SymOntoClay: White Paper

Author: Sergiy Tolkachov aka metatypeman
Date: December 2025

Website: symontoclay.github.io
GitHub: github.com/Symontoclay

Disclaimer

⚠ **Important Notice**
This project is currently at the stage of an early prototype and is being developed as a hobby during spare time. At present:

- it is a DIY effort maintained by a single developer in free time;
- the project is unstable and not optimized;
- examples demonstrate only basic capabilities;
- no demonstration scenes are available;
- performance is very low;
- a full-fledged community has not yet been established.

Use is possible only for research and experimental purposes. The project **is not intended for production environments** and should not be regarded as a ready-to-use tool.

Introduction

SymOntoClay is my experiment in developing a specialized language for describing the behavior of game NPCs.

This document reflects the gradual evolution of the idea: from a simple scripting language to a more complex solution. It should be regarded as a map of concepts and architectural decisions rather than a description of a fully completed system.

The purpose of this document is not to provide a detailed account of formal semantics or algorithmic implementations. I limit myself to a conceptual overview and the architectural role of the ideas, leaving technical details for future publications.

I approach the topic as a programmer rather than as an academic researcher. Therefore, the emphasis is placed on architectural concepts and practical applicability, rather than on rigorous proofs or exhaustive formalizations.

This document is primarily intended for:

- Myself, as the author of the project, since it serves as a reference point and guide for further development;
- Authors and enthusiasts interested in experimental approaches to describing NPC behavior;
- Independent developers who require a tool for specifying complex NPC behavior with minimal code volume.

The White Paper does not claim universality and is not intended as a manual for industrial development. Its purpose is to share ideas and outline directions for further experimentation.

Problem Statement and Motivation

Despite the extensive functionality of modern game engines, tools for declarative and human-readable specification of NPC behavior remain underdeveloped and fragmented. Frequently, the required functionality is provided through libraries rather than the language syntax itself, which significantly reduces code expressiveness.

Problem Statement:

- **Imperative Dominance**: most existing tools are oriented toward imperative programming, which complicates the declarative specification of knowledge and behavioral rules. As a result, such constructs are embedded locally within imperative code rather than represented as full-fledged declarative computation models. This limits expressiveness and complicates the development of systems where declarativity should be primary.
- **Lack of Abstractions**: built-in mechanisms rarely provide means for formalizing logical reasoning, heuristics, or working with knowledge bases.
- **Fragmentation of Solutions**: there is no unified specialized language that can be applied across different game environments while supporting non-standard AI paradigms.
- **Limited Support for Innovation**: new syntactic constructs in programming languages are primarily designed to address industrial development tasks and improve efficiency. Non-standard concepts do not receive full language-level support and remain outside the mainstream, forcing developers to implement them through external libraries or custom extensions.

Motivation:
The project is conceived as an experimental platform for exploring alternative approaches to describing NPC behavior. It serves as an environment for testing ideas.

The open-source nature of the project and its MIT license make it freely available for use and modification.

Principles and Project Philosophy

Any open project inevitably reflects the personality of its author.
These principles represent my personal guidelines, an expression of my values and preferences.
This is the way I find the work most engaging.

System Philosophy

The System Focuses Exclusively on Agent (NPC) Behavior

The system is designed solely for describing NPC behavior. It does not provide access to game scene objects, particle systems, animations, sounds, or similar components. In cases where the system might be applied in other domains—for example, in robot control—it will likewise not provide access to low-level hardware management.

This design approach enables:

- Abstraction from other game components and even from the execution environment
- Cleaner and more maintainable code
- The possibility of applying the system beyond the game industry—in simulations, robotics control, expert systems, knowledge bases, and ontologies

Since my current primary focus is the use of the system in game development, I will henceforth use the term NPC (Non-Player Character) to denote an agent.

Declarativity, Expressiveness, and Readability

The developer should focus on the required behavioral logic rather than struggle with language or framework constructs.

Code must be easily readable, understandable, and aesthetically clean. The developer should be able to grasp, at first glance, what the code describes and what it performs.

The number of places requiring modifications should be minimized—ideally limited to a single location.

In the design of the system, priority is given to declarative means.

Simplicity as Clarity and Expressiveness Through Understanding

In this system, simplicity is understood as a pursuit of clarity, expressiveness, and semantic transparency. It is a simplicity that arises from understanding rather than from the rejection of complexity.

My ideal is the simplicity found in Haskell: where expressive constructs allow complex behavior to be described concisely and intelligibly.

The system does not aim for artificial simplicity that restricts expressiveness. If, in the course of development, the system results in a very high entry threshold or even appears esoteric—so be it.

At the same time, it avoids excessive complexity that hinders comprehension. If, in the course of development, the system results in a very low entry threshold and appears overly primitive—so be it.

The essential requirement is that simplicity serves meaning, not form.

Usability Takes Priority Over Dogmatic Adherence to a Specific Programming Paradigm

The system is not rigidly bound to a single programming paradigm. Any paradigm is acceptable if its use enhances declarativity, expressiveness, and readability, and makes the specification of NPC behavior more convenient.

It is not advisable to implement logical programming exclusively through imperative or functional programming constructs merely to remain within the boundaries of the corresponding paradigm. Such an approach results, at minimum, in a loss of expressiveness and makes it difficult to discern the required logic amid large amounts of auxiliary code.

Similarly, I do not consider it necessary to repeat the attempt made in Visual Prolog to express imperative code solely through logical programming constructs, simply to remain within Prolog.

A paradigm is merely a means to achieve results, not an end in itself.

Implicitness Is Acceptable for Convenience and Expressiveness, but Control Must Also Be Possible

Implicitness helps avoid excessive amounts of code by removing the need to declare everything explicitly. This is a deliberate compromise: a trade-off of strictness and formal safety in favor of convenience and expressiveness.

Convenience requires reducing unnecessary code. Expressiveness requires freedom from technical noise.

At the same time, there must be optional mechanisms for controlling and overriding the system's default behavior. The developer should have the ability to understand, refine, and modify implicit decisions.

The key is to maintain balance.

Reducing Boilerplate Code Is Important, but Should Not Become Fanatical

Declarativity, expressiveness, readability, usability, and simplicity all imply minimizing template-like, repetitive code.

In this sense, the purpose of my system is precisely to reduce boilerplate code—for example, by allowing an entity to be declared without the need to register it in multiple places.

At the same time, eliminating every single "extra character" is not a priority, especially if declarativity, expressiveness, readability, usability, and simplicity are degraded as a result of such removal.

Reducing unnecessary characters is a means, not an end. If a bracket helps clarify structure, it should remain.

Determinism Is Important, but Non-Determinism Should Also Be Convenient When Needed

In human behavior, determinism manifests itself in logic, instructions, habits, rituals, and adherence to rules. This is a significant aspect, particularly when behavior must be explainable and reproducible.

Within the system, priority is given to determinism—so that behavior remains predictable, debuggable, and reproducible.

At the same time, mechanisms for enabling non-deterministic behavior are also necessary where appropriate. Randomness, variability, and improvisation are likewise part of realistic behavior, and they should be available without sacrificing transparency or controllability.

Architectural Design

Isolation of NPCs: From Each Other and from the Game Scene

Each agent (NPC) in the system is isolated—it has no direct access to the game scene or to the data of other NPCs.
NPC behavior is formed on the basis of its own knowledge and the data obtained through perception channels.

Modularity and Incremental Construction of Agent (NPC) Behavior

The behavior of agents (NPCs) can be described in parts, with the ability to gradually extend, override, and refine it. The system supports a modular structure in which each behavior block can be added, disabled, or adjusted without the need to rewrite the entire behavior.

Advantages of modularity:

- **Incrementality**: development can begin with simple behavior and gradually add new reactions, refinements, and scenario-specific exceptions.
- **Reusability**: behavioral blocks can be structured as modules and reused across different contexts.

- **Overriding**: more specific modules can override or refine behavior defined at a more general level.
- **Isolation**: modules are not required to be aware of each other—they can be combined by the system based on priorities, context, or scenario.

A module may be represented as a book, scroll, or artifact that grants an NPC new knowledge or abilities. This is particularly relevant in an architecture where NPCs are isolated and do not have global access to behavior definitions.

For example, a game scenario might proceed as follows:

1. The NPC finds, purchases, or receives a book from a teacher.
2. The game triggers a reading scene or a knowledge-acquisition animation.
3. At that moment, the engine attaches the corresponding module to the NPC.
4. The NPC acquires new knowledge or a behavioral capability.

This approach makes the architecture not only technically flexible but also narratively expressive—behavior becomes part of the world rather than merely code.

Full Control over Execution

The system provides the ability to stop execution at any moment and save it as an image. After loading from the image, execution can be resumed exactly from the point at which it was interrupted.

In practice, this means:

- **Stop and Save**: execution can be interrupted at any point, preserving all data about the agent's state, its modules, and its context.
- **Resume**: after loading from the image, execution continues without loss of information—the NPC behaves as if no pause occurred.
- **Deterministic Restoration**: it is guaranteed that the restored behavior is identical to the original, with no hidden discrepancies.
- **Instrumentality**: developers or researchers can use save/resume functionality for debugging, experimentation, or scenario control.

The Engine's Application Should Not Be Limited to Games

The engine is designed as a universal platform for describing and executing agent behavior. Its application is not restricted to the game industry—the architecture is intended to support use in other domains where knowledge, logic, and control are essential.

Possible additional application areas:

- Knowledge base
- CLI in the style of an interactive Prolog interpreter
- AI for robot control (with server-side execution)

All Execution Must Occur Locally on the User's Computer Without Requiring Internet Connectivity

The engine must operate entirely locally on the user's computer. All functions are executed without the need for an internet connection. This ensures autonomy, reliability, and control over the process.

In practice, this means:

- **Autonomy**: the system does not depend on external services or servers.
- **Reliability**: operation continues even in the absence of a network or with limited access.
- **Transparency**: the user knows that all computations occur on their device.
- **Security**: NPC data and game states never leave the local machine.
- **Predictability**: no external factors can alter the behavior of the engine.

The Use of Third-Party Components Requiring or Potentially Requiring License Purchases Is Not Permissible

The engine must not include third-party components that require or may require the purchase of licenses. All elements of the system must be free from hidden restrictions, paid dependencies, and mandatory subscriptions.

A Simple Imperative Scripting Language

Let us take as a basis a relatively simple imperative programming language. Imagine the simplest imperative language of the kind commonly used for game scripting, and especially for modding.

It should support:

- Function declaration and invocation
- Conditional statements (if–then–else)
- Loops
- Assignment
- Arithmetic operators
- Comparison operators

This is sufficient for basic tasks: any game scenario can be expressed through a combination of these constructs.

At this stage, this hypothetical simple imperative programming language remains a pure and undefined abstraction. One may imagine any programming language typically applied to game scripting.

Now it is necessary to determine the style. However minor the importance of style may seem compared to other features of the language, defining the style crystallizes this hypothetical language into a more concrete form.

Currently, the primary styles of game scripting are:

- **C-like**. Most familiar to me due to continuous work with C-like languages.
- **Python-like**. Renowned for minimalism—this is where the fight against boilerplate extends to the very last character.
- **Lua-like**. A vast number of games are scripted using it.

Other styles worth recalling include:

- **Pascal-like**. Elegant and concise, visually reminiscent of English text.
- **Basic-like**. Also resembles English text, though with its own characteristics.
- **Forth-like**. Extremely concise, esoteric. Its stack-based nature provides certain advantages over other languages, but also introduces specific challenges.
- **Smalltalk-like**. An esoteric style ideally suited for message passing between objects.

Styles can also be mixed, as demonstrated by Ruby.

I choose the C-like style. This decision is driven by my professional habits, as I program in C# and JavaScript, and previously in C++. My eye is accustomed to blocks delimited by curly braces, which significantly reduces cognitive load for me.

C-like languages are actively used in game development or have been used historically: C++, C#, Java, JavaScript / TypeScript, and of course the legendary QuakeC, AngelScript, Squirrel, UnrealScript, and TorqueScript.

Units of Measurement

In the previous section, we introduced a simple imperative programming language. It is entirely mainstream and could be any existing C-like language commonly used for game development.

Now let us take a small step away from simplicity and conventionality.

First, consider the units of measurement for numerical values. The presence of units of measurement helps prevent errors caused by assigning incorrect values. For example, passing a time value where a length is expected, or specifying a number in seconds where hours are required.
Moreover, units of measurement are highly intuitive, as they surround us in everyday life.

Units of measurement in a programming language can be applied in two ways:

- **As built-in language syntax.** Similar to how this is implemented in F#, Ada, and formerly in Fortress.
- **Through libraries.** As is commonly done in C#, Java, C++, JavaScript / TypeScript, Rust, and others.

Code using units of measurement via libraries looks approximately as follows. This is an example of C# code using the UnitsNet library:

```
Length distance = Length.FromMeters(100);
TimeSpan time = TimeSpan.FromSeconds(9.58);
Speed speed = distance / time;
```

Or like this, when using custom Value Object implementations:

```
var distance = new Meter(100);
var time = new Second(9.58);
var speed = new Speed(distance.Value / time.Value);
```

Now let us consider a possible syntax with units of measurement integrated directly into the language. We take as a basis programming languages that provide such capabilities. It is important to remember that the goal of this experiment is conciseness, expressiveness, and proximity to natural language notation.

The result would look something like this:

```
var distance = 10.0 m;
var time = 5.0 s;
var speed = distance / time;
```

This appears much more compact and concise.

Nevertheless, units of measurement alone are unlikely to serve as a compelling reason to design a new language. Games can be scripted perfectly well without them. And if they are truly needed, libraries provide a sufficient solution.

Full Control over Execution

The next noteworthy feature is **Full Control over Execution**. This means that program execution can be stopped, the program state saved into an image, and later that image can be loaded to resume execution. Ideally, the program should continue precisely from the point where it was halted.

Process flow:
[Program Start] → [Pause] → [Save Image] → [Load Image] → [Resume Execution]

This approach is used in Smalltalk / Pharo / Squeak, and occasionally in Lisp / Scheme, particularly in Emacs.

Potential use cases include:

- **Saving and restoring a game level.** In this case, NPC behavior code remains clean and transparent, since no additional code is required for stopping, saving, loading, and restoring.
- **Work in an interactive environment.** This allows saving the system image and continuing later. A similar mechanism existed in early versions of Smalltalk and still exists in its modern implementations.

When saving a game level, it becomes straightforward to implement time suspension for NPCs. If time is tracked using an internal counter that is also saved and restored, then from the NPC's perspective, time halts when the program is paused. On one hand, this simplifies and clarifies NPC behavior modeling. On the other hand, it complicates interaction with external (physical, wall-clock) time.

It is evident that saving program images has many drawbacks. This is why the approach has not gained widespread popularity, despite the long time since its first practical implementation.

The most apparent drawbacks include:

- **Size and complexity of images**: saving the entire state can be heavy and inefficient, especially in large systems.
- **Security**: saving the complete process state includes everything — even private keys, passwords, and network connections. This makes images potentially unsafe for distribution.
- **Non-determinism**: external resources (network, files, APIs) change over time, so upon restoring an image, the expected environment may already differ.
- **Code evolution**: if the code changes after saving an image, restoring the old state may be impossible or may lead to conflicts. In industry, the ability to update systems without “frozen” dependencies is highly valued.
- **Difficulty of comparison**: tests can be easily compared to identify differences. For binary blobs, this is very difficult, if not impossible, to obtain a human-readable diff.

- **Integration with tools**: CI/CD, testing, modularity — all these practices assume working with source code and data, not with monolithic images. Images do not integrate well with modern development practices.

These drawbacks apply to already implemented systems.

In the case of building a system from scratch, this implies the need to independently develop all required tools, primarily the interpreter and debugger.

It is, of course, possible to implement a translation into an existing language whose runtime supports full control over execution. In this case, such a language would be Smalltalk-like or Lisp-like.

However, in any case, this goes beyond the boundaries of existing C-like languages.

Here arises a key choice: either accept industry constraints and rely on data serialization and frameworks based on existing systems, or take the risk of building a custom environment.

On one hand, this opens a vast and engaging space for experimentation.

On the other hand, it involves all the risks of developing a system from scratch under limited resources.

I choose the path of experimentation. I am interested in building a fairly complex system from the ground up and observing where it leads. Even if the risks outweigh the advantages, this is precisely what makes the task genuinely compelling.

Creating a DSL: Advantages and Disadvantages

The desire to have full control over execution, combined with the need for concise syntax for units of measurement, shifts the balance toward creating a custom DSL.

Before moving on to the creation of a DSL and enriching it with interesting features, it is useful to examine the general advantages and disadvantages of DSLs, as well as certain aspects of their implementation. It is also worth considering the cultural context — how the emergence of yet another DSL is perceived by developers.

A DSL (Domain-Specific Language) is a programming language specialized for a particular domain. Unlike general-purpose languages, a DSL addresses a narrow set of tasks but does so with maximum efficiency and clarity for experts in that domain. In this case, the purpose of the DSL is to describe NPC behavior.

DSLs offer several advantages:

- **Semantic clarity**: A DSL models the domain with high fidelity, reducing the cognitive gap between concept and code.
- **Accelerated development**: Tasks are formulated closer to business logic rather than low-level details.
- **Accessibility for non-programmers**: A DSL can serve as a bridge between engineers and domain experts.
- **Safety and robustness**: The grammar of a DSL defines boundaries that make logical errors less likely. The compiler enforces correctness.
- **Optimization potential**: Under the hood, a DSL can be translated into efficient code while maintaining simplicity at the surface level.

However, accessibility for non-programmers is not a priority in this development, nor is lowering the entry barrier.

Additionally, creating a DSL provides freedom to experiment with language design.

DSLs also have notable disadvantages:

- **Development cost**: Creating and maintaining a DSL requires time, expertise, and infrastructure.
- **Learning curve**: Users must learn a new syntax, even if it is simpler than a general-purpose language.
- **Limited expressiveness and universality**: DSLs rarely cover all scenarios, requiring parts of the code to be written in another, more universal language. If a user knows only the DSL and its environment, using another language may present difficulties.
- **Lack of documentation and community**: A custom DSL almost always lacks sufficient educational materials, an active community, and answers on platforms like StackOverflow.

For this particular DSL, additional disadvantages include:

- **Development timeline**. It is evident that a solo developer faces challenges in handling a large workload: DSL specification with required features, interpreter, debugger, integration with game engines, integration with development environments or building a custom environment, documentation and usage examples, a demo scene showcasing engine capabilities, and much more.
- **Dependence on the author's vision**. SymOntoClay is a 100% visionary project. This creates the risk of misalignment with actual user needs. Even if the current vision matches expectations, it may lose relevance as the project evolves.

⚠ **Important note**: At present, SymOntoClay has no users. The only user is the author himself.

It is useful here to briefly examine how developers perceive the emergence of new languages.

Many developers view DSLs simply as another tool — one that either solves part of their problems and tasks or, conversely, is entirely useless to them. This is a purely pragmatic perspective.

It should be noted that the appearance of new DSLs often provokes skepticism among some developers: some are irritated by excessive promotion by the author, others consider such projects a waste of effort compared to contributing to mature Open Source initiatives, while for others the very existence of a new language is perceived as a threat to established practices and as additional fragmentation.

Thus, a DSL remains a tool with a dual nature: it provides semantic clarity and freedom for experimentation but requires significant effort and carries risks associated with the author's vision and cultural perception. With these advantages and disadvantages in mind, one can proceed to examine approaches useful for game AI.

Adding OOP

We use OOP (Object-Oriented Programming) to structure the code. At this stage, the key advantage of OOP lies in combining data and methods into a single entity (encapsulation). Such an object can be considered analogous to a character template (Prefab, stat block) in game design and role-playing games: it defines a general structure that can be applied to different NPCs.

Another capability of OOP is expressing the *is-a* relationship through inheritance. This allows us to easily describe hierarchical relationships between objects.

For example:

- **Base object**: Human — a general NPC with basic properties (health, position, movement and interaction methods).
- Derived object from Human:
- Derived object from Soldier:

    - **Fighter** — a versatile combatant skilled in weapons and armor.
    - **Paladin** — a holy warrior combining combat and magic, protecting allies.
    - **Ranger** — a hunter and scout, master of ranged combat and survival.
    - **Barbarian** — a fierce fighter relying on strength and rage.
    - **Knight** — a disciplined soldier bound by a code of honor.
    - **Mercenary** — a pragmatic fighter focused on profit.

```
Human
├── Trader
└── Soldier
├── Fighter
├── Paladin
├── Ranger
├── Barbarian
├── Knight
└── Mercenary
```

Under the hood, the system will employ prototype-based inheritance, similar to JavaScript. That is, an object and its prototype maintain a connection, and changes in the prototype immediately affect all its descendants.

The prototype model aligns with the “everything-is-an-object” philosophy: every element of the system is an object that can serve as the basis for others. This approach naturally reflects game design, where NPCs are created from templates and extended with new roles and abilities.

Prototypes represent general concepts. Thus, a general concept is treated as an object just like an instance. This adds flexibility to the system and allows uniform extension of both instances and general concepts. NPCs can be assembled like building blocks, adding unique abilities. It is even possible to create an object from scratch and gradually transform it into the desired form.

Despite the advantages of the prototype approach, the class-oriented inheritance model offers benefits that are significant for the system under development:

- Separation between the class level and the object level effectively distinguishes sets of general concepts from sets of instances.
- The class-oriented model is familiar to most developers (including myself). This is why JavaScript introduced class syntax in ES6, even though the underlying model remained prototype-based.

Therefore, I will use a class-oriented approach to inheritance, while retaining the prototype model under the hood and the ability to create any object from scratch and incrementally transform it.

In addition to the universal keyword “class,” a number of specialized constructs will be introduced (and examined in detail later in this document) to reflect specific roles in the system and make the code more semantic. These constructs include:

- “world” — global context of the game world
- “app” — NPC
- “lib” — library of reusable code
- “module” — module
- “state” — NPC state
- “action” — functor
- “task” — HTN task

The language supports multiple inheritance: a class or object can inherit behavior from several sources simultaneously. This expands code reuse and enables modeling of complex roles and scenarios. For example, an NPC may combine multiple roles, such as Warrior-Mage.

Multiple inheritance:

- Simplifies role combinations.
- Facilitates code reuse.
- Enhances expressiveness.

At the same time, multiple inheritance introduces certain challenges, such as:

- Name conflicts — if two parents define the same method, explicit resolution is required.
- Hierarchy complexity — diamond-shaped inheritance (the diamond problem) may occur.

Thus, multiple inheritance provides advantages that appear attractive for the system under development. Experience from many languages that use multiple inheritance shows that its drawbacks can be successfully addressed.

Conflict resolution will be demonstrated through the use of priorities. In addition, inheritance linearization will be applied, which, together with priorities, will help resolve conflicts.

The diamond problem is solved by ensuring that a common ancestor is accounted for only once.

For more complex conflicts, resolution mechanisms will be developed based on the experience of existing languages with multiple inheritance.

Interfaces are not planned at this stage, but this may be reconsidered in the future as the project evolves.

Code reuse is achieved in several ways:

- **Inheritance**. A class or prototype can serve as the basis for others. Common logic is placed in a base object, while derived objects inherit it and may add or override methods. Example in game design: the base Human defines everything that makes an NPC human, while Soldier inherits these properties and adds combat skills.
- **Traits**. Behavior modules that can be “mixed in” to any object or class. Unlike inheritance, traits do not define a hierarchy but allow combining independent pieces of functionality. Example in game design: the trait Flyable adds the ability to fly, while Tradable adds the ability to trade. The same NPC can acquire both traits, regardless of its base role.

In the language, all functions are first-class objects. This means they have the same capabilities as other objects: they can be stored in variables, passed as arguments, returned from other functions, and extended with methods, state, and rules.

At this stage, such an implementation of OOP will cover most needs for constructing NPC behavior. As new features and mechanisms are introduced, the OOP implementation will be refined, which will be described in subsequent sections.

Knowledge Base and Logic Programming

Logic programming enables the following:

- Storing knowledge in the form of explicitly defined facts. This improves code readability compared to storing knowledge through imperative programming constructs.
- Rules declaratively define the derivation of new knowledge from existing facts, thereby reducing the need to explicitly encode large amounts of knowledge. Declarative rules are more transparent and expressive than distributing logic across imperative code.

This implies the presence of a small expert system within each NPC.

Facts can serve as a universal means of describing data for NPCs. For example, sensor data may be provided as sets of facts that describe the game scene. The NPC will process them in the same way as internally defined facts.

Knowledge exchange with ontologies can also be implemented, allowing integration of the knowledge base with external semantic models. This may be particularly useful when using SymOntoClay as a knowledge base engine.

The classical implementation of logic programming is Prolog, which provides the following core mechanisms:

- Facts
- Rules
- Logical inference from existing facts using rules

Even these basic capabilities significantly improve NPC programming by offering a more declarative, expressive, and human-readable representation of knowledge.

The Prolog syntax will not be copied verbatim:

- Rules will use the arrow “->” instead of the symbol “:-” for greater expressiveness.
- Variables will have the prefix “$”. This helps distinguish, for example, the concept “x” from the variable “$x”.
- Instances will begin with the symbol “#”. For example: “dog” — the concept of “dog”; “#dog1” — an instance.
- Comparison operators in SymOntoClay may also differ from classical Prolog.
- Since facts and rules exist within multiparadigm code, they will be enclosed between “{:” and “:}”.

In the future, it will be possible to move from classical predicate syntax to more human-readable forms.

Beginning to Connect Paradigms

In the previous section, logic programming was added to the imperative language. At present, the imperative and logical paradigms remain separate. Our task is to integrate them into a unified system.

At this stage of evolution, simple mechanisms will suffice. In the following sections, the refinement of paradigm integration mechanisms will be described.

Query Operators

To begin, let us introduce simple query operators:

- **INSERT** — adds a fact or rule to the knowledge base.
- **SELECT** — retrieves data that satisfies a condition. Equivalent to “?-” in Prolog.

These operators are partially inspired by the interactive mode of the Prolog interpreter, where the user formulates goals using “?-”, and partially by SQL.

Implicit Queries in Conditions

Let us add the ability to specify facts within conditions. When such a condition is evaluated, the system automatically performs a hidden query to the knowledge base: if the fact is found, the condition is considered true.

Imperative-Logical Dualism of Inheritance

Let us consider the *is-a* relationship. In the system under development, it can be defined through:

- An inheritance relationship in OOP
- The assertion of a fact in the knowledge base

At the same time, the *is-a* relationship must be accessible in the system regardless of how it is defined.

The solution is to establish a connection between the inheritance relationship and the *is-a* relationship in the knowledge base.

Defining inheritance in OOP automatically creates the corresponding fact in the knowledge base. Conversely, asserting the corresponding fact in the knowledge base automatically establishes the inheritance relationship in OOP.

Imperative-Logical Dualism of Properties

Let us examine knowledge/facts in more detail. Facts are convenient as a declarative set of knowledge. At the same time, in imperative code it is often much more practical to assign a value to a property or retrieve a value from a property.

The solution is to establish a connection between a predicate and a property of the same name. This connection can also be configured to link a predicate and a property with different names.

Writing to a property automatically creates or updates the corresponding fact. Creating or modifying a fact automatically changes the value obtained from the property.

Imperative form	Logical form
dog.color = black	color(dog, black)
house.owner = #alice	owner(house, #alice)
#npc1.health = 42	health(#npc1, 42)

Thus, a property serves as a bridge between imperative and logical code. This imperative-logical dualism enables the use of notations that are convenient within their respective paradigms.

Triggers

Let us add triggers to our system, thereby introducing an event-driven paradigm. This will enable NPCs to respond to events.

Step 1: Triggers for NPC System Events

At the initial stage, triggers function as lifecycle events for NPCs and other objects. This makes it possible to uniformly express, through triggers, what in other languages is represented by constructors, destructors, Python dunder methods, or PHP magic methods.

Initially, the following will suffice:

- **Enter**: for initialization
- **Leave**: for handling the end of existence

Additional lifecycle events can be introduced later.

Step 2: Triggers on Imperative Elements

Let us add the ability to specify conditions in a trigger using variables and properties. Such triggers will be activated only if the condition evaluates to true.

Step 3: Triggers on Knowledge Base Events

Let us add to triggers the ability to monitor changes in the knowledge base. We also introduce the option to specify a fact within the trigger condition. This will allow imperative actions to be automatically executed when facts change.

A useful mechanism will be declarative mapping of values extracted from the knowledge base into variables at the trigger handler level.

Thus, the trigger becomes a bridge between the logical and imperative components.

Since events from sensors in the game world will be represented as sets of facts, there is no longer a need for a dedicated sensor trigger — all sensor events can be processed within the knowledge base event trigger.

Step 4: Triggers on Knowledge Base Meta-Events

Let us add triggers for knowledge base meta-events:

- Addition of a new fact
- Deletion of a fact
- Modification of a logical rule

The list of meta-events can be expanded in the future.

These triggers allow modification of facts before they are added to the knowledge base. In this way, an additional middleware layer is created, making the system more flexible.

Step 5: Automatic and Forced Trigger Reset

Up to this stage, a trigger was reset automatically immediately after activation. This is convenient in many cases.

However, in some situations it is necessary for a trigger to remain active for an extended period and be reset only upon the occurrence of a specific condition or a timeout.

At this stage, this mechanism helps prevent premature or excessively frequent activations.

It also establishes a foundation for subsequent improvements.

Step 6: Triggers on Other Triggers

At this stage, the following improvements are introduced to triggers:

- Add the ability to create named triggers
- Add the ability to use named triggers in the conditions of other triggers

A named trigger can appear in a condition like a regular property.

In the previous step, the ability to forcibly reset a trigger based on a condition was added. Thus, a named trigger can be activated by one condition and reset by another. In its active state, such a trigger will appear as a property with the value True, and in its reset state it will have the value False.

These triggers become interconnected: they can activate one another, forming cascading scenarios. This transforms the system into a flexible network of reactions, where complex behavior is constructed from simple event-driven links.

At this stage, the system gains the ability to process complex events. A complex event is a combination of several events (simple or complex), related by time or logic. For example:

- “NPC lost 10 HP three times within 5 seconds” -> “Critical threat”
- “Night and the bell rang three times” -> “Start of attack”

Thus, SymOntoClay turns NPCs into a mini-CEP (Complex Event Processing) system, capable of modeling NPC behavior at the level of event patterns.

Step 7: Triggers as Timers

At this stage, we add the ability to use triggers as timers:

- Periodically execute handlers at specified intervals
- Execute a handler once after a certain period of time
- Reset triggers based on a timeout

Time becomes a fully-fledged source of events, expanding system capabilities and enabling the modeling of scenarios with delays, periodicity, and duration.

Step 8: Combined Triggers

Now let us combine the previous stages by creating a combined trigger. Different conditions can now be joined using the operators AND / OR / NOT.

This makes it possible to describe complex NPC behavior scenarios through logical combinations, turning the system into a fully-fledged language of event patterns.

Thus, after completing eight steps, we have transformed simple reactive mechanisms into a flexible network of triggers capable of working with facts, meta-events, time, and complex combinations. SymOntoClay becomes a mini-CEP (Complex Event Processing) system, where NPC behavior is described through a rich language of events and reactions.

Fuzzy Logic

Let us add fuzzy logic capabilities to the system. This enables reasoning under conditions of uncertainty.

Since the DSL is being developed from scratch, support for fuzzy logic can be easily introduced at the syntax level.

Thus, syntax can be added for defining:

- Linguistic variables
- Fuzzy sets
- Operators
- Membership functions

Initially, built-in Membership Functions and Operators provided by the engine can be used.

Fuzzification and Defuzzification algorithms can also be implemented within the engine, and later made configurable.

At this stage, it becomes possible to operate with fuzzy concepts. Rules can directly employ notions such as “near,” “far,” “many,” “few,” “very,” “almost,” and similar terms. All of this remains highly human-readable and transparent.

Applications of fuzzy logic in the system include:

- Defining fuzzy values in imperative code
- Comparison mechanisms in imperative code
- Defining fuzzy values in facts
- Logical inference mechanisms
- Fuzzy inheritance, where a fuzzy value is used as a priority

Extending Logical Programming

Up to this point, the system in the area of logical programming essentially replicated classical Prolog.

However, despite its elegance, this approach has limitations that hinder its use in more complex and “dynamic” scenarios:

- **Binary truth**: a fact is either true or false. In the real world, we often deal with probabilities, degrees of certainty, or exceptions.
- **Requirement of complete knowledge**: classical logic presupposes a complete and consistent description of the domain.
- **Cumbersome expressions**: reasoning about space, time, or context requires lengthy and verbose constructions that overload the code.
- **Lack of flexibility**: classical logic is monotonic — new facts only add knowledge but cannot revoke previous conclusions. This poorly reflects the dynamics of the real world, where knowledge often contradicts itself.
- **Weak handling of norms and rules**: classical logical programming lacks built-in means to express “mandatory,” “permitted,” or “forbidden.”
- **Limitations at the meta-level**: there are no convenient mechanisms for managing access to facts, their visibility, or combining different logics.

Therefore, there arises a need to extend classical logical programming: to add probabilistic, temporal, spatial, deontic, and other dimensions, as well as to introduce a meta-layer that governs their interaction. This will make reasoning closer to human reasoning and more suitable for complex systems — from games and simulations to expert systems and knowledge bases.

All of the reasoning described can also be implemented in classical Prolog using predicates and rules. However, such an approach becomes cumbersome: spatial relations must be encoded through numerous nested facts and manual checks, while deontic and modal categories require additional layers of abstraction. As a result, the logic loses clarity and becomes difficult to maintain.

Thus, the task arises: to introduce the necessary operators directly into the DSL, making the description of NPC behavior more compact, expressive, and closer to natural language.

Probabilistic Logic

Traditional logic operates with binary values: a statement is either true or false. This approach is convenient for formal proofs but poorly reflects the real world, where knowledge is often incomplete and conclusions carry varying degrees of certainty.

Problems of binary logic:

- **Lack of confidence gradations**. The system cannot distinguish between “almost certain” and “unlikely.”
- **Weak handling of uncertainty**. In real scenarios, facts may be incomplete or contradictory.
- **Limitations in behavior modeling**. Game NPCs or expert systems must make decisions under incomplete information, and binary logic provides no tools for this.
- **Inability to integrate** with data from statistics or machine learning, where probability is a key concept.

Probabilistic logic removes these limitations and enables the system to:

- **Work with incomplete information** — draw conclusions even without a complete fact base;
- **Model uncertainty** — express degrees of confidence in facts and rules;
- **Make decisions based on risk** — choose actions not only by logical necessity but also by probabilistic benefit;
- **Integrate with other extensions** — for example, combine temporal or spatial reasoning with probabilistic forecasts.

Principles of probabilistic logic:

- Each statement can have a degree of confidence (e.g., 0.8 instead of “true”). Expressiveness is further enhanced by the use of linguistic variables, such as “quite likely” instead of 0.8.
- Conclusions are derived with consideration of probabilities, not only formal truths.
- The system can combine probabilities from different sources, refining conclusions.
- Decisions are made with regard to risk and uncertainty, not only strict rules.

Probabilistic logic extends the classical system, enabling work with uncertainty and degrees of confidence. It makes reasoning closer to reality, where conclusions are rarely absolutely true or false.

During system development, different approaches can be explored — ProbLog, Bayesian Logic Programs, Markov Logic Networks — aiming for a balanced combination to achieve optimal trade-offs between declarativity, expressiveness, and computational efficiency.

Temporal Logic

Classical logical programming treats facts as static: they are either true or false within a single knowledge base. However, many systems operate with processes where temporal dynamics are essential: events occur sequentially, states change, and conclusions depend on the moment of observation.

Temporal logic enables the system to:

- **Model the evolution of states** — reason not only about what is true, but also when it is true;
- **Formulate temporal constraints** — for example, “event A must occur before event B”;
- **Verify process properties** — “the safety condition is always satisfied,” “the target configuration will eventually be reached”;
- **Integrate with other extensions** — for instance, combine temporal reasoning with probabilistic forecasts or deontic norms (“an action must be performed before the deadline”).

Temporal logic transforms the system from a static reasoner into a dynamic observer, capable of accounting for event sequences and ensuring the fulfillment of conditions over time. It complements classical logic with the dimension of time, making reasoning more suitable for processes rather than only static facts.

Spatial Logic

In the real world, as well as in games, reasoning is often connected to the spatial arrangement of objects. Spatial logic makes it possible to formalize relations such as “near,” “inside,” “above,” “in front of,” and other spatial dependencies.

For game NPCs, this means that their behavior can take into account not only facts and rules but also the geometry of the environment: where objects are located, which zones are accessible, and which obstacles hinder actions.

Examples of application:

- **Example 1. Shooters**. An NPC must evaluate the situation in terms of the positions of enemies, cover, and possible routes. For instance: “Enemy to the right behind the wall,” “There is an opportunity to quickly cross the corridor.” Such reasoning allows the NPC to choose tactics: attack, take cover, or retreat.
- **Example 2. Commander Simulators**. A commander specifies a movement location not as an abstract identifier but through spatial relations: instead of “platform #E041CF1D-DE0C-4680-B73D-805AE7759197,” the description “platform to the left between two large stones and a bush” is used. The NPC interprets such instructions by matching them with the environment map and selecting the correct point.

Types of spatial relations:

- **Topological**: inside, outside, intersects, covers. Zone definition: “NPC is inside the fortress,” “the road intersects the river.”
- **Orientational**: left, right, in front, behind, above, below. Tactical orientation: “enemy on the right,” “cover in front of the NPC.”
- **Distance-based**: near, far, at distance X. Action selection: “cover is near,” “enemy is far — shooting is possible.”
- **Hierarchical**: nested areas, belonging. Navigation: “room inside the building,” “building inside the city.”
- **Relative**: between, next to, opposite. Location specification: “platform between stones and a bush,” “NPC opposite the gate.”
- **Dynamic**: approaching, moving away, moving along. Motion tracking: “enemy is approaching,” “NPC is moving along the wall.”

The introduction of specialized operators for spatial relations makes expressions more clear and easier to maintain compared to classical expressions based solely on predicates.

Modal Logic

In the real world, it is important to reason not only about facts but also about the possibility or necessity of their fulfillment.

Modal logic adds the dimension of possibility and necessity to reasoning. It enables the modeling of alternative scenarios, normative rules, and mandatory conditions. For game NPCs, this means that their behavior can take into account not only facts but also obligations, permissions, and potential courses of action, making the system more realistic and flexible.

For this purpose, the following operators are introduced:

- $\Box$ — necessary
- $\Diamond$ — possible

To facilitate NPC behavior modeling, a fuzzy degree of modality is introduced: the extent to which something is necessary or possible. The system can distinguish “strictly necessary,” “most likely possible,” or “barely necessary.” This allows NPCs to make decisions not in a binary manner, but with consideration of the strength of modality.

Deontic Logic

While modal logic describes possibility and necessity in general terms, deontic logic specializes in reasoning about norms and rules. It enables the formalization of obligations, permissions, and prohibitions that govern the behavior of agents.

For game NPCs, this means that their actions can take into account not only facts and possibilities but also normative constraints, making the system more realistic and closer to human reasoning.

Main categories:

- **Obligation** — an action must be performed.
- **Permission** — an action is permissible.
- **Prohibition** — an action is impermissible.

Differences from modal logic:

- “Necessary” refers to objective inevitability (true in all scenarios).
- “Obligated” refers to a normative prescription, which may be violated but entails consequences.

For greater expressiveness, fuzzy degrees of modality can be used. In this case, all deontic components coexist as weighted forces. During logical inference, the strongest prescription will be selected.

We also introduce a division of deontic categories into two dimensions:

- **Objective norms** — rules defined in the knowledge base (e.g., “a soldier is obligated to guard the gate”)
- **Subjective norms** (Self-obligation) — the NPC’s own perception of these rules

This division makes it possible to model not only the existence of rules but also their interpretation by characters, resulting in more realistic and diverse behavior.

Epistemic Logic

Let us add the ability to formalize reasoning about knowledge and belief. This makes it possible to describe not only facts and norms but also what is known or assumed by an agent.

For game NPCs, this means their behavior can take into account not only objective data but also subjective representations: knowledge about the world, belief in events, and imagined roles they adopt.

Main categories:

- **Knowledge** — the NPC knows that a statement is true.
- **Belief** — the NPC considers a statement true, even if it may be false.
- **Imagination** — the NPC accepts a belief “as if” it were true (for example, “imagine you are a king”).

Examples of application:

- The NPC knows that an enemy is behind the wall but does not know that there is cover there.
- The NPC believes that an ally will come to help, although this is not certain.
- The NPC can act within an imagined scenario: “imagine you are a commander” → the NPC adopts the role of commander and begins reasoning as one.

Nonmonotonic Logic

Traditional logic is monotonic. It is built on axioms accepted as true, from which conclusions are derived. Adding new information to such a system can only increase the set of true statements. In other words, adding new knowledge never cancels previous conclusions. They can only expand, but not be reduced.

Thus, monotonicity presents several problems:

- **No revision of knowledge**. New facts cannot cancel previous conclusions, even if they clearly contradict them. In the real world, this leads to the system being stuck in incorrect inferences.
- **Inability to model exceptions**. Default rules (e.g., “birds fly”) cannot be adjusted with exceptions (“penguins do not fly”).
- **Weak adaptability**. The system cannot change its conclusions when new information appears. This makes it inflexible and distant from human reasoning.
- **Problems in dynamic environments**. In games, robotics, or expert systems, facts are constantly updated. Monotonic logic cannot cope with changing conditions.

These limitations can be overcome by applying the principles of nonmonotonic logic:

- **Revision of conclusions**: new facts can cancel or adjust previously made inferences.
- **Default rules**: the system can make assumptions (“birds usually fly”) that remain valid until information about an exception appears (e.g., “penguin”).
- **Exceptions and context**: logic accounts for the fact that rules are not absolute but depend on context.
- **Adaptability**: the system can change its conclusions when new data becomes available.

Nonmonotonic logic transforms the system from a “rigid deducer” into a flexible reasoner, capable of canceling conclusions and adapting to new facts. It enables modeling of common sense, exceptions, and the dynamics of knowledge, making it particularly useful for complex and changing scenarios.

Attention should also be given to Default Logic, proposed by Raymond Reiter in 1980.

It should also be noted that a more rigorous formalism was proposed by Raymond Reiter as an example of formalizing default rules.

For our purposes, it is sufficient to introduce the operator “^:”. With this operator, one can express:

```
A($x) ^: B($x) -> C($x)
```

This means: if A($x) is provable and if it does not contradict knowledge that allows assuming B($x), then C($x) can be inferred.

Nonmonotonic logic is one of the variants of Contextual Constraints, the advantages of which I will examine below.

Meta-Conditions

Up to this point, we have considered facts and rules as always existing and accessible to all participants in the system. This approach is convenient for initial steps, but it oversimplifies reality: in the real world, facts may be relevant only under certain conditions, and access to them may be restricted. In other words, there are meta-conditions that must be taken into account during reasoning.

Introducing meta-conditions brings NPC behavior modeling closer to reality.

It is important to emphasize the significance of distinguishing meta-conditions for greater clarity, expressiveness, and declarativity.

In existing logical programming languages, meta-conditions are written directly into the body of a fact or rule. As a result, the code becomes cumbersome, and the developer experiences additional cognitive load in separating meta-conditions from the main expression.

The introduction of meta-conditions allows the code to be divided: the main expression remains clean, while the context is moved into the meta-layer.

Conditions for the Existence of Facts and Rules

A fact or rule may be relevant only under certain conditions. In the real world, their applicability depends on multiple circumstances: time, location, participant roles, availability of resources, or knowledge of an event. Therefore, in an NPC reasoning system, it is important to define the conditions under which a fact or rule is considered to exist at all. For example, the fact “the shop is open” is valid only during the day, while the rule “patrol” applies only within the fortress.

Access Levels for Facts and Rules

Facts and rules should not be equally accessible to all participants in the system. In the real world, information is distributed unevenly: some knowledge remains internal, some is available only to specific roles, and some becomes publicly known. To reflect this in NPC logic, access levels are introduced to determine who can use a particular fact or rule and under what conditions.

This mechanism enables the modeling of privacy, secrecy, and information visibility. NPCs no longer possess complete knowledge of all facts but act within the scope of what is accessible to them. This makes reasoning more realistic and opens possibilities for strategic concealment or disclosure of facts.

Main access levels:

- **private (default)** — a fact or rule is available only within a specific NPC. Example: “The NPC knows that it has few bullets.”
- **protected** — access is restricted to certain NPC classes or even specific characters. Example: “The secret order is available only to officers.”
- **public** — publicly known facts about an NPC, accessible to everyone. Example: “This NPC is a merchant.”
- **visible** — publicly known facts, but only if the NPC is within the field of view. Example: “The NPC is armed” — visible only if the character is observable.

Conditional Entities

In the process of describing NPC logic, it becomes necessary to work with any object in the game scene that meets certain conditions. However, direct use of scene objects makes rules rigidly tied to a specific implementation.

Therefore, our task is to abstract from direct references to objects by expressing conditions:

- Declaratively
- Independently of a specific game engine

The solution is **Conditional Entities** — objects that encapsulate conditions for matching with elements of the game scene. They serve as a link between the logical description of rules and the actual state of the world.

Conditional entities can be named or anonymous.

Named entities have a unique identifier. They are declared once and can be reused multiple times in different parts of the code.

Anonymous conditional entities are created on the fly and used locally or passed for use where needed.

Binding to game scene objects can be:

- Single-use — the condition is checked once, and the binding does not disappear even if the game scene object no longer meets the conditions.
- Multiple-use — the condition is continuously checked, and the binding can change to a more suitable object.

Data Exchange Operators

For building complex systems, it is important to have a unified mechanism for transferring information between components. Standardized data exchange operators allow:

- Standardizing interaction between heterogeneous sources and receivers
- Simplifying the description of exchange logic in terms of DSL or formal models

Thus, working with data becomes simpler and more expressive.

A data source can be any entity that generates or provides information. A data receiver is a component that accepts and interprets information.

As a starting point, let us list the key types of sources and receivers in the designed system:

- Data storage
- Channel
- Primitive source
- Function
- Generator

Now let us examine each item in detail.

Data storage is any entity capable of retaining data in a defined form. The key property of storage is the ability to hold data and allow repeated retrieval. Consequently, storage can act both as a source and as a receiver of data.

Main types of storage:

- **Knowledge base**. At the current stage, it is assumed that each NPC has one personal knowledge base, used as its default global base. In the future, the possibility of creating multiple knowledge bases within a single NPC is not excluded.
- **Variables**. These can be considered as the simplest storage units, holding individual values.
- **Properties**. These can also be considered as simple storage units, holding individual values.

The concept of a **Channel** has not yet been fully developed. As a prototype, Go language channels are considered.

At present, system channels for output are implemented:

- @>log — a channel for recording diagnostic messages and logging
- @>say — a channel for generating text or voice messages simulating NPC speech

Such system channels help to clean up the namespace from service functions.
In the future, additional channels may be introduced:

- **Channel for runtime code modifications**. A system channel may be implemented through which changes can be made to the executing code during runtime. This channel would serve as an interface for dynamically modifying system behavior.
- **User-defined channels**. A mechanism for creating and configuring channels by users has not yet been developed. In the future, the architecture may be extended to allow defining custom channels for specific tasks.

Potentially, a channel can act as both a source and a receiver of data. However, at the current stage, it functions only as a receiver.

A **Primitive source** of data is represented by a literal. In this case, the data serves as its own source.

Functions and **generators** can be considered special types of data sources. It should be noted that they become data sources at the moment of invocation, when the result of computation or generation is passed further into the system. Otherwise, they can themselves be passed to a receiver as first-class objects.

After defining the main types of data sources and receivers, it becomes possible to formalize the process of exchange itself. At this stage, it is appropriate to introduce special operators that serve as a universal means of describing information transfer.

Since one of the goals is expressiveness and readability, we adopt the elegant syntax of the operators “<<” and “>>” in the sense of stream input/output.

The operators “<<” and “>>” are interchangeable:

- source >> receiver
- receiver << source

The source is always placed on the “blunt end” of the operator, while the “sharp end” points to the receiver. This notation makes the direction of data transfer clear and easily readable.

Thus, we obtain a convenient and clear unified mechanism for data exchange between heterogeneous sources and receivers. The introduced operators allow describing the process of information transfer in a compact and readable form, making component interaction transparent and consistent.

GOAP and HTN

At this stage, the system is already quite capable. However, applying it in practice may lead to a mixture of rules, triggers, methods, and similar constructs. Such unstructured code appears inconsistent for a project striving for declarativity and expressiveness. Moreover, it is difficult to debug and maintain.

The solution is to use more structured approaches to action description: HTN (Hierarchical Task Networks), BT (Behavior Trees), FSM (Finite State Machines), and GOAP (Goal-Oriented Action Planning).

Among these structured approaches to action planning, I am particularly interested in **HTN (Hierarchical Task Networks)** and **GOAP (Goal-Oriented Action Planning)**. Both possess qualities that help avoid chaotic accumulation of rules and conditional operators while preserving the expressiveness and declarativity of the system.

HTN provides a strict hierarchical structure, where complex goals are decomposed into subtasks and atomic actions. This approach makes behavior transparent, modular, and manageable, while preventing the exponential growth of complexity typical of finite state machines.

GOAP allows agents to dynamically select goals based on preconditions and effects, making the system flexible and adaptive to environmental changes. This approach enhances expressiveness, as behavior is formed not rigidly but according to the current context.

Together, these approaches form a hybrid model: GOAP is responsible for goal selection, while HTN ensures structured execution. This combination merges adaptability and flexibility with transparency and modularity, overcoming the limitations of “spaghetti code” as well as traditional FSM and BT.

HTN and GOAP are extensive topics. Below, only the fundamental principles are outlined.

At the core of HTN lies the concept of a task: primitive or compound.

Primitive task — a final atomic step that an NPC can perform directly without further decomposition. It requires no additional planning. Main components:

- **Operator** — a method invoked when executing the task. It serves as the link between HTN and imperative code.
- **Preconditions** — conditions that must be satisfied for the action to be possible. Example: “NPC holds food in hand.”
- **Effects** — changes in the state of the world after the action is executed. They allow reasoning about the future. Example: “The agent is no longer hungry.”

I am also exploring the possibility of optionally specifying required resources in a domain-oriented manner.

Compound task — a high-level goal that cannot be executed directly. It requires decomposition into simpler steps.

A compound task may contain one or more execution methods. Typically, an execution method is called a “method,” but I prefer to call it a “case.” Each case contains a list of subtasks and optionally preconditions. A compound task may also contain preconditions common to all its cases.

In addition to regular cases that describe different scenarios for achieving a goal, the system may include special cases.

Before-cases are always executed before the main scenario. Their role is preparation: checking state, equipping weapons, gathering necessary resources. They act as a prologue, ensuring that the agent enters the case in the correct state. For example, before attacking an enemy, the EquipWeapon case may be executed.

After-cases serve as an epilogue. They are executed after the main scenario and handle consequences: updating inventory, clearing state, recording results. For instance, after trading with a merchant, the UpdateInventory case may be executed to synchronize changes.

Before- and After-cases help avoid duplication of subtasks common to all main cases of a given compound task. Such shared subtasks can be moved into these special cases.

Background-cases run in parallel with the main scenario and may have conditions fully analogous to those in imperative triggers. For example, during shooting, the agent may periodically execute the AimPeriodically case to adjust aim or MonitorAmmo to track ammunition. These cases create the impression that the NPC is not merely executing a linear plan but continuously adapting to a changing world.

To keep NPC behavior manageable and optimizable, compound tasks can be divided into several levels. Each level reflects the degree of abstraction and depth of elaboration:

- **Root** — the root task that defines the agent’s global goal. Examples: “Survive,” “Guard the territory,” or “Escort the player.” The root task sets the direction of all behavior.
- **Strategic** — strategic-level tasks. They describe large-scale plans: “Find resources,” “Eliminate threat,” “Establish trade.” At this level, the agent considers long-term goals without yet descending to specific steps.
- **Tactical** — tactical tasks. These are more concrete scenarios: “Gather firewood,” “Patrol the area,” “Attack the enemy.” The tactical level determines the approach in the current situation.
- **Compound** — low-level compound tasks, close to primitive actions. Examples: “Approach the tree,” “Select a weapon,” “Take a shot.” Compound tasks serve as a bridge between tactics and primitive actions.

This structure is necessary for:

- **Goal separation by levels** — enabling NPC behavior to be conceived as a multi-level system, from global strategy to specific steps.
- **Execution optimization** — if the NPC is far from the player or invisible, low-level compound tasks need not be calculated. They can be replaced with more primitive implementations (e.g., “simulate result” instead of detailed

animation). This reduces system load while maintaining plausible behavior.
- **Scalability** — adding new strategies or tactics does not break the structure but simply extends the task tree at the appropriate level.

At this stage of system development, HTN provides structure: tasks, cases, levels. However, it has a limitation — goal selection is always externally defined. The NPC executes what is prescribed but does not decide what is most important at the moment. GOAP solves this problem: it allows the agent to independently select the current goal based on the state of the world.

Here arises a natural idea: why not use the already described HTN tasks as GOAP actions? After all, we already have primitive tasks with preconditions and effects, compound scenarios, and cases. All of this can be reused as an “action library” for the GOAP planner.

Reusing HTN components:

- **HTN tasks → GOAP actions**: each primitive HTN operator already contains preconditions and effects. This is exactly what GOAP requires for plan construction.
- **Compound tasks → goals**: strategic and tactical HTN tasks can be interpreted as GOAP goals. The planner will select which is relevant and then expand it through HTN cases.
- **Special cases → GOAP context**: before/after/background can be integrated as additional conditions or background processes accompanying plan execution.

GOAP in the designed system primarily provides:

- **Continuous replanning mechanism** — attempts to generate a more optimal plan when the state of the world changes
- **Management of plan success criteria** — criteria are defined to determine what constitutes the most suitable plan in the current context

Thus, if the goal is “survive at all costs,” certain plans will be considered most successful. If the goal is “eliminate the enemy at all costs,” then with the same resources, entirely different plans will be recognized as successful.

The goal can also be changed at any time, for example, by command. This makes the system more dynamic.

States

In the previous section, a choice was made in favor of a hybrid of GOAP and HTN, leaving FSM aside. This choice is explained by the limitations of the classical finite state machine: it handles simple scenarios well but scales poorly and does not allow flexible control of NPC behavior logic.

Nevertheless, the concept of state remains important. In the real world, state defines the context of human behavior. Behavior appropriate in one state may be less appropriate in another, and entirely inappropriate in a third, under otherwise identical conditions.

Introducing state as a context in which goals, tasks, facts, and rules operate allows NPC behavior to be modeled more closely to the real world. A state becomes not merely a node of an automaton but a full-fledged frame that unifies all elements of reasoning and planning into a single system.

Applied to GOAP and HTN with FSM, each state functions as a meta-context:

- From FSM, it inherits the structure of transitions and events that trigger mode changes
- From GOAP, it receives a set of goals relevant only within the given state
- From HTN, it provides tasks and decomposition methods available within this context

Changing state restructures not only the NPC’s mode but its entire reasoning system: facts, rules, goals, and tasks are reactivated according to the new context. This enables NPCs to act not according to a rigid scheme but to adapt to the situation, while maintaining the structural clarity of FSM.

States may be nested or have levels of granularity. For example, the state “war” may be subdivided into “cold” and “hot” phases, each with its own rules and tasks.

Modularity

Let us examine Modularity in more detail. It is not a key feature or primary objective of the designed system, unlike programming languages with explicit modularity. Nevertheless, it has a certain influence on the design of the DSL.

First, let us focus on the technical aspects of modularity. Above all, this concerns code reuse and the resulting benefits: reduction of duplication, simplification of maintenance, and increased architectural robustness.

Undoubtedly, classes can serve the role of modules. In object-oriented languages, classes and objects perform the same tasks that modules previously addressed: they provide encapsulation, reuse, and access control.

For greater convenience in organizing code, we introduce the concept of a Library. A library is understood as an organized set of code entities (classes, functions, facts, rules) intended for repeated use across different projects. A library corresponds to a Library Project in C#, C++, or Java.

A library can be connected through a package manager, similar to NuGet packages in .NET or npm packages in JavaScript. This approach allows developers to reuse their own work more conveniently, and in the future, to integrate external solutions published by the community.

A library is a purely technical solution for ensuring modularity. However, an NPC cannot reason about it.

We extend the library with semantic meta-information, enabling NPCs to reason about it. Such an extended library will be referred to as a module, to distinguish it from a regular library.

The presence of meta-information allows an NPC to analyze a module and decide whether to connect it, block its connection, or disable a previously connected module.

For example, an NPC may reason:

- This book describes trading rules; therefore, it is useful in this region.
- This is information about secret passages in the enemy castle; hence, it will be useful for completing the mission.

But the opposite may also occur:

- This book describes secret combat techniques, while my belief rejects violence.
- This is a demonic book; it must not be opened and should be destroyed immediately.

For greater domain orientation, programming a module can be approximated to writing a book by introducing corresponding syntax.

Neural Networks

Despite the fact that neural networks have long successfully solved tasks beyond the reach of symbolic approaches, and despite the current boom in LLMs, neural networks remain peripheral to my focus and interests.

All neural networks have certain drawbacks:

- **Neural networks remain a "black box".** A neural network may produce an unexpected result that disrupts balance or breaks the narrative. Methods for verifying neural network reasoning are still limited and difficult to apply.
- **Lack of guarantees.** Game mechanics require predictability: the player must understand the rules.

Public LLM services provide impressive capabilities but also have significant disadvantages:

- **Dependence on third-party services.** Public LLMs are highly likely to involve substantial costs for end use and carry the risk of sudden shutdown or service blocking.
- **Unpredictability.** Hallucinations and strange NPC responses may appear humorous but more often break atmosphere and storyline. Game designers need to control narrative branches and dialogues. LLMs create chaos instead of a managed narrative.

At the inception of the project (2014), small neural networks were widespread, which had the following drawbacks:

- **Rigid task binding.** In games, such networks could recognize gestures, faces, or objects, but this was too narrow and provided little value for game design.
- **Limited flexibility.** For each new mechanic, a separate network would need to be trained, which was unjustified in development.

Nevertheless, many tasks cannot be solved, or are almost unsolvable, by other approaches. Therefore, when designing the system, possible integration with neural networks must be considered.

Running LLMs on the user's computer appears attractive. Currently, this applies to Small models (2B–7B), but with increasing availability of RAM, Medium models (13B–14B) will also become easily accessible.

Promising directions include:

- Using a neural network as a nonlinear function
- Applying Neuro-Symbolic AI approaches

Natural Language Processing

Today, Natural Language Processing (NLP) technologies have already become commonplace and are gradually being applied in games. The ability to speak with characters or issue commands in free form turns gameplay into a more natural and engaging experience.

The greatest potential of NLP technologies is revealed in genres where interaction is built around commands, dialogue, and management. Primarily, these include:

- **Military simulators**
- **Management simulators**
- **RPG and adventure games**

In the system under development, two types of natural language interaction can realistically be implemented:

- **Short voice commands**
- **Documents**

Short voice commands are one of the most obvious and practical applications of NLP in games. A player can issue orders or express intentions in free form: "cover me," "take position on the left," "hold the defense," "occupy the house on the left and hold it until reinforcements arrive." This allows NPCs to respond to commands as naturally as human players, creating the effect of real team interaction.

The main advantage of this approach is immersion and speed. The player does not waste time selecting menu items or hotkeys but interacts with bots in the same way they would with people.

Another, more "strategic," application of NLP is **management through documents**. The player formulates textual orders, reports, or plans in free form, and NPCs execute them. For example:

- In a military simulator, the commander writes a "combat order"
- In economic strategies, the player formulates directives, plans, or orders
- In role-playing games, these may take the form of contracts, agreements, or decrees

This approach enhances realism and depth of management. The player feels not merely like an interface operator but as a direct participant in events, composing authentic documents.

Documents can also become artifacts of the game world, which may be lost, found, or intercepted.

Here, the ability of NPCs to analyze modules and subjective norms becomes useful. An NPC can analyze a received or discovered document and decide whether to execute it or simply retain the information.

NLP integrates with the knowledge base without requiring any additional changes to the system.

Text is converted into facts. The system then operates with these facts.

Knowledge base triggers allow control over fact import: whether to execute the received order or simply store the facts from it.

NPC voice responses are carried out by outputting a fact to the "@>say" channel. The engine automatically converts the output fact into text or speech.

Additional Features

In addition to its core functions, the system includes a number of supplementary mechanisms. Their significance is secondary; however, they can enhance the flexibility and expressiveness of the architecture. In this section, I will limit myself to a brief description.

Reflection allows the system to examine its own structure and behavior during execution.

Metaprogramming enables:

- On the one hand, extending customization capabilities for parsing and compilation using the DSL itself

- On the other hand, adding mechanisms of adaptation and self-development to the system

User-defined operators allow the language of the system to be extended for specific tasks, making it more expressive and closer to the domain. Instead of being limited to a standard set of syntactic constructs, the developer can introduce new forms of notation that reflect the specifics of the modeled logic. This reduces the gap between technical implementation and the semantic level, enabling rules and interactions to be described in the most natural way possible.

Implementation Architecture

The DSL is implemented as an interpreted language, since this provides:

- Full control over code execution
- The ability to modify object structures at runtime

The runtime environment is written in C#, as the use of a language with automatic memory management allows focus on implementing DSL features without the need to concentrate on low-level details (primarily memory management). C# offers sufficient performance for experiments with DSL and small demo scenes.

In the future, the runtime environment may be rewritten in C++ or Rust.

The runtime environment consists of the following components:

- **Engine (interpreter)** — the core of the system. Executes DSL. Written in .NetStandard 2.0
- **Unity component set** — integration of DSL functionality into Unity scenes. Written in .NetFramework 4.7.1
- **CLI (Command Line Interface)** — initialization of new projects, package installation, execution of DSL code from the command line. Written in .NET 9
- **Plugins** — non-core but necessary code for the project. Minimal implementations planned to be replaced in the future. Written in .NetStandard 2.0

All project components are distributed under the MIT license.

The interpreter is based on a custom parser implemented in C#. This approach ensures full control over the DSL grammar and allows free experimentation with syntax.

During interpreter design, options for using existing parser generators (ANTLR, yacc/bison) and the LLVM infrastructure were considered. However, it was decided not to use them due to concerns about their limitations in syntax experimentation. Additionally, the GNU GPL license of yacc/bison is incompatible with MIT.

In the future, I plan to reconsider the use of ANTLR and LLVM.

The DSL is compiled into bytecode, which is executed in a virtual stack-based machine (stack-based VM).

JIT compilation is not yet used but is planned for the future.

Monitoring and Debugging

At the current stage, the system has only a basic logging mechanism. Events are recorded in JSON format and saved to disk. A separate utility then transforms them into a text log with filtering applied, enabling post-factum analysis of system behavior.

This approach provides the minimally required level of observability but remains limited: there is no capability for real-time monitoring or interactive debugging.

Several directions for further development are proposed.

Storage format optimization: transition from textual JSON files to binary formats (Avro, Protobuf). This will reduce data volume, accelerate serialization and deserialization, and provide a strict schema for event evolution.

Real-time monitoring: organization of an event stream with visualization and on-the-fly filtering, allowing system state to be tracked without delays.

Debugger: creation of a tool for interactive analysis. It should support replaying event sequences, setting breakpoints, and viewing system state at the moment an event occurs.

Initially, the debugger should be integrated into Visual Studio as an extension, providing a familiar interface for step-by-step debugging and system state analysis during game development.

In the future, the emergence of a dedicated IDE for this DSL with an integrated debugger is not excluded.

Performance Considerations

The system's performance is significantly lower than that of industrial-grade interpreted languages. This is due to each NPC containing multiple subsystems with complex and, at present, poorly optimized algorithms.

At the same time, there is considerable potential for optimization:

- Caching computation results
- Applying more efficient algorithms
- Simplifying computations for non-visible NPCs
- Introducing JIT compilation
- Eventually rewriting the engine in C++ or Rust

The increasing power of modern computers temporarily compensates for high resource consumption but does not eliminate the need for optimization.

At present, the primary focus is on expanding functionality and improving development convenience. Performance optimization is considered an important task for future stages, but the current priority remains the implementation of the system's key features.

Additional Architectural Aspects

NPC Isolation

Each agent (NPC) in the system is isolated — it has no direct access to the game scene or to the data of other NPCs.
NPC behavior is formed based on its own knowledge and data received from perception channels.

NPCs interact with scene objects through **conditional entities** — abstract references defined by specified conditions. The engine interprets these conditions, finds a suitable object, and binds it to the conditional entity.

Advantages of this approach:

- Prevention of cheating AI
- Simple implementation of stealth mode
- Ability to model acquisition, forgetting, and transfer of knowledge
- Simplifies testing and debugging. An NPC can be launched in isolation, with controlled input and output of information

The drawback of this approach is the complexity or even impossibility of implementing game mechanics that critically depend on an NPC having complete knowledge of the game scene.

Platform Abstraction

The system architecture is based on the principle of independence from a specific engine or runtime environment.

The engine (core) is implemented separately and built on .NetStandard 2.0, which ensures the possibility of integration into any .NET-based solutions without requiring modifications to the core logic.

In the future, integration with solutions in other languages may be possible:

- Through cross-language mechanisms, including deploying the engine as a server application and accessing it via interprocess protocols
- Implementing the engine in C++ or Rust, which would allow it to be provided as a native library and linked to environments not built on .NET

Thus, platform abstraction ensures both compatibility within the .NET ecosystem and the possibility of extension beyond it, creating a foundation for the long-term universality of the architecture.

Integration with Unity

Unity is considered the primary platform for prototyping and practical implementation. The architecture provides adapters and API layers that ensure proper interaction between the system core and Unity components. At the same time, NPC logic remains independent of the engine, which preserves architectural integrity and facilitates solution portability.

Other Possible Platforms

Thanks to modularity and platform abstraction, the system can be integrated not only into game engines but also into other technological environments. This broadens the scope of application and makes the architecture more versatile.

Robotics control. The architecture can be used for integration with robotic control systems. Primarily, I am interested in hobbyist robotics on platforms such as Raspberry Pi or Arduino.

Expert systems. The system can be adapted for building expert systems. The modularity of the architecture allows different rule bases and scenarios to be connected while maintaining a unified knowledge representation format.

Standalone knowledge base. The core can function as an independent knowledge base, accessible through an API or interprocess communication. In this case, it becomes a universal repository of facts and rules that external systems can access — whether game engines, robotic systems, or expert applications.

Project Status

The project has been developed by a single developer since May 2014. However, its progress has occurred exclusively during free time and has been accompanied by long pauses.

Often these pauses served as time for reflection on the project — an opportunity to look at it from a different perspective and, in many cases, to reconsider key ideas. But frequently, the pauses were occupied with everyday matters, and work on the project was suspended.

Since 2020, the project has been called SymOntoClay (**Sym**bolic **onto**logical **clay**).

At the current stage of development, the DSL features have been fully or largely implemented:

- Core constructs (statements) and imperative code operators
- Knowledge base and logical inference
- OOP, but without traits
- Triggers, including those for adding new facts
- Imperative-logical dualism of inheritance
- Imperative-logical dualism of properties
- Knowledge base query operators (basic implementation)
- Implicit knowledge base queries in conditions
- Fuzzy logic
- Deontic logic (including subjective norms for NPCs)
- Access levels for facts and rules
- Conditional entities
- System channels "@>log" and "@>say"
- Data exchange operators
- HTN
- States (basic mechanisms)
- Libraries
- Basic implementation of Natural Language Processing based on ATN

A standard library package is gradually beginning to take shape.

Currently, its packages are available only from the local cache on the user’s computer. At this stage, I am not prepared to create and administer a dedicated server for a single package intended for one user’s computer (my own).

As of today, the project has two main entry points: a set of Unity components and a command-line interface (CLI).

The Unity component set allows NPCs to be run using code written in SymOntoClay DSL.

The CLI provides:

- Initialization of a new project
- Installation of packages from the local cache on the user’s computer
- Execution of DSL code from the command line

As practical examples, small scenes with one or two NPCs have been created to demonstrate basic capabilities. However, a fully playable demo scene does not yet exist.

Detailed documentation is available only for DSL syntax.

There is basic documentation for usage in Unity and CLI.

Documentation for other components and aspects of the system is either absent or in a very early stage.

The project remains under development: it has no users and no established community, making it entirely an author-driven and research-oriented initiative.

What’s Next

SymOntoClay is still far from completion. Many features have already been implemented and are available for use, but much remains to be done. In fact, such a project can only be discontinued, not truly finished.

Development plans are easier to group by levels, covering different aspects and directions of progress:

- Technical evolution
- Practical application
- Community creation and development

SymOntoClay is evolving as a DIY project: I work on it alone, exclusively in my free time. Therefore, the pace of development is rather slow.

To outline the direction of progress, I divide plans into three horizons:

- **Near-term steps (≈ 1 year)** — priority tasks that can realistically be accomplished in the near future
- **Mid-term tasks (2–3 years)** — tasks of lower priority, as well as those dependent on the completion of near-term tasks
- **Long-term horizons (3+ years)** — strategic goals achievable only through sustained development

Technical Evolution

Near-term steps (1 year):

- Implementation of spatial logic
- Completion of HTN (Hierarchical Task Network) development
- Basic implementation of GOAP (Goal-Oriented Action Planning)

Mid-term tasks (2–3 years):

- Serialization into an image and deserialization from an image
- Full implementation of extended logic programming
- Further development of NLP (Natural Language Processing)

Long-term horizons (3+ years):

- Implementation of all remaining planned functions and features

Practical Application

Near-term steps (1 year):

- A minimal playable demo scene in the battle royale genre

Mid-term tasks (2–3 years):

- A minimal playable demo in the tactical shooter genre with voice control

Long-term horizons (3+ years):

- The first game created by an external developer using SymOntoClay
- Expansion of SymOntoClay application into genres with more complex logic: tactical simulators, strategy games
- Gradual extension beyond the gaming domain: robotics, expert systems

Community Creation and Development

Near-term steps (1 year):

- Preparation of the project website and extended documentation: detailed descriptions, usage examples
- Creation of a Discord channel for communication and support of initial discussions
- Publication of initial materials (articles, notes, posts) introducing the audience to SymOntoClay and inviting discussion on open platforms

Mid-term tasks (2–3 years):

- Emergence of the first community members who begin following the project and discussing it
- Initial experiments by participants with SymOntoClay: test scenes, trial projects, feedback
- Development of the website and documentation
- Regular publications to maintain interest and inform about progress
- Launch of a subreddit to expand the audience
- Emergence of the first independent community materials (articles, posts, videos) describing SymOntoClay

Long-term horizons (3+ years):

- Formation of a stable community around SymOntoClay
- Emergence of third-party games and applications built on SymOntoClay
- Appearance of the first contributors
- Expansion of the community beyond the gaming domain

Conclusion

I have proposed an architecture for describing NPC behavior, based on borrowing effective solutions from different languages while remaining free from the constraints of their syntax.

At its core lies a knowledge representation system that provides powerful logical inference capabilities. By itself, it does not simplify the writing of behavior, but it establishes a foundation for more expressive tools.

To make the development process more convenient, I introduced a specialized DSL. Its initial version already includes many features familiar from other languages, but unified within a single system specifically oriented toward describing NPC behavior. This approach raises the entry threshold but enables indie developers to create more complex behavior models with less code.

In the future, the DSL may be ported to different platforms, opening opportunities for use not only in games but also in other domains where agent behavior modeling is required. Performance issues will be addressed both through implementation optimization and the natural evolution of computational power.

Thus, SymOntoClay represents a step toward greater expressiveness and flexibility in behavior description, reflecting the natural evolution of ideas in the field of game worlds and artificial intelligence.