Reasoning with Ontologies for Non-player Character’s Decision-Making in Games

Sylvain Lapeyrade
Université Clermont Auvergne, CNRS, LIMOS, France
sylvain.lapeyrade@uca.fr

Abstract
In most games, the decision-making of non-player characters (NPCs) is usually constructed using variants of state machines, behaviour trees, utility-based AI or planning. These methods are relatively simple to implement, but have drawbacks in that it can be difficult to create complex non-hard-coded behaviour for many agents and to maintain the algorithms, especially when scaling up. Game designers usually think of their games with rules that closely resemble logic rules. A methodology is introduced to design both general and modular behaviour using a logic reasoner with hierarchical ontologies. This approach is combined with the well-founded semantics (WFS) to solve the problem of representation and reasoning despite the lack of NPC knowledge.

Introduction
In video games, non-player characters (NPCs) often have a very basic decision-making process which leads to behaviours with a low level of credibility and therefore a less intense experience for the player. This manifests itself in overly simple or repetitive behaviour and inconsistent actions. This problem can be observed with NPCs in commercial games, virtual assistants in educational or serious games, or with virtual agents in simulations.

The game artificial intelligence (AI) research community has made ongoing efforts to try to identify the reasons for this lack and to improve the credibility of NPCs (Yannakakis and Togelius 2018; Millington 2019).

Game Development Policies
The basic decision-making regarding NPC behaviour is partly explained by game development policies. The video game industry faces strict resource constraints, game designers want to control the game experience and the player does not always want believable NPC behaviour.

Industry resource constraints Game development is a complex process where deadlines are often tight (Borg et al. 2020), human resources are limited, and hardware optimisation is very important. The game must be ready as soon as possible, at the lowest possible cost, and have the lowest material requirements possible.

These resource constraints can lead to pressure to impose periods of intensive work for developers during the development of a game, known as ”crunch time”. These periods logically lead to a decrease in the quality of their work. (Brogan 2021). This situation gives studios a strong incentive to reuse as much code as possible from previous games, use relatively simple algorithms and ultimately leaves little room for innovation (Schmalz 2015).

Control the Game Experience Game designers may wish to keep as much control over their game as possible, to ensure a smooth and enjoyable experience for the player. Limiting the player’s action allows for better control of the flow of the game and thus avoids more inconsistent behaviour or bugs at the expense of the player’s freedom of action. Decision-making algorithms that can generate a large number of complex behaviours are therefore more likely to also generate behaviours not intended by the game designer.

Simplicity Can Be Desired Simple, even stereotypical, behaviours can be implemented on purpose. Indeed, NPC tasks can sometimes be very simple, so there is no need to code these behaviours in a complex way, in an environment where computing resources are very precious.

Moreover, NPCs can appear very intelligent while resulting from very simple algorithms and vice versa, this is the complexity fallacy (Millington 2019). This is even more common when the NPC is only seen for a short time. It is then better to choose the right behaviour rather than systematically the most complex.

Furthermore, it is important that the AIs are not too strong compared to the players, especially beginners, as the goal of the NPC is seldom to be as strong as possible but to be as fun as possible for the player.

Overview of Decision-Making in Game AI
The most popular methods for decision-making in game AI, according to (Yannakakis and Togelius 2018; Millington 2019; Simonov, Zagarskikh, and Fedorov 2019) include Finite-State Machine (FSM), Behaviour Trees (BT), Utility Based AI and Action Planning techniques. All of these techniques have shortcomings, whether in terms of emergent behaviour generation, scaling up, the need for a lot of data or calibration, or the computational power required for behaviour generation.
Reasoning With Ontologies

It seems surprising that none of these decision AI approaches are based on logical reasoning. Yet symbolic logic programming is an important part of AI and has proven its effectiveness for several decades in various AI applications such as natural language analysis (Pereira and Warren 1980), compilers (Van Roy 1990), database management (Ceri, Gottlob, and Tanca 1990), the semantic web (Berners-Lee, Hendler, and Lassila 2001) and expert systems (Jackson 1999). According to Peter Jackson (1999), expert systems are software that emulates the decision-making ability of a human expert, which is very similar to what we are aiming at. Therefore, is it an adapted AI method for NPC decision-making?

Is Declarative Programming Too Complicated? Prolog (Colmerauer and Roussel 1996), one of the most popular logic programming languages, is a declarative programming language. This means that the programmer can express the rules of the game in a declarative way as opposed to imperative languages such as C, Python or Java where the rules are written in a procedural way. Imperative languages are undoubtedly more popular than declarative ones, especially among game developers where the most popular game engines Unity (Unity Technologies 2022) and Unreal Engines (Epic Games 2022) use respectively C# and C++ for scripting. But the same can be said for action languages such as STRIPS or PDDL which are used for planning. With a proper methodology and a beginner-friendly interface, such as a game engine plugin, game designers should be able to use logic programming without having to be a Prolog expert. However, this will not guarantee mass adoption of the technique, as other declarative programming tools such as EmbASP (Calimeri et al. 2018a) and Potassco (Gebser et al. 2011) offer beginner-friendly interfaces and documentation but are not very popular.

Rules-Based Systems The AI approach to modelling NPC behaviour that we have found most similar in principle to the way logic programming works are rule-based systems. They are described by Millington as "the most complex non-learning decision makers [covered in his book] [...] a formidable programming task that can support incredible sophistication of behaviour. It can support more advanced AI than any seen in current-generation games" (Millington 2019). As exposed in the last section, Millington also points out that the main weaknesses of rule-based systems are the difficulty of writing good rules, known as the knowledge acquisition problem. This makes them more difficult to use compared to behaviour trees or state machines that can be directly created from popular game engines. This is why, despite some attempts, including that of (Horswill 2015) and (Wright and Marshall 2000), they are not very common. The issue does not seem to be the quality of the approach but the handling of the algorithm.

Advantages of Using Logic Programming

The main advantages of using logic programming over simple rule-based systems is the utilisation of backward chaining (Nilsson and Maluszynski 1995) and knowledge inference through an inference engine. The use of an appropriate methodology for constructing ontologies may also address the problem of knowledge acquisition described in the last paragraph. Logic-based AI also allows for a very easy and complete explanation of the results, unlike e.g. learning-based techniques, which can be very useful in explaining the AI’s behaviour to the player.

Knowledge Inference An inference engine like Prolog can also deduct new facts from the facts already in the knowledge base. It works by combining available data and inference rules to extract more data until a specified goal is reached. Every fact does not have to be known as inference rules can be used to derive them. This again emphasises the need to have the most precise rules and ontologies possible.

Planning with Backward Chaining An inference engine like Prolog can do planning using backward chaining. Backward chaining aims, via a deep first search algorithm, to find the conditions necessary to fulfil the conditions of a given goal (Russell and Norvig 2021). By giving the inference engine the goal that the agent is trying to reach, it will be able to return the set of sub-goals (e.g. actions) to achieve the main goal and thus lead to an intelligent action sequence. This is very powerful since the sequences are not hard-coded by the game designers and potential sequences not imagined by the game designer may emerge. However, these behaviours are framed by the rules declared by the game designer and, if the rules are well defined, should not result in inconsistent actions.

Hierarchical Ontologies

An ontology is the structured set of terms and concepts representing the meaning of a field of information, such as the elements of a knowledge domain. They are used in Knowledge Representation and Reasoning (KRR) so that agents can represent their knowledge about the world and make reasoning.

Our ontologies are organised as hierarchical packages, like in OOP and its principle of encapsulation. Only specific parts of the ontology are accessible from other ontologies. This is to have generic and modular ontologies and to help the developer to know how to use the different ontologies.

Representing False and Unknown Facts

Prolog is based on the Close-World Assumption (CWA). The CWA is the presumption that a true statement is also known (i.e. present in the knowledge base or derivable from the knowledge base) to be true. Conversely, a statement that is not known to be true is considered as false. Therefore, any statement of which we have no knowledge, or which cannot be proven, is evaluated as false. This is known as the Negation As Failure (NAF) inference rule (Nilsson and Maluszynski 1995). Furthermore, in logic programming, one cannot assert false facts, or rules that lead to false facts. This inability to differentiate between facts that are false because they are factually false and facts that are false because they are unknown is restrictive for the modelling of agents’ knowledge and for the quality of their reasoning.
to be true out leading to inconsistency, the cases where facts are known
the solutions we used in a prototype. For the game engine,
vironment is integrated with a game engine. In brackets are
Figure 1 shows how a logic programming development en-
Global Architecture
tain and take the
other and choose its truth value, or choose to remain uncer-
undefined. This indefiniteness can be used to represent the
as possible, although some of the consequences may remain
tradictions and proceeds to derive as many two-valued facts
Ross, and Schlipf 1991) to circumvent the presence of con-
formation from several different sources. The WFS provides
A Third Truth Value  The WFS allows to manage, with-
true and false at the same time. This type of case is
Unless Model Semantics makes it possible to model
simply the logical negation (i.e. to indicate explicitly that a
statement is false) but generate multiple models for each
query. A similar piece of work to what we are trying to do has been undertaken in (Calimeri et al. 2018b), where an
ASP framework has been integrated in Unity to use rule-
base systems and planning.
However, the WFS was preferred because it only gener-
ates one model and introduces a third truth value for undefined values (Nilsson and Maluszynski 1995). One contribu-
tion of this work is to use this third truth value to represent the absence of knowledge.
A Third Truth Value  The WFS allows to manage, with-
out leading to inconsistency, the cases where facts are known to be true and false at the same time. This type of case is
particularly possible when the NPC receives conflicting in-
formation from several different sources. The WFS provides the possibility for Well-Founded Partial Models (Van Gelder, Ross, and Schlipf 1991) to circumvent the presence of con-
tradictions and proceeds to derive as many two-valued facts as possible, although some of the consequences may remain undefined. This indefiniteness can be used to represent the
unknown facts according to some deduction rules. The NPC can also choose to give more credence to one source than the
other and choose its truth value, or choose to remain uncer-
tain and take the undefined truth value for this fact.

Global Architecture
Figure 1 shows how a logic programming development en-
vironment is integrated with a game engine. In brackets are
the solutions we used in a prototype. For the game engine,
we used the Unity game engine arguably the most popular
game engine. As, for the Prolog environment, we chose
SWI-Prolog (Wielemaker et al. 2010) which is one of the
most popular Prolog environments and one of those that im-
plement the most features. One of the features we were par-
ticularly interested in was the ability to use Prolog from an-
other programming language. SWI provides interfaces for
this, especially for C#1 used by Unity and C++2 used by the
Unreal Engine.

To use the Prolog interface from Unity, we just need to
import the interface DLL file into Unity Plugins and call the
interface functions in a C# script. We personally decided to
separate the code that interacts directly with Prolog into a
separate file from the rest of the game code. This is to make
the code more modular, and to be able to integrate the inter-
face into an existing game. Putting the script code interacting
with Prolog in a library would allow a game designer to use
the interface knowing only a minimum of Prolog.

Future Work
We are currently developing a proof of concept of our ap-
proach. This game prototype is developed on Unity, using
SWI-Prolog and allows us to test the basics of our method
in a simple game unit.

We would like to develop an interface that can be easily
used by game designers, like a plugin to be integrated di-
rectly into the game engines. They will then be able to eas-
ily declare their ontologies without the need for declarative
logic programming knowledge.

We are currently in a research collaboration with a game
studio for the creation of a commercial video game. The stu-
dio’s developers have no experience in declarative logic pro-
gramming, so we will be able to see how well they master
the approach and correct any difficulties they encounter.

It will also be a great opportunity to test the approach
against the demands of a commercial game. Namely, the
presence of several NPCs that may have, depending on the
type of game, relatively long action plans. All the more so as
these action plans will have to be generated within a time-
frame that respects the real time constraints of most games.
Faced with the constraints of material resources, the calcu-
lations for the AI will also have to consume as few resources
as possible. It will allow to push the limits of our prototype
and truly exploit the potential of our approach.

Finally, a real commercial game will allow us to have
feedback from real players on the quality of the AI’s de-
cisions and to have potential new leads for improvement.

Acknowledgments
This research was funded by the French National Research
Agency (ANR) and the European Regional Economic De-
velopment Fund (FEDER). Jan Wielemaker, the lead main-
tainer of SWI-Prolog, is acknowledged for his helped with
the use of the WFS.

---

1C# Interface: https://github.com/SWI-Prolog/contrib-swiplcs
2C++ Interface: https://github.com/SWI-Prolog/packages-cpp
References


Wright, I.; and Marshall, J. 2000. RC++ A Rule Based Language for Game AI. In International Journal of Intelligent Games & Simulation - IJIGS, 42–.