UNIVERSIDADE DE LISBOA
FACULDADE DE CIÊNCIAS
DEPARTAMENTO DE INFORMÁTICA


# SmartScribble, a protocol language for Plutus smart contracts

Alexandre de Alegria Junceiro Mascarenhas Monteiro

**Mestrado em Engenharia Informática**

Dissertação orientada por:
Prof. Doutora Andreia Filipa Torcato Mordido
e co-orientado pelo Prof. Doutor Vasco Manuel Thudichum de Serpa Vasconcelos

2022

# Agradecimentos

Gostaria de começar por agradecer à minha família. Graças ao esforço que fizeram para eu poder continuar a estudar, consegui chegar a este ponto onde estou na minha vida. Sem eles, eu não estaria aqui neste momento a terminar esta fase da minha vida, e por isso dedico-lhes este trabalho.

Agradeço aos meus amigos que conheci e que estiveram comigo até este momento desde o início do meu percurso académico. Juntos enfrentamos todas as dificuldades que passámos durante estes anos. Agradeço por me animarem quando mais precisei, por me apoiarem sempre, e por estarem sempre do meu lado.

Agradeço também à minha namorada por todo o apoio que me deu. Agradeço-lhe por me ter apoiado todos estes anos, incluindo os meus anos de faculdade, por estar sempre comigo e por me ter ajudado sempre quando mais precisei.

Por fim, agradeço aos meus orientadores, Professora Andreia Mordido e Professor Vasco Vasconcelos, pelo suporte que me foi dado nesta tese. Agradeço os conhecimentos que me transmitiram e a paciência que tiveram comigo. E agradeço também ao Afonso Falcão, o pai do SmartScribble, por me ter deixado tirar dúvidas sobre o trabalho dele.

*À minha família, amigos e namorada.*

# Resumo

Uma *blockchain* consiste numa rede descentralizada, cujos nós armazenam a informação que flui através da rede. Uma das aplicações mais populares das *blockchains* é a transacção de criptomoedas. *Smart contracts* permitem a um programador criar programas para a *blockchain*, normalmente utilizados para lidar com activos valiosos. No entanto, como qualquer outro *software*, os *smart contracts* são propensos a vulnerabilidades e são alvos de ataques. Um exemplo famoso é o ataque do DAO [18] que ocorreu em 2016 na rede Ethereum e que resultou no roubo de aproximadamente 70 milhões de Dólares. Os ataques recentes a *smart contracts* levaram os fornecedores de *blockchain* a concentrarem-se no desenvolvimento de linguagens de programação mais seguras. Exemplos de linguagens de programação baseadas em verificação formal são o Move da Facebook [2] e o Plutus da IOHK [38, 22]. No entanto, Falcão et al. [17] identificou várias falhas no Plutus.

O SmartScribble [16], é uma linguagem de protocolos que gera *smart contracts* e tem como objetivo aumentar a fiabilidade destes contratos. Com base num protocolo especificado pelo programador, esta linguagem gera código para *smart contracts* escritos na linguagem Plutus, a linguagem de programação nativa da *blockchain* da Cardano. Para assegurar a execução do protocolo pela ordem correta, o SmartScribble recorre a máquinas de estados. Dado um protocolo, o compilador gera uma máquina de estados e incorpora-a no *smart contract* que é criado. A incorporação da máquina de estados permite garantir a ordem correta de interações com o *smart contract*: a invocação de uma interação, também conhecida como *endpoint* em Plutus, é impedida pela máquina de estados se não estiver prevista nas transições do estado atual. O SmartScribble gera automaticamente os *imports* necessários para o funcionamento do contrato, as declarações de alguns tipos de dados e funções auxiliares, o código necessário para o funcionamento da máquina de estados, e as funções de endpoints para cada interação do protocolo a ser compilado. O *smart contract* seguidamente tem de ser completado manualmente com a lógica de negócio de cada *endpoint*. Para cada *endpoint*, também é gerada uma função auxiliar que é responsável pela respetiva lógica de negócio.

O SmartScribble foi introduzido em 2021. Dado que é uma proposta recente, a linguagem ainda está num estado embrionário e não permite a especificação de protocolos mais complexos. E por outro lado, algumas das características da linguagem ainda não estão completamente implementadas, não tendo efeito no código gerado ou bloqueando as interações de alguns utilizadores.

Neste trabalho foram acrescentadas várias funcionalidades ao SmartScribble de modo a oferecer uma versão mais robusta e completa da linguagem. A primeira funcionalidade que integrámos

na linguagem foi a validação dos *roles*. Num protocolo, os *roles* determinam quem está a participar num protocolo e quem é que pode chamar um determinado *endpoint*. Apesar de terem sido introduzidos desde o início na sintaxe, os roles ainda não são considerados no código gerado. Na nossa solução, habilitamos a linguagem com a especificação e validação de *roles*. Os *roles* podem representar uma única entidade que tem determinadas permissões. Mas pode também representar várias entidades que estão a agir na posição desse *role*. No exemplo do *Guessing Game* apresentado por Falcão et al. [16], existem dois roles: o *Owner* e o *Player*. O *Owner* é só um sujeito que está a controlar o jogo, e o *Player* são várias entidades que estão a jogar e a competir pelo prémio do *Owner*. Como as *blockchains* recorrem a criptografia assimétrica, neste trabalho propomos o registo das chaves públicas das *wallets* dos utilizadores que estão a interagir com o contrato para posterior validação. As chaves públicas das *wallets* são guardadas no contrato, e com a ajuda de funções oferecidas pelo Plutus, é validado se um *endpoint* está a ser chamado por uma chave pública válida.

A máquina de estados que é atualmente criada é global, o que significa que o estado atual é partilhado e todos os utilizadores estão e transitam para o mesmo estado, ao mesmo tempo. A especificação de máquinas de estados individuais foi outra funcionalidade que implementámos. As máquinas de estados individuais permitem a um utilizador estar no seu próprio estado independentemente dos outros utilizadores. O programador especifica um segundo protocolo no mesmo ficheiro para o SmartScribble gerar as máquinas de estados individuais. O protocolo que é especificado para máquinas individuais, utiliza a mesma sintaxe para especificar as interações necessárias. A máquina global é a máquina que já era gerada para o contrato, e a máquina individual é gerada com os tipos de dados que representam os estados e *inputs* dessa máquina, juntamente com a sua própria função de transição. Dentro da máquina global mapeamos as chaves públicas às respetivas máquinas de estados, juntamente com o estado atual. Estes estados podem guardar outras informações do utilizador necessárias para o seu protocolo individual.

Na implementação atual, o contrato Plutus tem de ser completado manualmente com a lógica de negócio. A lógica de negócio consiste em poucas linhas de código, em comparação com o resto do contrato que apresenta um elevado número de linhas de código. Neste trabalho propomos a especificação da lógica de negócio diretamente no protocolo. Deste modo, o SmartScribble irá gerar o contrato completo, não sendo necessário acrescentar ou modificar o código manualmente. Com o auxílio de Template Haskell, permitimos integrar expressões em Plutus dentro do protocolo (escrito em SmartScribble). O compilador irá ler estas expressões, e incorporá-las no local correto do contrato. O código que é escrito no protocolo é redirecionado para as funções de lógica de negócio. Para facilitar o retorno destas funções, o compilador cria funções auxiliares no seu esqueleto. Estas funções auxiliares oferecem uma forma mais fácil de retornar e torna o código mais fácil de ler.

Desde a origem do SmartScribble que os *triggers* desempenham um papel importante: um *trigger* permite interromper o fluxo de um bloco de interações quando uma condição é verificada. Na implementação original, um *trigger* só pode reagir a um de dois fatores do contrato: tempo ou

dinheiro. Além disso, o registo de um *trigger* impede o utilizador de interagir com o contrato. O utilizador só consegue voltar a interagir com o contrato quando a condição do *trigger* é satisfeita. Neste trabalho, propomos uma alteração à sintaxe (e compilador) que tornam possível a especificação de *triggers* mais complexos. As condições agora podem não só reagir a tempo e dinheiro ao mesmo tempo, e assim como também pode reagir aos dados guardados no contrato. Estas podem ser especificadas dentro do protocolo com o uso de expressões Plutus. Na solução apresentada neste trabalho restruturamos o modo como os *triggers* são integrados no contrato.

O protótipo resultante deste trabalho constitui uma versão mais robusta do SmartScribble [1]. Permitimos agora que protocolos mais complexos sejam especificados. Garantiu-se que os *endpoints* são chamados pelos utilizadores corretos. Foi possível gerar o contrato completo sem a necessidade de intervenção manual. E os endpoints já não impedem os utilizadores de interagirem com o contrato devido ao efeito dos triggers.

**Palavras-chave:** Linguagem de programação, Smart contracts, Máquina de estados, Protocolo de comunicação, Geração de código

[1] https://git.lasige.di.fc.ul.pt/amordido/smartscribble

# Abstract

A blockchain consists of a decentralized network whose peers store the information that flows through the network. One of the most popular applications of blockchains is the transaction of cryptocurrencies. Smart contracts allow a developer to create programs for blockchains, commonly used to handle valuable assets. However, like any other software, smart contracts are prone to vulnerabilities. SmartScribble [17] is a protocol language aimed at generating more reliable smart contracts. A programmer specifies a protocol, and SmartScribble automatically generates a smart contract and a state machine to validate the parties' interactions with the contract. In its original formulation, a working contract is obtained after the manual insertion of the business logic in the generated contract.

SmartScribble relies on state machines to ensure correct interactions, triggers for control flow, namely to exit loops, and roles to identify the participants. However, the language still lacks many essential features. In this work, we enrich SmartScribble, making it more reliable and expressive.

We start by enabling the validation of roles which was not present in the original version. We extend the expressivity of SmartScribble's triggers by generalizing the original definition that was restricted to one of two factors (money or time) to allow mixed conditions with time and money. To enhance usability, we extended SmartScribble's syntax to enable the definition of the business logic in the protocol specification. Finally, to improve the expressivity of SmartScribble, we introduce more expressive state machines that enable a more refined specification of protocols.

The evaluation shows that our extensions compare pretty well in performance when compared to the original version. The slight increase in performance is a fair price to pay for reliability.

**Keywords:** Programming language, Smart contracts, State machine, Communication protocol, Code generation

# Contents

# List of Figures

# List of Tables

# Glossary

**ADA** Cardano's cryptocurrency.

**Boilerplate code** Skeleton code with the necessary imports, data type declarations, and endpoint's skeletons.

**CPU** Central Processor Unit.

**eDSL** Embedded Domain-Specific Language.

**Endpoint** Plutus smart contract interface exposing a communication channel.

**EUTxO** Extended Unspent Transaction Output.

**Fee** Cryptocurrency cost to perform a transaction.

**LOC** Lines of Code.

**Lovelace** ADA's unit. One ADA is equal to one million Lovelace (1 ADA = 1.000.000 Lovelace).

**UTxO** Unspent Transaction Output.

# Chapter 1

# Introduction

Smart contracts are computer programs that can implement decentralized applications among several parties, such as auctions, crowdfunding campaigns, or even games [1, 17, 50]. Several blockchains are endowed with their own programming languages, for smart contract development [2, 41, 47]. Ethereum is the most famous blockchain for developing smart contract; its native programming language is Solidity [3, 8].

The popularity of smart contracts in Ethereum revealed the major weakness of programming directly over assets: a faulty contract can cause a significant loss of money due to bugs present in the code. A famous example is the DAO attack, which occurred in 2016, and resulted in a theft of approximately 70 million USD due to the exploitation of vulnerabilities in a smart contract [18].

The recent attacks against smart contracts led blockchain providers to focus on developing safer and stronger programming languages. Examples of programming languages based on formal verification are Move by Facebook [2] and Plutus by IOHK [38, 22]. However, Falcão et al. [17] identified vulnerabilities in Plutus.

SmartScribble [17] is a protocol language introduced in 2021 with the aim of increasing the reliability of Plutus smart contracts. It is based on state machines to ensure the correct order of interactions with a smart contract. Given a protocol description, SmartScribble generates the contract code, creates the respective state machine, and integrates it into the generated contract; the developer only needs to define the business logic to get a fully working contract.

According to Falcão et al. [16], SmartScribble distinguishes itself from other languages by using a protocol specification, abstracting the specification of the interactive behavior and side stepping the details of the target language. It also reduces the learning curve by using the smart contract language only to implement the business logic.

SmartScribble is a recent proposal and still presents some limitations for smart contract development such as:

- In a protocol specification, the developer can specify which role can call a specific endpoint, but the roles are still useless in the current version of SmartScribble. They are not verified nor validated;

- The created state machine is global and all users share the same state. Two different users

cannot be in different states.

- The smart contract's business logic is implemented manually after the boilerplate is generated by SmartScribble, which is not the ideal scenario;
- The developer can define triggers that activate with time or funds in the contract, but they are limited to only one of those two factors, which precludes from defining a trigger that results in a combination of these two factors.

In this work, we propose and implement an extension of SmartScribble that addresses these limitations, more specifically:

- Enables to specify and validate the users' roles.
- Enables the implementation of state machines for different clients.
- Incorporates the business logic in the protocol specification (and generated code).
- Enables the specification and implementation of more advanced triggers.

This document is organized as follows:

**Chapter 2** We present the basic concepts underlying this work: blockchain and smart contracts, we also describe some known attacks against smart contracts. We present the Scribble protocol language, which is the language in which SmartScribble was based on. We also present some related programming languages for smart contracts and complementary tools.

**Chapter 3** We present the motivation of our work, mention some features that are missing in SmartScribble and present some examples.

**Chapter 4** We propose some extensions to SmartScribble to address the limitations presented in Chapter 3. This chapter presents the main contributions of this work and provides a thorough description of the methods used to implement each extension.

**Chapter 5** We provide a thorough description of SmartScribble's syntax, considering the extensions implemented in Chapter 4.

**Chapter 6** We analyze each extension in action applied to some particular smart contracts. Then we provide an evaluation of performance of our solution, comparing it with the performance of the original implementation.

**Chapter 7** We conclude the work and present some proposals of future work for SmartScribble.

# Chapter 2

# Related work

This section contains the basic concepts required to understand our work as well as a brief overview of the related work.

## 2.1 Blockchains

A blockchain is a peer-to-peer network [50] where the peers record the information that flows through the network into a ledger. Usually, the content in the ledger is public [50, 21]. This technology has many applications and is particularly popular in the world of cryptocurrencies [50] where the ledger stores the transactions submitted by the network parties.

Users submit transactions that are broadcasted to the network peers. The peers generate blocks containing these transactions. All peers reach an agreement through a consensus protocol to decide the next state of the blockchain.

There are many types of blockchains. Herlihy et al. [21] mention two types:

- **Permissionless (or Public)**: Parties are unknown and anyone can participate in the consensus. Most cryptocurrencies reside in public blockchains.
- **Permissioned (or private)**: Parties are identified and must have permission to join the network. Private blockchains are preferred for business applications.

These types of blockchains have their own consensus protocols. Two famous protocols are mainly used in permissionless blockchains. They are:

- ***Proof of Work*** **(PoW)**: Peers are known as miners. Miners compete with each other to find a new block to add to the ledger, and this is done by using their computational resources to solve a cryptographic puzzle. When one of the miners first solves the puzzle, its block is broadcasted to the other miners for validation. If most miners validate the new block, all the miners append the new block to their copy of the ledger, the transactions within are confirmed [7], and the miner is rewarded for finding a new block [12, 42].
- ***Proof of Stake*** **(PoS)**: Users have to spend an amount of cryptocurrency to pay the validation of transactions and create new blocks to the blockchain. Moreover, just like in PoW, the

block that has been found is broadcasted and added to the blockchain if the majority of nodes approve it (according to some consensus protocol) [24, 42].

The blockchain technology makes use of asymetric cryptography. In this type of cryptography, users possess a public key that can be known by anyone and a private key that only the owner knows. Only the user that owns the respective private key can read the message encrypted with the respective public key. On the other hand, when encrypting (on signing) a message with a private key, it is possible to faithfully identify the user that sent the message [40]. Asymmetric cryptography is used in blockchains to ensure the integrity and identification of the transactions.

### 2.1.1 Ethereum

Ethereum is a blockchain platform that provides a native Turing-complete programming language for smart contract development, Solidity [3, 8]. This language is widespread but also the primary target for attacks against smart contracts, which motivates this and many other works. The consensus protocol used in this network is *proof of work*, where the miners, as mentioned in Section 2.1, compete against each other to solve a cryptographic puzzle and to add a new block to the chain. However, Ethereum has plans to switch to the *proof of stake* protocol [15].

Ethereum follows an account-based model, where users possess accounts that are used to store their balance of Ether, the currency of Ethereum. An account in Ethereum can either belong to a human user or a smart contract deployed into the blockchain. In the case of smart contracts, accounts also contain the contract's code.

When a user submits a transaction for funds transfer, for example, the information of all the accounts involved is updated. Transactions can also be used to deploy new contracts or to execute a function from a deployed contract [11]. Ethereum transactions use a fee called *gas*. Before submitting a transaction, the user must pay an initial gas value to be used during the execution of the transaction. When the transaction has been completed, the remaining gas is refunded to the user that submitted the transaction. However, if the gas ends during the execution, all that has been done is reverted.

### 2.1.2 Cardano

Cardano is the blockchain ultimately hosting the protocol language that constitutes the basis of this work. Cardano is a blockchain platform born from scientific development [22, 36]. This blockchain hosts the Ada cryptocurrency and enables the development of smart contracts. It was designed with security, scalability, and interoperability as its main principles; its consensus protocol is based on *proof of stake*.

The Cardano blockchain uses the Extended Unspent Transaction Output model (EUTxO), an extension of the Unspent Transaction Output model (UTxO). Imagine that you have a 100€ bill, and you want to give 40€ to a friend. What will you do? You are not going to cut the bill with a scissor. So you give your friend the 100€ bill, and you receive back a change of 60€. In the UTxO model, the scenario is similar, and users have their balance of Ada (the currency of

Cardano) represented in the form of unspent outputs. Like the previous example, you have 100 Ada in your account, and your friend has 20 Ada. If you want to give 40 Ada to your friend, you can't split your output into different values, so a transaction will be created, and it will receive as input your output of 100 Ada. That output will now be spent in the transaction, and two new outputs will be created, one with 40 Ada for your friend and another with 60 Ada for you, which is the change. Now you have 60 Ada, which is your 60 Ada output, and your friend has 60 Ada as well because your friend now possesses two outputs, one with his initial 20 Ada and another with the 40 Ada he received. The outputs in this model consist of a pair of a validator and a value. In the extended UTxO model, the outputs, besides the validator and the value, also contain some arbitrary data known as the datum. When a transaction attempts to spend an output, the validator uses the datum, the value, the transaction, and some arbitrary data for validation, known as the redeemer. The validator verifies if the transaction using the redeemer can spend the output [4]. With the extended UTxO model, it is possible to implement state machines, which is paramount to integrate since the integration of one machine in each contract is required.

## 2.2 Smart contracts

The blockchain technology can be applied for many purposes; one of them is the use of smart contracts. Smart contracts are programs that establish an agreement among two or more parties without the need of a third party [50]. The contract defines the rules of the agreement, and the execution of the contract ensures that nobody involved can violate the rules that were determined in the agreement [38]. With smart contracts, programmers can implement different decentralized applications such as auctions, crowdfunding campaigns, and even games.

In Ethereum, a smart contract is a collection of functions and data residing in the blockchain [11]. Smart contracts are not controlled by the users. Instead, they can interact with a contract by using the functions that have been programmed.

An important feature of these contracts is the ability to transfer the virtual assets of the user (in the case of Cardano, the transference of Ada) to the contract and the assets of the contract to the users or other contracts [1].

### 2.2.1 Attacks against smart contracts

During the development of any program, it is vital to check for errors to avoid vulnerabilities or exploits. Smart contract development is not an exception. A smart contract cannot be altered after it has been published into the blockchain [50] leaving it subject to attacks from malicious users. Sayeed et al. [50] and Atzei et al. [1] have documented attacks that happened against the Ethereum network. Some of the exploits that have been documented in these surveys are the following.

**Reentrancy** attacks can happen if a function of a contract is written to use an external call to a possible vulnerable contract before any internal operations, like validation or state update [51]. This allows the attacker to exploit external calls as he desires. The attacker can

create an (un)intended loop that is repeated many times. The DAO attack resulted from this exploit [18]. Plutus can avoid a reentrancy attack because Cardano uses the K framework (see Section 2.4) [16, 55].

**Overflow and Underflow** are two similar attacks. Overflows happen when the result of an operation exceeds the maximum value that can be represented by the system. Underflow is similar, but it happens oppositely. Overflows in smart contracts allow an attacker to increase his digital assets illegally [43]. In Plutus, it is possible to exploit overflows because Cardano's virtual machine does not check for this type of vulnerability [55].

**Ether lost in transfer** can happen because transactions are sending Ether to addresses that do not belong to anyone, also known as orphan addresses [1]. In Solidity, this happens because it only checks if the addresses' length is not larger than 160 bits [16]. The programmer is responsible for validating the target address before any transaction is committed. In Plutus, funds can also be lost in this way or lost to the contract as well due to the wrong order of interactions [16]. SmartScribble can solve this vulnerability with a state machine.

**Default visibilities** is related to Solidity's specifiers. Programmers can define if a function or a variable is public, private, internal, or external. Without specifying it, the visibility is public by default. With this vulnerability, a malicious user can perform an unauthorized or unintended action. In Plutus, only the endpoints of the smart contract are visible. Other operations, like SmartScribble's triggers, are not visible. In some situations, some of the visible endpoints are meant to be called by a specific user. This scenario is addressed further on.

**Insufficient Gas Griefing** occurs in Solidity contracts that accept data for another contract's subcall [53]. This attack is achieved by providing a certain amount of gas to deliberately prevent sub-calls from succeeding. A malicious user can use this vulnerability to sabotage transactions. Cardano's transactions require a fee in ADA. However, defining the transaction's fee value for an endpoint is impossible. Hence, replicating this vulnerability in Plutus is not possible.

## 2.3 Programming languages for smart contracts

Many programming languages have been proposed for smart contracts development to tackle vulnerabilities that can result in the loss or theft of assets. Most of them rely on formal validation.

**Solidity.** Solidity is one of the most famous languages for smart contract development [5], but it is also a popular target for cyberattacks [8, 13]. In addition, several vulnerabilities have been found [1, 50] that encouraged the proposal of other solutions.

Solidity is a high-level programming language inspired in C++, Python, and JavaScript to develop smart contracts for the Ethereum blockchain. The language is Turing-complete and runs

on an environment known as the Ethereum Virtual Machine, also known as EVM for short.

**Plutus.** Plutus is a Turing-complete functional programming language developed by IOHK, embedded in Haskell, to develop smart contracts for the Cardano blockchain [22, 38] and is the target of SmartScribble. Plutus promises significant security advantages by providing more manageable and more robust means to verify the correctness of a contract.

A Plutus contract consists of two parts, the off-chain code and the on-chain code. The off-chain code is executed on the user machine to build and submit transactions to the ledger. This part of the contract is written in Haskell and uses the Plutus Library. The on-chain code is executed in the blockchain to validate transactions. On-chain code is also written in Haskell and is compiled into Plutus Core, the language used to build the scripts to perform the on-chain validation.

Just like in Ethereum, Plutus' users interact with the smart contract by calling the functions, known as endpoints in Plutus, specified in the contract. Compared to Solidity, Plutus can ensure a predictable behavior [16]. But Plutus' coding process is complex, while Solidity is easier to code. And developers are more acquainted with Solidity than Plutus. Falcão et al. [16] identified some vulnerabilities in Plutus.

**Move.** Move is a language for smart contract development for the Libra Blockchain [2]. Move ensures a trustworthy behavior of the contracts, which is also one of the objectives of SmartScribble. Move allows the definition of custom resource types and their respective business logic and allows the programmer to implement custom transactions; these transactions can interact with the created custom resources. Move also ensures that no resource is duplicated, reused, or lost. Before a programmer publishes or executes his Move contract, it must be submitted to a static verifier. The verification is done at the bytecode level and ensures that all Move programs satisfy resource, type, and memory safety.

**FSolidM.** FSolidM [41] is a framework for designing smart contracts providing a graphical interface that allows users to create a smart contract as a finite state machine easily. Given the defined state machine, FSolidM translates it into Solidity code. The language also provides plugins that provide security features against reentrancy and unpredictable state vulnerabilities and implement common design patterns to ensure the correctness of more complex contracts.

**SmartScribble.** SmartScribble is a protocol-based domain-specific language designed for smart contract development and introduced by Afonso Falcão [16]. Given a user-specified protocol, SmartScribble creates a Plutus smart contract with an integrated state machine. This language shields the programmer from Plutus' interactive behavior and technical details, while ensuring the execution of a contract in the correct order (contrarily to Plutus [17, 16]).

SmartScribble is the focus of this work, the prototype resulting of this work shall be an extended and more robust version of SmartScribble.

The syntax of SmartScribble was based on the language Scribble. Indeed, Scribble is also one of the motivations for the work underlying SmartScribble. Scribble [59] is a protocol programming language that allows a developer to specify an application-level protocol among communication systems.

The language is intended to describe the protocols of conversation for distributed applications and to generate prototype code. SmartScribble adopts Scribble's syntax, adapting it to the blockchain setting (see Chapter 5).

**Plutarch.** Plutarch is a similar proposal to SmartScribble, both generate Plutus code. While FSolidM uses a graphical interface to target Solidity smart contracts, Plutarch uses a typed eDSL in Haskell to target Plutus smart contracts [47]. Its main objective is to generate on-chain for Plutus smart contracts, focusing on CPU, memory, and script size efficiency. Currently, on Plutarch, off-chain code is added manually. Plutarch's code is written with Haskell syntax, like how Plutus is. This DSL is facing the same problem as Plutus, which consists on the lack of familiarity with functional languages that some programmers face. SmartScribble, in contrast with this DSL, offers an easy-to-learn syntax and generates both on-chain and off-chain code.

## 2.4 Additional tools

As a complement to programming languages there are several analysis tools and frameworks. In this section, we cover some tools for smart contracts.

**Manticore.** Manticore is a tool for Solidity's smart contracts [44]. This tool performs code coverage through a dynamic symbolic execution. Manticore can be used to find the code correctness of a client.

**K semantic framework.** The goal of this framework is to define the formal semantics of a programming language [49, 48]. A programmer can define configurations and rules for creating programming languages, analysis tools, and type systems. One of K's creations is IELE. IELE is deployed in the Cardano testnets and has the advantage of making reliable smart contracts easier to write [35].

**Mythril.** Mythril scans issues in Solidity smart contracts [45]. The tool retrieves the bytecode from the smart contract and executes it to explore all the possible states. When the analysis encounters a vulnerable state, it computes the inputs necessary to reach this state. This tool is helpful for determining the cause of a vulnerability and creating exploits if the user wishes to.

**Securify.** Securify is a fully automated security analyzer for Ethereum smart contracts [56]. This tool can determine the safety of the contract behavior. The analysis of a contract is executed

in two stages. First, the tool extracts the semantic information from the contract's code, then proves whether a property holds by checking compliance and violation patterns.

**Cubicle.** Cubicle is a model checker that encodes a state machine as a smart contract or a blockchain transactional model [6]. While SmartScribble creates contracts from a protocol, Cubicle takes a smart contract and turns it into a model to verify different kinds of functional properties. Cubicle models the blockchain transaction mechanism and the functions and variables of the contract under verification. This model checker is compatible with multiple smart contract languages, and the authors also provide a way to describe error traces to aid contract development.

Although verification tools are useful for safer software (and smart contract) development, we still believe that delivering a programming language with a powerfull verification mechanism results in a more reliable and friendly development environment for the programmers.

# Chapter 3

# SmartScribble has room for improvement

SmartScribble is a protocol language introduced in 2021 by Falcão et al.[17]. The language was designed to address vulnerabilities resulting from out-of-order interactions with an easy-to-learn syntax, inspired in Scribble [16]. Given a protocol specification, the SmartScribble's compiler generates the boilerplate code for a Plutus smart contract. Plutus is the native programming language of the Cardano Blockchain. SmartScribble infers a state machine from the protocol description to avoid out-of-order interactions and integrates it into the generated contract. The generated boilerplate code consists of the following elements:

- All necessary imports.
- Data type declarations.
- Code for the state machine integrated into the contract.
- Functions' signatures for each protocol interaction points, known as *endpoints* in Plutus.

Once the boilerplate is created, the programmer must complete the contract manually with the business logic code. SmartScribble is a recent proposal, and the language presents some limitations and a vast scope of features to be added. This chapter presents some examples that feed the motivation of this work.

## 3.1 Trigger conditions

In SmartScribble, triggers allow a protocol to interrupt the flow of a block of interactions. In the protocol, the programmer chooses whether the trigger will react to a time or monetary factor.

```
1  protocol Auction (role Seller, role Buyer) {
2      field Value, ByteString;
3      initialise (Value, ByteString) from Seller{
4          trigger slot closeAuctionTrigger;
5      };
6      do{
7          rec Loop {
```

```
8             bid (Value) from Buyer;
9             Loop;
10          }
11      } interrupt {
12        closeAuctionTrigger () from Contract;
13      }
14      collectFundsAndGiveRewards () from Seller;
15  }
```

Listing 3.1: Auction protocol

The protocol in Listing 3.1 creates a smart contract for an Auction smart contract. The contract stores two variables inside. The first variable, of type **Value**, is aimed at representing the current bid. The second variable, of type **ByteString**, represents the item being auctioned. The protocol starts with the initialise endpoint. The auction Seller calls this interaction to register the item to be sold and its initial value. Once the auction has been initialized, the Buyers can offer the highest bids. In line 4, the initialise endpoint is registering a trigger named closeAuctionTrigger. The keyword slot in the **trigger** statement indicates that the condition reacts to time. When its condition is met (line 11), no more bids can be accepted (line 12), and the Seller can collect the funds (line 14) and terminate the auction.

These triggers are limited to only time or funds. They cannot react to a combination of these two factors at the same time. If the Seller wanted the auction to interrupt after a certain amount of time or when the current bid matches a value, he could not express this condition with Falcão's version of SmartScribble. In this work, we propose triggers with mixed conditions (see Section 4.2).

## 3.2 Blocked wallets

Triggers use not only limited conditions but also block the interactions of those who register them. Consider the protocol in Listing 3.2.

The protocol Bargain is similar to the Auction protocol (Listing 3.1), but instead of multiple Buyers, this protocol only has one. The protocol starts with an action from the Seller. The Seller puts an item for sale and defines its initial value. Then, the Buyer must propose an offer. With an offer on the table, the Seller gets to decide if he accepts it or not. If he does, the Buyer pays for the item, which is sold. If not, the Buyer must propose another offer. This process is eventually interrupted by the trigger stop, registered in line 5. The idea of this protocol is to give the Buyer a time limit to propose an acceptable offer.

However, this protocol blocks the Seller from interacting with the protocol. When the trigger is registered, the Seller's actions get blocked. The Seller is released when the trigger activates and interrupts the interactions inside the **do**{} block. Consequently, the Buyer can propose an offer, but the Seller cannot accept or deny it. We design a solution for the blocked wallets and present it in Section 4.2.

```
protocol Bargain (role Seller, role Buyer){
    field Integer, Value;

    putForSale(Integer) from Seller{
        trigger slot stop;
    };
    do{
        rec Deal{
            proposeOffer(Value) from Buyer ;
            choice at Seller {
                acceptOffer :{
                    payForItem(Value) from Buyer;
                }
                refuseOffer:{
                    Deal;
                }
            }
        }
    }interrupt{
        stop() from Contract;
    }
}
```

Listing 3.2: Bargain protocol

## 3.3 No user validation

In SmartScribble, roles determine who participates in a protocol and who calls a determined endpoint. Consider the protocol example in Listing 3.3. This protocol involves two entities in this crowdfunding campaign: the Owner and the Contributor. The Owner is responsible for starting and managing the crowdfunding protocol. During the process, if he decides to continue, the Contributor can store part of his funds in the contract with the contribute endpoint. If the Owner decides to close the crowdfunding campaign, no more contributions are accepted, and the Owner must collect all the contributions to finish the protocol.

Role validation was not implemented in the first release of SmartScribble. Although roles

```
protocol Crowdfunding (role Contributor, role Owner){
    init () from Owner;
    rec Loop {
        choice at Owner{
            continue : {
                contribute (Value) from Contributor;
                Loop;
            }
            closeCrowdfund : {
                collectFunds () from Owner;
            }
        }
    }
}
```

Listing 3.3: Crowdfunding protocol

specify who is participating and who calls which endpoint, they do not affect the smart contract's code. The compiler does detect and validate roles in the semantic analysis step. However, during the code generation step, they are not applied anywhere. As a result, the contracts created by SmartScribble do not check who is calling a specific endpoint. If one of the contributors of the ongoing crowdfunding is malicious, he can easily steal all the contributions from this protocol. If this malicious user calls the collectFunds endpoint, the contract accepts this request since it is not verifying who the caller is. This is not an add-on to SmartScribble; it is a requirement. In Section 4.3, we present the solution for validating roles.

## 3.4 Two-phase development

As stated at the beginning of this chapter, the creation of the contract is divided into two phases. The first phase is the code generation from a protocol. The second phase is the insertion of the business logic, which must be done manually. Creating a smart contract this way is not the ideal scenario. Indeed, in the generated code, there are specific functions where the programmer must add the missing code. If the programmer notices something is missing in the protocol or some interaction is out of place, he will change the protocol and recompile it. Recompiling the protocol overwrites what was generated and added to the prior version. Consequently, the logic code he wrote is deleted by the compiler. It is not ideal for a programmer to rewrite the same code due to a possible minor error.

Another issue with the business logic implementation regards the protocol's variables.

```
1 protocol GuessingGame (role Owner, role Player){
2     field ByteString;
3     lock(String,Value) from Owner;
4     guess(String) from Player;
5 }
```

Listing 3.4: Protocol of a simple version of the Guessing Game

Declaring variables in the field statement or the parameters of an endpoint only requires the specification of their data type. The protocol in Listing 3.4 presents a simplified version of Falcão's guessing game [16]. The variable in the field statement represents the secret word of the current game. The String and the Value in the lock endpoint defines the secret word and the prize, whereas the String in the guess endpoint represents the Player's word he is attempting to guess. Only the type is defined. The name of a variable is never specified in a protocol; the compiler automatically assigns a name. For instance, the field's ByteString is named "param1" by the compiler. In the lock endpoint, the parameters are named "param1" and "param2", being "param1" the String and "param2" the Value. This method of assigning the names automatically is not a problem when there are a small number of variables. However, it is not ideal if the protocol requires a more considerable number of variables. For instance, if an endpoint had one hundred parameters, the first would be named "param1" and the last "param100", which makes the parameters hard to identify from each other, leading to eventual errors introduced by the programmer.

Lastly, the business logic functions present an issue when returning the result of their computation. They return the next values of the state the smart contract is transiting to. They can either have the same value as before or a new one. Returning the next values requires them to be specified in the returning expression in the same order they are in the **field** statement if there is any. And the last value specified in the returning statement is the current funds of the contract. Working with a small number of variables is not a problem. However, looking again at the one hundred variables scenario, we realize that identifying and handling the variables becomes challenging.

## 3.5 Global state machines, only

SmartScribble relies on a state machine to ensure the correct execution of a protocol; this was the most important feature that enabled SmartScribble to properly handle the smart contracts that could be exploited in Plutus due to out-of-order interactions [17]. However, SmartScribble only considers global state machines. Consider the scenario described below.

When a client arrives at a supermarket, he starts collecting the items he wishes to buy. Once the client has everything, he proceeds to the payment phase. After the payment, the client leaves the supermarket with his bought items. If we translate this process into a state machine, the client arrives at the supermarket in the first state. The client transits to the next state at the moment he decides to begin browsing and gathering items. Once he has everything, he stops browsing and steps into the payment state. After the payment, he transits to the final state, where he leaves the supermarket.

```
1  protocol SuperMarket (role Client){
2      field [Integer];
3
4      start() from Client;
5
6      rec Browsing{
7          choice at Client{
8              keepBrowsing:{
9                  pick(Integer) from Client;
10                 Browsing;
11             }
12
13             stopBrowsing:{
14                 pay(Value) from Client;
15             }
16         }
17     }
18 }
```

Listing 3.5: Supermarket protocol

The protocol in Listing 3.5 is a representation of a shopping process for a client. However, in the original implementation of SmartScribble, this protocol cannot handle multiple clients. Due to how SmartScribble creates its contracts, the several clients of this supermarket cannot be in different states. The state machine created for each protocol is global. This means that the current

state is shared; everyone simultaneously transits to the same state. This state machine cannot track different states for different clients. In Section 4.5, we design a solution where several state machines can be specified according to Listing 3.5.

# Chapter 4

# Extending SmartScribble

In this chapter we present the solutions implemented to extend SmartScribble and how they are implemented. These solutions address the problems mentioned in the examples in Chapter 3, namely:

- the limited expressivity of triggers,
- the (undesired) blocking mechanism for the user who sets up the triggers,
- the lacking of user validation,
- the development process divided into two phases,
- the limitation of the global state machine.

## 4.1 Variable naming

Before we dive into the identified features, we introduce a small (but useful) change to SmartScribble's syntax. In the original implementation of SmartScribble [16, 17], when the programmer wishes to use variables in the protocol's field or the parameters of an endpoint, he can declare the variables by specifying their data types. Since he does not name the variables, the compiler automatically assigns their names.

```
1  protocol VariableProtocol (role Subject) {
2    field ByteString, Integer, Value;
3
4    someEndpoint (String, Value) from Subject;
5  }
```

Listing 4.1: Example of how variables are used in a protocol

In the protocol in Listing 4.1, three variables are being declared in the **field**, in line 2. The compiler will automatically name these variables as `param1`, `param2`, and `param3`, respectively. The variables in the endpoint's parameters, in line 4, will also be automatically named `param1` and

param2. As we mentioned in Section 3.4, the automatic assignement of names is only feasible for small and controlled examples; it's not scalable. In this work we endow SmartScribble with the capability of naming variables in the protocol specification. It addresses the issue we presented and is relevant for some of the extensions we have implemented. Since SmartScribble was based on the Scribble language [59], we adapted its variable's syntax for our language:

variableName : variableType

To name a variable, the programmer assigns the name first and then declares its data type next to the name, separated by a colon. When the programmer compiles the protocol, SmartScribble will assign all variables with the respective names defined in the protocol. In Listing 4.2 we rewrite the guessing game [17] with the new syntax, identifying the variable names (compare with Listing 3.4).

```
1 protocol GuessingGame (role Owner, role Player) {
2   field secret:ByteString;
3   lock(newsecret:String, prize:Value) from Owner;
4   guess(attempt:String) from Player;
5 }
```

Listing 4.2: Guessing game with variable naming.

## 4.2 New triggers and unblocked wallets

SmartScribble's triggers are registered inside the environment of a protocol. The original syntax uses the keywords slot and funds to indicate what the trigger reacts to. The condition for a trigger to activate is programmed in the generated code. Programming a trigger's condition requires functions provided by Plutus [28].

In this work we restructure the definition of triggers and enable the specification of more complex conditions. The Plutus' functions used in the original triggers have a condition programmed in them given their context. They verify their conditions each time the contract has a new set of UTxOs. If the condition is met, they return. Otherwise, they wait for the next set. Triggers work inside the endpoint they were registered in. Consequently, that endpoint must wait for the trigger to return to finish its computation. These triggers block the contract instance that called the endpoint with a trigger, blocking users from interacting.

We propose that triggers are still registered in the endpoint where the protocol declared it, but instead of waiting for the condition inside that endpoint, the condition is verified in the endpoints to be interrupted. The compiler creates a unique function for the trigger whose the purpose is activating the trigger if a specific condition is met. The condition is inside an auxiliary function that returns a Boolean value. The programmer is now allowed to write a complex expression for the condition.

Subsection 4.2.1 presents the changes in the syntax for triggers, and subsection 4.2.2 explains how they are integrated in the contract.

### 4.2.1 Triggers' new syntax

To implement complex triggers, we start by omiting the slot and funds keywords. Triggers are still implemented inside the call body of an interaction from the protocol. First, the programmer must declare the **trigger**. Then, he must assign a name for the trigger and end the trigger's statement with a semicolon. Listing 4.3 showcases the definition of the trigger closeGame in the lock endpoint.

```
lock (secret:String,prize:Value) from Owner{
  trigger closeGame;
};
```

Listing 4.3: Example of trigger declaration

The logic for the trigger's condition is implemented afterwards, in the generated code. Alternatively, the programmer can program the condition directly in the protocol. For that, we extend SmartScribble with Template Haskell [52], a Haskell extension for meta-programming. Listing 4.4 presents the syntax for the specification of a condition in the body of a trigger.

```
lock (secret:String,prize:Value) from Owner{
  trigger closeGame[|
    -- Condition code for the closeGame trigger --
  |];
};
```

Listing 4.4: Syntax for trigger declaration with a specified condition

The corresponding interaction activates and performs its transition when the trigger's condition is met. However, before that, it has a logic function that must run first. A trigger's logic function is different from its condition function. Condition functions define the activation condition, and logic functions define the behavior when activated. These logic functions can change some values stored inside the current state of the smart contract. Programming the condition or the behavior can be programmed in the protocol specification or later in the smart contract.

```
1  protocol TriggerGuessingGame (role Owner, role Player) {
2      field secret:ByteString;
3
4      lock (secretWord:String,prize:Value) from Owner{
5        trigger closeGame[|
6          time >= timeStamp + 10000
7        |];
8      };
9
10     do {
11         rec Loop {
12           guess (attempt:String) from Participant;
13           Loop;
14         }
15     }
16     interrupt {
```

```
17          closeGame () from Contract[|
18            returnOk
19          |];
20      }
21  }
```

Listing 4.5: Guessing game protocol using complex triggers

A do notation is generated in the condition and behavior function, allowing the use of other Haskell statements such as if or let. In Listing 4.5, the trigger closeGame is set to activate after 10 seconds (10000 milliseconds) the moment it was registered. Once the trigger activates, the protocol executes its respective interaction, located in line 17. The code inside the closeGame interaction in line 17 returns the flow without errors and doesn't change any value on the contract's variables represented by returnOk, that is further explained in Section 4.4.2.

### 4.2.2 Implementation of triggers in Plutus

To allow complex conditions and solve the problem of a wallet getting blocked due to an active trigger, we changed the code generated for triggers. The Plutus functions used by the original implementation have their condition programmed within. Moreover, most of them check the condition each time there is a new set of UTxOs. For the new triggers, to avoid blocking a wallet from operating, we check the condition each time an endpoint that is specified inside the do block of a global escape is called. The endpoint that registers the trigger records the current slot and timestamp (in milliseconds). With the trigger registered, when an endpoint inside a do block is called, the condition is checked first. Checking a condition calls a function whose primary purpose is to check the condition. If the condition is met, that function calls the trigger's respective logic function and then applies the transition that belongs to the trigger. The result of that transition moves the protocol flow out from the do block intended to interrupt. Otherwise, if the condition is not met, it performs the endpoint's regular computations.

The condition checked in our triggers is an auxiliary function integrated into the contract that returns a Boolean. The programmer can define the conditions inside the protocol using Haskell expressions. We achieve this with Template Haskell (we explore this topic further in Section 4.4). Our compiler transfers the Haskell code written in the protocol to the condition's function. The Plutus compiler is responsible for checking if the given code is syntactically and semantically correct. That auxiliary function receives five parameters as input: time and slot, which are, respectively, the current time (in milliseconds) and the current slot; timeStamp and slotStamp, which are, respectively, the time (in milliseconds) and the slot the current trigger was registered, moreover, funds, which are the current funds present in the smart contract. Our triggers allow any custom condition to be specified; we also gave our triggers the possibility to react to the variables in the protocol's field.

```
1  protocol Counter (role Subject) {
2      field counter:Integer;
3
4      init() from Subject{
```

```
5           trigger stop:[|
6             time >= timeStamp + 10000
7             || funds 'V.geq' (Ada.lovelaceValueOf 3000000)
8             && counter == 10
9           |];
10        };
11
12        do {
13            rec Loop {
14              count () from Subject;
15              Loop;
16            }
17        }
18        interrupt {
19          stop() from Contract;
20        }
21  }
```

Listing 4.6: Protocol for a counter using complex triggers

The protocol in Listing 4.6 has a variable counter, of the type **Integer**, on the **field**, and has two endpoints, **init**() and count(). The protocol starts with **init**(), which will start the counter, and then count is called multiple times to increment the counter variable until the stop() trigger interrupts the flow of the interactions inside the **do** block. The **init**() endpoint records the moment the trigger was registered, represented by the timeStamp and slotStamp parameters used in the condition. Furthermore, the count() endpoint will first check the trigger's condition (Listing 4.7 is the condition function created for the stop trigger). If the condition is met, the flow of the **do** block is interrupted. Otherwise, it proceeds with the normal flow. The trigger in the protocol above is set to activate when ten seconds have passed (in Plutus, the current time is given in milliseconds, so the time in the condition must be specified in milliseconds) or when the smart contract has 3 million Lovelace (3 ADA) stored in it, and the value of the counter variable is equal to 10.

```
stopCondition :: POSIXTime ->
                 Slot -> POSIXTime -> Slot -> Value -> Integer -> Bool
stopCondition time slot timeStamp slotStamp funds counter = (do time >= timeStamp
    + 10000) || funds 'V.geq' (Ada.lovelaceValueOf 3000000) && counter == 10
```

Listing 4.7: Condition function created for the trigger in Listing 4.6

## 4.3 Role validation

In Section 3.3, we observed that SmartScribble's roles do not affect the created smart contracts. This feature was not implemented when SmartScribble's first version was introduced and is absolutely required to ensure that the (semantics of the) program is correct.

In Section 2.1, we mentioned that asymmetric cryptography is used to ensure the authentication of the transactions because transactions are signed with the private key. In Solidity, for instance, users are identified by their addresses. An address is stored in the smart contract to validate if the correct user is calling a Solidity function. This stored address is compared with the address of the function caller [14].

```
contract Coin {
    address public minter;

    constructor() {
        minter = msg.sender;
    }

    function mint(address receiver, uint amount) public {
        require(msg.sender == minter);
        balances[receiver] += amount;
    }
}
```

Listing 4.8: Caller's address validation in Solidity [14]

The code in Listing 4.8 is part of the Coin smart contract where the address is validated. The constructor is called when a Solidity contract is deployed into the blockchain. This constructor stores the address of the user that deployed the contract, implying that that user must call the mint function. In Plutus, we validate if the proper role calls an endpoint by looking at the caller's public key. In Solidity, a variable for an address is declared at the beginning of the contract as a state variable. In SmartScribble, we store them in the Plutus state machine alongside the variables declared in a **field**.

Subsection 4.3.1 explains how keys are registered and Subsection 4.3.2 explains how the validation works in a smart contract.

### 4.3.1 Registration of public keys

To validate the roles' keys, they need to be registered first. The public keys can be saved inside the states of the state machine. In some scenarios, a role can be taken by multiple users. For example, in the Crowdfunding protocol, we know that only one user represents the Owner, and anyone can be a Contributor. For this scenario, we just need to validate if the endpoints meant to be used by the Owner are being called by the Owner. Hence, we just need to provide the set of keys of the Owner.

In the syntax of our language, we added the keyword **signed**. This keyword precedes the roles in the protocol's signature. When a role is identified as **signed** in the signature, then that role requires its keys to be registered and validated.

```
protocol Crowdfunding (role Contributor, signed role Owner){
    init () from Owner;
    rec Loop {
        choice at Owner{
            continue : {
                contribute (contribution:Value) from Contributor;
                Loop;
            }
            closeCrowdfund : {
                collectFunds () from Owner;
            }
        }
```

```
13      }
14  }
```

Listing 4.9: Crowdfunding protocol declaring the Owner **role** signed

In the protocol in Listing 4.9, all endpoints, except the contribute() endpoint, validate a set of public keys since they are being called by a **signed role**. The smart contract generated from this protocol will store a list of public keys for each **signed role**, in this case, only one list of keys for the Owner role. To register the Owner's keys, the programmer must pass a list of the keys through the command line with the command used to compile the protocol. The compiler will read all the lists of keys passed in the command line and integrate them into the contract, associating each to the correct role.

```
$ cabal run stable-playground crowd.scr "Owner=[owner-wallet-pk]"
```

Listing 4.10: Command to compile the protocol in Listing 4.9

The command in Listing 4.10 demonstrates how the Crowdfunding protocol is compiled. The command is creating a smart contract for the Plutus Playground from a protocol in a file name "crowd.scr". The last part of the command is the set of keys of the Owner's wallets. Commas separate keys in this list. The key "owner-wallet-pk" is just a placeholder for demonstration purposes. The actual keys are more extensive sets of hexadecimal characters. If one of the **signed** roles do not have a key set assigned in the command, the compiler throws an error.

### 4.3.2 Validation of public keys

Plutus provides a function to validate the keys. The function txSignedBy [25] looks at the information of the submitted transaction and verifies if it was signed by the entity representing the provided public key.

Each endpoint submits a transition in the state machine. For each transition, a transaction is submitted to change the current state of the state machine. The transition is only accepted if a check function validates it successfully. The function txSignedBy is used inside the check function to perform the validation of the signer. This will ensure that the endpoint is called by the wallet representing the respective role. If a non-authorized wallet calls the endpoint, the validation of the transaction submitted by the transition will fail, and the state does not change.

Two users can also represent the same role, or the user representing the role can possess two wallets. The txSignedBy only works with one key. Checking if one of the keys from the list of one signed role is signing the transaction can be easily achieved with a function from Haskell named 'any'. Applying txSignedBy to each key, alongside the 'any' function, validates if one of the signed role's keys is calling an endpoint.

Instead of performing a transition, the initial interactions were responsible for starting the state machine instance with the initial state. Starting the state machine instance does not permit the action to be verified. To get around the problem, we introduced a ghost state, where the initial

transitions can start from. Initial interactions start the machine with this ghost state, then perform the initial transition.

```
1  transitionCheck :: CrowdfundingState -> CrowdfundingInput -> ScriptContext -> Bool
2  transitionCheck state input context = case (state, input) of
3      (None ownerId triggerTimeStamps, InitInput _ _ _) -> checkKeys ownerId
4      (ContinueState ownerId triggerTimeStamps, ContributeInput _ _ _) -> True
5      (InitState ownerId triggerTimeStamps, ContinueInput _ _ _) -> checkKeys ownerId
6      (InitState ownerId triggerTimeStamps, CloseCrowdfundInput _ _ _) -> checkKeys ownerId
7      (CloseCrowdfundState ownerId triggerTimeStamps, CollectFundsInput _ _ _) -> checkKeys
           ownerId
8
9      _ -> False
10     where
11         checkKeys keys = any (txSignedBy $ (scriptContextTxInfo context)) $ map
               unPaymentPubKeyHash keys
```

Listing 4.11: Validation function used by the state machine

Plutus uses a checking function that must return `True` before submitting a transaction. Listing 4.11 is the checking function that we to validate the keys of a role. Line 4 returns `True` because a Contributor makes the transition. Lines 3, 5, 6, and 7 verify the transitions made by the Owner.

Appendix A presents the Crowdfunding protocol complete with its business logic and the generated code for role validation.

## 4.4 Business logic generation

As stated in Chapter 3, the compiler only generates part of the smart contract. The missing code is the business logic that must be added manually. However, this is not the ideal approach.

We extended the SmartScribble's syntax so that the language can automatically generate the business logic and provide the user with a fully working contract. We achieved this by allowing Plutus code to be integrated into the protocol using Template Haskell. Template Haskell [52] is a Haskell extension for meta-programming to integrate inside the smart contract the code written in the protocol. This solution allows a programmer to keep the business logic code written in case the protocol needs to be patched.

Template Haskell is written inside special brackets known as quasi-quoters. We use these brackets to integrate the Plutus code in SmartScribble. SmartScribble captures the code between the quasi-quoters as a string. The logic function is taken as a string, and we insert the captured code inside its body. We use a library provided by Haskell to parse the string into a Template Haskell AST [20]. Template Haskell can detect some errors introduced and passes them to SmartScribble's compiler to be displayed in the console. The Template Haskell AST is then converted into Plutus code and inserted in the smart contract. The errors we can detect with this approach are:

- Bad code indentation,
- The last statement is not an expression,
- Mismatched brackets.

Other errors in the smart contract, especially concerning Plutus, are detected with the Plutus compiler. The return expressions used in logic functions do not convey the correct idea of what they give back (Listing 4.12). We provide, alongside these expressions, functions generated by the compiler to aid the endpoint programming (Section 4.4.2).

```
pure $ Left variableA variableB variableC ...

pure $ Right variableA variableB variableC ...
```

Listing 4.12: Logic functions current return expressions

Subsection 4.4.1 demonstrates the syntax to incorporate the business logic in SmartScribble and Subsection 4.4.2 demonstrates how it is integrated.

### 4.4.1 Protocol's syntax for the business logic

We integrated the business logic in SmartScribble, but we do not force it. The programmer may not want to define the logic of each endpoint in the protocol or may wish to add it later. So we made this feature optional. To demonstrate how the business logic is integrated, we use a simple version of the Guessing game as an example (Listing 4.13).

```
1 protocol GuessingGame (signed role Owner, role Player) {
2   field storedSecret:ByteString;
3   lock (secret:String, prize:Value) from Owner;
4   guess (playerGuess:String) from Player;
5 }
```

Listing 4.13: Simple version of the Guessing game

After defining which role is calling the endpoint, the business logic is specified. We based our syntax on Template Haskell, using its brackets to integrate the Plutus code in the endpoint. Listing 4.14 illustrates the template of the business logic. In Section 4.4.2 we will provide several examples of business logic to add in lines 4 and 8.

```
1  protocol GuessingGame (role Owner, role Player) {
2    field storedSecret:ByteString;
3    lock (secret:String, prize:Value) from Owner[|
4          --Lock endpoint's code--
5    |];
6
7    guess (playerGuess:String) from Player[|
8          --Guess endpoint's code--
9    |];
10 }
```

Listing 4.14: Integration of Plutus code in SmartScribble

In the case of a **choice**, the business logic is defined between the name of the choice and the colon. Listing 4.15 demonstrates how Plutus code is integrated inside a **choice**.

```
choice at Subject{
    choiceA [|
        --choiceA code--
    |]:
```

```
    choiceB [|
        --choiceB code--
    |]:

    choiceC [|
        --choiceC code--
    |]:
}
```

Listing 4.15: Demonstration on how to integrate Plutus code in SmartScribble

Since SmartScribble generates a logic function for each endpoint, the compiler inserts the code of every endpoint, if defined, in the corresponding logic function.

### 4.4.2 Integration of the Plutus code into the smart contract

Before we demonstrate how the protocol integrates the business logic into the contract, we will first explain how the logic functions work. Each function receives as input:

- The endpoint's parameters, if they are defined.
- The state's current **field** values, if **field** was defined.
- The lists of public keys from each **signed role**.
- The list of individuals and their current states, if the protocol uses individual state machines (as we will see in Section 4.5).
- The list of registered triggers.
- The funds currently stored in the contract.

The function returns either an error or the new values for the updated state. If it does not return an error, it shall return:

- The state's current or new **field** values, if **field** is defined.
- The current value or a new value of the funds stored in the contract.
- The transaction's constraint.

The logic functions for the individual protocol's endpoints have as input and output the elements above, in addition to the **field** values defined in the individual protocol. Initially, logic functions only returned the values for the **field** and the contract's funds. However, every endpoint submits a transaction, and a constraint can be added to define the behavior of the transaction. For this reason, our logic functions now return a constraint in addition to what was already being returned. To form a constraint with SmartScribble, the programmer must add to the endpoint's return expression:

```
Constraints.someConstraint
```

Where someConstraint is one of the functions Plutus provides to build constraints [26]. Multiple constraints can be concatenated with the <> operator. In case the programmer does not wish to apply constraints to the transaction, he must assign mempty to the value of the constraint.

When template Haskell receives the code from the protocol, the compiler inserts it in the logic function, and then Template Haskell compiles the whole function and formats it. If Template Haskell launches an error, our compiler will print that error in the console. The business logic code must be correctly indented, and the last statement must be an expression.

The return type of these functions is an **Either** [19], a Haskell data type representing one of two values, which in this case are the output values mentioned above or an Error. For the function to return, in the case of an error, the programmer has to write the following:

```
pure $ Right $ Error "This is an Error message."
```

If there is no error, for the function to return the result of its computation, the programmer has to write the following:

```
pure $ Left (variableA, variableB, variableC,...) --Tuple with all values inside--
```

Using the two lines of code shown above may look confusing from the protocol's point of view, and the more fields the protocol has declared, the bigger the line of code for the function's return will be. Our compiler creates auxiliary functions inside the function skeletons to aid the programmer when writing the return expression. For instance, from the protocol in Listing 4.14 the compiler generates for the skeleton of lock's endpoint the following functions:

```
output :: LogicOutput
output = LogicOutput storedSecret stateVal mempty

printMsg :: String -> Contract () GuessingGameSchema T.Text ()
printMsg msg = logInfo @String msg

printError :: String -> Contract () GuessingGameSchema T.Text ()
printError err = logError @String err

returnError :: String ->
               Contract () GuessingGameSchema T.Text (Either (BuiltinByteString,
                    Value, TxConstraints GuessingGameInput GuessingGameState)
                    GuessingGameError)
returnError err = pure $ Right $ Error $ T.pack err

returnOk :: Contract () GuessingGameSchema T.Text (Either (BuiltinByteString,
      Value, TxConstraints GuessingGameInput GuessingGameState)
      GuessingGameError)
returnOk = pure $ Left (storedSecret, stateVal, mempty)

returnOutputOk :: LogicOutput ->
                  Contract () GuessingGameSchema T.Text (Either (
                       BuiltinByteString, Value, TxConstraints GuessingGameInput
                       GuessingGameState) GuessingGameError)
```

```
returnOutputOk out = pure $ Left $ getOutput out

returnOkWith :: BuiltinByteString ->
                Value ->
                TxConstraints GuessingGameInput GuessingGameState ->
                Contract () GuessingGameSchema T.Text (Either (BuiltinByteString,
                    Value, TxConstraints GuessingGameInput GuessingGameState
                    ) GuessingGameError)
returnOkWith storedSecretRet stateValRet constraintRet = pure $ Left (
    storedSecretRet, stateValRet, constraintRet)
```

These functions can be applied in the code written in the protocol. This is what each function returns and how to apply it:

**returnOk.** This function returns all the fields with the initial values the logic function received and the exact value of funds the contract currently has. It also returns mempty for the transaction's constraint. This function is helpful if the target endpoint is not meant to change any value of the smart contract and if there is no constraint to be applied.

```
1 protocol GuessingGame (role Owner, role Player) {
2   field storedSecret:ByteString;
3
4   lock (secret:String, prize:Value) from Owner[|
5     returnOk --All fields maintain their values, in this example the secret and the
          prize are not being stored in the contract--
6   |];
7 }
```

Listing 4.16: Example use of returnOk.

**returnOkWith.** This function replaces the pure $ Left expression. Instead of writting pure $ Left (A,B,C), the programmer can use the expression returnOkWith A B C in order to return.

```
1 protocol GuessingGame (role Owner, role Player) {
2   field storedSecret:ByteString;
3
4   lock (secret:String, prize:Value) from Owner[|
5     returnOkWith secret prize mempty --The mempty is the constraint value--
6     --This expression is storing the secret and the prize--
7   |];
8 }
```

Listing 4.17: Example use of returnOkWith.

**returnError.** This function replaces the pure $ Right expression. Instead of writing pure $ Right $ Error "This is an Error message", the programmer can use the expression returnError "This is an Error message" in order to return.

```
protocol GuessingGame (role Owner, role Player) {
  field storedSecret:ByteString;

  lock (secret:String, prize:Value) from Owner[|
    returnError "This is an error message."
  |];
}
```

Listing 4.18: Example use of returnError.

**printMsg.** This function logs a standard message in a specific step of an endpoint's logic.

```
protocol GuessingGame (role Owner, role Player) {
  field storedSecret:ByteString;

  lock (secret:String, prize:Value) from Owner[|
    printMsg "This is a message before this function returns"
    returnOk
  |];
}
```

Listing 4.19: Example use of printMsg.

**printError.** This function logs an error message in a specific step of an endpoint's logic. It does not make the function return an error.

```
protocol GuessingGame (role Owner, role Player) {
  field storedSecret:ByteString;

  lock (secret:String, prize:Value) from Owner[|
    printError "This is an error message, but I am not returning an error."
    returnOk
  |];
}
```

Listing 4.20: Example use of printError.

**output.** This is not a return expression, but a crucial element for the returnOutputOk function. It is an object of the LogicOutput data type. The compiler, based on the variables declared on the protocol's **field**, generates the LogicOutput data type. The fields of this data type are the fields declared in the protocol and the fields declared in the individual protocol if there is one with the **field** statement. In addition, it is created with a field for the contracts' funds and a field for a constraint. When output is called, it always returns the data type with field values passed in the logic function's parameters.

```
-- | Data structure used in the logic functions to track the change of the fields that
     may be altered from the execution of an endpoint.
data LogicOutput = LogicOutput{
```

```
    storedSecret :: BuiltinByteString,
    stateVal :: Value,
    constraint :: TxConstraints GuessingGameInput GuessingGameState
}

-- | Changes the value of the storedSecret field.
setStoredSecret :: BuiltinByteString -> LogicOutput -> LogicOutput
setStoredSecret newStoredSecret output = output{ storedSecret = newStoredSecret }

-- | Changes the value of the stateVal field.
setStateVal :: Value -> LogicOutput -> LogicOutput
setStateVal newStateVal output = output{ stateVal = newStateVal }

-- | Changes the value of the constraint field.
setConstraint :: TxConstraints GuessingGameInput GuessingGameState ->
    LogicOutput -> LogicOutput
setConstraint newConstraint output = output{ constraint = newConstraint }

getOutput :: LogicOutput -> (BuiltinByteString,Value,TxConstraints
    GuessingGameInput GuessingGameState)
getOutput out = (storedSecret out,stateVal out,constraint out)

getSoloOutput :: LogicOutput -> (BuiltinByteString,Value,TxConstraints
    GuessingGameInput GuessingGameState)
getSoloOutput out = (storedSecret out,stateVal out,constraint out)
```

Listing 4.21: LogicOutput data type created for the protocol in listing 4.13

The last two functions in Listing 4.21, getOutput and getSoloOutput, are used by the returnOutputOk function to convert the data type into an output the logic functions can return. getOutput is used by the global endpoints, and getSoloOutput is used by the individual endpoints.

**returnOutputOk.** This function must be used with the output function. It is a cleaner method of returning if only a few variables are changed in the contract. If a protocol had one thousand variables declared, an endpoint that will change just one of them needs to return all those variables, including the updated one. If the **field** had declared the variables A, B, C,..., Z, to return, the programmer has to write, for example, the following line:

```
    returnOkWith A B C ... Z
```

This situation is chaotic, inconsistent, and prone to changing the wrong variable due to human error. returnOutputOk solves this situation. The function output mentioned before returns an object that keeps track of the changes an endpoint's business logic is performing. output is initialized with the values given in the function's parameters, and the setter functions are used to update the output's fields. returnOutputOk then converts the data type into a tuple that can be returned by the function and terminates the computation of the logic function.

```
1  protocol GuessingGame (role Owner, role Player) {
2    field storedSecret:ByteString;
3
4    lock (secret:String, prize:Value) from Owner[|
```

```
5       returnOutputOk $ setStoredSecret (sha2_256 $ toBuiltin $ C.pack secret)
6                        $ setStateVal prize output
7   |];
8 }
```

Listing 4.22: Example use of returnOutputOk.

The protocol in Listing 4.23 showcases the guessing game presented in Listing 4.14 completed with the business logic.

```
1  protocol GuessingGame (role Owner, role Player) {
2    field storedSecret:ByteString;
3    lock (secret:String, prize:Value) from Owner[|
4          returnOutputOk $ setStoredSecret (sha2_256 $ toBuiltin $ C.pack secret)
5                           $ setStateVal prize output
6    |];
7
8    guess (playerGuess:String) from Player[|
9          let attempt = toBuiltin $ C.pack playerGuess
10         if attempt == storedSecret
11         then returnOutputOk $ setStateVal mempty output
12         else returnError "Wrong guess, try again!"
13   |];
14 }
```

Listing 4.23: Guessing game with the business logic incorporated

## 4.5 Individual state machines

The state machines currently created by SmartScribble only provide global states. This means that every user is in the same state that the contract is, which might not be the case (remember the Supermarket example in Section 3.5). In this Section we propose the integration of individual state machines to solve this problem. An individual state machine allows one user to be in his state regardless of the other users. This solution is achieved by creating individual machines inside the global state machine. A list of individual states is stored inside the global state machine.

Figure 4.1 represents the ideal state machine for the Supermarket example. Individual transitions are only allowed in the global state to which the *open* transition transits to. All the current individual states are recorded inside the global state machine. When a client transits to the next state, the other clients and the contract do not change their state (example in Figure 4.2).

Subsection 4.5.1 introduces the syntax for these new machines and Subsection 4.5.2 explains how they are integrated into a smart contract.

### 4.5.1 Syntax for individual state machines

To build the individual state machines, we need to specify their transitions. In the same way the protocol we write creates a global state machine for the contract, where everyone shares the same state, we enable the specification of a second protocol. This second protocol will create the state machine that each client governs and is written under the primary protocol.

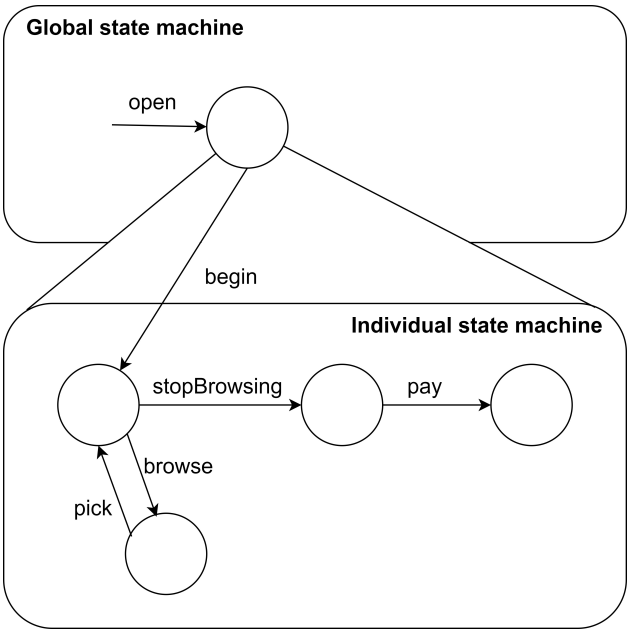



Figure 4.1: Individual state machines for the Supermarket scenario.

| Entity | State |
|---|---|
| Contract | open |
| Wallet 1 | Browsing |
| Wallet 2 | Browsing |
| Wallet 3 | Browsing |

Wallet 2 picks and pays his item

| Entity | State |
|---|---|
| Contract | open |
| Wallet 1 | Browsing |
| Wallet 2 | Paid |
| Wallet 3 | Browsing |

Figure 4.2: Individual client transiting to a new state.

```
1  protocol Supermarket (role Owner, role Client){
2      openMarket() from Owner;
3      share begin() from cli : Client do BuyingProcess(cli);
4  }
5
6  individual protocol BuyingProcess (marketClient : Client){
7      field cart:[Integer];
8      rec Browsing{
9          choice at marketClient{
10             keepBrowsing:{
11                 pick(item:Integer) from marketClient;
12                 Browsing;
13             }
14             stopBrowsing:{
15                 pay() from marketClient;
16             }
17         }
18     }
19 }
```

Listing 4.24: Supermarket protocol

Listing 4.24 is a representation protocol of the supermarket scenario. We have two protocols. The first one is the primary protocol responsible for opening the supermarket. This protocol creates the global state machine that already exists. The second protocol represents a client's buying process, distinguishing it from the primary protocol with the **individual** keyword. This protocol is responsible for creating the instances for the individual state machine.

Line 2 is a global interaction from the Owner opening the supermarket. The begin() interaction in line 3 is an entrance point for a client to his machine. This interaction counts as an individual endpoint and not a global one. The **share** keyword means that many clients share the endpoint in the global protocol. Interactions with the **share** keyword always count as an individual protocol and must always define the protocol the client is entering.

Once an entity named cli, whose role is Client, called the begin() endpoint, he enters into his BuyingProcess protocol. The BuyingProcess protocol is performed by a marketClient entity whose role is Client. Unlike the Supermarket protocol, clients do not share their state.

These shared interactions cannot be the first statements because individual state machines work within the state the previous global interaction transited to. The individual protocol must be under the global protocol because it requires the global state machine to operate.

### 4.5.2 Implementation of individual state machines in Plutus

The individual state machines work inside the global state machine. From the individual protocol, our compiler generates data types for the individual state machine's states and inputs. Based on how the global state machine is created with states that store the information specified in the **field** keyword, the individual states can also store information, enabling the use of the **field** keyword inside the individual protocol. Our compiler also creates a transition function for these state machines, containing each pair of states and inputs for every transition. This function is later used to return the next state or to check if the transition is valid.

```
1  data SoloState =
2      SoloNone [(Integer,(POSIXTime,Slot))]
3      | BeginState [(Integer,(POSIXTime,Slot))]
4      | KeepBrowsingState [(Integer,(POSIXTime,Slot))]
5      | StopBrowsingState [(Integer,(POSIXTime,Slot))]
6      | PayState [(Integer,(POSIXTime,Slot))]
7      deriving (Show)
8
9  PlutusTx.unstableMakeIsData ''SoloState
10 PlutusTx.makeLift ''SoloState
11
12 data SoloInput =
13     BeginInput [(Integer,(POSIXTime,Slot))]
14     | KeepBrowsingInput [(Integer,(POSIXTime,Slot))]
15     | PickInput [(Integer,(POSIXTime,Slot))]
16     | StopBrowsingInput [(Integer,(POSIXTime,Slot))]
17     | PayInput [(Integer,(POSIXTime,Slot))]
18     deriving (Show)
19
20 PlutusTx.unstableMakeIsData ''SoloInput
21 PlutusTx.makeLift ''SoloInput
22
```

```
23 {–# INLINABLE soloTransition #–}
24 soloTransition :: SoloState –> SoloInput –> Maybe SoloState
25 soloTransition state input = case (state,input) of
26     (SoloNone _, BeginInput soloTriggerTimeStamps) –> Just $ BeginState
           soloTriggerTimeStamps
27     (BeginState _, StopBrowsingInput soloTriggerTimeStamps) –> Just $
           StopBrowsingState soloTriggerTimeStamps
28     (BeginState _, KeepBrowsingInput soloTriggerTimeStamps) –> Just $
           KeepBrowsingState soloTriggerTimeStamps
29     (KeepBrowsingState _, PickInput soloTriggerTimeStamps) –> Just $ BeginState
           soloTriggerTimeStamps
30     (StopBrowsingState _, PayInput soloTriggerTimeStamps) –> Just $ PayState
           soloTriggerTimeStamps
31     _ –> Nothing
```

Listing 4.25: Individual state machine elements created for the BuyingProcess protocol in Listing 4.24

The elements created from the individual protocol will shape the individual state machines. The global state machine stores a list to track each client's information. This list stores tuples. Each tuple contains a public key, identifying the client, and a state. The global state machine must be initialized before the clients start operating the individual machine. To perform actions in the individual machines, the compiler creates a unique global input that transits from the current global state to that state, allowing the list of individuals to be updated. The list of tuples is initially empty. However, each time a client performs an action that initiates his machine, a new entry is added to the list, and other actions that are not initial will update the current state. To validate new transitions and change the state, we use the functions created for the global state machine using the created unique input and the individual transition function. Individual machines are not replicated for all users. The global state machine governs all the clients by storing their states, with a single representation of the individual state machine.

For the clients to interact with the individual machine, the compiler creates endpoints for each interaction. Individual endpoints will verify if the list of individuals exists and if the client calling the endpoint is registered in the list, returning its current state if found in the list. If the logic function does not return an error, changing the endpoint's caller state, updating or keeping the value of his fields, updates the list of individuals. The endpoint will perform a global transition, using the input created for the individual state machines carrying the new or current information of the global fields, alongside the current individual state and input, representing the individual transition and the updated list.

# Chapter 5

# SmartScribble 2.0 syntax

As we have seen throughout this work we made several changes to the language's syntax. In this chapter we present the syntax of SmartScribble extended with the features we propose in this document.

## Data types

SmartScribble[1] supports data types definitions limited to the functionality of Plutus on-chain code. The supported on-chain types are:

**ByteString** - representation of a String in bytes to be stored in the ledger.

**Integer** - basic type that represents an integer number

**Value** - represents the value of a cryptocurrency.

**PubKeyHash** - identifies a public key.

**Ada** - currency of Cardano.

**Slot** - unit to measure time in slots.

**TokenName** - represents the name of a Token.

**POSIXTime** - unit to measure time in milliseconds.

**Bool** - represents a boolean value: `True` or `False`.

[1] `https://git.lasige.di.fc.ul.pt/amordido/smartscribble`

## Protocol definition

```
protocol <ProtocolName> (signed? role <Role>(, signed? role <Role>)*){...}
```

This constructor is the base of the protocol. It defines the name of the protocol and the roles that the users have.

```
1 protocol ProtocolDefinitionExample (role A, role B) {
2     funcForA () from A;
3 }
```

Listing 5.1: Example of the protocol constructor

In Listing 5.1 we see a protocol being defined with the name "ProtocolDefinitionExample" and declaring the participating roles, named A and B. The name of the protocol must start with a capital letter. At least one Role must be declared, and duplicate roles are not allowed.

```
1 protocol ProtocolDefinitionExample (signed role A, role B) {
2     funcForA () from A;
3 }
```

Listing 5.2: Example of the protocol constructor with a signed role

Roles can be labeled with the keyword signed (Listing 5.2). This implies that any interaction called by a signed role must validate the keys of the caller. The keys for a signed role must be passed through the command line, as presented in Section 4.3.1.

## Interaction

```
<Endpoint> ((id:<Type> (,id:<Type>)*)?) from <Role> ([|<PlutusCode>|])? ({<Trigger
    >*})?
```

This constructor defines the interaction endpoints that the protocol follows between a user with role < Role> and the contract. The developer can define as many parameters to the endpoint as he wants or no parameter and can declare a trigger inside.

```
1 protocol InteractionExample (role A, role B) {
2     funcA () from A;
3     funcB (intVal:Integer) from B;
4     funcC (adaVal:Ada, stringVal:String) from A;
5 }
```

Listing 5.3: Example of the interaction constructor

In Listing 5.3, we present an example protocol containing three interactions. First, a user with Role A must call the endpoint funcA. Then a user with role B must call the endpoint funcB and pass an Integer value. The protocol ends with a user with Role A calling funcC and passing an Ada value and a String. All the roles used in the interactions must be defined in the protocol definition. One interaction cannot be defined as a final interaction if it was defined before in the protocol. For example: this(),that(),another(),that(); in this exact order with that() as second and last interaction. In this scenario, the first time the duplicated interaction is called, Plutus thinks the state machine has reached its final state.

Interactions can receive Plutus code to define their behavior when called (Listing 5.4). The code for their business logic is defined after defining the caller role and before the triggers' declaration.

```
1 protocol InteractionExample (role A, role B) {
2     funcForA () from A [|
3         --Plutus Code--
4     |];
5 }
```

Listing 5.4: Example of the interaction constructor with the business logic defined

## Choice

```
choice at <Role> {(<Label> ([|<PlutusCode>|])?: {...})+}
```

The choice constructor allows the protocol to branch into two different paths depending on the user's decision. When a user with role < Role> picks one of the <Label>, the protocol must follow interactions defined inside the selected <Label>.

```
1 protocol ChoiceExample (role A, role B) {
2     firstFunc () from A;
3     choice at B{
4         doOptA:
5             {makeA () from A;}
6         doOptB:
7             {makeB () from A;}
8     }
9 }
```

Listing 5.5: Example of the choice constructor

In Listing 5.5, the protocol starts with the firstFunc interaction by someone with role A. Then the flow follows the path decided by a user with role B. The choices present inside the choice constructor, doOptA and doOptB, creates two endpoints in the smart contract. If the user calls the doOptA endpoint the next interaction must be the makeA endpoint from a user with role A. But if the user calls the doOptB endpoint, the next interaction must be the makeB endpoint from a user with role A.

Choices are endpoints the user must select to progress the flow of the smart contract. Consequently, the programmer must define their business logic. The business logic is defined between the choice's label and the colon (Listing 5.6).

```
1 protocol ChoiceExample (role A, role B) {
2     firstFunc () from A;
3     choice at B{
4         doOptA[|returnOk|]:
5             {makeA () from B;}
6         doOptB[|if stateVal == mempty
7                 then returnOk
```

```
8                   else returnError "There are funds in the contract!"|]:
9               {makeB () from B;}
10      }
11  }
```

Listing 5.6: Example of the **choice** constructor with business logic definition

## Recursion

```
rec <Label> {...<Label>;}
```

The **rec** constructor defines a recursive flow of interactions. This allows an endpoint or a set of endpoints to be called multiple times in case of being necessary to repeat a specific procedure in the protocol.

```
1  protocol RecursionExample (role A, role B) {
2      firstFunc () from A;
3      rec Loop{
4          act () from B;
5          Loop;
6      }
7  }
```

Listing 5.7: Example of the **rec** constructor

The protocol in Listing 5.7 begins with the call of the firstFunc endpoint from a user with role A. Next, the flow enters inside a loop labeled "Loop". When a user with role B calls the act endpoint, the protocol reaches the continue statement Loop in line 5. This statement sends the flow back to the beginning of the body of the **rec** constructor. The flow inside the **rec** constructor loops every time all the interactions in the body have been called. The continue statement must be the tail of the **rec** body, and at least one interaction is defined inside. The reference of the continue statement must be the label of the **rec** statement. If the continue statement is not used, the **rec** body follows a linear flow.

## Global escape

```
do {...} interrupt {...}
```

Global escape defines a set of interactions (specified inside the **do** block) that can be interrupted by a trigger invoked inside the **interrupt** block. The statement inside the **interrupt** block must be a trigger invocation by the Contract.

```
1  protocol GlobalEscapeExample (role A, role B) {
2      begin () from A {
3          trigger terminate;
4      };
5      do{
6          rec Loop{
7              act () from B;
8              Loop;
9          }
```

```
10      }interrupt{
11          terminate () from Contract;
12      }
13  }
```

Listing 5.8: Example of the global escape constructor

The protocol in listing 5.8 starts with the begin endpoint invoked by a user with role A. When invoked, the begin endpoint sets up a trigger named "terminate". The protocol proceeds to the global escape constructor, executing the recursive flow inside the **do** block. Upon activation of the terminate, the flow inside the **do** block stops and continues inside the **interrupt** block. Since terminate is the only statement defined, the protocol reaches its end. The trigger inside the **interrupt** block must be declared as an interaction and must be a call from the Contract. Triggers must be declared first inside an interaction before being used, and they have to be the first statement of the **interrupt** block. Unlike user interactions, interactions from the contract cannot declare triggers.

The interaction from the Contract in line 11 of the protocol can also include a specification of its business logic.

## Trigger

```
trigger <TriggerName> (: [|<PlutusCode>|])?;
```

This constructor defines a trigger. Triggers in Plutus are functions that only return after a specific condition is met. In the protocol in Listing 5.8 we see a trigger being declared inside the begin endpoint. The trigger's activation is based on the contract's funds. It interrupts the flow of the **do** block when activated. Triggers can be declared, but they are useless if they are not used. Looking back at the protocol in listing 5.8, if the terminate interaction was not present inside the global escape constructor, then the activation of the trigger would not stop the recursive loop.

The trigger condition is programmed either in the boilerplate or in the protocol. If programmed in the protocol, the code for the condition is added after the trigger's name and must be defined between a colon and a semicolon (Listing 5.9).

```
trigger alarm:[|
    time >= timeStamp + 10000
|];
```

Listing 5.9: Trigger definition with condition programmed in the protocol

The trigger in Listing 5.9 is programmed to activate 10 seconds after it was registered. Both time and timeStamp are values in miliseconds. The variable time is the current time and timeStamp is the moment the alarm trigger was registered.

## Field

```
field id:<Type>(,id:<Type>)*;
```

This constructor identifies the data types stored inside the states of the contract's state machine. These states will be saved in the ledger, so the data types must be compatible with the Plutus on-chain code. The compatible types of the **field** statement are mentioned in the beginning of this chapter.

```
1 protocol FieldExample (role A, role B) {
2     field byteVal:ByteString, pubkeyVal:PubKeyHash;
3
4     funcA () from A;
5     funcB (integerVal:Integer) from B;
6 }
```

Listing 5.10: Example of the **field** constructor

In the protocol in Listing 5.10 the states and the inputs of the state machine will store a ByteString, named "byteVal", and the hash of a public key, named "pubkeyVal". By default, states and inputs always store a parameter of the type **Value**. Other variables stored by default are a list of registered triggers, a list of users' public keys alongside an individual state, and sets of keys if there is any **signed role**. When this field is declared, the user must declare one or more types to be stored in the states. However, the **field** statement cannot be called twice.

## Individual protocol definition

```
individual protocol <ProtocolName> (id : <Role>) {...}
```

This constructor defines the protocol for the individual users to operate independently of each other. The state machine created by this protocol is shared. Any client running this protocol has his current state of the individual state machine recorded.

```
1 protocol ProtocolDefinitionExample (role A, role B) {
2     func () from A;
3 }
4
5 individual protocol IndividualProtExample (userOfRoleA : A){
6     soloFunc() from userOfRoleA;
7 }
```

Listing 5.11: Example of the **individual protocol** constructor

As shown in Listing 5.11, individual protocols are declared under the primary protocol. The interaction inside the individual protocol, soloFunc(), affects only the instance of the caller's state machine. They require the primary protocol to work. Hence individual protocols cannot be declared without it. One instance of the individual state machine is executed by only one entity. Hence, we do not need to declare many roles. Instead, we declare the role as a variable. The protocol example shows that the individual protocol has a role variable named "userOfRoleA" whose role is A. Endpoints must be called by "userOfRoleA" and not A. The compiler throws an error if the role type declared in the individual protocol is not declared in the primary protocol.

## Shared interaction

```
share id() from id : <Role> do <ProtocolName>(id);
```

This constructor defines an interaction inside the primary protocol. It is used as an entry point to an individual protocol. This interaction always counts as an individual interaction and as an initial transition and must always define the protocol it transits to.

```
1  protocol ProtocolDefinitionExample (role A, role B) {
2      firstStep () from B;
3      share joinTheDarkSide() from solosubject : A do IndividualProtExample(
           solosubject);
4      nextStep() from B;
5      finalstep() from B;
6  }
7
8  individual protocol IndividualProtExample (userOfRoleA : A){
9      soloStep() from userOfRoleA;
10 }
```

Listing 5.12: Example of the Shared interaction constructor

Listing 5.12 demonstrates how the shared interaction is used. The **share** keyword indicates that multiple users can call the interaction The shared interaction cannot be the first statement of a global protocol. Individual state machines require a global state machine. Hence the first interaction must be a global interaction to initialize the global machine. The roles used to define who calls this endpoint are defined like a variable just like how it is in the individual protocol signature; users in the variable "solosubject", whose role is A, can call the joinTheDarkSide() endpoint to initiate their instance of the individual state machine. When a user calls joinTheDarkSide(), the endpoint executes its business logic and creates an instance of the individual state machine from the protocol referred after the **do** keyword. After calling that endpoint, he must follow the individual protocol on his instance. joinTheDarkSide() always counts as an individual interaction. In this protocol, individual interactions are only accepted in the global state to which the firstStep() endpoint transits. When the nextStep() endpoint is called, no more individual interactions are accepted unless a shared interaction is defined after this endpoint's statement. If an individual protocol is defined but has no shared interactions, the compiler throws a warning.

# Chapter 6

# Evaluation

In this Section, we analyze the performance of the proposed extensions of SmartScribble. We start by performing a functional analysis of the examples given in Section 3. Then we study the performance of these examples compiled by the new version. Lastly, we compare them with contracts compiled by the former version of SmartScribble.

## 6.1 Functional evaluation

In our functional analysis, we run some protocols to observe how our implemented extensions behave. Namely, we focus on the extensions proposed in the previous chapters: smart contracts with role validation, complex triggers, and individual state machines. We use the Plutus playground to test our extensions. We address the use cases presented in chapter 3 to understand if our implementation is functional. We do not include the business logic integration in this section because the outcome between defining the business logic code in SmartScribble or incorporating it later in the generated smart contract is the same.

### 6.1.1 Testing role validation

Consider the Auction protocol in Listing 6.1 (Appendix B presents the protocol complete with the business logic). The protocol has one signed role, which is the Seller.

```
1  protocol Auction (signed role Seller, role Buyer) {
2      field currBid:Value,lastBidder:PubKeyHash,asset:Integer;
3      initialise (initBid:Value, assetToSell:Integer) from Seller{
4          trigger closeAuctionTrigger;
5      };
6      do{
7          rec Loop {
8              bid (newBid:Value) from Buyer;
9              Loop;
10         }
11     } interrupt {
12       closeAuctionTrigger () from Contract;
13     }
14     collectFundsAndGiveRewards () from Seller;
15 }
```

Listing 6.1: Auction protocol

We simulate the role validation with this protocol in the Plutus Playground. The first endpoint, initialise(), is called by the protocol's signed role. Hence the contract is going to validate if the Seller is calling it. We run a simple simulation to test this feature by initializing the auction with Wallet 1 (Figure 6.1) and Wallet 2 (Figure 6.2).

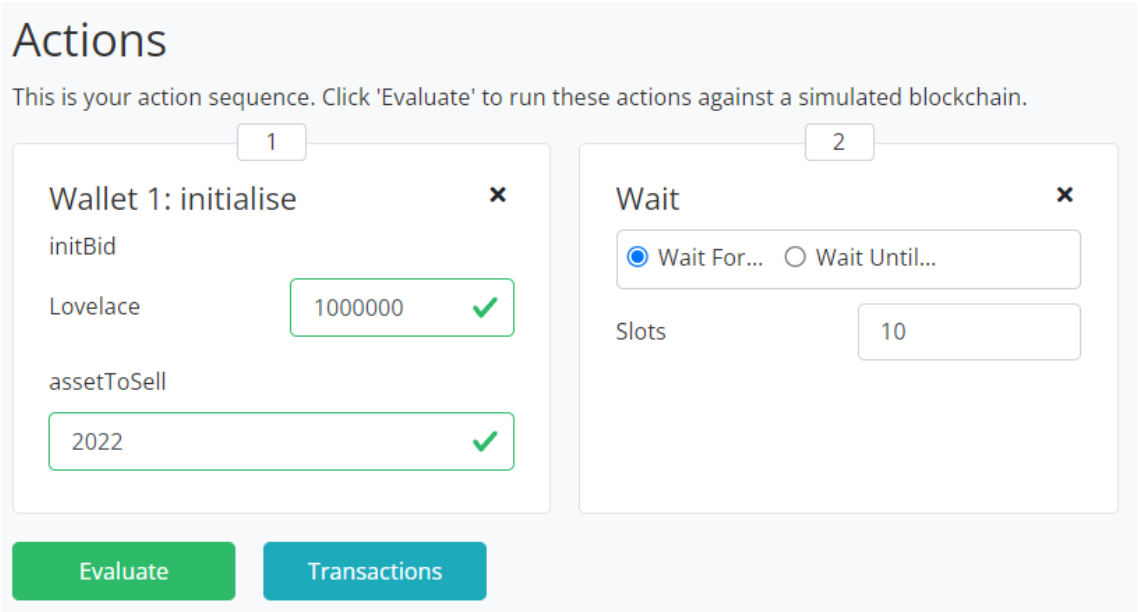



Figure 6.1: Auction set of actions from Wallet 1 in the Plutus Playground

We bind the key from Wallet 1 to the Seller role in the command line presented in Listing 6.2. Being auction.scr the file with the protocol code and the set of hexadecimal characters the public key from Wallet 1 in the Plutus playground. Wallet 2 behaves as the Buyer in this protocol.

```
$ cabal run stable-playground auction.scr "Seller=[
    a2c20c77887ace1cd986193e4e75babd8993cfd56995cd5cfce609c2]"
```

Listing 6.2: Command to compile the protocol in Listing 6.1

Each endpoint submits a transaction responsible for changing the contract's state. Calling an endpoint implies a transaction to be submitted. The caller signs this transaction. When Wallet 1 performs the initialise interaction, it will submit a transaction signed by itself. Before its transaction is accepted, the transaction must go through the validation function we use to validate the caller. Remember that the Seller role is bound to Wallet 1. In this case the key set only contains one public key (Listing 6.2). The validation function grabs the Seller's key set and the transaction's information. The transaction was signed by Wallet 1. The function we use, txSignedBy, validates the public key in the Seller's set and allows the transaction, consequently accepting the initialise interaction from Wallet 1 (see Figure 6.3).

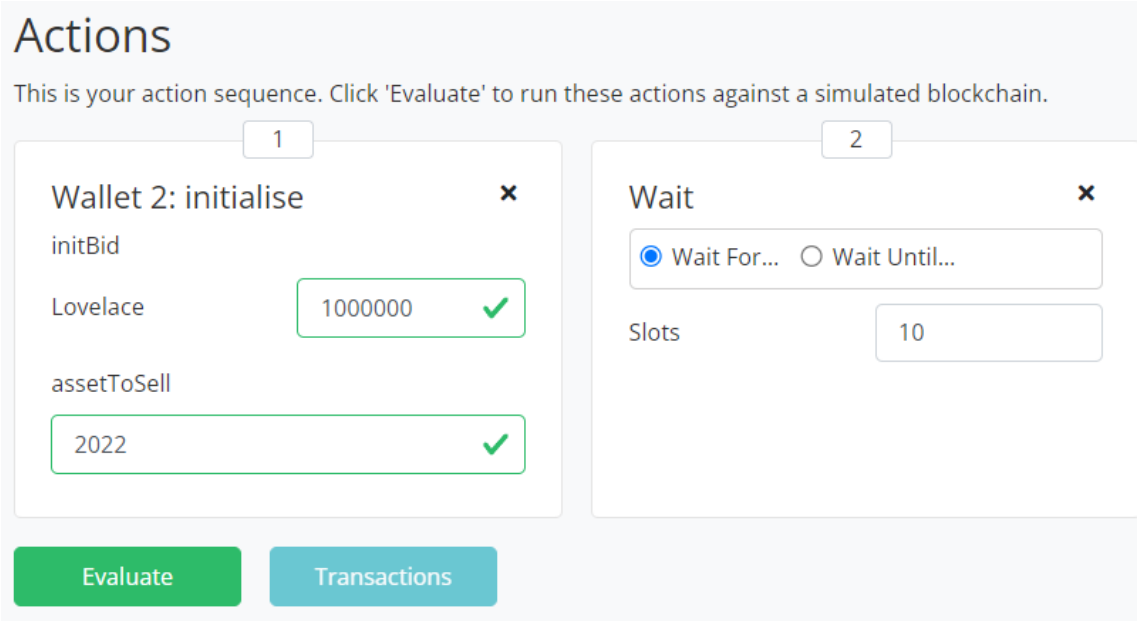



Figure 6.2: Auction set of actions from Wallet 2 in the Plutus Playground

```
==== Add slot 2 ====
Validating transaction: 78ad739c61e0b06ab7a0fe452b8154744d053da1b227d1814a14afd242ee3774
==== Add slot 3 ====
Contract instance for W[1]: (ContractLog (RawJson "Successful transaction to state: InitialiseState
   (Value (Map [(,Map [(\"\",1000000)])])) 0000000000000000000000000000000000000000000000000000000000000000
   2022 [a2c20c77887ace1cd986193e4e75babd8993cfd56995cd5cfce609c2] [(0,(POSIXTime {getPOSIXTime =
   1596059093999},Slot {getSlot = 2}))]"))
```

Figure 6.3: Wallet 1 calls initialise and the validation succeeds

When Wallet 2 calls initialise, the transaction is signed by Wallet 2's key. This transaction must also pass the validation function. The txSignedBy function grabs the Seller's key set and the transaction's information again. Since the key in Seller's set doesn't belong to Wallet 2, the validation fails, and the initialise operation is rejected. Figure 6.4 shows that an error is thrown and displays two error codes. The error codes shown in Figure 6.4 indicate the following:

**S1** - Invalid transition [34]

**PT5** - The validation of the transaction submitted by the transition returned **False** [31]

With the initialise operation from Wallet 2, the state machine does not transit to the next state.

```
==== Add slot 2 ====
Validation failed: 5aae2af25211e712d4a58097ae1adc55a11ca91f1a49962e6487e4349c95858c
   (ScriptFailure (EvaluationError ["S1","PT5"] "CekEvaluationFailure"))
```

Figure 6.4: Wallet 2 calls initialise and the validation fails.

### 6.1.2 Testing complex triggers

To test complex triggers, consider the Counter protocol presented in Listing 6.3. Each time the count() is called, the counter variable is incremented by one. Whenever the counter increments to 3, its value is changed to 0, and the resets variable is incremented by one. The protocol defines a trigger named stop that is set to activate when counter is greater than or equal to two and the resets is greater than or equal to 1.

```
1  protocol Counter (role Subject) {
2      field counter:Integer,resets:Integer;
3      startCounter() from Subject [|returnOutputOk $ setCounter 0
4                                                    $ setResets 0 output|]{
5          trigger stop:[|counter >= 2 && resets >= 1|];
6      };
7
8      do{
9          rec Loop{
10             count() from Subject[|
11                 returnOutputOk $ if (counter + 1) >= 3
12                                    then setCounter 0
13                                  $ setResets (resets + 1) output
14                                    else setCounter (counter + 1) output|];
15             Loop;
16         }
17     }interrupt{
18         stop() from Contract [|returnOk|];
19     }
20 }
```

Listing 6.3: Counter protocol

The sequence of actions we use for this protocol is straightforward: we call the startCounter() endpoint first, and then we repeatedly call count() until the trigger's condition is met. The setup in the Plutus Playground is presented in figure 6.5.

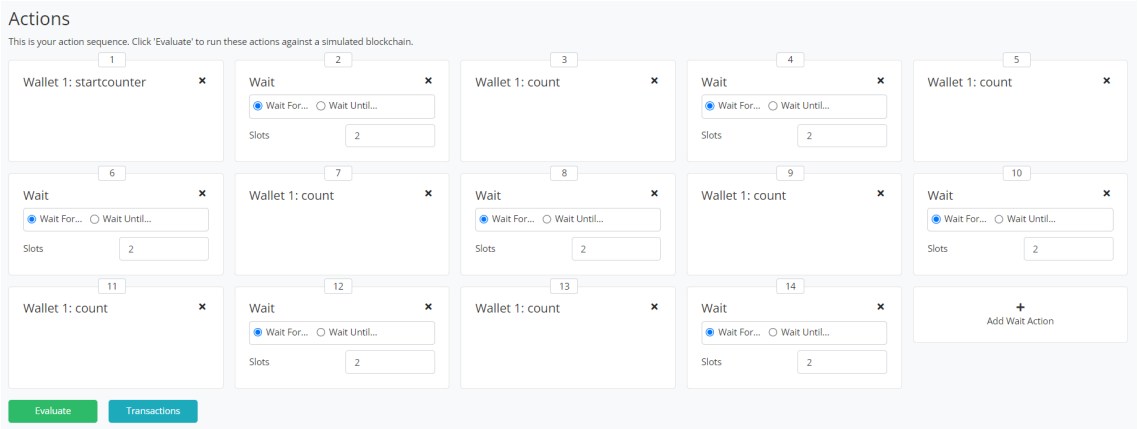



Figure 6.5: Sequence of actions for the Counter protocol in the Plutus Playground.

The first five count() calls are accepted, from these five, the last one activates the stop condition. The last count() operation is denied because the trigger already stopped the flow in the do block. Figure 6.6 shows the trigger activating. The trigger operates within the execution flow of the last

acceptable count(). The count() operation after the trigger is rejected (see the last line of Figure 6.6). Figure 6.7 illustrates the behavior of the action sequence in Figure 6.5, and the full log is in Appendix C.

```
==== Add slot 11 ====
Contract instance for W[1]: (ReceiveEndpointCall (EndpointDescription { getEndpointDescription: "count" }) (RawJson "{\"contents\":
  [{\"getEndpointDescription\":\"count\"},{\"unEndpointValue\":[]}],\"tag\":\"ExposeEndpointResp\"}"))
Validating transaction: 38bf94104160df4f5519d58a91706e671d0d57fde398e06e7ecc4436f55b35ce
==== Add slot 12 ====
Contract instance for W[1]: (ContractLog (RawJson "Successful transaction to state: StartCounterState 2 1 [(0,(POSIXTime {getPOSIXTime =
  1596059093999},Slot {getSlot = 2}))]"))
Validating transaction: 0da403be61fb78335e9a0458062d624aa755b32dc0b54b520ffe432ded61d00f
==== Add slot 13 ====
Contract instance for W[1]: (ContractLog (RawJson "Trigger successfully transited to state: StopState 2 1 [(0,(POSIXTime {getPOSIXTime =
  1596059093999},Slot {getSlot = 2}))]"))
Contract instance for W[1]: (ReceiveEndpointCall (EndpointDescription { getEndpointDescription: "count" }) (RawJson "{\"contents\":
  [{\"getEndpointDescription\":\"count\"},{\"unEndpointValue\":[]}],\"tag\":\"ExposeEndpointResp\"}"))
Contract instance for W[1]: (ContractLog (RawJson "Invalid invocation of endpoint count"))
```

Figure 6.6: The count() operation is rejected after the trigger's activation

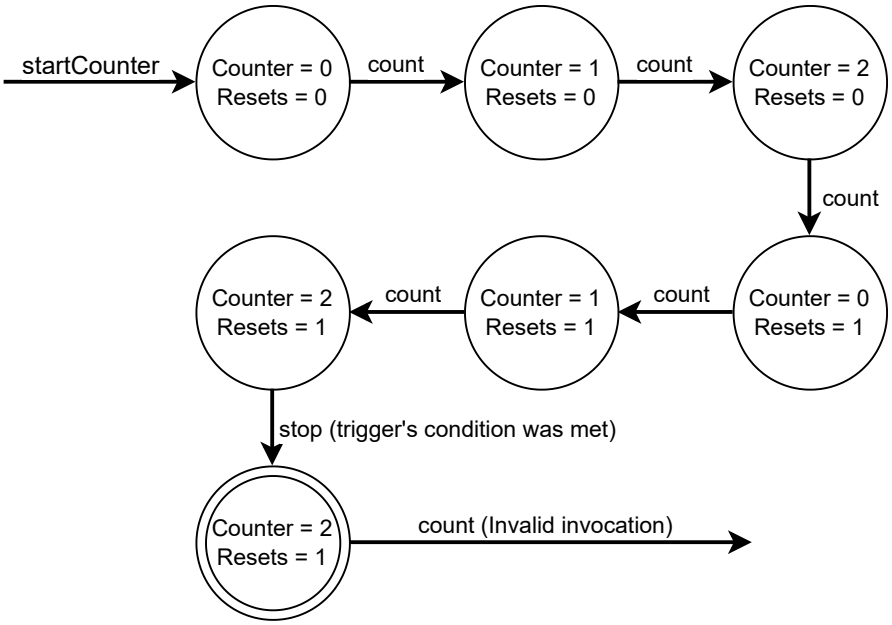



Figure 6.7: Counter protocol action sequence output

### 6.1.3 Testing individual state machines

We use our Supermarket protocol in Listing 4.24 (Appendix D presents the complete protocol with the business logic) to see the individual state machines in action. Recall that the protocol behaves as prescribed by the following SmartScribble piece of code (we repeat the protocol for reader's convenience): The protocol begins with the Owner opening the supermarket with the openMarket() operation. The next operation, begin(), allows clients to initiate their instances of the individual state machine described in the individual protocol named BuyingProcess. The individual protocol allows a Client to pick as many items as he wants by providing the identification of the item. The Client's protocol terminates when the stopBrowsing operation is called, and the total is paid with

the pay() operation. Wallet 1 is the Owner, and we have two clients, Wallets 2 and 3. We test this protocol in the Plutus playground with the action sequence shown in the interaction diagram in Figure 6.8. In the Plutus playground, we wait for 2 slots after each operation. Figure 6.9 illustrates the result of the inserted actions.

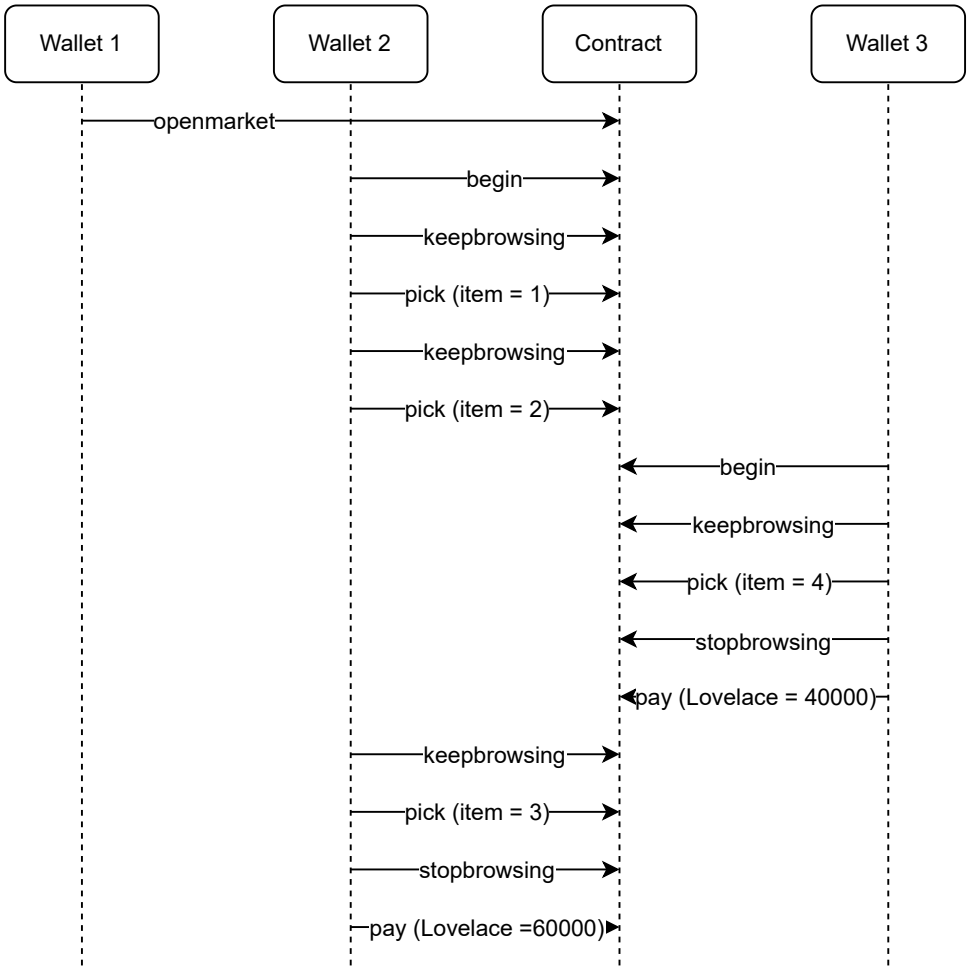



Figure 6.8: Supermarket protocol action sequence

Wallet 1 starts by opening the supermarket. Once it is open, the supermarket can now accept individual interactions from the clients. Wallet 2 enters the supermarket by calling the begin() endpoint, creating an instance of the individual state machine for him. He calls keepBrowsing() and then pick() to add item number 1 to his cart. Wallet 2 calls keepBrowsing() again, and then pick() to add item number 2 to his cart. Figure 6.10 shows that the global state machine's current state is OpenMarketState, meaning that the supermarket is open. Inside that global state, we see a list with one tuple inside. This list keeps a record of the Wallet's state in the individual state machine. This single tuple currently inside the list is the individual state machine instance belonging to Wallet 2. The first element of the tuple is the public key from Wallet 2, and the second element is its current individual state. Wallet 2's current individual state is BeginState and has a list of integers in it. That list is his cart with items number 1 and 2 inside. The empty list next to the list representing

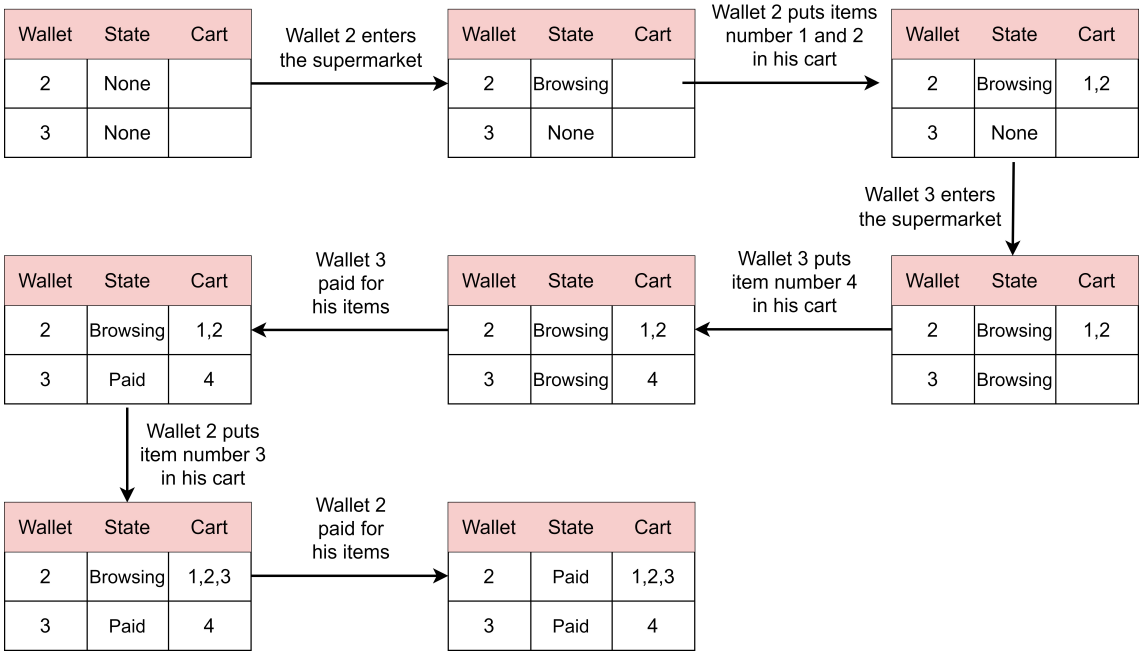



Figure 6.9: Illustrative representation of the Supermarket protocol in action

the Wallet's cart is a list for individual registered triggers. This list is added by default and allows triggers to be compatible with individual protocols.

Meanwhile, Wallet 3 enters with the begin() endpoint. Wallet 3 calls keepBrowsing() and then pick() to add item number 4 to his cart. Then he calls stopBrowsing() to finish browsing the supermarket and proceeds to pay for his items with the pay() endpoint. Figure 6.11 presents the log with Wallet 3's actions. Another tuple is added to the list inside the global state, representing Wallet 3's instance of the individual state machine. We can see Wallet 3's instance updating while Wallet 2's instance is still in the same state after he picked up item number 2. After Wallet 3 has called the pay() endpoint, he moves to the PayState with item number 4 in his cart while Wallet 2 is still in BeginState with items 1 and 2.

Next, Wallet 2 puts item number 3 on his cart. And finally, he calls stopBrowsing() and then pay () to pay for the items in his cart. Figure 6.12 shows these last steps from Wallet 2. His individual state is updated while Wallet 3 remains in PayState with item 4. Item 3 is added to Wallet 2's cart, and after the payment, he ends in PayState with items number 1, 2, and 3 in his cart. Appendix E presents the full log from this test.

## 6.2 Performance evaluation

We analyze the impact that our contracts have on the network, which depends on the on-chain code of the Plutus contract. Plutus provides tools to measure the performance of the on-chain code [30, 32]. These tools measure the minimum CPU and memory usage in ExCPU and ExMemory.

We do not analyze transaction fees because their evaluation is not reliable. Their costs depend on too many factors [16]. Table 6.1 presents the performance results from contracts compiled with

```
==== Add slot 7 ====
Contract instance for W[2]: (ReceiveEndpointCall (EndpointDescription { getEndpointDescription: "pick" }) (RawJson "{\"contents\":
  [{\"getEndpointDescription\":\"pick\"},{\"unEndpointValue\":{\"item\":1}}],\"tag\":\"ExposeEndpointResp\"}"))
Validating transaction: 0d9b8785be459c8eaab3c223e2d3b7d68e586c2a4e130e774d42fdc2b94649b0
==== Add slot 8 ====
Contract instance for W[2]: (ContractLog (RawJson "Successful transaction to solo state BeginState [1] [] : \nOpenMarketState
  [a2c20c77887ace1cd986193e4e75babd8993cfd56995cd5cfce609c2] [(80a4f45b56b88d1139da23bc4c3c75ec6d32943c087f250b86193ca7,BeginState [1] [])] []"))
==== Add slot 9 ====
Contract instance for W[2]: (ReceiveEndpointCall (EndpointDescription { getEndpointDescription: "keepbrowsing" }) (RawJson "{\"contents\":
  [{\"getEndpointDescription\":\"keepbrowsing\"},{\"unEndpointValue\":[]}],\"tag\":\"ExposeEndpointResp\"}"))
Validating transaction: d94842839282ed21e667ef6a3a2b15d663f1614e073df0709adb7f3e3dd02fc1
==== Add slot 10 ====
Contract instance for W[2]: (ContractLog (RawJson "Successful transaction to solo state KeepBrowsingState [1] [] : \nOpenMarketState
  [a2c20c77887ace1cd986193e4e75babd8993cfd56995cd5cfce609c2] [(80a4f45b56b88d1139da23bc4c3c75ec6d32943c087f250b86193ca7,KeepBrowsingState [1] [])] []"))
==== Add slot 11 ====
Contract instance for W[2]: (ReceiveEndpointCall (EndpointDescription { getEndpointDescription: "pick" }) (RawJson "{\"contents\":
  [{\"getEndpointDescription\":\"pick\"},{\"unEndpointValue\":{\"item\":2}}],\"tag\":\"ExposeEndpointResp\"}"))
Validating transaction: ebcd292df9568ebb48a50171654398ea29dd6c1fb841d7f1b15ec91fa23c12d4
==== Add slot 12 ====
Contract instance for W[2]: (ContractLog (RawJson "Successful transaction to solo state BeginState [2,1] [] : \nOpenMarketState
  [a2c20c77887ace1cd986193e4e75babd8993cfd56995cd5cfce609c2] [(80a4f45b56b88d1139da23bc4c3c75ec6d32943c087f250b86193ca7,BeginState [2,1] [])] []"))
```

Figure 6.10: Wallet 2 puts two items in its cart

```
==== Add slot 17 ====
Contract instance for W[3]: (ReceiveEndpointCall (EndpointDescription { getEndpointDescription: "pick" }) (RawJson "{\"contents\":
  [{\"getEndpointDescription\":\"pick\"},{\"unEndpointValue\":{\"item\":4}}],\"tag\":\"ExposeEndpointResp\"}"))
Validating transaction: 2c38f3d03c7ad68bff4a26c9166e42b5f3e29f1206249dc194efea6b98b977f3
==== Add slot 18 ====
Contract instance for W[3]: (ContractLog (RawJson "Successful transaction to solo state BeginState [4] [] : \nOpenMarketState
  [a2c20c77887ace1cd986193e4e75babd8993cfd56995cd5cfce609c2] [(80a4f45b56b88d1139da23bc4c3c75ec6d32943c087f250b86193ca7,BeginState [2,1] []),
  (2e0ad60c3207248cecd47dbde3d752e0aad141d6b8f81ac2c6eca27c,BeginState [4] [])] []"))
==== Add slot 19 ====
Contract instance for W[3]: (ReceiveEndpointCall (EndpointDescription { getEndpointDescription: "stopbrowsing" }) (RawJson "{\"contents\":
  [{\"getEndpointDescription\":\"stopbrowsing\"},{\"unEndpointValue\":[]}],\"tag\":\"ExposeEndpointResp\"}"))
Validating transaction: 8ee1e5aa0c12fa80c7742dc52c6948488a2c47c0360c1a947896d874cb94ecf4
==== Add slot 20 ====
Contract instance for W[3]: (ContractLog (RawJson "Successful transaction to solo state StopBrowsingState [4] [] : \nOpenMarketState
  [a2c20c77887ace1cd986193e4e75babd8993cfd56995cd5cfce609c2] [(80a4f45b56b88d1139da23bc4c3c75ec6d32943c087f250b86193ca7,BeginState [2,1] []),
  (2e0ad60c3207248cecd47dbde3d752e0aad141d6b8f81ac2c6eca27c,StopBrowsingState [4] [])] []"))
==== Add slot 21 ====
Contract instance for W[3]: (ReceiveEndpointCall (EndpointDescription { getEndpointDescription: "pay" }) (RawJson "{\"contents\":
  [{\"getEndpointDescription\":\"pay\"},{\"unEndpointValue\":{\"total\":{\"getValue\":[[{\"unCurrencySymbol\":\"\"},
  [[{\"unTokenName\":\"\"},40000]]]}}}],\"tag\":\"ExposeEndpointResp\"}"))
Validating transaction: e3c0118a6d2ac06a649049a06af2e0ec440a8e3bbdc8a757e91f2e274de9a864
==== Add slot 22 ====
Contract instance for W[3]: (ContractLog (RawJson "Successful transaction to solo state PayState [4] [] : \nOpenMarketState
  [a2c20c77887ace1cd986193e4e75babd8993cfd56995cd5cfce609c2] [(80a4f45b56b88d1139da23bc4c3c75ec6d32943c087f250b86193ca7,BeginState [2,1] []),
  (2e0ad60c3207248cecd47dbde3d752e0aad141d6b8f81ac2c6eca27c,PayState [4] [])] []"))
```

Figure 6.11: Wallet 3 puts an item in its cart and pays for it

our extended compiler version.

| Protocol | ExCPU | ExMemory | LOC |
|---|---|---|---|
| Auction | 10986337 | 37000 | 769 |
| Bargain | 10986337 | 37000 | 798 |
| Crowdfunding | 10807699 | 36400 | 688 |
| Supermarket | 12951355 | 43600 | 1022 |

Table 6.1: CPU and memory usage for Section 3 example protocols compiled with the extended version of SmartScribble

All these contracts use **signed** roles and define the business logic in the protocol code. The auction protocol uses complex triggers, and the supermarket protocol is the only one that uses individual state machines. We observe that the resource usage from the Auction, Bargain, and Crowdfunding protocols are very similar. Compared to the other three protocols, the supermarket protocol uses more resources. This happens because individual state machines append more code

```
==== Add slot 25 ====
Contract instance for W[2]: (ReceiveEndpointCall (EndpointDescription { getEndpointDescription: "pick" }) (RawJson "{\"contents\":
  [{\"getEndpointDescription\":\"pick\"},{\"unEndpointValue\":{\"item\":3}}],\"tag\":\"ExposeEndpointResp\"}"))
Validating transaction: abcd51613733f9ee2271c8c130b18a438e094fea41ad1676603e95a152dcd91f
==== Add slot 26 ====
Contract instance for W[2]: (ContractLog (RawJson "Successful transaction to solo state BeginState [3,2,1] [] : \nOpenMarketState
  [a2c20c77887ace1cd986193e4e75babd8993cfd56995cd5cfce609c2] [(80a4f45b56b88d1139da23bc4c3c75ec6d32943c087f250b86193ca7,BeginState [3,2,1] []),
  (2e0ad60c3207248cecd47dbde3d752e0aad141d6b8f81ac2c6eca27c,PayState [4] [])] []"))
==== Add slot 27 ====
Contract instance for W[2]: (ReceiveEndpointCall (EndpointDescription { getEndpointDescription: "stopbrowsing" }) (RawJson "{\"contents\":
  [{\"getEndpointDescription\":\"stopbrowsing\"},{\"unEndpointValue\":[]}],\"tag\":\"ExposeEndpointResp\"}"))
Validating transaction: a575e5d9ff904ffa22000d923ea09fd1baaf0e4ef27160ebf6e6c7774b71d2cc
==== Add slot 28 ====
Contract instance for W[2]: (ContractLog (RawJson "Successful transaction to solo state StopBrowsingState [3,2,1] [] : \nOpenMarketState
  [a2c20c77887ace1cd986193e4e75babd8993cfd56995cd5cfce609c2] [(80a4f45b56b88d1139da23bc4c3c75ec6d32943c087f250b86193ca7,StopBrowsingState [3,2,1] []),
  (2e0ad60c3207248cecd47dbde3d752e0aad141d6b8f81ac2c6eca27c,PayState [4] [])] []"))
==== Add slot 29 ====
Contract instance for W[2]: (ReceiveEndpointCall (EndpointDescription { getEndpointDescription: "pay" }) (RawJson "{\"contents\":
  [{\"getEndpointDescription\":\"pay\"},{\"unEndpointValue\":{\"total\":{\"getValue\":[[{\"unCurrencySymbol\":\"\"},
  [[{\"unTokenName\":\"\"},60000]]]}}}],\"tag\":\"ExposeEndpointResp\"}"))
Validating transaction: f872b3694df96e1daab045b65fc508ba2edccd718fd4e1407db04b0b0352f109
==== Add slot 30 ====
Contract instance for W[2]: (ContractLog (RawJson "Successful transaction to solo state PayState [3,2,1] [] : \nOpenMarketState
  [a2c20c77887ace1cd986193e4e75babd8993cfd56995cd5cfce609c2] [(80a4f45b56b88d1139da23bc4c3c75ec6d32943c087f250b86193ca7,PayState [3,2,1] []),
  (2e0ad60c3207248cecd47dbde3d752e0aad141d6b8f81ac2c6eca27c,PayState [4] [])] []"))
```

Figure 6.12: Wallet 2 puts one more item in its cart and pays for all the items he picked

to the on-chain part of the contract, resulting in more computations in the network. Figure 6.13 is a visual representation of Table 6.1.

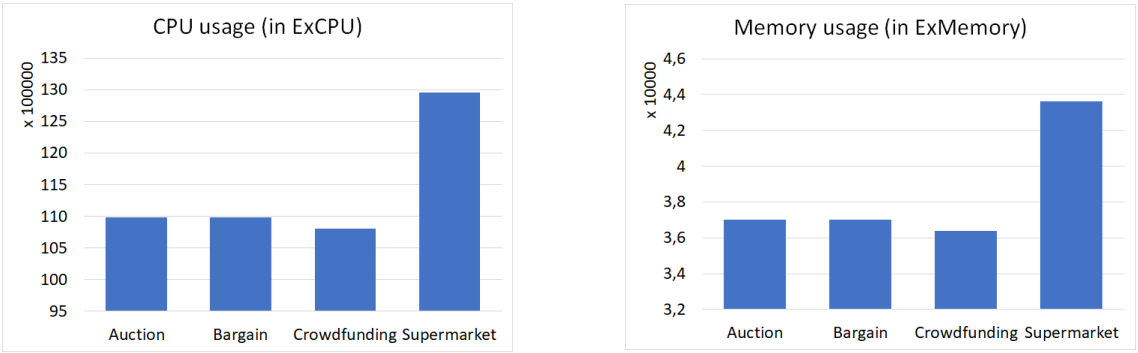



Figure 6.13: Visual representation of the data shown in Table 6.1

Business logic and triggers work in the off-chain part of the contract, causing no impact on the network's performance. Individual state machines have a part that impacts the on-chain performance. As seen in Table 6.1, they affect the performance. Role verification works in the on-chain code. We analyzed if just the presence of signed roles in the signature, regardless of having an interaction assigned or not, affects the performance. For this analysis, we utilized the Auction protocol. Table 6.2 presents the results.

The results demonstrate that the presence of at least one signed role slightly impacts the performance. The number of signed roles does not vary performance.

Interactions only perform the user validation if the protocol assigns it to a signed role. Since the number of signed roles in the protocol does not differentiate resource usage, we analyzed the number of verified interactions. For this analysis, we wrote the protocol in Listing 6.4.

| Number of signed roles | ExCPU | ExMemory |
|---|---|---|
| 0 | 10242012 | 34500 |
| 1 | 10986337 | 37000 |
| 2 | 10986337 | 37000 |
| 3 | 10986337 | 37000 |
| 4 | 10986337 | 37000 |
| 5 | 10986337 | 37000 |
| 6 | 10986337 | 37000 |

Table 6.2: Analysis of the presence of signed roles in the protocol signature

```
1  protocol MultiSig (role Requester,
2                     role Signer1,
3                     role Signer2,
4                     role Signer3,
5                     role Signer4,
6                     role Signer5,
7                     role Signer6) {
8      field sigs:[PubKeyHash];
9      requestSignatures() from Requester;
10     sign1() from Signer1;
11     sign2() from Signer2;
12     sign3() from Signer3;
13     sign4() from Signer4;
14     sign5() from Signer5;
15     sign6() from Signer6;
16     verifySignatures() from Requester;
17 }
```

Listing 6.4: Simple multi-signature protocol

Adding the keyword **signed** to a Signer adds a verified interaction for the on-chain code to process. We added the **signed** keyword to each role in the Multisig protocol and obtained the results in Table 6.3.

| Number of verified interactions | ExCPU | ExMemory |
|---|---|---|
| 0 | 10063374 | 33900 |
| 1 | 10718380 | 36100 |
| 2 | 10718380 | 36100 |
| 3 | 10718380 | 36100 |
| 4 | 10718380 | 36100 |
| 5 | 10718380 | 36100 |
| 6 | 10718380 | 36100 |

Table 6.3: Analysis of the number of verified interactions in the protocol in Listing 6.4

Like how the number of signed roles does not vary the performance, neither does the number of verified interactions. There is only a small increment if at least one verified interaction exists.

## 6.3 SmartScribble 1.0 versus SmartScribble 2.0

After analyzing the resources that the contracts created by the new version of the compiler consumes, we compared the new version with the old version. We used the Auction, Bargain, and Crowdfunding protocols and modified them to be compilable by both versions. We excluded the supermarket protocol from this analysis because the old version cannot handle individual state machines. The results obtained from this analysis are presented in Table 6.4.

| Protocol | CPU Old SmartScribble | CPU New SmartScribble | CPU Increase (%) | Memory Old SmartScribble | Memory New SmartScribble | Memory Increase (%) |
|---|---|---|---|---|---|---|
| Auction | 9497687 | 10986337 | 15,67 | 32000 | 37000 | 15,63 |
| Bargain | 9319049 | 10986337 | 17,89 | 31400 | 37000 | 17,83 |
| Crowdfunding | 9140411 | 10807699 | 18,24 | 30800 | 36400 | 18,18 |

Table 6.4: Performance comparison between both versions of SmartScribble

Old SmartScribble is the version we started working on. New SmartScribble is the version that resulted from the implementations of our extensions. CPU Increase and Memory Increase represents the resource usage increase from the Old version to the new one. As we can observe, there is an increase in resources from the old version. This is due to the signed role implementation. The old version did not verify who called each interaction which means that the old version did not actually complied with the specified protocol. We had to change the code generated for the on-chain part if we wanted to add this layer of security. So we should consider this increment in performance as necessary, rather than as a burden. Individual state machines would significantly impact more, but only if specified in the protocol. Figure 6.14 displays a visual representation of the results shown in Table 6.4.

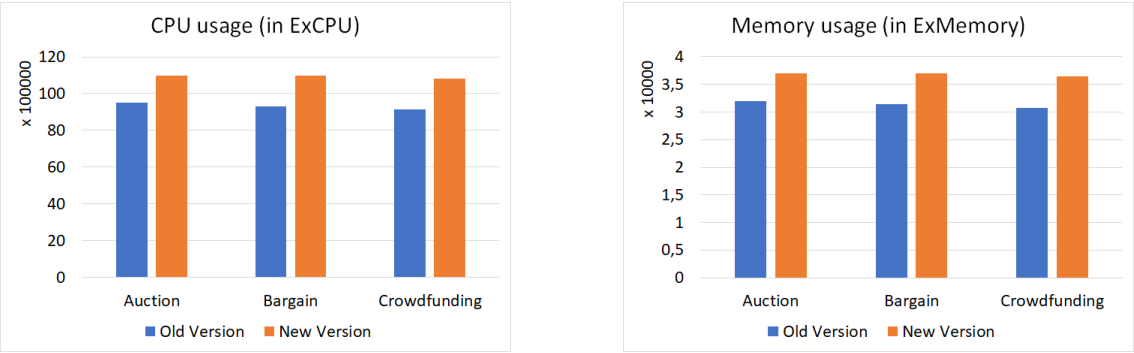



Figure 6.14: Visual representation of the data shown in Table 6.4

## 6.4 Discussion

SmartScribble 2.0 behaves as expected with the implementations from this work. Roles have the capability of validating the endpoint's caller; triggers accept conditions with more complexity; the compiler can provide a complete contract with the business logic incorporated in the protocol specification; and the users of a smart contract can have their own state with individual state machines.

The performance results show that our implementations consume approximately the same amount of resources as in the original version of SmartScribble. Individual state machines consume more resources because they append more on-chain code. We also observed that the number of interactions with role validation does not vary the resource usage. However, we observed that resource usage increased when compared to the old version. This increase resulted from the implementation of role validation and individual state machines. Business logic integration and complex triggers work off-chain and have no impact on the network's performance.

Is this increase bad? Everyone wants to spend as little as possible. However, cheap decisions do not always ensure good results. Signed roles are essential to ensure the protocol's security and reliability. Triggers provide more expressiveness. Individual state machines also provide more expressiveness and allow the specification of richer protocols. If we want a safe smart contract, we must ensure it is reliable. The increase is worth it if spending more resources provides a secure contract and allows SmartScribble to create more expressive contracts. Falcão's primary motivation for creating SmartScribble was security. Not investing in security would defeat the primary purpose of SmartScribble.

# Chapter 7

# Conclusion and future work

In this work we focused on the improvement of SmartScribble. SmartScribble is a protocol-based language designed to generate smart contracts in Plutus. It was proposed with the purpose of addressing vulnerabilities in Plutus smart contracts. Given a protocol specification, SmartScribble's compiler generates the boilerplate code and hands the contract to the developer to implement the required business logic.

The original implementation of SmartScribble lacked many essential features for smart contract development. With this work, we added more features to the language. Roles are now validated by verifying the public key of the user. Users are assigned to a specific role through the command line. Once the contract is generated, the user is bound to that role.

We endowed SmartScribble with the capability of tracking users' states through the specification of individual state machines. The definition of individual protocols recycles the syntax used for global protocols. Like global states, individual states can store data with variables defined in the `field`.

In the new version of SmartScribble, we allow the business logic to be integrated in the protocol. The compiler generates a complete smart contract and removes the need for further manual changes. However, this feature is optional: the programmer can still program the business logic later, manually, directly in the smart contract.

The new triggers enable the specification of more complex conditions. Besides that, a previous issue where users would become blocked when they specify a trigger is now resolved. Adopting the syntax that we used for the business logic, we enable the usage of Template Haskell to allow the programmer to implement the trigger's condition in the protocol specification.

The evaluation of the new version of SmartScribble exhibits an increase in resource usage compared to the old version. However, investing in security and reliability justifies the increase. We observed that the performance among all the implemented extensions is approximately the same for the use cases we have. However, the performance of the individual state machines is greater than the other extensions. We also observed that the smart contract's performance increases if signed roles are introduced, but the number of signed roles or verified interactions doesn't impact the performance.

Due to lack of time, the performance analysis was limited to our use cases. We could have

performed the functional evaluation by other means, such as surveys, but the Plutus community is not as large as other programming languages. All the implementations presented in this document first went through a proposal phase on the language's repository (Appendix F).

For the future, we can identify the following avenues for future research towards a more reliable and more expressive SmartScribble:

**Smart contract termination.** So far, we can ensure the correct order of interactions and who calls each endpoint, but we cannot ensure if the smart contract terminates. In some scenarios, we want a protocol to run forever, for instance, a protocol for an online shop. But in other cases, we need the protocol to terminate, for instance, the auction and the crowdfunding protocols. For future work, we plan to look into methods of ensuring that a smart contract terminates.

**Multiple individual state machines for each global state.** We introduced individual state machines. A programmer can define an individual protocol, and the user that follows that protocol does not need to depend on other clients. Our individual state machine is the same in all global states in which it can operate. For future work, we plan to work on individual state machines that can divide the work among the global states it can operate. The idea is to allow only certain individual operations in one of the global states.

For instance, a game with a registration phase before the participants can play. Individual clients are allowed to register for the game in one global state. No more registrations are allowed in the next global state, and the registered participants can start playing. We want to model this scenario with SmartScribble's individual state machines.

**SmartScribble's syntax for business logic** In the protocol, the business logic is defined with Plutus code. However, since the compiler accepts any Plutus expression, a programmer may introduce vulnerabilities. This situation also applies to the triggers' conditions. The Plutus code introduced in the protocol may not be reliable, so further work should be done to ensure that no vulnerabilities arise in this way. Either:

- validating the Plutus code inserted, or
- creating a syntax for the business logic and translating it to Plutus.

# Bibliography


[1] Nicola Atzei, Massimo Bartoletti, and Tiziana Cimoli. A survey of attacks on ethereum smart contracts (sok). In *International conference on principles of security and trust*, pages 164–186. Springer, 2017.

[2] Sam Blackshear, Evan Cheng, David L Dill, Victor Gao, Ben Maurer, Todd Nowacki, Alistair Pott, Shaz Qadeer, Dario Russi Rain, Stephane Sezer, et al. Move: A language with programmable resources. Technical report, 2019.

[3] Vitalik Buterin. A next-generation smart contract and decentralized application platform. *white paper*, 3(37), 2014.

[4] Manuel Chakravarty, James Chapman, Kenneth MacKenzie, Orestis Melkonian, Michael Peyton Jones, and Philip Wadler. The extended utxo model. In *International Conference on Financial Cryptography and Data Security*, pages 525–539. Springer, 2020.

[5] Chia-Chi Chang, Sean Sie, Marvin Janssen, Chi-Chen Liang, and Alejandro Pinto. Comparison of the top 10 smart contract programming languages in 2021. `https://pontem.network/posts/comparison-of-the-top-10-smart-contract-programming-languages-in-2021`, 2021.

[6] Sylvain Conchon, Alexandrina Korneva, and Fatiha Zaïdi. Verifying smart contracts with cubicle. In *International Symposium on Formal Methods*, pages 312–324. Springer, 2019.

[7] Damien Cosset. Blockchain: What is mining? `https://dev.to/damcosset/blockchain-what-is-mining-2eod`, 2018.

[8] Chris Dannen. *Introducing Ethereum and solidity*. Apress, 2017.

[9] Giuseppe Destefanis, Michele Marchesi, Marco Ortu, Roberto Tonelli, Andrea Bracciali, and Robert Hierons. Smart contracts vulnerabilities: a call for blockchain software engineering? In *2018 International Workshop on Blockchain Oriented Software Engineering (IWBOSE)*, pages 19–25. IEEE, 2018.

[10] Chris Dornan, Isaac Jones, and Simon Marlow. *Alex: A lexical analyser generator for Haskell*, 2003.

[11] Ethereum. Introduction to smart contracts. https://ethereum.org/en/developers/docs/smart-contracts/, 2020.

[12] Ethereum. Proof-of-work. https://ethereum.org/en/developers/docs/consensus-mechanisms/pow/, 2020.

[13] Ethereum. Solidity documentation. https://docs.soliditylang.org/en/latest/, 2021.

[14] Ethereum. Introduction to smart contracts. https://docs.soliditylang.org/en/latest/introduction-to-smart-contracts.html, 2022.

[15] Ethereum. Proof-of-stake. https://ethereum.org/en/developers/docs/consensus-mechanisms/pos/, 2022.

[16] Afonso Falcão. A protocol language for smart contracts. Master's thesis, Faculty of Sciences of the University of Lisbon, 2021.

[17] Afonso Falcão, Andreia Mordido, and Vasco T Vasconcelos. Protocol-based smart contract generation. *WTSC 22*, 2021.

[18] Osman Gazi Güçlütürk. The dao hack explained: Unfortunate take-off of smart contracts. medium. https://ogucluturk.medium.com/the-dao-hack-explained-unfortunate-take-off-of-smart-contracts-2bd8c8db3562, 2018.

[19] Haskell. Data.either. https://hackage.haskell.org/package/base-4.16.1.0/docs/Data-Either.html, 2017.

[20] Haskell. Language.haskell.meta.parse. https://hackage.haskell.org/package/haskell-src-meta-0.8.10/docs/Language-Haskell-Meta-Parse.html, 2021.

[21] Maurice Herlihy. Blockchains from a distributed computing perspective. *Communications of the ACM*, 62(2):78–85, 2019.

[22] IOHK. Cardano docs. https://docs.cardano.org/introduction, 2020.

[23] IOHK. Plutus playground. https://playground.plutus.iohkdev.io/, 2020.

[24] IOHK. Proof of stake. https://docs.cardano.org/new-to-cardano/proof-of-stake, 2020.

[25] IOHK. Ledger. https://playground.plutus.iohkdev.io/doc/haddock/plutus-ledger/html/Ledger.html, 2021.

[26] IOHK. Ledger.constraints. https://playground.plutus.iohkdev.io/doc/haddock/plutus-ledger-constraints/html/Ledger-Constraints.html, 2021.

[27] IOHK. Plutus pioneer program & alonzo testnet notes. https://plutus-pioneer-program.readthedocs.io/en/latest/index.html, 2021.

[28] IOHK. Plutus.contract. https://playground.plutus.iohkdev.io/doc/haddock/plutus-contract/html/Plutus-Contract.html, 2021.

[29] IOHK. Plutus.contract.statemachine. https://playground.plutus.iohkdev.io/doc/haddock/plutus-contract/html/Plutus-Contract-StateMachine.html, 2021.

[30] IOHK. Plutuscore.evaluation.machine.exmemory. https://playground.plutus.iohkdev.io/doc/haddock/plutus-core/html/PlutusCore-Evaluation-Machine-ExMemory.html, 2021.

[31] IOHK. Plutustx.errorcodes. https://github.com/input-output-hk/plutus/blob/master/plutus-tx/src/PlutusTx/ErrorCodes.hs, 2021.

[32] IOHK. Plutus.v1.ledger.api. https://playground.plutus.iohkdev.io/doc/haddock/plutus-ledger-api/html/Plutus-V1-Ledger-Api.html, 2021.

[33] IOHK. Plutus.v1.ledger.contexts. https://playground.plutus.iohkdev.io/doc/haddock/plutus-ledger-api/html/Plutus-V1-Ledger-Contexts.html, 2021.

[34] IOHK. Troubleshooting. https://playground.plutus.iohkdev.io/doc/plutus/troubleshooting.html#error-codes, 2021.

[35] IOHK. Advantages of using iele. https://testnets.cardano.org/en/virtual-machines/iele/about/advantages-of-using-iele/, 2022.

[36] Aggelos Kiayias, Alexander Russell, Bernardo David, and Roman Oliynykov. Ouroboros: A provably secure proof-of-stake blockchain protocol. In *Annual International Cryptology Conference*, pages 357–388. Springer, 2017.

[37] Sunny King and Scott Nadal. Ppcoin: Peer-to-peer crypto-currency with proof-of-stake. *self-published paper, August*, 19(1), 2012.

[38] Polina Vinogradova Lars Brünjes. *Plutus: Writing reliable smart contracts*. Input Output HK Limited, 2019.

[39] Simon Marlow and Andy Gill. *Happy: The Parser Generator for Haskell*, 2001.

[40] Demiro Massessi. Blockchain public / private key cryptography in a nutshell. https://medium.com/coinmonks/blockchain-public-private-key-cryptography-in-a-nutshell-b7776e475e7c, Nov 2021.

[41] Anastasia Mavridou and Aron Laszka. Designing secure ethereum smart contracts: A finite state machine based approach. In *International Conference on Financial Cryptography and Data Security*, pages 523–540. Springer, 2018.

[42] Du Mingxiao, Ma Xiaofeng, Zhang Zhe, Wang Xiangwei, and Chen Qijun. A review on consensus algorithm of blockchain. In *2017 IEEE international conference on systems, man, and cybernetics (SMC)*, pages 2567–2572. IEEE, 2017.

[43] MITRE. Nvd - cve-2018-10299. `https://nvd.nist.gov/vuln/detail/CVE-2018-10299`, Aug 2018.

[44] Mark Mossberg, Felipe Manzano, Eric Hennenfent, Alex Groce, Gustavo Grieco, Josselin Feist, Trent Brunson, and Artem Dinaburg. Manticore: A user-friendly symbolic execution framework for binaries and smart contracts. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 1186–1189. IEEE, 2019.

[45] Bernhard Mueller. Practical smart contract security analysis and exploitation. `https://medium.com/hackernoon/practical-smart-contract-security-analysis-and-exploitation-part-1-6c2f2320b0c`, 2019.

[46] Zari Oualid. Smart contract gas griefing attack: The hidden danger. `https://www.getsecureworld.com/blog/smart-contract-gas-griefing-attack-the-hidden-danger/`, 2022.

[47] Plutonomicon. Plutarch. `https://github.com/Plutonomicon/plutarch-plutus`, 2022.

[48] Grigore Roșu. K framework - an overview. `https://runtimeverification.com/blog/k-framework-an-overview/`, 2018.

[49] Grigore Roșu and Traian Florin Șerbănuță. An overview of the k semantic framework. *The Journal of Logic and Algebraic Programming*, 79(6):397–434, 2010.

[50] Sarwar Sayeed, Hector Marco-Gisbert, and Tom Caira. Smart contract: Attacks and protections. *IEEE Access*, 8:24416–24427, 2020.

[51] Will Shahda. Protect your solidity smart contracts from reentrancy attacks. `https://medium.com/coinmonks/protect-your-solidity-smart-contracts-from-reentrancy-attacks-9972c3af7c21`, 2020.

[52] Tim Sheard and Simon Peyton Jones. Template meta-programming for haskell. In *Proceedings of the 2002 ACM SIGPLAN workshop on Haskell*, pages 1–16, 2002.

[53] SmartContractSecurity. Insufficient gas griefing. `https://swcregistry.io/docs/SWC-126`, 2020.

[54] JP Smith. Echidna, a smart fuzzer for ethereum. https://blog.trailofbits.com/2018/03/09/echidna-a-smart-fuzzer-for-ethereum/, 2018.

[55] Mihail Sotnichek. Smart contract security guaranteed by the network: Cardano and zilliqa. https://www.apriorit.com/dev-blog/573-contract-security-cardano-zill, 2018.

[56] Petar Tsankov, Andrei Dan, Dana Drachsler-Cohen, Arthur Gervais, Florian Buenzli, and Martin Vechev. Securify: Practical security analysis of smart contracts. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, pages 67–82, 2018.

[57] Shuai Wang, Wenwen Ding, Juanjuan Li, Yong Yuan, Liwei Ouyang, and Fei-Yue Wang. Decentralized autonomous organizations: concept, model, and applications. *IEEE Transactions on Computational Social Systems*, 6(5):870–878, 2019.

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

[59] Nobuko Yoshida, Raymond Hu, Rumyana Neykova, and Nicholas Ng. The scribble protocol language. In *International Symposium on Trustworthy Global Computing*, pages 22–41. Springer, 2013.

# Appendix A

# Crowdfunding code

## A.1 Protocol

```
protocol Crowdfunding (role Contributor, signed role Owner){
    init () from Owner[|printMsg "Starting campaign"
                       returnOk|];
    rec Loop {
        choice at Owner{
            continue [|returnOk|]: {
                contribute (contribution:Value) from Contributor[|
                 printMsg "Collecting funds"
                 returnOutputOk $ setStateVal (stateVal + contribution) output|];
                Loop;
            }
            closeCrowdfund [|returnOk|]: {
                collectFunds () from Owner[|
                 printMsg "Collecting funds"
                 returnOutputOk $ setConstraint (Constraints.mustPayToPubKey (
                     head ownerId) stateVal)
                                  $ setStateVal mempty output|];
            }
        }
    }
}
```

## A.2 Crowdfunding Role validation code

```
module Crowdfunding where

-- | Declaration of the possible states for the State Machine:
data CrowdfundingState =
    None [PaymentPubKeyHash] [(Integer,(POSIXTime,Slot))]
    | InitState [PaymentPubKeyHash] [(Integer,(POSIXTime,Slot))]
    | CloseCrowdfundState [PaymentPubKeyHash] [(Integer,(POSIXTime,Slot))]
    | CollectFundsState [PaymentPubKeyHash] [(Integer,(POSIXTime,Slot))]
    | ContinueState [PaymentPubKeyHash] [(Integer,(POSIXTime,Slot))]
    deriving stock (Show, Generic)

-- | Declaration of the inputs that will be used for the transitions of the State Machine
data CrowdfundingInput =
    InitInput [PaymentPubKeyHash] [(Integer,(POSIXTime,Slot))] Value
```

```
15      | CloseCrowdfundInput [PaymentPubKeyHash] [(Integer,(POSIXTime,Slot))]
            Value
16      | CollectFundsInput [PaymentPubKeyHash] [(Integer,(POSIXTime,Slot))] Value
17      | ContinueInput [PaymentPubKeyHash] [(Integer,(POSIXTime,Slot))] Value
18      | ContributeInput [PaymentPubKeyHash] [(Integer,(POSIXTime,Slot))] Value
19      deriving stock (Show, Generic)
20
21
22  stringToKey :: String -> PaymentPubKeyHash
23  stringToKey key = PaymentPubKeyHash{unPaymentPubKeyHash = fromString key}
24
25  {-# INLINABLE validateKeys #-}
26  validateKeys :: [PaymentPubKeyHash] -> TxConstraints Void Void
27  validateKeys keys = if null keys
28      then mempty
29      else Constraints.mustSatisfyAnyOf $ map (Constraints.mustBeSignedBy) keys
30
31  -- | Beginning of the boilerplate code for the State Machine
32  {-# INLINABLE transition #-}
33  transition :: State CrowdfundingState -> CrowdfundingInput -> Maybe (
        TxConstraints Void Void, State CrowdfundingState)
34  transition State{stateData=oldData,stateValue} input = case (oldData, input) of
35      (None _ _, InitInput ownerId triggerTimeStamps stateVal) -> Just(mempty, State{
            stateData = InitState ownerId triggerTimeStamps, stateValue = stateVal})
36      (ContinueState _ _, ContributeInput ownerId triggerTimeStamps stateVal) ->
            Just(mempty, State{stateData = InitState ownerId triggerTimeStamps,
            stateValue = stateVal})
37      (InitState _ _, ContinueInput ownerId triggerTimeStamps stateVal) -> Just(
            mempty, State{stateData = ContinueState ownerId triggerTimeStamps,
            stateValue = stateVal})
38      (InitState _ _, CloseCrowdfundInput ownerId triggerTimeStamps stateVal) ->
            Just(mempty, State{stateData = CloseCrowdfundState ownerId
            triggerTimeStamps, stateValue = stateVal})
39      (CloseCrowdfundState _ _, CollectFundsInput ownerId triggerTimeStamps
            stateVal) -> Just(mempty, State{stateData = CollectFundsState ownerId
            triggerTimeStamps, stateValue = mempty})
40
41      _ -> Nothing
42
43  {-# INLINABLE transitionCheck #-}
44  transitionCheck :: CrowdfundingState -> CrowdfundingInput -> ScriptContext ->
        Bool
45  transitionCheck state input context = case (state, input) of
46      (None ownerId triggerTimeStamps, InitInput _ _ _) -> checkKeys ownerId
47      (ContinueState ownerId triggerTimeStamps, ContributeInput _ _ _) -> True
48      (InitState ownerId triggerTimeStamps, ContinueInput _ _ _) -> checkKeys
            ownerId
49      (InitState ownerId triggerTimeStamps, CloseCrowdfundInput _ _ _) -> checkKeys
            ownerId
50      (CloseCrowdfundState ownerId triggerTimeStamps, CollectFundsInput _ _ _) ->
            checkKeys ownerId
51
52      _ -> False
53      where
54          checkKeys keys = any (txSignedBy $ (scriptContextTxInfo context)) $ map
                unPaymentPubKeyHash keys
55
56  {-# INLINABLE machine #-}
57  machine :: SM.StateMachine CrowdfundingState CrowdfundingInput
58  machine = SM.StateMachine
59          { SM.smTransition = transition
```

```
60            , SM.smFinal = isFinal
61            , SM.smCheck = transitionCheck
62            , SM.smThreadToken = Nothing
63            }
64      where
65          isFinal (CollectFundsState _ _) = True
66          isFinal _ = False
67
68  initialiseSM :: Contract () CrowdfundingSchema SM.SMContractError (Maybe
        CrowdfundingState)
69  initialiseSM = do
70      currentState <- getCurrentStateSM client
71      case currentState of
72          Nothing -> do
73            let triggerTimeStamps = []
74                ownerId = [stringToKey "aaa"] --Placeholder key
75            SM.runInitialise client (None ownerId triggerTimeStamps) mempty
76            pure Nothing
77          x -> pure x
```

# Appendix B

# Auction code

## B.1 Protocol

```
1  protocol Auction (signed role Seller, role Buyer) {
2      field currBid:Value,lastBidder:PubKeyHash,asset:Integer;
3      initialise (initBid:Value, assetToSell:Integer) from Seller[|
4       returnOutputOk $ setCurrBid initBid
5                        $ setLastBidder noKey
6                        $ setAsset assetToSell output|]{
7          trigger closeAuctionTrigger:[|time >= timeStamp + 10000|];
8      };
9      do{
10         rec Loop {
11             bid (newBid:Value) from Buyer[|
12                 pkh <- ownPaymentPubKeyHash
13                 if (newBid `V.gt` mempty) && (newBid `V.gt` currBid)
14                 then returnOutputOk $ if (lastBidder == noKey)
15                     then setCurrBid newBid
16                          $ setLastBidder pkh
17                          $ setStateVal newBid output
18                     else setCurrBid newBid
19                          $ setLastBidder pkh
20                          $ setStateVal newBid
21                          $ setConstraint (Constraints.mustPayToPubKey lastBidder
                                currBid) output
22                 else returnError "Invalid Bid!"
23             |];
24             Loop;
25         }
26     } interrupt {
27       closeAuctionTrigger () from Contract[|returnOk|];
28     }
29     collectFundsAndGiveRewards () from Seller[|
30      returnOutputOk $ setConstraint (Constraints.mustPayToPubKey (head sellerId)
            stateVal)
31                       $ setStateVal mempty output|];
32 }
```

## B.2 Auction trigger code

The code below only presents the segments of the contract related to the functioning of triggers

```
1  module Auction where
2
3  data AuctionState =
4      None [PaymentPubKeyHash] [(Integer,(POSIXTime,Slot))]
5      | InitialiseState Value PaymentPubKeyHash Integer [PaymentPubKeyHash] [(
           Integer,(POSIXTime,Slot))]
6      | CloseAuctionTriggerState Value PaymentPubKeyHash Integer [
           PaymentPubKeyHash] [(Integer,(POSIXTime,Slot))]
7      | CollectFundsAndGiveRewardsState Value PaymentPubKeyHash Integer [
           PaymentPubKeyHash] [(Integer,(POSIXTime,Slot))]
8      deriving stock (Show, Generic)
9
10 -- | Declaration of the inputs that will be used for the transitions of the State Machine
11 data AuctionInput =
12     InitialiseInput Value PaymentPubKeyHash Integer [PaymentPubKeyHash] [(
           Integer,(POSIXTime,Slot))] Value
13     | BidInput Value PaymentPubKeyHash Integer [PaymentPubKeyHash] [(Integer,(
           POSIXTime,Slot))] Value
14     | CloseAuctionTriggerInput Value PaymentPubKeyHash Integer [
           PaymentPubKeyHash] [(Integer,(POSIXTime,Slot))] Value
15     | CollectFundsAndGiveRewardsInput Value PaymentPubKeyHash Integer [
           PaymentPubKeyHash] [(Integer,(POSIXTime,Slot))] Value
16     deriving stock (Show, Generic)
17
18 initialise :: Promise () AuctionSchema T.Text ()
19 initialise = endpoint @"initialise" @InitialiseParams $ \(InitialiseParams initBid
       assetToSell) -> do
20     -- | Received the parameters
21     currentState <- mapSMError' $ getCurrentStateSM client
22     if isValidCallInState currentState (InitialiseInput undefined undefined undefined
           undefined undefined undefined) then do
23     -- | Calling business logic function...
24       mapSMError' $ initialiseSM
25       oldVal <- fundsInContract
26       (sellerId, triggerTimeStamps) <- getContractInfo
27       logic <- initialiseLogic initBid assetToSell mempty (stringToKey "
             0000000000000000000000000000000000000000000000000000000000000000") 0
             sellerId triggerTimeStamps oldVal
28       case logic of
29         Left (currBid, lastBidder, asset, stateVal, constraint) -> do
30           currTime <- currentTime
31           currSlot <- currentSlot
32           let newTriggerTimeStamps = triggerTimeStamps ++ [(0,(currTime,currSlot))
                 ]
33           res <- mapSMError' $ runContractStep client (InitialiseInput currBid
                   lastBidder asset sellerId newTriggerTimeStamps stateVal) constraint
34           case res of
35                 Just s -> do
36                   logInfo ("Successful transaction to state: " <> show s)
37                 _ -> logError @String "Invalid operation in endpoint."
38         Right (Error x) -> logWarn @Text x
39     else
40       logError @String "Invalid invocation of endpoint initialise"
41
42 bid :: Promise () AuctionSchema T.Text ()
43 bid = endpoint @"bid" @BidParams $ \(BidParams newBid) -> do
44     -- | Received the parameters
45     currentState <- closeAuctionTriggerTrigger
46     if isValidCallInState currentState (BidInput undefined undefined undefined
           undefined undefined undefined) then do
47     -- | Calling business logic function...
```

```
48          oldVal <- fundsInContract
49          (currBidOld, lastBidderOld, assetOld, sellerId, triggerTimeStamps) <- getFields
50          logic <- bidLogic newBid currBidOld lastBidderOld assetOld sellerId
                triggerTimeStamps oldVal
51          case logic of
52            Left (currBid, lastBidder, asset, stateVal, constraint) -> do
53              res <- mapSMError' $ runContractStep client (BidInput currBid lastBidder
                      asset sellerId triggerTimeStamps stateVal) constraint
54              case res of
55                  Just s -> do
56                      logInfo ("Successful transaction to state: " <> show s)
57                      void $ closeAuctionTriggerTrigger
58                  _ -> logError @String "Invalid operation in endpoint."
59            Right (Error x) -> logWarn @Text x
60      else
61        logError @String "Invalid invocation of endpoint bid"
62
63  closeAuctionTriggerTrigger :: Contract () AuctionSchema T.Text (Maybe AuctionState)
64  closeAuctionTriggerTrigger = do
65      currentState <- mapSMError' $ getCurrentStateSM client
66      if isValidCallInState currentState (CloseAuctionTriggerInput undefined
            undefined undefined undefined undefined undefined)
67      then do
68          (currBid, lastBidder, asset, sellerId, triggerTimeStamps) <- getFields
69          stateVal <- fundsInContract
70          currTime <- currentTime
71          currSlot <- currentSlot
72          case (Prelude.lookup 0 triggerTimeStamps) of
73              Just (timeStamp,slotStamp) -> do
74                  let condition = closeAuctionTriggerCondition currTime currSlot
                        timeStamp slotStamp stateVal currBid lastBidder asset sellerId
75                  if condition then do
76                      triggerLogic <- closeAuctionTriggerLogic currBid lastBidder asset
                            sellerId triggerTimeStamps stateVal
77                      case triggerLogic of
78                          Left (argCurrBid, argLastBidder, argAsset, argStateVal,
                                constraint) -> do
79                              res <- mapSMError' $ runContractStep client (
                                    CloseAuctionTriggerInput argCurrBid argLastBidder
                                    argAsset sellerId triggerTimeStamps argStateVal)
                                    constraint
80                              case res of
81                                  Just s -> do
82                                      logInfo ("Trigger successfully transited to state: "
                                            <> show s)
83                                      pure res
84                                  _ -> do
85                                      logError @String "Trigger failed to perform its
                                            transition."
86                                      pure res
87                          Right (Error x) -> do
88                              logWarn @Text x
89                              mapSMError' $ getCurrentStateSM client
90                  else mapSMError' $ getCurrentStateSM client
91              Nothing -> do
92                  logWarn @Text $ T.pack "Trigger closeAuctionTrigger not registered!"
93                  mapSMError' $ getCurrentStateSM client
94      else mapSMError' $ getCurrentStateSM client
95
96  closeAuctionTriggerCondition :: POSIXTime ->
97                                  Slot ->
```

```
98                                  POSIXTime ->
99                                  Slot ->
100                                 Value ->
101                                 Value ->
102                                 PaymentPubKeyHash -> Integer -> [
                                        PaymentPubKeyHash] -> Bool
103 closeAuctionTriggerCondition time slot timeStamp slotStamp funds currBid lastBidder
        asset sellerId = do time >= timeStamp + 10000
```

# Appendix C

# Counter protocol log

```
Validating transaction: 0312
    c21da481701ccdc534cdc3d080111f688e5ba147eb5d74865fb180be287a
Add slot 1
Contract instance for W[1]: (ReceiveEndpointCall (EndpointDescription {
    getEndpointDescription: "startcounter" }) (RawJson "{\"contents\":[{\"
    getEndpointDescription\":\"startcounter\"},{\"unEndpointValue\":[]}],\"tag\":\"
    ExposeEndpointResp\"}"))
Validating transaction:
    fbbf62230878c4226b424632fc1304aee6e31ae85095f444c153af61a9e23163
Add slot 2
Validating transaction: 6893908201
    cf17599b083700b585d8a066997654864c49d08460d8cc1a6808e5
Add slot 3
Contract instance for W[1]: (ContractLog (RawJson "Successful transaction to state:
    StartCounterState 0 0 [(0,(POSIXTime {getPOSIXTime = 1596059093999},Slot {
    getSlot = 2}))]"))
Contract instance for W[1]: (ReceiveEndpointCall (EndpointDescription {
    getEndpointDescription: "count" }) (RawJson "{\"contents\":[{\"
    getEndpointDescription\":\"count\"},{\"unEndpointValue\":[]}],\"tag\":\"
    ExposeEndpointResp\"}"))
Validating transaction:
    ebc5fff221cebe314964c905a7b2c5097b952336368f9087cf886d4cfbb6b723
Add slot 4
Contract instance for W[1]: (ContractLog (RawJson "Successful transaction to state:
    StartCounterState 1 0 [(0,(POSIXTime {getPOSIXTime = 1596059093999},Slot {
    getSlot = 2}))]"))
Add slot 5
Contract instance for W[1]: (ReceiveEndpointCall (EndpointDescription {
    getEndpointDescription: "count" }) (RawJson "{\"contents\":[{\"
    getEndpointDescription\":\"count\"},{\"unEndpointValue\":[]}],\"tag\":\"
    ExposeEndpointResp\"}"))
Validating transaction:
    c0382b35d42d3da25e741269304367f59e782aefcd766b988741061cb35e572c
Add slot 6
Contract instance for W[1]: (ContractLog (RawJson "Successful transaction to state:
    StartCounterState 2 0 [(0,(POSIXTime {getPOSIXTime = 1596059093999},Slot {
    getSlot = 2}))]"))
Add slot 7
Contract instance for W[1]: (ReceiveEndpointCall (EndpointDescription {
    getEndpointDescription: "count" }) (RawJson "{\"contents\":[{\"
    getEndpointDescription\":\"count\"},{\"unEndpointValue\":[]}],\"tag\":\"
    ExposeEndpointResp\"}"))
Validating transaction: 5
    d25da25936e7159f8c6eabfbc428f7ebd16f583150537d35dd40efd5f7694de
```

```
Add slot 8
Contract instance for W[1]: (ContractLog (RawJson "Successful transaction to state:
    StartCounterState 0 1 [(0,(POSIXTime {getPOSIXTime = 1596059093999},Slot {
    getSlot = 2}))]"))
Add slot 9
Contract instance for W[1]: (ReceiveEndpointCall (EndpointDescription {
    getEndpointDescription: "count" }) (RawJson "{\"contents\":[{\"
    getEndpointDescription\":\"count\"},{\"unEndpointValue\":[]}],\"tag\":\"
    ExposeEndpointResp\"}"))
Validating transaction: 99
    aa628e8bafdf0fda8e44e9dd00369945bc13aeb32726aa2a9c7c4971147792
Add slot 10
Contract instance for W[1]: (ContractLog (RawJson "Successful transaction to state:
    StartCounterState 1 1 [(0,(POSIXTime {getPOSIXTime = 1596059093999},Slot {
    getSlot = 2}))]"))
Add slot 11
Contract instance for W[1]: (ReceiveEndpointCall (EndpointDescription {
    getEndpointDescription: "count" }) (RawJson "{\"contents\":[{\"
    getEndpointDescription\":\"count\"},{\"unEndpointValue\":[]}],\"tag\":\"
    ExposeEndpointResp\"}"))
Validating transaction: 38
    bf94104160df4f5519d58a91706e671d0d57fde398e06e7ecc4436f55b35ce
Add slot 12
Contract instance for W[1]: (ContractLog (RawJson "Successful transaction to state:
    StartCounterState 2 1 [(0,(POSIXTime {getPOSIXTime = 1596059093999},Slot {
    getSlot = 2}))]"))
Validating transaction: 0
    da403be61fb78335e9a0458062d624aa755b32dc0b54b520ffe432ded61d00f
Add slot 13
Contract instance for W[1]: (ContractLog (RawJson "Trigger successfully transited to
    state: StopState 2 1 [(0,(POSIXTime {getPOSIXTime = 1596059093999},Slot {
    getSlot = 2}))]"))
Contract instance for W[1]: (ReceiveEndpointCall (EndpointDescription {
    getEndpointDescription: "count" }) (RawJson "{\"contents\":[{\"
    getEndpointDescription\":\"count\"},{\"unEndpointValue\":[]}],\"tag\":\"
    ExposeEndpointResp\"}"))
Contract instance for W[1]: (ContractLog (RawJson "Invalid invocation of endpoint
    count"))
Add slot 14
Add slot 15
```

# Appendix D

# Supermarket code

## D.1 Protocol

```
protocol Supermarket (signed role Owner, role Client){
    openMarket() from Owner[|returnOk|];
    share begin() from cli : Client [|returnOk|] do BuyingProcess(cli);
    closeMarket() from Owner[|returnOk|];
}

individual protocol BuyingProcess (marketClient : Client){
    field cart:[Integer];
    rec Browsing{
        choice at marketClient{
            keepBrowsing[|returnOk|]:{
                pick(item:Integer) from marketClient[|
                    returnOutputOk $ setCart ([item] ++ cart) output
                |];
                Browsing;
            }
            stopBrowsing[|returnOk|]:{
                pay(total:Value) from marketClient[|
                    let totalVal = Ada.lovelaceValueOf $ (foldl (\tot next -> tot+next)
                        0 cart) * 10000
                    if total `V.geq` totalVal
                    then returnOutputOk $ setStateVal (stateVal + totalVal) output
                    else returnError "Your payment does not cover the total of your
                        cart!"
                |];
            }
        }
    }
}
```

## D.2 Supermarket individual state machine code

The code below presents relevant parts of the code regarding individual state machines. The code only shows two of the individual interactions of the protocol, one initial and one non-initial.

```
module Supermarket where


data SoloState =
```

```
5       SoloNone [(Integer,(POSIXTime,Slot))]
6       | BeginState [Integer] [(Integer,(POSIXTime,Slot))]
7       | KeepBrowsingState [Integer] [(Integer,(POSIXTime,Slot))]
8       | StopBrowsingState [Integer] [(Integer,(POSIXTime,Slot))]
9       | PayState [Integer] [(Integer,(POSIXTime,Slot))]
10      deriving (Show)
11
12  PlutusTx.unstableMakeIsData ''SoloState
13  PlutusTx.makeLift ''SoloState
14
15  data SoloInput =
16      BeginInput [Integer] [(Integer,(POSIXTime,Slot))]
17      | KeepBrowsingInput [Integer] [(Integer,(POSIXTime,Slot))]
18      | PickInput [Integer] [(Integer,(POSIXTime,Slot))]
19      | StopBrowsingInput [Integer] [(Integer,(POSIXTime,Slot))]
20      | PayInput [Integer] [(Integer,(POSIXTime,Slot))]
21      deriving (Show)
22
23  PlutusTx.unstableMakeIsData ''SoloInput
24  PlutusTx.makeLift ''SoloInput
25
26  {-# INLINABLE soloTransition #-}
27  soloTransition :: SoloState -> SoloInput -> Maybe SoloState
28  soloTransition state input = case (state,input) of
29      (SoloNone _, BeginInput cart soloTriggerTimeStamps) -> Just $ BeginState cart
            soloTriggerTimeStamps
30      (BeginState _ _, StopBrowsingInput cart soloTriggerTimeStamps) -> Just $
            StopBrowsingState cart soloTriggerTimeStamps
31      (BeginState _ _, KeepBrowsingInput cart soloTriggerTimeStamps) -> Just $
            KeepBrowsingState cart soloTriggerTimeStamps
32      (KeepBrowsingState _ _, PickInput cart soloTriggerTimeStamps) -> Just $
            BeginState cart soloTriggerTimeStamps
33      (StopBrowsingState _ _, PayInput cart soloTriggerTimeStamps) -> Just $
            PayState cart soloTriggerTimeStamps
34      _ -> Nothing
35
36  type EntityList = [(PaymentPubKeyHash,SoloState)]
37
38  -- | Declaration of the possible states for the State Machine:
39  data SupermarketState =
40      None [PaymentPubKeyHash] EntityList [(Integer,(POSIXTime,Slot))]
41      | OpenMarketState [PaymentPubKeyHash] EntityList [(Integer,(POSIXTime,Slot))
            ]
42      | CloseMarketState [PaymentPubKeyHash] EntityList [(Integer,(POSIXTime,Slot))
            ]
43      deriving stock (Show, Generic)
44
45  -- | Declaration of the inputs that will be used for the transitions of the State Machine
46  data SupermarketInput =
47      OpenMarketInput [PaymentPubKeyHash] EntityList [(Integer,(POSIXTime,Slot))]
            Value
48      | CloseMarketInput [PaymentPubKeyHash] EntityList [(Integer,(POSIXTime,Slot))
            ] Value
49      | SoloSMInput [PaymentPubKeyHash] SoloState SoloInput EntityList [(Integer,(
            POSIXTime,Slot))] Value
50      deriving stock (Show, Generic)
51
52  -- | Beginning of the boilerplate code for the State Machine
53  {-# INLINABLE transition #-}
54  transition :: State SupermarketState -> SupermarketInput -> Maybe (TxConstraints
        Void Void, State SupermarketState)
```

```
55 transition State{stateData=oldData,stateValue} input = case (oldData, input) of
56     (None _ _ _, OpenMarketInput ownerId soloSMs triggerTimeStamps stateVal) ->
           Just(mempty, State{stateData = OpenMarketState ownerId soloSMs
           triggerTimeStamps, stateValue = stateVal})
57
58     (OpenMarketState _ _ _, CloseMarketInput ownerId soloSMs triggerTimeStamps
           stateVal) -> Just(mempty, State{stateData = CloseMarketState ownerId
           soloSMs triggerTimeStamps, stateValue = mempty})
59     (OpenMarketState _ _ _, SoloSMInput ownerId soloState soloInput soloSMs
           triggerTimeStamps stateVal) ->
60         case (soloTransition soloState soloInput) of
61             Nothing -> Nothing
62             Just newstate -> Just(mempty, State{stateData = OpenMarketState
                   ownerId soloSMs triggerTimeStamps, stateValue = stateVal})
63     _ -> Nothing
64
65 getSoloFields :: SoloState -> Contract () SupermarketSchema T.Text ([Integer], [(
       Integer,(POSIXTime,Slot))])
66 getSoloFields state = case state of
67     (BeginState cart soloTriggerTimeStamps) -> pure $ (cart,
           soloTriggerTimeStamps)
68     (KeepBrowsingState cart soloTriggerTimeStamps) -> pure $ (cart,
           soloTriggerTimeStamps)
69     (StopBrowsingState cart soloTriggerTimeStamps) -> pure $ (cart,
           soloTriggerTimeStamps)
70     (PayState cart soloTriggerTimeStamps) -> pure $ (cart, soloTriggerTimeStamps)
71
72 getEntitys :: Contract () SupermarketSchema T.Text (Maybe [(PaymentPubKeyHash,
       SoloState)])
73 getEntitys = do
74     currState <- mapSMError' $ getCurrentStateSM client
75     case currState of
76         Just (OpenMarketState _ soloSMs _) -> pure $ Just soloSMs
77         Just (CloseMarketState _ soloSMs _) -> pure $ Just soloSMs
78         _ -> pure $ Nothing
79
80 updateEntity :: PaymentPubKeyHash -> SoloState -> EntityList -> EntityList
81 updateEntity _ _ [] = []
82 updateEntity pkh newstate (x:xs)
83     | fst x == pkh = (pkh,newstate):xs
84     | otherwise = x:updateEntity pkh newstate xs
85
86
87 begin :: Promise () SupermarketSchema T.Text ()
88 begin = endpoint @"begin" $ \() -> do
89     -- | Received the parameters
90     pkh <- ownPaymentPubKeyHash
91     currentState <- mapSMError' $ getCurrentStateSM client
92     ents <- getEntitys
93     case ents of
94         Just oldEntities -> do
95             let entities = oldEntities ++ [(pkh,(SoloNone []))]
96             case (Prelude.lookup pkh entities) of
97                 Just currstate -> do
98                     if isValidCallInState currentState (SoloSMInput undefined
                           currstate (BeginInput undefined undefined) undefined
                           undefined undefined) then do
99                         oldval <- fundsInContract
100                        (ownerId, soloSMs, triggerTimeStamps) <- getFields
101                        (soloTriggerTimeStamps) <- getSoloContractInfo currstate
102                            -- | Calling business logic function...
```

```
103                         logic <- beginLogic ownerId soloSMs triggerTimeStamps []
                                soloTriggerTimeStamps oldval
104                         case logic of
105                             Left(cart, val, constraint) -> do
106                                 let newinput = BeginInput cart
                                        soloTriggerTimeStamps
107                                 case (soloTransition currstate newinput) of
108                                     Just newstate -> do
109                                         let newlist = updateEntity pkh newstate
                                                entities
110                                         res <- mapSMError' $ runContractStep
                                                client (SoloSMInput ownerId
                                                currstate newinput newlist
                                                triggerTimeStamps val) constraint
111                                         case res of
112                                             Just s -> do
113                                                 logInfo ("Successful transaction
                                                        to solo state " <> show
                                                        newstate <> " : \n" <>
                                                        show s)
114                                             _ -> logError @String "Invalid
                                                    operation in endpoint begin."
115                                     Nothing -> logError @String "Invalid
                                            operation in endpoint begin."
116                             Right (Error x) -> logWarn @Text x
117                     else logError @String "Invalid invocation of endpoint begin"
118                 Nothing -> logError @String "Entity not registered."
119         _ -> logError @String "List of entities not found."
120
121 keepBrowsing :: Promise () SupermarketSchema T.Text ()
122 keepBrowsing = endpoint @"keepbrowsing" $ \() -> do
123     -- | Received the parameters
124     pkh <- ownPaymentPubKeyHash
125     currentState <- mapSMError' $ getCurrentStateSM client
126     ents <- getEntitys
127     case ents of
128         Just entities -> do
129             case (Prelude.lookup pkh entities) of
130                 Just currstate -> do
131                     if isValidCallInState currentState (SoloSMInput undefined
                            currstate (KeepBrowsingInput undefined undefined)
                            undefined undefined undefined) then do
132                         oldval <- fundsInContract
133                         (ownerId, soloSMs, triggerTimeStamps) <- getFields
134                         (cartOld, soloTriggerTimeStamps) <- getSoloFields currstate
135                             -- | Calling business logic function...
136                         logic <- keepBrowsingLogic ownerId soloSMs
                                triggerTimeStamps cartOld soloTriggerTimeStamps
                                oldval
137                         case logic of
138                             Left(cart, val, constraint) -> do
139                                 let newinput = KeepBrowsingInput cart
                                        soloTriggerTimeStamps
140                                 case (soloTransition currstate newinput) of
141                                     Just newstate -> do
142                                         let newlist = updateEntity pkh newstate
                                                entities
143                                         res <- mapSMError' $ runContractStep
                                                client (SoloSMInput ownerId
                                                currstate newinput newlist
                                                triggerTimeStamps val) constraint
```

```
144                                    case res of
145                                        Just s -> do
146                                            logInfo ("Successful transaction
                                                   to solo state " <> show
                                                   newstate <> " : \n" <>
                                                   show s)
147                                        _ -> logError @String "Invalid
                                               operation in endpoint
                                               keepBrowsing."
148                                    Nothing -> logError @String "Invalid
                                           operation in endpoint keepBrowsing."
149                                Right (Error x) -> logWarn @Text x
150                            else logError @String "Invalid invocation of endpoint
                                   keepBrowsing"
151                        Nothing -> logError @String "Entity not registered."
152                _ -> logError @String "List of entities not found."
```

# Appendix E

# Supermarket output log

## E.1 Full output log

```
Add slot 1
Validating transaction:
    f12240eb2e8ac5a96c8c62b755c061183861e2bf6c233ee93b87ac0db5e199f3
Add slot 2
Validating transaction: 2
    a759db6e61630742eb484cccf6084525732105a4a1d0a6734eaa7f04ba2e85e
Add slot 3
Contract instance for W[1]: (ContractLog (RawJson "Successful transaction to state:
    OpenMarketState [
    a2c20c77887ace1cd986193e4e75babd8993cfd56995cd5cfce609c2] [] []"))
Contract instance for W[2]: (ReceiveEndpointCall (EndpointDescription {
    getEndpointDescription: "begin" }) (RawJson "{\"contents\":[{\"
    getEndpointDescription\":\"begin\"},{\"unEndpointValue\":[]}],\"tag\":\"
    ExposeEndpointResp\"}"))
Validating transaction: 4760
    aa2713168a788e4659b8118914b8329bd750547bed86493651b15b711b66
Add slot 4
Contract instance for W[2]: (ContractLog (RawJson "Successful transaction to solo
    state BeginState [] [] : \nOpenMarketState [
    a2c20c77887ace1cd986193e4e75babd8993cfd56995cd5cfce609c2] [(80
    a4f45b56b88d1139da23bc4c3c75ec6d32943c087f250b86193ca7,BeginState []
    [])] []"))
Add slot 5
Contract instance for W[2]: (ReceiveEndpointCall (EndpointDescription {
    getEndpointDescription: "keepbrowsing" }) (RawJson "{\"contents\":[{\"
    getEndpointDescription\":\"keepbrowsing\"},{\"unEndpointValue\":[]}],\"tag\":\"
    ExposeEndpointResp\"}"))
Validating transaction: 79
    f499712ff118182387b1a2be8c4b568c11ec8a7a091f28f9ad8d091205163e
Add slot 6
Contract instance for W[2]: (ContractLog (RawJson "Successful transaction to solo
    state KeepBrowsingState [] [] : \nOpenMarketState [
    a2c20c77887ace1cd986193e4e75babd8993cfd56995cd5cfce609c2] [(80
    a4f45b56b88d1139da23bc4c3c75ec6d32943c087f250b86193ca7,
    KeepBrowsingState [] [])] []"))
Add slot 7
Contract instance for W[2]: (ReceiveEndpointCall (EndpointDescription {
    getEndpointDescription: "pick" }) (RawJson "{\"contents\":[{\"
    getEndpointDescription\":\"pick\"},{\"unEndpointValue\":{\"item\":1}}],\"tag\":\"
    ExposeEndpointResp\"}"))
Validating transaction: 0
    d9b8785be459c8eaab3c223e2d3b7d68e586c2a4e130e774d42fdc2b94649b0
```

```
Add slot 8
Contract instance for W[2]: (ContractLog (RawJson "Successful transaction to solo
    state BeginState [1] [] : \nOpenMarketState [
    a2c20c77887ace1cd986193e4e75babd8993cfd56995cd5cfce609c2] [(80
    a4f45b56b88d1139da23bc4c3c75ec6d32943c087f250b86193ca7,BeginState [1]
     [])] []"))
Add slot 9
Contract instance for W[2]: (ReceiveEndpointCall (EndpointDescription {
    getEndpointDescription: "keepbrowsing" }) (RawJson "{\"contents\":[{\"
    getEndpointDescription\":\"keepbrowsing\"},{\"unEndpointValue\":[]}],\"tag\":\"
    ExposeEndpointResp\"}"))
Validating transaction:
    d94842839282ed21e667ef6a3a2b15d663f1614e073df0709adb7f3e3dd02fc1
Add slot 10
Contract instance for W[2]: (ContractLog (RawJson "Successful transaction to solo
    state KeepBrowsingState [1] [] : \nOpenMarketState [
    a2c20c77887ace1cd986193e4e75babd8993cfd56995cd5cfce609c2] [(80
    a4f45b56b88d1139da23bc4c3c75ec6d32943c087f250b86193ca7,
    KeepBrowsingState [1] [])] []"))
Add slot 11
Contract instance for W[2]: (ReceiveEndpointCall (EndpointDescription {
    getEndpointDescription: "pick" }) (RawJson "{\"contents\":[{\"
    getEndpointDescription\":\"pick\"},{\"unEndpointValue\":{\"item\":2}}],\"tag\":\"
    ExposeEndpointResp\"}"))
Validating transaction:
    ebcd292df9568ebb48a50171654398ea29dd6c1fb841d7f1b15ec91fa23c12d4
Add slot 12
Contract instance for W[2]: (ContractLog (RawJson "Successful transaction to solo
    state BeginState [2,1] [] : \nOpenMarketState [
    a2c20c77887ace1cd986193e4e75babd8993cfd56995cd5cfce609c2] [(80
    a4f45b56b88d1139da23bc4c3c75ec6d32943c087f250b86193ca7,BeginState
    [2,1] [])] []"))
Add slot 13
Contract instance for W[3]: (ReceiveEndpointCall (EndpointDescription {
    getEndpointDescription: "begin" }) (RawJson "{\"contents\":[{\"
    getEndpointDescription\":\"begin\"},{\"unEndpointValue\":[]}],\"tag\":\"
    ExposeEndpointResp\"}"))
Validating transaction: 70
    fbbaec0e30564963aefaaf291eeb661f9967f3c1d5903d7fbe604a4c1921c1
Add slot 14
Contract instance for W[3]: (ContractLog (RawJson "Successful transaction to solo
    state BeginState [] [] : \nOpenMarketState [
    a2c20c77887ace1cd986193e4e75babd8993cfd56995cd5cfce609c2] [(80
    a4f45b56b88d1139da23bc4c3c75ec6d32943c087f250b86193ca7,BeginState
    [2,1] []),(2e0ad60c3207248cecd47dbde3d752e0aad141d6b8f81ac2c6eca27c,
    BeginState [] [])] []"))
Add slot 15
Contract instance for W[3]: (ReceiveEndpointCall (EndpointDescription {
    getEndpointDescription: "keepbrowsing" }) (RawJson "{\"contents\":[{\"
    getEndpointDescription\":\"keepbrowsing\"},{\"unEndpointValue\":[]}],\"tag\":\"
    ExposeEndpointResp\"}"))
Validating transaction: 88
    e4ceb0350c5a26c69915a9fc68f4ddbddd614eeaf9254395ec36d1649249b1
Add slot 16
Contract instance for W[3]: (ContractLog (RawJson "Successful transaction to solo
    state KeepBrowsingState [] [] : \nOpenMarketState [
    a2c20c77887ace1cd986193e4e75babd8993cfd56995cd5cfce609c2] [(80
    a4f45b56b88d1139da23bc4c3c75ec6d32943c087f250b86193ca7,BeginState
    [2,1] []),(2e0ad60c3207248cecd47dbde3d752e0aad141d6b8f81ac2c6eca27c,
    KeepBrowsingState [] [])] []"))
Add slot 17
```

```
Contract instance for W[3]: (ReceiveEndpointCall (EndpointDescription {
    getEndpointDescription: "pick" }) (RawJson "{\"contents\":[{\"
    getEndpointDescription\":\"pick\"},{\"unEndpointValue\":{\"item\":4}}],\"tag\":\"
    ExposeEndpointResp\"}"))
Validating transaction: 2
    c38f3d03c7ad68bff4a26c9166e42b5f3e29f1206249dc194efea6b98b977f3
Add slot 18
Contract instance for W[3]: (ContractLog (RawJson "Successful transaction to solo
    state BeginState [4] [] : \nOpenMarketState [
    a2c20c77887ace1cd986193e4e75babd8993cfd56995cd5cfce609c2] [(80
    a4f45b56b88d1139da23bc4c3c75ec6d32943c087f250b86193ca7,BeginState
    [2,1] []),(2e0ad60c3207248cecd47dbde3d752e0aad141d6b8f81ac2c6eca27c,
    BeginState [4] [])] []"))
Add slot 19
Contract instance for W[3]: (ReceiveEndpointCall (EndpointDescription {
    getEndpointDescription: "stopbrowsing" }) (RawJson "{\"contents\":[{\"
    getEndpointDescription\":\"stopbrowsing\"},{\"unEndpointValue\":[]}],\"tag\":\"
    ExposeEndpointResp\"}"))
Validating transaction: 8
    ee1e5aa0c12fa80c7742dc52c6948488a2c47c0360c1a947896d874cb94ecf4
Add slot 20
Contract instance for W[3]: (ContractLog (RawJson "Successful transaction to solo
    state StopBrowsingState [4] [] : \nOpenMarketState [
    a2c20c77887ace1cd986193e4e75babd8993cfd56995cd5cfce609c2] [(80
    a4f45b56b88d1139da23bc4c3c75ec6d32943c087f250b86193ca7,BeginState
    [2,1] []),(2e0ad60c3207248cecd47dbde3d752e0aad141d6b8f81ac2c6eca27c,
    StopBrowsingState [4] [])] []"))
Add slot 21
Contract instance for W[3]: (ReceiveEndpointCall (EndpointDescription {
    getEndpointDescription: "pay" }) (RawJson "{\"contents\":[{\"
    getEndpointDescription\":\"pay\"},{\"unEndpointValue\":{\"total\":{\"getValue\":[[{\"
    unCurrencySymbol\":\"\"},[[{\"unTokenName\":\"\"},40000]]]}}}],\"tag\":\"
    ExposeEndpointResp\"}"))
Validating transaction:
    e3c0118a6d2ac06a649049a06af2e0ec440a8e3bbdc8a757e91f2e274de9a864
Add slot 22
Contract instance for W[3]: (ContractLog (RawJson "Successful transaction to solo
    state PayState [4] [] : \nOpenMarketState [
    a2c20c77887ace1cd986193e4e75babd8993cfd56995cd5cfce609c2] [(80
    a4f45b56b88d1139da23bc4c3c75ec6d32943c087f250b86193ca7,BeginState
    [2,1] []),(2e0ad60c3207248cecd47dbde3d752e0aad141d6b8f81ac2c6eca27c,
    PayState [4] [])] []"))
Add slot 23
Contract instance for W[2]: (ReceiveEndpointCall (EndpointDescription {
    getEndpointDescription: "keepbrowsing" }) (RawJson "{\"contents\":[{\"
    getEndpointDescription\":\"keepbrowsing\"},{\"unEndpointValue\":[]}],\"tag\":\"
    ExposeEndpointResp\"}"))
Validating transaction:
    cb9300446c871d2a91c4bb511a252d386594c86f33c66967334775579f11d751
Add slot 24
Contract instance for W[2]: (ContractLog (RawJson "Successful transaction to solo
    state KeepBrowsingState [2,1] [] : \nOpenMarketState [
    a2c20c77887ace1cd986193e4e75babd8993cfd56995cd5cfce609c2] [(80
    a4f45b56b88d1139da23bc4c3c75ec6d32943c087f250b86193ca7,
    KeepBrowsingState [2,1] []),(2
    e0ad60c3207248cecd47dbde3d752e0aad141d6b8f81ac2c6eca27c,PayState [4]
     [])] []"))
Add slot 25
Contract instance for W[2]: (ReceiveEndpointCall (EndpointDescription {
    getEndpointDescription: "pick" }) (RawJson "{\"contents\":[{\"
    getEndpointDescription\":\"pick\"},{\"unEndpointValue\":{\"item\":3}}],\"tag\":\"
```

```
    ExposeEndpointResp\"}"))
Validating transaction:
    abcd51613733f9ee2271c8c130b18a438e094fea41ad1676603e95a152dcd91f
Add slot 26
Contract instance for W[2]: (ContractLog (RawJson "Successful transaction to solo
    state BeginState [3,2,1] [] : \nOpenMarketState [
    a2c20c77887ace1cd986193e4e75babd8993cfd56995cd5cfce609c2] [(80
    a4f45b56b88d1139da23bc4c3c75ec6d32943c087f250b86193ca7,BeginState
    [3,2,1] []),(2e0ad60c3207248cecd47dbde3d752e0aad141d6b8f81ac2c6eca27c,
    PayState [4] [])] []"))
Add slot 27
Contract instance for W[2]: (ReceiveEndpointCall (EndpointDescription {
    getEndpointDescription: "stopbrowsing" }) (RawJson "{\"contents\":[{\"
    getEndpointDescription\":\"stopbrowsing\"},{\"unEndpointValue\":[]}],\"tag\":\"
    ExposeEndpointResp\"}"))
Validating transaction:
    a575e5d9ff904ffa22000d923ea09fd1baaf0e4ef27160ebf6e6c7774b71d2cc
Add slot 28
Contract instance for W[2]: (ContractLog (RawJson "Successful transaction to solo
    state StopBrowsingState [3,2,1] [] : \nOpenMarketState [
    a2c20c77887ace1cd986193e4e75babd8993cfd56995cd5cfce609c2] [(80
    a4f45b56b88d1139da23bc4c3c75ec6d32943c087f250b86193ca7,
    StopBrowsingState [3,2,1] []),(2
    e0ad60c3207248cecd47dbde3d752e0aad141d6b8f81ac2c6eca27c,PayState [4]
     [])] []"))
Add slot 29
Contract instance for W[2]: (ReceiveEndpointCall (EndpointDescription {
    getEndpointDescription: "pay" }) (RawJson "{\"contents\":[{\"
    getEndpointDescription\":\"pay\"},{\"unEndpointValue\":{\"total\":{\"getValue\":[[{\"
    unCurrencySymbol\":\"\"},[[{\"unTokenName\":\"\"},60000]]]}}}],\"tag\":\"
    ExposeEndpointResp\"}"))
Validating transaction:
    f872b3694df96e1daab045b65fc508ba2edccd718fd4e1407db04b0b0352f109
Add slot 30
Contract instance for W[2]: (ContractLog (RawJson "Successful transaction to solo
    state PayState [3,2,1] [] : \nOpenMarketState [
    a2c20c77887ace1cd986193e4e75babd8993cfd56995cd5cfce609c2] [(80
    a4f45b56b88d1139da23bc4c3c75ec6d32943c087f250b86193ca7,PayState
    [3,2,1] []),(2e0ad60c3207248cecd47dbde3d752e0aad141d6b8f81ac2c6eca27c,
    PayState [4] [])] []"))
Add slot 31
```

# Appendix F

# Git proposals for this work's extensions

## F.1 Variable names

Created 5 months ago by **@amonteiro**

### Allow Variable Naming

### Abstract

Currently in SmartScribble it is only possible to declare in the protocol code the type of the variables and endpoints' parameters. This proposal consists in allowing the naming of SmartScribble's variables.

### Motivation

When writing the protocol, the programmer can specify the data types of the field variables and endpoint's parameters. But the generated code will automatically create the variables' names as param1, param2, param3, and so on depending on the number of variables declared.

Given a protocol for the Guessing game:

```
protocol GuessingGame (role Owner, role Player) {
  field ByteString;
  lock (String, Value) from Owner;
  choice at Owner {
    proceedWithGame: {
      guess (String) from Player;
    }
    cancelGame: {
    }
  }
  closeGame () from Owner;
}
```

There is a field with a bytestring variable in the protocol, since the programmer can't give a name the compiler names the variable as param1 in the generated code.

```
{-# INLINABLE transition #-}
transition :: State GuessingGameState -> GuessingGameInput -> Maybe (TxConstraints Void Void, State GuessingGameState)
transition State{stateData=oldData,stateValue} input = case (oldData, input) of
    (LockState _, ProceedWithGameInput param1 param2) -> Just (mempty, State{stateData = ProceedWithGameState param1, stateValue = p
    (LockState _, CancelGameInput param1 param2) -> Just (mempty, State{stateData = CancelGameState param1, stateValue = param2})
    (ProceedWithGameState _, GuessInput param1 param2) -> Just (mempty, State{stateData = GuessState param1, stateValue = param2})
    (CancelGameState _, CloseGameInput param1 param2) -> Just(mempty, State{stateData = CloseGameState param1, stateValue = mempty})
    (GuessState _, CloseGameInput param1 param2) -> Just(mempty, State{stateData = CloseGameState param1, stateValue = mempty})
    _ -> Nothing
```

The same scenario applies to the endpoint's parameters.

```
lockLogic :: String -> Value -> Contract () GuessingGameSchema T.Text (Either (BuiltinByteString, Value) GuessingGameError)
lockLogic param1 param2 = do
  -- TODO: Implement the logic here
  pure $ Right $ Error "undefined function"

guessLogic :: String -> Contract () GuessingGameSchema T.Text (Either (BuiltinByteString, Value) GuessingGameError)
guessLogic param1 = do
  -- TODO: Implement the logic here
  pure $ Right $ Error "undefined function"
```

If the field had 100 variables, they would be named param1, param2, param3, ..., param100. This situation can make it difficult to implement the business logic.

In another scenario, proposals to enhance SmartScribble that involve the protocol's variables can be difficult to implement since working with only the type of variables is not practical.

### Proposal

In this issue i propose to allow the naming of variables in the protocol. Since SmartScribble was adapted from Scribble, its syntax for variable naming can be adapted into SmartScribble. Declaring a variable would look like the following:

```
variableName:variableType
```

When the code is generated, instead of param1 the code uses the name given by the programmer.

## Example

The protocol for the Guessing game would look like this:

```
protocol GuessingGame (role Owner, role Player) {
  field Secret:ByteString;
  lock (Secret:String, Prize:Value) from Owner;
  choice at Owner {
    proceedWithGame: {
      guess (Guess:String) from Player;
    }
    cancelGame: {
    }
  }
  closeGame () from Owner;
}
```

## F.2 New triggers

Created 5 months ago by **@amonteiro**

### Implementing more complex Triggers

### Abstract

SmartScribble protocols can declare triggers, but the triggers can only address time or funds. In addition, triggers block the actions of a wallet until the trigger's condition is met. This issue proposes the implementation of more complex triggers that can address both time and funds, and address other factors in the protocol. And proposes a fix for the blocked actions due to an active trigger.

### Motivation

The programmer can register a trigger in an endpoint of the protocol, that will be later activated after a certain condition is met. But trigger's conditions can only address time or funds at the moment, they cannot address both factors at the same time and it is not possible to register more than one trigger.

```
protocol Auction (role Seller, role Buyer) {
  field Value,ByteString;
  initialise (Value, ByteString) from Seller{
      trigger slot closeAuctionTrigger;
  };
  do{
    rec Loop {
      bid (Value) from Buyer;
      Loop;
    }
  } interrupt {
    closeAuctionTrigger () from Contract;
  }
  collectFundsAndGiveRewards () from Seller;
}
```

The example above introduces an auction, when the Seller initialises the auction all the Buyers can bid the highest value for the asset on sale until a certain moment in time before the auction closes. The trigger condition is programmed after the Plutus code is generated alongside its logic when the trigger is activated. Now lets imagine that the Seller wants to meet a certain goal of funds as well. If the programmer wanted to program the trigger to activate if a certain time has passed or if the latest highest bid is equal or bigger to the Seller's goal, he couldn't program that cause triggers are limited to only one factor.

In another scenario, currently the endpoints that register a trigger block the actions of a wallet until the condition of the registered trigger is met.

```
protocol CounterTrigger (role Subject){
  field Integer, Integer;

  start() from Subject{
    trigger slot stop;
  };

  do{
    rec Loop {
      count() from Subject;
      Loop;
    }
  }
  interrupt{
    stop() from Contract;
  }
}
```

For example, the protocol above introduces a counter that involves only one role. The subject starts the counter and a trigger is registered, then the subject increments the counter with `count()` until the trigger condition is met. But with the current version of SmartScribble, when the subject calls the `start()` endpoint he can't interact with any other endpoints until the trigger condition is met.

### Proposal

This issue proposes the implementation of more complex triggers and a fix for the scenario where a trigger blocks the actions of a wallet.

## Syntax

To express the condition of the trigger, the programmer could specify the condition on the protocol instead of specifing it in the Plutus code. The condition would be injected into the generated smart contract and the programmer only needs to implement the logic of the trigger when it is activated.

```
trigger stop{
    time >= 10000 ||
    funds >= 2000000
}
```

The condition above is dependent on two factors connected by an OR operator ( `||` ), in alternative an AND operator ( `&&` ) could be used. In the trigger, `time` represents the current time of the contract and `funds` represent the funds that are currently stored in the contract. The current keywords that are used in the protocol to create the trigger, `slot` and `funds` , will be omitted from the syntax. The trigger above is set to activate when 10 seconds have passed (10000 milliseconds) or when the contract has 2 ADA in it (2 ADA = 2000000 Lovelace).

In alternative, instead of expressing the values in the units used by Plutus, milliseconds and Lovelace, the programmer can express the value in another unit with the respective unit next to the value. In the generated code, this units will be converted to the units used by Plutus. The syntax below demonstrates an example.

```
trigger stop{
    time >= 10 sec ||
    funds >= 2 ADA
}
```

In addition, besides reacting to time and money, triggers will react as well to the variables declared on the protocol's field. To achieve this, this issue will consider the proposal in issue#1.

```
field InfinityStones:Integer;

trigger snap{
    InfinityStones == 6 &&
    time >= 180000
}
```

The trigger above will activate when the variable `InfinityStones` is equal to 6 and when 3 minutes have passed (3 minutes = 180 seconds = 180000 milliseconds).

## Generated code

If we look at the Counter protocol mentioned in the Motivation section, the `start()` endpoint registers the trigger `stop` . The generated code for this endpoint looks like this:

```
start :: Promise () CounterOriginalSchema T.Text ()
start = endpoint @"start" $ \() -> do
    -- | Received the parameters
    currentState <- mapSMError' $ getCurrentStateSM client
    if isValidCallInState currentState (StartInput undefined undefined undefined) then do
    -- | Calling business logic function...
      logic <- startLogic
      case logic of
        Left (arg1, arg2, arg3) -> do
          case currentState of
            Nothing -> do
              res <- mapSMError' $ initialiseSM client (StartState arg1 arg2) arg3
              case res of
                Nothing -> do
                  logInfo @String "Initializing State Machine..."
                  condition <- startFundsLogic
                  outcome <- mapContractError' $ fundsAtAddressCondition condition contractAddress
                  triggerLogic <- stopLogic
                  case triggerLogic of
                    Left (argLogic1,argLogic2,argLogic3) -> void $ mapSMError' $ runContractStep client $ StopInput argLogic1 argLog
                    Right (Error x) -> logWarn @Text x
                _ -> logError @String "Operation failed because the SM was already initialised"
```

```
            _ -> do
              res <- mapSMError' $ runContractStep client (StartInput arg1 arg2 arg3)
              case res of
                Just s -> do
                  logInfo ("Successful transaction to state: " <> show s)
                  condition <- startFundsLogic
                  outcome <- mapContractError' $ fundsAtAddressCondition condition contractAddress
                  triggerLogic <- stopLogic
                  case triggerLogic of
                    Left (argLogic1,argLogic2,argLogic3) -> void $ mapSMError' $ runContractStep client $ StopInput argLogic1 argLog
                    Right (Error x) -> logWarn @Text x
                _ -> logError @String "Invalid operation in endpoint."
        Right (Error x) -> logWarn @Text x
    else
      logError @String "Invalid invocation of endpoint start"
```

What the code above is doing is managing the state machine with the transitions that belong to the `start()` endpoint and the `stop` trigger. Since `start()` is the first interaction of the protocol, the code above will first initialize the state machine, then register the trigger. The endpoint waits for the trigger's condition, and when that condition is met the code will execute the transition that belongs to the trigger.

The code that is meant to be written for the condition are functions from the Plutus API that wait for a certain condition to be met, and this conditions only address one factor. To address more than one factor the programmer has to call one of the functions from the API, wait for its condition, then call another function from the API, and wait for its condition as well. The trigger would address these factors one by one, and this would only give us an AND effect, and there are no functions to address the field's variables.

To solve this problem, this issue will adapt how the functions from Plutus are implemented. For instance, the code below is how the function `fundsAtAddressGeq` is implemented in Plutus.

```
fundsAtAddressCondition
    :: forall w s e.
       ( AsContractError e
       )
    => (Value -> Bool)
    -> Address
    -> Contract w s e (Map TxOutRef ChainIndexTxOut)
fundsAtAddressCondition condition addr = loopM go () where
    go () = do
        cur <- utxosAt addr
        let presentVal = foldMap (view ciTxOutValue) cur
        if condition presentVal
            then pure (Right cur)
            else awaitUtxoProduced addr >> pure (Left ())

-- | Watch an address for changes, and return the outputs
--   at that address when the total value at the address
--   has reached or surpassed the given value.
fundsAtAddressGeq
    :: forall w s e.
       ( AsContractError e
       )
    => Address
    -> Value
    -> Contract w s e (Map TxOutRef ChainIndexTxOut)
fundsAtAddressGeq addr vl =
    fundsAtAddressCondition (\presentVal -> presentVal `V.geq` vl) addr
```

What the function does is check if the contract funds are greater than or equal to a given value each time new Utxos are produced for the contract.

```
trigger stop{
    time >= 10000 ||
    funds >= 2000000
}
```

Given the trigger above, based on how Plutus implements its functions to wait for a condition this issue will integrate the function below in the generated contract.

```
triggerCondition :: Contract () Schema Text (Map.Map TxOutRef ChainIndexTxOut)
triggerCondition = loopM go () where
    go () = do
        cur <- utxosAt contractAddress
        time <- currentTime
        timestamp <- getTriggerTimestamp
        let funds = foldMap (\(_,a) -> view ciTxOutValue a) $ Map.toList cur
        if time >= timestamp + 10000 || (funds `V.geq` Ada.lovelaceValueOf 2000000)
            then pure (Right cur)
            else awaitUtxoProduced contractAddress >> pure (Left ())
```

And the old block of code that treats the trigger in the endpoint will change from this:

```
condition <- startFundsLogic
outcome <- mapContractError' $ fundsAtAddressCondition condition contractAddress
triggerLogic <- stopLogic
```

To this:

```
outcome <- triggerCondition
triggerLogic <- stopLogic
```

And the endpoints that register a trigger will register the timestamp they were called soo that the time factor can be implemented alongside the other factors.

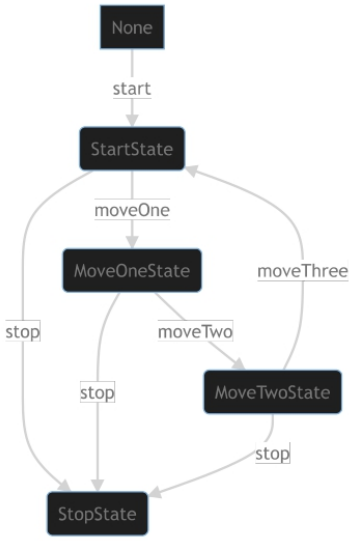
### Fix for the blocked wallets

As mentioned in this issue, the wallets that call a endpoint that registers a trigger are blocked until the condition is met. The fix this issue proposes, is to check the condition each time an endpoint, that is going to be interrupted by the respective trigger, is called.

Lets take a look at the following toy protocol below

```
protocol TriggerEx (role Subject){
  start() from Subject{
    trigger slot stop;
  };

  do{
    rec Loop {
      moveOne() from Subject;
      moveTwo() from Subject;
      moveThree() from Subject;
      Loop;
    }
  }
  interrupt{
    stop() from Contract;
  }
}
```

The state machine that is created is the following:




The effect the triggers create is a transition from the states where the endpoints that are going to be interrupted are called.

In Plutus, a wallet can't call an endpoint while another is still working. To address this problem, instead of waiting for the condition in the `start()` endpoint, this issue will make the `moveOne()`, `moveTwo()` and `moveThree()` endpoints check the trigger's condition before proceeding with the code they are meant to treat. In each endpoint that is going to be interrupted, the following block of code will be added to the beggining of each endpoint in the Plutus code.

```
trigger <- stopCondition
currentState <- if trigger && isValidCallInState currentState (StopInput undefined undefined undefined)
    then mapSMError' $ runContractStep client (StopInput counter resets mempty)
    else pure $ currentState
```

`stopCondition` is the function that will check the trigger's condition.

```
stopCondition  :: Contract () CounterVerifierSchema T.Text (Bool)
stopCondition = do
  time <- currentTime
  timestamp <- getTriggerTimestamp
  pure $ time >= timestamp + 10000
```

As for the `start()` endpoint, this issue will not make that endpoint wait but will make it save the timestamp when `start()` is called, just like the previous solution mentioned.

The solution in this section is intended to address the problem of blocked wallets due to active triggers, and in comparison with the solution in the section above, it consumes slightly more CPU and memory. And applying this solution will discard the other solution where this issue implements waiting triggers based on the implementations made in Plutus.

## Examples

### Auction

The seller wants to put an asset for auction. The auction will close automatically when the latest highest bid is equal or higher than the seller's goal or when 10 seconds have passed.

```
protocol Auction (role Seller, role Buyer) {
  field Value,ByteString;
  initialise (Value, ByteString) from Seller{
      trigger closeAuctionTrigger{
        funds >= 2000000 ||
        time >= 10000
      };
  };
```

```
  do{
    rec Loop {
      bid (Value) from Buyer;
      Loop;
    }
  } interrupt {
    closeAuctionTrigger () from Contract;
  }
  collectFundsAndGiveRewards () from Seller;
}
```

### Counter

A subject starts a counter that increments its value. The counter will reset to 0 if the value incremented from 3 to 4 and increments the number of resets that were made. The counter will stop when the counter value reaches 3 and the number of resets reaches 5.

```
protocol CounterTrigger (role Subject){
  field counter:Integer, resets:Integer;

  start() from Subject{
    trigger stop{
        counter >= 3 &&
        resets >= 5
    };
  };

  do{
    rec Loop {
      count() from Subject;
      Loop;
    }
  }
  interrupt{
    stop() from Contract;
  }
}
```

## F.3 Role validation

Created 4 months ago by **@amonteiro**

### Role Validation

### Abstract

Roles in SmartScribble describe who calls a certain endpoint in the protocol, however they currently have no effect in the generated code. This issue proposes the validation of users roles in order to give a purpose to SmartScribble's roles.

### Motivation

The roles of a protocol determines who is participating and who calls which interaction. But SmartScribble's roles have currently no effect in the generated Plutus smart contract, and this can create a security problem.

Lets take a look at the guessing game:

```
protocol GuessingGame (role Owner, role Player) {
  field ByteString;
  lock (String, Value) from Owner;
  choice at Owner {
    proceedWithGame: {
      guess (String) from Player;
    }
    cancelGame: {
    }
  }
  closeGame () from Owner;
}
```

At the end of the protocol, the Owner calls the `closeGame()` endpoint, and this endpoint could be programmed to return the funds stored as a prize if no Player managed to guess the secret word. But since the generated contracts do not check if it is the Owner calling the endpoint, a malicious player can call that endpoint to steal the funds.

In another example, lets look at a protocol for an auction:

```
protocol Auction (role Seller, role Buyer) {
  field minBid:Value,tokenInAuction:ByteString;
  initialise (initBid:Value, token:ByteString) from Seller{
      trigger slot closeAuctionTrigger;
  };
  do{
    rec Loop {
      bid (newBid:Value) from Buyer;
      Loop;
    }
  } interrupt {
    closeAuctionTrigger () from Contract;
  }
  collectFundsAndGiveRewards () from Seller;
}
```

The Seller of the auction collects the funds of the highest bidder at the end of the protocol. But since the generated contracts don't verify who is collecting the funds, one of the Buyers or someone else can steal the funds that should belong to the Seller.

### Proposal

The proposal in this issue consists of a solution to ensure that endpoints are called by the right role. As we saw in the protocols illustrated above, roles can one or more associated users.

#### Syntax

In the guessing game we know there is only one Owner who starts the game by picking a secret word and depositing the prize. But in the case of the Player role, anyone can be a player. The same situation applies to the auction, where we know that there is just one seller and possibly many buyers.

This proposal will keep the `role` keyword to identify the roles that can be taken by more than one user. To identify the roles that can be taken by only one user, in the case of the Owner and Seller role, we propose a new keyword: `signedRole`.

Just like the `role` keyword:

```
role Player
```

`signedRole` keywords will have the same syntax:

```
signedRole Owner
```

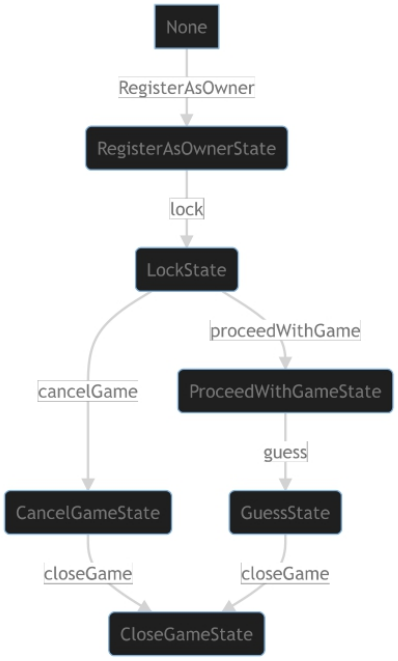
These will be the roles that will be validated in the smart contract.

## `signedRole` Registration

To validate the role we need an identifier. In Solidity, contracts use addresses to identify the users. In Plutus, contracts use the hash of the public key of the users' wallets.

So, `signedRole`s must provide the participant's public key to the contract before proceeding to the protocol. SmartScribble will generate an endpoint for each `signedRole`.

These endpoints will enable users to commit their public key to the signed roles, and these endpoints will be the first transitions in the state machine before proceeding to the transitions that belong to the specified protocol. For the guessing game protocol shown in the Motivation section, if the Owner is a `signedRole`, the state machine will be the following:




## `signedRole` Validation

After each `signedRole` gets a public key associated, it is now possible to identify and verify if an endpoint is being called by the right role.

Fortunately, Plutus provides functions for this task. SmartScribble creates a state machine with the help of an API from Plutus, each transition will submit a transaction. The `transition` function of the state machine returns the next state, with the data and the value to create a new UTxO, and the constraints for the transaction.

The contracts generated currently return `mempty` in their `transition` function, which means its state machines will submit transactions with no constraints.

```
{-# INLINABLE transition #-}
transition :: State CounterTriggerState -> CounterTriggerInput -> Maybe (TxConstraints Void Void, State CounterTriggerState)
transition State{stateData=oldData,stateValue} input = case (oldData, input) of
    (StartState _ _, CountInput id counter stateValue) -> Just (Constraints.mustBeSignedBy id, State{stateData = StartState id count
    (StartState _ _ _, StopInput id counter stateValue) -> Just(mempty, State{stateData = StopState id counter, stateValue = mempty}
    _ -> Nothing
```

One of the constraints that Plutus provides is mustBeSignedBy. This constraint implies that a transaction must be signed by a certain public key. And consequently, this constraint will force an endpoint to be called only by the wallet whose key is given to the constraint.

For every endpoint that is being called by a `signedRole`, their respective transition will use the `mustBeSignedBy` constraint, with the respective public key in the function's parameter. And endpoints called by a `role`, will keep `mempty` since `role` can represent anyone.

(The code above is just an example using the `mustBeSignedBy` function, public keys will be stored in the state and not passed by the input.)

# Example

## Guessing game

There is only one Owner, and anyone can be a player:

```
protocol GuessingGame (signedRole Owner, role Player) {
  field ByteString;
  lock (String, Value) from Owner;
  choice at Owner {
    proceedWithGame: {
      guess (String) from Player;
    }
    cancelGame: {
    }
  }
  closeGame () from Owner;
}
```

## Auction

Anyone can be a Buyer but there is only one Seller:

```
protocol Auction (signedRole Seller, role Buyer) {
  field minBid:Value,tokenInAuction:ByteString;
  initialise (initBid:Value, token:ByteString) from Seller{
      trigger slot closeAuctionTrigger;
  };
  do{
    rec Loop {
      bid (newBid:Value) from Buyer;
      Loop;
    }
  } interrupt {
    closeAuctionTrigger () from Contract;
  }
  collectFundsAndGiveRewards () from Seller;
}
```

## F.4 Individual state machines

Created 4 months ago by **@amonteiro**

# Implementation of individual state machines

## Abstract

SmartScribble creates a state machine to ensure the correct order of interactions in a smart contract, but this state machine is global and, consequently, all users share the same state. This issue proposes the implementation of individual state machines where each user can be in its own state regardless of the other users' and the contract's current state.

## Motivation

The state machine that is being currently being created only addresses global states. Regardless of the contract's current state, all users will be in the same state as the contract. But in some scenarios, users can have their own state independently of the other users.

Lets take a supermarket as an example. A client starts by entering the supermarket and picking up a shopping Basket. Next, he will browse the supermarket a pick up the items he wishes to purchase. Then, he heads to the counter to make the payment of the items he picked and finally leaves the supermarket.

In the supermarket scenario, some clients are entering the supermarket, others are browsing for items and others are paying the total of what they picked. All of this is happening in the same moment. In a state machine, their states are individual and they don't share it with other clients soo that they can be in their own state.

## Proposal

This issue will propose the implementation of individual state machines, where users can have their own state regardless of the other users. For the sake of this issue, the state machine that is currently created by SmartScribble will be addressed as the Global state machine (GlobalSM) and the the state machine for individual clients will be addressed as the Individual state machine (IndividualSM).

### Plutus Code

The state machine that is integrated in a smart contract is created with the aid of an API provided by Plutus. This API creates a GlobalSM, but there is no way to create an IndividualSM with the aid of a specific API at the moment.

To implement an individualSM in the Plutus code, the solution in this proposal will make use of the GlobalSM to store information and operate over the individual states.

#### Storing information

Given this protocol:

```
protocol Example (role S1){
  step1() from S1;
  step2() from S1;
  step3() from S1;
}
```

SmartScribble creates in the contract a data type for the states and inputs of the GlobalSM.

```
-- | Declaration of the possible states for the State Machine:
data ExampleState =
    Step1State [(Integer,(POSIXTime,Slot))]
    | Step2State [(Integer,(POSIXTime,Slot))]
    | Step3State [(Integer,(POSIXTime,Slot))]
    deriving stock (Show, Generic)

-- | Declaration of the inputs that will be used for the transitions of the State Machine
data ExampleInput =
    Step1Input [(Integer,(POSIXTime,Slot))] Value
    | Step2Input [(Integer,(POSIXTime,Slot))] Value
    | Step3Input [(Integer,(POSIXTime,Slot))] Value
    deriving stock (Show, Generic)
```

(The `[(Integer,(POSIXTime,Slot))]` is a list of timestamps to be used by the complex triggers, it is not relevant for this proposal.)

Lets say we want to implement an IndividualSM with this interactions:

```
sub1() from someone
sub2() from someone
sub3() from someone
```

SmartScribble will create, specificaly for this interactions, data types for the individual states and inputs.

```
data SoloState = None | Sub1State | Sub2State | Sub3State
    deriving (Show)

PlutusTx.unstableMakeIsData ''SoloState
PlutusTx.makeLift ''SoloState

data SoloInput = Sub1 | Sub2 | Sub3
    deriving (Show)

PlutusTx.unstableMakeIsData ''SoloInput
PlutusTx.makeLift ''SoloInput
```

To store the information about the state of each user, a list will be added to the fields of the Global states. That list will contain tuples with a Public key of the user and its current state. Public keys will be used to identify and track the information of one user. The states for the GlobalSM, with the IndividualSM integrated, will look like this:

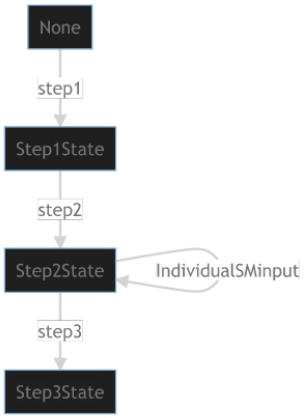
```
type EntityList = [(PaymentPubKeyHash,SoloState)]

-- | Declaration of the possible states for the State Machine:
data ExampleState =
    Step1State EntityList [(Integer,(POSIXTime,Slot))]
    | Step2State EntityList [(Integer,(POSIXTime,Slot))]
    | Step3State EntityList [(Integer,(POSIXTime,Slot))]
    deriving stock (Show, Generic)

-- | Declaration of the inputs that will be used for the transitions of the State Machine
data ExampleInput =
    Step1Input [(Integer,(POSIXTime,Slot))] Value
    | Step2Input [(Integer,(POSIXTime,Slot))] Value
    | Step3Input [(Integer,(POSIXTime,Slot))] Value
    deriving stock (Show, Generic)
```

### Operating over the individual state machines

In order to operate over the IndividualSM, a specific transition function will be created:

```
{-# INLINABLE soloTransition #-}
soloTransition :: SoloState -> SoloInput -> Maybe SoloState
soloTransition state input = case (state,input) of
    (None,Sub1) -> Just Sub1State
    (Sub1State,Sub2) -> Just Sub2State
    (Sub2State,Sub3) -> Just Sub3State
    _ -> Nothing
```

For this transitions to take effect, the transition function for the GlobalSM will be used to aid the IndividualSM. Since we do not want to change the global state, a specific global transition will be added to the GlobalSM that goes from a global state to that same global state.




All that this new global transition will mainly change is the information about the current individual states. The off-chain code will perform the individual transition. Just like how the GlobalSM creates endpoints for each interaction, the IndividualSM will do the same, alongside a logic function:

```
sub2 :: Promise () SingSMversion2Schema T.Text ()
sub2 = endpoint @"sub2" $ \() -> do
    -- | Received the parameters
    pkh <- ownPaymentPubKeyHash
    currentState <- mapSMError' $ getCurrentStateSM client
    if isValidCallInState currentState (IndividualSMInput undefined undefined undefined undefined) then do
    -- | Calling business logic function...
      logic <- sub2Logic
      ents <- getEntitys
      case logic of
        Left (stateVal) -> case ents of
            Just entities -> case (lookup pkh entities) of
                Just currstate -> case (soloTransition currstate PickTwo) of
                    Just newstate -> do
                        let newlist = updateEntity entities pkh newstate
                        res <- mapSMError' $ runContractStep client (IndividualSMInput currstate Sub2 newlist stateVal)
                        case res of
                            Just s -> logInfo ("Successful transaction to state: " <> show s)
                            _ -> logError @String "Invalid operation in endpoint."
                    Nothing -> logError @String "Invalid operation in endpoint."
                Nothing -> logError @String "Entity not registered."
            _ -> logError @String "List of entities not found."
        Right (Error x) -> logWarn @Text x
    else
      logError @String "Invalid invocation of endpoint sub2"
```

And the on-chain code will be responsible for verifying if the individual transition is valid:

```
{-# INLINABLE transition #-}
transition :: State ExampleState -> ExampleInput -> Maybe (TxConstraints Void Void, State ExampleState)
transition State{stateData=oldData,stateValue} input = case (oldData, input) of
    (Step1State oldentities _, Step2Input triggerTimeStamps stateVal) -> Just(mempty, State{stateData = Step2State oldentities trigg
    (Step2State oldentities _, Step3Input triggerTimeStamps stateVal) -> Just(mempty, State{stateData = Step3State oldentities trigg

    (Step2State oldentities _, IndividualSMInput s i entities triggerTimeStamps stateVal) ->
        case (soloTransition s i) of
            Nothing -> Nothing
            _ -> Just(mempty, State{stateData = Step2State entities triggerTimeStamps, stateValue = stateVal})

    _ -> Nothing
```

## Syntax

To write a protocol that will address individual clients, the global protocol will be required. The Global protocol will define when the Individual protocol can operate. The definition of the interactions of the Individual protocol will use the same syntax as the Global protocol.

```
protocol Example (role S1, role S2){
  step1() from S1;
  step2() from S1{
      letProtOperate IndExample;
  };
  step3() from S1;
}

individual protocol IndExample (role S2){
    sub1() from S2;
    sub2() from S2;
    sub3() from S3;
}
```

The example above defines a global and an indivual protocol. The global protocol is defined as it always was, and the individual protocol is defined in the same way as the global with the addition of the `individual` keyword in the protocol's signature.

The individual protocol is used by clients with the role `S2`, and they can only operate when `S1` calls the `step2()` endpoint. When the next endpoint is called, in this case `step3()`, the individual protocol cannot perform individual transitions unless it is specified in the `step3()` interaction that the individual protocol can operate, like how is specified in the `step2()` interaction.

In some scenarios, only certain transitions are allowed in a certain current global state. We can take IVAucher as an example, there is a period where consumers can register in the IVAucher program, after that period, no one else can register. Then in another period, which would be another global state, registered users can start accumulating benefits, but they won't accumulate more once that period is over. In other words, only certain actions are allowed in certain global states.

Lets say that in the example protocol above, calling `step1()` will allow only `sub1()` and `sub2()` to be called, and calling `step2()` allows only `sub3()` to be called. The programmer will be able to fragment the individual protocol like this:

```
protocol Example (role S1, role S2){
  step1() from S1{
      letProtOperate IndExampleFrag1;
  };
  step2() from S1{
      letProtOperate IndExampleFrag2;
  };
  step3() from S1;
}

individual protocol IndExampleFrag1 (role S2){
    sub1() from S2;
    sub2() from S2;
    jumpTo fragment2;
}

individual protocol IndExampleFrag2 (role S2){
    checkpoint fragment2
    sub3() from S3;
}
```

This will allow the programmer to determine when can a individualSM operate and what can it do in a certain global state.

Keywords `checkpoint` and `jumpTo` will link the many `individual protocol` to form the only IndividualSM that will exist in the smart contract. Which means that this:

```
individual protocol IndExampleFrag1 (role S2){
    sub1() from S2;
    sub2() from S2;
    jumpTo fragment2;
}

individual protocol IndExampleFrag2 (role S2){
    checkpoint fragment2
```

```
    sub3() from S2;
}
```

Is equivalent to this:

```
individual protocol IndExample (role S2){
    sub1() from S2;
    sub2() from S2;
    sub3() from S2;
}
```

`checkpoint` and `jumpTo` don't have to necessarily be used to connect the end of a fragment to the beginning of another fragment. For example, this can be specified:

```
individual protocol MainFrag (role S2){
    start() from S2;

    checkpoint mainfrag;

    choice at S2{
        proceed:{
            jumpto alterfrag;
        }
        moveAlong:
    }

    end() from S2;
}

individual protocol AlterFrag (role S2){
    checkpoint alterfrag;

    sub1() from S2;
    sub2() from S2;
    sub3() from S2;

    jumpTo mainfrag;
}
```

The fragments above are the equivalent to this individual protocol:

```
individual protocol MergedFrags (role S2){
    start() from S2;

    rec FragLoop{
        choice at S2{
            proceed:{
                sub1() from S2;
                sub2() from S2;
                sub3() from S2;
                FragLoop;
            }
            moveAlong:
        }
    }

    end() from S2;
}
```

## Examples

### Mini Amazon

```
protocol MiniAmazon (signed role Owner, role Buyer){
    openAmazon() from Owner{
      letProtOperate PurchaseProcess;
    };
    closeAmazon() from Owner;
}

individual protocol PurchaseProcess (role Buyer){
    startBrowsing() from Buyer;

    rec BrowseLoop{
        choice at Buyer{
            Add:{
                addToCart(item:Integer) from Buyer;
            }
            Remove:{
                RemoveFromCart(item:Integer) from Buyer;
            }
        }
        choice at Buyer{
            ContinueBrowsing:{
                BrowseLoop;
            }
            FinishBrowsing:{
                makePayment(total:Value) from Buyer;
            }
        }
    }
}
```

### A Race

```
protocol Race (signed role RaceWatcher, role Runner){
    openSignUps() from RaceWatcher{
      letProtOperate SignUpPhase;
    };
    startRace() fromRaceWatcher{
      letProtOperate RacePhase;
    };
    finishRace() fromRaceWatcher;
}

individual protocol SignUpPhase (role Runner){
    signUp() from Runner;
    jumpTo racephase;
}

individual protocol RacePhase (role Runner){
    checkpoint racephase;
    startRacing() from Runner;

    rec RaceLoop{
        choice at Runner{
            race:{
                RaceLoop;
            }
            endRace:
        }
    }
}
```