Web Based Monitoring, Orchestration and Simulation

Dave Raggett(✉)

ERCIM/W3C, Biot, France
dave.raggett@ercim.eu

Abstract. This paper describes a framework for web-based monitoring, orchestration and simulation for industrial settings such as highly automated factories and warehouses. HTML5 is used for a 2.5D visualization with local prediction for smooth animation and 3D models for robot arms. The web server aggregates data from the devices for streaming to the web page, for generating situation reports, and enabling users to intervene as needed. Simulation is possible using simple looping sequences of actions. A cognitive architecture is described for richer behaviour that dynamically adapts to the context, decoupling reasoning from real-time control using asynchronous intents. Stochastic rules allow agents to escape from futile patterns of behaviour. Agents can control multiple devices. For a larger numbers of devices, control can be distributed across multiple agents in a way that decouples applications from the underlying protocols and addressing schemes. Iterative refinement is applied as new requirements come to light, e.g. to handle faults, or where humans have intervened in unexpected ways. The paper closes with a short summary of previous work and suggestions for future work on integrating generative AI to further simplify development and provide supervisory control.

Keywords: automation · cognitive control · swarm computing · orchestration · simulation · multi-agent systems

1 Introduction

In the context of control systems, agents are computer systems that can operate autonomously, making decisions and taking actions without constant human oversight. Agent behaviour can be classified into broad categories, e.g., reactive, model-based, goal-based, and utility based, as well as on how they interact with other agents, e.g. collaborative or competitive. Reactive agents take actions purely on the basis of current information on their environment. Model-based agents build and exploit models of their environment, e.g. a robot vacuum cleaner that builds a map of the rooms it operates in. Goal-based agents combine models with search and planning. Utility-based agents have a function for scoring the desirability of different states or outcomes, and focus on maximising the utility, not just fulfilling a goal.

Building upon the capabilities of such agents, this paper describes approaches to web-based monitoring, orchestration and simulation for industrial systems. Specifically, web-based monitoring leverages web browsers to observe the real-time state of industrial environments like factories or warehouses. This monitoring functionality can then be extended to facilitate orchestration, providing a comprehensive means for coordinating, synchronizing, and overseeing complex, automated workflows across diverse systems and applications.

Orchestration can be applied to either control an industrial system or to drive a simulation of an industrial system. This paper describes a basic approach to describing repeated fixed behaviours that are typical of current industrial control systems, and the extension to more flexible context sensitive adaptive rule-based control, and the means to synchronise behaviour in a multi-agent system. Rule-based orchestration uses rules to define the order, conditions, and dependencies of these behaviours, ensuring they operate together smoothly.

The paper concludes with an assessment of opportunities for further work, e.g. on decentralised decision making, multi-agent reinforcement learning, and the combination of generative AI with rule-based approaches.

2 Web-Based Monitoring

Users can use the web browser to see a 2.5D isometric view of the factory or warehouse floor, with the means to pan and zoom the view using game controllers, keystrokes or gestures on the computer's touchpad. Isometric views show objects at the same size regardless of their location, just as is the case for 2D maps. Users can further interact with the presentation to request further information and to change system parameters.

Figure 1 shows a screenshot from a simulation of a smart warehouse [2] where robot forklifts move pallets carrying different kinds of products from the incoming trucks (yellow) to the outgoing trucks (red) to fulfil customer orders. Pallets are temporarily stored in the racks if appropriate. Forklifts are assigned jobs, and plan the shortest route following the traffic lanes (marked in red and green) near the racks, and direct routes away from the racks. The forklifts are sent to recharge when their battery level falls below a given threshold. Each forklift uses its first-person view to avoid other forklifts, and all forklifts stop to avoid colliding with nearby humans. The user can request information on a given forklift, see the pop-up pane for an example.

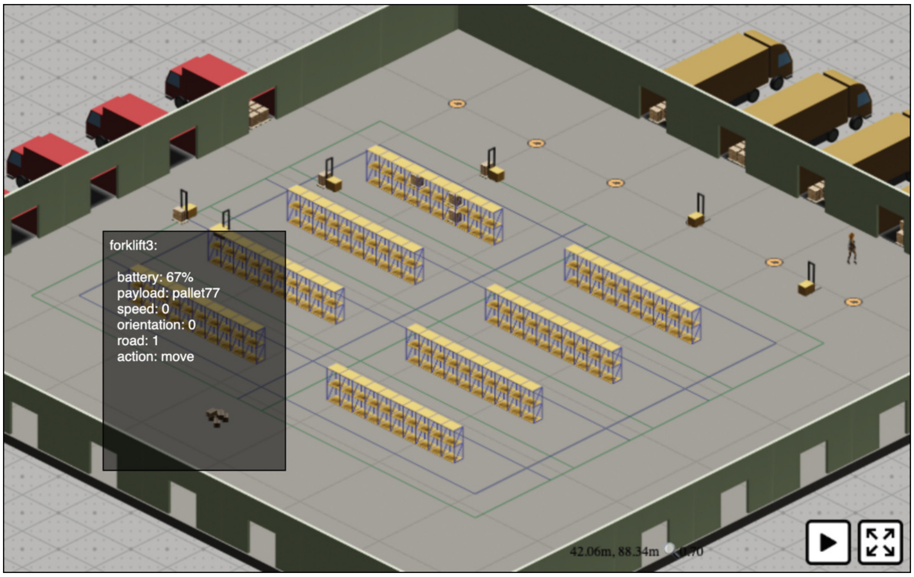
3 Digital Twins

In this paper, we define digital twin as a term for a computer model of physical systems and processes. Digital twins can be used for:

- monitoring current state of their physical twins
- controlling their physical twins
- diagnosing and fixing faults using causal models
- planning and simulation
- optimisation based upon applying machine learning to recorded data

This paper focuses on web-based monitoring, orchestration and simulation. The starting point is a web server that hosts the digital twin models as virtual objects with properties, actions and events, as inspired by the Nephele project [3] and the W3C Web of Things [4].

For monitoring, we need to dynamically update the state of the digital twins from the physical systems and processes, and to stream the updates to the web pages connected to the web server. Orchestration is the process for remotely managing industrial systems. It can apply to controlling the actual devices, or it can be used in a simulation to control virtual digital twins.


forklift3:

battery: 67%
payload: pallet77
speed: 0
orientation: 0
road: 1
action: move

Fig. 1. Screenshot from SimSwarm, a simulation of a smart warehouse operated by robot forklifts. See the demo at: https://www.w3.org/Data/demos/chunks/warehouse/.

The web server acts as a relay, communicating with the devices using whatever protocols are appropriate, e.g. HTTP, MQTT or DDS, to update the digital twins held by the server.

The author has developed an opensource NodeJS solution that combines a JavaScript module for HTTP + Web Sockets with a custom module that updates the digital twin models, and streams the updates as JSON messages to the web pages currently connected to the server. NodeJS is a strong community with plenty of support for a wide range of protocols [5].

If your devices feature ROS (the robot operating system), then you will find it convenient to use the *rosnodejs* module [6] for server-side communication with a networked ROS Master. Alternatively, you can write your own drivers, e.g. using one of several DDS modules for NodeJS with a mapping between IDL and JavaScript datatypes (Figs. 2 and 3).

```
let pallet1 = new Pallet({
    name:"pallet1",
    x:20,
    y:30,
    orientation:0,
    loaded:true
});
let forklift1 = new Forklift({
    name:"forklift1",
    x:10,
    y:30,
    z:0,
    orientation:0,
    speed:1,
    forkspeed:0.5,
});
```

Fig. 2. Example for declaring a pallet and a forklift.

```
# example message for adding a pallet with ID "pallet2"

{"name":"pallet2","type":"pallet","x":10,"y":40,
   "orientation":3,"loaded":false}

# example message for updating forklift state

{"name":"forklift1","type":"forklift","x":16.58,"y":30,"z":0,
   "orientation":0,"speed":1,"held":false}
```

Fig. 3. Examples of JSON messages sent by the server to update connected web pages.

4 Simple Sequences of Instructions

In today's factories it is commonplace to program robots and other factory machinery to indefinitely repeat the same sequence of instructions. The JavaScript module for digital twins provides a simple means to provide such instructions:

The *move* action prepends a *turn* action to turn to the appropriate *orientation*. Likewise, the *grab* action prepends *move*, *turn* and *forks* actions as needed, having selected which side of the pallet is the nearest to grab the pallet. To raise or lower the forks explicitly use the *forks* action, e.g. {act:"forks", z:0.5} which raises or lowers the forks to 0.5m above the floor. To turn to a given orientation, you can use the *turn* action, e.g. {act:"turn", orientation:4}, noting that orientation is an integer in the range 0 to 7 reflecting the eight camera orientations for the 2.5D image tiles used in rendering (Fig. 4).

```
forklift1.setTask([
    {act:"wait", time:4},
    {act:"grab", pallet:"pallet1"},
    {act:"move", x:10, y:30},
    {act:"release"},
    {act:"move", x:20, y:30},
    {act:"grab", pallet:"pallet1"},
    {act:"move", x:20, y:30},
    {act:"release"},
    {act:"move", x:10, y:30},
    {act:"next"}
]);
```

Fig. 4. Example declaring a sequence of control instructions for a forklift.

You can instruct the forklift to wait for a fixed number of seconds, e.g. {act:"wait", time:4} waits for 4 s. You can make it a little more interesting by instructing a wait that is randomly selected between a minimum and maximum duration, e.g. {act:"wait", min:1, max:5} waits between 1 and 5 s.

The *next* action defaults to transferring control to the first action in the task. However, if you provide a list of named steps, one of those steps is randomly selected.

Note that the *move* action doesn't itself include support for avoiding obstacles, whether stationary or moving. This is something to consider in future work on integrating Chunks & Rules for more flexible behaviour, and as part of a plan for scaling up to a much larger range of factory devices, e.g. robot arms, conveyor belts, palletization and manufacturing cells. The SimSwarm demo uses collision avoidance heuristics implemented directly in JavaScript. This could be replaced by an explicit system of rules.

5 Cognitive Control for Context Sensitivity

Simple repeating sequences of instructions are fine most of the time, but can run into difficulties in the case of faults or when a human has intervened in some manner that upsets the assumptions in the programming. An example is where a robot tries grab something which isn't in quite the correct position. Another case is where a human worker has removed something that the robot expects to find.

5.1 Chunks and Rules

This is a syntax and open-source implementation for facts and rules inspired by John Anderson's cognitive architecture ACT-R, which is grounded upon decades of research in the Cognitive Sciences [7]. It is at a higher level than RDF and uses a simple easy to use syntax.

The architecture features one or more cognitive modules, each of which is associated with a buffer that can contain just one chunk. A chunk is a set of properties including a type and a unique chunk identifier. Property values are names, numbers or a list thereof. In respect to RDF, a chunk corresponds to a set of triples with the same subject node, and to RDF lists for list values. A convenient short hand is provided for unannotated relations (corresponding to RDF triples), which are internally reified into single chunks.

For more details, see the specification from the W3C Cognitive AI Community Group [8].

The architecture is motivated by the structure of the brain. The cognitive modules correspond to cortical regions. The rule engine corresponds to the basal ganglia in the centre of the brain. Cognitive buffers correspond to bundles of nerve fibres that connect the basal ganglia to the cortex. Each chunk thus corresponds to a vector in a high dimensional space (Fig. 5).

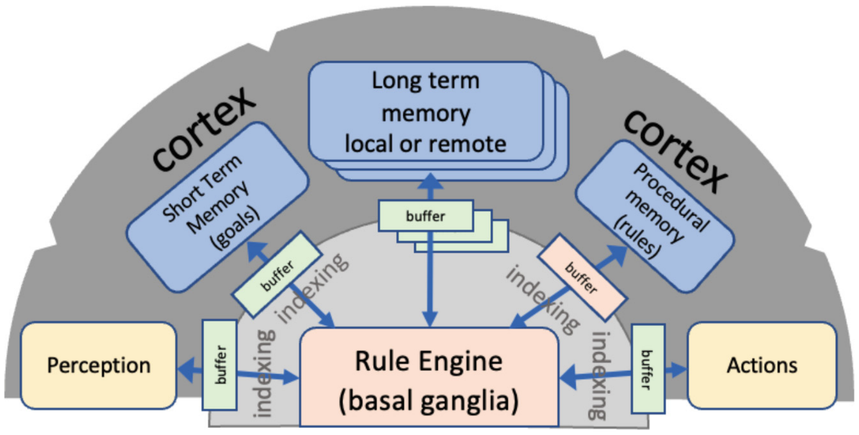

cortex
cortex
Long term memory local or remote
Short Term Memory (goals)
Procedural memory (rules)
buffer
buffer
buffer
buffer
buffer
indexing
indexing
indexing
indexing
Perception
Rule Engine (basal ganglia)
Actions

Fig. 5. Cognitive architecture for Chunks & Rules.

Rule antecedents are a conjunction of conditions expressed as chunks. Rule consequents are an unordered list of actions expressed as chunks. Rules are matched to the current state of the module buffers. If multiple rules are matched, a stochastic choice is made influenced by rule strength. Rule execution is sequential.

The approach borrows from ACT-R in respect to sub-symbolic parameters corresponding to the strength of facts and rules, mimicking the forgetting curve for human memory, likewise for spreading activation and the spacing effect. Performance is fast as a) the rules are matched to a handful of buffers rather than the entire set of chunk databases, and b) actions are asynchronous, enabling cognition to continue whilst actions are being executed. There is a suite of built-in actions for operations on chunk databases and an API for programming complex operations. Applications can register additional actions as needed, e.g. to operate a robot arm.

Chunks & Rules are well suited to orchestrating devices via their digital twins using intent-based actions for concurrent threads of behaviour. Intents specify the desired outcome rather than the details of how to realise that outcome, something best suited for delegation to specialist systems. Work is underway on extending Chunks & Rules to support messaging and synchronization across agents. This is described in a later section of this paper (Fig. 6).

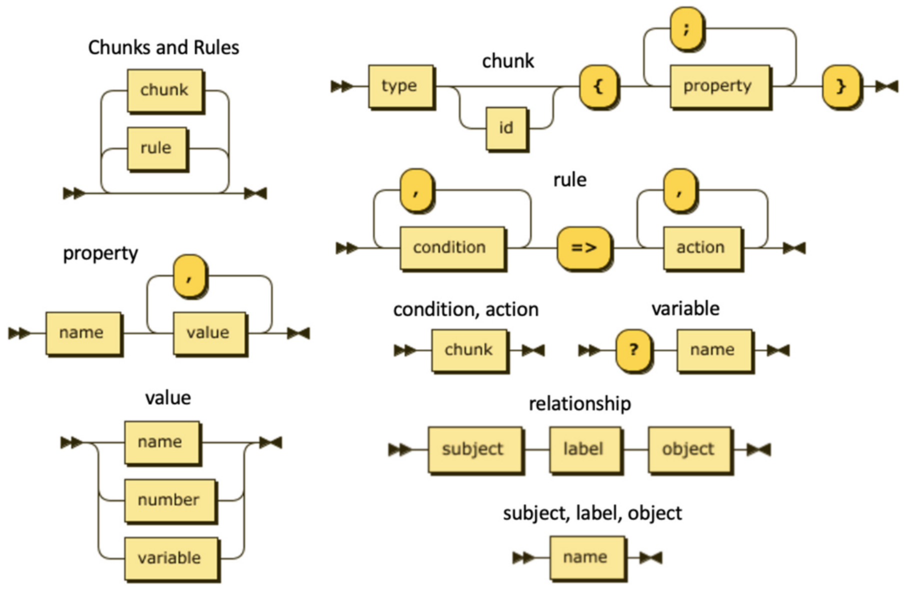


Fig. 6. Chunks & Rules syntax as railroad diagrams.

Note that you are free to use whitespace between tokens except between the "?" and name in a variable. Variables are used in rule conditions and actions (Fig. 7).

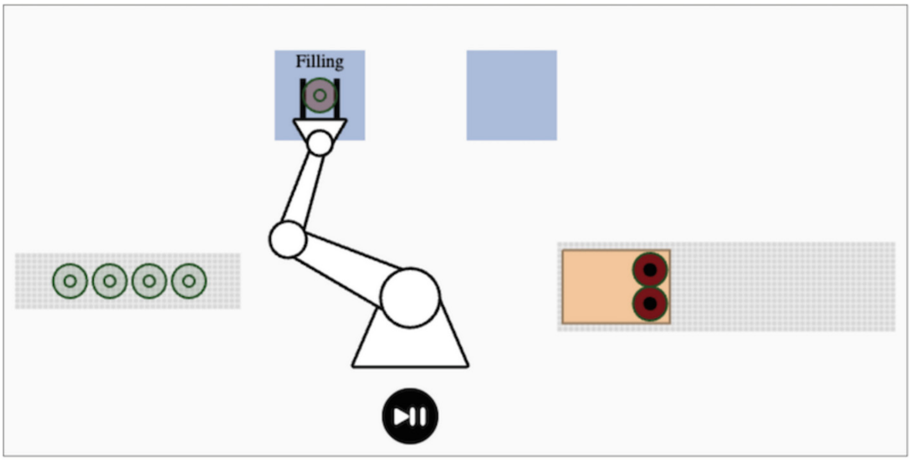


Fig. 7. Example web demo for bottling plant.

Here are some example rules from a web-based demo for a bottling plant [9] where a cognitive agent controls two conveyor belts, one robot arm as well as the filling and capping machines (Fig. 8).

```
# move robot arm into position to grasp empty bottle
after {step 1} =>
    robot {@do move; x -170; y -75; angle -180; gap 30; step 2}
# grasp bottle and move it to the filling station
after {step 2} =>
    goal {@do clear},
    robot {@do grasp},
    robot {@do move; x -80; y -240; angle -90; gap 30; step 3}
```

Fig. 8. Example rules for robot control.

Here the *move* and *grasp* commands are application defined and act as intent-based directives to the robot. The move command describes the position and orientation of the robot gripper along with the gap between its opposed fingers. This is expanded into a plan for smoothly accelerating and decelerating the various actuators in the robot arm. When the plan completes, a chunk "*after {step 3}*" is queued to the module buffer to trigger following behaviour. This demo uses some twenty rules in total. Note that rules can include variables which are instantiated when the rule is matched to the buffers. This is used in the demo when packing bottles six to a box.

The set of chunks and rules can be iteratively refined as new requirements come to light, e.g. in unusual situations like faults, or where humans have intervened in an unexpected way. This may involve the need for a systems programmer e.g. to update the machine vision system used to monitor physical processes, e.g. has a bottle fallen over on the conveyor belt? Has the bottle been correctly filled and capped?

In some circumstances, robots may find themselves trapped in futile repeating patterns of behaviour. To mitigate this, you can introduce stochastic non-deterministic behaviour by defining multiple rules with the same conditions. The agent will then randomly select which rule to execute, thereby introducing variability.

5.2 Extension to Coordinating Multiple Agents

Whilst a single cognitive agent can control multiple devices, for large numbers of devices, it makes sense to partition the work across multiple collaborating agents. This raises the question for how to enable agents to communicate with each other. This section extends Chunks & Rules to support direct messaging to named agents or topic-based distribution where agents subscribe to the topics they are interested in. This decouples the cognitive models from the underlying communication technologies used for message transfer, which can be selected according to the requirements, e.g. Zenoh, MQTT or DDS and handled via existing open source libraries.

Agents send arbitrary chunks using *@message* to identify the recipient, e.g. as in the following rule action property. You are free to set the chunk type and provide additional properties as appropriate to your needs. The addressee sees the chunk with the @message property stripped out. You can determine which agent sent the message by using *@src* with a variable in the conditions for a rule that matches the message (Fig. 9).

```
# tell agent4 to start
start {@message agent4}
```

Fig. 9. Example for direct messaging between agents.

Messages can be sent with @*topic* to subscribers of named topics, e.g. (Fig. 10)

```
#send stop message on topic12
stop {@topic topic12}
```

Fig. 10. Example for topic-based messaging between agents.

Agents can subscribe to topics with @*subscribe* as in the following example (Fig. 11):

```
# subscribe to topic12
listen {@subscribe topic12}
```

Fig. 11. Example for subscribing to a topic.

5.3 Agent Synchronisation Using Tasks

We can model a task as a sequence of rules that achieve some given aim. Each agent can be executing multiple tasks concurrently, at the same time as different agents are executing different tasks. To support task synchronisation, we need a means to make tasks explicit. This can be done using the @*do task* action property. This generates a unique task identifier. To access this identifier, use the @*task ?taskID* action property, which binds the variable to the new task identifier.

You indicate that a given task has succeeded or failed with the respective action properties @*do done* and @*do fail*, where you use @*task* to name which task you are referring to. The task identifier can be passed to the rule via a variable in one of the rule's condition chunks.

To synchronise across tasks, use @*all* to queue a chunk when all of the listed tasks have succeeded. Here is an example that initiates three tasks and then queues a chunk when they have all successfully completed. The action chunk with @*failed* is used to specify a chunk to queue when any of the listed tasks have failed (Fig. 12).

```
# rule to initiate several tasks and queue chunk
# process2 {} when all these tasks have completed

process1 { } =>
    a {@do task; @task ?task1},
    b {@do task; @task ?task2},
    c {@do task; @task ?task3},
    process2 {@all ?task1, ?task2, ?task3},
    recover2 {@failed ?task1, ?task2, ?task3}
```

Fig. 12. Example that starts three local tasks and initiates further behaviour when they have all succeeded or any one of them has failed.

Use *@any* in place of *@all* when you want to trigger behaviour when any one of the listed tasks have successfully completed. Note that you are free to set the chunk type and additional properties for action chunks with *@all, @any* and *@failed.*

@do task is used for tasks running on this agent. To initiate tasks on another agent you should use *@on*, as shown in the following example (Fig. 13):

```
# same thing, but this time on different agents
# initiating agent notified when tasks complete

process1 { } =>
    a {@do task; @on agent1; @task ?task1},
    b {@do task; @on agent2; @task ?task2},
    c {@do task; @on agent3; @task ?task3},
    process2 {@all ?task1, ?task2, ?task3}},
    recover2 {@failed ?task1, ?task2, ?task3}
```

Fig. 13. Example that starts three remote tasks and initiates further behaviour when they have all succeeded or any one of them has failed.

The *@on* action delegates a task to a named agent, and further ensures that the assignee informs the originating agent when the task succeeds or fails. Note that task identifiers are locally scoped to each agent, so that *@on* has the effect of running *@do task* on the assignee agent, generating a task identifier local to that agent. The reverse mapping of task identifiers is automatically applied when the assignee informs the assigning agent when a task completes.

5.4 Ideas for Further Exploration

Decentralised decision making has the potential for improved system resilience to faults and attacks. This can include the means for agents to take on and switch roles as needed, and the use of techniques based upon consensus, auctions and automated negotiation.

Agents may need to temporarily suspend tasks when they need to attend to higher priority interruptions *(@suspend ?taskID).* When conditions permit, the task can be resumed *(@resume ?taskID).* For this purpose, the task identifiers can be recorded in the chunk database for a given cognitive module and retrieved when needed. This is akin to asking yourself what was I doing before I was interrupted, and what should I do next?

It is natural to divide tasks across a hierarchy of sub-tasks. One way to achieve that is with *@all* and *@any*. This further relates to mechanisms to support reinforcement learning where the agent improves its skill by trying out different ways to achieve its goals. The agent needs to split its attention between working on getting better at a given task, and exploring fresh possibilities. Stochastic heuristics can be used to propose new rules to try out. We can then use success or failure to adjust the strengths of the rules to improve the agent's performance on the next attempt at this task.

One approach is to record the sequence of executed rules and propagate the reward backwards through time. From a biological perspective, this isn't convincing. Our episodic memory is better tuned to recalling events forwards in time rather than backwards. Another approach is to propagate rewards across the task hierarchy to update conditional probabilities for selecting a given rule in a particular context. This requires the agent to track the currently executing tasks and their relationships.

Single agent learning generalises to multi-agent reinforcement learning (MARL) where agents collaborate to explore the training space with the possibility of beneficial emergent behaviours. Agents can be fully cooperative, or more realistically, can have their own agenda with potentially conflicting goals.

As the number of agents is scaled up, inspired by human organisations, we are likely to require ways to limit agent to agent communication, e.g. via the environment (stigmergy), communication with physically nearby agents, or constrained by functional roles, using a scale-free peer-to-peer network where some agents are better connected than others. Agent to agent communication could be structured, e.g. JSON messages, or unstructured, e.g. using natural language, sound and images for agents based upon Generative AI.

Further inspiration comes from the relationship between the motion of individual water molecules and use of fluid mechanics to describe the motion of very large numbers of such molecules. If we can find efficient ways to describe the behaviour of large populations of agents, then we may be able relate the behaviour of an individual agent to the field equations for the population.

6 Longer Term Prospects

The success of Generative AI has shown the effectiveness of artificial neural networks for learning complex statistical relationships from large corpora. Recent work has focused on applying large language models to agentic frameworks. How does this relate to symbolic frameworks such as Chunks & Rules?

One idea is to focus on the generation of pertinent situation reports (SITREP). According to the Persimmon Group [10]: *The SITREP provides a clear, concise understanding of the situation, focusing on meaning or context in addition to the facts. It does not assume the reader can infer what is important; rather, it deliberately extracts and highlights the critical information. A good SITREP cuts through the noise to deliver exactly what matters: what is happening, what has been done, what will be done next, and what requires attention or decision.*

In principle, we could use large language models to generate SITREPs using plugins to access external systems in what is a neurosymbolic approach. An agentic framework such as *langchain* [11] could be used to guide a large language model (LLM) through engineering the prompts and interpreting the responses in an extended dialogue.

Another idea is to train LLMs to generate chunks and rules conditioned by natural language descriptions, so that users can describe their objectives rather than having to specify the details of how to fulfil those objectives. Generative AI is well suited to dealing with gaps and ambiguities in their input. The feasibility of this approach is boosted by success on accelerating conventional programming tasks. The main challenge is to curate a large enough corpus of training data. If this can be achieved, we could then use LLM based agents to supervise lower-level agents using symbolic approaches.

An intriguing research question is whether neural networks are better suited to controlling industrial systems than symbolic approaches? In principle, neural networks could be better suited for managing ad hoc situations involving ambiguity and novelty based upon their breadth of knowledge. Neural networks could likewise offer greater performance at learning new skills given back propagation and gradient descent as a means to train models with billions of parameters. This will be particularly effective when training models in risk-free simulated environments rather than having to try everything with real robots.

7 Relationship to Previous Work

In some cases, you want the control behaviour to always be exactly the same in the same context. In other cases, non-deterministic behaviour is appropriate when flexibility, adaptability, and the means to handle uncertainty are needed. They allow multiple possible outcomes and can incorporate randomness or probabilistic decision-making. Chunks & Rules occupies a middle ground. It is deterministic except a) when multiple rules match the current state of the buffers and b) when the chunk in the buffer matches multiple chunks in a memory retrieval operation (@do get). The developer can choose whether to take advantage of this flexibility as appropriate to the use case.

Another approach is fuzzy control based upon approximate reasoning (fuzzy logic). This treats scalar values as a blend of named values, e.g. a temperature could be given as 30% cold, 60% warm and 10% hot. The fuzzy mapping is defined in terms of transfer functions for the named terms in respect to the scalar value (fuzzification).

Control is expressed using if-then rules, e.g. if it is cold, turn the heating to high; if it is warm, turn the heating to low; if it is hot, turn the heating to off. The outputs of the rules are blended based upon the input blend, and the control output determined by the reverse mapping (defuzzification). Boolean operations in rule conditions are mapped to Zadeh operators over the blend values, e.g. 30% for cold in the above example. Logical AND corresponds to the minimum such value, OR to the maximum value, and NOT to 100 minus the value when working with percentages.

FIPA (the Foundation for Intelligent Physical Agents, part of IEEE) [12] defines a framework for agent-to-agent interaction using communication acts based upon speech act theory, and enabling agents to express intentions, make requests, and share beliefs, subject to a shared ontology. FIPA further defines a variety of protocols that specify the rules and patterns for sequences of messages, ensuring orderly and predictable conversations between agents, as a basis for complex multi-step interactions like negotiations and resource allocation.

IEC 61131-3 defines a suite of graphical and textural programming languages for programmable logic controllers (PLCs) [13]. Programs can be executed once, repeatedly on a timer or on an event. The initial version of the standard was released in 1993. The 3rd edition released in 2012 features object-oriented extensions.

CAYENNE [14] is a rule-based control logic generation solution for industrial automation that introduces a rule base with domain-specific, reusable rules to automate simple, re-occurring design and implementation tasks for control logic. A rule engine applies pre-specified rules on the requirement documents, e.g. process control piping and instrumentation diagrams, to automatically generate parts of the IEC 61131-3 code.

Node-RED [15] is a NodeJS-based framework for developing event-driven distributed systems using a web-based tool for editing flow diagrams for connected nodes, and JavaScript for the code implementing the nodes. Synchronisation support includes the means for a node to wait for events from all of its input connections before proceeding.

By way of contrast, Chunks & Rules uses a simple text-based representation for facts and rules rather than structured diagrams. Actions are asynchronous, proceeding concurrently with reasoning. This mirrors the brain where conscious decisions delegate actions to the cortico-cerebellar circuit, where the cerebellum coordinates in real-time the activation of large numbers of muscles based upon sensory models in the cortex.

Learning is initially heavily dependent on cognition, but with repetition, is compiled into fast and subjectively effortless skills, e.g. riding a bicycle or playing a musical instrument. An exciting challenge for future work is to understand how we can replicate this for multi-agent neurosymbolic systems.

Acknowledgments. The author is grateful for support from the Nephele and SmartEdge projects by the European Union's Horizon Europe research and innovation programme under grant agreements No 101070487 and No 101092908, respectively.

References

1. ERCIM. https://www.ercim.eu
2. Smart Warehouse demo. https://www.w3.org/Data/demos/chunks/warehouse/
3. Nephele. https://nephele-project.eu
4. W3C Web of Things. https://www.w3.org/WoT/
5. Demo Source. https://github.com/w3c/cogai/tree/master/demos/Swarms/visualise
6. rosnodejs. https://github.com/RethinkRobotics-opensource/rosnodejs
7. ACT-R home page. http://act.psy.cmu.edu
8. Chunks & Rules spec. https://w3c.github.io/cogai/chunks-and-rules.html
9. Robot control demo. https://www.w3.org/Data/demos/chunks/robot/
10. Persimmon Group SITREP template. https://thepersimmongroup.com/situation-report-sitrep-template/
11. Langchain home page. https://www.langchain.com
12. FIPA home page. http://fipa.org
13. IEC 61131-3. https://webstore.iec.ch/en/publication/68533
14. CAYENNE. https://www.koziolek.de/docs/Koziolek2020-ICSE-SEIP-preprint.pdf
15. Node-RED. https://nodered.org