

SkRobot with TeleoR/QuLog: A Pseudo-Realtime Robotics Data Distribution Service Extended with Production Rules and Reasoning

Giovanni De Gasperis^a, Daniele Di Ottavio^b and Sante Dino Facchini^c

Dipartimento di Ingegneria e Scienze dell'Informazione e Matematica, Università degli Studi dell'Aquila, Italy

Keywords: Agents, Cognitive Robotics, Distributed Systems, Middleware, Real-Time Systems, Robot Development, Framework, Robot Production Rules, Reasoning.


Abstract: Designing and developing robots, particularly those with cognitive capabilities, is a complex task. The design platform and middleware Data Distribution Service we present in this paper, SkRobot, is meant to simplify this process. Built on the C++ SpecialK framework, it offers several functions to model robot behaviour, like active data brokering, distributed storage and processing, and pseudo-realtime synchronisation. SkRobot brings efficient communication between system entities using FlowProtocol, a custom protocol that guarantees robust typed binary data transfer over network channels. In this work the SkRobot architecture is extended and integrated with QuLog/TeleoR. QuLog (Query Language for Ontologies) and TeleoR (Teleological Reasoning) are two related technologies that enable robots to reason about their goals, actions, and the environment. QuLog is a query language that allows robots to ask questions about their knowledge base, while TeleoR is a Prolog logic reasoning system that enables robots to plan and execute actions to achieve their goals. To prove the successful integration between SkRobot and QuLog/TeleoR we implemented a virtual robotics simulation involving a NAO humanoid robot performing a target retrieval task.


1 INTRODUCTION


Taking as starting reference the theoretical model of agents presented in the book *"Artificial Intelligence: Foundations of Computational Agents"* (Poole and Mackworth, 2010), we postulate an programming ecosystem to design and develop computational agents relevant to cognitive robotics in the context of the embodied artificial intelligence (Chrisley, 2003). Each agent is defined as an entity that pursues goals by interacting with and modifying its environment based on sensory inputs and feedback. Agents are classified by their environmental impacts and goal attainment methods: with either reactivity or proactivity. Reactive agents respond to stimuli with pre-determined actions and are prone to errors in complex scenarios. Proactive agents increase their autonomy to analyse information and make decisions based on context, a knowledge base and past experiences (Costantini et al., 2017). Cognitive agents must be structured hierarchically with at least three main

layers: (i) the decision-making one to evaluate perceptions and managing goals; (ii) the central one to control, produce perceptions and proprioceptions, and get feedback; (iii) the peripheral one to interface with hardware, including sensors and actuators.

The peripheral layer acquires environmental stimuli from sensors, process them to feed the control layer with perceptions, while actuators combine commands coming from upper layers decisions with information and feedback coming from monitoring devices, so to adjust and execute actions (Poole and Mackworth, 2010; Mitchell et al., 1991; Raven et al., 1991). In particular, internal sensors monitor action progress and provide feedback that modulates actuator activities, which is essential for rapid response adjustments. Perceptions are instead multi-stimulus aggregations requiring efficient data management for real-time responsiveness (Poole and Mackworth, 2010; Moulin-Frier et al., 2017). When simulating robotic systems, implementing attention mechanisms to filter out irrelevant stimuli is crucial. Also, integrating real-time, parallel, and asynchronous communication is important as in distributed programming, were problems pose significant challenges and offer innovative solutions through advanced middle-

^a  <https://orcid.org/0000-0001-9521-4711>

^b  <https://orcid.org/0009-0008-2531-2170>

^c  <https://orcid.org/0000-0002-2009-5209>

ware designs (Costantini et al., 2021; Dyoub and De Gasperis, 2017).

In this scenario, we introduce our extension of the **SkRobot** server application (Di Ottavio, 2024b) to foster effective data transfer between agents and introduce rigorous production rules and reasoning.

2 RELATED WORKS

Designing a cognitive robot requires a multidisciplinary approach that combines many areas of expertise: systems and network engineering, physical-environmental sciences, control systems and electronics. Actual important stems of research are: (i) robot's operability and relative hardware resource management, (ii) software control both at high and low level, (iii) integration of multiple complex programs that interface with environmental inputs such as audio or video, and (iv) objects or sounds recognition.

2.1 Middleware Solutions

Researching of pre-built middleware and specialised applications that simplify the development in aforesaid stems is a valuable activity. As standards default in this middleware solutions, we consider two most valuable tools: **Redis**¹ that is commonly used for real-time data brokering in distributed systems, while in the SDKs sector **ROS** (Robot Operating System)² offers essential tools and libraries, facilitating robotic functionalities without starting from scratch (Quigley et al., 2009). Robot developers often implement many features using ROS as middle-ware, abstracting hardware to manage processes and communication. Its modular architecture allows focused development on navigation, perception, and control. The active ROS community contributes to a repository of software packages, solving common robotics challenges and promoting innovation and efficiency in robotic design. Despite this, ROS proves to be very complicated in the installation and setup phases related to the development environment and in its integration into the operating system. Additionally, creating and maintaining development projects based on ROS involves numerous procedures and formalities, often in the form of artefacts that are unnecessary for the development itself. The result is a very steep learning curve, specially for robotics students, which often drain resources from the theoretical concepts underlying robot agent theory. Moreover, the only

¹<https://redis.io>, last accessed July 2024

²<https://www.ros.org>, last accessed July 2024

ROS supported Unix-based operating systems are the most recent versions of Ubuntu, Fedora, and Debian GNU/Linux distributions. This excludes the possibility of creating minimal embedded systems which may not even require an OS, such as those based on micro-controllers. SkRobot require few, at least (i.e. without computer-vision, audio support and others), Unix base system dependencies, as: *libc*, *libstd++*, *libssl*, *libpthread*, *libm*, *libz*, other than an essential C++ compilation environment, based on *gcc* (this is preferable due to its open-source license, other than its permissive checks) or also *clang*, with their tool-chains, followed by *make*. These system parts are installed by default o many Unix-base system, anyway, if not, their installation are commonly very simply on each of these operating systems. If the need is to compile some code based on **SpecialK** (Sk) (Di Ottavio, 2024c) that requires extended supports dependencies, these could be more than many (i.e., the computer-vision support requires the development package for OpenCV-4.x, with its several dependencies).

2.2 SkRobot Server Application

The SkRobot FlowNetwork service, proposed in this paper for TeleoR integration, relays upon SpecialK, a previous work of authors those involved extensive experimentation with various application frameworks, notably the **Qt** SDK (Dalheimer, 2002). In particular, inspired by it, we adapted Qt's **Signal/Slot** paradigm to SpecialK, improving the traditional callback mechanisms used in ROS and creating complex yet manageable connection graphs among class functionalities. However, Qt was eventually deemed unsuitable due to its high commercial costs and event management system, which does not prioritise time — a critical factor for distributed systems with pseudo-realtime synchronisation used in robotics. SpecialK also integrated programming modalities found in other platforms like C++ sketches from the **Arduino** platform for firmware-oriented micro-controller programming (Monk and McCabe, 2016). An integration layer of SpecialK's is represented by the **Sk/PySketch** engine (Di Ottavio, 2024a)^{3,4}. It is a Python binding of the **FlowProtocol**, compatible with Python versions 2.7.x/3.6 (and upper), that permits an high-level and simple development way to create so called *satellites* for **FlowNetwork** apparatuses⁵ (Fig. 1a).

³<https://gitlab.com/Tetsuo-tek/SkRobot/-/blob/main/examples/publisher.py>, last accessed June 2024

⁴<https://gitlab.com/Tetsuo-tek/SkRobot/-/blob/main/examples/subscriber.py>, last accessed June 2024

⁵<https://gitlab.com/Tetsuo-tek/SpecialK/-/tree/master/LibSkCore/Core/System/Network/FlowNetwork>, last

2.3 Rule-Based Languages

On the reasoning and production rules side, here we are going to consider the integration of the **TeleoR/QuLog** (T/Q) reasoning system (Clark and Robinson, 2017) on the FlowNetwork, with the aim of creating a decision-making layer for the robotic agent.

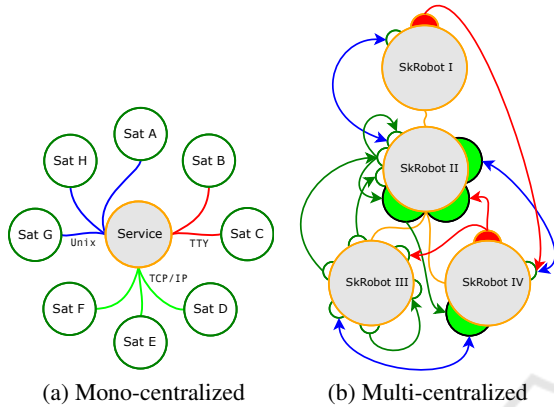


Figure 1: (a) Satellites in a Mono-centralized FlowNetwork. They are applications that manage sensors and actuators exchanging data with SkRobots. (b) Multi-centralized FlowNetwork/DipoleNetwork where multiple hubs have their own satellites, arrows are P2P connections.

QuLog/TeleoR (Clark and Robinson, 2017) is a major extension of Nilsson’s **Teleo-Reactive** (TR) rule-based robotic agent programming language (Nilsson, 1993). It consists of sequences of guarded action rules grouped into parameterised procedures. The guards are deductive queries to a set of dynamic facts in the agent’s Belief Store, and the actions can be primitive actions for external robotic resources or calls to TeleoR procedures. TeleoR enhances TR by being typed, higher-order, and offering more rule forms for finer control over task behaviour. Its belief store inference language, QuLog a Prolog extension, is a higher-order logic and function rule language that also supports action rules for agent behaviour threads. TeleoR introduces task atomic procedures for high-level multitasking with multiple robotic resources, ensuring non-interference, deadlock-free, and starvation-free task execution through compiler-generated coordination code. The programmer is abstracted from the coordination details handled by the TeleoR compiler.

Both, TeleoR and TR, are mid-level robotic agent programming languages that rely on lower-level routines in languages like C for sensor interpretation and complex robotic actions. Sensor interpretation results are stored as percept facts in the agent’s belief store, such as detecting a block on a table. Actions, like

accessed June 2024

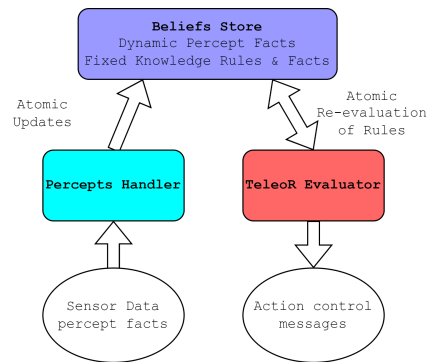


Figure 2: Simple Two Thread TeleoR Agent Architecture.

placing a block, are initiated based on these beliefs, though external events can interfere, delaying or altering outcomes.

TR and TeleoR determine when to invoke these actions to achieve sub-goals linked to a larger task, assuming percept beliefs accurately reflect the environment. TeleoR agents use a two-thread architecture (Fig. 2): one thread updates the belief store with new percepts atomically, while the other determines and executes action responses based on these updates.

TeleoR syntax is similar to TR, with formal operational semantics detailed in specific literature. TeleoR introduces features for both single-task and multi-task agents, allowing them to share and manage robotic resources efficiently. Future extensions of TeleoR and its agent architecture are planned.

3 METHODOLOGY

The goal of our work is to propose an extensions to our SkRobot application in order to add reasoning capabilities offered by TeleoR/QuLog. The idea underlying this proposal is to increment SkRobot possibilities acting as a collector (it offer bigger flexibility and scalability), enhancing and possibly substituting the **Pedro** as message broker and ultimately integrating TeleoR/QuLog with FlowNetwork (Fig 3). The SkRobot defines a structure that enables the entity to receive **stimuli** from its surroundings and perform valuable and logical **actions** by modifying the state of the environment itself, without allowing processes of acquisition and activity to interfere with each other. The main features of the communication layer of SkRobots that connects all the entities are (i) parallelism, (ii) functional asynchrony, (iii) reactivity efficiency, (iv) real-time (as much as possible), (v) data and events distribution.

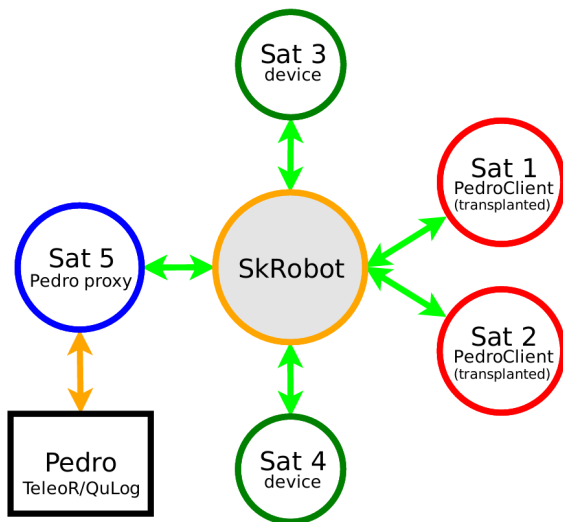


Figure 3: TeleoR/QuLog/QuProlog integration on FlowNetwork.

3.1 Software Engineering Aspects

The SkRobot server application, based on the SpecialK framework, has design philosophy that enhance low-level communication use for managing asynchronous input/output. It provides a streamlined method for developing robotic systems, addressing various implementation needs in robotics and related fields. SkRobot helps developers quickly understand and apply different study cases and solutions, simplifying the design process. Key concepts in SkRobot and SpecialK include active databrokering, distributed storage, distributed processing, and pseudo-real-time synchronisation. SpecialK, a C++ framework compatible with the **Standard Template Libraries** (STL), enforces paradigms such as asynchronous and recursive destruction of objects, Signal/Slot interactions, and event/pulse management. The design of robust and lightweight time management applications in robotics avoids using mutexes or semaphores, instead leveraging asynchronous collaboration as per Sk's paradigms for efficient parallel programming. Similar to the Qt framework, Sk uses the Signal/Slot paradigm for single process programming-flow concurrency. Sk minimizes dependencies on external libraries, typically requiring only open-source components like OpenCV⁶, PortAudio⁷, FFTW⁸, OggVorbis⁹, and FLTK¹⁰ for GUI support. Features can be toggled via compilation macros, allowing/disallowing direct inclusion of

⁶<https://opencv.org>, last accessed June 2024

⁷<https://portaudio.com>, last accessed June 2024

⁸<https://fftw.org>, last accessed June 2024

⁹<https://xiph.org>, last accessed June 2024

¹⁰<https://fltk.org>, last accessed June 2024

framework artefacts in application code, thus enhancing control over changes. At now, Sk and its tools are released as **rolling** framework environment.

The foundational class in SpecialK's hierarchy is SkFlatObject¹¹, providing basic functionality beyond instance naming. All Sk data structures derive from SkFlatObject, they designed for simple instantiation and automatic memory management when stack-allocated¹². The SkObject class¹³, a very important SkFlatObject derivative, introduces enhancements for efficient programming way. Instances of SkObject derivatives should be created with the new operator and destroyed asynchronously using `destroyLater()` to avoid runtime errors, ensuring proper integration with system interactions.

Objects are automatically destroyed in two cases: when a parent object is destroyed (forming a destruction tree) and when the event manager terminates its activity, so avoiding the existence of a dedicated *garbage collector* process. Otherwise, the `destroyLater()` method must be used to release resources, with resource allocation managed by the SkEventLoop manager¹⁴, which operates on a pulse or tick basis similar to an Arduino sketch's `loop()` function. This approach is fundamental to the SpecialK Python sketches (Sk/PySketch) and the FlowProtocol, where each thread operates under an SkEventLoop instance generating configurable ticks.

3.1.1 Pulsating Tick Timing

In the Sk framework, optional threads can have customised tick intervals and modes or inherit defaults from the main application thread, enabling synchronised or individualised operations. Each event loop manager emits three types of ticks at different speeds:

- **FastTick:** The fastest, non-divisible tick representing maximum processing speed.
- **SlowTick:** A passive tick with an interval equal to or greater than FastTick.
- **OneSecTick:** A passive tick occurring every second.

Passive waiting for SlowTick and OneSecTick is managed using SkElapsedTime, a nanosecond-

¹¹<https://gitlab.com/Tetsuo-tek/SpecialK/-/blob/master/LibSkFlat/Core/Object/skflatobject.h>, last accessed June 2024

¹²<https://gitlab.com/Tetsuo-tek/SpecialK/-/tree/master/LibSkFlat/Core/Containers>, last accessed June 2024

¹³<https://gitlab.com/Tetsuo-tek/SpecialK/-/blob/master/LibSkCore/Core/Object/skobject.h>, last accessed June 2024

¹⁴<https://gitlab.com/Tetsuo-tek/SpecialK/-/blob/master/LibSkCore/Core/App/skeventloop.h>, last accessed June 2024

resolution timer, while active waiting in FastTick mode involves suspending the thread using `usleep(...)` or `nanosec(...)`. SlowTick and OneSecTick are for less frequent tasks like monitoring and control, while FastTick handles immediate tasks such as network communication.

For multiple-thread applications, consumer threads must pulse as frequently as or more than producer threads to avoid issues like queue overflow. Monitoring job-time amplitude during tick processing ensures it doesn't exceed FastTick intervals, which could slow the tick rate and degrade performance.

The cadence for FastTick, SlowTick, and OneSecTick can follow various modes:

- **Regular Coarse Timing:** Uses `usleep(...)`, being lighter on resources.
- **Pseudo-Real-Time Timing:** Uses `nanosec(...)`, more CPU-intensive but precise.
- **Socket I/O Timing:** CPU-intensive if traffic is high, based on socket activity.
- **GUI Activity Timing:** lighter on CPU, based on FLTK event handler.
- **No Timing:** Uses an external blocking call to slow down the pulse activity, comparable to a `while(1){...}` construct.

The Signal/Slot paradigm, crucial in programming workflow, evolved from callback functions. Callback functions, used in languages like C and Python, also adopted in ROS, handle specific events, such as a button click triggering a predefined function. In the Signal/Slot paradigm, a Signal is a method declared without scope, while a Slot functions like a standard method but always returns void. Unlike callbacks, Signals can be connected to multiple Slots, even from different object unknown types, using the `Attach` functional macro^{15, 16, 17}.

Signals and Slots must be publicly declared within a class to be observable and manageable by the event manager. Private or protected declarations cause runtime errors due to visibility restrictions. The `'extends'` macro ensures public inheritance for managing interactions across derived types, facilitating synchronisation without mutexes or wait conditions, thus streamlining Inter-Object Communication (IOC) and enhancing responsiveness.

`Attach` and `Detach` operations are asynchronous, scheduled for the next pulse by the event man-

ager. A Signal triggered immediately after an `Attach` won't activate the connected Slot until the next tick. `Attach` operations typically occur in the object's `Constructor`, and `Detach` happens during the object's destruction, maintaining stable connections.

3.1.2 Signal/Slot Connection Modes

When a Signal is triggered, it acts like a method call. Connected Slots execute their code immediately or asynchronously in their respective threads, ensuring immediate Signal trigger and flexible Slot execution. Connection modes between Signal and Slot differ on application needs. Connection mode can be: **(i) Direct:** Slots are invoked directly when the Signal is triggered. This is similar to the Slot method being called to execute the code in the triggering thread where the code that requested the triggering Signal lives. **(ii) Queued:** Slots are queued for future invocation by the event manager at the next round and in the programming-flow of the owning thread, even when the Signal call comes from another manager, hence a different thread. Triggering a connected Signal with queuing mode never blocks the triggering call, even when the Signal and Slot reside in the same thread. **(iii) OneShot** (direct or queued): Slots are invoked as illustrated in the previous two points but only once; immediately after the invocation, the disconnection occurs automatically.

A Signal can connect to multiple Slots but not the same Slot more than once. When triggered, a Signal can pass `SkVariant` arguments to the Slots. Direct calls allow Slots to access pointers to the original values, while queued calls provide argument copies, avoiding critical sections and mutual exclusion issues for multi-threaded approach¹⁸. The `SkVariant` class encapsulates diverse data types, including primitives, complex structures, and pointers.

For thread synchronisation without mutexes or wait conditions, queued-type connections between Signals and Slots from different threads are recommended to prevent deadlocks and efficiency losses. This allows processing tasks to be isolated across different classes, maintaining interaction through dynamic runtime meta-links.

The `SkRobot` application, developed in C++, manages data Flow in autonomous devices, robotic systems, and industrial production lines. It supports I/O management and custom network services in Unix-like environments with minimal dependencies, adhering to the POSIX standard. `SpecialK` and its subordinated applications like `SkRobot`, simply com-

¹⁵<https://gitlab.com/Tetsuo-tek/SpecialK/-/blob/master/LibSkFlat/skdefines.h>, last accessed June 2024

¹⁶<https://gitlab.com/Tetsuo-tek/SpecialK/-/blob/master/LibSkCore/Core/Object/sksignal.h>, last accessed June 2024

¹⁷<https://gitlab.com/Tetsuo-tek/SpecialK/-/blob/master/LibSkCore/Core/Object/skslot.h>, last accessed June 2024

¹⁸<https://gitlab.com/Tetsuo-tek/SpecialK/-/blob/master/LibSkFlat/Core/Containers/skvariant.h>, last accessed June 2024

pile and run on all unix-based system; SkRobot was tested on: Void-Linux, Gentoo, Ubuntu, Elementary-OS, Raspberry-OS, Armbian, GNU/Hurd, Haiku-OS, Minix, NetBSD, FreeBSD, and finally MacOS (through Homebrew) - we excluded Microsoft Windows from tests.

3.2 Flow Protocol

Sk defines the FlowProtocol for inter-process communication between entities (modules and satellites with their hubs, (see Fig.1a and Fig.1b) on various network supports, such as serial TTY lines, Unix-domain sockets, TCP, UDP, and WebSockets.

Communication uses a binary format with structured frames, distinguishing between synchronous (blocking) and asynchronous (non-blocking) commands. Errors in communication result in connection termination. The `SkFlowServer` class manages these communications, supporting both synchronous and asynchronous connections. All connections start as synchronous and can become asynchronous after authentication, facilitating distributed computational tasks across different processes and threads. Database service management are handled through the `SkFlowPairDatabase` class.

In SkRobot, database operations depend on the current database label setup, executed whenever the database target changes. Variables are stored as labeled `SkVariant` instances, which can handle various data types (primitives and more complex structures) with the possibility to convert them to/from JSON structures. Protocol commands using variables have JSON-text counterparts for platforms where `SkVariant` does not exist yet, like Python. Asynchronous commands do not receive responses but can trigger messages. For synchronous data requests, a temporary connection is established and closed after retrieval. Data distribution occurs through Flow channels within asynchronous connections, preventing interference from different data types. SkRobot can manage up to 32768 flow channels, each uniquely identified by ID, name and hash. Channels can either (i) distribute streaming data to multiple consumers (1:N) or (ii) provide request/response services (1:1); service channels dialogues can be of synchronous (blocking response) or asynchronous type (FlowNetwork non-blocking event). Disconnection of satellites removes their channels along with their established relationships.

SkRobot's modular architecture uses FlowNetwork for communication, with internal modules enhancing system synchronisation. These modules derive from the `SkAbstractModule` interface, managing parameters via JSON and requiring redefined

virtual methods. External satellites, managed by the class `SkFlowSat`, automate connection, Flow set management, and event subscription, reducing repetitive coding and ensuring compatibility within the network. Examples of C++ and Python code can be acquired from SkRobot repository ¹⁹.

4 TeleoR INTEGRATION

Regarding the integration of TeleoR/QuLog into the FlowNetwork, various potential solutions have been explored. The most immediate and promising for future developments appears to be the one where a satellite implemented in Python, based on the PySketch engine, incorporates both the client component for the FlowNetwork (represented by the `FlowSat` class) and the client component for the Pedro server (represented by the Python `PedroClient` class).

This approach enables a seamless integration between the two communication protocols, with the involved satellite functioning primarily as a proxy. The satellite, which adheres to the integration protocol, also opens one or more input channels to accept perceptions generated by other satellites within the same FlowNetwork. These perceptions are subsequently forwarded to Pedro. TeleoR, operating within the Pedro framework, processes the incoming perceptions and produces a decision response, which can be transmitted back as feedback to the perceptions acquired.

The perceptions themselves can be captured through various mechanisms, such as a microservice operating in a request/response format (either synchronous or asynchronous, and either textual or binary), or via binary or textual streaming channels that can be subscribed to by the proxy. These streams originate from distribution satellites within the same FlowNetwork. By utilizing these diverse input methods, the satellite effectively serves as a transparent intermediary, ensuring communication between the distributed entities in the system while delegating decision-making processes to the TeleoR reasoning engine in Pedro.

5 NAO ROBOT CONTROL

As a proof-of-concept implementation of the methods presented in previous Section, we integrate a decision maker developed with TeleoR/QuLog with a `NaoSat` robot performing a target retrieval task. The

¹⁹<https://gitlab.com/Tetsuo-tek/SkRobot/-/tree/main/examples>, last accessed June 2024

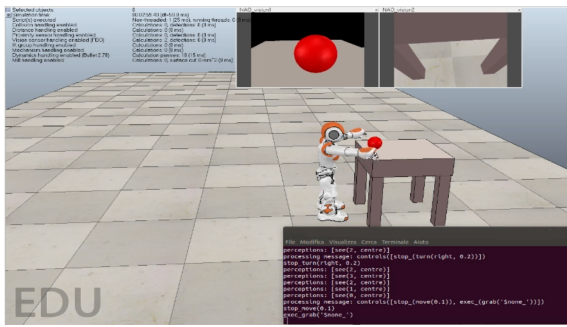


Figure 4: Simulation of the NAO humanoid robot after running the QuLog/TeleoR control program, getting close to its target.

NAO robot (Gouaillier et al., 2009) has been simulated with the Coppelia robot simulator²⁰, in a simple scene with a table where on top lays a red ball. The NAO sensors are the two torso sonars, left and right, and the frontal camera. The NaoSat from the sensor readings creates the perceptions

```
def dir ::= left | centre | right
def object ::= ball | duck
```

```
percept holding(), see(num, dir)
```

to determine where the red ball is seen in the visual range of the camera. It then subscribe to the PedroSat/TeleoR notify channel to receive motion commands. The QuLog/TeleoR agent program receives perceptions as QuLog-subscribe string, through a service-channel, and plans accordingly the motion command to engage, with the following main TeleoR program:

```
tel collect_object()
collect_object(){
  taken()
  ~> ()
  too_close()
  ~> get_next_to()
  next_to(centre)
  ~> grab()
  next_to(Dir)
  ~> turn(Dir,0.2)
  holding()
  ~> release()
  true
  ~> get_next_to()
}
```

Fig.4 shows the simulation with the NAO humanoid robot that almost accomplished the task of finding and grabbing the target red ball, after the exploration of the environment.

In doing this we replace the standard Pedro message broker with the PedroSat (an external satellite

²⁰<https://www.coppeliarobotics.com>, last accessed July 2024

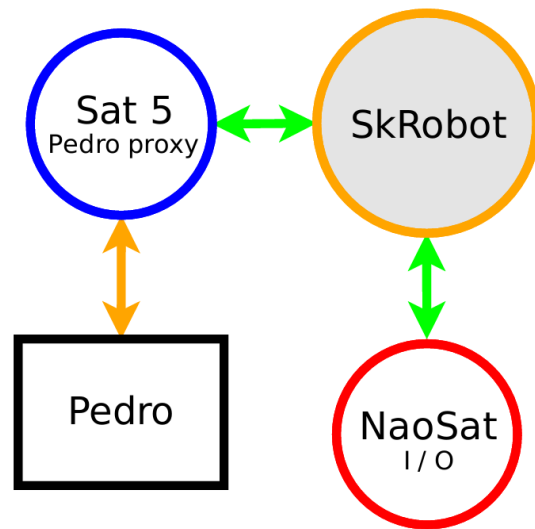


Figure 5: The integration of the TeleoR/QuLog/QuProlog satellite in the NAO robot controller.

Python sketch) connecting to a SkRobot service hub (see Fig.3 and 5), and also accepting old style Pedro clients, where can connect TeleoR (living its code untouched during this first integrating step).

Another external satellite sketch is represented by the agent (named NaoSat in the Fig.5) that is able to collect all physical inputs and outputs. This satellites inherit a PedroClient (transplanted) able to work directly on the FlowNetwork, bypass the canonical Pedro connection. Through SkRobot the NaoSat uses TeleoR to make light and logic reasoning, simply on CPU with a very low power-energy needs.

6 CONCLUSIONS

In this work we extended the SkRobot development environment and data distribution service to integrate logical reasoning and production rules that guide a robust and dynamic robot planner in real-time, while exploring an unknown and unstructured environment. We proposed a design and implementation methodology for augmenting SkRobot satellite applications with a computational logic module and discussed a proof-of-concept preliminary demonstrator simulating a cognitive humanoid robot guided by QuLog/TeleoR program.

6.1 Paper Contribution

The main contribution of our paper, consist in providing a framework that paves the road for the application in real-world scenarios of intelligent multi-robots systems acting with pseudo real-time timing constraints.

Such innovation could be a key enabler for swarm-robots industry where real-time features are very important for management and control. Furthermore SkRobots can supply a reliable base to develop distributed applications with reactive and proactive behaviours as well as reasoning system creating a real decision-making layer for robots.

6.2 Future Works

Future developments of SkRobots may include the addition of accountability and trackability properties to foster better Human-Swarm Interaction (HSI) integration in the fields of multi-robots. This could be implemented inserting Distributed Ledger Technologies (DLT) in satellites and communication channels that would allow decisions and commands to be immutable and inspectable.

ACKNOWLEDGEMENTS

This research was partially funded by **NextGenerationEU** under the Italian Ministry of University and Research (MUR) National Innovation Ecosystem grants with the following codes: ECS00000041 VITALITY CUP: E13C22001060006; ADVISOR - PRIN 2022 PNNR Prog. P202277RJ2-PE6 CUP: E53D23016270001; TRUSTPACTX - PRIN 2022 Prog. 20228FETWM CUP: E53D23007850001.

Authors used occasionally on line generative AI tools to improve the readability of the text; they reviewed and edited the content as needed, and took full responsibility for the content of the publication.

REFERENCES

- Chrisley, R. (2003). Embodied artificial intelligence. *Artificial intelligence*, 149(1):131–150.
- Clark, K. L. and Robinson, P. J. (2017). Concurrent task programming of robotic agents in teleor. In *RuleML+RR (Supplement)*.
- Costantini, S., De Gasperis, G., Lauretis, L., et al. (2021). An application of declarative languages in distributed architectures: ASP and DALI microservices. *International Journal of Interactive Multimedia and Artificial Intelligence*, 6(5):66–79.
- Costantini, S., De Gasperis, G., and Nazzicone, G. (2017). DALI for cognitive robotics: Principles and prototype implementation. In *Practical Aspects of Declarative Languages: 19th International Symposium, PADL 2017, Paris, France, January 16-17, 2017, Proceedings 19*, pages 152–162. Springer.
- Dalheimer, M. (2002). *Programming with QT: Writing portable GUI applications on Unix and Win32*. ” O’Reilly Media, Inc.”.
- Di Ottavio, D. (2024a). Pysketch engine emulating arduino-sketches, useful to build satellites, compatible with python 2 and 3.
- Di Ottavio, D. (2024b). Skrobot application server, an hub for flow-sat clients based on flow-protocol.
- Di Ottavio, D. (2024c). Specialk c++ framework based on signal/slot, pseudo-real-time pulsing and recursive object destruction.
- Dyoub, A. and De Gasperis, G. (2017). Rule-based supervisor and checker of deep learning perception modules in cognitive robotics. In *RuleML+ RR (Supplement)*.
- Gouaillier, D., Hugel, V., Blazevic, P., Kilner, C., Monceaux, J., Lafourcade, P., Marnier, B., Serre, J., and Maisonnier, B. (2009). Mechatronic design of NAO humanoid. In *2009 IEEE international conference on robotics and automation*, pages 769–774. IEEE.
- Mitchell, L. G., Mutchmor, J. A., and Dolphin, W. D. (1991). *Zoologia*. Zanichelli.
- Monk, S. and McCabe, M. (2016). *Programming Arduino: getting started with sketches*, volume 176. McGraw-Hill Education New York.
- Moulin-Frier, C., Fischer, T., Petit, M., Pointeau, G., Puigbo, J.-Y., Pattacini, U., Low, S. C., Camilleri, D., Nguyen, P., Hoffmann, M., et al. (2017). Dachs: A proactive robot cognitive architecture to acquire and express knowledge about the world and the self. *IEEE Transactions on Cognitive and Developmental Systems*, 10(4):1005–1022.
- Nilsson, N. (1993). Teleo-reactive programs for agent control. *Journal of artificial intelligence research*, 1:139–158.
- Poole, D. L. and Mackworth, A. K. (2010). *Artificial Intelligence: foundations of computational agents*. Cambridge University Press.
- Quigley, M., Conley, K., Gerkey, B., Faust, J., Foote, T., Leibs, J., Wheeler, R., and Ng, A. (2009). ROS: an open-source robot operating system. *ICRA Workshop on Open Source Software*, 3.
- Raven, P. H., Evert, R. F., and Eichhorn, S. E. (1991). *Biologia delle piante*. Zanichelli.