Genesis World

IDE for distributed development of DApp and smart contracts

on new programming language

July 2017

The global IT solutions market has reached over 350 billion dollars. We live in an era of capitalism dominated by IT giants such as Microsoft, Apple, Google, Oracle, SAP, etc. The work is largely done by programmers, IT designers, systems architects and other binary content creators, but the fruits of their labor are owned by corporations. **This is a morally bankrupt, unequal system controlled by centralized organizations that serve as middlemen and take the lion's share of the profits.**

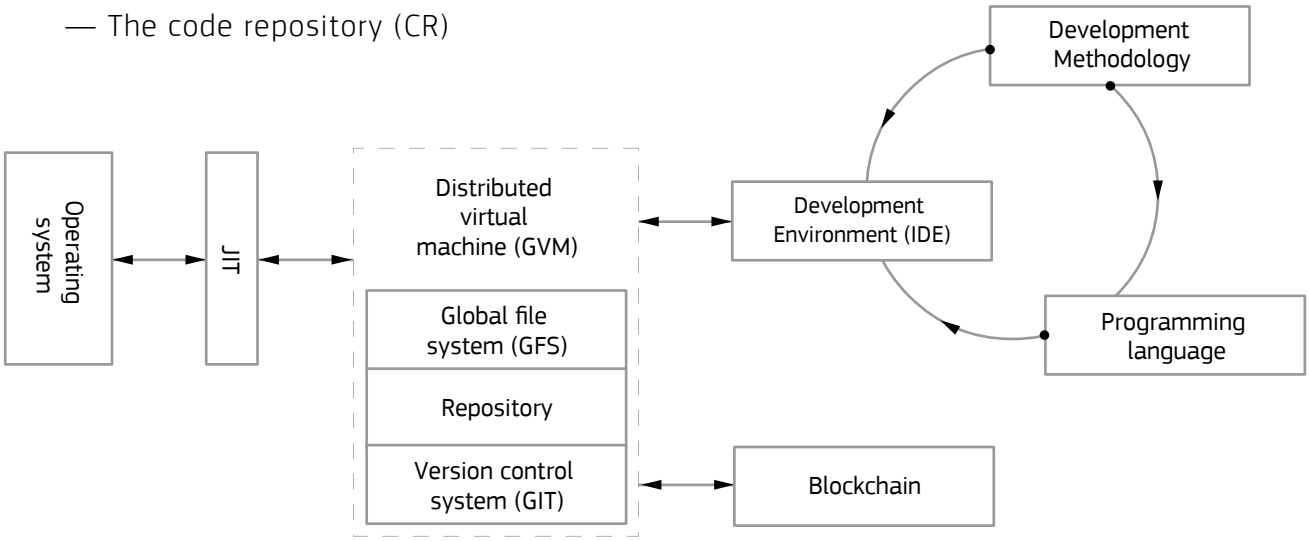
Genesis is a new paradigm of distributed real time labor that uses honest profit sharing between all creators of a product, cutting out the middlemen.

The Genesis platform is:

1. IDE for distributed development in real time
2. Permanent identification of a piece of code's author(s) using the blockchain
3. Automatic profit sharing for the author(s) of the code
4. The development technology
5. Automated analysis of software to check for vulnerabilities, contradictions in terms, interactions with other softwares, etc.
6. A universal programming language
7. A global file system
8. Roadmap
9. ICO

1. Distributed development in real time

The Genesis platform consists of:
— An integrated development environment (IDE)
— The virtual machine (VM)
— The programming language (PL)
— The code repository (CR)

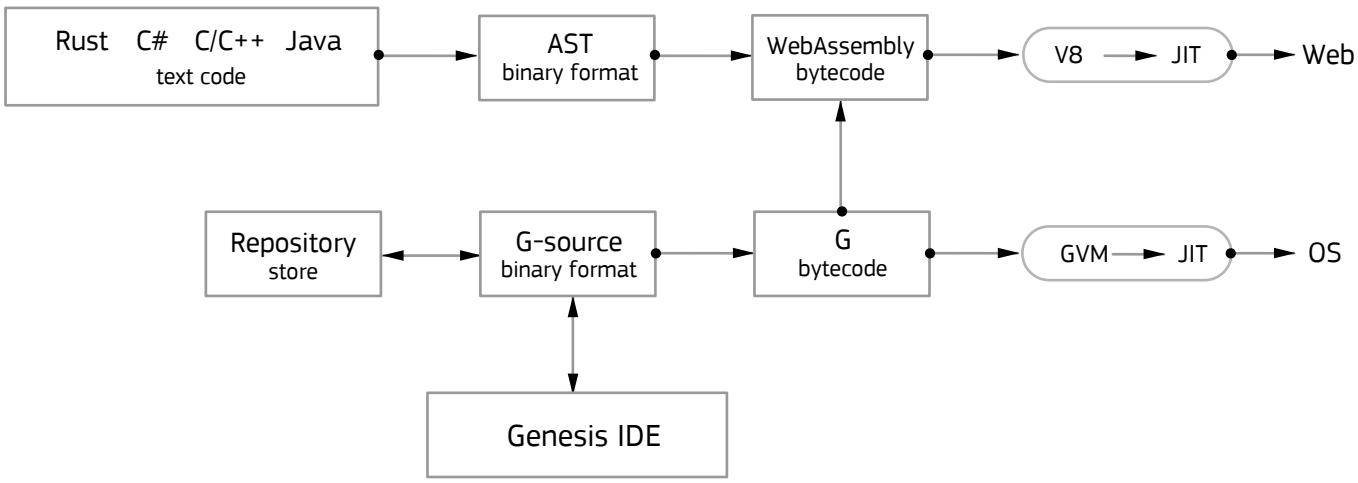


Schematic of the Genesis platform

Genesis is built using microcomponent programming. This gives anyone an opportunity to control individual services and link them, allowing for multiple visual representations of code and data. Genesis supports multiple simultaneous links between services; the IDE and VM can track these links during design and execution. The target software is heavily virtualized during the creation process and can be implemented in the hardware at any point. The exact moment of execution is determined by the developer; the software can be supported using flexible links until reaching a release state, or implemented as soon as a distribution is created or the software is installed. Genesis' flexibility allows constant conversions from byte code to machine code and programs can change/execute on the fly.

“Notepad development” becomes impossible, because the document describing the program code and data is a composite from the set of separately editable blocks. Program data can be effectively represented by tables and trees; function code is edited separately and does not affect the other parts of the file or program.

At any point during code editing (such as when typing or saving a file), the text code of the function is converted to a special byte code, which completely preserves the original text but is compact and interpretable. Interpretation of the byte code is needed for debugging and tracing the program; its purpose is to be effectively transformed into machine code with the help of specialized compilers. It is also possible to convert byte code to text (for example, if the program is written in C).



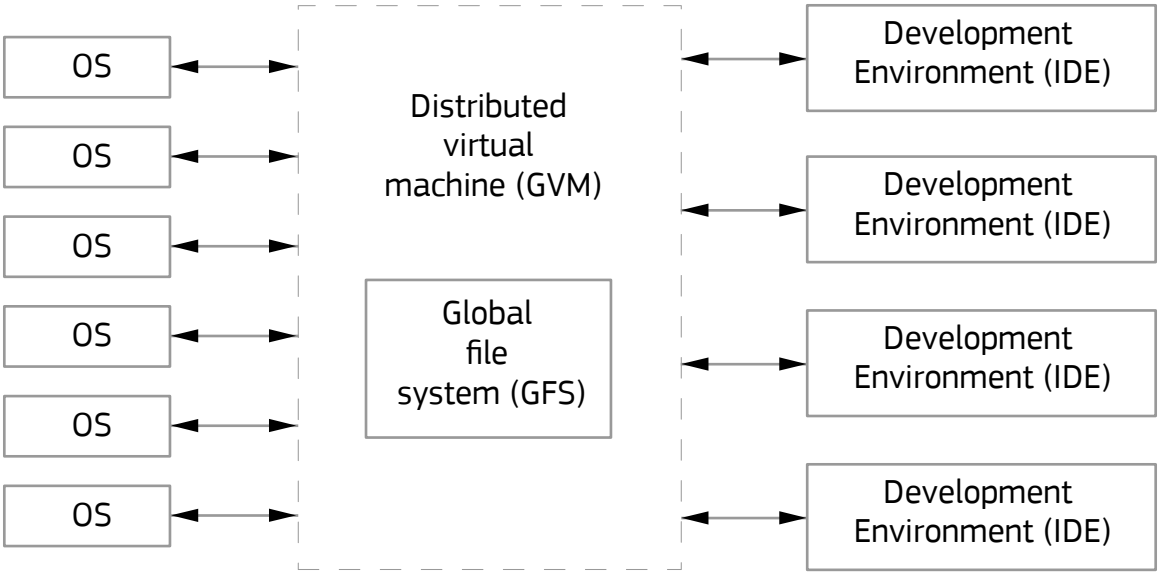
Genesis’ integration schematic

The master data and the IDE and VM formats are binary analogues of well known markup languages (XML, CSS, etc.)

When programmers work with the IDE, they are no longer using files in the classical sense of that term. Instead, they work with classes, objects and structures in their own development tree.

All of this data carries a global unique identifier (GUID), which allows creators to effectively share data and programs between different developer and user repositories.

IDE effectively allows programmer collectives, simultaneous group work on different parts of a program, and visualization of each other's work in real time (LiveCoding).



Collective work schematic

The IDE introduces special policies for this kind of collaboration. IDE allows developers to work on a program without having to restart it multiple times during development. The IDE can work on algorithms without the necessity to create a test application. An entire team can work on a single app instance. Multi-core applications that run on several devices at once are also possible. There is no need to separate out release and debugging applications; different parts of a program can be in different modes at the same time. Any developer can define their own breakpoints in a single copy of the running program, then fix the code and data in debug mode.

A distinctive feature of the Genesis VM and language is unlimited support of unitary data types. Developers working on hardware, emulators or virtual devices can all introduce such data types into the language. Existing VMs such as JVM, Dalvik, CLR and ABC all have very sparse unitary data support; this is a side effect of the need for fast interpretation of byte code, which is not a requirement in Genesis.

The language does not contain any defaults (they are left up to the OS or the compiler); all entities, such as the order of instructions or calculating the arguments of functions, are strictly single value.

The language introduces the concept of a phantom object. Such objects can be forcibly removed from the system. In some languages, this isn't possible (for example, Java, JS and AS3 require the destruction of all links, which can often lead to excessive memory use) or leads to a software crash (C/C++, when running into 'dead links'). Calling the phantom object method and accessing the object's data returns the data by default and does not lead to a crash; the syntax of the language notifies the programmer that the object reference type may be Phantom. The programmer then has some options:

— Preemptively test the reference to NULL;

— Do nothing and call the method of the object, returning no data but also no crash;

— Receive a notification about the forced destruction of the object by setting the callback function on the link.The control system may force this last step, if this is desired.

The language also has basic support for callback functions, streams, and parallel executions.

Horizontal inheritance in the object structure is also possible, along with separating out a class of entities or objects that are not required for a given application.

The language introduces the notion of "monitoring agreements". Different APIs often prescribe what needs to be done, but don't specify the resources that should be used to do it with. Monitoring agreements greatly raise the security of an application based on inter-component interaction (such as plugins and extensions of different programs and systems) and conform to existing architecture (for example, MVC, which is needed to build an effective terminal).

At the level of the language (visual entities in the IDE) we introduce entities that analyze currently running code, allow developers to quickly analyze it, verify it and turn off any unnecessary checks.

Global:
But, if necessary, the system also allows the user to write a simple script without strict typification and other strict checks. The IDE is capable of producing native code for any platform with “hard” links outside of the VM; for example, it can generate EXE, DLL and installation files for use within Windows. It can also work with lightweight objects without metaprogramming and simple sets of modules (such as libraries of subroutines.)

In order to remove reliance on rigid data sets and library methods, Genesis introduces new entities for universal algorithmization. These algorithms operate on data, outside of structures and functions and without being tied to a specific library, but they can also be linked to any or all of these after the fact.

The goal is universal code, in which all non-algorithmic concepts can be declared interchangeable. The developer may edit the template and fully see and debug the generated hard-bound algorithms. Such templates are very useful when data sets are formed from structures and objects that are of different types. These templates can create machine code directly upon program executing, even when input structures containing the required fields are unknown until the last moment.

Basic network interaction between developers is built using the P2P model. Written code is automatically distributed between the developer and user code repositories according to the given set of policies.

Open sourced code may be deep patched and referential structures may be replaced without asking the original developer, though he will be informed of the use and modification of that code.

Code sharing between developers and users is possible in several modes:

1. The author’s code: contains names, comments and documentation

2. System bytecode: in this case, unnecessary entities, including variable names, are discarded. However, the program can be recovered by following language instructions.

3. Byte code of the pseudomachine register (similar to Dalvik or LLVM). In this case, the original instructions are not recoverable and some reverse engineering will be required, which reduces the qualitative conversion to machine code (slightly decreasing performance.)

Global:
To pay for any reverse engineering costs, instructions for decoding the byte code can also be attached separately into the software protection system.

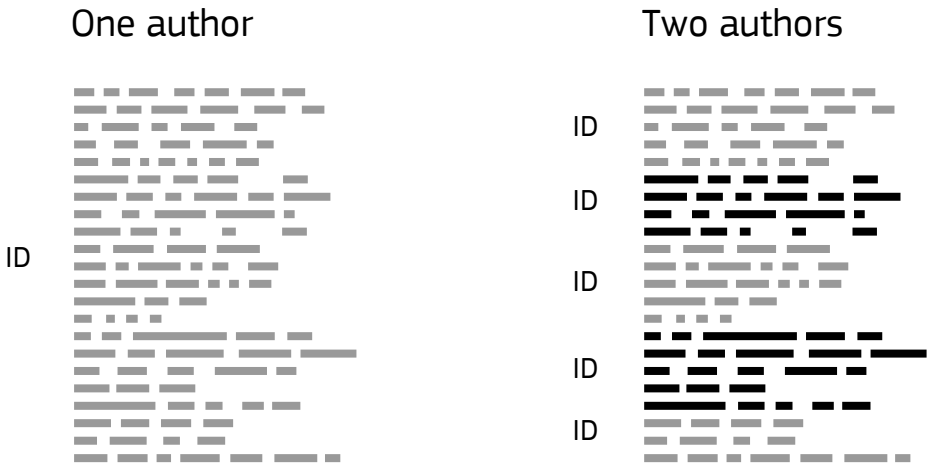
LiveCoding

Genesis provides LiveCoding with two way support from the IDE and the VM. Modules and classes are given a VMT (Virtual Method Table) and hot swapping of functions is possible on the fly. The IDE describes a class or program module in such a way that functions can be autonomously edited. Byte code is built on the fly at the time of the edit and can be instantly transferred to the executing program.

2. Code authorship and the blockchain

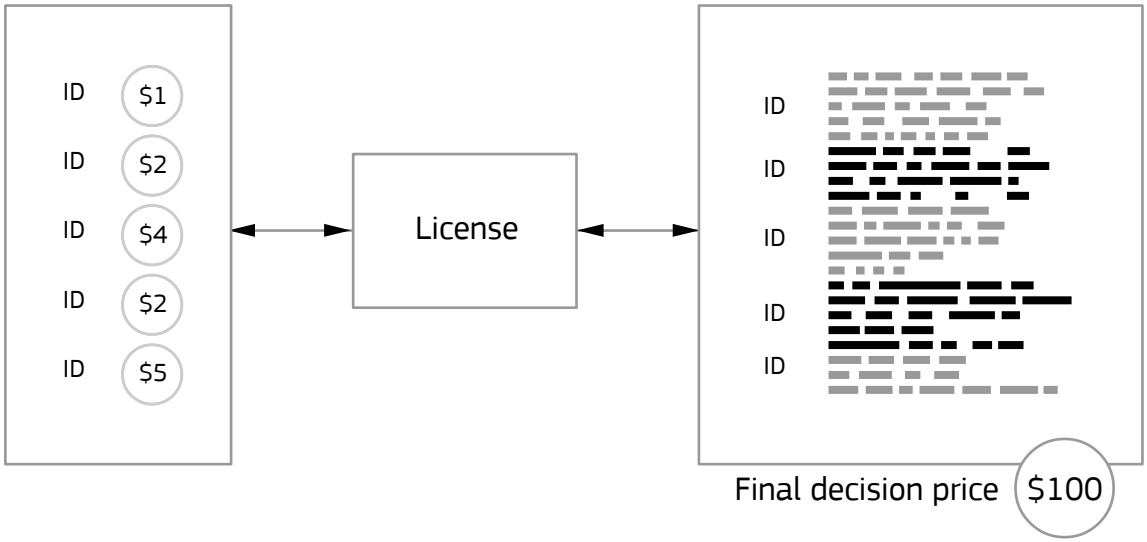
Using the Genesis GFS that serves as a code repository, along with its version control system that always stores all changes, Genesis can always determine when code was written or altered and the code's author. This allows us to permanently tag authors for all projects of all sizes on our platform.

The blockchain is used to store the tag / ID of the code, effectively serving as an automated patent bureau.



3. Automatic royalty payments to code creators

The proceeds that are paid out to the authors of the code are regulated by the code's development license. Software sales take place using an internal marketplace with Genesis tokens.



4. Development methodology transformed into technology

The term «technology» is shorthand for a set of organizational efforts, operations and methods aimed at building, servicing, repairing and/or exploiting a product of a nominal quality and optimal expenses, conditioned on the current level of scientific, technological and societal development.

The Genesis technology is aimed at:

— Developing the most appropriate solutions at a time of high uncertainty;
— Monitoring the execution of all decisions taken by the organization;
— Correcting any mistakes made during the execution;
— Continually raising the quality of the decision-making

Genesis use cases

Genesis may be used for:

a) Developing complex software and hardware systems aimed at:
— Automating management functions;
— Supporting decision-making managers and executives;
— Self-learning.

b) Creating formalised methods of management and decision making.

c) Analyzing weakly defined problems for contradictions.

Genesis solves:

a) Formalizing the areas of activity.

b) Integration and interaction problems:
— At the internal level;
— At the external level.

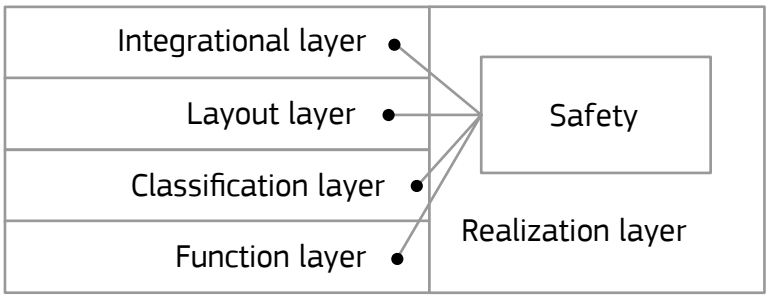
c) Issues arising from the implementation of formalized solutions, such as::
— Software and hardware systems aimed at enhancing the effectiveness and decision-making of automated systems;
— Applied standards and methods;
— Hardware solutions.

d) Solving problems arising from the development and maintenance of existing information systems, such as:
— Modernizing existing elements;
— Adding new elements.

Genesis Architecture

Genesis consists of:

a) The architectural levels;
b) Classification of transactions into types;
c) The principles of construction;
d) The interactivity ruleset;
e) Restrictions imposed upon the application of the ruleset and principles.



This technology is used in the development environment of Genesis.

5. Automated analysis of software

Automatic analysis of software is conducted through a basic search model on a Turing machine.

To understand the internal processes of such an analysis and how a machine can "request to remove uncertainty", it's best to present this as «a question». The concept of a question has a standard representation in all human languages; for a machine, this transforms into a set of states, connections, available control actions, and criteria for estimating the probability of success or failure.

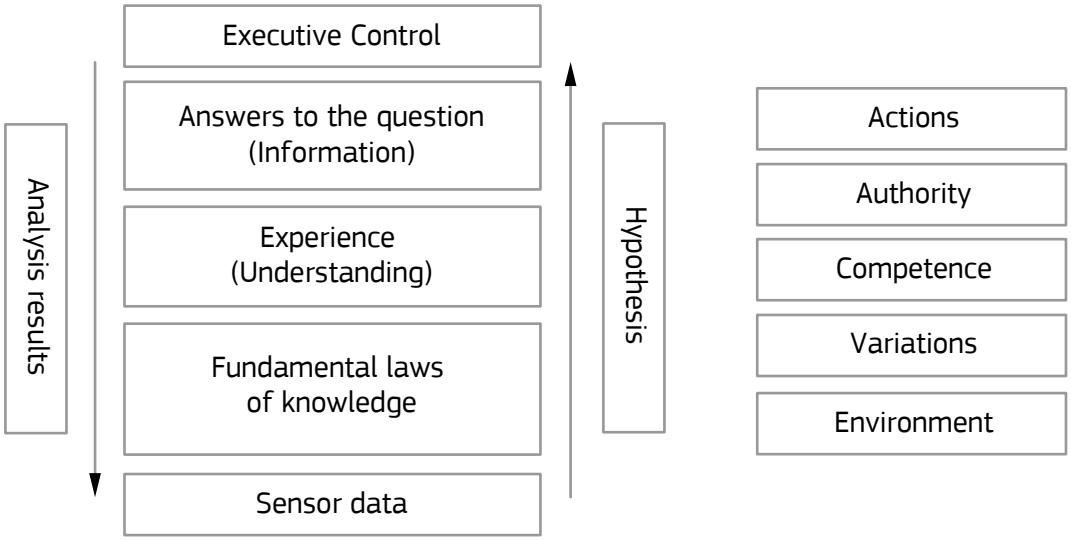
To take a control action, the system must unambiguously «understand» whether to use it or not; therefore, such requests must be semantically translated into questions with a «yes» or «no» answer.

The decision making process thus reduces to formalizing a sequence of control actions, selected as follows:

— Estimating the completeness of the model (i.e. forecasting the degree of uncertainty)
— Estimating the probability of obtaining the predicted result (i.e. measuring the potential reduction of uncertainty)
— Comparing the forecast with the previous values to arrive at a yes/no decision

In other words, the decision making process is a tree of yes/no answers to a question. This fully corresponds to the 1/0 capabilities of a binary computer.

The decision supporting process is the process of processing hypotheticals, knowledge, and laws based on the sequence of answers to yes/no questions. Within this framework, we can construct a sequence of control actions leading to the desired result.



The cognitive decision making scheme

To describe the decision supporting process, we'll use a mathematical construct with the following concepts:

Axioms — to represent the basis and the concept of the "theorem". This represents a single value effect (proof) from the overall set of axioms. It forms the basis within which we will describe the representation of the decision making process.

Signal — the full set of external conditions creating the need to make a decision.

Experience — the full set of recorded actions and consequences that follow a signal.

Factor — a parameter (concept) identified within the framework of experience.

Pattern — a set of correlated factors and the ranges of their potential values.

Communication — a pattern containing rules for changing correlated factors.

Control action — a pattern containing the rules for specifying communication factors, the range of their potential values and their current values.

Transition — changing the values of a given set of factors due to a control action.

Hypothesis — a correlated set of patterns.

Decision tree — the set of possible transitions within a hypothesis.

Effect — a pattern that describes the rules for the convolution of a decision tree into a factor symbolizing positive influence from the outside environment.

Risk — a pattern describing the convolution of a decision tree into a factor that symbolizes negative influence from the outside.

Within this framework, we can consider the decision making process as follows:

The process is kicked off by a signal. In order to justify a decision, the process needs past experience of previous decisions based on prior signals. Thus, we must have a relevant correlation of Signal — Experience and an unambiguous classification of the signal. This represents the key problem of identification with experience; the process of correct identification of a signal is a recursive function of decision making.

Experience is built on the basis of formal representations (data) about the environment and the links (rules) between the elements of each formal representation. There are also conditions (patterns) that kick off communication from a passive (potential) to an active (executed) state.

Because IT technology cannot currently describe the outside environment as an infinite set of factors, any description must exist within a bounded set and range of possible values describing past experience. This creates conditions within which experience is applicable and controlled as a hypothesis. Each separate hypothesis provides a link between the current state of all factors and possible changes based on potential links between them.

Links between factors can have several distinct natures:
— Rules (knowledge);
— Dependencies (laws);
— Assumptions.

Rules are relationships formed from experience and built upon the possibility of a deterministic or probabilistic interaction. They are used to show us an unambiguous parameter-based forecast about the surrounding environment.

Dependencies are relationships based upon continuous, uninterrupted dependencies (formulas). They give us a numerical dependency of the number of communication parameters and allow us to unequivocally state what will happen to each interlinked parameter.

Assumptions are a type of communication that stems from rarely repeated interrelated facts that create the feeling of a connection between them. We use the word ‘feeling’ because it’s not possible to confirm the presence of a link, but from past experience we have indirect evidence of a dependency between the two. Indirect evidence may involve statistical distributions, similarities and correlations, etc. We can also formulate an assumption based on an axiom: this connection exists.

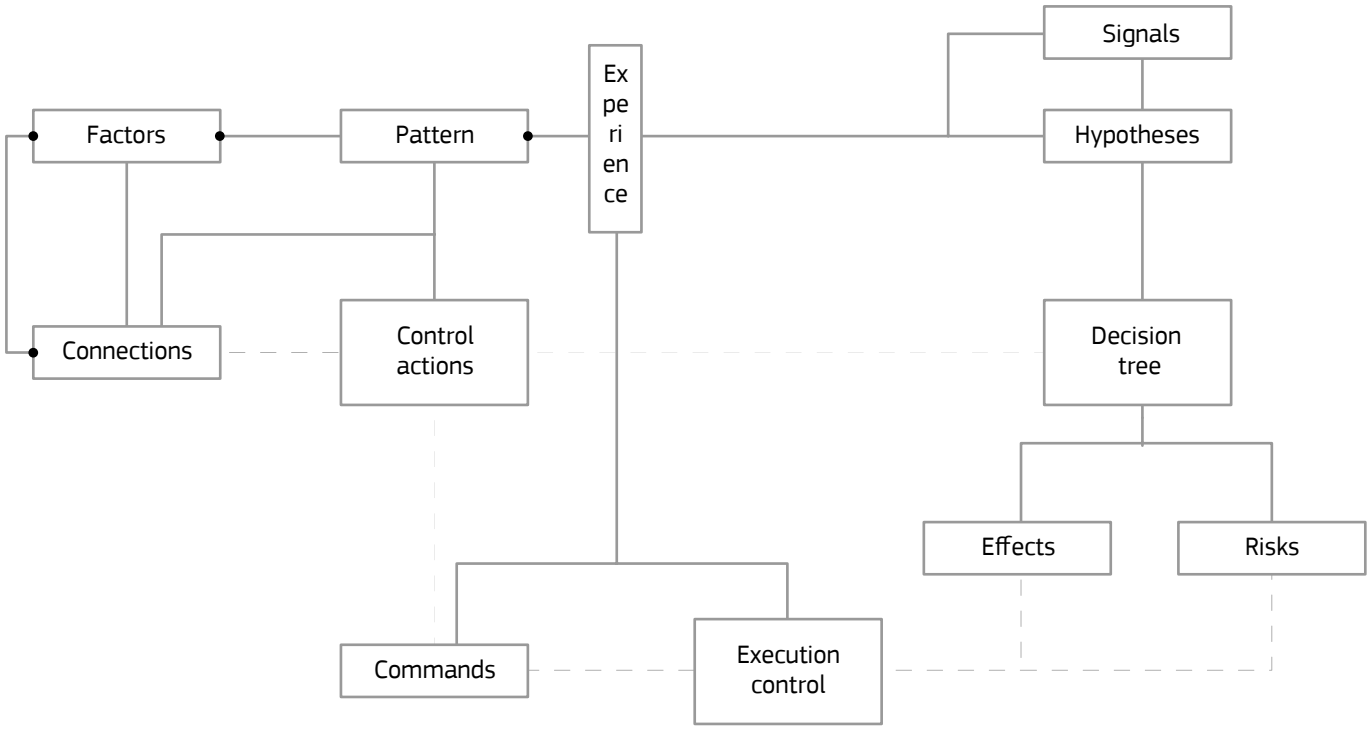
The decision supporting process

— On the basis of experience within the framework of an existing hypothesis in the environment , there are highlighted signals requiring a decision (positioning);

— On the basis of experience within the framework of an existing hypothesis and a given position, Genesis forms a set of possible control actions, activating all aggregate relationships the results of which become the decision tree. The depth and width of the decision tree determines the degree of adequacy of the eventual decision;

— On the basis of the decision tree within the framework of the experience and the hypothesis , Genesis determines the aggregate total of positive (effects) and negative (risk) consequences;

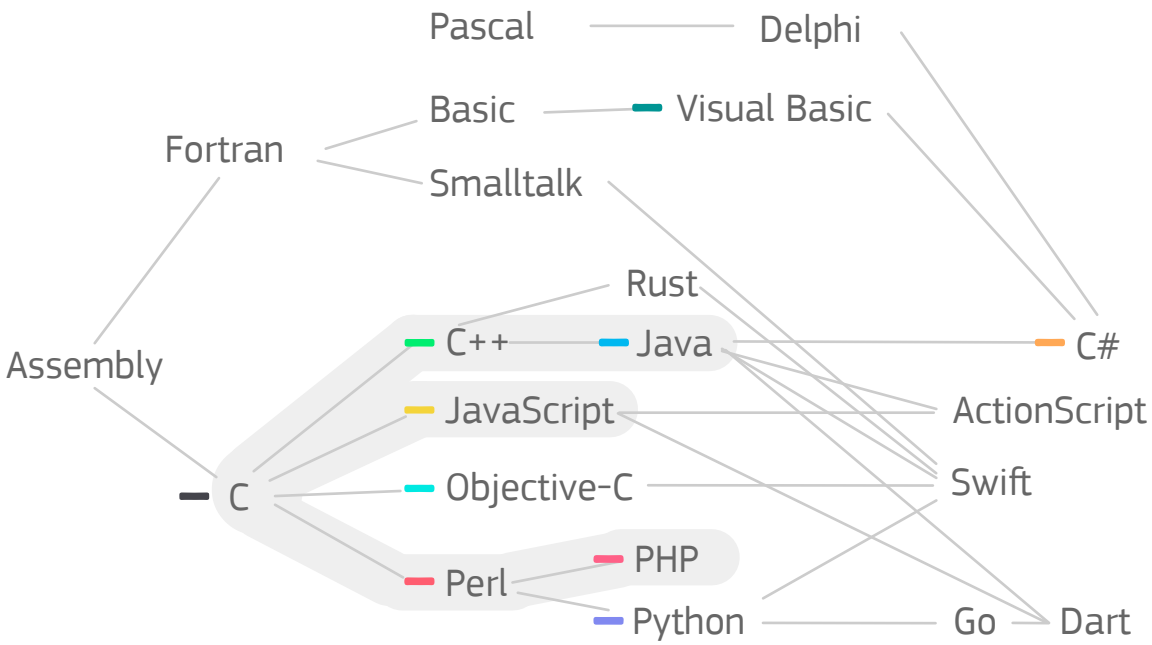
— Control actions are classified based on the ratio of the total set of effects vs. the total set of risks associated with each decision.



The decision supporting process

6. The Genesis universal programming language

The language has three application levels: (1) A systemic level, (2) An applied level, (3)An informational level.



Scheme of the evolution of programming languages

The first two levels are a hybrid of several mainstream programming languages (C/C++, Java, Perl, PHP, Javscript, and AS3.)

By default, the language uses a clear calculation sequence: all calculations are performed from left to right.

```
i = 5; i = i++ + i++;  // Result: i = 11
```

Function arguments are also calculated left to right by default.

The language has an automated import table with the ability to create aliases for all imported classes.

Import

	Path	Alias
1	com.genesis.sky.kernel.OpDescription	OpDesc
2	com.genesis.sky.parser.mc.OpDescription	OpDescEx
3	com.genesis.sky.utils.Serialize;	
4	com.genesis.sky.kernel.OpDescription	
5	com.genesis.sky.parser.mc.OpDescription	
6	com.genesis.sky.utils.Serialize;	
7		

Coder1

By default, it's possible to call an argument of a function anywhere:

```
my_function(*, true, *, "string", *, *, 575, my_var);
```

A function written in Genesis may return multiple values:

```
[a, b] = moddiv(48, 5);
```

Genesis can use tables to edit blocks of variables. It uses special boxes for WYSIWIG and rich text editing.

Genesis uses special, dedicated standalone boxes to edit text strings and special data. These boxes support a variety of encoding formats and line auto-forwarding.

```
debug private override synchronized native callback
function code_stack_set_b_op (desc:OpDescription, op:EntityOp) :void
```

	Type	Name	Value	Classification	Comment
1	Rectangle	rc	(0,0,0,0)	2D-Rectangle:pixel	Main Rectangle
2	Int<32>	x	100	X-coordinate:pixel	
3	Float<64>	angle	1,96	Angle:radian	Angle of rotation
4	Int<32>	i	0		Variable for iteration
5					
6					
7					

Functions

As stated above, Genesis supports the creation of various internal boxes. Supported box types include text strings, metadata, binary parameters and rich text. They can contain tables and auto-text strings.

There is also a special box format that excludes a fragment of code. Unlike code that would normally be commented out, when code is excluded using this box type, it is still refactored and checked for errors.

Declaring variables uses another special box table type with active headers.

The tabular method (inserting the table into the body of a function within the structural editor) has many advantages:

1. The ability to move columns.
2. The ability to sort columns.
3. Reducing collisions (identically named variables) between types, the names of variables and structural operators. For example, it is possible to enter types (if, for and max and x.)

4. Clear positioning of names, types, initializations, classifications and comments. Comment cells are rich text with autotransfer.

5. Variables may only be declared and described at the beginning of a function. Authors may arbitrarily put initialization tables in a function. However, you can also use VIEW or PREVIEW to combine all announcements into one table.

6. Tables can have multiple views. For example, comments can be on the right, the top or the bottom.

```
Source Code Pro, is also interesting, but unfortunately, the last
time I tried it, it didn't support Cyrillic. The authors promise to
include it in future releases.

At one point I liked Droid Sans Mono. I use it in PyCharm with the
monokai theme.

Source Code Pro, is also interesting, but unfortunately, the last
time I tried it, it didn't support Cyrillic. The authors promise to
include it in future releases.
```

```
debug private override synchronized native callback
function code_stack_set_b_op (desc:OpDescription, op:EntityOp) :void
```

```
1  if (desc.inp == OP.I_REF_REF)
2   var e1:Entity = code_stack.pop ();
3   var e2:Entity = code_stack.pop ();
4   code_stack.push (op);
```

Multithreading

The language supports several types of multithreading, including:

1. Multithreading with isolated threads and asynchronous communication between them;

2. Java-type multithreading.

Built in virtual constructor

The framework includes a special virtual constructor somewhat similar to HTML's CSS specification. It allows you to override the initial creation of new objects for virtually all classes of the application.

The constructor appears as a visual editor with a list of documents. The constructor has a strong communication system with its application classes, including refactoring, renaming and deletion. An automatic analysis of link integrity is also performed.

Localization

Genesis contains built in tools to support software localization while maintaining link integrity. Inline visualization of localized elements is supported.

7. Global file system

A file in Genesis can be systematically, functionally fragmented.

A file stored in GFS has a unique name / GUID and a clear ContentType, which is also a GUID. This means the file is identified strictly by its GUID, not a folder / filename structure as in a local file system (LFS).

Because of this, there are many opportunities for one file to have multiple names and occur throughout many folders of the LFS.

A system such as GitHub stores “non-structural” files and operates on differences within a file, i.e. byte by byte or character / pixel comparisons. For example, GitHub does not functionally distinguish between code changes. By comparison, Genesis operates on structural and functional differences between files.

The Genesis GFS repository can distinguish among such changes as:

— Added, deleted, or renamed classes

— Added, modified, or removed functions

— Similar changes to code

— Genesis has fixed functional and structural changes, as well as a different file system; this includes an active code version control system, not just making changes offline and merging them into the main branch later

GitHub cannot support full code storage. By contrast, in Genesis, centralized development and storage is combined with active code version control tracking — a key building block of security practices.

8. **Road Map**

Work begins

Prototype

MVP

ICO

Tokens listed on exchanges

Alpha

Beta

Genesis 1.0 platform release

IT-ICO fund*

A secure, decentralized OS*

9. ICO

Number of tokens	100,000,000
Individual token price:	$1
ICO amount: between	20,000 and 200,000 ETH
Bonuses	from 5% to 25%
Blockchain used	Ethereum
Token distribution	1% presale, 1% bounty campaign, 30% development, 56% ICO. A minimum of 10% of all tokens is held in reserve (all unsold tokens will also be used as a reserve), 2% to partners.
Token name	GW

Goal: Raising the ICO minimum of 20,000 ETH will allow us to finish the release of the Genesis platform.

**The IT-ICO fund will feature automatically controlled development stages of IT projects (i.e. scam protection).*
**The Genesis OS will feature the secure Genesis programming language and secure network code.*

Genesis World