Free Proprietary Software Community

Nataliya A. Urakhchina
nurakhchian@berkeley.edu
https://github.com/CCT-token
https://github.com/CCT-token/free-proprietary-software-community
https://github.com/CCT-token/compiler-compiler-technology

Introduction

For the last decade, many technological advancements emerged.
A few, listed:

- Blockchain
- Artificial Intelligence
- Decentralized Internet
- Internet of Things

The hardware/software industry approach to building software products integrated in new hardware products hasn't changed much. In general, the approach is a combination of open source and proprietary software.

In 2017, we have seen a shift in the technological world. This year, we've seen ICOs surpassing the funding of regular VC funding, blockchain technology companies created by the thousands, and an urge to build decentralized internet applications based on blockchain technology in different application domains, including banking, educational, social networking, internet of things, almost everything. The success of those ICOs is driven by the desire of transitioning to decentralized development. Decentralization as it is presented by Bitcoin and Ethereum, means having a completely independent network running on the internet that is not controlled by any organization. Decentralization is an attempt to escape the corporate world.

The spread of decentralization is apparent and just like any technology we see, bottlenecks are not resolved, creating roadblocks for expansion and scaling. What Bitcoin and Ethereum build is decentralized, but the team who builds it, is in essence, a corporation, strictly centralized, having control over source code. Problems with open source and centralized development of open source were very well understood during the last decade and some solutions were proposed, including the DAO.

Within Ethereum, anyone can introduce a community with its own coin to be used as a digital currency to reward distributed app developers of that community. In general, that community creates open source software to be used by anybody. In this situation, there is a disconnect between software rewards given to developers and the software that they create.

Blockchain and Internet of Things, as a technology exist in a form of a variety of platforms, each platform is an extended traditional stack of protocols. To address hardware/software challenges, new decentralized internet protocols have been developed. Some of them are an attempt to build a completely new internet protocol stack from scratch. Unlike Blockchain or IoT, Artificial Intelligence, which is currently still is a collection of algorithms and big data environment to maintain AI processing results, has not organized itself together to reach a platform stage.

Interoperability, scalability, standardization, security, identity, content management are well known problems that have not been addressed completely.

<u>Current Blockchain solutions:</u>
Interoperability – No solution, one blockchain is not compatible with another.
Scalability – Not scalable at all, each node keeps its own version of the ledger that is replicated across all nodes.
Standardization – No solution.
Security – Revolutionary solution, based on proof of work.
Identity – Some solution.
Content management – No solution. Blockchain ledger is a custom form of a transactional database.

<u>Current Platform solutions:</u>
Interoperability – Usually, any platform interoperability solution is based on its platform API.
Scalability – Usually, it is provided in a form of platform cloud services.
Standardization – In many cases, a platform introduces its own set of standards asking other software/hardware manufacturers to adopt introduced standards

Security – Traditional approach based on encryption/decryption.
Identity – Some solution.
Content management – No generic solution.

The Free Proprietary Software Community (FPSC) is a new community to be built. First, we would like to introduce a modified version of open source software, called Free Proprietary Software. The corresponding Free Proprietary Software License (FPSL) is presented in appendix A. The FPSL is exactly the same as GNU GPL v3.0 for any software research and development. In accordance with FPSL, FPSC software is patent protected. As a result, all FPSC software that is distributed for profit, is under subject to a custom license agreement setting a fee structure for all involved parties. Currently, in Appendix A, only one patent is listed. As FPSC grows, more patents will be added.

FPSL is an attempt to address the main problem of the open source model: developers produce software that they no longer have ownership to and in result have no economic advantages. The idea for FPSL is that it will create conditions for developers to be directly rewarded for the software they create.

The Free Proprietary Software Community will be completely decentralized, including both the software products built and the software business run. To leverage the Ethereum platform DAO capabilities, we encourage FPSC members to create custom tokens for the corresponding software they build.

For bootstrapping FPSC ideas, the Compiler Compiler System (CCS) is used. The CCS is patent protected and provides advanced solutions for the problems discussed earlier: interoperability, scalability, standardization, security, identity, and content management. These will be discussed in detail, later in the paper. View CCS whitepaper in Appendix B.

The Free Proprietary Software Community will start with the Compiler Compiler Technology Token (CCT Token) to bootstrap CCS. The C++ version of the Compiler Compiler System (CCS) is already developed and the corresponding papers, source code, and FPSL are published on GitHub (https://github.com/cctToken/compiler-compiler-technology).

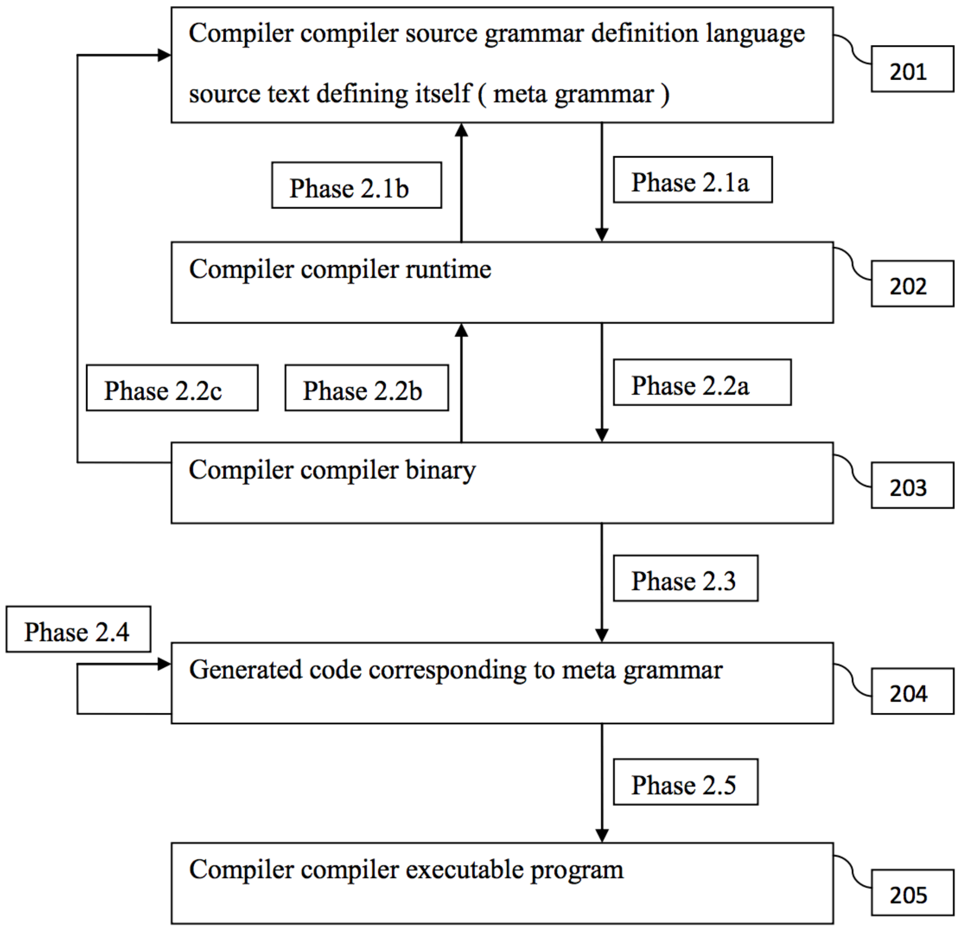
CCT Token is not a single token within FPSC. FPSC is considered as a collection of different groups and software products. They could be related to each other or can be completely independent. The CCT Token aspires to create a variety of other tokens. What makes those sub-communities with their own token to be a part of FPSC is sharing a common business model, that is based on FPSL and Ethereum usage. The idea is to create a decentralized network of developers.

It is well known that an average software developer across different software industries, produces about three lines of code per day and this very low number hasn't changed for the last 60 years. It is also well known that some software developers can produce hundreds of times more than an average developer. We believe that FPSC is a potential solution to change that reality, further helping to resolve the hardware/software growth mismatch.

1 Compiler Compiler System

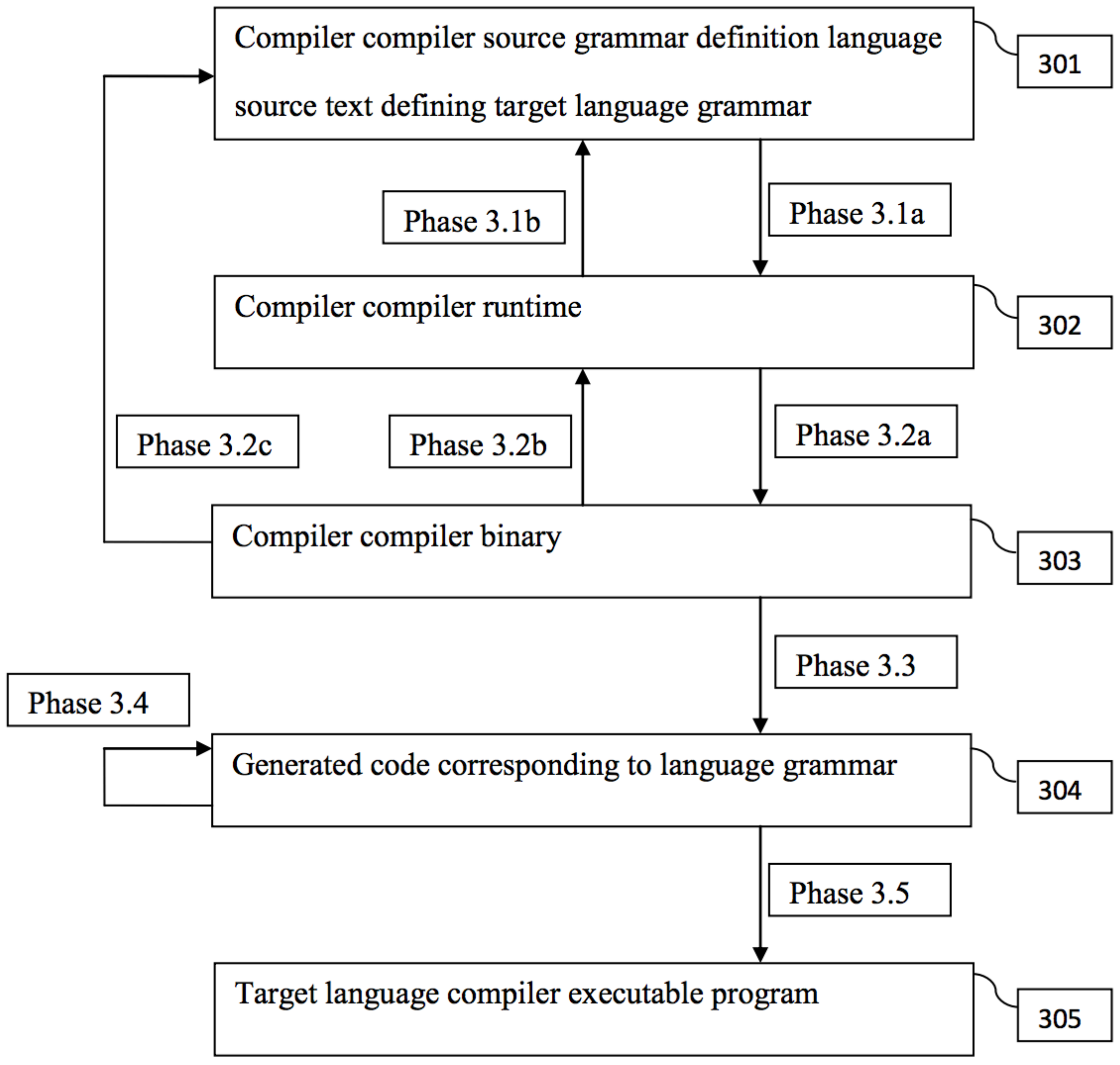
The Free Proprietary Software Community serves software developers and engineers. As mentioned before, the Compiler Compiler System will be used to bootstrap FPSC ideas and essentially will be a sub-community within the FPSC. The CCS is an example of how sub-communities can be formed and how their products can be distributed through tokens or fiat currency. All related CCS products will be connected with the CCT Token. Any other sub-communities created, either a sub-community that works with IoT solutions or AI solutions etc., can create their own token. This setup creates a software developer community where developers will be incentivized by sharing their code through tokens (or fiat). Every software created within the FPSC, is considered Free Proprietary Software, which will enable developers to share code for research and development for free but once they start selling for profit, you would have to pay the developer with these tokens or pay with fiat money. The next sections describe how FPSC can grow at a large scale by using the Compiler Compiler System.

1.1 CCS Architecture

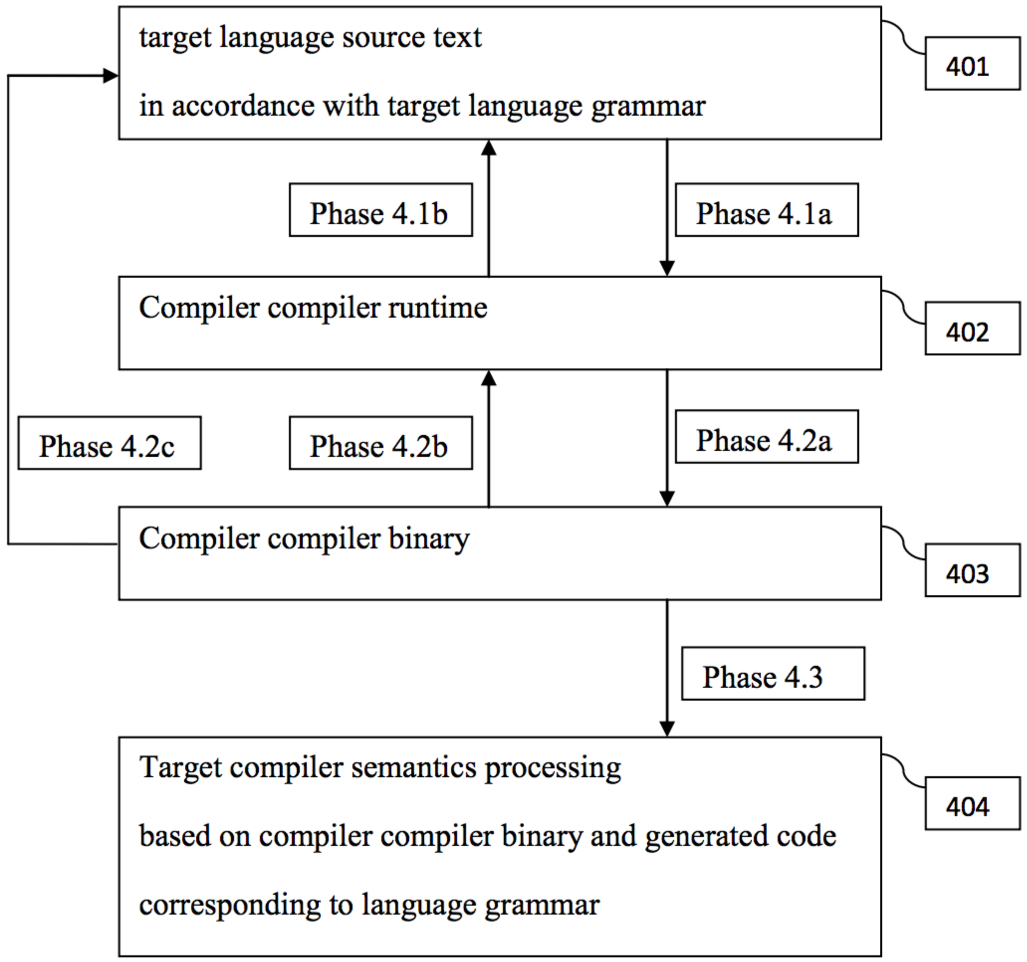


Compiler compiler source grammar definition language source text defining itself (meta grammar)
201
Phase 2.1b
Phase 2.1a
Compiler compiler runtime
202
Phase 2.2c
Phase 2.2b
Phase 2.2a
Compiler compiler binary
203
Phase 2.3
Phase 2.4
Generated code corresponding to meta grammar
204
Phase 2.5
Compiler compiler executable program
205

The diagram shows the present invention, The Compiler Compiler System, and its phases for building itself.


Compiler compiler source grammar definition language source text defining target language grammar
301
Phase 3.1b
Phase 3.1a
Compiler compiler runtime
302
Phase 3.2c
Phase 3.2b
Phase 3.2a
Compiler compiler binary
303
Phase 3.3
Phase 3.4
Generated code corresponding to language grammar
304
Phase 3.5
Target language compiler executable program
305

Shows Compiler Compiler System phases according to the invention for building a target compiler.


target language source text
in accordance with target language grammar
401
Phase 4.1b
Phase 4.1a
Compiler compiler runtime
402
Phase 4.2c
Phase 4.2b
Phase 4.2a
Compiler compiler binary
403
Phase 4.3
Target compiler semantics processing
based on compiler compiler binary and generated code
corresponding to language grammar
404

Shows target compiler phases when a target compiler is built by the Compiler Compiler System according to the invention.

2 Compiler Compiler System Solutions

2.1 List of Problems in Industry (which CCS has solutions to)

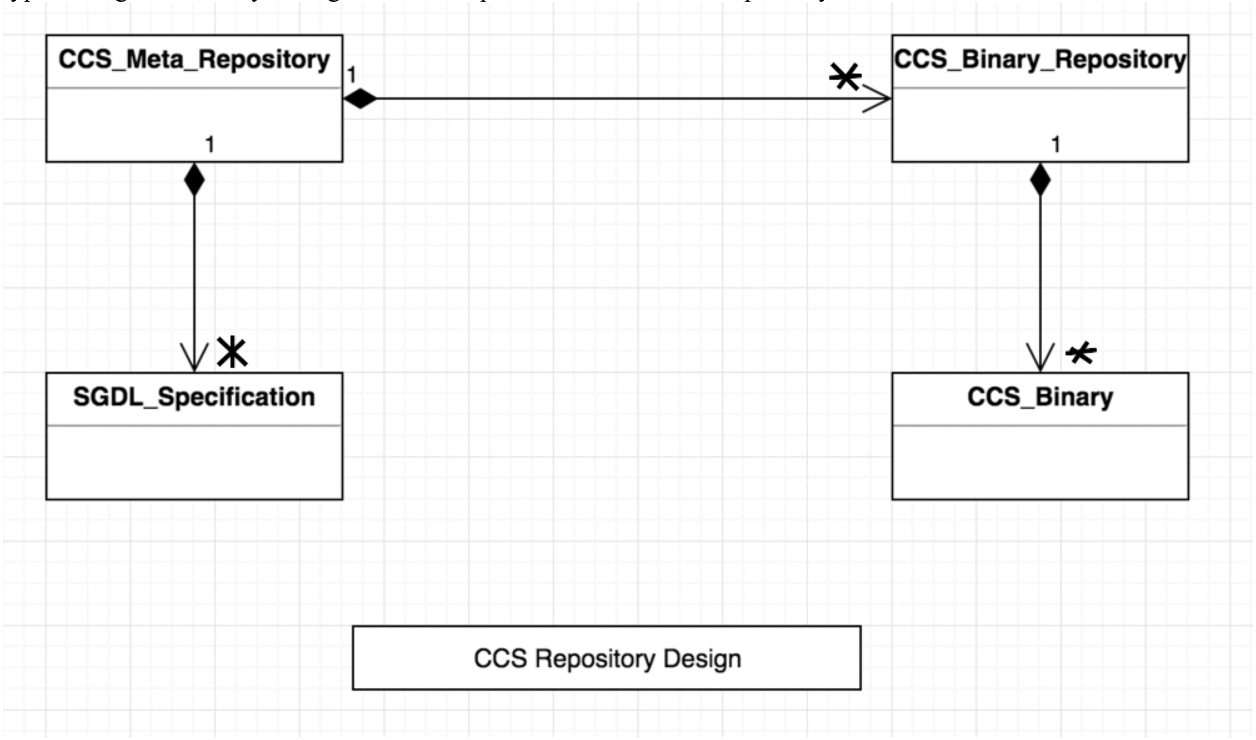
- Standardization
- Interoperability
- Scalability
- Content Management
- Security Based on Obfuscation

2.2 Repository

CCS repository acts as NOSQL database management system.

There are two components within the CCS Repository: CCS Meta Repository and CCS Binary Repository. The CCS Meta Repository acts as a registry for any language that has CCS SGDL specification. The CCS Binary Repository acts as a storage where any CCS binary for any registered CCS SGDL specification can be maintained: saved, updated, deleted, retrieved. CCS repository can be implemented as a service on a cloud, or physical instance on a given host. Also, many instances of CCS Binary Repositories can be maintained under the control of the CCS Meta Repository.

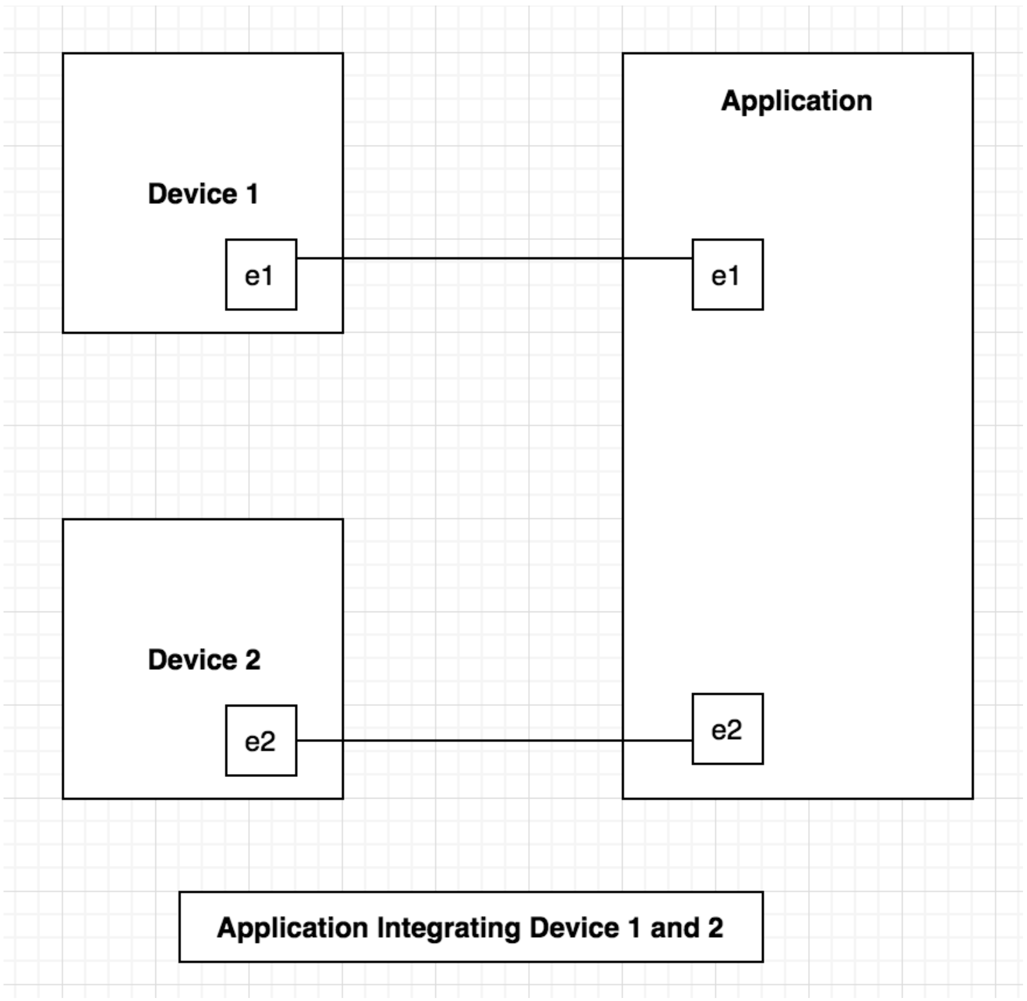
We will discuss in further detail below but mainly, CCS is capable of providing scalability and interoperability solutions based on CCS Repository and CCS Light Router that supports only one message type having CCS Binary for a given SGDL specification and CCS Repository ID.



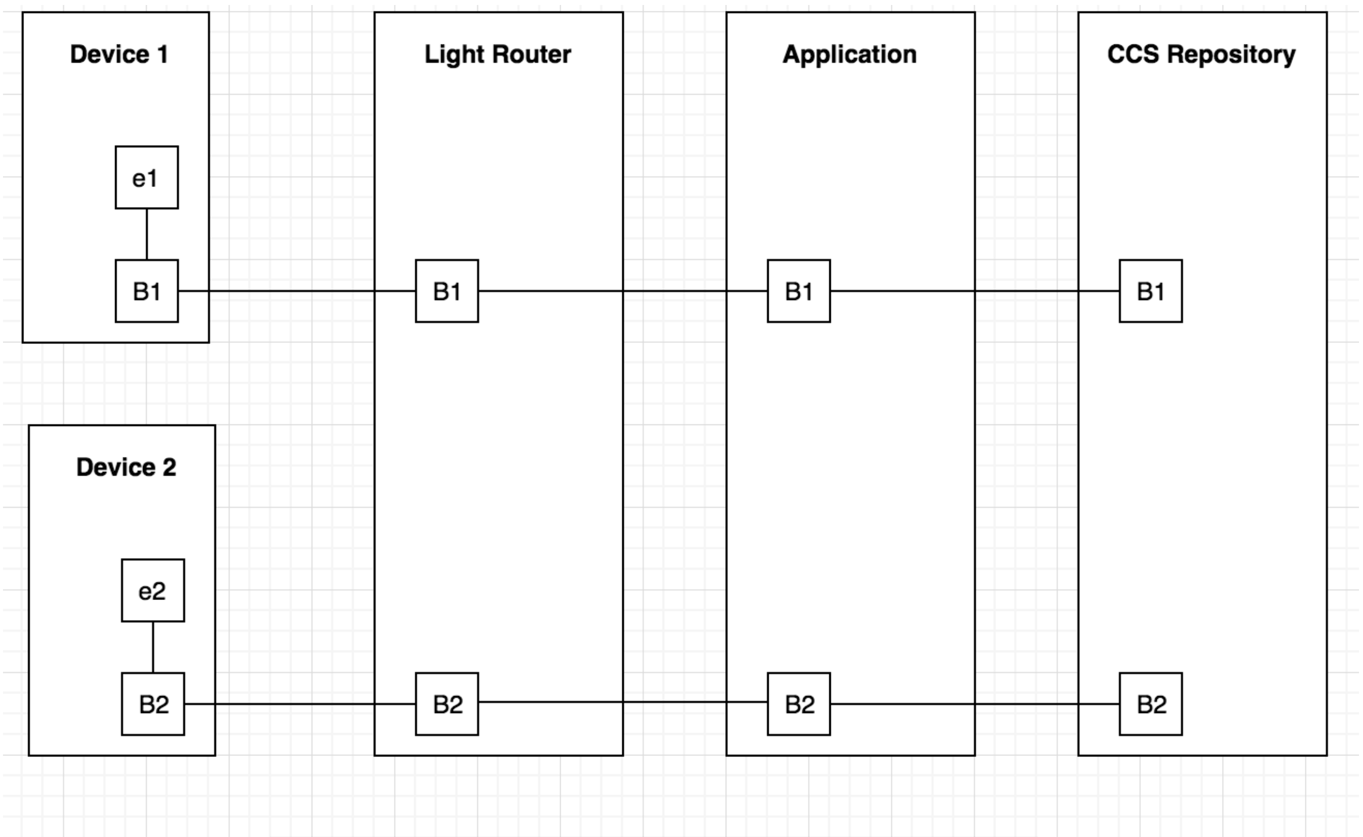
In the presented UML diagram, CCS Binary corresponds to a large scope identified by the corresponding SGDL. In general, CCS Binary may represent any oriented graph or another data structure with custom entities and their relationships.

2.3 Light Router

A well known industry solution for distributed application integration is based on a message bus. The traditional idea of integrating two applications with different end point types is based on the ability of having a router that supports both endpoint types and provides end point transformations from one point to the other. This approach in general, makes the router heavy with multiple endpoints and applications. In case of the CCS, CCS Binary is a final representation of a data processing chain and in this case, CCS Light Router can be implemented with a single endpoint type that is a CCS Binary endpoint.



In the diagram above, Device 1 talks to the application by means of the endpoint, e1, and Device 2 talks to the application by means of the endpoint, e2. Device 1 endpoint e1, is understood by the application's endpoint e1. The maintenance burden at the application level to support multiple versions of the introduced endpoints is a heavy task in terms of time and cost. For large applications, this scenario creates a well known issue called fragmentation. The idea of CSS Light Router combined with CCS Repository is capable to provide a better solution for integration. The solution is presented in the figure below.



Light Router Solution for the Integration of Device 1 and 2

In the proposed diagram, Device 1 and Device 2 are connected with an external world by means of CCS Binary1 (B1) and CCS Binary2 (B2). Traditionally, any endpoint defines a collection of messages. It is not a problem to create, for any given collection of messages, a corresponding CCS SGDL specification that would be an equivalent representation of that collection of messages. The Light Router accepts only one type of endpoint, processing CCS Binary, interacting either with Application or CCS repository directly. In the diagram above, CCS Light Router talks to the Application and that Application submits the results to the CCS Repository. Keep in mind, the proposed solution allows application development based on CCS Binary API only (only one endpoint!). This way, dependencies with Device 1 and Device 2, with custom manufacturers of software, is eliminated.

2.4 Standardization

Traditionally, standardization comes to place every time when multiple hardware/software vendors have to create consumer products. Traditionally, it is done by specifying standards as a formal specification, defining interchange formats. In general, an interchange format is enough to integrate hardware/software components from different manufacturers. But, in practice, having different versions of the same standard implemented by different vendors in form of their own APIs into a consumer product is a real challenge not only as a final cost of a product but also a huge issue maintaining that product in a life cycle. A well known software fragmentation issue is a real challenge.

Unfortunately, in this standardization reality, the situation gets worse for software/hardware manufacturers forcing to implement their own software to support the given standards. Usually, any standard is a collection of some data models specifying the data layer in a formal way as well as specifying the meta data layer. Having data models defined under the standard specification, software/hardware manufacturers have to have their own software implementations of those data models. The process of transforming standard specification into software products is always per software manufacturer and it is extremely expensive. Any automation for the process of transforming standard specification into software products may reduce those expenses. In this standardization reality, only big software/hardware companies receive real benefits enabling them to compete with the market. The compiler compiler system provides a solution for automating the standard specification transformation into software products. This approach reduces the expenses for implementing standard specifications in hardware/software products.

The C++ Compiler Compiler System generates C++ code for front-end, parser, syntax-controlled runtime API and syntax-controlled binary API. The front-end is an executable program that provides compilation from source into syntax-controlled binary. During this process, parser builds syntax-controlled runtime that is formally converted into syntax-controlled binary. Also, backward operations from syntax-controlled runtime and syntax-controlled binary into the source program are provided. The business logic operations can be defined by means of syntax- controlled binary API. This way for any given standard there is an option to use CCS syntax- controlled binary API instead of creating custom implementations of custom APIs by different software/hardware providers.

2.5 Interoperability

Industry created universal approach for interoperability between different applications by having interchange format approach. Any interchange format can be defined in form of CCS SGDL specification with subsequent automatic compilation of that CCS SGDL specification into related interchange format front-end, syntax-controlled binary and runtime APIs. Note, that , syntax-controlled binary API can be implemented for any programming language and any software platform including Unix/Linux, Microsoft .NET, Java, GO, etc...

CCS is capable to provide interoperability solutions based on CCS repository and light router that supports only one message type having CCS binary for given SGDL specification and CCS repository ID. In the Light Router section above, possible solutions for interoperability were discussed based on CCS Light Router, CCS Repository in more details.

2.6 Scalability

“Scalability is the capability of a system, network, or process to handle a
growing amount of work, or its potential to be enlarged to accommodate that growth".
The most well known scalability solution is the internet itself. Modern cloud solutions with decentralized architecture provide another approach for scalability.

CCS is capable to provide scalability solutions based on CCS repository and light router that supports only one message type having CCS binary for given SGDL specification and CCS repository ID.

2.7 Content Management/ Digitization

Content management is a process that takes information in any format and provides an environment for manipulating that data by means of programming. The idea of digitization of any business related information is vital because it is a source for better performance.

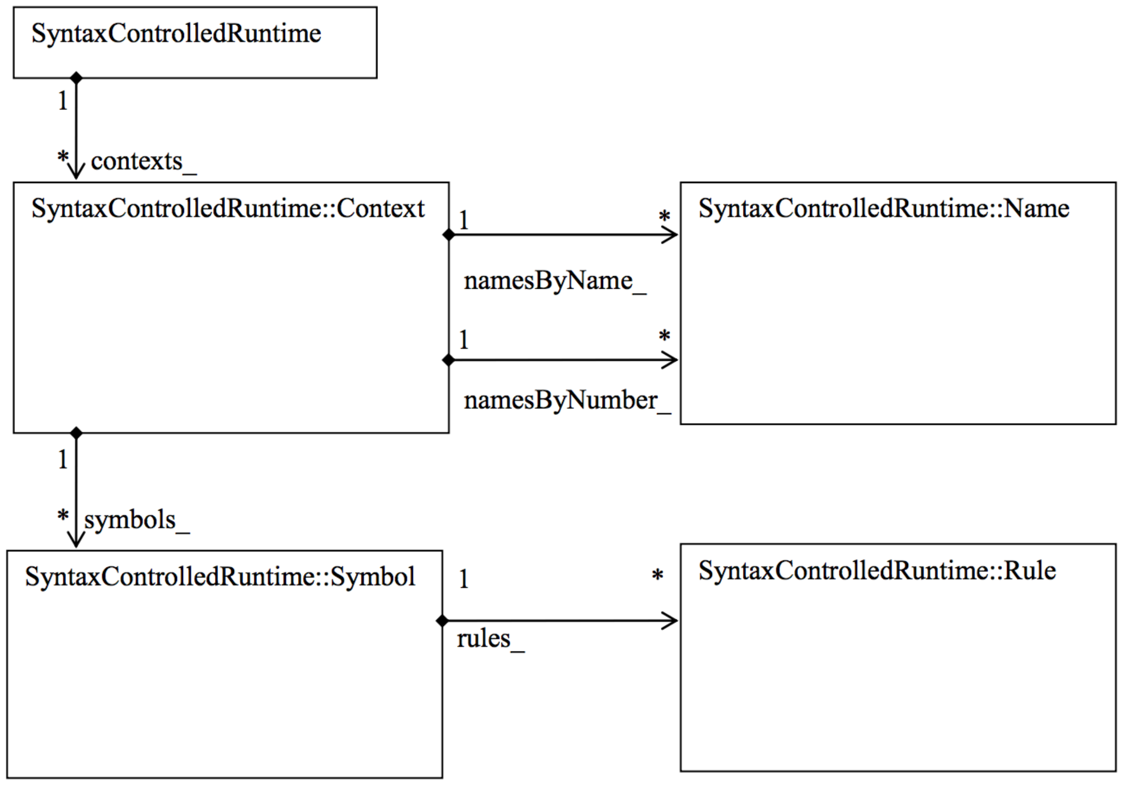
The Compiler Compiler System can be used for binary file processing. In this case, a corresponding binary file format has to be designed in the form of Compiler Compiler Source Grammar Definition Language specification. After that, a custom convertor from binary file format into Compiler Compiler Runtime format is implemented. Having a Compiler Compiler Runtime built for a binary file with a given format allows all other Compiler Compiler phases to work automatically without any extra code development.

Traditional text files and/or text specifications on some custom languages including any form of XML, interchange formats, other markup languages, etc. can be transformed by designing the corresponding SGDL specifications, building corresponding front ends and converting those text representations into CCS binary. Keep in mind that CCS Runtime and Binary APIs (completely automatically generated from SGDL) provide a foundation for content management solutions.

2.8 Security Based on Obfuscation

Obfuscation is a method that transforms data in a way that the original data elements and their relationships become modified to hide their original content, i.e., to be obfuscated. The key component for obfuscation is to provide an efficient algorithm for removing the introduced new items and new relationships to completely restore the original content, i.e., providing de- obfuscation.

It is known that grammar recognition tasks are NP-complete. In simple words, NP-complete means that the task would run almost forever. Having CCS syntax-controlled binary, the task of the grammar recognition is NP-complete. Having CCS syntax-controlled runtime, obfuscated one way or another, with subsequent formal conversion into CCS syntax-controlled binary would make grammar recognition tasks even harder. Implementing obfuscation/de-obfuscation algorithms for CCS syntax-controlled runtime and binary is an advanced, patent protected way of transmitting data with built-in security features and content management. Keep in mind that patent describes any binary file transformation into CCS syntax-controlled runtime including video/audio streams. Having all forms of data transformed into CCS syntax-controlled runtime with a well defined logical syntax structure generic obfuscation/de-obfuscation algorithms can be implemented to provide security solutions.

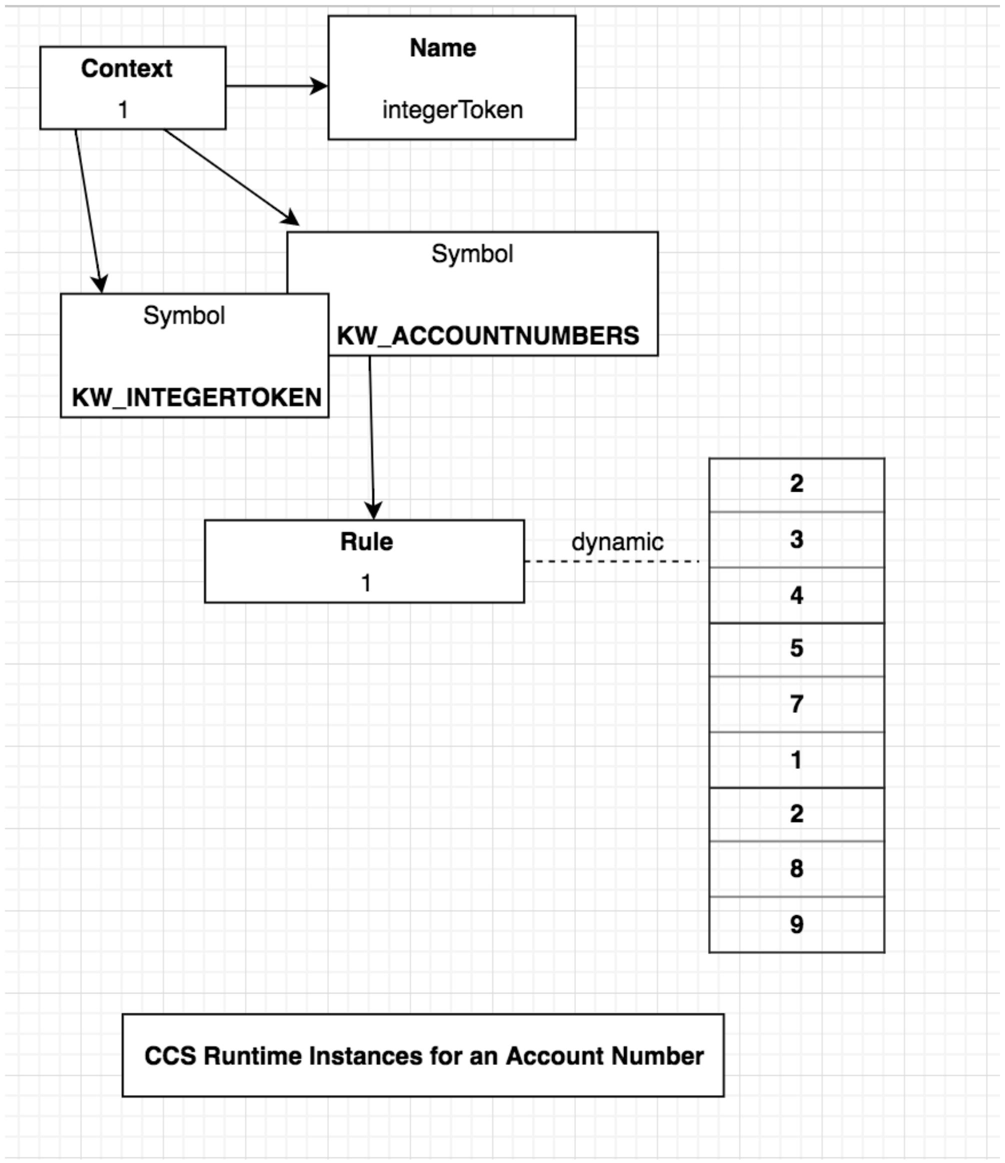

SyntaxControlledRuntime
1
*
contexts_
SyntaxControlledRuntime::Context
1
*
namesByName_
1
*
namesByNumber_
SyntaxControlledRuntime::Name
1
*
symbols_
SyntaxControlledRuntime::Symbol
1
*
rules_
SyntaxControlledRuntime::Rule

Above is a UML diagram of Compiler Compiler Runtime SyntaxControlledRuntime class and its inner classes with their relationships regardless containers implementation details. Actually this figure defines compiler compiler runtime logical view as a collection of individual context instances. Logically each compiler compiler runtime context maintains its own collection of names ordered alphabetically or by sequential number with direct access by name or sequential number. Logically, each compiler compiler runtime context maintains its own collection of symbols ordered by symbolID with direct access by symbolID. Logically, each compiler compiler runtime symbol maintains its own collection of rules representing each rule invocation during parsing for a given symbol. Actually, compiler compiler runtime logical view and compiler compiler binary logical view are the same since compiler compiler runtime and compiler compiler binary are interchangeable. However, compiler compiler runtime is designed to be an effective environment for processing parsing results during parsing itself, and compiler compiler binary is designed to be an effective environment for processing final parsing results in read only mode serving as a multiplatform interchange format. In other words, the figure above shows one form of compiler compiler parsing model with entities such as Context, Name, Symbol, and Rule with their relationships.

Let’s consider a simple use case to demonstrate our obfuscation ideas.
Imagine you have the following SGDL:

```
(account
   (accountNumbers ::=
            0 =" METAACTBEG();"=
            { integerToken }
            0 =" METAACTEND();"=
    )
)
```

This is a simple form of representing a credit card number in a sequence of integerToken values. In this use case, we will describe CCS Runtime for some input examples, one strategy for doing obfuscation and description of CCS Binary processing, extracting directly account integer numbers.



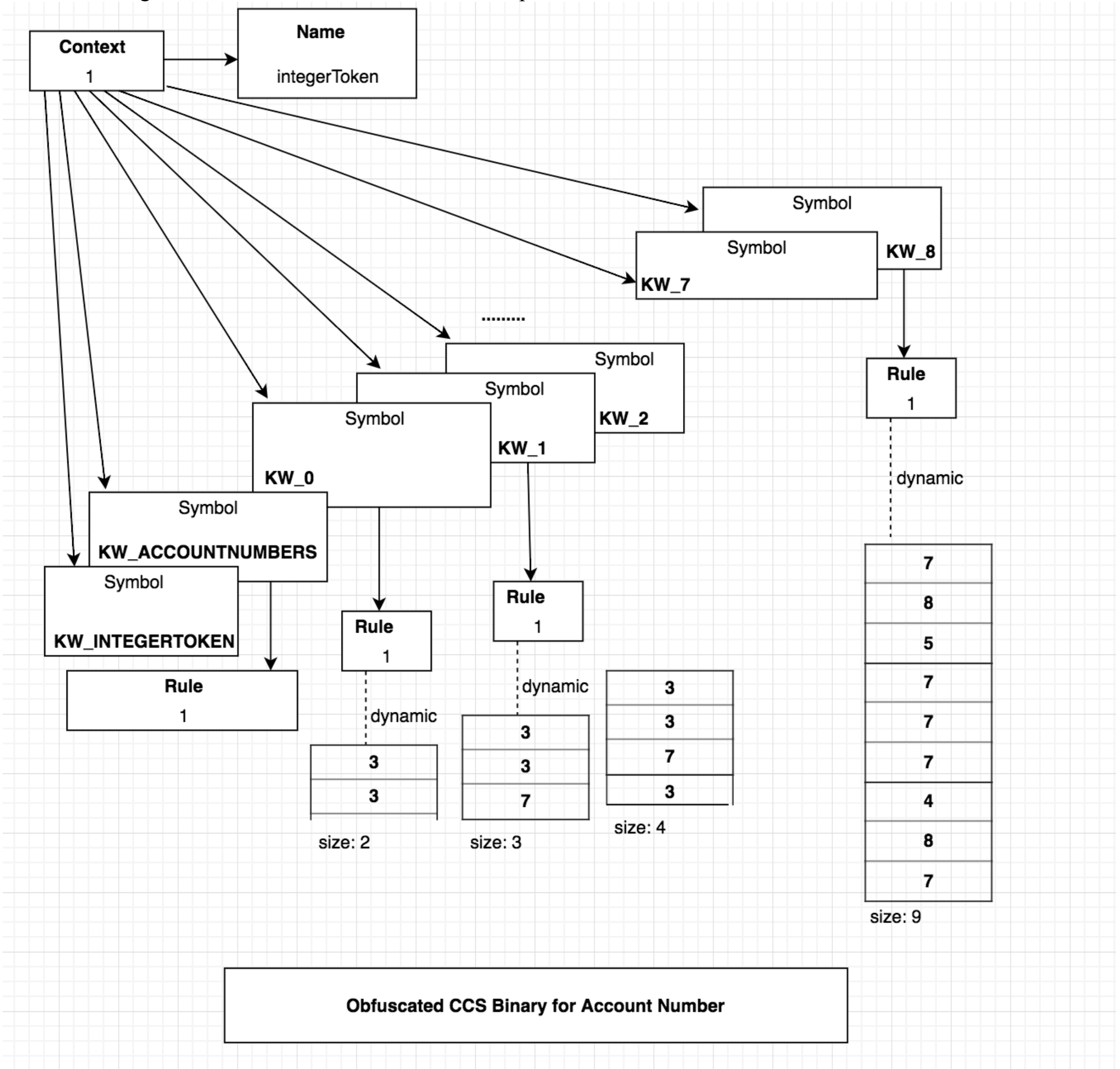
The diagram above depicts CCS Runtime Instances for an Account Number. Let our account number be represented as follows: 2 3 4 5 7 1 2 8 9.

When *account* front-end compiles text containing that sequence of account digits, it creates CCS Binary. It includes one instance of Context and default Name instances. Only one instance of Name is shown, representing integerToken. The Context instance owns two Symbol isntances shown with ID KW_INTEGERTOKEN and ID KW_ACCOUNTNUMBERS. The KW_INTEGERTOKEN Symbol represents integerToken as a terminal symbol. The KW_ACCOUNTNUMBERS Symbol represents a non-terminal accountNumbers. The KW_ACCOUNTNUMBERS Symbol instance owns a single Rule instance with ID 1 and that Rule instance contains only a dynamic vector representing an iteration of integerToken instances of the defined grammar rule:

```
(accountNumbers ::=
        0 =" METAACTBEG();"=
        { integerToken }
        0 =" METAACTEND();"=
  )
```

In the diagram, that dynamic vector shows all account digits, i.e., 2 3 4 5 7 1 2 8 9.
Note that the CCS Parsing Model defines both CCS Runtime and CCS Binary. When the presented CCS Runtine is converted into corresponding CCS Binary, there is an opportunity to process CCS Binary using CCS Binary API following the same logic: access Context instance, access KW_ACCOUNTNUMBERS Symbol instance, access its Rule instance, and retrieve account numbers from the Rule dynamic vector field.

In the next diagram, we will describe the Obfuscation process.



In the diagram above, Context instance, Name instance, KW_INTEGERTOKEN Symbol instance, KW_ACCOUNTNUMBERS Symbol instance have the same meaning as the original CCS Runtime. The first step of the obfuscation algorithm is to create fake Symbol isntances for each digit in the original account number. This table below represents a mapping from an account digit to a fake symbol ID.

2	3	4	5	7	1	2	8	9
0	1	2	3	4	5	6	7	8

So, KW_0 represents the first account digit, 2, KW_1 represents the second account digit, 3, etc. In the diagram above, KW_<ID> owns its own Rule instance and each instance allocates a dynamic vector member with the size of the pertaining account digit. So, when the obfuscated CCS Runtime for this use case is converted to CCS Binary, that CCS Binary representation would be obfuscated versus the original one, because the obfuscated representation has those fake symbols where the original does not. Even more, account digit numbers are represented as the length of the related dynamic vectors, of the Rule instances, of the fake Symbol instances. To restore the original account number as a collection of account digits from the obfuscated CCS Binary, we have to execute the following for_each loop: process all fake Symbol instances, for a given Symbol instance get access to its Rule instance, for that Rule instance extract the dynamic vector size, and finally append it as an account digit to the generated account number container. Our obfuscation/de-obfuscation process is then complete with the original account number restored.

Keep in mind, this way to extract the original account number, we don't have to execute any expensive de-obfuscation algorithm. The only expense we have in this presented use case is memory overhead. It is noteworthy to add that our obfuscation algorithm can be combined with the traditional encryption/decryption scheme, which dominates the current industry's approach to build security solutions. The CCS Runtime and CCS Binary are universal/generic, making it have the opportunity to build security solutions common to as many use cases as the current traditional encryption/decryption scheme. The CCS Obfuscation based on CCS Runtime transformation cannot be worse than the current security scheme, but because of the complexity of security issues, the security solution based on Obfuscation is a subject of research and development.

3 FPSC Bootstrapping

In this section we will describe the potential work that can be done by FPSC in form of a software/business bootstrapping. The software bootstrapping method usually is affiliated with compiler development. When you have to implement a compiler for some new language, the traditional first step is to implement that compiler for the new language, using the existing language. E.g. the first version of the GO language compiler can be implemented using plain C. After that, one can implement the GO language compiler, using the GO language itself. This is a form of compiler bootstrapping, having the final version of the compiler, implemented by means of itself. CCS is based on bootstrapping, because there is a SGDL defined in SGDL and CCS can compile SGDL and build CCS front-end automatically. Software bootstrapping is a powerful method because the software you build reuses its own facilities in the earliest stages. The quality of software that is bootstrapped is trivial: it uses itself and this means it is tested before it is delivered to users.

Business bootstrapping is a well known technique with various advantages. Regarding FPSC bootstrapping, it entails the idea of expanding the community in various ways. One direction is based on Compiler Compiler Technology. The tasks to be completed in that area are depicted in the form of tiers: tier I., tier II., and tier III. CCS software products. Another direction is Ethereum capabilities to build smart contracts with dedicated coins. For CCT related development, there is a plan to create a CCT Ethereum token to be used within FPSC to sell/buy built software products paying with the Ethereum CCT token and with other Ethereum tokens introduced within FPSC. There are two important factors that bring together the FPSC: Free Proprietary Software License (FPSL) and the incentives that FPSC members issue to each other for licensing their software products using various Ethereum tokens. It is important to distinguish that FPSC is not a community comprised of only software products derived from CCS products.

The tier I. is comprised of CCS versions for Java, C#, GO, etc.; a unified multi-platform, multi- language versions of CCS syntax-controlled binary as a collection of compatible CCS syntax- controlled binary API versions for different software platforms and programming languages.

The tier II. is comprised of any compilers (front-ends) built by means of any CCS from tier I.

The tier III. is comprised of any software built by means of CCS runtime and binary APIs including obfuscation algorithms, interoperability tools, etc... By software platform here we mean Unix/Linux platform with C/C++ and other languages, Microsoft .NET platform with C# and other languages, Java platform with Java language, GO language as an independent from Microsoft .NET, or Java, or Linux. Also, the CCS Repository belongs to the tier III. category.

3.1 CCS Tasks

Currently the CCS exists in two versions: C and C++. Also, a few front-ends have been developed including generic XML, JSON, and Google Protocol buffers. The discussed CCS Repository and CCS Light Router have to be implemented. Also, CCS Binary has to be transformed into JSON format to be accessible from JavaScript in a form of two operations, i.e., CCS Binary to equivalent JSON specification and backward transformation from JSON into the original program text in accordance with target language grammar. This addition to compile/decompile operations provided by CCS is helpful to build web applications based on JavaScript. Also, keep in mind that traditional software platforms like Unix/Linux C, C++; Microsoft .NET, C#, C, C++, Java, Java, GO have defined APIs to process JSON. With this straight forward way, we can integrate CCS Binary with those traditional software platforms.

3.2 FPSC and Blockchain

The key components of blockchain platforms are:

- A distributed ledger to keep track of all transactions.
- Proof of Work (PoW) feature implemented as a security solution against malicious intruders and making a blockchain platform not just distributed but decentralized too. Other ideas including Proof of Stake are in discussion in the blockchain community.
- Smart Contracts. Eg., Ethereum has a smart contract language, Solidity.
- Replication of the ledger across all blockchain network nodes.
- Peer to peer network messaging.

Peer to peer network messaging and ledger replication creates a condition where scalability is not an option.

Compiler Compiler Technology is capable to provide blockchain scalability solutions the following way:

- Introduce a peer to peer messaging layer in a form of SGDL specification.
- Implement ledger based on CCS Repository.
- Instead of PoW, provide blockchain security solutions through obfuscation.
- Combine security solutions based on obfuscation with ledger represented in CCS Repository that is the replicated in all nodes. It is possible to implement other strategies for ledger replication but the main point is to avoid ledger replication in each node, since it is redundant and a blocking factor for scalability.
- Since CCS Repository is scalable and security solutions based on obfuscation can eliminate the necessity of having PoW, the proposed blockchain technology based on CCS is equivalent to the existing blockchain technology.
- Regarding smart contract languages including Solidity, it is possible to implement Solidity compiler and other smart contract language compilers based on CCS. This way, interoperability solutions between different blockchain networks will be granted from day one.

The most complicated area that is less researched is building security solutions replacing PoW based on obfuscation. It requires deep research and development but when it is finalized, scalability issues would not be an issue at all.

Major overall solution for Blockchain platforms: Solving scalability issue.

3.3 FPSC and AI

The growth and popularity of AI rapidly expanded and has become a significant topic in technology. Despite the growth, AI still exists in a form of some exclusive products promoted to the market by large corporations. During the last two decades, a lot different standards have emerged in AI marketplace. We believe that providing solutions for the existing AI standards, e.g., Ontology, based on CCT would be able to change the AI landscape solving interoperability issues.

Major overall solution for AI: Solving interoperability issue.

3.4 FPSC and IoT

For the last decade, IoT has become possible for many reasons, one is the tremendous growth of hardware capabilities. Another one is wireless network capabilities. Another is 5G capabilities including new options to run TV services. As a result, at least a hundred IoT platforms have been created in the last decade. As a result, at the consumer level, we have an issue of lack of interoperability between those platforms and corresponding products**. CCT is capable to deliver solutions for standardization, content management/digitization, scalability, security based on obfuscation and interoperability making it possible to integrate existing IoT platforms.**

4 FPSC Overview

The Free Proprietary Software Community is dedicated for software developers to perform their software development work in a decentralized manner without having traditional corporations. We named the community to depict the overall fact that it is a community based on free proprietary software. Community for software developers organized by sharing FPSL and having dedicated Ethereum coins for the software they created. Purpose of FPSC is to incentivize developers to do more work for themselves since they get paid directly for the software they created. Those payments can be performed though Ethereum coins, including CCT token for CCS products, or fiat currencies.

We are planning to create Ethereum CCT token first by running the standard Ethereum smart contract. We are not planning for traditional ICO since we do not need financing to start the described ideas since the C and C++ CCS have been developed. The CCT token would be used to bootstrap a blockchain like community based on Ethereum platform advantages for building DAO.

We also believe that institutional investors can speed up FPSC growth.

Appendix A:

Free Proprietary Software License (FPSL)

FPSL Draft v0.1

Free Proprietary Software License (FPSL) is GNU GPL v3.0 for software research and development within the Free Proprietary Software Community (FPSC).

The FPSC software products are patent protected. Currently there is US 8,464,232 patent, issued Jun. 11, 2013, "Compiler Compiler System with Syntax-controlled Runtime and Binary Application Programming Interfaces.", Aleksandr F. Urakhchin.

FPSC software products to be used for profit are fee based, having fee license agreement with FPSC members and any involved parties.

Appendix B:
https://github.com/CCT-token/compiler-compiler-technology/blob/master/RESOURCES-ccs-white-paper.pdf