Porthos: Macroprogramming Blockchain Systems

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

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

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

Abstract: The rise of blockchain technology has paved the way for an increasing number of blockchain systems, each having different characteristics. The need for distributed applications that span across multiple blockchain systems is increasing and in particular, there is a need for the ability to write a single-description smart contract which can be compiled to span across multiple blockchain systems.

In this paper we present Porthos, a macroprogramming framework and domain specific language for writing commitment-based smart contracts that span multiple blockchain systems. The language allows programmers to write smart contracts at a higher level of abstraction by composing together contract blocks, without the need to specify how logic should be split across different blockchain instances. A runtime framework, including both on-chain and off-chain functionality, harmonises the features of different blockchain systems as well as enables communication across the smart contracts. A proof of concept, built on the Ethereum and Hyperledger Fabric blockchain systems and extendible to other systems, illustrates the technique and framework. We also show how the Porthos language is expressive enough to define a variety of applications.
Porthos: Macroprogramming Blockchain Systems*

Adrian Mizzi  Joshua Ellul
University of Malta, Malta  University of Malta, Malta
adrian.mizzi.00@um.edu.mt  joshua.ellul@um.edu.mt

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

Abstract: The rise of blockchain technology has paved the way for an increasing number of blockchain systems, each having different characteristics. The need for distributed applications that span across multiple blockchain systems is increasing and in particular, there is a need for the ability to write a single-description smart contract which can be compiled to span across multiple blockchain systems.

In this paper we present Porthos, a macroprogramming framework and domain specific language for writing commitment-based smart contracts that span multiple blockchain systems. The language allows programmers to write smart contracts at a higher level of abstraction by composing together contract blocks, without the need to specify how logic should be split across different blockchain instances. A runtime framework, including both on-chain and off-chain functionality, harmonises the features of different blockchain systems as well as enables communication across the smart contracts. A proof of concept, built on the Ethereum and Hyperledger Fabric blockchain systems and extendible to other systems, illustrates the technique and framework. We also show how the Porthos language is expressive enough to define a variety of applications.

*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 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. We predict 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. A single application may handle payments on the Bitcoin network [Nak08], Ethereum [Woo14] for public interactions and Hyperledger Fabric [Cac16] for specific private point-to-point interactions. Implementing such multi-chain DApps is non-trivial. A 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 requires the use of strategies such as relays, notaries or atomic swaps [But16]. Cosmos [cos] and Polkadot [pol] 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. Other solutions available today provide a one-way communication channel between two blockchain systems — BTCRelay [Eth17] is an Ethereum contract used to verify, store and inspect incoming Bitcoin blocks.

What is currently missing 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 [MMW08, MEP18a, GGG05, NMW07, KGMG07]. 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.

Rather than using an existing language, we propose a domain specific language (DSL) for defining commitment-based smart contracts [MHS05]. DSLs provide a higher level of abstraction than general purpose programming languages and are ideal to make it possible to write a smart contract in a quick and efficient way, without sacrificing security. To overcome the initial investment needed to develop a language [Hud98], 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 such that it becomes possible to use a fully-fledged programming language to support the domain specific notions in the DSL [CP02].

Using this 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 support other systems. The Turing-incomplete language allows a programmer to describe commitment-based smart contracts that may span multiple blockchain systems. Our language is inspired from financial contracts of Peyton Jones [PJES00] and the work done with Marlowe [ST18], but we extend this to contracts that span across multiple chains.

The contribution of this paper 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.

\section{Background and Related Work}

A smart contract is a program that runs on a blockchain system. The contract can encode a set of rules which determine when and under what conditions transfer of assets may occur. Smart contracts can be used to implement a wide range of applications, according to the capabilities of the underlying blockchain system and the programming language used.

\(^1\)Available at https://github.com/adrianmizzi/porthos-1
2.1 Languages

Bitcoin was the first blockchain implementation. Bitcoin Script, the programming language used to write smart contracts 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. For example, a hash of the spender’s private 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.

To overcome the limitations of Bitcoin, a new generation of blockchain systems emerged. The Ethereum Virtual Machine (EVM) 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. As a result of the richer set of operations, Ethereum is much more powerful than Bitcoin and supports a wider range of applications. Solidity [HK18], the most popular language on Ethereum, uses an imperative-style programming paradigm where contract intermediate state is managed explicitly by the programmer. The increased expressivity available on the Ethereum platform has increased multifold the risk of bugs in smart contracts as programs become more complex to reason about and implement. History has shown that even code reviews performed by the most experienced smart contract programmers have failed to detect complex security flaws — the DAO hack and other high-profile heists are evidence of this [LCO+16].

Different programming paradigms have emerged to address the risks associated with unrestricted languages such as Solidity. Explicit state transition languages, including Scilla [SKH18], Rholang [MPS+18], Bamboo [Hir18] and Obsidian [Cob17], 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. Functional programming paradigms are used in Vyper [But], Simplicity [O’C], Bamboo [Hir18] and Pact [Pop]. 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 [BKT17] where an Ethereum smart contract is used to execute Findel contracts (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 [PE16], Frantz and Nowostawski [FN16] and in Marlowe [ST18].
The PORTHOS language is a domain specific language inspired by Marlowe. It is embedded in Haskell and the language constructs can be combined with Haskell primitives for additional expressivity. In the same way as Marlowe, the PORTHOS language is compositional, in that complex contracts are built by connecting simpler contracts. However, while Marlowe is intended for the Cardano [car] blockchain system, PORTHOS is intended to be deployed on multiple different blockchain systems such as Ethereum and Hyperledger Fabric.

2.2 Macroprogramming

Macroprogramming is a technique mostly used in the Internet of Things (IoT) and sensor networks domain. Through a higher level of abstraction, the network is programmed as a whole rather than through the programming of the individual components and the communication between them. Macroprogramming languages provide flexibility to allow for a wide range of applications where information may flow between different nodes of the network, and can be processed directly in the network.

There are three main approaches for macroprogramming. The first approach, used in Pleiades [KGMG07] and Kairos [GGG05], is one where the programmer has a centralised view of the network and each node can be addressed individually. The second approach, as used in Regiment [NMW07] and Flask [MMW08], is one where macroprograms are written from the perspective of individual nodes and all nodes get a copy of the same code. In the third approach, a macroprogram is written from the network perspective and the generated code is different for the individual nodes to reflect their role in the application. This latter approach is ideal for heterogeneous networks as shown in D’ARTAGNAN [MEP18a, MEP18b].

PORTHOS follows an approach which is similar to D’ARTAGNAN, 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.

2.3 Interoperability

A requirement that is becoming increasingly important is the ability to execute DApps across multiple blockchain systems. Vitalik Buterin [But16] 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 swapping assets across separate 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.

Cosmos [cos] and Polkadot [pol] are two upcoming solutions in the blockchain interoperability space aiming to create a network of blockchains with a relay-based strategy. To date, both systems are still in development stage but are expected to be generally available later in 2019.

In Porthos, we provide a notary scheme type of interoperability where a trusted group of entities react to events by triggering other smart contracts. Interoperability of blockchain systems is not the main focus of our work, and a basic implementation is only provided to enable a proof-of-concept of macroprogramming for blockchain systems. In the future, we expect other technologies such as Cosmos and Polkadot to replace our off-chain runtime environment, as these become available and offer a more comprehensive solution.

3 The Porthos Framework

Traditionally, smart contracts are written to be executed on a specific blockchain system. Interactions between smart contracts located on different blockchain systems requires complex mechanisms to be implemented — including atomic swaps, notary schemes and relays [But16]. 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 need only focus on the overall logic of the smart contract using only one programming language.

Our aim is to use techniques from the field of embedded languages to define an embedded domain specific language. We embed our language 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 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.
3.1 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.

3.1.1 Blockchain Extensions

A blockchain system can be supported in the PORTHOS framework 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, which is capable of holding assets, is not supported. Other blockchain systems which satisfy the requirements are supported — however, different blockchain 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. Smart contracts must be called by an external participant or by another contract (which has been ultimately called by an external participant). On some platforms, third party solutions already exist to extend this functionality — for example, Ethereum Alarm Clock [eth] and Oraclize [ora] where a contract function can be called back at a specific time on the Ethereum blockchain. For our proof-of-concept, a blockchain-specific extension is used to complement existing functionality. Each extension is specific to that particular blockchain system, and third party solutions (e.g. Ethereum Alarm Clock) may be used as part of that extension.
3.1.2 Message Routing

Blockchain systems are unable to communicate together 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 blockchain 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 (as described in Section 2.3), 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. Messages are signed by the originating blockchain system, and validated by the receiving blockchain system before being processed. Duplicate messages are not processed multiple times. 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 the future, the communication layer may be replaced with state of the art solutions for blockchain interoperability, such as Cosmos and Polkadot\(^2\) as these become generally available and offer a more comprehensive solution capable of offering interoperability between several different blockchain systems.

3.2 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. Different strategies exist for slicing code including (i) an execution-cost optimised strategy — executing code on some blockchain systems may be more expensive\(^3\) 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.

\(^2\)Cosmos and Polkadot solutions are expected to be live later in 2019

\(^3\)Some blockchain systems have the notion of gas, a unit of consumption for executing smart contract logic
In our proposal, we use a location-based placement strategy because it 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.

### 3.3 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.

### 4 Porthos as a smart contract language

**Porthos** is a domain specific language for composing commitment-based smart contracts [dKW17]. 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.

**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.

As a simple example to introduce the PORTHOS language we show how a simple savings-plan contract is implemented by composing constructs together (see Listing 1). 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.

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

Listing 1: A Time-Locked Savings Plan

The implementation of the time-locked savings plan is made up by combining three basic constructs: `repeatCommit` followed by `releaseAll` and finally `end`. 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 the cryptocurrency Ether (denoted by ETH) 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.

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 2) 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
swap :: (Participant, Asset Currency) \rightarrow 
(Participant, Asset Currency) \rightarrow 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))

Listing 2: An Asset Swap Contract

crowdFunding :: Participant \rightarrow (Currency, Currency, Float) \rightarrow Asset Currency \rightarrow Contract

crowdFunding recipient (x, y, f) targetY = both (campaignX, campaignY) .>>>.

  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)

Listing 3: Crowd funding across multiple assets

groupPay :: [(Participant, Asset Currency)] \rightarrow Participant \rightarrow 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

Listing 4: Group Pay
committed must be as specified by $a_1$. The commitment from Participant 2 ($p_2$) must be in $\text{XYZ}$ (a fictitious cryptocurrency or asset type), the recipient must be Participant 1 ($p_1$) and the quantity as specified by $a_2$. 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.

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 multi-asset crowd funding campaign, where deposits are made with assets located on different blockchain systems (see Listing 3), an exchange rate is used to determine if the overall total meets the declared target. Contracts can be composed sequentially with the \'.>>>.' (followed-by) operator.

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 4), 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.

5 Use Case: 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 3.1, 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.
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 (whereRecipientIs(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))

Listing 5: A property sale agreement
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.

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 5. 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 6). Assets may all be located on the same blockchain (in that case, only one smart contract is generated), or alternatively located on different blockchains.

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

data Currency = ETH
instance AssetType Currency where
  chainOf ETH = "Ethereum"
```

Listing 6: Asset location information

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.
6 Evaluation

We evaluate PORTHOS in three parts. First, we evaluate the programming abstraction by showing that it is expressive enough to implement a variety of commitment-based smart contracts. Second, we evaluate the security aspects of the proposed model. Finally, we evaluate the framework’s extensibility.

6.1 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 paper, we have shown a number of smart contract applications where assets can be traded — crowd funding, group payments, asset swapping and a property sale agreement. 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 a prior commitment 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.

Unlike other smart contract languages (e.g. Solidity), the language is Turing incomplete so some applications are not describable — that is, applications involving loops are not possible. 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.

6.2 Security Analysis

Blockchain systems have a high level of security, despite the absence of a central authority, due to their decentralised and immutable characteristics. Third parties cannot alter contract state or interfere with contract execution. However blockchain systems are still not completely immune to attacks or weaknesses. For example, a 51% attack can alter stored data and influence the behaviour of a blockchain system, creating opportunity for attackers to double-spend funds. Also, oracles provide infor-
mation to smart contracts, but if this information is falsified, then the behaviour of the smart contract will be incorrect.

Applications that span multiple blockchain systems may introduce new attack vectors and security weaknesses. The weaknesses described here are applicable to any multi-chain application, rather than specifically to the macroprogramming approach proposed in this paper — 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 between blockchain systems must be signed by the originating blockchain system. During application initiation stage, cryptographic keys are exchanged between chains such that messages can be verified.

**Distributed Transactions** – Transactions that span multiple systems are susceptible to situations where a part of the transaction fails, 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.

### 6.3 Extensibility

The Porthos framework is extensible in different directions. First, different types of assets are supported as long as a contract interface is implemented for the respective asset. Secondly, the model can be extended to support new blockchain systems by
adding code generation from the intermediate representation to the target language, and by 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 3.1.1 are satisfied.

7 Discussion and Conclusions

In this paper we presented PORTHOS, an embedded DSL framework for describing commitment-based smart contracts that span across 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 within this paper include: D’ARTAGNAN for programming IoT devices at a network level [MEP18a] and for writing a single macroprogram for blockchain connected IoT devices [MEP18b]; Marlowe for specifying financial contracts on Cardano [ST18].

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.

References


[car] Cardano.


[cos] Cosmos.


[eth] Ethereum alarm clock.


[ora] Oraclize.


Polkadot.


