kadena.io/try-pact/`.

Gregory J Pottie and William J Kaiser. Wireless integrated network sensors. *Communications of the ACM*, 43(5):51–58, 2000.

Cauligi S Raghavendra, Krishna M Sivalingam, and Taieb Znati. *Wireless sensor networks*. Springer, 2006.

Vijay Raghunathan, Curt Schurgers, Sung Park, and Mani B Srivastava. Energy-aware wireless microsensor networks. *IEEE Signal processing magazine*, 19(2):40–50, 2002.

Niels Reijers and Chi-Sheng Shih. Ahead-of-time compilation of stack-based jvm bytecode on resource-constrained devices. In *Proceedings of the 2017 International Conference on Embedded Wireless Systems and Networks*, pages 84–95. Junction Publishing, 2017.

Niels Reijers, Joshua Ellul, and Chi-Sheng Shih. Making sensor node virtual machines work for real-world applications. *IEEE Embedded Systems Letters*, 11(1):13–16, 2018.

M. Samaniego and R. Deters. Hosting virtual iot resources on edge-hosts with blockchain. In *2016 IEEE International Conference on Computer and Information Technology (CIT)*, pages 116–119, Dec 2016. doi: 10.1109/CIT.2016.71.

Pablo Lamela Seijas and Simon Thompson. Marlowe: Financial contracts on blockchain. In *International Symposium on Leveraging Applications of Formal Methods*, pages 356–375. Springer, 2018.

Ilya Sergey, Amrit Kumar, and Aquinas Hobor. Scilla: a smart contract intermediate-level language. *arXiv preprint arXiv:1801.00687*, 2018.

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

Ross Kitsis Shidokht Hejazi-Sepehr and Ali Sharif. Transwarp-conduit: Interoperable blockchain application framework. `https://aion.network/media/TWC_Paper_Final.pdf`, 2019. [Online; accessed 5-May-2019].

A. Stanciu. Blockchain based distributed control system for edge computing. In *2017 21st International Conference on Control Systems and Computer Science (CSCS)*, pages 667–671, May 2017. doi: 10.1109/CSCS.2017.102.

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

Bharath Sundararaman, Ugo Buy, and Ajay D Kshemkalyani. Clock synchronization for wireless sensor networks: a survey. *Ad hoc networks*, 3(3):281–323, 2005.

Josef Svenningsson and Emil Axelsson. Combining deep and shallow embedding for edsl. In *International Symposium on Trends in Functional Programming*, pages 21–36. Springer, 2012.

Stefan Thomas and Evan Schwartz. A protocol for interledger payments. *URL https://interledger. org/interledger. pdf*, 2015.

William W Wadge and Edward A Ashcroft. *LUCID, the dataflow programming language*, volume 198. Academic Press London, 1985.

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. doi: 10.1109/EWSN.2005.1462003.

Gavin Wood. Ethereum: A secure decentralised generalised transaction ledger. *Ethereum project yellow paper*, 151:1–32, 2014.

Gavin Wood. Polkadot: Vision for a heterogeneous multi-chain framework. `https://polkadot.network/PolkaDotPaper.pdf`, 2018. [Online; accessed 30-January-2019].

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

Ryan Yates and Brent A Yorgey. Diagrams: a functional edsl for vector graphics. In *Proceedings of the 3rd ACM SIGPLAN International Workshop on Functional Art, Music, Modelling and Design*, pages 4–5. ACM, 2015.