NARS-Python v0.3 — Technical Overview

Christian G. Hahm

Temple AGI Team
Department of Computer & Information Sciences
Temple University
Philadelphia, PA, USA
`christian.hahm@temple.edu`

Abstract. *NARS-Python* is an open-source implementation of a Non-Axiomatic Reasoning System (NARS) programmed in Python. This paper describes the overall architecture, control mechanism, and functionalities of *NARS-Python* v0.3, as well as some ideas for future work.

Open-source system (Python): `https://github.com/ccrock4t/NARS-Python`

1 Introduction and Relevant Works

NARS-Python is a Python implementation of a Non-Axiomatic Reasoning System (NARS), as based on NARS theory as proposed by Dr. Pei Wang [3]. This NARS project started in October 2020, and its design is inspired by the two most mature NARS projects *OpenNARS* [5] and *OpenNARS for Applications* (ONA) [1]. NARS systems all share a common element, which is their use of Non-Axiomatic Logic[1] (NAL). However, these systems can be quite intricate and there is room for subjective design decisions, meaning every NARS implementation will have its own unique architecture, control mechanism, and behavior. The programming language used is more or less unimportant when it comes to NARS functioning, although every language has its own strengths and weaknesses. In this case, Python was chosen for its strengths: modern, readable, easy to import libraries (if needed). Python is weak in some aspects, such as speed and memory efficiency. However, the hardware of current modern computers is already quite capable of running a NARS-Python instance. Additionally, issues regarding slow computation will become less relevant in the coming years as computing hardware improves[2].

The *NARS-Python* **v0.3 alpha** implements many critical aspects of NAL 1-5, and some aspects of NAL-7 (events) and NAL-8 (goals). At this point in the system's development, it is simply equipped to accept inputs, parse Narsese grammar, and use some basic NAL inference rules (syllogistic rules, compositional rules, immediate rules, etc.). The grammar code enforces the use of valid Narsese, and the code defining custom NARS objects enforces the use of standard components (such as *buffer*, *Task*, etc.) throughout the project. The system has a functioning bag memory structure (see [3,4]),

[1] Read the NAL book[4] for a fantastic in-depth tutorial into the logic and theory behind NARS.
[2] Of course, we should still always strive for the most efficient code!

handles real-time knowledge inputs (e.g. *judgments*, *events*, *goals*) and performs logical inference. Most importantly, v0.3 introduces the ability for the system to derive *new* goals and execute desired operations (i.e. actions), making it capable of autonomous interaction with its environment. While the v0.3 system does accept / reason with goals, it still lacks any useful level of autonomy since it is not good at constructing or refining useful procedural knowledge and has no sense of anticipation or timing. The system's temporal inference and timing mechanisms need improvement in future versions.

2 Architecture

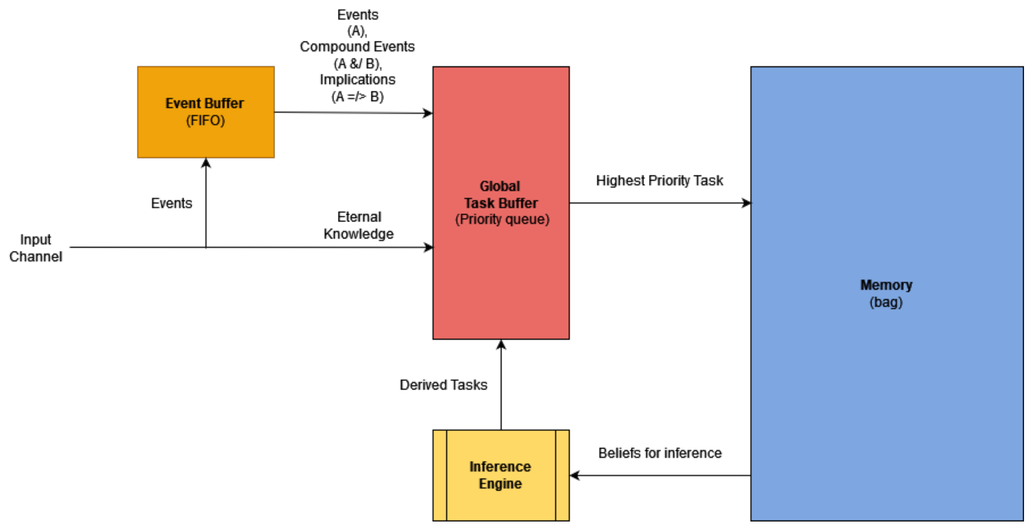

Fig. 1: The system architecture. The Control Mechanism works across all components.

Figure 1 shows the system architecture. The system operates every component during each working cycle. While the system architecture is modularized into separate components for clarity and organization, the system's Control Mechanism works *across* all the modules in a unified fashion. That is, data transfer between components is not strictly enforced or timed as in the case of an integrated system, since by necessity the components must interact often and intimately. It is the job of the central Control Mechanism to harmoniously use the architecture components together.

2.1 Input Channel and System Buffers

There is a single FIFO **Input Channel**, which is currently the only way to give external inputs to NARS; these inputs are strings and must represent valid Narsese syntax. Input strings are parsed into sentences and then processed into *Tasks*, which are containers for sentences that exist outside the main memory. Events are fed into the **Event Buffer**, a queue which maintains the temporal order of events so that they may be used in temporal inference. The contents of the Event Buffer, along with their combinations by temporal inference are dumped into the **Global Buffer** whenever the buffer contains two events. Eternal knowledge bypasses the Event Buffer and is instead sent directly to the Global Buffer. The Global Buffer is a standard priority queue, sorted by highest priority and where the highest priority object is selected when retrieving from the queue.

Experimental: Visual inputs The Input Channel can accept visual images as raw pixel inputs, by first typing `vision:` followed by a 2D array of pixel intensity, or a 3D array of pixel RGB. These are converted by the Input Channel into Narsese *sensations*, which are recursively implemented Narsese *arrays* (using $@$ to denote array). Arrays can be standalone terms (e.g. $@S$, which is a term that contains an array of terms) or array sentences (e.g. $<\{@S\} \rightarrow P>$., a *judgment* which *contains* array terms and is also an array itself). Narsese arrays are still highly theoretical and experimental; they are subject to change, but may be useful for native NARS perception in the future.

2.2 Memory

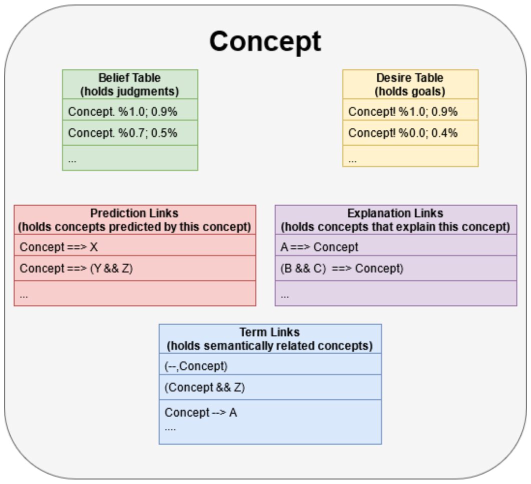

Fig. 2: Example showing the structure of a concept.

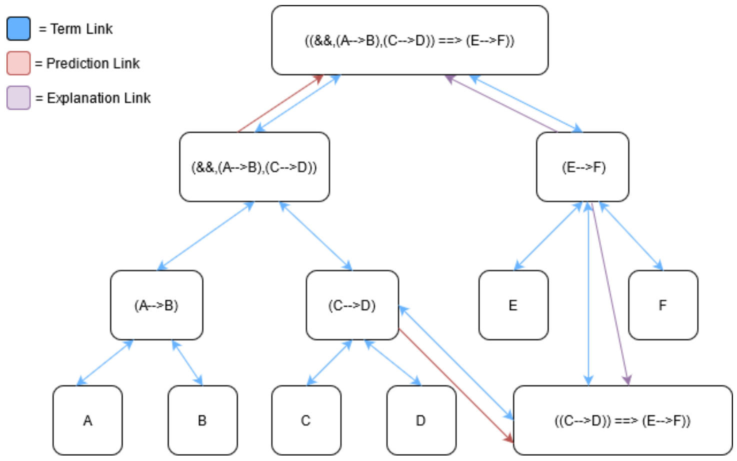

Fig. 3: A graph depicting how concepts are linked within NARS memory.

NARS **Memory** is where the system stores its experience and premises to use during inference. It consists of a standard *bag* data structure. The bag is a probabilitistic priority queue, which means its objects are probabilistically selected weighted according to each object's priority, relative to the other objects' priorities. The Memory should have a large capacity, since it stores all of the system's concepts (and by extension, knowledge) during the system's lifetime.

Each **concept** (see *Figure 2*) within the Memory is named by a unique term. A concept contains useful beliefs and desires which stored respectively in **Belief** and **Desire Tables**. A *table* is a minmaxheap sorted by highest sentence *confidence*.

A concept also stores links to other concepts within the memory (see *Figures 2,3*). Concept links are useful for finding semantically similar concepts, such as to use them in reasoning. In the current version, every concept has 3 bags of links: **term links** (to subterms and superterms), **prediction links** (to concepts predicted by this concept) and **explanation links** (to concepts which explain this concept). Term links are used by the system to find inference premises that share a term. Prediction and explanation links are created when processing a newly encountered implication statement; links to the overall implication statement are created within the concepts named by the precondition and postcondition terms. These links are not used in the current version, but will be useful for goal reasoning and anticipation in future versions.

2.3 Inference Engine

The **Inference Engine** performs inference on given premises, then outputs the results. The Engine can accept a single premise (for immediate inference), two semantically related premises (for syllogistic inference), or two temporally-near events (for temporal inference). Once the premises are provided into the engine, it identifies the relationship between them and performs the relevant inference rule computation; it also *stamps* each inference result with standard information (similarly as defined in [2]) like evidential base, creation time, occurrence time, sentence ID, etc.

3 Control Mechanism and Working Cycle

The system runs an infinite loop of working cycles. In a given working cycle, the system 1.) processes a pending sentence from the Input Channel, 2.) processes events from the Event Buffer into the Global Buffer, and 3.) either **Observes** a Task from the Global Buffer or **Considers** a concept from Memory; the selection ratio of Observe to Consider is probabilistic, thresholded according to a new system parameter called *mindfulness*.

A NARS system working in a real-time application must try to prevent its buffers from flooding by quickly processing its Tasks. When NARS **Observes**, it consumes the highest priority Task out of the Global Buffer. The contents of the Task are integrated into memory. The sentence is added to the relevant (belief or desire) Table, and new concepts / links are created as necessary. Finally, the concept associated with the Task has its priority raised since it is relevant to the current context.

The NARS cannot only consume new information from the outside world, but must also reason about, or consider, what it currently knows. When NARS **Considers**, it first *probabilistically* selects a "primary" concept from Memory, selects a belief or desire, then uses a concept link to find a related belief. Then, the system performs inference between the two selected sentences, putting the results into the Global Buffer. Finally, the initially selected concept has its priority decayed since it has been partially processed and received its fair share of the system's time and resources.

References

1. Hammer, P., Lofthouse, T.: 'opennars for applications': Architecture and control. In: Goertzel, B., Panov, A.I., Potapov, A., Yampolskiy, R. (eds.) Proceedings of the Thirteenth Conference on Artificial General Intelligence. pp. 193–204. Springer (2020)
2. Hammer, P., Lofthouse, T., Wang, P.: The opennars implementation of the non-axiomatic reasoning system. In: Steunebrink, B., Wang, P., Goertzel, B. (eds.) Proceedings of the Ninth Conference on Artificial General Intelligence. pp. 160–170. Springer (2016)
3. Wang, P.: Non-Axiomatic Reasoning System: Exploring the Essence of Intelligence. Ph.D. thesis, Indiana University (1995)
4. Wang, P.: Non-Axiomatic Logic: A Model of Intelligent Reasoning. World Scientific, Singapore (2013)
5. Wang, P., Hammer, P., Isaev, P., Li, X.: The conceptual design of opennars 3.1.0 (2020)