Lehrstuhl Angewandte Informatik IV
Datenbanken und Informationssysteme
Prof. Dr.-Ing. Stefan Jablonski

Institut für Angewandte Informatik
Fakultät für Mathematik, Physik und Informatik
Universität Bayreuth

Project Report

Philipp Scholz, Anatoly Obukhov

August 6, 2021

Version: Final

Universität Bayreuth

Fakultät Mathematik, Physik, Informatik
Institut für Informatik
Lehrstuhl für Angewandte Informatik IV

Blockchain-based Process Execution with Chrysalis

Project Report

Philipp Scholz, Anatoly Obukhov

1. Reviewer Prof. Dr.-Ing. Stefan Jablonski
Fakultät Mathematik, Physik, Informatik
Universität Bayreuth

2. Reviewer Dr. Lars Ackermann
Fakultät Mathematik, Physik, Informatik
Universität Bayreuth

Supervisors Christian Sturm and Lars Ackermann

August 6, 2021

Philipp Scholz, Anatoly Obukhov

Project Report
Blockchain-based Process Execution with Chrysalis, August 6, 2021
Reviewers: Prof. Dr.-Ing. Stefan Jablonski and Dr. Lars Ackermann
Supervisors: Christian Sturm and Lars Ackermann

Universität Bayreuth
Lehrstuhl für Angewandte Informatik IV
Institut für Informatik
Fakultät Mathematik, Physik, Informatik
Universitätsstrasse 30
95447 Bayreuth
Germany

Abstract

TODO Abstract

Contents

1 Introduction

TODO BPM more prevalent, interorg bpm, bla bla.

1.1 Business Processes on Blockchain

TODO classic BPM

TODO how blockchain solves that

TODO hint at christian's paper

1.2 Problem Statement

TODO prototype was handed to us -> 'Chrysalis'

TODO Prototype intention: "proof of concept"

TODO our task: refine that system, show that it can be made compatible with modern, modular architecture and demands

1.3 Results

1.4 Thesis Structure

Chapter 2

In the first content chapter, we will explain the way Chrysalis generally functions. Starting from an abstract and high-level view where the intended uses of the application are explained, we will the continue to detail the components that make up Chrysalis - what their role in the system is, how they interact with each other and with what external components they interface. At last, we will describe how business processes are modeled inside a blockchain node, so that our application may interact with them in a defined way.

Chapter ??

This chapter being the biggest of all, we intend to describe our programming work done here. This includes all improvements, additions and removals in the code. To do this, for every major task we will first describe its meaning and implications, then give a modeler's overview of the changes done. Where needed, we will give some technical insights to our work. Concluding every task as well as the entire chapter, we will summarize our results.

Chapter 4

Given that this project wasn't intended to be perfected once our work was done, and also given that some problems arose that hampered the quality of our results, this chapter will describe said issues. We will both clarify where they stem from and propose some ways of solving them, so those who will be handed this project may fix them with relative ease.

Chapter 5

TODO

Chapter 6

Due to the project having grown in complexity during our work on it, we decided to build a repository of design documents and other helpful files. In this short section, we intend to present these resources.

Chapter 7

Finally, we will evaluate the success of our project and speak about the opportunities and limits it offers to those interested in deploying a decentralized process management engine.

Architecture of Chrysalis

2

Since *Chrysalis* was handed to us as an already running prototype including a predefined layer structure and a way to interact with it in a browser, those designs were therefore a given. The Application is written entirely in JavaScript code, with the code deployed to the Blockchains being an exception sometimes. The prototype version was able to run on an *Ethereum*-Blockchain-Network, demonstrating it's functionality. In this chapter, we elaborate on the general structure of *Chrysalis*, omitting technical details and focusing on how the user and the components interact.

2.1 Intended Usage

From the user's point of view, three interactions with the system are generally intended, as they're described below. The configuration aside, the other two, encapsulating BPM, are shown in figure 2.1.

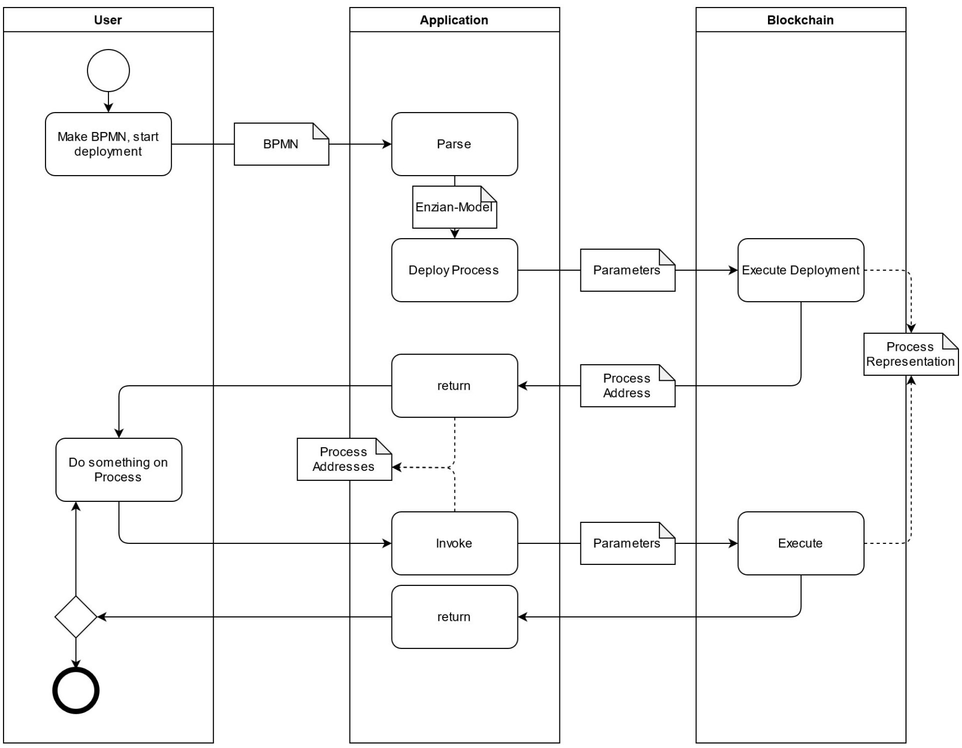

Fig. 2.1: Layer structure of the original *Chrysalis* System.

Configuration

Before BPM actions are possible, some parameters must be set. On one hand, the endpoint URL of the local Blockchain application (the Blockchain *node*) must be declared. Additionally, the authentication of the user in front of the node is needed, usually a private key or cryptographic token of the likes. In the case of the original *Chrysalis* application, the browser plugin *Metamask* could also be used to provide these credentials, with the app automatically detecting the plugin and using it's connections.

Process Deployment

For instantiating a process, first and foremost the user needs to provide the underlying process model. This is done by handing over a file written in BPMN format (the XML extension) containing the model. Additionally, the user selects where to deploy said process - usually the previously configured private network - and, in the original application, via which interface the Blockchain application shall be contacted. Then, the user simply hits the 'Deploy' button and the application handles the rest, parsing and deploying the provided model to the provided target, returning and saving the address where the process instance is situated on the blockchain data structure.

Process Execution

Given a deployed process's address, the user can then switch to the execution tab, where they may select the process and specify the task they wish to be executed. All other details of the execution are handled in the background. After every task execution and at the beginning, the user is also shown the current event log of the selected process instance, giving feedback towards the current process state. This is currently the only way of telling whether a process might be executable.

2.2 Components and their Interactions

The name-giving component, *Chrysalis*, is a front-end application written in *React* (for disambiguation, we will refer to the component as 'Chrysalis' and the entire system as 'project' or 'application'). While it serves as a window for the user to interact with, it also handles the storage of any relevant data, like the addresses of deployed processes and authentication credentials of the user. *Chrysalis*, like any React-app, deploys a packaged version of itself and it's dependencies into the user's browser, to be executed there. As for business process interactions, it instantiates an *Enzian-Yellow* object with the fitting configuration and delegates all commands to it.

Enzian-Yellow, being an installed dependency of *Chrysalis*, does the abstraction work between BPM actions and Blockchain operations. When instantiated and

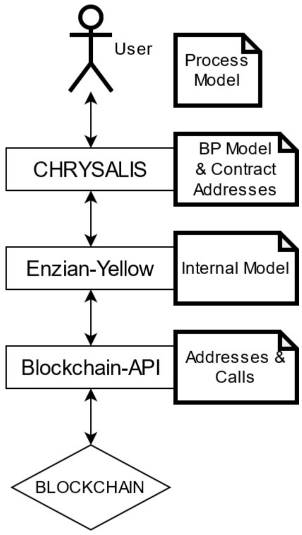

Fig. 2.2: Layer structure of the original *Chrysalis* System.

configured, it will in turn instantiate and configure a Blockchain-API to connect to the corresponding network node. *Enzian-Yellow* offers methods like creating and deploying processes and executing tasks, which are internally translated into the corresponding API invocations, and vice versa for the node's responses.

The *Blockchain-API* is the gateway from a local program to interact with a specified Blockchain network node. When instantiated, it establishes a connection to the specified node, handles authorization work and abstracts the networking away, so that the Blockchain's data and functions may be used as if they were local to the code.

The *Blockchain Network Node*, depicted in figure 2.2 as 'Blockchain', usually acts like a server that has a local copy of the Blockchain. It is fully synchronous with the network and any operation on the chain (i.e., addition of blocks) will be mirrored on every node in the network. In our case this means that every BPM action will be synchronized for every network participant that way.

2.3 Modeling of Processes

To store and interact with data on the blockchain in a defined way, the Blockchains usually offer a specific gateway to guard the chain's state from abuse: *Smart Contracts*. A Smart Contract is simply an executable program or routine written onto the Blockchain itself, so that every peer of the network may see it's definitions. This has multiple advantages in the *Chrysalis* application's use case:

- *Transparent logic:* Since the process logic is defined in the smart contract, every peer has a transparent definition of how a process may be changed - in extension, we define that a process may never be changed outside contract definitions.

- *Security:* Since a contract is the only way to interact with the process, every peer may check on the validity of a proposed contract invocation. As peers have to agree on Blockchain interactions before they are written, the trust issue mentioned in TODO is solved.

- *Data handling:* The smart contract either internally memorizes the location of deployed process models ir translates external keys into their location. Either way, the contract abstracts memory addresses inside the Blockchain away for the user and the developer.

3 Improvements

This chapter is split into the 4 programming tasks we were given to improve the *Chrysalis* system and is therefore the place where we present the work we've done.

- **Chapter 3.1:** Restructuring the System to make it easier to maintain and expand.
- **Chapter 3.2:** Adding a persistence layer that stores information outside the browser and the Blockchain.
- **Chapter 3.3:** Creating an expansion to the system by implementing BPM on a *Hyperledger* Blockchain.
- **Chapter 3.4:** Improving the Smart Contract code that the *Ethereum* application deploys and invokes to make it more efficient and easier to maintain.

3.1 Restructuring the Code

With the *Chrysalis* system only being a prototype when it was handed to us, code quality had not been a focus so far. To mend this, we were tasked to make way for future expansions and changes to the codebase by generally improving it's architecture.
Since the front-end component *Chrysalis* looked fairly well-built and flexible, we focused on the underlying *Enzian-Yellow*, updating Chrysalis only accordingly when a signature change elsewhere made it necessary.

As seen in figure 3.1, the architecture of *Enzian-Yellow* consists of three parts, giving us the following understanding:

- On the top, the name-giving class *EnzianYellow* is the interface to the applications that use it (like *Chrysalis*): It offers BPM-related methods like creating new processes or tasks and executing the latter, independent of the underlying Blockchain specifics. When constructed, it also instantiates *BasicEnzianYellow* and *Web3Wrapper*.

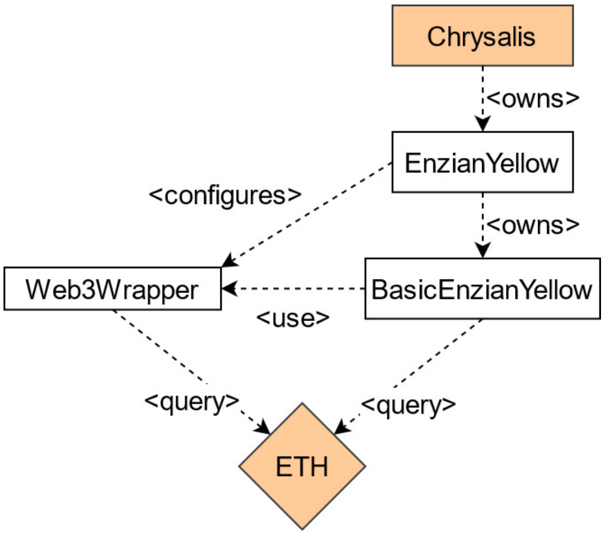

Fig. 3.1: Component structure of the original *Enzian-Yellow* Repository. The components marked in orange are not part of Enzian-Yellow, but interact with it.

- *BasicEnzianYellow*, as the name might suggest, is a specialization of *EnzianYellow* (though not inheriting from it), offering the same kind of method signature as it's 'parent', but translating these BPM actions into the corresponding Blockchain-transactions. To make these transactions, it sometimes calls the *Web3Wrapper* and sometimes directly queries the *Web3* instance within.

- *Web3Wrapper* wraps, as the name indicates, an instance of the class *Web3*, which is the API used to connect and interact with a local *Ethereum* node. It offers functions like deploying contracts, however doesn't fully implement all BPM functionality that *EnzianYellow* offers.

Tasks

From the given structure, as well as some other observations, the following issues were selected to be solved in this restructuring:

- The **Dependencies** in the code does not follow a good and strict hierarchy in the sense of a layer-based architecture, with each layer offering abstraction to and fully satisfying the functional demands of the layer above (Section 3.1.1).

- In many places, multiple methods with different parameters have the same effect, allowing for **code duplicates**. Additionally, **construction** does sometimes not imply object initialization, leaving room for hard-to-explain errors and 'zombie states' (Section 3.1.2).

- It is not immediately clear, where an **Expansion**, like adding another Blockchain protocol, would be tied into (Section 3.1.3).

- The overall **Transparency**, especially via log outputs, should be improved (Section 3.1.4).

3.1.1 Improving the Dependency Hierarchy

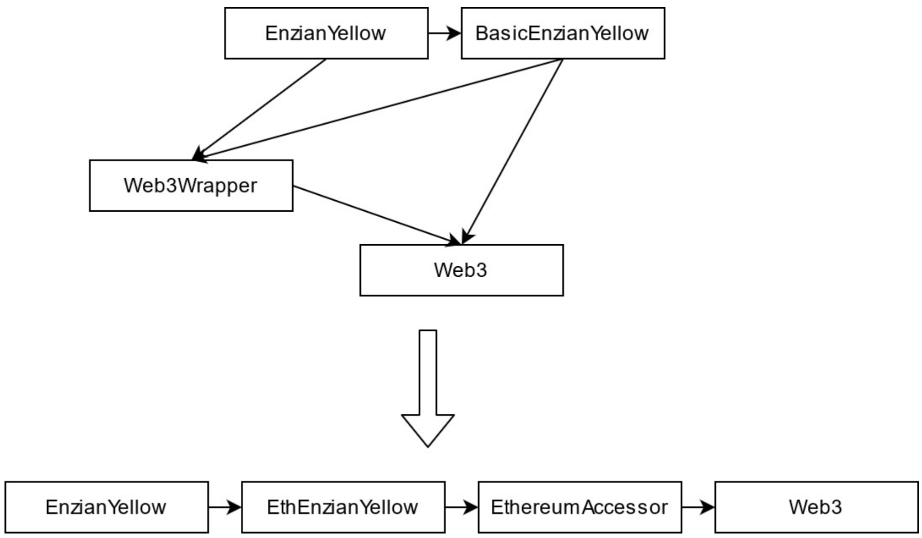

Fig. 3.2: The dependency structure of the *Enzian-Yellow* module before and after being reworked, sketched.

In a fully layered architecture, as we meant to achieve, one layer should never interact with another layer except that above and below. In *Enzian-Yellow* that was not entirely the case. For example, the *Web3Wrapper* object would do interactions with the Blockchain using the contained *Web3* interface, yet the 'parent' object, *BasicEnzianYellow*, would also directly access *Web3*, essentially skipping the wrapper entirely. While this pattern is functional, it is hard to maintain when changes are made to a component, as abstractions of layers are not strict and changes therefore may affect more layers. To achieve a more strict layer-architecture, the following was done:

- To fulfill every component's requirements towards lower layers, the first layer underneath was extended to offer all necessary methods, removing the need for layer-skipping.

- Code was moved to the specific place where it was meant to be, e.g. actions that required using *Web3* would only be allowed inside the *Web3Wrapper*.

- With delegations for every possible method in place, the dependency structure could now be flattened, as shown in figure 3.2.

- Finally, in the new structure, a few components were renamed to better reflect their functionality - especially with future expansions in mind. *BasicEnzianYellow* became *EthereumEnzianYellow* and the *Web3Wrapper* became the *EthereumAccessor*.

3.1.2 Method and Object Signatures

In the prototype system, code was significantly scattered and duplicated, offering the same functionality multiple times with differing parameters, like the example shown in figure 3.3. As this affects maintainability negatively, we sought to eliminate such redundancies wherever possible.

```
async executeTask(contractInstance, task, account) {...}

async executeTaskByAddress(contractAddress, task) {...}

async executeTaskByAddressSelfSigned(contractAddress, task, privateKey) {...}
```

Fig. 3.3: The 'execute task' method signatures in the original system.

Reducing the code scatter boiled down to finding the minimum information with which a method could be executed. In our example case, the *contract instance* argument could be created from the *contract address* - in fact, the instance was always internally created from the address. Additionally, for abstraction purposes, it should not be necessary for the above layers to use such contract instances. Similarly, The *account* and *private key* arguments are equivalent; we opted for only using private keys, as it is a string and therefore more compatible with higher layers, as well.

```
/** @param contractAddress ...*/
async executeTask(contractAddress, taskID){...}
```

Fig. 3.4: The method signature from figure 3.3 after being merged.

Additionally, some objects like *Web3Wrapper* would allow being instantiated and configured without being initialized - like connecting to the Blockchain's endpoint, for example. It is generally recommendable to initialize an object immediately when instantiating it, as it could be created successfully while in an invalid state otherwise, leading to potentially hard to understand errors.

With the described changes, the previous example was reduced to the method signature shown in figure 3.4, being a clear improvement, as was done in many other places in the code.

3.1.3 Interface for Expansions

With the codebase cleaned up with the methods and principles described in the previous sections, it makes sense to tackle the next design question: How and where should an expansion to the existing system implemented?

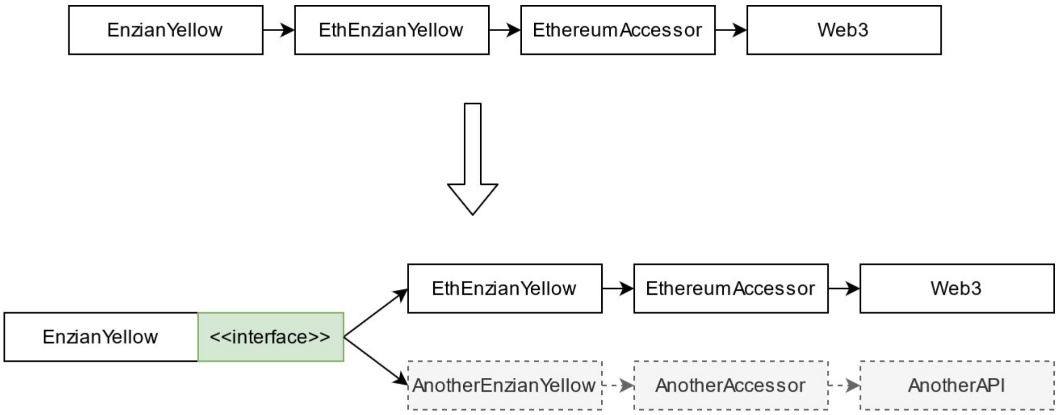

Fig. 3.5: *Enzian-Yellow*'s Structure before and after introducing an interface for expansion, with a sample expansion sketched in (grey).

As the introduced naming scheme might have already hinted, the perfect way to put an expansion into the current system would be to offer replacements for *EthereumEnzianYellow* that can fill it's signatures. In figure 3.5 we sketched this as a proper interface, but note that JavaScript as a language does not support such constructs and is weakly typed, meaning that any object with the same method names and signatures may already replace *EthereumEnzianYellow*, no inheritance or implementations needed.

Expanding the *EnzianYellow* module with a new Blockchain of type 'x' is therefore quite simple: One must only make their 'xEnzianYellow' known to the parent class, *EnzianYellow*, and define under which circumstances this new Blockchain application may be used. We opted for a simple constant, passed down to *EnzianYellow* as a string to configure it. If "ethereum" (or, in fact, nothing) is passed for example, *EthereumEnzianYellow* will be selected.

3.1.4 Transparency

TODO nested logs

3.1.5 Result

As per the definitions of this task, the behaviour of the codebase has essentially not changed after implementing these improvements. However, the goals of maintainability and expandability were definitely reached, with unnecessary code scatter removed, transparency increased in many places and many structures simplified. The only opportunity missed is to separate *Enzian-Yellow* into it's own secluded REST server, as this would've prevented the issue described in section 4.1.

3.2 Persistence layer

In this section we will discuss implementation of the Persistence layer of CHRYSALIS. We will list the problems solved by this tasks, details of the implementation both on the front- and backend side and results achieved by it.

3.2.1 Problem statement

Initially, when we got our hands on the project, CHRYSALIS stored all of the off-chain configuration data in the browser's local storage. Not only is it a safety concern (since the frontend user can easily manipulate data however they want), but also it is not reliable, since the browser's local storage could be cleaned on the user side and all of the data would be lost.

The other point of concern was that the only way to access deployed process model was by explicitly typing in its contract address, which renders the user interface completely useless and ruins user experience. Furthermore, to pick a task to execute, one had to explicitly specify the id assigned to it after the parsing into enzian model, which the user might not have even noticed. Lastly, there was no constraints on the task identifiers that could be chosen, so the user was perfectly capable of choosing an incorrect one and getting an error. To combat that, it was decided to implement a persistence layer, which would store all the off-chain information in a database, including information on associations between tasks and deployed processes.

3.2.2 Software stack

To implement this functionality it was decided to build a separate REST-server. For the server's implementation express.js framework was chosen, since the entire project is in javascript and express.js is meant for RESTful API implementation. PostgreSQL was chosen as a database management system, mainly because it is widely used and

Fig. 3.6: Process execution page before implementation of the persistence layer

optimized for production, but free at the same time (unlike, for example, Oracle). It also provides a wide array of object-relational functionality, which could be useful further down the line in CHRYSALIS development. For exchange between the server and the database Sequelize ORM is used.

3.2.3 Backend

As is required by Sequelize ORM and express.js framework, the server code is divided into four packages: models, migrations, controllers and routes. Models contain a representation of database entities, including column datatypes, constraints, associations and cardinalities. Migrations contain scripts used for propagating the database schema created in the model package and all the changes made in that schema to the database. Every migration contains a function for propagating the changes and a function for undoing them. The controller package contains the server's business-logic, the database interactions in particular. The routes package provides REST-API endpoints for communication with the server.

The database schema is rather simple and consists of six entities (apart from the system ones, needed for the Sequelize ORM to work). Those entities are: Process, task, connection, abi, setting and account. The entities "Process" and "Task" are self-explanatory. Connections contain information about blockchain networks that the user could connect to. Account contains information regarding user's account on the blockchain network, such as their private key for signing transactions. Abi is an entity containing compiled smart contract code for executing a process model and is deployed every time a new process is uploaded to the system. The "Settings" entity contains current set of connection configurations, chosen by the user. Processes and Tasks are connected by a one-to-many association.

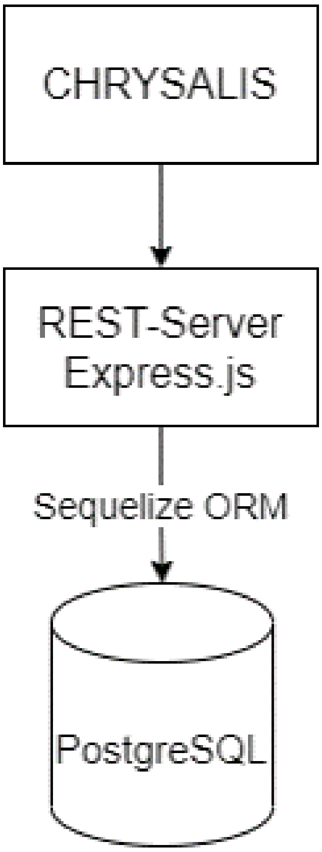

Fig. 3.7: Persistence layer architecture

3.2.4 Frontend

On the frontend side we had to work within the confines of the existing application. The main means of RESTful exchange in react.js is Fetch-API. But the problem is that Fetch-API is very wordy and we would have to reuse large blocks lots of times, in every component of the application. Not only that, nut one would have to call this API with a wide array of different parameters, depending on the caller's intention. So it was decided to implement some universal exchange handling functionality within the frontend application.

To implement this it was decided to create an ExchangeHandler class which would be called by the application components to handle their requests. The components would pass to it the request method, URI and optionally data that they have to send, and then based on the method the exchange handler would find the appropriate instance of one of the sender classes and call them to execute the request. These sender classes are GetRequestSender, PostRequestSender, PatchRequestSender and DeleteRequestSender. They are instantiated and kept in the SenderRepository class. It contains a map with request methods as keys and sender instances as values. The exchange handler gets an appropriate sender from this map and calls its send()

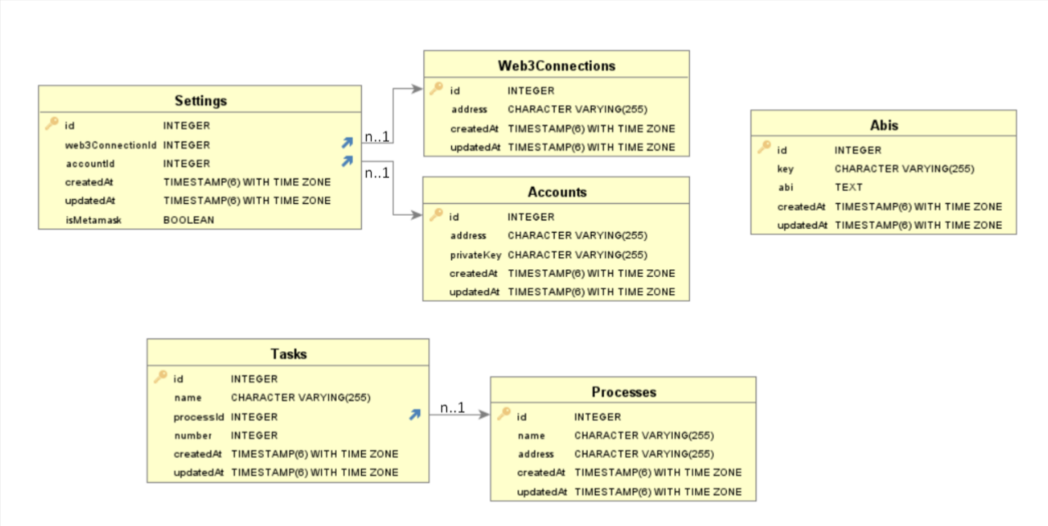

Fig. 3.8: Database schema

method to make a request to the server, returning a special promise objects, which provides methods to specify a logic, that should be executed once the response is received.

3.2.5 Result

After the changes made, all of the off-chain data was moved from the local storage to a separate database, improving safety and reliability. User experience was also significantly improved, since it became possible to choose deployed processes by their name from a list provided by the server, as well as choose from tasks associated with the process, by their name as well. The functionality of retrieving deployed processes by their contract addresses (if they are not present in the database) was preserved as well, but it underwent slight changes to make it compatible with the new architecture, and these changes will be discussed in the smart contract optimization section.

3.3 Hyperledger-based Application

With the interfaces needed to easily expand the system being in place, we deemed it a good next step to add another Blockchain protocol. Specifically, we chose *Hyperledger*. The application was successfully expanded with all necessary components built and tested, although the front-end still has problems trying to pack this new addition and sending it to the user's browser. This issue is elaborated in section 4.1. This chapter will focus on firstly giving a broad overview of Hyperledger in section 3.3.1 and then following up with detailing the implemented expansion in the later sections.

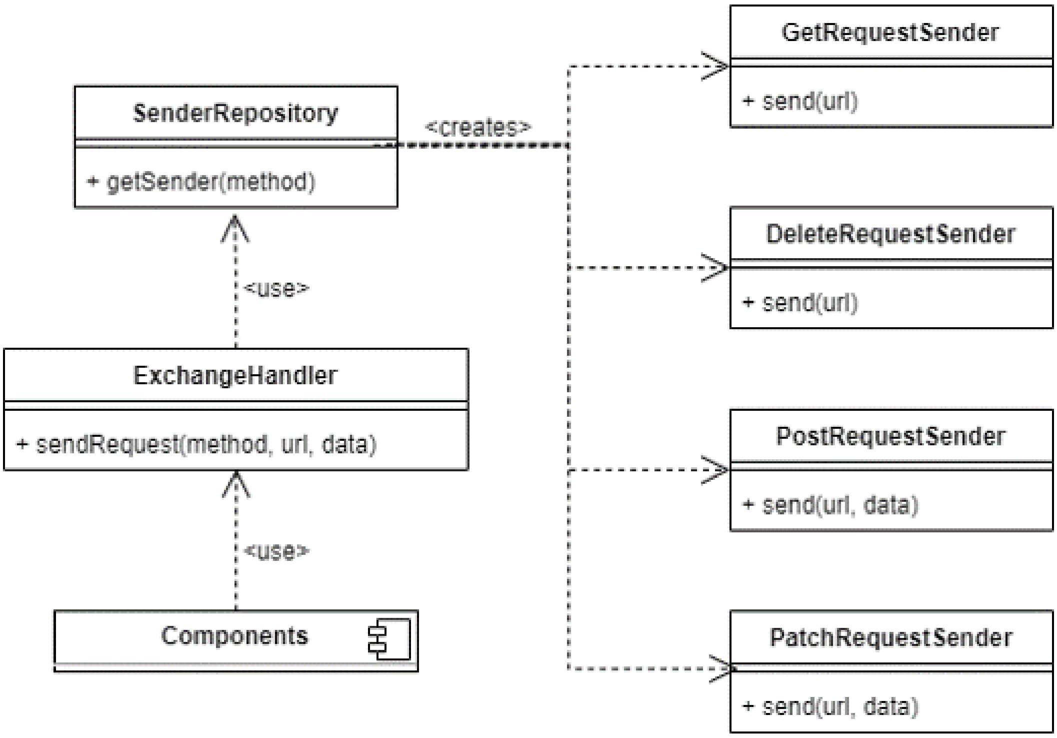

Fig. 3.9: Database exhange module

3.3.1 Hyperledger as a Ledger Protocol

To explain how the Systems of *Hyperledger Fabric* work and what special features they offer, it makes sense to offer a comparison to the *Ethereum* Blockchain - feature by feature. *Hyperledger* wants to set itself apart by being a business-oriented information transfer protocol (hence, a shared ledger). Values, like tokens (e.g., *Bitcoin, Ethereum*), are not a fundamental part of it.

Authorization

If configured, a Hyperledger Network will be split into groups, called organisations. Each organisation has a (potentially distinct) *Certificate Authority (CA)*, through which membership in an organization is validated for every peer and account. Therefore, if one wants to interact with the network by for example invoking a smart contract, they must first be authenticated by the CA. The CA even allows for group roles, so not every query to the network is accessible to everyone. These group roles were not further regarded in this task, however.

Block Sealing

Block sealing, the process of adding blocks to the Blockchain and therefore changing it's active state and log, is an essential part of Blockchain technology. In more known protocols like *Bitcoin* or (at least at the time of writing) *Ethereum*, the content of the next block is decided by the node that first managed to solve a cryptographic

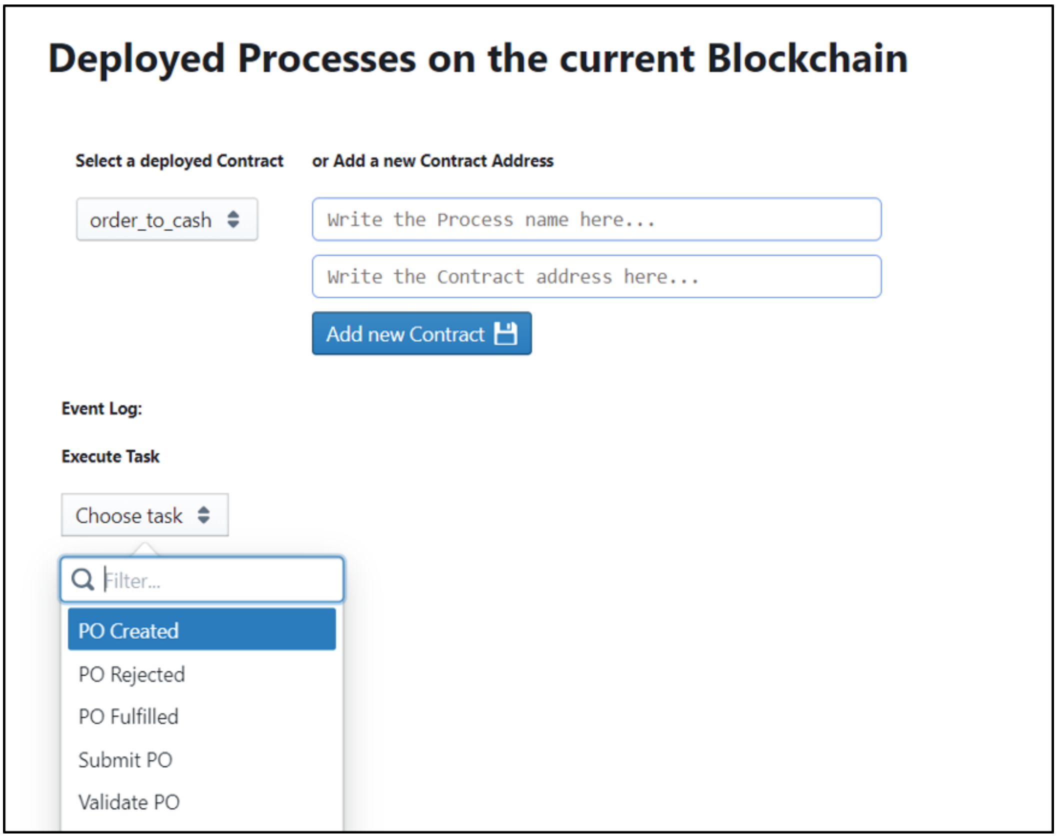

Fig. 3.10: Process execution page after the improvements

puzzle - the solving of this puzzle being called *mining*. This, however, means that the time of sealing a block is somewhat random - and a bidding process is necessary to convince the block sealer of writing one's data.
Hyperledger circumvents this competition-based method and instead introduces an *orderer* node, which generally tries to apply block-additions in a first-in-first-out fashion (and accounts for concurrency issues). This means a more reliable way of querying the blockchain, since all actions are executed as fast as resources allow, and also because competing peers are treated equally instead of based on their bid. Additionally, from a security standpoint, a network can not as easily be 'poisoned' (by holding such a large amount of tokens or mining power that the holder can essentially decide which transactions to incorporate), since the finances and mining capacities of peers are disregarded completely.

Abstraction: *World State & Chaincode*

As depicted in figure 3.11 by an 'engine' component (although this is a big simplification), Hyperledger has functions in place to display the underlying Blockchain's contents in a more abstract way, so that the user or developer doesn't need to bother with physical addresses, but may instead find them in a more human-readable format.
The *World State* Container is a synchronous representation of all data present in Hy-

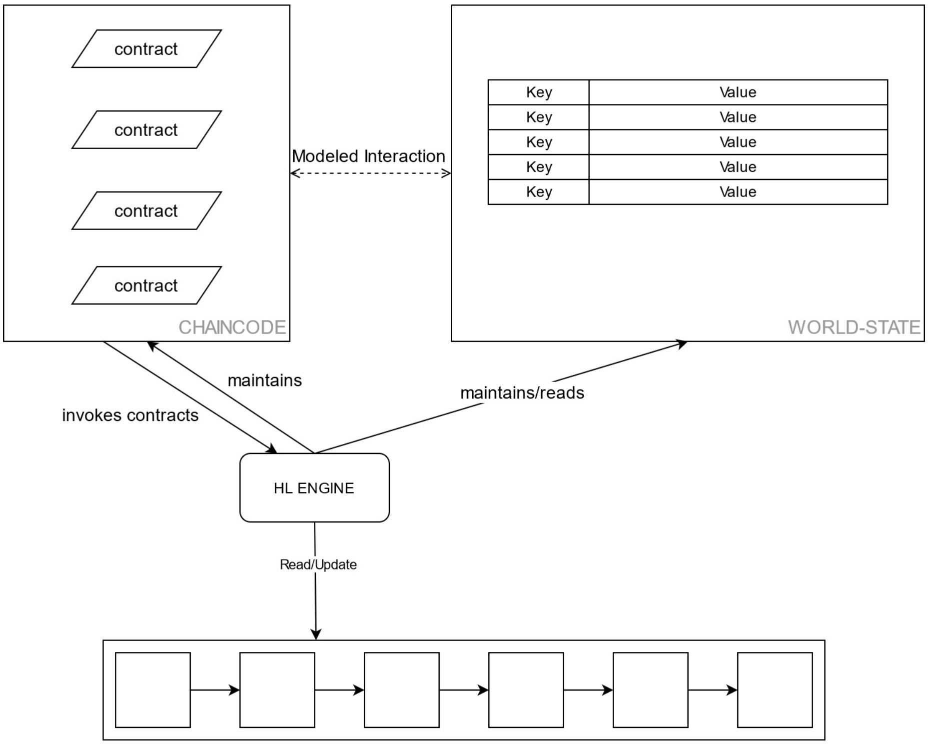

Fig. 3.11: Sketch of *Hyperledger's* abstraction mechanism.

perledger's underlying Blockchain, meaning that the data points present are always a representation their most recent update. Additionally, the World State is arranged like a dictionary, so that every structure inside is reachable by a key, defined by the user itself. This way, the programmer doesn't have to worry about physical addresses of the data.
In the *Chaincode* containers one can find the Smart Contract objects one would also find in other Blockchain application. However, these contracts are not stateful and therefore do not contain any data besides the location of the World State, where the data may be contained. In comparison, an *Ethereum*-based contract may have private data and would therefore be stateful. Additionally, Chaincodes are also invocable via name, specifically by a combination of the parent contract name and the function that is to be executed.

3.3.2 Component Overview

Three components will be needed to build a running Hyperledger-based application for the Chrysalis project.
First of all, a Hyperledger Network must be established in the first place, so all necessary structures can be deployed there. Secondly, being the most complex component of the three, the Chaincode must be defined along with the data structures it uses. This module may be written and deployed in generic JavaScript thanks to

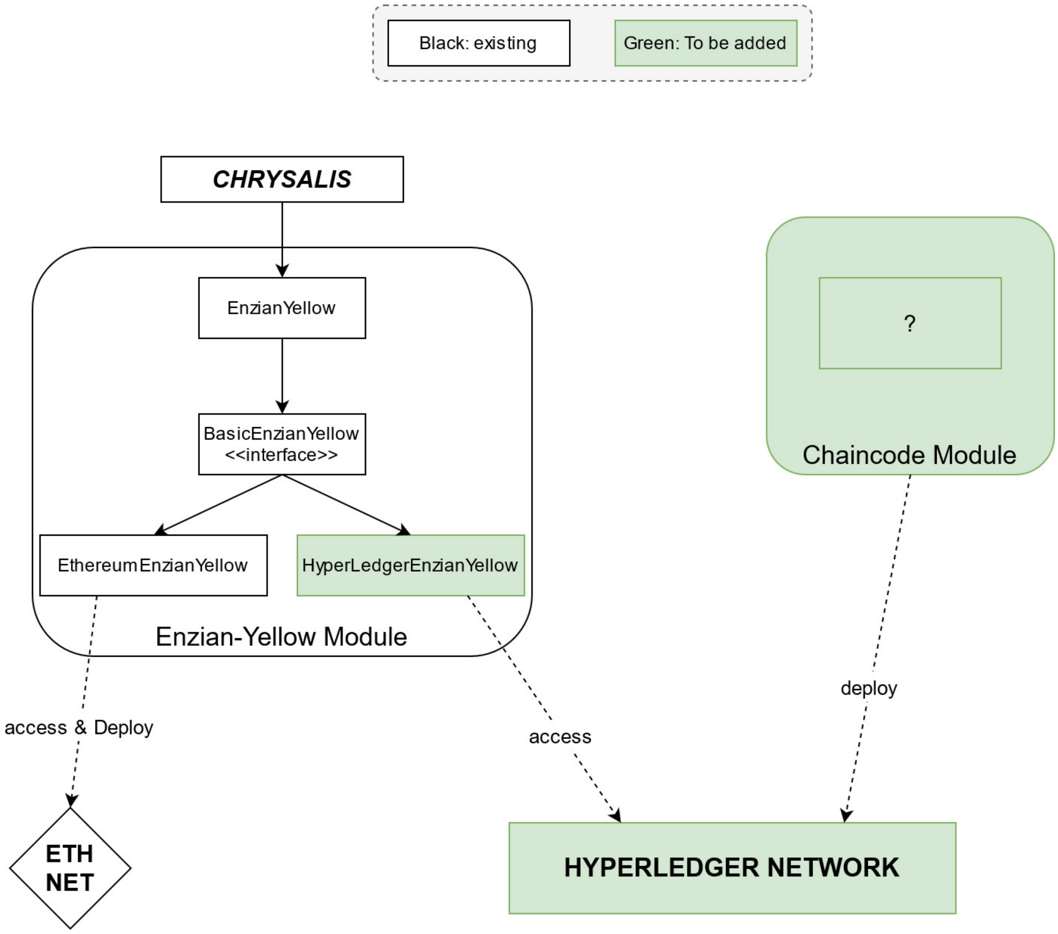

Fig. 3.12: Planned and implemented module structure after adding the *Hyperledger* protocol

Hyperledger's flexibility regarding the deployed language (e.g., this module could also be written in *Go*). As per Hyperledger's requirements, the module must be contained in a separate package that can locally install dependencies and can be packed into a compressed file. Once this package is deployed, an API is needed in *Enzian-Yellow* to access the Network and use the deployed components. This step is fairly straightforward, since Hyperledger Fabric provides this API and the structure of the *HyperledgerEnzianYellow* can mostly be mirrored from that of *EthereumEnzianYellow*.

3.3.3 Test-Network

To keep development effort low, the Hyperledger Network where the contracts and data may be deployed was not built from scratch, but instead the "test-network" from Hyperledger's *fabric-samples* repository was used. This network came with the features stated below.

Network Deployment: Instead of having to configure every network node, having to start it manually and registering it to the network, the test-network spares the developer from this work by making a running network with isolated components available as a composition of *Docker*-Images. Additionally, with the help of pre-written scripts, the user may easily instantiate said composition.

Reset functionality: With a pre-written script, the developer may also simply wipe the entire network and bring it down. This makes development especially easy, since no trace (potentially even illegal states) of previous work will be left on the Blockchain.

Multi-Organization Scenario: If instantiated with the right parameters, the network would come up with two organizations created, both containing one network peer each and reporting to a certificate authority. This configuration is useful, since the developer can make sure their application would run in a realistic scenario where their application and contracts must be validated by all network-registered organizations. The test-network also offers a lot of helpful scripts to make interactions with it a bit less cumbersome, e.g., so that Chaincodes may be deployed with less steps.

3.3.4 Representation of Processes

It makes sense to split the data model deployed onto the ledger into two parts: The first defines how a JavaScript object may be represented inside the *World State* as well as in memory and how these two versions are to be synchronized. The second part, building on those definitions, defines how a process as well as it's tasks shall be modeled as such objects, how they interact and how they may be operated upon. The resulting components are visible in figure 3.13 and described below.

Objects on the World State

As the World State acts like a dictionary, any data placed on it is modeled as an extension of the class *State*. This State object defines its own key with which it is found on said dictionary-like structure, as well as a static method to serialize and deserialize it to/from a JSON representation, as it has to be stored in a more permanent form.
In addition to this data class, a 'manager' object is needed, in our case of the type *StateList*, to store the location of the World State itself, to write or update deserialized State objects onto it, and to cast the 'rehydrated' objects back onto the correct class.

Process Representation

With the definitions of *State* and *StateList* given, we can now implement the process

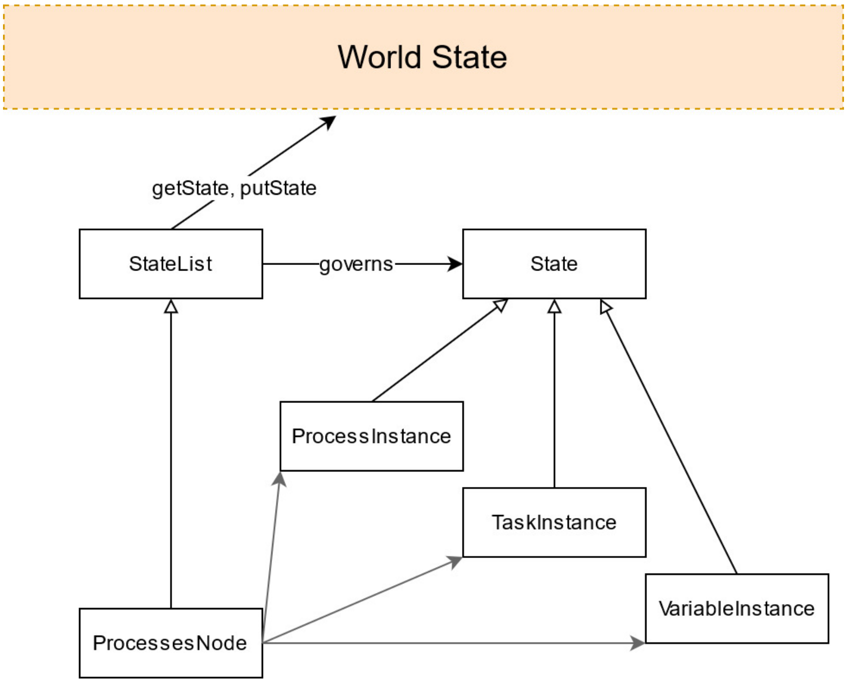

Fig. 3.13: Data Structure deployed on the World State (State) as well as the objects that actively read and write on it (StateList)

structure as a simple extension of those, mostly not having to worry about the internals of the World State.

- **ProcessInstance**, being an extension of State, is the structure that binds an active process together: It stores the keys that point toward it's subordinated Tasks and Variables, and also contains the event log, which is an ordered list of task IDs, appearing in the order the tasks were executed (a newly created ProcessInstance would therefore contain an empty log). As the ProcessInstance is the 'root' object of the data structure, it's Key is simply defined as a positive integer.

- **TaskInstance**, representing the Task State, contains - besides it's key and ID - all necessary data that makes up tasks in BPMN: A name, a list of competing tasks and completed task IDs required for execution, and so on. Additionally, to model BPM gateways it contains a *precedingMergingGateway* attribute as well as a *decision* structure, similar to (TODO: Christian's paper). The key is defined as follows: *[key of parent process]:task:[task ID]*

- **VariableInstance**, lastly, is a simple wrapper for a value of a given type, with a log of previous values as an addition. Currently, only variables of type integer and string are supported. The key is defined as follows: *[key of parent process]:var:[variable ID]* - also showing why the middle section differentiating between tasks and variables is needed: Variable and task IDs may clash and must therefore be expanded by some prefix.

The **ProcessesNode** class, extending StateList, acts as a gateway to the previously defined objects, offering the usual *get, set* and *update* methods. Also, since it makes sense to immediately write every action on the data structure onto the World State after executing it to prevent bugs, ProcessesNode also offers many BPM-based functions like *executeTask*, therefore encapsulating a lot of Process logic as well. Objects of this class are meant to make it possible to interact with the process model entirely without knowing the underlying model, only using provided keys.
How this ProcessesNode finds its way into a Chaincode, however, will be explained in the next chapter.

3.3.5 Process Deployment and Execution

A Smart Contract (Chaincode) in Hyperledger Fabric, as well as the data structure, can be written in multiple supported programming languages, including JavaScript, which was used here. Implementing it is as simple as creating a class that extends the *Contract* class from the Fabric API, as seen in figure 3.14.

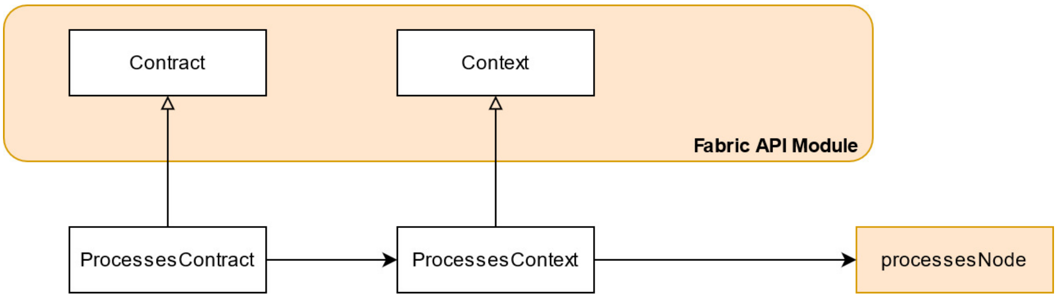

Fig. 3.14: Structure of a Chaincode (Contract) together with it's gateway object that points to the World State (Context)

Contracts

Defining what would be an invocable contract from the outside is as simple as giving the extending *Contract* class a method: Users will be able to invoke this function simply by its name coupled with the class's name, e.g., "ProcessesContract" and "createProcess", together with it's arguments. However, a few special rules apply here: Firstly, while parameters and return values can be generic JavaScript objects that do not have to be serialized, all non-primitive types must be transferred via *Buffer* to account for the networking between the caller and the callee. Secondly, the contract methods will always be provided a *Context* object (see next passage for details) as first positional argument. This context will be handed to the method by the Hyperledger API, not the caller, therefore only the second argument and all those after may be used for argument passing.

TODO(screenshot of ctx object?)

Context

The Context, as described before, is another object provided by Hyperledger's API and handed to every smart contract during execution. Foremost, it contains the network-*stub*, which is the gateway to the World State. We can now extend this Context into *ProcessesContext*, that always contains a newly instantiated ProcessesNode object, therefore also giving access to all the data structures and logic as we defined them in section 3.3.4. During instantiating, the ProcessesNode is also handed a reference to the stub, so that it may have an access point to the World State. To make clear to Hyperledger's API that our Chaincode is supposed to use this version of Context, we override the Chaincode's method *createContext* to return our version instead of the default one.

With these two components, it is now clear how a contract invocation results in manipulations or queries on the processes: When a method of ProcessesContract is invoked, it is handed the ProcessesContext and uses it to access ProcessesNode to then execute all process-related logic there. The whole interaction is currently purely based on passing and returning keys and primitive construction parameters, however it would also be possible for the caller to receive objects from the data structure as copies of their current state - still, it should generally be disallowed to write self-made objects to the network, as this might cause trust issues.

3.3.6 Integration into Chrysalis

From the side of HyperledgerEnzianYellow, interaction is fairly simple. Due to the underlying interaction model in the Hyperledger network being similar to the model deployed to the *Ethereum* network, the general structure of HyperledgerEnzianYellow is basically the same as that of EthereumEnzianYellow.
When HyperledgerEnzianYellow is instantiated, the HyperledgerAccessor is also constructed, building up a working connection to the Hyperledger network during construction (or throwing, if the connection fails). Currently, provided arguments aren't used, but instead read from a static definition, as an integration error (see section 4.1) made using it properly impossible.
Once the connection is built, the accessor can then freely pull a Chaincode by simply specifying the correct name, being returned an interface object. This interface can then be used to invoke the contracts by name and handing over parameters, as if the method were local. Using the previously defined Chaincode, HyperledgerEnzianYel-

low can offer the same functionality for BPM as EthereumEnzianYellow with the same parameters, the only difference being that the process addresses are in fact structured keys instead of proper addresses.

3.3.7 Result

All in all, the task of implementing a Hyperledger-based process execution engine as a prototype can be considered successful and fruitful. The Hyperledger Smart Contracts are able to mirror the Ethereum Contracts, can be deployed on a running system successfully and can be connected to - especially by HyperledgerEnzianYellow. From a BPM perspective, the Hyperledger application fulfills all required functions and even partly surpasses Ethereum in a few features:

- **Execution Speed:** As Hyperledger doesn't rely on mining to sign and validate new additions to the Blockchain, the deployed contracts can be invoked with fairly low delay - about two seconds per invocation. A direct comparability with Ethereum is not given, though: A mining-based protocol can technically be sped up massively by adding more miners and reducing the hashing difficulty of the mining puzzles. Still, Hyperledger's test-network is rather feature-rich and probably contains a lot of overhead that might not be needed in our case, and might also be a lot quicker if optimized.

- **Reliability:** Compared to a mining blocks, adding them after a consensus is not based on random numbers. Therefore, the time between a contract invocation and the completion of the underlying transaction is extremely constant. In Ethereum's case, we saw huge discrepancies between the times individual transmissions took.

- **Readability:** As a small bonus, all addresses Hyperledger uses - even internally - are developer-defined and therefore have a human-readable structure, like those of the process data structure defined in section 3.3.4, and therefore make development and back end work a lot easier and more transparent even without usage of a persistence layer.

- **Efficiency:** Without mining, we perceived a significant reduction in processing load on the Hyperledger application compared to the Ethereum one. For example, where the Hyperledger application occupied a few threads with barely any load on the processor, the Ethereum miner alone (before process Chrysalis even ran), needed to put multiple processor cores on full load only to facilitate a transaction delay low enough to comfortably work with. This issue might not arise on more powerful server architectures, though.

3.4 Ethereum overhaul

In this section we will discuss improvements regarding the smart contract structure of the project. Smart contracts are used for the deployment and execution of process instances within the system. They contain structures and methods for creation and handling of tasks and control flow decisions. For each process instance one new instance of the BasicEnzian smart contract is deployed.

3.4.1 Problem statement

In its initial state, the project kept all of its on-chain functionality in one single smart contract. There was no separation of concerns, a lot of the universal, non instance-specific functionality was left in, which lead to not only poor scalability and reusability of the smart contract code in the future, but also to significant performance costs, which, on the ethereum network, are measured in a unit called "gas". Gas consumption defines how much computational power the network needs to execute a certain operation, which then determines how much currency should be transferred from the account of a node requesting to execute this operation, to the account of the owner of the node executing it. So in case of Ethereum, performance costs literally translate to money spent, that is why the main goal of the optimization was to minimize them. The listing below provides an example of code redundancy within the smart contract functionality. It contains a part of the task execution function, and as can be seen, mostly consists of parts which do not have to be in the contract and could be delegated to external components.

Listing 3.1: Task execution before the optimization

```
function completing(uint taskId) public returns (bool success){

        require(!tasks[taskId].completed, "DO NOT REPEAT TASKS!!!");

    uint endBoss = 0;
    Task memory thetask = tasks[taskId];
    // ORGANISATIONAL PERSPECTIVE

        address resource = tasks[taskId].taskresource;
        require(resource == address(0) || resource == msg.sender,
            tasks[taskId].activity);

    // INFORMATIONAL PERSPECTIVE

        // evaluate Decision

        if(tasks[taskId].decision.exists) {
```

```
            endBoss = tasks[taskId].decision.endBoss;
            bool result = evaluateDecision(tasks[taskId].decision);
            require(result, 'Process Variable is not correct.');
        }

    // CONTROL-FLOW PERSPECTIVE

        uint[] memory requiredTasksIds = tasks[taskId].requirements;
        if (requiredTasksIds.length == 0) {

            success = true;
        }
        else {

            GatewayType gateway = tasks[taskId].
                preceedingMergingGateway;

            if (gateway == GatewayType.NONE) {
                if (isTaskCompletedById(requiredTasksIds[0]) == true)
                    {
                    success = true;
                }
            }
```

3.4.2 Solution

To achieve this, we had to optimize the most expensive operation in our system - smart contract deployment, since it happens for every new process instance in the system. Also the issue of reusability of components becomes more crucial, because as the project will become progressively more complex in the future, the on-chain functionality will do so as well, so we have to split the smart contract code into elementary parts, which would allow to implement new features by adding and deploying new modules, using existing ones, instead of rewriting and redeploying everything, which would also decrease performance costs during future development.

As an implementation model for this task clean architecture was chosen, since this model is designed to increase reusability of application components and overall scalability of the application. The main principle of this model is that all the modules are diveded into three layers - domain layer, application layer and presentation layer. Domain layer defines structures and entities used in the application, basic objects carrying its state. Application layer, or middle layer, contains business logic of the application, the modules responsible for its behaviour, dealing with the object from the domain layer. Presentation layer provides entry points for external systems and users, that call functions and methods of the middle layer. The main requirement is that modules can only interact either with the modules from the same layer or the inner layer modules.

3.4.3 Implementation

On the domain model layer, the libraries containing structs defining the objects of a process model along with the process state were implemented. These libraries were TaskEntities, DecisionEntities and ProcessEntities.
TaskEntities library define the structure of a process task, which contains information on Tasks name and id within the process instance, its completion state, as well as the set of requirements, that have to be fulfilled for this task to be up for execution. It also contains a reference to a Decision object, which is described in the DecisionEntities library.

DecisionEntities library defines structures and enumerators needed for evaluating control flow requirements in the gateways of a process model. The decision structure utself contains information on the gateway type, variables compared as well as the comparison operator for those variables. The enumerators containing gateway types and operators are also defined in this library.
ProcessEntities library defines the structure containing information on the process structure and state. It contains the process' tasks, its integer variables and string variables.

On the application layer there are two libraries: TaskLibrary and DecisionLibrary. TaskLibrary implements methods creating tasks on the process initialization, as well as methods handling execution of tasks including evaluation of task requirements. To evaluate gateway conditions it calls the DecisionLibrary, which implements appropriate functionality.
The presentation layer consists of the BasicEnzian smart contract, which provides entry points for the EnzianYellow library to interact with, as well as keeps the state of the process instance and its event log.

3.4.4 Result

As an outcome of the smart contract overhaul, the structure of the smart contract code became more scalable and flexible, it was divided into small modules which can be reused in the new contracts developed further down the line. Other than that, the cost of process instance deployment was lowered from 4300000 gas to 3600000 gas, and overall the code became much cleaner, which could be seen in the following listing, which showcases the same function handling task execution. After the refactoring most of the method's logic is incapsulated in the TaskLibrary, which is called, and then the results from the library function are used to update process state.

Listing 3.2: Task execution after the optimization

```
function completing(uint taskId) public returns (bool success){
        require(!processState.tasks[taskId].completed, "DO NOT REPEAT
             TASKS!!!");
        TaskEntities.Task memory thetask = processState.tasks[taskId
            ];
        uint endBoss;
        (success, endBoss) = TaskLibrary.completeTask(thetask,
            processState);

        if(thetask.decision.exists) {
            processState.enabled[endBoss] = success;

            //LOCKING
            for (uint i = 0; i < thetask.competitors.length; i++) {
                processState.tasks[(thetask.competitors[i])].
                    completed = success;
            }

        }
        processState.tasks[taskId].completed = success;
        emit TaskCompleted(success);

        if (success) {
            debugStringeventLog.push(thetask.activity);
            theRealEventLog.push(taskId);
        }

        return success;
    }
```

3.5 Summary of Improvements

TODO section even neccessary?

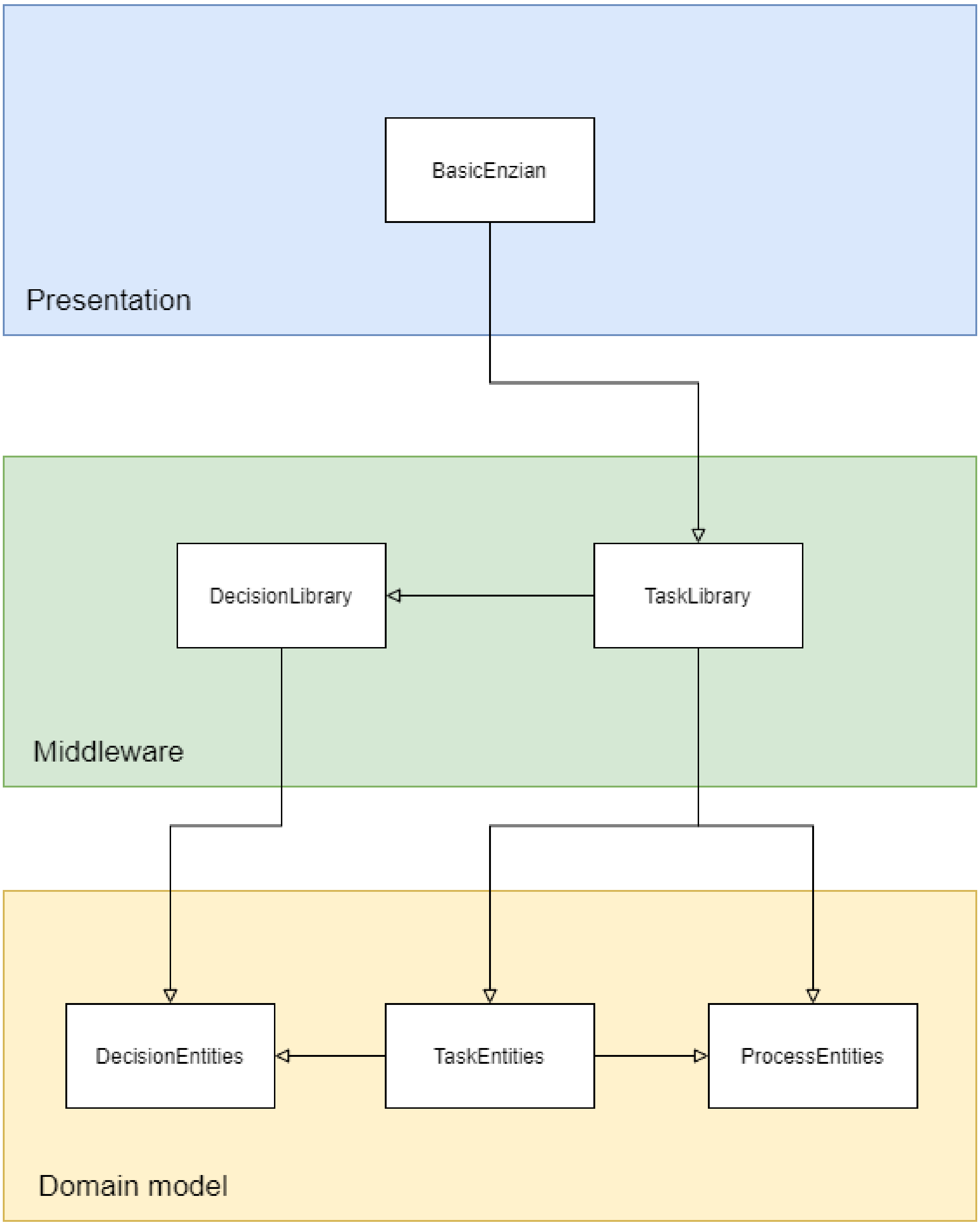

Fig. 3.15: Structure of the ethereum smart contracts

4 Open Issues

While we generally consider our tasks successfully completed, we've encountered a few issues that we did not have the time left to solve. Also, we weren't meant to bring the *Chrysalis* project into a final state to never be touched again, but to pave the way for future iterations and improvements.
With this chapter we want to lay out what issues exactly are not solved as we hand in our work, as well as give some suggestions what steps would generally be smart to tackle next - either to prevent more problems or to expand functionality.

4.1 Integrating Hyperledger into Chrysalis

With a *Hyperledger* application introduced to the project, we were not able to deploy the front-end application into browsers any more, as the packing process of the *react-webpack* application failed because of a Hyperledger dependency. Because of this, all Hyperledger-related changes are currently shelved into different branches:

- For *Chrysalis*, no changes would be needed. The *main* branch therefore should suffice. An incomplete fixing attempt to do a version upgrade (see below for details) may be found in the branch *webpack-upgrade*.

- *Enzian-Yellow*'s running version, excluding all *Hyperledger*-related changes, is found in the *main* branch. The Hyperledger-related changes may be found in the branch *HyperLedger*.

The issue

Currently, the *Enzian-Yellow* module is installed as a dependency in the *Chrysalis* module, meaning that the entire code of Enzian-Yellow will be deployed onto and executed in the target browser. To do this, in a process called *minifying*, the *webpack* library parses all code, including dependencies, and outputs a minimal 'bundle' to save bandwidth for the browser. However, webpack's parser has a bug that appears only when parsing a specific statement ('import.x'), which incidentally exists in the Hyperledger API. Therefore, while everything is functional, Chrysalis is unable to deploy the new code.

Solution 1: Version upgrade

As the parser bug stems from *webpack*, we attempted to upgrade it's version from 4.x to 5.x, but didn't find the time to complete it. However, we propose a more permanent solution to such compatibility issues below.

Solution 2: Separating Enzian-Yellow

While being a viable option, installing all project dependencies and deploying their packed version into a browser may become unhandy and slow once the project reaches a certain size. As an alternative solution, future developers could focus on creating a REST-server similar to the one constructed in section 3.2 and instantiate *Enzian-Yellow* only there - offering it's functions via a network interface while keeping the local browser version lightweight.
Additionally, it makes sense to keep *Enzian-Yellow* physically close to the Blockchain nodes: As the application uses multiple APIs to interface with said nodes, network traffic may become rather complex between *Enzian-Yellow* and the nodes. Keeping the APIs close, however, would keep such issues at a minimum.

4.2 Improving on the Process Model

As we've done a lot of architectural work, focusing on giving future developers as many tools to expand the system as possible, we propose indeed expanding the system further as a next step: The currently implemented BPM engine is rather limited, for example not being able to execute tasks more than once and therefore making loops impossible.
With the architecture we created, it should be a relatively easy task to overhaul the engine model of *Chrysalis*, adding more BPM features and maybe even fully supporting all BPMN features one day.

5 Caterpillar

One of the tasks of our project was to analyze a system similar to CHRYSALIS and compare it to our project. A project called Caterpillar has been chosen for comparison. In this chapter we give an overview of different versions of Caterpillar, its architecture and working principles, as well as compare it to CHRYSALIS in terms of performance and features implemented.

5.1 General overview

Caterpillar is an ethereum-based process model execution system. It is available in two versions: Version 2 is a compilation-based engine. It compiles process models into smart contracts being executed by ethereum blockchain. Version 3 is an interpreter-based engine, the model is processed by an interpreter smart contract, which executes the tasks of the process. It implements the compliance-by-design approach - the parties execute each step of a business process by executing transactions on the blockchain. When a transaction is invoked, the blockchain platform checks the current state of the process and the inputs and outputs of the transaction. The transaction is accepted if and only if it complies with the collaborative process model. This approach is suitable for a scenario, where the level of trust between the parties is low, the impact of non-compliance is high, and conflict resolution is expensive. This scenario is addressed by Caterpillar.

5.1.1 BPMN features supported

Caterpillar supports execution of subprocesses, which are implemented through a smart contract hierarchy in the runtime registry contract. For the types of tasks it supports user tasks, service tasks (with solidity smart contracts implementing service logic) and script tasks (with solidity scripts attached). For the beginning of the process it only supports a plain start event. Moreover, the evaluation of exclusive, parallel and event-based gateways is implemented within caterpillar. The types of events supported are terminate, default, message, signal, error and escalation events. t also supports multi-instance activities, parallel as well as sequential and events attached to the boundary of an activity.

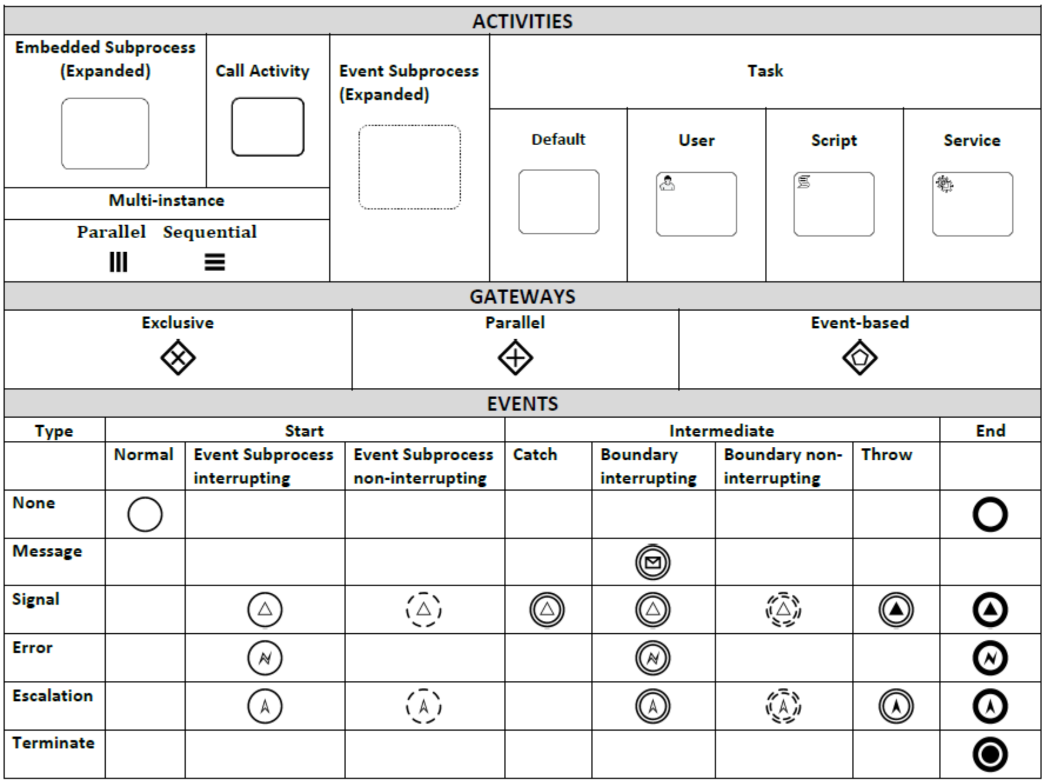

Fig. 5.1: Overview of the features supported

5.1.2 Architecture

In general, the architecture of Caterpillar is similar to this of our project. It consists of three layers - on-chain layer, responsible for deployment and execution of processes, backend layer, responsible for processing bpmn-models and interacting with the blockchain and frontend layer, providing a user interface (except for version 3, which does not have the frontend layer). The only difference on this level is that the backend layer is implemented as a REST-server and not a library (which allows users to interact with the version 3 of caterpillar even without the web application).

The On-chain layer supports the execution of smart contracts that fully encode a set of process models. The events generated by the contracts are recorded and stored in the Ethereum log, which can be accessed from outside the blockchain. The process repository is an off-chain storage, keeping and providing access to BPMN-models, solidity code generated from them (only in version 2), and metadata linking solidity contracts to elements of bpmn-models. The process repository is implemented on the top of Interplanetary File System (IPFS).

The off-chain runtime layer (referred previously as backend layer) includes tools to compile (only in version 2), deploy and monitor business processes in the blockchain, which allow external applications to interact with the on-chain components and the repository. Finally, the top-most layer incorporates a set of tools for editing process models, packaging process configurations and initiate and monitor execution of process instances.

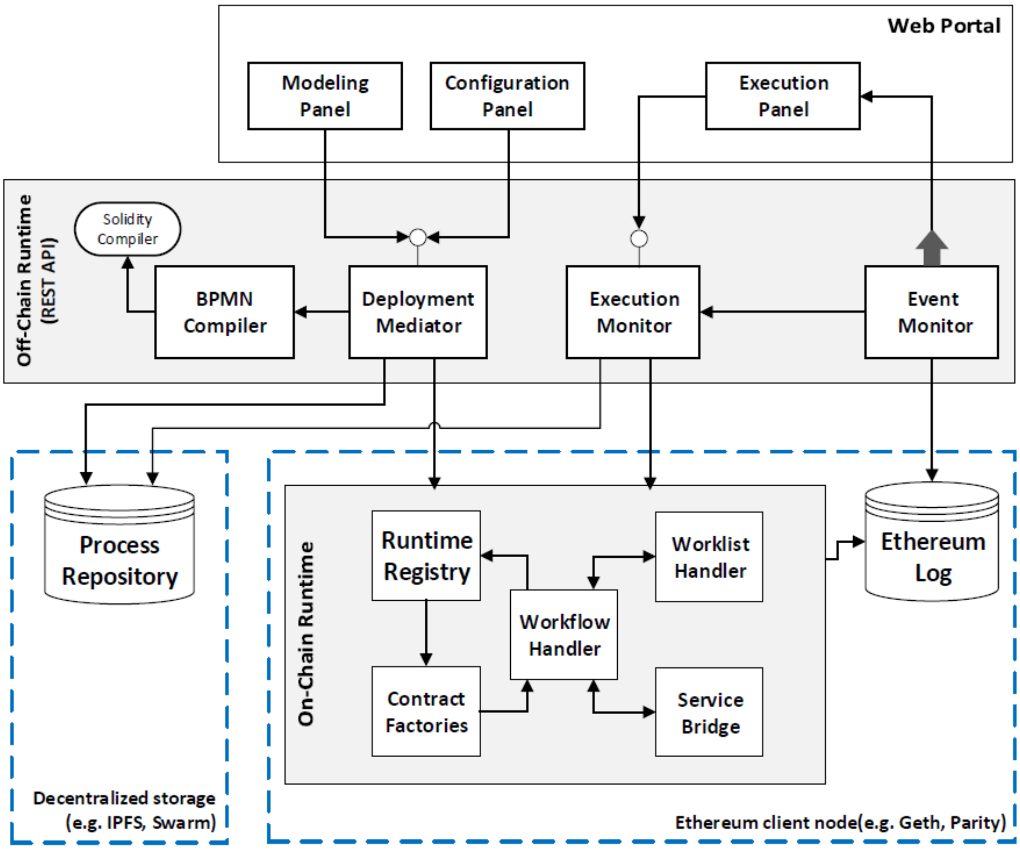

Fig. 5.2: General architecture of Caterpillar

5.2 Compilation engine

The version 2 of Caterpillar provides an engine based on the compilation of BPMN-models into solidity smart contracts. In this section we will give an overview of the compilation process as well as the smart contract structure of this version of Caterpillar.

5.2.1 Compilation process

For each process model uploaded Caterpillar V2 generates a set of solidity smart contracts. The compilation is conducted in two steps. On the first step, the solidity code for the process contracts is generated, as well as additional metadata, called the compilation dictionary, which is used for monitoring processes. It is a data structure which maps the elements of the source model to the generated code. This information includes the name of the contract method associated with an activity, a unique integer index assigned to each element, as well as the element type.

On the second step, the generated smart contracts are put together, as well as pre-existing contracts, i.e attached to the service tasks. These contracts are then passed to the solidity compiler, which produces EVM-bytecode and ABI definitions for each contract, that are needed for deploying smart contracts on Ethereum. These definitions are later used by off-chain components to interact with the contracts and trigger their execution. Artifacts of the compilation are stored in the process repository.

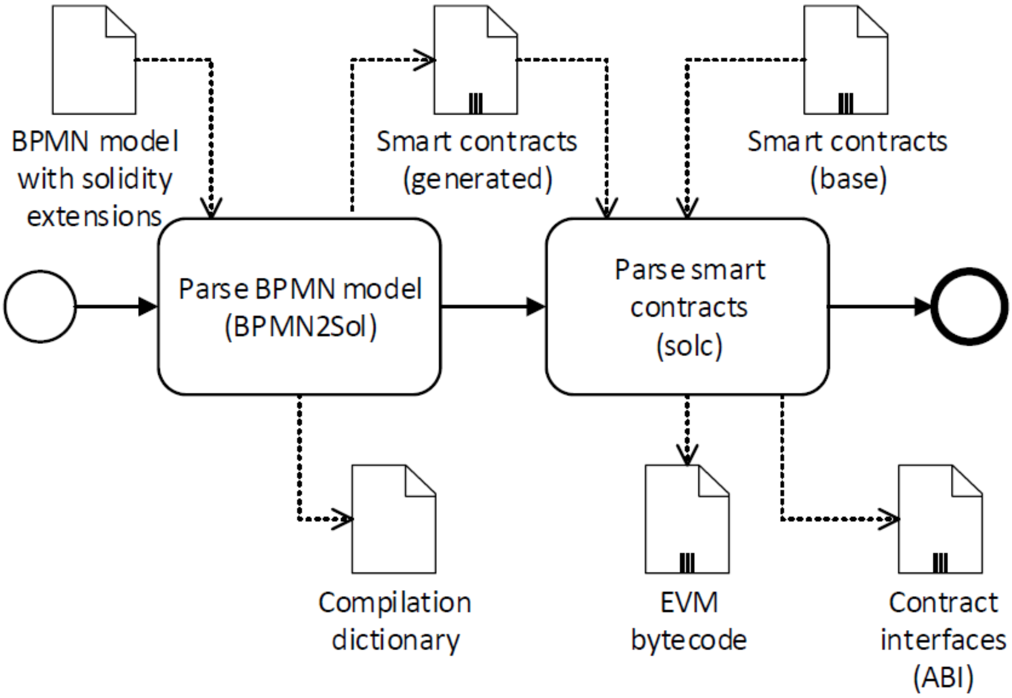

Fig. 5.3: Caterpillar V2 compilation process

5.2.2 Smart contracts

The contract that is deployed at the start of the application is the process registry contract. It keeps track of the deployed process models, their corresponding contracts, started process instances and their execution states. It also stores the subprocess hierarchy by keeping links between smart contract bundles associated with each process therein.

The contracts generated for each of the process models implement interfaces AbstractProcess, AbstractFactory and AbstractWorklist. The contracts implementing AbstractProcess interface contain information on the process structure, methods implementing execution of different process model elements, including firing and handling events. It also contains a reference to the parent process if it exists, as well as a reference to the worklist contract. The contracts implementing AbstractWorklist interface implemement data perspective of the process model execution. They store process variables with association to the model elements. The contracts implementing AbstractFactory interface contain logic relating to creating and starting execution of new process instances. They are called from the process registry on instantiation and from the process contract on execution.

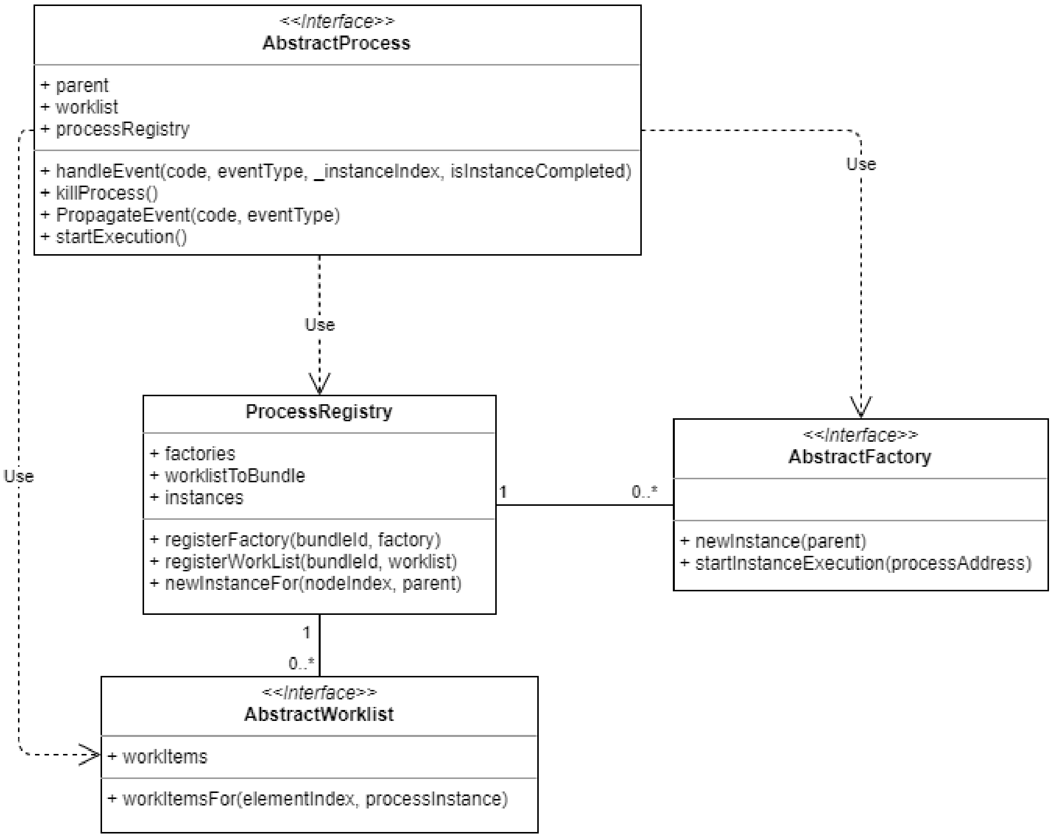

Fig. 5.4: Structure of the Caterpillar V2 smart contracts

5.3 Interpretation engine

In the version 3 of the caterpillar application, the process execution paradigm was changed significantly. Instead of compiling process models into smart contracts, the models were read and executed by an interpreter contract. In this section we give an overview of this version of caterpillar

5.3.1 Interpretation process

The control flow component stores the information about structure of the process model, its elements and relations. The data is organized in a tree-like structure to implement the subprocess hierarchy. Each node in the tree maps for each enclosed BPMN element the model-related information to be used by the interpreter. As a result, each node is deployed once for each subprocess in the model and is identified by the corresponding address.

The process case factories include the set of contracts to instantiate and start the execution of a business process. Therefore, if a subprocess is linked into the hierarchy, the parent has to store an address of the corresponding factory to instantiate the subprocess.

The data and scripts component implements the data perspective. To implement separate data requirements for each process, smart contracts are compiled from each process model. The scripts related to user/service/script tasks and the conditions to decide the paths in exclusive gateways mostly interact with the process data. Thus, such instructions are also compiled from the model into the contract implementing the data perspective.

Bpmn interpreter implements the process execution logic. It keeps no information about any of the process perspectives, but queries data from other components during execution. The runtime handler keeps track of the process instances. The operation of the runtime handler is similar to that of the Caterpillar version 2.

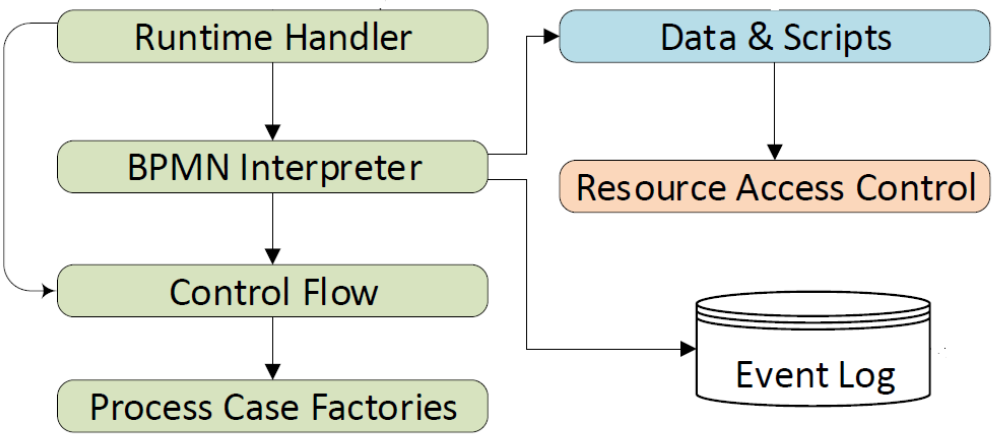

Fig. 5.5: General architecture of the V3 on-chain components

5.3.2 Smart contracts

Aside from the process registry contract discussed in the section regarding Caterpillar v2, the BPMN Interpreter contract is also deployed only once at the start of the application. It contains functions responsible for the execution of process, including execution of activities, throwing and catching events. It also provides an entry point for the off-chain part of the application to interact with the on-chain runtime, initiating most of the actions on it.

During contract instantiation it calls the appropriate contract imlementing IFactory interface, which contains logic relevant for creating process cases/instances. The IData contract is deployed for each process case/instance and is used by the interpreter to handle data perspective during the process element execution. It also keeps the information on the process state (started activities field) and a link to its parent in the subprocess hierarchy.

The IFlow contract is deployed for each process model and contains the data relevant to the hierarchy of subprocesses (each deployed IFlow contract is a node in a subprocess tree), control flow perspective, as well as created IData instances. It is also linked to the interpreter and uses its functionality for addition of newly created IData instances.

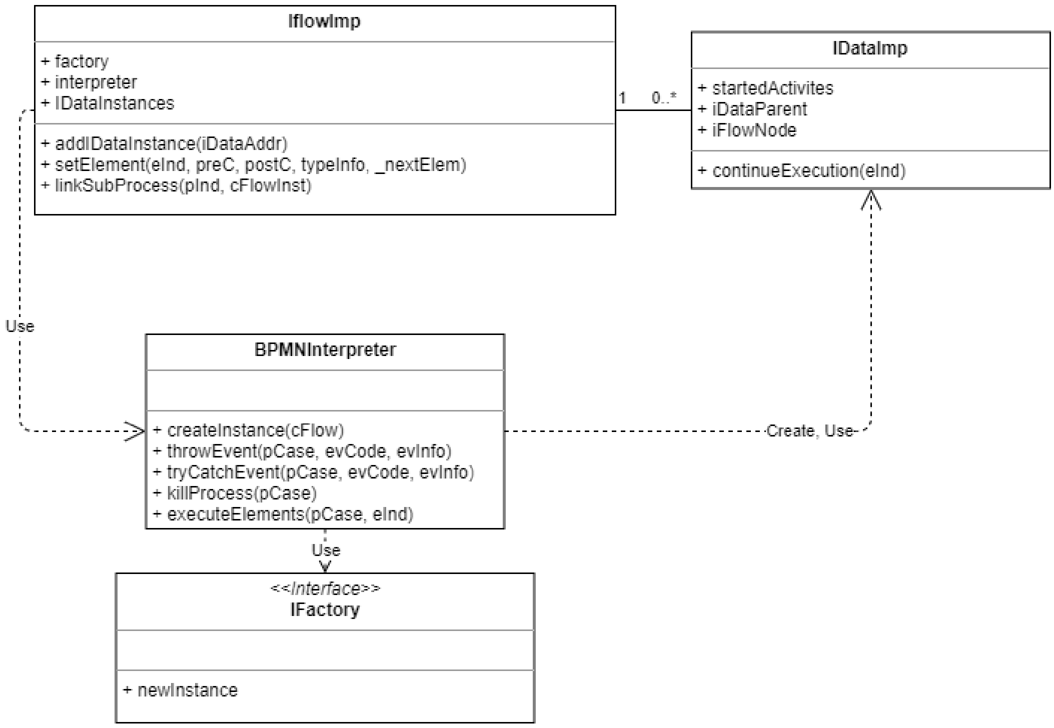

Fig. 5.6: Structure of the V3 smart contracts

5.4 Evaluation and comparison

In this section we compare Caterpillar and CHRYSALIS in terms of the features supported and project architecture, as well as evaluate the performance of Caterpillar and Chrysalis on the process deployment and instantiation

5.4.1 Comparison

Caterpillar supports much wider array of BPMN-features: Sub processes, User tasks, Script tasks, Service tasks, events, control flow decisions. Whereas CHRYSALIS only supports execution of plain tasks and evaluation of defferent types of gateways. The business logic layer of Caterpillar is implemented as a REST-service, whereas CHRYSALIS implements it as a library, which works to its disadvantage, since one should integratet it in some other application to use it, and also ensure the library's compatibility with this project, which has already lead to complications during development, when we could not integrate the enzian library, containing logic for interaction with Hyperledger due to the Hyperledger components' incompatibility with the Webpack version used in the web application.

In terms of runtime customization of the blockchain connection parameters CHRYSALIS has a significant advantage, since it allows the user to set the Ethereum node address and account for signing the transaction, whereas in Caterpillar all of these parameters are hardcoded into the application, which makes it less applicable for an actual production environment.

Process deployment and instantiation in CHRYSALIS are done in a single step, which means that for each new process case, one has to redeploy the whole model. Whereas in Caterpillar one can create multiple process instances for each model, thus reducing performance costs.

	BPMN-feature support	Business-logic implementation	Blockchain connection customization	Separate process instantiation
Caterpillar	Sub processes, User tasks, Script tasks, Service tasks, events, control flow decisions	REST-service	No	Yes
CHRYSALIS	Plain tasks, control flow decisions	Library	Yes	No

Tab. 5.1: Summary of comparison between CHRYSALIS and Caterpillar

5.4.2 Evaluation

As a base for evaluation a simplified version of the process model "Order to cash" from the Caterpillar repository was used. The simplification consisted of eliminating the features not supported by CHRYSALIS, to be able to compare the performance on the same model. As a result, the subprocesses were eliminated from the model replaced by simple tasks. This model is chosen to evaluate the performance on the most basic BPMN processes, namely activity execution and control flow handling with a parallel and merging gateway.

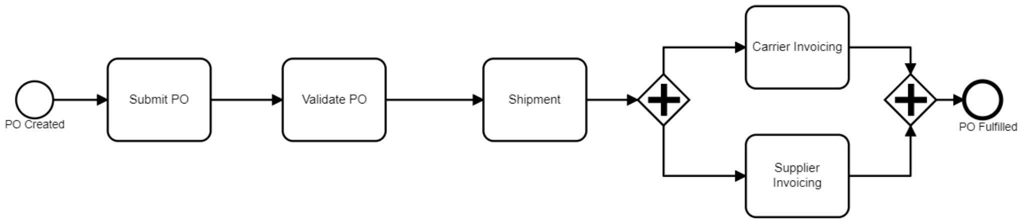

Fig. 5.7: Model used for evaluation

Gas consumption was used as a performance metric, the process deployment operation process execution were taken into account, since these are the most frequent and the most expensive operations, which dominate in the performance cost. In the result table deployment and instantiation aperations are counted separately for Caterpillar, but in essence they are equivalent to the CHRYSALIS deployment operation, which implicitly does the process case instantiation as well, so to correctly compare the costs of process deployment between Caterpillar and CHRYSALIS one has to take a sum of deployment and instantiation costs for Caterpillar.

	Deployment	Instantiation	Execution
Caterpillar V2	2015121	826808	~80834 p. task
Caterpillar V3	1390012	600116	~55005 p. task
CHRYSALIS	3638445	-	~116109 p. task

Tab. 5.2: Caterpillar and CHRYSALIS evaluation

As a result of the evaluation one can see, that the performance costs for the Caterpillar interpreter engine are significantly lower than those for Compiler engine of Caterpillar and CHRYSALIS, and that can be explained by the fact that most of the logic, including external entry points is handled by BPMN interpreter, which is deployed only at the start of the application and is not included in this evaluation. The deployed contracts contain only simplest logic for storing process-related information, and giving it to the interpreter upon request.

Compiler engine of Caterpillar seems to perform less costly than CHRYSALIS, but this result is influenced by the model used. Since Caterpillar v2 generates its contracts for each specific model, it can optimize those contracts for the specific model, whereas CHRYSALIS smart contract system is more universal and the same code is deployed for each model. That leads to Caterpillar being more efficient on deployment for relatively small process models, like the one used in the initial evaluation, but with the growth of the model the deployment costs will also grow, whereas for CHRYSALIS these costs will remain constant. To prove it, we have deployed models with number of tasks ranging from 5 to 30 with a step of 5, and measured the costs. The resulting measurements confirm our prediction - deployment costs for Caterpillar V2 grow in linear dependence with the size of the model, and deployment costs for Caterpillar V3 and CHRYSALIS remain constant.

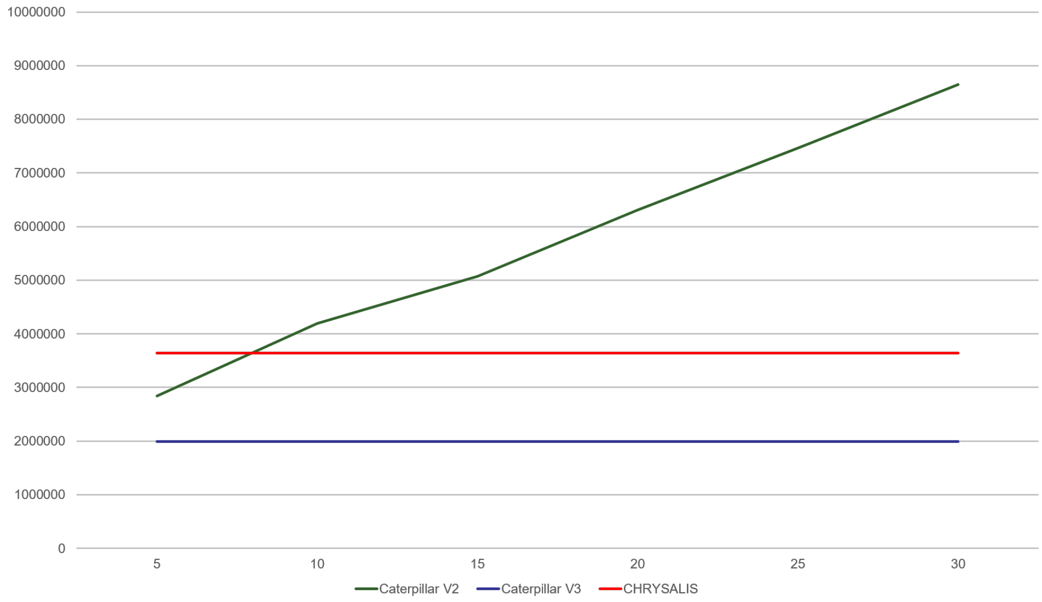

Fig. 5.8: Deployment costs in relation to model size

6 Resources

7 Conclusion

List of Figures

List of Tables

Declaration

Hiermit erklären wir, dass wir den vorliegenden Abschlussbericht selbständig verfasst und keine anderen als die angegebenen Hilfsmittel benutzt habe. Die Stellen des Abschlussberichtes, die anderen Quellen im Wortlaut oder dem Sinn nach entnommen wurden, sind durch Angaben der Herkunft kenntlich gemacht. Dies gilt auch für Zeichnungen, Skizzen, bildliche Darstellungen sowie für Quellen aus dem Internet.

Bayreuth, August 6, 2021

Philipp Scholz

TODO You can put your declaration here, to declare that you have completed your work solely and only with the help of the references you mentioned. Alternatively delete this and put your signature under the other declaration.

Bayreuth, August 6, 2021

Anatoly Obukhov