Macroprogramming the Blockchain of Things

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

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

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

***Abstract*—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 paper, we propose a macroprogramming approach 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.

***Index Terms*—macroprogramming, blockchain, edge, embedded systems, IoT**

I. INTRODUCTION

The rise of smart contracts on blockchain or other distributed ledger technologies have brought the possibility of regulated interaction and resource exchange between parties without the need of a trusted entity. With smart contract technologies such as Ethereum [1], 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. We have previously looked at providing means to explicitly switch word-size in order to have virtual machine-level code executable across blockchain and IoT devices transparently [2].

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 recent cases in which the equivalent of millions 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 [4] — 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 paper, 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.

D'ARTAGNAN [5] is a macroprogramming language we have developed 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 paper, we discuss how D'ARTAGNAN can be extended to push the limits of heterogeneity to edge devices and even onto smart contracts on the 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

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"*.

[1]See [3] for a list of cases, although since its publication many other high profile, high loss cases have happened, including a recent Parity Multi-Sig Wallet bug which resulted in a loss of over 300 million USD.

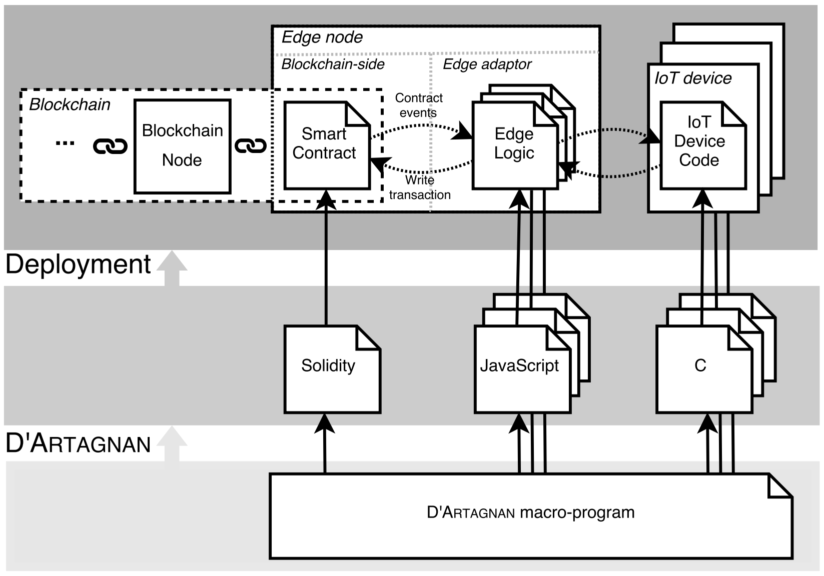

Fig. 1. Proposed Macroprogramming Blockchain of Things Architecture

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 the paper is organised as follows. In Section II we describe the architecture and workflow of our proposed solution. Then, in Section III 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 IV, an evaluation in Section V and conclude in Section VI.

II. 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 itself 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 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 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.

III. D'ARTAGNAN: A MACROPROGRAMMING FRAMEWORK

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.

Designing and building such a domain-specific language to enable programming the stream processor from a global perspective, and developing a compiler which automatically deploys the different parts onto different devices possibly using different technologies would thus enable high level programming of such systems. However, the need for constant updates to the language to cater for new devices, technologies and native capabilities of these devices can be a major challenge. One approach which has been proposed in the literature has been that of *embedded languages* [6], in which a domain-specific language is integrated into an existing general-purpose language as a library whose use appears like domain-specific subprograms in the host language. Such an approach has been advocated due to the ease of development (since the infrastructure and tools of the host language can be directly adopted) and the fact that the host language acts as a meta-language of the domain-specific one, and can thus be generated, manipulated and transformed within the same programming environment. For these reasons, embedded domain-specific languages have been argued to be ideal for prototyping languages and for languages which may undergo frequent language-design changes.

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 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 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 importantly, 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.

A. 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).

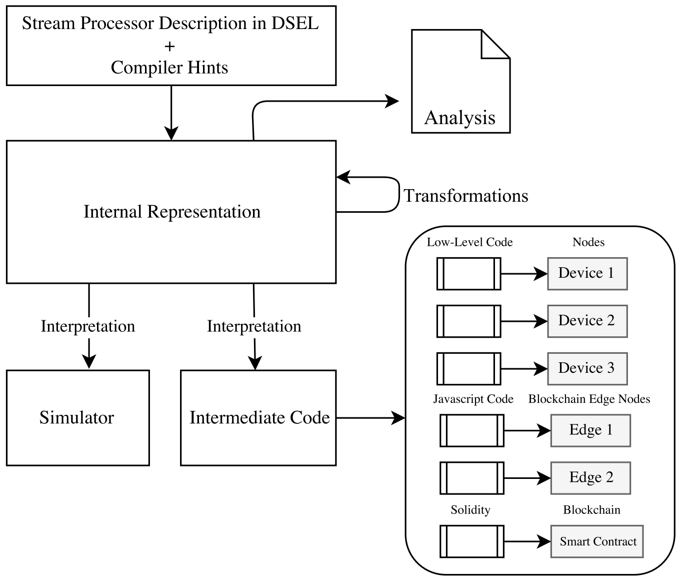

Fig. 2. The D'ARTAGNAN Framework.

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.

B. *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 our prior work with D'ARTAGNAN to be extended and 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

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

an interesting challenge for which we believe D'ARTAGNAN can contribute. Our framework allows the application logic to be placed in an optimal manner according to information provided by the programmer (i.e. hints) and the target code generated accordingly.

Listing 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 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)
```

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 were introduced to 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 [7]. 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.

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

As mentioned at the beginning of this section and depicted in Figure 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 2 which extends our previous example `app` with two sensors.

Listing 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 steps
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**: Where possible, code is placed on IoT devices to provide in-network processing and filtering. In the example above, the code for steps 3, 4 and 5 is 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**: 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]

Different applications may have different requirements. The D'ARTAGNAN framework allows the programmer to use placement directives to influence how business logic is placed across the network (see Listing 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 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, we intend to add compiler directives (such as 'low-communication','low-gas' or 'high-trust') and placement algorithms such that the programmer does not need to explicitly indicate where application logic should be placed, but is handled by the framework.

IV. USE CASE: END-TO-END SMART RENT MANAGEMENT

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 the optimal rate is one of the biggest challenges to maximise on profits. A high day-rate may mean less nights are booked, whereas a low day-rate could possibly lead to financial losses, or small margins if tenants are high consumers of commodities. A potentially good solution may be a fixed low daily rate, which attracts low-budget travellers and increases nights booked, 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 units, attract an additional charge.

To illustrate the basic concepts of the D'ARTAGNAN framework we use a simple application (see Listing 4) which shows a simple smart-rent application that calculates electricity consumption cost. 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.

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 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
```

V. 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 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).

	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 I
LINES OF CODE COMPARISON

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 I, 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 devices, our evaluation in [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, 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 II
CONSUMPTION COMPARISON FOR DIFFERENT PLACEMENT OPTIONS

To calculate consumption (Table II), 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 III-B) 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 [8]. 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, optimal placement of application logic across both IoT and Edge devices (for low-trust scenarios) will be introduced by taking into consideration both radio messages and computation complexity.

VI. Discussion and Conclusions

In this paper 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 within this paper include: our other work on allowing for IoT device behaviour to be manipulated based upon the contents of smart contracts by making use of a virtual machine that allows for switching from expensive 256-bit operations to lower bit operations [2]; and work that enables programmability of blockchain connected edge nodes using a control systems approach [9]. Other related work includes: IoT devices making use of the blockchain as a means of storing data [10]; defining virtual resources within IoT device firmware that can be instructed to execute a sequence of function invocations [11]; and making use of the blockchain to store IoT firmware [12].

Several macroprogramming solutions for wireless sensor networks have been proposed over the past decade. Regiment [4], Wavescript [13] and Flask [7] 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. However, our current restriction to stream processing systems can be a handicap in some scenarios, and we are considering means of providing other combinators allowing for a wider range of applications for which D'Artagnan can be considered a suitable language.

References

[1] G. Wood, "Ethereum: A secure decentralised generalised transaction ledger," *Ethereum Project Yellow Paper*, vol. 151, pp. 1–32, 2014.

[2] J. Ellul and G. 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*. IEEE, 2018, pp. 1–4.

[3] N. Atzei, M. Bartoletti, and T. Cimoli, "A survey of attacks on ethereum smart contracts (sok)," in *POST*, ser. Lecture Notes in Computer Science, vol. 10204. Springer, 2017, pp. 164–186.

[4] R. Newton, G. Morrisett, and M. Welsh, "The regiment macroprogramming system," in *2007 6th International Symposium on Information Processing in Sensor Networks*, April 2007, pp. 489–498.

[5] A. Mizzi, J. Ellul, and G. Pace, "D'artagnan: An embedded dsl framework for distributed embedded systems," in *Proceedings of the Real World Domain Specific Languages Workshop 2018*. ACM, 2018, p. 2.

[6] C. Elliott, S. Finne, and O. De Moor, "Compiling embedded languages," *Journal of functional programming*, vol. 13, no. 03, pp. 455–481, 2003.

[7] G. Mainland, G. Morrisett, and M. Welsh, "Flask: Staged functional programming for sensor networks," in *ACM Sigplan Notices*, vol. 43. ACM, 2008, pp. 335–346.

[8] G. J. Pottie and W. J. Kaiser, "Wireless integrated network sensors," *Communications of the ACM*, vol. 43, no. 5, pp. 51–58, 2000.

[9] A. Stanciu, "Blockchain based distributed control system for edge computing," in *2017 21st International Conference on Control Systems and Computer Science (CSCS)*, May 2017, pp. 667–671.

[10] S. Huh, S. Cho, and S. Kim, "Managing iot devices using blockchain platform," in *2017 19th International Conference on Advanced Communication Technology (ICACT)*, Feb 2017, pp. 464–467.

[11] 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)*, Dec 2016, pp. 116–119.

[12] A. Boudguiga, N. Bouzerna, L. Granboulan, A. Olivereau, F. Quesnel, A. Roger, and R. 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.

[13] R. R. Newton, L. D. Girod, M. B. Craig, S. R. Madden, and J. G. Morrisett, "Design and evaluation of a compiler for embedded stream programs," in *Proceedings of the 2008 ACM SIGPLAN-SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems*, ser. LCTES '08. New York, NY, USA: ACM, 2008, pp. 131–140.