Autocatalytic Endogenous Reflective Architecture

Nivel E.[1], Thórisson K. R.[1], Dindo H.[2], Pezzulo G.[4], Rodriguez M.[3], Corbato C.[3], Steunebrink B.[5], Ognibene D.[4], Chella A.[2], Schmidhuber J.[5], Sanz R.[3], Helgason H. P.[1]

[1] Reykjavik University / CADIA
[2] Universita degli studi di Palermo / DINFO
[3] Universidad Politecnica de Madrid / ASLAB
[4] Consiglio Nazionale delle Ricerche / ISTC
[5] Scuola Universitaria Professionale della Svizzera Italiana / IDSIA

RUTR-SCS13002

2013

Reykjavik University
School of Computer Science

Technical Report

Table of Content

1 Executive Summary

The document describes the architecture of the system being developed in the project. The principal contribution of this work is to the engineering of autonomous systems in a general fashion, i.e. independently of the target domain. This work is not “biologically inspired” and contributing to modeling natural intelligences is not an objective of the project.

The main challenge set for the system is to adapt in dynamic open-ended environments with insufficient knowledge and limited resources. The system is to perform in real-time and extract knowledge from the domain it operates in. In particular, this means discovering meaningful states in the environment and learning skills by observing intentional agents in the domain.

No system facing the complexity of the real world is able to learn effectively and efficiently from scratch. For each domain our system is plunged in, we hand craft a bootstrap code (called the Masterplan) that consists of the necessary initial and minimal knowledge to observe and act in the domain.

The system is model-based and model-driven, meaning that its architecture is unified and consists essentially of dynamic hierarchies of models that capture knowledge in an executable form. The system is thus composed of executable models. From this perspective, learning translates into building models, integrating them into the existing hierarchies and revising them continuously. This perpetual rearranging of the internal agency of the system addresses the first objective of the HUMANOBS project:

- to design an auto-reconfigurable architecture.

Learning is based on model building, which in turn is driven by goals and predictions, i.e. the evaluation by the system of the observed phenomenology of the domain. In other words, the system infers what it shall do (specification) and observes ways to reach these goals (implementation). In that regard, this addresses the second objective of the project:

- to build a system that can auto-generate specifications for skills and behaviors based on their observation.

Learned specifications and implementations are highly context-dependent, which raises the challenge of identifying when to reuse (or refrain from reusing) learned models. Specifications and implementations are built hierarchically, which means that the system is able to reuse previously learned skills, however said skills are by design executable in parallel: this raises the challenge of coordinating (or sequentializing) the operation of the models that implement said skills. The architecture has been designed from the onset to solve these issues, and this addresses the last objective of the project:

- to build behavior generation and coordination mechanisms for the reproduction and reuse of observed skills.

The architecture has been designed in a principled way (each part of the architecture is based on the same architectural template), and organizes the cooperation of four continuous processes: Model Acquisition, Model Revision, Compaction (or model compression) and Reaction (reactive behavior in the domain). It is the interplay of said processes that not only ensure the viability of the whole system but also improves its performance. For example, acquiring models only requires a few examples, performs in real-time and is fast, but this process requires another process that revises models also rapidly and in real-time, and this in turn is supported by the reactive behavior of the system that pays attention only to meaningful entities and phenomena – which it has learned previously, the whole cycle having been bootstrapped by the Masterplan. This is in sharp contrast with traditional Machine Learning approaches which ignore the other cognitive processes of a system and thus, left on their own, require enormous quantities of training examples and cannot perform in real-time.

A functional prototype has been developed as a proof of concept of the architecture, and the preliminary results reported in this deliverable strongly indicate that the architecture is sound and that our approach towards the engineering of autonomous systems is tractable and promising.

This prototype has been expanded, generalized and optimized furthermore into the final state of the architecture (with respect to the time frame of the project). The implemented final system satisfies the requirements of the project (although the evaluation results are presented in a separate deliverable) and, in particular, is completely domain-independent.

Future developments and related research avenues have been identified and will bring the architecture beyond the requirements of this project. In the main, these developments are aimed at addressing the issues of (a) adding curiosity and imagination to the system's capabilities and, (b) controlling the autonomous bottom-up growth by means of top-down allonomic constraints.

2 Introduction

This document describes the model of the system being developed in the HUMANOBS project: a system able to learn socio-communicative skills by observing people.

We have to mention from the onset that the system described here is being developed as a contribution to the *engineering of autonomous machines*. The architecture is not “biologically inspired” and making contributions to the fields of cognitive science, philosophy of mind, is not per se an objective of the HUMANOBS project (although the project results might prove to be relevant to these fields, and others). Furthermore, it is worth noting that the HUMANOBS architecture has been designed to be domain-independent. In particular, the domain of application of the architecture (the socio-communicative domain) has been chosen only because it presents a set of technical challenges suitable for stressing soft real-time systems.

In the main, the system is designed *to reason in dynamic open-ended environments with insufficient knowledge and limited resources* – in that we adopt the working definition of intelligence proposed by Wang [2]. Additional requirements include *scalability* with respect to the complexity of the environment and *generality* – independence from the application domain.

Knowledge is understood as *operational constructive* knowledge: it is executable code, which implements the system’s ability to act in the application domain. Bearing this definition in mind, a system acquiring/revising/integrating knowledge is in fact re-programming some of its own constituents – as has been proposed in [3]. From this developmental perspective, the problem is cast into a Machine Learning framework, within which the system controls the expansion of its own code during the several stages of development that are needed to keep said system adapted to its environment. This expansion is controlled using some given prior knowledge (called the Masterplan) and stems from some bootstrap code – essentially, hand-crafted learning and programming code, also provided in the Masterplan.

We identify two sub-systems, the Domain System – designed to act in the primary application domain (social communication), as defined by the project - and the Meta-System – whose domain is the Domain System itself. The Meta-System controls the development of the Domain System, i.e. the former modifies the code of the latter - including the Masterplan.

The architecture is called AERA, which stands for Auto-catalytic Endogenous Reflective Architecture. In a nutshell, this acronym refers to both the operational and semantic closures, discussed in [3] – see section 7.1 for details.

AERA is to be described at three increasing levels of detail:

- the architectural principles – this was the object of the first release of this document;
- the plunging of these principles in the domains identified for the Domain System and the Meta-System – this constitutes the focus of this second release, and focuses on the implementation of a functional prototype;
- last, this release (the third and last) specifies the architecture in an executable form: all the constituents of the architecture are specified at code-level – that is, all structures and processes are specified at the level of detail needed for a direct implementation in the Replicode language – the language and its underlying executive are specified in [1].

The work reported here is two-sided: it is (a) the inception of the architecture of a real-time system as a coordinated set of components that cooperate for the efficient and effective continuous learning from the environment it is plunged in and, (b) the integration of efforts carried out by “satellite” work-packages focusing on specific components and concepts put in use in the whole architecture. These works are: model observers (WP2), imitation learning (WP5), agent in the domain (WP6) and, last but not least, reasoning process (WP3). In particular, WP3 and WP5 each conducted case studies on other domains than the target domain of the project. These domains are, respectively the Pong video game which addresses specifically the time issues of a real-time system plunged in the continuum of physics which does not involves hard challenges in planning, and the Sokoban video game which addresses both the problem of imitating goals from observed surface behaviors as well as the case of decision making in a discrete environment more suited for investigating reactive planning issues. Both case studies propose specific architectures that shall not be confused with the reference architecture presented in this document. These studies constitute feasibility studies and proofs of concepts that proved valuable for the achievements presented here. Most of the results of these studies have been integrated as parts of the whole final architecture.

Following this introduction, section 2 summarizes the general requirements for the system and section 3 describes our general design approach. Section 4 presents the set of principles that govern the design of the architecture. Section 5 describes the architecture that results from putting the principles into action in the domain for meeting the requirements. Section 6 presents the implementation of a prototype developed as a proof of concept for the principles presented throughout this document. Finally, section 7 describes the final and complete incarnation of the architecture in an implementable form.

3 General Requirements

This section summarizes the high-level requirements of the system having a significant impact on the architectural design. They are formulated independently from the domain and task environment. The behaviors expected from the system, its constraints and evaluation procedures are described in a separate document [4].

Achieving Multiple Goals – Initial goals are defined by the programmers, and the system has to build new ones dynamically, in accordance with the situations it faces. It also has to build improved ways of achieving goals - either known or new.

The system has to handle cases where multiple goals have to be achieved concurrently. In particular, constraints are considered goals (constraints define states to be avoided) and, as such, compete with other goals.

Resource Limitation – The system has to run in realistic conditions and has therefore to perform despite computing power limitations, and within acceptable times depending on the task. This requires the system to control the resources it allocates to its various internal processes.

The maximum available computing power is a cluster of 8 computing nodes[1]. This computing power envelope does not include I/O processing, which is to be performed on additional machines. I/O processing comprises any task that is not performed cognitively, i.e. hard-coded functions that do not evolve over time.

Acceptable response times vary depending on the task environment, but, as we strive for a level generality encompassing at least the experimental setup defined by the project (learning socio-communicative skills), it is safe to establish these response times in the range of 0.5 to 5 seconds, that is, reasonable response times compared to the latencies observed in human communication.

Knowledge Limitation – The system has to handle situations for which it has incomplete knowledge. In particular, the environment is not axiomatized (nor can it be) and the prior knowledge given to the system does not have to be a consistent and exhaustive model of the domain.

Notice however that part of the system can be fully specified: this is the case for example, for the system's operational boundaries.

Knowledge Extraction[2] – The system has to be able to identify and capture skills as they are performed by relevant entities[3] in the environment, and integrate them, i.e. adapt them to its own capabilities and reuse them. It also has to revise this new knowledge according to new facts.

This acquisition/integration/revision cycle does not have to be performed on the fly: the system can do so during dedicated sessions where the system becomes a passive observer of a scene where entities demonstrate the skills to be acquired, and it is acceptable that knowledge integration and revision are performed while the system is not busy interacting in the environment.

Dynamic Open-Ended Environment – The environment is expected to change as the system performs: events unforeseen by the programmers are to be expected. In addition, the environment is open-ended: the system has to handle new phenomena resulting from principles for which little knowledge has been given or accumulated previously.

However, the causal principles and the ontology of the environment are *not* expected to change dramatically over short periods of time. Only *incremental changes* are part of the requirements.

Cross-Modality – The system has to handle simultaneous inputs sampled from several sub-domains (e.g. auditory and visual inputs), possibly at different frequencies.

[1] Each node features eight 2.66 GHz CPU cores (reference CPU: Xeon E5430) and 8 GB of memory; the interconnects are one 4x DDR Infiniband link and two gigabit Ethernet links.

[2] Although this is not a requirement of the system per se – it is rather a mean to an end – it is nevertheless part of the requirements of the project.

[3] Such entities are any individual displaying consistent behaviors in the environment. This includes but is not limited to humans: other entities are also considered. Depending on the domain, these can be, for example, software components, automatic devices, animals, etc.

Preemptibility – The system has to be able to suspend its current operation in favor of more urgent tasks, and resume the former when possible, if it is still desired.

Robustness – When the system achieves consistently a set of goals within some constraints, it shall continue to do so despite novelty occurring in the environment, provided this novelty changes the initial goals and constraints only marginally and provided the system can possibly remain in its (fixed) operational boundaries.

Scalability – The system shall maintain a constant level of performance independently from the complexity of the environment and of the complexity of the knowledge it has accumulated.

The level of performance is defined as the number of goals actually achieved, moderated by their respective importance – these goals include keeping accuracy and response times within the expected limits. All these goals are defined by the programmers as part of test procedures.

Generality – Whereas the system has to include domain-dependent knowledge, its model – the architecture – cannot, and shall be applicable to any domain and task environment of a complexity similar to the one of the initial domain (social communication). In that respect, we have to mention two notable limitations to the generality of the architecture: (a) it does not target hard real-time systems - full determinism is not a requirement and response times are not to be guaranteed - and, (b) optimality of either the system’s behaviors or of their delivery is not a requirement either.

4 General Design

Given the set of requirements mentioned in the previous section, the system is designed to exploit experimental facts and prior knowledge rather than to rely on a decision theoretic framework. The system is case-based and accumulates representations of these cases, derives models of their outcomes and, possibly, models describing actions that lead to reaching some goals in these cases. In that sense the system is *model-based*. These models are executable, meaning that the processes within the system are the execution of such models. This execution is in turn controlled by other models: the system is also *model-driven*.

Actually most of the knowledge is represented as models: these either predict the behaviors of entities of interest (*forward models*) or prescribe courses of actions to be taken in given contexts (*inverse models*). Entities are not restricted to individuals observed in the environment but also include internal phenomena: in other words, models are also used to predict the system's own internal behavior or to generate these behaviors. The representation of knowledge using executable models is independent of the scale and scope of what is described. In particular, components of the system are modeled at arbitrary levels of granularity: the system is thus composed almost entirely of models.

According to this approach, learning is a process that generates models, integrates them in the existing structure that forms the system, and evaluates the resulting behaviors to compute feedback for generating better models. Learning is a process of the system and as such, results from the execution of models and could also be learned, in the same fashion. Notice however, that meta-learning is not a capability that is expected from this version of the design.

The system performs four fundamental activities, largely independent from each other: they are performed *concurrently* and *continuously*. These activities can be outlined as follows:

I. **Model Acquisition –** the system strives to transform observations of interesting behaviors into reusable models (either predictive or normative):
   - build models of the behaviors of the entities it interacts with; these new models either add to the existing set of models or are replacement candidates for existing models in need of improvement;
   - among these models, to identify the ones that are relevant to the system itself as describing possible behaviors that can solve some known goals and to integrate them in the system for future reuse.

II. **Reaction –** the system reacts to events using its current models:
   - sample events from the environment;
   - identify the situation and select the models deemed most appropriate to predict the outcome of the situation or to prescribe courses of action for reaching goals;
   - execute these models – some of which produce commands to control the sampling of the environment;
   - measure the progression with regards to the current goals, and possibly generate new goals;

I. **Model Revision –** the system checks the validity of its current models:
   - evaluate the performance of the newly integrated models and generate feedback (i.e. internal events); performance evaluation is carried out by monitoring the execution of the models and compare their outcome with new evidence with respect to goals;
   - according to the revision results, increase or decrease the rating of existing models (roughly speaking, identify them in a scale ranging from good to poor performance), and produce more appropriate candidates: this impacts the model selection during the Reaction activity.

II. **Compaction –** the system strives to transform its models into more compact forms – also called compression. This activity addresses specifically scalability issues:
   - from existing models, produce new models that are more general: these can replace a

number of narrower models and thus free memory and reduce computation time (as less models would need to be executed). Generalization possibly leads to generating new ontological categories;

- integrate models known consistently as good performers into the Masterplan. The Masterplan contains robust knowledge and its models are revised less often than others: assuming that the environment does not change dramatically over short periods of time, models having been identified as good performers over a long period are less likely to be invalidated by small changes in the environment – such models are thus likely to be fairly general.

These four activities are high-level processes that result themselves from model execution and, theoretically, other instances of the same activities can be applied to the former instances, in a recursive fashion. For example, the system has to cope with possibly large amounts of inputs, not all of them relevant for its current operation: some filtering has to be performed for the sake of keeping performance in the prescribed envelope. However, such a filtering cannot be entirely hard-coded as the system is meant to operate in open-ended environments and sampling the environment has to be learned. In other words, we have to assume that all inputs are a-priori relevant, and asserting actual relevancy is a process to be acquired. Sampling the environment is a behavior of the system, resulting from the execution of some models: these models can in turn be subjected to the aforementioned activities, respectively: new models for sampling can be acquired (I), put in use in a reactive fashion (II), revised (III) and, generalized and integrated in the Masterplan (IV).

Conceptually, we refer to the Reaction process as the Domain System, since it is responsible for the production of actions in the domain, whereas the Meta-System encompasses the other three activities, being their overall mission to tune the Reaction, by modifying its models, to improve its performance.

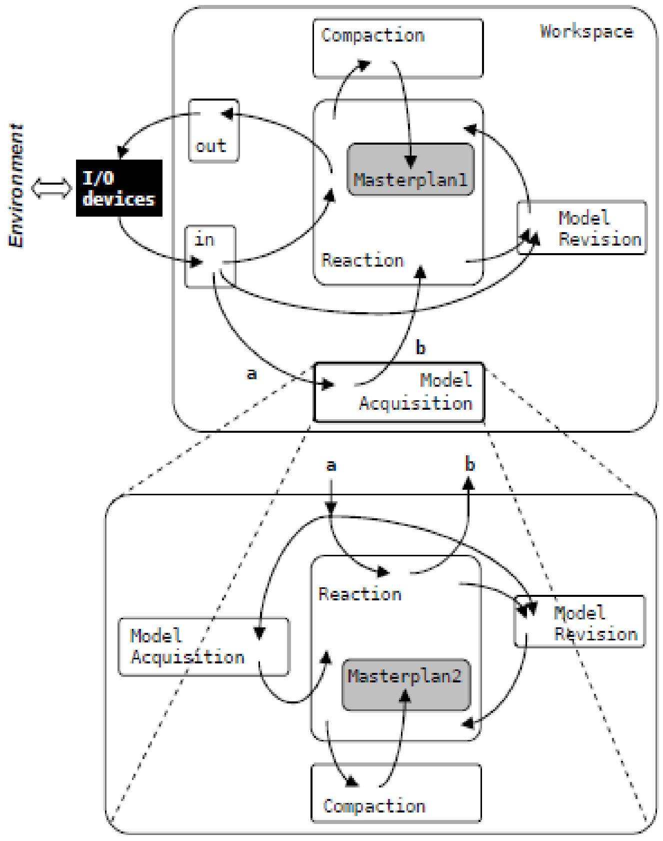


Figure 1 – General Architecture

The global workspace is organized in sub-workspaces, each containing models implementing a specific process. For example, each of the fundamental activities *is implemented this way.*

Flows of knowledge (arrows) form several feedback loops spreading all over the system. Encapsulation is not strictly enforced and workspaces (groups in Replicode, defined in [1]) allow inspection by code from other workspaces. For example, the models forming the reactive part are modified by both the Model Acquisition and the Model Revision.

Some of the system's activities can themselves be implemented as the main system is – here, the Model Acquisition is broken down into an organization similar to the top-level one. The input workspace for the Model Acquisition is the same as for the reactive part of the top level. Its output workspace is the top level reactive part itself.

The Masterplan is distributed in the system: in the case depicted here, it is the union of Masterplan1 *and* Masterplan2*. The bootstrap code is part of the Masterplan.*

I/O devices are hard-coded external software components that constitute the interface between the dynamical part system (the "cognitive" part) and the environment.

To predict the outcome of its own actions in a given situation or the outcome of other's actions, the reactive part of the system relies on models of the entities it interacts with. Such models are also executable models and are organized internally as the system itself: the model of an entity implements the same four main activities - notice that, as models are executable, the model of the system is the system itself. Models of relevant entities are part of the reactive component of the system where they are executed concurrently and potentially continuously, depending on their relevance and available computing resources.

The proposed architecture is of a "single mind". The system maintains models of some entities in the environment but this does not mean that the system results from some kind of collaborative operation of these models. The system is not a multi-agent system in the sense that each of the models does not have any goal of collaborating to achieve the system's own goal - they merely capture the behavior of observed entities.

The general structure of the system is a global workspace containing inputs, processed by models. As models constitute knowledge, they are themselves possible inputs for other models: we therefore make no distinction between code and data: everything is executable code[4], contained by the global workspace. Notice however, that the workspace is organized in sub-workspaces (see section 4.1 below).

From a technical point of view, the system relies on a distributed computational substrate - Replicode. Its execution model is *data-driven*: model execution is triggered by the occurrence of salient knowledge, that is, knowledge deemed worth processing by other models. Models operate like rules: they match input knowledge and produce other knowledge – possibly other models. Models are all executed concurrently and their execution is also controlled by other models (models can be activated or deactivated). Once produced, knowledge cannot be modified, but instead, is granted a limited life expectancy: useful knowledge has to be explicitly kept resilient[5] whereas useless knowledge is simply left to its fate and dies unattended. Last, the substrate injects runtime reflective knowledge in the workspace. These are notifications of the execution of the models and constitute internal inputs - the substrate provides the system with a trace of its own execution. In short, such a system is a distributed reflective production system capable of adding/deleting models very frequently and in which the control of both the execution of the models and the relevancy of inputs is performed collectively (by sub-sets of models) – consult [1] for full details.

The system is *unified* as every knowledge fragment is executable code and every process is implemented by knowledge fragments. Unification applies primarily to cognitive processes, i.e. processes that have to be modified at runtime. Depending on what the system is required to produce in a given application domain, some processes are not identified as cognitive processes: the implementation of these can be externalized, that is, hard-coded in software components that communicate with the global workspace but are not included in it. Externalization applies in particular to device drivers – components that perform actions in the

[4] at least at this high-level of description: Replicode allows for example the definiton of markers on code, goals and sub-workspaces: these are not executable – see section 4.1.

[5] i.e. models have to produce it again or, more simply, to extend its life time.

environment in a blind manner, i.e. controlling the sensors and actuators of the system. They constitute the interface between cognition - the domain of reflective code modification, driven by “programming laws” - and the environment - driven by the laws of physics - and transform representations between these two domains. Depending on the requirements, these transformations can vary in complexity: the less complex the more able the system is to adapt its control over the real world. For example, in the HUMANOBS project, the main part of speech recognition is left to external components: this means that the way phonemes and words are recognized cannot be altered dynamically by the system, thus limiting the system’s ability to improve its connection to the environment. However, the system is still able, a-posteriori, to filter and modify the output of the speech recognition component, for example, to reduce the amount of false positives according to its current knowledge (e.g. its expectations).

5 Architectural Principles

This section describes the architectural principles governing the design of the HUMANOBS architecture. These principles define the general organization of computation, that is, the exploitation/generation/compaction of the structure of the system by fundamental algorithms: learning, decision making, attention control, resource allocation and generalization.

The structure of the system develops dynamically and incrementally as the enrichment/revision of an initial code base, the Masterplan, also described in this section.

5.1 Low-Level Organization

The fundamental building blocks of the system are objects, instances of classes defined in Replicode. We thereafter list the most essential of these structures in a short summary, as they are already described in the language specification [1].

States – States are defined as patterns of temporal sequences of knowledge fragments. Such fragments are instances of object classes defined in Replicode. Patterns specify constraints on both the content of each fragment, on correlations between these fragments and on the relative order of their occurrence.

Goals – Goals are specified as states, i.e. as patterns specifying states to be reached. We distinguish between goals and instantiated goals. A goal is an objective that a system *may* pursue, whereas an instantiated is a goal *actually* pursued.

Groups – Groups are workspaces where reduction occur - reduction is the production of knowledge triggered by the matching of an input with a pattern. Groups aggregate knowledge, and knowledge can be member of several groups simultaneously: in other words, models are *shared*. Models can control a group as they would control any single object, by means of changing its saliency/activation[6] – thus impacting the behavior of the group's content.

Models - Knowledge is executable code: it is also said to be *grounded*, that is, it expresses ontologies in terms of actions[7] that can be performed either by the system itself or by a model of the described entity – in this case, this model is built and maintained by the system. Models are implemented at three levels of organization:

- **Level 0:** This is the lowest level, the level of Replicode constructs. Models are encoded as programs, forward models and inverse models: all are instances of predefined Replicode object classes, each being executed as a single lightweight thread. Programs produce code whenever some knowledge matches the patterns they define – as for specifying states, patterns specify complex time series of knowledge occurring in the workspace. Notice that Replicode also allows coding programs to react to negative knowledge: the *absence* of inputs able to match their patterns. Forward models are similar to programs except that they define two sets of patterns: a set to identify a state, and a set to identify some actions. They output predictions about the outcome of the specified actions taken in the specified state. Symmetrically, inverse models prescribe courses of actions to be taken in a given state for reaching a target state – here also, both states are specified in distinct sets of patterns.
- **Level 1:** Objects of level 0 output code that may become input for other objects - this forms reduction graphs. Models at this level of organization are implemented as such reduction graphs. For example, a model M_0 can specify patterns to identify the context of validity C of another model M_1 and activate M_1 when some inputs match the patterns that specify C.
- **Level 2:** Several models of level 1 can be aggregated in groups. This allows an entire aggregate to be treated by other programs as one single object.

Markers – Patterns of temporal sequences specify part of the dynamics of the system. This comes at the cost of a high apparent complexity – when such patterns are to be matched by

[6] Activation is an attribute of models defined to control whether they can be executed or not when salient inputs match their patterns. Saliency is another attribute of any kind of knowledge defined to control whether said knowledge is worth processing or not.

[7] We make no difference between sensing and actuating: both are actions since they control the I/O devices responsible for carrying out these operations in the enviroment.

others. Markers are structures for abstracting patterns: they encode and name the trace of a persistent relationship between objects. For example, the state of an object being an instance of a certain category is generally established by the execution of some model, and when this model asserts the state in a relatively constant way (e.g. the model always outputs the same category when it matches the object), it is useful to encode this fact as a static structure, thus saving computation time. This is achieved by defining a marker holding references to both the object and the class (and also possibly to the model mentioned above), in the fashion of a predicate. In particular, the notification of model execution is encoded as such markers. Notice also that markers are not executable code – they are the trace of an execution.

5.2 High-Level Organization

Low-level structures do not prescribe any constraints on the computation which is actually performed at the low-level of organization. In this section we introduce architectural constraints on the semantics of the computation implemented using the low-level structures.

5.2.1 Schemas

The basic control structure of the architecture is called a *schema*. A schema is an anticipatory controller composed of an inverse model coupled with a forward model, as proposed in [5] and [6] and put into an architectural perspective in [7]. We adapt the proposed structures and organizations to comply with (a) our data-driven execution model and, (b) our need for learning schemas (i.e. to modify them dynamically).

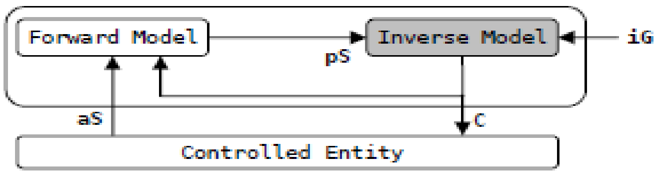


Figure 2 – Schema

The inputs of the inverse model are a prediction of the state of the system (pS) and an instantiated goal (iG). It outputs commands (C) that modify the behavior of the controlled entity. The inputs of the forward models are the actual state of the system (aS) and a copy of the outputs of the inverse model. Its outputs are a prediction of the state of the system, i.e. the outcome of the commands given the actual state.

Technically speaking, the controlled entity and the models depicted above are implemented as Replicode structures. In particular, the controlled entity is generally a group in which the forward model attempts to match its patterns and in which the inverse model injects commands, i.e. objects that will be processed by code located in the group. They will either match some inputs defined by models describing the entity – in case the system is predicting the behavior of the entity -, or be sent to I/O devices by programs in the group – when the system actually attempts to control the entity in the real world.

Both models in a schema are of arbitrary complexity: for example, it may require several models, each being valid only in some specific contexts. The ability to determine the context and to select the right model is out of the scope of the schema in question: this schema has the knowledge required to control one entity, but not for controlling itself in the whole system. This requires the operation of the schema to be *integrated* with the operation of other parts of the system which have the necessary knowledge. This means that other schemas are dedicated to control other schemas, in a recursive arrangement, also called *Integrated Cognitive Control* (see [8]), where control is distributed in a hierarchy of models that permeates the entire structure of the system.

Controlling the internal structure of a schema by models exterior to the schema is to be performed by other schemas, the former being the entity controlled by the latter – to allow such control, the structure of a schema is made *transparent* to its controller. To illustrate this approach, let us consider Wolpert's example [6]: to design a system able to lift a carton of milk, not knowing if it is empty or full. Depending on the context (empty or full), the system will have to behave differently (here, apply more or less force to lift the carton). This means that the forward model of the lifting schema depends on the context. if the carton is empty, applying a certain force will lead to a small change in the carton's position, and vice versa. In our approach, the forward model of the lifting schema is composed of two embedded forward models, one for each context. The identification of the context – and the selection of the right forward model – is performed by another schema: its input is the difference between the position of the carton predicted by the forward model of the lifting schema and the actual position sampled from the environment. Its output is the activation of one of the embedded model, and deactivation of the other.

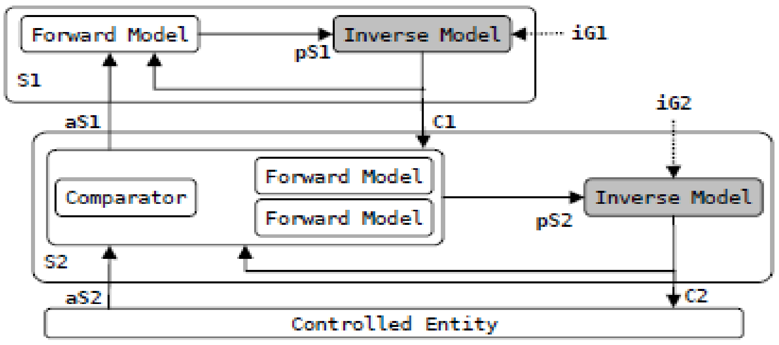


Figure 3 – Schema Control

The schema S1 controls the forward model of the schema S2. aS2 is the sampled position of the carton, and C2 is the command to an I/O device (a motor command). The forward models in S2 output predictions of the future position of the carton (pS2). A program (comparator) computes the difference between the prediction pS2 and the actual position aS2: this forms aS1.

In S1, the forward model anticipates the difference between the predictions made by S2, and the actual position of the carton: this forms pS1. The goal of S1 (iG1) is to have this difference equal to zero, and its output (C1) is the activation of one of the forward models in S2, say the forward model tuned to predict the next position of a heavy object. Another instance of S1 would be needed to control the activation of the other forward model in S2 (predicting the next position of a light object - not represented).

This way of controlling forward models in a schema also applies to the case where the inverse model of a schema is implemented as a group of models: some schemas can modify the implementation of the inverse models of another schema. Schema control can be exerted at different levels of granularity: from the constituents of schemas (as discussed above) to the schemas themselves: the activation which is controlled would then be the activation of the entire schema. Control is not restricted to model/schema activation: models can be added to a schema and schemas to schema assemblies (these are defined in section 4.2.2 below).

The activity of a schema is a trace of a state. In our example, the state of the carton being full or empty is captured by the activity of the schema controlling the validity (activation) of the forward model that predicts the position of the carton. Referring to Figure 3, the trace of the execution of S1 actually represents the state of the carton: the carton is full when S1 activates the corresponding model in S2. This high-level representation of a state is compatible with the low-level representation since the latter defines patterns of temporal sequences, whereas the former produces an instance of such sequences[8].

Shall a model (like S1) be needed to control more than one schema (like S2), then three main design options are available:

I. the model is replicated;

II. the model is shared, i.e. made a member of the groups implementing the other schemas;

III. the output of the model is encoded in a marker to be matched by some code in the controlled schemas, and internal parts of the controlled schemas are shared.

[8] Notification of schema execution – therefore of model execution – is provided automatically by the Replicode executive.

The main difference between these three techniques is a difference in plasticity (the ease of code modification): the last is to be used in parts of the system having reached a late stage of development, whereas the first is more adapted to earlier stages – the second standing in between. Since markers encode statically the trace of a process, the revision of such knowledge has to be explicitly coded for – a marker asserts that a predicate consistently holds and shall this have to be verified, the execution of some extra code would be needed which represents an overhead. This however is less likely to happen when knowledge has been consolidated, i.e. the system has accumulated reasons to believe that said knowledge has become a persistent fact. More over, the inputs of such a shared model have to be general, i.e. computed the same way regardless of the schemas to be controlled: in the example above, the inputs of S1 are computed locally by S2 and shall a schema S3, similar to S2, be also controlled by S1, then the two comparators shall be fused into a single one and externalized from the schemas it originates from. In contrast, the second option does not require such a generalization and uses as many variants of the comparator as needed. The first option is redundant and can sustain local modifications without impacting the other copies of the model: this technique is more appropriate to the cases where the system frequently needs to re-implement some of its components.

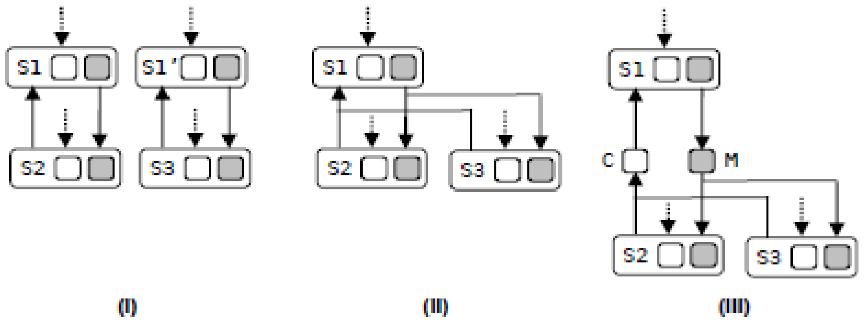


Figure 4 – Plasticity of Schema Control

A schema is represented as a pair of two groups: one implements a forward model (white) and the other, an inverse model (grey); a goal is represented as an arrow (dotted line) pointing downward to the middle of a schema.

I – Early stage of development: *the controlling schema (S1) is replicated. On one hand, this arrangement is robust to local modification, at a minimal cost in terms of processing time. On the other hand, it takes more time to execute than the other arrangements – roughly speaking, it involves more models and these are larger.*

II – Intermediate stage of development: *the controlling schema is shared by the controlled schemas (S2 and S3).*

III – Late stage of development: *sharing occurs at decreasing levels of granularity: the comparator (the model C, taken from Figure 3) is shared by S2 and S3; S1 produces a marker that in turn, is converted into a control signal (by the model M). This arrangement is more integrated than the first one (a): it is more general and requires less computation time, but is more expensive to modify.*

5.2.2 Schema Assemblies

There is no restriction on the semantics of schema outputs, and in particular, schemas can instantiate goals. These in turn, can match patterns defined by inverse models: such an organization leads to hierarchical assemblies of schemas where goals are generated from the top to the bottom.

In general, the collaboration between schemas (i.e. schemas achieving the sub-goals of a top level one) is indirect: there is very limited direct coupling between schemas, as their interactions are mediated by the common workspace – the controlled entity. This loose coupling is critical for allowing the system adding new schemas to an assembly without having to modify the existing ones (re-wiring).

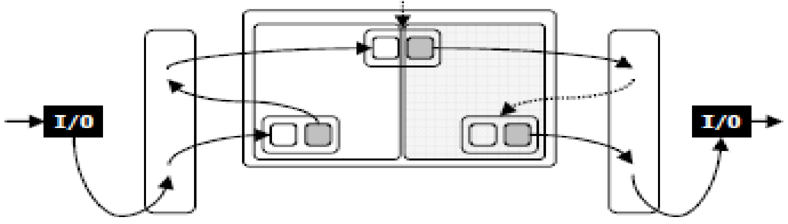


Figure 5 – Indirect Coupling

For clarity, the controlled entity has been split into two groups. Predictions flow from the bottom up (left side), while goals flow from the top down (right side). The rank of a schema in a hierarchy indicates the reactivity of the control loop it implements (the higher, the less reactive).

Schemas are unaware of each other: they communicate via the entity they control in common and the environment. For example, goals are published in the entity by the top schemas, and they match patterns specifying the admissible goals for the lower-ranking schemas.

Assemblies are dynamic: their composition depends on pattern matching and is thus subjected to change shall some models in the schema be replaced by others (e.g. integrated as a result of learning), or as a result of changing the activation of models reacting to the occurrence of the goal (as mentioned above, in the case where models are selected by other models), or also, as a result of changes of the goal saliency (as its importance may vary over time). In short, instead of static data structures, assemblies are identified as *reduction graphs*.

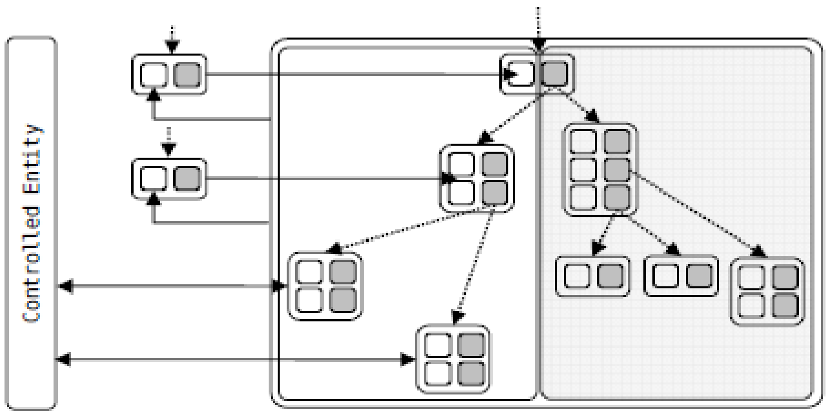


Figure 6 – Hierarchical Schema Assemblies

Each group of schemas at a given level in the hierarchy provides case-based alternatives for implementing a model of a schema one level higher. The selection of these alternatives according to context is performed by other schemas (left side), as mentioned above. This control of schemas' internals by other schemas (as depicted in Figure 3 above) is represented by plain arrows. The controlling schemas do not necessarily belong to the same hierarchy as the schema they control.

Schemas in a given hierarchy take their inputs from the same group (the controlled entity) and also output their productions in that group. The hierarchy has the same global structure as a schema's; it has also the same capabilities - controlling an entity in an anticipatory way.

The architecture follows the principle of Integrated Cognitive Control mentioned earlier and every activity of the system is implemented as a set of hierarchical schema assemblies: Reaction, Model Acquisition, Model Revision and Compaction are four distinct sets of assemblies, and this also holds for their sub-activities. The last three activity control the

Reaction in the same fashion a schema controls the internal models of another one, as discussed above.

A schema producing a goal is a specification of an action: essentially, it describes what to achieve when a state is reached, with regards to its own goal. Goal-driven schema assemblies thus constitute specification-to-implementation hierarchies[9].

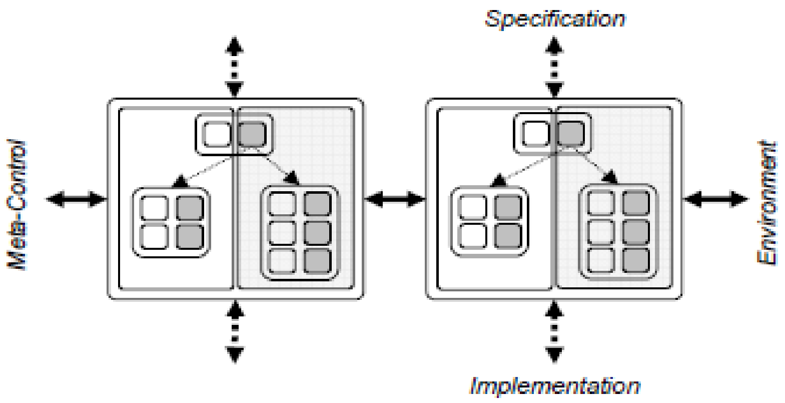


Figure 7 – High-Level Control

Schemas controlling other schemas can do so at arbitrary levels of granularity: here, an assembly (left) controls the internals of another one (right).

The system is organized in two global orthogonal hierarchies of schemas: (a) an implementation hierarchy (vertical) where predictions flow bottom-up and goals top-down and, (b) an integrated cognitive control hierarchy (horizontal) which drives the activation and addition of models or schemas in the first hierarchy.

High-level schema control can also be implemented at various levels of plasticity, as discussed above, using the same techniques.

5.3 Learning

The general learning principle is a goal-driven procedure involving three independent activities:

I. for a given goal, produce a candidate model able to solve the goal – i.e. implement a model according to specification;
II. integrate the resulting model in any pre-existing model structure that produce instances of the goal in question;
III. observe the performance of the model and revise it accordingly, i.e. resume the procedure: produce an alternate candidate, integrate it and revise it.

This procedure applies to learning models at arbitrary level of granularity – from individual low-level models to entire schema assemblies: this level of granularity is directly related to the level of abstraction of the goal for which models have to be learned.

5.3.1 Implementation From Specification

This activity is carried out by inferring causation between events and encoding such causal relations into models. This is performed in two distinct ways, outlined as follows:

I. observation: the system observes humans demonstrating a task, attempts to find a correspondence – to identify common capabilities between itself and the model of the demonstrators – and translates the causal chain of events it has observed into a model.
II. induction: the system applies case-based reasoning to infer *linear transfer functions* (more generally, piece-wise linear functions). Prior knowledge to do so is included in the Masterplan in the form of hand-crafted models.

Observation - The process of observing causal sequences of events is guided by prior

[9] When read the other way around (from bottom to top), these hierarchies can also be considered defining increasing levels of abstraction.

knowledge, and proceeds as follows – consult [9] for full details:

a) the system identifies the input data that are the most relevant to subsequent computations (essentially, decision making, see section 4.4 below), and ignores the others. Such relevant data form the Area of Interest (AoI) – a subset of the input data: the system will try to build models able to produce only these relevant data;

b) the system attempts to detect correlations between the input data and the data in the Area of Interest. In other words, the system tries to identify sequences of events that are believed to cause the occurrence of the data in the AoI.

As a result, the input data that have triggered the production of useful state predictions or useful actions with regards to the current goals are identified as forming the Area of Interest, i.e. a subset of the input data. For each event in the AoI, the system stores the patterns said events matched during the first iterations of the decision making processes. This allows the system to map the AoI to the set of patterns mentioned above.

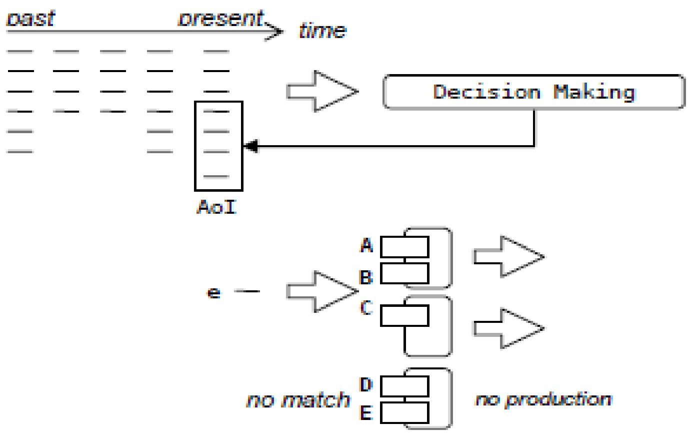


Figure 8 – Finding the Area of Interest (AoI)

Top - *One short horizontal line represents one event, the set of which is the stream of input data. The system (actually, the progress monitoring process, defined below) identifies the event(s) in the input section that triggered the production of useful data: these events form the AoI.*

Bottom – *When events from the stream of input data are presented as inputs to models (rounded rectangles) (by a decision making process), the system remembers which patterns (rectangles at the left of the models) they matched. This allows each event (e) in the AoI to be mapped to a set of patterns (A, B and C).*

Inferring causal relations among the input data is performed by a hard-coded component, a classifier called the *Correlator*. Its inputs are:

a) the input data;

b) one event taken from the AoI.

The output from the Correlator is a time-ordered sequence of pairs containing (a) a set of events that constitute a context for (b) a sequence of pairs – the definition is recursive, a terminal pair being a sequence where the context is empty and followed by a set of events. The Correlator’s output has the following form:

$(p_0, ..., p_n)$ where the pairs p_i are:

(c_i, s_i) where the context c_i is:

$(c_0^0, ..., c_0^{N0})$ a set of events,

and the sequence s_i is either a context c_j followed by a sequence of sets of events:

$(c_j, ((e_0^0, ..., e_0^{P0}), ..., (e_m^0, ...,e_m^{Pm})))$ - the set of events having occurred at the same time.

or:

a context c_j followed by a sequence of pairs.

Events are identified as members of a context when they constitute invariants for a sequence.

The transformation of the values carried by the output of the Correlator into variables – i.e. the transformation of objects into patterns – is performed by hand-crafted programs: they are encoded for each class of objects, and substitute variables to values that change over time according to the (known) semantics of the values.

The time horizon needed to limit the scope of the computation of the AoI is determined by the construction of *episodes*. An episode is, in our system, the history of all inputs leading to the satisfaction of a goal, from the time when the goal has been instantiated. Implementing by observation is carried out incrementally, one goal at a time for a given hierarchy. For example, an episode contains sequences of events such as $a \rightarrow b \rightarrow c$ where *c* matches a goal *g*, and what the system observes is in fact "to satisfy *g*, produce *c from a*". Episode construction is driven by goals and if the system knew already how to produce *b*, then the episode would have comprised only the events happening between *b* and *c*: what would have been learned is "to satisfy *g*, produce *c from b*".

When a model is built from a goal-driven observation as discussed above, the system has to assess whether or not it can reuse this model, i.e. it has to find a correspondence between the entity that was performing an action and its own capabilities. Finding such a correspondence is not an independent process per se, but is rather the result of integration (see section 4.3.2 below): essentially, no correspondence is found when integration fails. In other words the "correspondence problem" is a question of integration. Notice that correspondence is not only a question of ascribing capabilities observed in foreign entities to the system, but is also needed the other way around. The system has to learn not only skills from people, but also has to learn models of people and ascribe to people some of its own behaviors and goals. We have identified the finding of a correspondence to a successful integration and this still holds in the present case: foreign entities are modeled as assemblies and finding a correspondence from the system to such entities is also the positive result of an integration.

Induction – As for the transformation of events into patterns mentioned above, producing models able to reach a given goal results from the execution of hand-crafted code that leverages prior knowledge.

Technically, the task is akin to an informed goal regression, i.e. to produce a model *M* matching some patterns and producing some code able to match the given goal *g*: (a,b,c) $\rightarrow$ M(a,b,c) where *M* denotes a model such as, for any data *a'*, *b'* and c*'* matching respectively the patterns *a*, *b* and c, M(a',b',c') matches the target state defined by the goal *g*. Notice that *a*, *b* and *c* are taken from areas of interest computed as discussed above in the case of learning by observation. Producing *M* given *a*, *b*, *c* and *g* is guided by the ontology in the Masterplan, i.e. models that describe the semantics of *a*, *b*, *c* and *g*. For example if *g* is a pattern targeting a position, *a* a velocity, *b* an initial position and *c* a duration then, given an ontological inverse model such as "to compute a position update, given the initial position, the velocity and duration of the movement, compute initial_position+velocity*duration", finding *M* is quite simple, at least in non-ambiguous cases, assuming both sufficient coverage of surface phenomena (here, models for controlling trajectories in the intuitive geometrical space) and relevance in the hand-crafted ontology[10].

The activity of acquiring models - regardless of the way candidate models are computed – is a cognitive process and as such, is implemented as a goal-directed hierarchy of schemas, and controlled by another. The latter activates further model acquisition processes with respect to a goal when it has evidence of the system's poor performance while attempting to achieve the goal. This calls for a continuous evaluation of the system's abilities, carried out by a continuous process – a progress monitoring process – that accumulates the success rates of the existing schemas and implements decision procedures to control the need for further model acquisition.

5.3.2 Integration

Model candidates produced by the activity described above are low-level constructs; these models need to be integrated in schemas, and in turn, these schemas in pre-existing assemblies.

[10] To learn such ontological inverse models is not a requirement of the project and constitutes an open issue in itself (inductive programming): future projects will focus on teaching the system how to acquire such models, i.e. teaching it how to program by demonstration, leveraging the system's imitation learning capabilities as they are defined in the current architecture.

Integrating a Model in a Schema - This is performed in a rather straightforward fashion. In the case of an inverse model, integration is carried out by identifying the goal defined by an existing schema to the productions of the model: any newly acquired model is injected into the group receiving all inputs from the environment, and activated; any of its production that matches an existing goal triggers the injection of the model into the schema implementing the goal. Productions generated this way are quickly removed from the system (e.g. they are output with a very low resilience) since they are generated only for the purpose of model identification and do not correspond a-priori to any ongoing decision making process. The case of a forward model is handled in a similar way: instead of identifying the productions of the model with the goal of the schema, these productions are identified to the input patterns defined by the inverse models of the schema, the identification procedure being the same as for inverse models.

Integrating a Schema in an Assembly – This integration follows the same principle as above, the only difference being of granularity. This is possible thanks to the similarity in structure and semantics between a schema and an assembly – in short, a schema is to an assembly what a model is to a schema.

5.3.3 Revision

Revising models has to be performed differently depending on the semantics of the model:

- forward models: models are tested individually to evaluate their predictive performance;
- inverse models: models integrated in existing hierarchies are compared to existing ones to determine their functional performance.

Revising Forward Models - The system verifies continuously the adequacy of known forward models by trying to reproduce new input data from past input data. We call this process a *coverage test*, in other words, the measurement of the ability of existing models to predict data in the AoI according to new facts.

To check the validity of known models, the system proceeds as follows:

a) execute the models on a subset of the input data, i.e. on the input data from a time *t* to *t* minus a certain time horizon;

b) check if all the input data in the AoI at time t and after (i.e. at *t* plus another time horizon) have been produced by the models;

c) as for integration, the outputs resulting from the coverage test are discarded.

If the test is positive, then the system reinforces its belief (using a confidence value) that these models will produce accurate predictions in similar (pattern-wise) situations - the situation is recognized. Otherwise, some of the data in the AoI have not been produced by the models – then the system faces a situation it cannot anticipate.

In the case of success, nothing new is to be learned (only the confidence values are increased). In the case of failure, the confidence value associated with the model is decreased and a new model has to be built.

Revising Inverse Models – Newly acquired inverse models constitute tentative alternatives to existing models and to evaluate their relative performance the system has to undergo several *simulated executions*, one for each candidate model to be tested against the best known model. These multiple evaluations are performed offline – when the system is not expected to act in the environment (think “sleeping time”): the system reevaluates the production of its decisions using the episodes it has accumulated so far - they are selected according to the goals they correspond to.

Instead of performing in the actual environment, these simulations execute the models of the entities involved in the episodes. The procedure we just described assumes that the new model implements a behavior of the system. If instead, said model was implementing the behaviour of an entity in the world, the evaluation would be very similar to the one described above, except that the sparing partner would the model of the system itself, and the evaluated (sub-)system would be the model of the entity.

Confidence values are adjusted as for the case of evaluating the performance of forward models.

The revision activity is controlled by the same progress monitoring process that decides whether or not the current ability to achieve a goal is in need for better models – as discussed in section 4.3.1 above.

5.4 Reaction

The purpose of a Reaction activity in the system is to decide what course of action to take in order to achieve its goals. Put briefly, this activity merely exploits knowledge that has been either learned or provided beforehand in the Masterplan. Said knowledge is a set of schema assemblies, and the Reaction activity consists in executing these assemblies. The efficacy of the resulting behavior therefore depends directly on the performance of the schemas composing these assemblies, and on their relevance. No advanced planning abilities are expected from such a decision process which can be classified as a “reactive planning” system based on experience. its performance relies essentially on the performance of the Learning activity.

The models composing a schema are themselves possibly composed of several models forming alternate reduction graphs, not all of them equally performing with regards to prediction (forward models) or action (inverse models). Given our requirements, it is obviously out of question to try out all these possibilities and some pruning of the computation is needed before execution. To control a data-driven system, one has to control the data, i.e. the model inputs: this is performed by several processes.

Any reactive activity is supported by a process called *Attention Control* which filters the inputs to be processed according to the system’s expectations within a short time horizon. This process is dedicated to controlling continuously the saliency of the inputs of the reactive activity it supports. Since models only reduce inputs that become salient, the outcome of the Attention Control is that fewer reductions are actually performed.

The Attention Control focuses the computation on what the system expects to come next. There is the need for a similar process accounting this time for what other entities than the system are likely to do next – in the context of human communication, one could think of “mind reading”. This is performed by executing the model of these entities of interest. As discussed earlier, these models are also learned and maintained by the system’s other main activities (Model Acquisition, Model Revision, Compaction) and consist of other sets of schema assemblies. These assemblies can be considered implementing Reaction activities, similar to the system’s own and as such, are also supported by their own instance of Attention Control. As for the Attention Control, the outcome of executing models of foreign entities is a control of the saliency of the inputs.

Notice that both the Attention Control and the execution of models of foreign entities are performed continuously and independently from the main Reaction activity.

Technically, the control of input saliency results in saliency spreading across the graph of objects stored in memory. We have seen in section 4.1 above that the trace of reduction graphs can be encoded using markers, a Replicode low-level construct. These markers actually represent relations between objects, thus defining an object graph (objects are of any class, and in particular they can be models). Replicode provides a way to propagate changes of an object’s saliency to its markers and to the objects said markers hold a reference to: this instruments the spreading of saliencies across object graphs[11] in an *additive* way - the changes are accumulated and averaged. For example, if the system focuses on motion, the Attention Control will compute an increase of saliency for the category “position” (an entity assumed to be part of the system’s ontology) and propagate this increase to any input related to a marker encoding a position in space-time and in turn, to any moving object perceived by the I/O devices. If simultaneously, the execution of an assembly modeling one of these moving entities happens to predict some noticeable outcome (an outcome deemed significantly favorable or unfavorable with regards to the system’s current goals) say, for small, yellow round objects, then the combination of these saliency changes would drive the system to pay attention primarily to moving tennis balls and to discard other moving objects.

The Attention Control is aware of the goals of the system including its constraints (since they are implemented as goals). Saliency spreading applies to any object and therefore to goals and execution traces. In particular saliency spreading can be directed by goals that is, be controlled to flow from goals to related reduction traces and to the inputs having triggered the reductions and so on. When resources are critical (i.e. they would be typically guarded by a semaphore in a multi-threaded application), the system shall attempt to avoid conflicts before executing any action, as early as possible – although here again, no guarantee on conflict avoidance is

[11] N.B.: the breadth and depth of the propagation are controllable programmatically.

expected. To do so, the Attention Control allocates a high saliency to the constraints (goals) expressing said scarcity and direct a positive flow of saliency from these constraints to the objects not taxing the resources, negative otherwise. Conflict prevention results from conflict prediction and this requires representing the system resources in an executable form, as are regular entities: each of these resources is also modeled as a set of schema assemblies provided by the Masterplan since they are unlikely to change over time – no learning is necessary for the models of these resources.

The assemblies modeling foreign entities are learned and this shall also apply to the Attention Control: it is designed as a reaction activity, implemented by a set of assemblies, but unlike the assemblies modeling entities it cannot be learned by observation and relies solely on induction-based learning.

5.5 Generalization

Generalization is an independent process that runs continuously, in parallel with all other processes. Its purpose is to produce compact models from the available ones. This is intended to reduce the consumption of computational resources: general models encompass a breadth of cases that would otherwise need to be captured by more models which would in turn lead to costlier performance.

Generalization is performed primarily by making analogies between objects either at the low-level of organization (states and models) or at the high-level (schemas and assemblies).

Low-Level – Analogy is defined as the confluence of reduction chains – thereafter M(s) denotes the reduction of a state s by a model M:

- two states s_0 and s_1 are analogous in a context C with respect to a model M when there exist two set of models $\{M_0^0, \dots, M_n^0\}$ and $\{M_0^1, \dots, M_m^1\}$ such as $M_n^0=M_m^1=M$ and $M_n^0(M_{n-1}^0(\dots M_0^0(s_0)) \dots)=M_m^1(M_{m-1}^1(\dots M_0^1(s_1)) \dots)$. The context C is defined as the state of activation of the models of both sets[12].
- two models M_0 and M_1 are analogous in a context C with respect to a model M and the specification of a state, S, when there exist two sets of models $\{M_0^0, \dots, M_n^0\}$ and $\{M_0^1, \dots, M_m^1\}$ such as $M_n^0=M_m^1=M$ and, for each state s matching S, $M_n^0(M_{n-1}^0(\dots M_0^0(s)) \dots)=M_m^1(M_{m-1}^1(\dots M_0^1(s)) \dots)$. C is defined as above.
- two sets of models $\{M_0^0, \dots, M_n^0\}$ and $\{M_0^1, \dots, M_m^1\}$ are analogous in a context C with respect to the specification of a state, S, when $M_n^0=M_m^1$ and for each state s matching S, $M_n^0(M_{n-1}^0(\dots M_0^0(s)) \dots)=M_m^1(M_{m-1}^1(\dots M_0^1(s)) \dots)$. C is defined as above.

Given these definitions, low-level analogy detection is performed by analyzing the reduction traces within groups of interest (an instance of Attention Control is needed). In more details, this is either done by a "post-mortem" analysis or performed on the fly - executing probes[13] in parallel to the computation being investigated.

High-Level – There is no discontinuity between the levels of organization and analogies between schemas derive directly from analogies between models, as a combination of thereof: two schemas are analogous when (a) their internal models are and, (b) their goals are also analogous. Along the same line, sub-assemblies have the structure of and perform like schemas and thus, are deemed analogous when their respective goals and parts (schemas or sub-assemblies) are analogous.

The Generalization activity does not need to be performed online, but is instead performed offline, as are for example, the integration and revision processes.

5.6 Resource Allocation

To keep the performance of the Reaction activity within the envelope prescribed by the requirements, a resource allocation strategy needs to be implemented.

The following processes are executed while the system is not busy interacting in its environment, and have thus no impact on system performance:

[12] This includes their activation level in all the groups they are members of, the groups' activation thresholds, the group's update periods, the models' resilience, etc.

[13] models coded to detect confluence.

- induction;
- model acquisition,
- revision;
- integration;
- generalization;
- Masterplan integration.

The remaining processes have to share the available computing power - they all are processes supporting the system's main Reaction activity:

- execution of the system's own model;
- execution of the models of foreign entities: these are reactive activities themselves and are each also supported by a specific Attention Control,
- episode construction;
- Attention Control.

Accordingly, the principal resource allocation policy consists in driving the activity of the many instances of Attention Control, by implementing a trade-off between broad and narrow attention. The span and depth of attention refer to ways of feeding with inputs the hierarchies it controls. Broad attention means spending computation time on the execution of the many alternatives models in a schema can be composed of, whereas narrow attention means spending time on the execution of the successive levels of implementation found in the hierarchies. In other words, resource allocation control is implemented as a trade-off between breadth-first and depth-first execution of models in a given Reaction activity.

Narrow and broad attention have an impact on the system's behavior: broad attention supports the handling of unexpected events and novelty whereas narrow attention helps the system achieve better reactive performance (speed and accuracy).

Resource Allocation is a sub-process of an Attention Control process and is implemented by subsets of assemblies among the assemblies defining the Attention Control.

5.7 Masterplan

The Masterplan is the set of models, schemas and assemblies that represent knowledge given beforehand to the system by its programmers. This is a convenient way to bypass the intractability of teaching a machine from first principles. However, as the repository of past experience, the Masterplan has to be expanded with knowledge acquired and consolidated after the system is put in service. Consolidated knowledge is knowledge that has been consistently evaluated (by the revision process) as being consistently accurate - in the main, models with consistent good performance. In that sense, the Masterplan can be regarded as a long-term memory, and the models it contains are revised relatively rarely (also by the revision process). as these have already proven their robustness to incremental changes. In this respect, the revision process shall ideally contain policies to detect changes able to invalidate knowledge stored in the Masterplan – this has not been designed yet.

Integrating knowledge in the Masterplan is the duty of another independent activity, also performed offline.

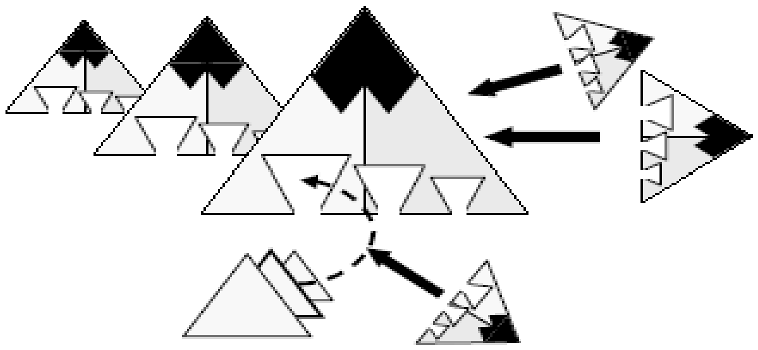


Figure 9 – Masterplan

Each process is the execution of a set of hierarchical goal-driven schema assemblies. Each set is potentially controlled by other processes (on the right).

Ideally a model shall be one assembly assuming that the programmers would design it from first principles: this is rarely the case - even when it is possible - and upper layers have to be left non unified (black regions). The Masterplan does not necessarily provide models covering the whole breadth of the entity it controls (dents at the bottom of the assemblies). For example, it may not cover the whole spectrum of surface phenomena observed in the environment. Candidates for these missing parts are produced as a result of learning, and the ones consolidated over time are integrated in the Masterplan by some other process (bottom).

Besides domain-dependent models and goals, the Masterplan contains learning and programming skills (implemented as schema assemblies if they are deemed to be expanded, modified and/or generalized, or coded “free-style” otherwise); as mentioned earlier these are essentially:

- induction of piece-wise linear transfer functions;
- induction of patterns from values;
- model acquisition;
- episode construction;
- model revision;
- model integration;
- compaction: generalization and Masterplan integration.

When implemented as assemblies, each of these skills are defined along with their own Attention Control (implemented as another set of assemblies if identified as having to change dynamically).

The bootstrap code is part of the Masterplan: it is the initial version of the system, i.e. the primitive version of its model.

6 System Architecture

The architecture results from applying the principles to a domain. In this section we give an overview of how the processes are to be designed, and we also present a summary of the respective contributions of each part of the architecture to the compliance to the general requirements. Finally, we propose guidelines to design the content of Masterplan that depends on to the domain.

6.1 Overview

As a general design rule, any process in the system can be implemented as the system itself – but that does not mean that they have to. In this section we present our design choices regarding the implementation of each of the processes.

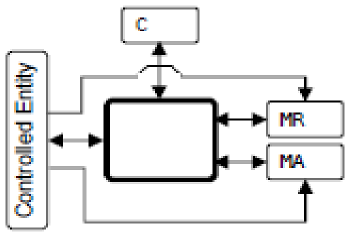


Figure 10 – Architectural Template

The architectural template taken from Figure 1, section 3. Not all components are mandatory when applied to the control of a given entity. The Model Acquisition (MA) and Model Revision (MR) parts take some of their inputs from the controlled entity itself – not only from the assembly they steer (in bold outline).

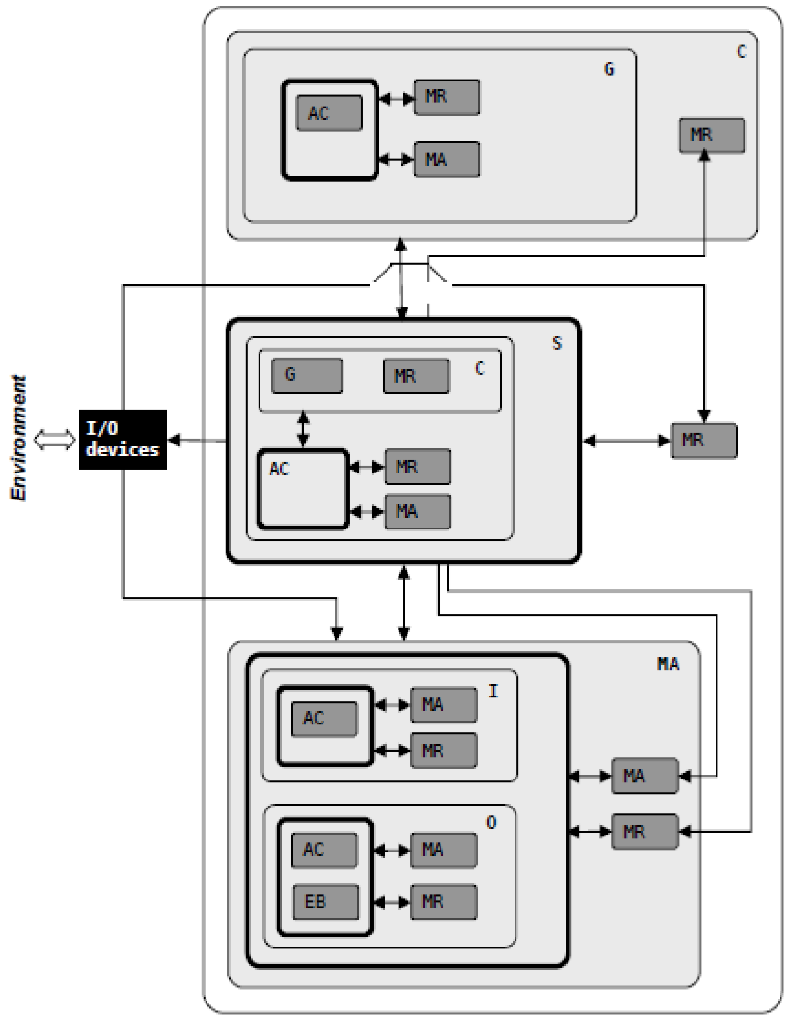


Figure 11 - Architecture

In gray, code that is not learned. In bold, reactive parts. In light grey, assemblies other than the one identified in rectangles (e.g. models of entities, etc.).
C: compaction - G: generalization - AC: attention control - MA: model acquisition
I: induction - O. observation – EB: episode building - MR: model revision

We give thereafter a summary of the top-level assemblies and their main constituents.

Self – The main reactive process. The main sub-component is an instance of the Attention Control, to be learned. the whole process is also learned.

Compaction – it is composed by two processes, an instance of the generalization process and an instance of the model revision process - the latter is dedicated to integrate consolidated knowledge in the Masterplan. The generalization part is learned and is driven by an instance of attention control.

Model Acquisition – Its two sub-components are the induction and observation processes. Both are learned. The whole process is also learned. Both of its sub-components are driven by an instance of attention control.

Model Revision – This process is not learned. It is controlled by an instance of the progress monitoring process (not shown in Figure 10).

6.2 Checklist

This section summarizes the contribution of each of the general principles and/or specific processes discussed above to the fulfillment of the general requirements.

Achieving Multiple Goals – The main contributor is the Reaction activity. Our parallel data-driven execution model allows the pursuit of multiple goals concurrently.

Resource Limitation – Over short time horizons, the Attention Control is the essential part of the architecture meant to allocate computation time. In a similar fashion, the execution of assemblies modeling entities of interest also contributes significantly to the allocation of resources. Over longer time horizons, the Compaction activity (generalization of models and integration of knowledge in the Masterplan) contributes to reducing the workload.

Knowledge Limitation – Learning models, schemas and assemblies is the principal way to acquire missing knowledge. Knowledge is revised in permanence, this maintaining its accuracy and relevance.

Knowledge Extraction – Imitation learning is the main contributor. Induction-based learning also contributes to this capability although to a minor extent.

Dynamic Open-Ended Environment – Models, schemas and assemblies are all designed for modification at runtime. Modification results from integration, which results itself from observation and/or induction. Response times vary depending on the depth of the reaction loops involved (their span across goal-directed hierarchies). These loops result from schema integration and are thus fully controlled by other schema assemblies, which are eventually controlled by the system's attention. the latter is initially coded so that urgent situations get a high saliency, thus steering the system toward handling them with low latencies (short loops).

Cross-Modality – Models, states and goals are defined using patterns that specify complex temporal sequences of events. Correlations and time constraints on these sequences are part of the definition of patterns. These are exploited concurrently by many models. High-level synchronization is one among many capabilities implemented in these models.

Preemptibility – At low-level, models are state-less. At higher levels, states are knowledge distributed in the system; as such they are subjected to Attention Control: resuming a process is performed by resuming the system's attention on some of the states produced before the preemption point. In particular, this applies to states that encode execution traces.

Robustness – Several candidates for a given model co-exist and this population is constantly trimmed down by the revision process: model evaluation is performed in isolation and no interference can actually happen.

Scalability – Compaction is the essential part addressing scalability issues.

Generality – We restrict our target to the class of sub-optimal soft real time systems. The architectural principles do not make any assumption on the target domain.

6.3 Plunging the Architecture in the Domain

The architecture remains to be plunged in the domain. This means coding in the Masterplan the models that are relevant to the experimental setup chosen for this project. The entities for which an initial model is to be provided are the following:

- a generic human: this means defining the goals and constraints that apply to a (caricature of a) human. These are for example models like "to convey information, look for feedback" or "during a conversation, humans have an attention limited to 5s". Notice that roles (for example, interviewer or interviewee) are mere sub-assemblies of the model of a human. They simply define goals and schemas for achieving them and thus are integrated in the main model. A model for each surface capability ascribed to humans in this project shall also be defined (as the counter-parts of the models of the system's own I/O devices, see below). For example, the model of a human shall include models for grasping, moving arms, etc.
- I/O devices: this means modeling the actions that they can perform in the environment (including sensing). Such models shall implement sensory feedback loops to convey immediate causality, i.e. immediate reactions of the environment to an action performed by the device;
- objects to be manipulated in the environment: these objects are always supposed to be driven by some control be it internal (e.g. a dog) or external (e.g. a cup a-priori subjected

only to the laws of physics), and thus have a goal (even if it is to “fall”, i.e. to follow the rules of gravity). In short, as counter-intuitive as it may seem, external laws are always represented as goals and schemas shared by the models of the entities subjected to these laws.

- concepts not reified in the environment (e.g. natural language constructs, abstract concepts, etc.): grounding knowledge associates action schemes and experience records to symbols. Since the machine will not have experienced some concepts involved in natural communication – nor will it ever be in position to do so – we have to fill this gap and make up some sort of “fake memories”. We would proceed by hand-crafting models for each of the needed non-reified entities;
- the machine: more accurately, the machine-in-the-domain. This means defining for the system all the prior knowledge that is domain-dependent, goals in particular (for example, extracting information from the interviewee).

The domain features “natural language” communication and this is to be understood as communication in “simplified English”. Our approach toward addressing this issue is to treat such communication the exact same way as any other complex sequence of events. Knowledge is grounded and we consider words as mere tags on models. Basic grammatical skills have to be encoded in assemblies as any other part of a decision making process, i.e. as the specification of (sequential) chains of state reduction.

7 Prototype

A functional prototype has been implemented to check the validity of the architectural design presented so far. This prototype does not constitute the demonstrator planned for release toward the end of the project. It is rather a technical demonstrator that includes some selected key processes of the proposed architecture and was used to test the desired inter-operation of said processes against a concrete experimental setup.

7.1 Objectives

The development of the prototype was driven by a few specific technical objectives, given below.

To validate the Model Acquisition/Revision processes. More specifically, this objective can be broken down in the following sub-objectives:

- To acquire models from as few observations as possible. This criteria was selected as a key requirement for tractability. Mainstream machine learning techniques usually demand large data sets to be able to operate and this raises the following two important issues: (a) the experiments need to be considerably slowed down as we would need to present the system with many cases of what has to be learned and, (b) as though it is in principle possible to do so, the computation time required to process these large data sets would prevent the evaluation and testing of one fundamental challenge: to have the system learn while it is acting, i.e. while it uses previously acquired models. It turns out in fact that requiring too many examples not only makes the experiments unduly complex and nearly intractable in an interactive setup such as ours, but also masks the inter-wining of learning and acting.
- To produce models as quickly as possible. This requirement follows the same rationale as mentioned above: it preserves the response times of the system, which is needed to evaluate the interplay of learning and acting. This objective means technically that, regardless of the number of observations needed, the system shall output models with response times commensurate to the response time of the decision making. In our case, we expect response times to be less than one second.
- To acquire models of particular cases - as opposed to models encoding general cases – and integrate them with the aforementioned general ones. We operate in this prototype under the assumption that the environment is more often regular than not, and thus, that more cases will be handled appropriately by general models rather than particular ones. However, particularities do still exist and the system needs to determine at runtime that particular models, when appropriate, shall rule over the general ones.
- To limit the proliferation of learned models. The system needs to detect the production of new models that are equivalent to existing ones – and discard the former - in order to limit the time needed to execute said models. Syntactic analysis is unlikely to suffice for solving the issue (as most of the models are context-dependent), therefore model equivalence has to be defined as a dynamic property: basically, the productions of two models operating on the same inputs under the same contexts shall be the same.

To validate the responsiveness of the resulting system. This objectives consists of two sub-objectives:

- To keep the system's response time in the appropriate envelope (five seconds for the domain of socio-communicative skills), regardless of the computation load.
- To ensure that the system can handle multi-modality in the aforementioned envelope, i.e. that the system keeps its commands on multiple channels (for example speech and deictic gestures) in synchronization.

To set the stage for further detailed socio-communicative interaction, namely:

- To design a way to handle spoken inputs.
- To design a way to produce spoken outputs.

- To design a way to handle speech contexts. This means handling spoken acts that spread over several cycles of interaction (for example, identifying an object, *then*, referring to it in another sentence).

N.B.: the system is not expected to learn words nor grammar.

To keep the size of the Masterplan as small as possible. For tractability reasons, we need to keep the ratio of innate vs learned models to a low value. The higher this ratio is, the less likely is our approach - as it stands - to be appropriate for real-world engineering purposes.

Limitations. Both the expected functionalities of the system and the complexity of the domain it operates in are limited in the following ways:

- The domain chosen for testing the prototype is the domain of socio-communicative skills limited to a low breadth (only a few words can be understood, and only a few actions performed) and to a low level of detail (the actuators for example are general and do not handle fine-grained physics). However, the real-time constraints have been kept to realistic values.
- The Attention Control is not meant to evolve and no meta-learning requirements have been established.
- The Model Acquisition and the Model Revision processes are not meant to be learned either. In short, the Meta system is fixed, and the Domain System evolves. The rationale behind this decision is that in this first implementation we need to validate the principles for operating in a domain first. The lessons learned will be applied later on, to expand the capabilities of the Meta System to control a system that is already proven. In short we did not want to evolve both systems at the same time.
- For similar reasons, we have also left the Compaction mechanism for future work, and this mechanism has not been implemented in the prototype.

All the objectives mentioned here are addressed in detail in the sections 6.4.3 and 6.6 below.

7.2 Requirements

This sub-section describes the experimental setup chosen for the prototype and the requirements for the system.

The experimental setup consists of two avatars facing each other across a table where some objects are displayed. An avatar can be controlled indifferently by either a human or the machine. The system is able to manipulate objects (grab, move, release), to gesture (pointing at some location on the table), to observe the scene (look at a location), to parse spoken commands (for example: “put a blue cube there”), and to speak (i.e. issue similar commands to any actor in the environment).

The experimental platform is the one developed for the project, and has been distributed initially on three machines (two of them hosting two standard cores – the I/O machines – the last one hosting at least four high-performance cores – this machine runs the Replicode executive).

The first requirement is to have the system learn the outcome of its low-level effectors (move, grab, etc.). This is akin to “motor babbling” and is triggered by hand-coded programs that inject random goals in the system. These goals are removed from the system after the system can consistently satisfy them.

The second requirement is to have the machine observe a basic interview between two humans, one – the interviewer – issuing commands to the other – the interviewee. The objective of this session is to test the ability of the system to learn the goal of the interviewee, namely, to comply with the goal expressed by the interviewer.

The third requirement is to have the machine take the role of the interviewee and act accordingly to what it has learned (i.e. to comply to the goal of the interviewer).

The fourth and last requirement is to have the machine take the role of the interviewer and ask the interviewee to manipulate objects, as observed previously.

7.3 Implementation Principles

Before going any further, we need to consider the technical principles on which the implementation of the prototype is based. These principles are derived from the operation of the Replicode executive and constructs. This section maps the general design presented in sections 5 and before to actual technical structures and operations.

7.3.1 Structure

This section presents the structure of the system at the (low) level of the basic building blocks of the architecture, that is, programs, models and groups.

A model operates in two ways: (a) it takes input objects (for example, samples of the environment) and produces predictions – this way of executing models is called *forward chaining* - and, (b) it takes goals as inputs and produces sub-goals – we refer to this execution mode as *backward chaining*. Notice that, with respect to the two modes of execution, a model can be viewed as either a forward model or an inverse model. It is also worth mentioning that these modes of execution are in general *simultaneous* (see section 6.3.2 below).

The encoding of a model specifies a set of output groups (where to inject the productions); a model takes its inputs from any group where it has been projected in an active state – see the figure below.

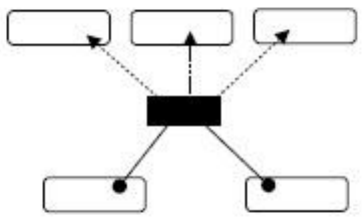


Figure 12 – A model in the architecture.

The model is represented as a black box; dashed lines represent injection of results (predictions/goals) in output groups. Plain lines denote projections of the model onto input groups (each of them is actually a view in Replicode terms, in an active state, as defined for programs). All groups (rounded boxes) are standard Replicode constructs. Computation flows from bottom (input groups) to top (output groups).

The architecture is composed of several components that execute online and concurrently, all patterned after the same template. This template is itself composed of several components (see figure below):

- One input group where samples of the entities to be controlled are injected; for example, samples of the environment are injected there by I/O devices. The code that implements the Attention Control takes its inputs from the input group and inject their results in the attention group (see below).
- One attention group in which the models and programs that realize the function of the Attention Control inject filtered input data – data of interest at a certain point in time. These data are inputs for the models that realize the functioning of the Reactive System (decision making).
- One model group that gathers all the models that constitute the reactive system and have been – possibly – dynamically produced (except the code for the Attention Control, which is not produced dynamically). This is a convenience to allow the Meta System to inspect the internal production of the Domain System. For consistency, models given in the Masterplan are also members of this group.
- Several output groups – groups where models inject their productions (goals and predictions), and where the executive injects notifications of the models' execution.

Three of these output groups are designed as the input groups of similar reactive systems that realize the functions of Model Acquisition, Model Revision and Compaction (the three of them forming the Meta System). Models shall at least inject their productions that are relevant to the domain (goals and predictions) in the input group of the system.

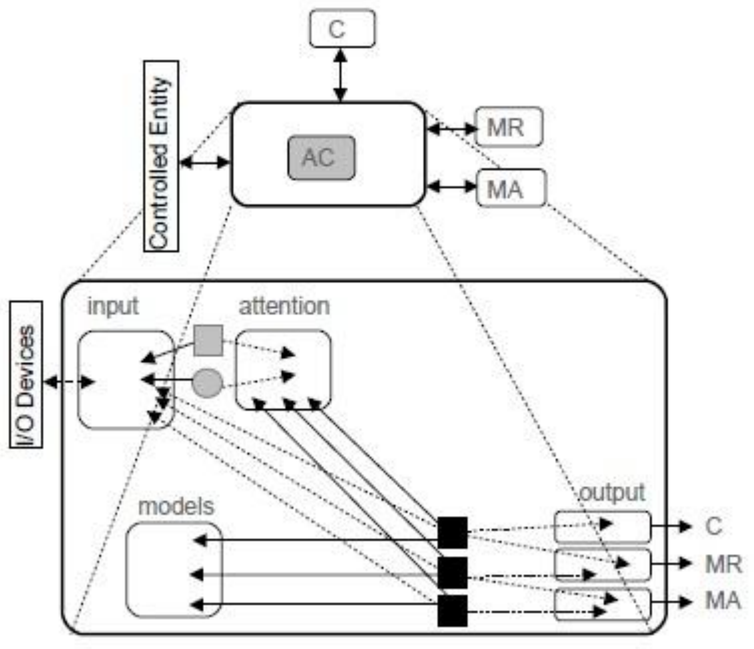


Figure 13 – Mapping of the Architectural Template for a Reactive System to Implemented Form.

Top: *the general pattern for a reactive system (as given in section 5.1 above). The controlled entity is sampled in the input group. The Domain system samples the environment via I/O devices that feed its input group. The three processes forming the Meta system sample the Domain system via the latter's output groups, projected (in a visible state) onto their own input group. These components can also read the filtered inputs of the Domain System (not represented).*

Bottom: *the programs and models that constitute the Attention Control are depicted in gray. Models that implement the decision making are represented in black. They take their inputs from the filtered raw data (located in the attention group). The "models" group is a convenience to gather all models in one place. Output groups are represented on the bottom left side. Output groups contain the result of the models' computation (goals and predictions) and also the notification thereof (reduction, success and failure). Notice that all models at least output in the input group.*

Output groups serve as input groups for any other sub-system in the architecture. Following the Integrated Cognitive Control pattern, the components that constitute the Meta System (for example the Model Revision component) admit as their inputs the notifications of the execution of the Domain System – these are for example, the notification of success or failure of a model, or the notifications of the reductions performed by models.

7.3.2 Operation

This section presents the operation of models and describes how schemas result from the operation of dynamically coupled models.

The Model Acquisition process produces not only models but also what we call *composite states*. A composite state is an object representing the conjunction of several objects (including, recursively, other composite states). For example, a composite state can define patterns like “an object that is a hand, belonging to an actor” - in Replicode terms, this pattern is encoded as a set of objects (some of which are variables, like in this case, “an object” or “an actor”) and markers (encoding the relations between objects). By definition, composite states form hierarchies of objects or patterns.

Models encode causal relations between objects (including composite states and models): some of these objects are the notification of the execution of other models (in Replicode terms, instantiated models: imdl), or the instantiation of composite states (in Replicode, instantiated composite states: icst). As mentioned in the previous section, these are injected in the output groups specified by the models, at least one of these groups being one of the input group (following the architectural template presented earlier). In other words, the notification of the execution of the system becomes an input for the system itself (and for other sub-systems, following the ICC pattern). It follows that correlations between a context of operation and the result of the execution of some models are available for processing by the Model Acquisition/Revision processes. The outputs in such cases are models that predict the instantiation of other models or composite states when matching one object (for example, a composite state) that forms a context. Such models are - by construction - one level of abstraction higher than the objects that compose them. The Model Acquisition process is driven by goals: one of these is to model the achievement or failure of models and the instantiation of composite states. That is to say that the Model Acquisition process is the primary source of organization of models in hierarchies, these stemming from the compositionability of models/composite states.

A model is essentially a directed pair of objects A and B containing variables x, y, etc.: A(x,y,...) → B(x,y,...). A and B are encoded in Replicode, as facts (fact) or negation of facts (|fact), each pointing to one object: for example (fact a t1 0.4) reads “the object a holds at time t1 with a confidence of 40%”, and (|fact a t2 0.9) reads “the object a does not hold at time t2 with a confidence of 90%”. A model whose right-side term is an instantiated composite state outputs predictions about the composite state (forward chaining), but also produces sub-goals upon receiving a goal that matched the composite state (backward chaining). If a model M0 built like: |A(x,y,...) → |B(x,y,...) - where the character “|” stands for the negation – admits an instantiated model as its right-side term (i.e. B, referring to a model M1), then such a model constitutes a strong requirement on the model pointed by B (in other words, the model reads “*without* A, there will be *no instantiation* of M1”). If two models are coded as: A(x,y,...) → B(x,y,...) and C(x,y,...) → B(x,y,...), then A and C are two *possible* solutions to obtain B (weak requirements). A given model can produce predictions (from forward chaining) and sub-goals (from backward chaining) simultaneously: the two flows - of predictions and goals – are inter-wined in the whole architecture.

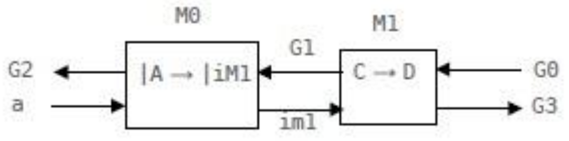
There is no explicit structure to encode a schema in our architecture. A schema results actually from the operation of two models that realize its function at a certain point in time, where the first model encodes a requirement on the second one. More precisely, the inverse and forward operation of the schema are both implemented by the coupled execution of both models – under the condition that the output of the second model matches the input of the first one – see the figure below.

Models:

M0[X,Y,...]: (|fact A[X,Y,...]) → (|fact iM1[X,Y,...])

M1[X,Y,...]: (fact (C[X,Y,...]) → (fact D[X,Y,...])

Backward chaining:

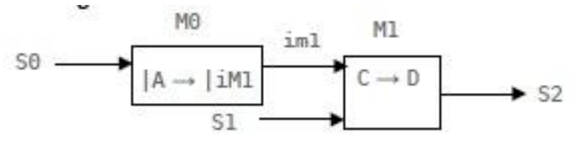


Input goal G0: (fact D[xd,yd,...]) produces (from M1) a sub-goal G1:(fact iM1[xd,yd,...]).

G1 (from M0) produces G2:(fact A[xd,yd,...])

If G2 is achieved, i.e. there exist an object a that matches A[xd,yd,...], then M0 produces an instantiation im1 of M1 and in turn, M1 produces G3:(fact C[xd,yd,...]): this implements the inverse model part of a schema.

Forward chaining:



Input state S0: (fact A[xa,ya,...]) produces (from M0) an instantiation of M1, im1: (fact iM1[xa,ya,...])

Input state S1: (fact C[xa,ya,...]) produces (from M1) a prediction S2: (fact D[xa,ya,...]): this implements the forward model part of the schema.

Schema operation:

When the pattern D in M1 matches the pattern A in M0, S2 constitutes an input for M0: this implements the forward model of a schema; the efferent copy is S1, produced as a result of the goal G3. For example, if C in M1 is a command, then G3 instantiates said command, which matches directly C in M1 (i.e. with no intermediary models). If C is not a command, then G3 will be matched by other models – these are said to be lower in the hierarchy as they operate (the process G3) as a consequence of the operation of a higher level (the level of M0 and M1).

Figure 14 – Schema as the result of the operation of dynamically coupled models.

Key: Models are denoted M[set-of-formal-arguments]: left-side term → right-side term*; Variables are written in uppercase; instantiated models are noted* iM[set-of-actual-arguments].*Bound variables are written in lowercase. N.B.: instantiating a model does not mean predicting its success: it merely allows the model to execute. Timings are not represented for clarity.*

Schema hierarchies are implemented as hierarchies of models *paired at run-time according to the patterns they match* – as our architecture relies exclusively on dynamic coupling, as opposed to static coupling (there are no explicit "wires" linking output streams to input streams).

In Replicode terms, a goal admits a member called "actor". It denotes the actor pursuing the goal. This means practically that since models react to any goal, they are in fact executed regardless of the actor pursuing the goal. In other words, if models can actually be adequate (pattern-matching-wise) to predict or (generate sub-goals) for another actor than the self (i.e. the system itself), then these models will be executed indifferently amongst any other with no need for specific additional control procedures. This characteristic of the executive allows the actual sharing of models between actors. Notice that nothing prevents additional models to be learned as specific requirements (with respect to the actor) to the shared models: this is the technical ground leveraged by the system to achieve differentiation between (the models of) actors (see section 6.4.3 below).

7.4 Implemented Architecture

7.4.1 Overview

The architecture of the prototype is a minimal instantiation of the general architecture presented in section 5. It is minimal in the sense that the Meta System is not evolvable, i.e. it does not learn. However, the Domain System does learn and expand – as we will see in section 6.6 below.

As mentioned earlier, the main goal of this implementation is to check the validity not only of key components but also of their inter-play. As it should be clear by now from the sections 5 and before, the capabilities of the architecture result from such an interplay. For example, the Correlator itself does not suffice for learning: learning is the result of the Model Acquisition, the Model Revision, the Attention Control and more indirectly, the Decision making processes. The sub-sections below will attempt to highlight the benefits of such a mutually-dependent organization of processes (i.e. dynamic coupling).

7.4.2 Domain System

The system has been implemented accordingly to the architectural template described in Figure 10 (exception: the Attention Control is not learned).

Interface with the environment. A set of programs have been developed to ensure the interfacing with I/O devices: these programs convert goals on commands into actual commands and re-inject the commands themselves in the main input group (efferent copies, required for the operation of schemas, as discussed in section 6.3).

The environment is sampled by the I/O devices at 5 Hz for the sub-domain of “physics”, i.e. information about geometry (resulting generally from the manipulation of objects by the avatars), and on-demand for the sub-domain of spoken/gestural interaction (that is, events belonging to this sub-domain are injected as they occur in the environment). Samples come in the form of generic markers - (mk.val entity attribute value) in Replicode terms. The “attribute” part is predefined and belongs to the ontology.

Attention Control. The Attention Control sub-system is composed of two distinct sets of code (a) a set of programs that propagate changes of saliency values across the graph of objects centered around a central object of interest and, (b) a set of programs that decide the centers of interest. These models have been built manually and are fixed – the Attention Control is not learned in this prototype. The function of these programs is to lower the saliency of objects which are *not*:

- simulations younger than an arbitrary time horizon (5s in our case) with regards to the injection time of the oldest hypothesis that triggered the simulation run;
- goals;
- predictions;
- notification of reductions;
- notifications of success or failure for goals and predictions.

Each of these objects (goals, predictions, etc.) constitute targets of attention – collectively they form the Area of Interest of the system. For each target, the Attention Control selects as potential sources of interest objects that are related to the target (by means of references, or pointers to or from the target at an arbitrary depth – 2 for the prototype). Then each of these sources of interest is abstracted, that is, any structure or object used or referenced by these sources of interest is replaced by variables – possibly shared by the targets and the related objects (other sources of interest). Objects sharing no common parts (either variables or direct references to entities in the ontology) with others are discarded – notice that time is one of the possible common parts: even if two objects do not share anything but are synchronous, they will be said to share their time variable and thus be retained as sources of interest.

Reaction. Multiple solutions may result from the injection of one goal: we have seen in section 6.3.2 that several models can feature as their right-side object similar instantiations of states or models (in other terms the branching out in the graph of models and states is equivalent to an OR operation in the case of weak requirements - or to an AND operation in the case of strong requirements). The system has thus to evaluate these multiple choices, some of these possibly conflicting with some others.

In a general manner, the system proceeds by simulating the outcome of a goal – technically, this means performing backward chaining on models and states and attaching simulation markers to the results. The depth of the simulation is restricted to an arbitrary time horizon (5s in our case) – this control is an operation performed by the Attention Control sub-system (as mentioned above). Goals are ranked by their intensity (technically, their saliency values) and their outcomes (sub-goals and achieved states) are traced back to goals using the reduction notification (telling basically, which model produced which outputs from which inputs). A conflict occurs when the simulated reductions produce (a) two states that negate each other or, (b) recursively along the path of the reduction notifications, when the goals that have triggered the reductions point to facts that negate each other, until the top-level goals are reached. In case of a conflict, the most salient goal among the conflicting goals is selected and injected (simulation ends and the actual operation begins). In the case of non-conflicting solutions for one goal, the system selects the solution that has been reached the quickest. In the case of non-conflicting solutions for multiple goals, the system selects the set of all the best (quickest) solutions for each goal. For each goal that has not been selected, the system lowers their saliency (so as to "deactivate" these goals) until either the selected goals succeeds or the simulation time allowance expires (5s timeout).

The activation values of the models are linked to their confidence values (see section 6.4.3 below): this results in selecting for execution the most reliable models, and delays the execution of the least. This is particularly of importance while running models in simulation mode, as this trims down the number of model candidates for achieving a simulated goal. Notice that the Replicode executive has been tuned so that it now presents salient inputs to active code (programs and models) in order of decreasing saliency, and in order of the code's decreasing activation. In other terms, the system has now the ability to process inputs by models in order of confidence (linked to activation, applies to models) and saliency (applies to inputs).

This constitutes rudimentary planning: it was not in the requirements for this prototype to investigate more sophisticated versions of planning, as the main goal was to validate the loop acquisition/reuse of skills. Programs for initiating simulations and controlling said simulations have been built manually.

7.4.3 Meta System

The purpose of the Meta System in the prototype is to control the expansion of the Domain System – i.e. to control the learning of models and states that augment the Domain System. The Meta System consists of an Attention Control (the Meta system is also a Reactive system and as such follows the same general architectural template as the Domain System), the Model Acquisition and the Model revision sub-systems.

Attention Control. This sub-system performs exactly like its counter-part in the Domain System: actually they are the same sub-system (they are shared) and can be so as the Attention Control described earlier is actually *domain-independent*.

Model Acquisition. The acquisition of models is a process that runs continuously: it is implemented as an instance of the Correlator. The latter takes its input sequences from the attention group of the Domain System and outputs models, ready to use. The Correlator in its first incarnation has been described elsewhere (Deliverable D11).

The Correlator is fed at run-time with target objects and their respective related objects - determined by the Attention Control (as discussed section 6.4.2). The Correlator spawns a thread for finding correlations between the target object and some of its related objects. This thread is actually a Replicode thread, which means that learning is completely inter-wined with all other activities of the system - like for example, making decisions, and eventually acting in the environment - and is thus performed concurrently and continuously. In other words, the system does not suspend its interactions with the environment while learning, nor does it postpone learning after acting. The Correlator proceeds by goal regression with a limited time horizon: for a given episode, when a model is found for satisfying the goal, its premises (or left-side term) is used as a sub-goal (i.e. a new target, and thus a new thread) and the *same episode* is used for finding models satisfying the sub-goal. Consequently, the "fusion" of learning and acting means that the system can make immediate use of the *full* episodes encountered during acting (or observation) for learning and does not have to observe again similar episodes when the time would come for learning how to reach sub-goals of models produced offline. Fusing action and learning actually constitutes a performance improvement and in turn, makes said fusion sustainable in term of computing time. This "virtuous circle" can actually be achieved if the Correlator (a) can learn from few episodes and, (b) can produce models quickly: these two characteristics have been identified as strong requirements for the design of the Correlator.

Apart from building states and models from observed events, the Model Acquisition process also eliminates equivalent states and equivalent models. This activity answers the need for limiting the proliferation of learned models and states – and thus also contributes to the realization of the “virtuous circle” discussed above. As mentioned earlier, pure syntactic code analysis is not enough to assert the equivalence of models, as these are context dependent, and also constitute contexts (requirements) for other models. This lead to the definition of equivalence as a dynamic property. Two models are equivalent if (a) their right-side object match each other (basically, they achieve similar goals, or, looking the other way around, they predict similar things) and, (b) their respective productions from matching inputs under the same contexts match each other. Notice that the aforementioned contexts designate *both* the requirements of the models *and* the models they are a requirement for. In other terms, it does not suffice that the models produce matching outputs from matching inputs: these outputs shall also trigger the (matching) instantiation of the models they are a requirement for. This definition is actually a variation on the definition of model analogy (given in section 4.5) in that it relies on the analysis of reduction chains. In a similar way, two states are equivalent if (a) they count the same number of objects and, (b) their respective instantiations match each other and what triggered said instantiations also match each other – recall that states are instantiated in three cases: (a) inputs match all their objects (straightforward pattern-matching), (b) some of their variables are bound during forward chaining (they constitute predictions) or, (c) some of their variables are bound during backward chaining (they constitute goals). The definition of equivalence comes with a syntactic prerequisite (noted (a) above) which is used to pair models (or states) that could, according to this prerequisite, constitute candidates for equivalence. The Model Acquisition process records all inputs that triggered the instantiation of models or states (including the notifications of the reductions) and re-enacts the reductions for the candidates in order to compare their outcome, accordingly to the dynamic part of the definition of equivalence (noted (b) and (c) above). This analysis is performed on the fly – i.e. as the system learns – and before releasing a new model (or state) in the system. The corresponding computational load incurred by the system is significantly lowered due to the following reasons: (a) inputs are heavily filtered by the Attention Control, (b) only inputs within a predefined time horizon are used, and (c) the Correlator learns quickly from few observations which means that targets are relatively short-lived. Notice also that equivalence detection is highly parallel and thus will benefit greatly from extra CPU cores.

As amply discussed, the Model Acquisition process produces models in an additive way, that is, models may constitute requirements for others. Weak requirements are options allowing the execution of models situated above in the hierarchy, whereas strong requirements must be avoided to allow the execution of the models above. The Model Acquisition process proceeds on objects abstracted by the Attention Control and by strictly complying to this rule could never identify particular cases, i.e. contexts where some general models acquired previously do not apply. This is the reason why the Model Acquisition process follows one extra rule: it is to detect the breaking of predictions (this is performed by making said prediction a target) and consequently add one strong requirement to the general model, while still performing as usual, that is in this case, to detect, possibly, a new outcome for the observed state which differs from the one predicted by the general model. The general case and the particular ones are kept in the system equally. During backward chaining, strong requirements are translated into states to be avoided and weak requirement as desired states. It turns out that if the system targets the outcome of a model more particular than another, then it will generate the sub-goal of reaching a state that will be more specific than the state that otherwise a more general model would have produce as the sub-goal.

Model Revision. The preliminary version of the Model Revision sub-system operates by evaluating (a) the performance of individual models and, (b) the performance of the system as a whole.

The performance of a model is defined as the success rate of the model. The success rate is the number of errors from the model's target state to the actual state divided by the life time of the model. A model's target state is to be understood relatively to the type of chaining that produced the target state (prediction – forward chaining - or goal – backward chaining). In principle, these errors are to be computed by progress monitoring models, using domain-dependent metrics. In the prototype, these metrics come in their simplest form, that is, pattern-matching (if the actual state matches the target state, no error is asserted, and vice versa). Confidence values and activation values for the models are changed accordingly to the success rate. Models are selected for scrutiny – i.e. their productions and instantiations become targets for the Model Acquisition sub-system - when (a) they are active and, (b) their confidence value is above an arbitrary fixed threshold. We have defined a scale for evaluating the performance of models along with a threshold of acceptability. Above the threshold, models are removed from the target list of the Model Acquisition sub-system. These models are added back when their performance crosses the threshold downward. Notice that a model presenting a high error rate is not necessarily a bad model. It just means that it must be executed in contexts (encoded as requirements) that have not been discovered yet. In a similar way, a model presenting a low error rate means that, *so far*, appropriate contexts for its execution have been found, but this does not entail the "goodness" of the model forever.

The performance of the system as a whole is determined by the measurement of the *progression* of the success rate of the targets managed by the Model Acquisition sub-system. A measurement falling under a predetermined threshold means that no improvement on the success rate of the target can be expected in the near future, and the target is removed from the Model Acquisition sub-system (and its corresponding thread is aborted).

The aforementioned measurements of performance (for both individual models and for the system itself) are performed by the Replicode executive.

7.5 Masterplan

This section gives the content of the Masterplan, organized in sub-sections for clarity: "physics" stands for the sub-domain of object manipulation, "speech" stands for the sub-domain of verbal and gestural communication, "goals" stands for the initial goals of the system and "control" gathers the programs and models to allow the experimenter to control the system. The content of the Master plan is given below in pseudo-code - the formal (compilable) Masterplan is given in Annex 1.

7.5.1 Physics

Ontology. It consists of the following class and entities:

- vec3: a class to encode three-dimensional quantities.
- essence: used to encode ontological properties (for example, (mk.val x essence "cube) means "x is a cube").
- part_of: used to encode sub-part relationships (for example a hand is a sub-part of an actor).
- position: the location of an object in space.
- color. the color of an object.
- attachment: the fact that an object seems attached to another (this is used as a visual feedback for grasping).

Commands. These are:

- grab_hand. Parameters: the hand, a deadline.
- release_hand. Parameters: the hand, a deadline.
- move_hand. Parameters: the hand, a target position, a deadline.

States. These are:

- hand_pos[H P T]:[(H position P)T,(H essence hand)T]: position of a hand.
- same_hand_pos[H X P T]:[(icst hand_pos_cst X P)T,(X position P)T]: an object has the same position as a hand.

Models. None have been needed in this section of the Masterplan (see section 6.6 below).

7.5.2 Speech

Ontology. It consists of the following class and entities:

- speech context: a class to encode that an actor (speaker) speaks to an other actor (listener). Instances of speech contexts are refined by markers that encode, respectively, that what is said maps to an action, that the action of a speech (verb) context has a target (object). The properties “action” and “action_target” are entities that belong to this part of the ontology. Qualifiers on action targets are picked from the ontology for “physics” presented above (for example, color, essence, position).

- speaking: used to encode the fact that an actor speaks a word.
- listening: used to encode the fact that an actor is a listener in a speech context.
- pointing: used to encode that an actor points to a location in space.
- knowing. Used to encode that an actor knows a fact.

Commands. These are:

- speak. Parameters: a word, a deadline.
- point_to. Parameters: a target position, a deadline.
- look_at. Parameters: a target position, a deadline.

States. The following are states needed for mapping incoming speech to goals on instantiated states and vice versa. They are:

- stand_col_ess_pos[X P E C T]:[(X color C)T,(X essence E)T,(X position P)T]: an object stands in a position, has some color and is of some class of object.
- stand_col_ess_pos_sln[X P E C T]:[(X color C)T,(X essence E)T,(X position P)T,(X most_salient nil)T]: an object stands at a position, has some color and is of some class of object; it is also the most salient object featuring these attributes.
- stand_pos_sln[X P T]:[(X position P)T,(X most_salient nil)T]: the most salient object stands in a position.
- taken_col_ess[H X P E C T]:[(X color C)T,(X essence E)T,(icst same_hand_pos H X P T)T]: an object with a color and of some class has been grabbed.
- taken_col_ess_sln[H X P E C T]:[(X color C)T,(X essence E)T,(X most_salient nil)T,(icst same_hand_pos H X P T)T]: an object with a color and of some class has been grabbed; it is also the most salient object featuring these attributes.
- taken_sln[H X P T]:[(X most_salient nil)T,(icst same_hand_pos H X P T)T]: the most salient object has been grabbed.

Models. None have been needed in this section of the Masterplan (see section 6.6 below).

Programs. Language parsing is implemented as a set of programs – each targeting one part of a sentence – that collectively assemble a speech context (as defined earlier). In the present state of the system, words are encoded in the programs that build the speech context: this information had to be given manually, as the system is as of today, incapable of learning words. A speech context translates eventually into a goal targeting the instantiation of a model or a state. For example the sentence “put a blue cube there” translates into “the speaker has the goal that the listener has the goal to have a blue cube happen at the specified location” - actually, a goal about a goal about an instance of the state stand_col_ess_pos (mentioned in the list above).

Language synthesis is performed by another set of programs that basically work in reverse when compared to the parsing programs: they translate a goal on an instance of a model or a state into a stream of speech commands, each speaking one word.

These two sets of programs are given in Annex 1.

7.5.3 Goals

Ontology. It consists of the following entities:

- role: used to encode the role of an actor.
- interviewee: this denotes a particular value assigned to the role of an actor.
- interviewer: this denotes a particular value assigned to the role of an actor.

- top_level: the top-level goal of the system. This abstract goal is referred to by a model generating the actual top-level goal of the system (see models below).

States. These are:

- interviewer_thanks_interviewee[IR IE T]:[(speech_context IR IE) (IR speaking "thanks you")T,(IR role interviewer)T,(IE role interviewee)T]: the interviewer thanks the interviewee.

Models. These are:

- self_speak[S T1 T0]:[(cmd speak S T1)T0-> (self speaking S)T1]: the outcome of the system speaking.
- top_level_model[A T]:[(icst interviewer_thanks_interviewee A T)T -> (top_level)T]: maps the abstract top-level goal to an actual goal (the interviewer – whoever it is – thanks). The system pursues the goal of having the interviewer thanking the interviewee (regardless of who is actually the interviewer).

Programs. This section of the Masterplan contains programs. They are:

- a program for killing the top level goal when satisfied (to avoid the system pursuing its goal ad infinitum: this is needed since the top-level goal is not produced by any model and thus, is not monitored).
- a program that assumes that if a speaker has the goal that a listener has a goal about some state, then the listener actually has the goal in question.
- a program that confirms the reality of the production of the program above. Precisely, this program asserts that the actor *had* a goal about a fact f when f actually happens.

The actual code of these programs is given in Annex 1.

7.5.4 Control

Ontology. It consists of one entity:

- programmer: this denotes the programmer in control of the experiments. The programmer issues commands to the system using speech. This entity is used to differentiate commands issued by an actor in the scene from direct orders coming from the programmer (for example: “switch to interviewer”).

States. None are required.

Models. None are required.

Programs. This section of the Masterplan makes use of the following programs:

- a program for switching roles between the interviewee and the interviewer.
- a program for setting the system with no role at all (used for passive observation).
- two programs for setting the content of the task list to the machine-interviewer.

The actual code of these programs is given in Annex 1.

7.6 Preliminary Results

This section presents the results of the experiments conducted with the prototype, organized in sessions, following the list of requirements defined in section 6.2.

All the learned models and states are given in pseudo-code, with human-readable names instead of the machine-generated code and names. The compilable form for these states and models is given in Annex 1.

7.6.1 Session 1 – Motor Babbling

Setup. In the “motor babbling” session the system was acting alone, issuing randomly goals on its own effectors to learn the outcome of the commands that resulted. The incentives for doing so were hand-crafted programs that have been purged from the memory after the session was over.

Learned Models. These models are either the direct outcome of the execution of a command, or requirements on the former models, or temporal consequences between szazes (like the model called “link”):

- grab[H X T1 T0]:[(grab_hand H T1)T0 -> (H attachment X)T1]
- req_grab[H X P T T0 T1]:[(icst same_hand_pos H X P T0)T -> (imdl grab H X T1 T)T]
- release[H X T1 T0]:[(release_hand H T1)T0 -> |(H attachment X)T1]
- req_release[H X T1 T]:[(H attachment X)T -> (imdl release H X T1 T)T]
- move[H P T1 T0]:[(move_hand H P T1)T0 -> (icst hand_pos H P T1)T1]
- link[H X P T1 T]:[(H attachment X)T -> (icst same_hand_pos H X P T1)T1]

In addition to these models two states have been learned during this session: these are the states mentioned in the “physics” section of Masterplan. This begs the question: why are these states listed in the initial Masterplan, in other words, why are they considered innate states? The reason is as follows. These states have been learned and named by the system but the same states are needed (indirectly) by the programs parsing speech, and these programs are hand-coded. We had no choice but re-coding these learned models in order to give them a name that can be referenced by the programs in question. See section 6.6.2.

7.6.2 Session 2 – Observation of an Interview

Setup. The system is set with no assigned role but still pursues the goal of the interviewer thanking the interviewee. This is the main drive for learning during this session. Recall that models and states can be actor-agnostic (the “actor” member of a goal can be abstracted).

Learned Models. These models are:

- interviewee_complies[IR IE X T1 T0]:[(goal X IE)T0 -> (icst interviewer_thanks_interviewee IR IE T1)T1]: when the interviewee pursues a goal X, the interviewer thanks him; this model has a requirement (below).
- generic_task[IR IE X T1 T0]:[(goal (goal X IE) IR)T0 -> (icst interviewee_complies IR IE X T1 T0)T1]: this is a requirement on the model above. It states that the model above will be instantiated when the interviewer has the goal of having the interviewee having the goal X.

The learning of the two models above demonstrate the “imitation” of the goal of the interviewee. Actually, this vocabulary is misleading: “imitation” conveys the false impression of “mimicking”. This is not the case at all: the system observes that pursuing the same goal as the interviewee does will bring success to his own goal (being thanked by the interviewer). We would rather talk of “goal-driven learning by observation”.

In addition to these models the system has also learned all the states listed in the section “speech” of the Masterplan. These models had to be re-coded by hand to allow the manual construction of speech parsing programs as discussed in the previous section). It is these states that actually need the states from the section “physics” of the Masterplan.

Learning words and grammar has not been listed as an objective of this project. However, it turns out that the impossibility to learn these is – in the case of speech - hampering the capacity of the system to learn without the programmer's intervention. In that respect we have started to investigate ways to alleviate this drawback ([10]).

7.6.3 Session 3 – System as the Interviewee

Setup. The system is assigned the role of the interviewee. It has previously learned to comply to the commands issued by the interviewer, and actually did so.

Learned Models. No new models or states have been learned during this session, as no new events were occurring that did not already in the previous session.

7.6.4 Session 4 – system as the Interviewer

Setup. The system is assigned the role of the interviewer. The system is instructed to have the interviewee perform tasks from a hand-coded list. This task list had to be hand-coded since no event during any of the sessions have indicated any reason for the list being what it was. In other therm, the system, being goal-driven, lacked a goal and means to assemble such a task list. This can find a simple remedy in the form of another model in the Masterplan that instructs the system to mimic the doings of the interviewer in lack of any good reason.

Learned Models. No new models or states have been learned during this session, as no new events were occurring that did not already in the previous three sessions.

7.7 Summary

What is the ratio between the models (and states) that have been learned and the total number of models (and states)? It depends on how one counts. If we ignore the re-coding of some learned models (see sections 6.6.2 and 6.6.1) then we have 5 innate constructs (states and models) and 16 learned constructs – roughly 76%. If however we take the re-coding into account we have: 13 innate for 8 learned constructs – roughly 38%. So, it appears that – when parsing human speech is required – the benefits we expected for the engineering of such systems are greatly diminished, at least commensurately to the number of the constructs referred to by spoken interactions. Otherwise – the linguistic sub-domain being ignored – our approach seems to meet our expectations with regards to the learning and integration capabilities.

On other less contingent fronts, the development of the prototype has been beneficial for addressing satisfactorily all of the key objectives listed in section 6.1. In a nutshell, the prototype has demonstrated the feasibility and tractability of an architecture that generates and revises dynamically a significant part of its own code from observation, while displaying good response times. Scalability has not been measured yet, but given the minimal computing resources actually needed by the prototype, we have gained more confidence that the system is likely to scale reasonably well.

This work constitutes a major milestone of the project as this prototype can be considered a successful proof of concept of the architecture design – within the envelope defined by its inherent limitations. The demonstrators planned for the end of the project will be built – by and large – on the firm ground that this prototype has established.

8 Final Architecture

Stemming from both our architectural principles and the implementation of the prototype, the architecture has reached its final incarnation (with regards to the time-frame of the project): essentially, what distinguishes the final architecture from the prototype is its implementation, which underwent significant expansion, optimization and generalization. This section presents the final architecture - called AERA - at two levels of description: functional and structural.

As mentioned earlier, the architecture is entirely domain independent: its plunging in the domain of socio-communication is presented in the WP6 deliverable D16. D16 also offers a detailed presentation of the methodological aspect of our approach – bootstrapping from a Masterplan followed by further expansion based on experience - both in principle and in the target domain. The results of the evaluation of the final system are also presented in D16.

The description of how imitation learning (imitation of goals) stems from the low-level processes described thereafter is given in the WP5 deliverable D21.

8.1 Overview

8.1.1 Improvements over the Prototype

The prototype has been used as a scaffold for both the design and the implementation of the system. The main differences between the prototype and the final architecture are:

- the Replicode syntax has been reworked to unify the encoding of models and programs;

- the high-level reasoning engine in the Replicode executive has been re-implemented almost entirely to increase its performance;
- the Replicode logic has been improved to allow the definition of time values as intervals (see design principle 5, section 7.1.3);
- backward chaining has been augmented to produce assumptions. An assumption results from abduction and is encoded as a fact with a confidence value computed exactly as for goals (goals are produced by the same abduction process);
- drives have been introduced (see section 7.2.8);
- the Replicode executive now produces periodic measures of its performance (see section 7.2.9);
- simulation was partially implemented at the level of Replicode constructs. This is not the case anymore: simulation is now completely integrated in the Replicode executive (for performance reasons);
- the process for model acquisition has been reworked. The episodes models are built from (see section 7.2.3) are now analyzed in a more sophisticated way: data produced by existing models are removed from the episodes, which prevents – by design – the production of equivalent models, and thus, eliminates the need to check for duplicates; this results in a significant improvement of the system's performance;
- compaction has been added;

8.1.2 AERA

AERA stands for Auto-catalytic Endogenous Reflective Architecture. In [3] we have adopted a stringent definition for autonomy, i.e. an autonomous system is operationally and semantically closed. In our context, auto-catalysis refers directly to the operational closure, that is the ability for a system to expand and modify its own internal agency by means of the operation of said agency (its structure). Reflectivity refers to the semantic closure, i.e. the ability for a system to control the reorganization of its agency. An AREA-based system is also endogenous in the sense that it is (a) self-maintained and, (b) originates from itself – the processes implementing the two aforementioned closures are internal and cannot be modified by anyone else but the system itself.

8.1.3 Summary of the AERA Design Principles

Design Principle 1: ***"Holistic Design"***. The desired operation of the system results indirectly from the inter-operation of a multitude of general-purpose underlying processes. In other words, there is no component called "learning" or "planner" and so on. Instead, learning and planning are emergent processes that result from the same set of system-wide functions. More over, high-level processes (like planning and learning) influence each other (positively and also negatively): they are dynamically coupled, as they both result from the execution of the same knowledge - the very core of the system, its models.

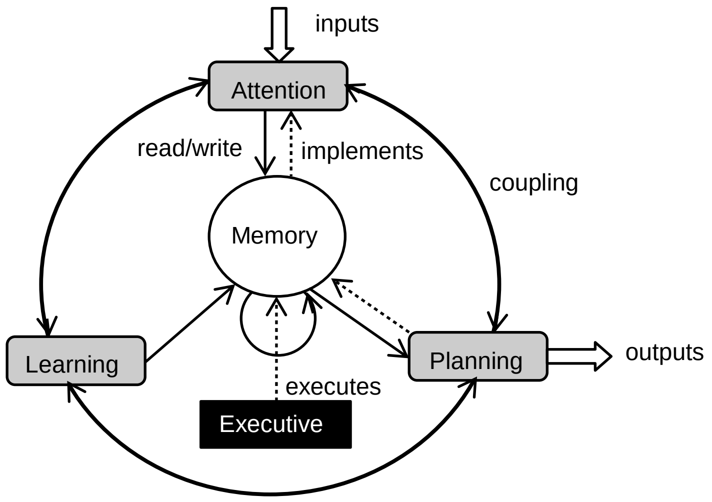


Figure 15 – Holistic Design

Three main cognitive processes – attention. learning and planning – are (partially) implemented and controlled by models which constitute the evolvable part of the system (the fixed part being the Replicode executive). These processes are indirectly and dynamically coupled through the memory of the system as models are added, deleted, activated or phased out, depending on the context and the goals pursued by the system.

In addition to observed states, assumptions, goals and predictions, the memory contains executable code (models) and as such, constitutes an active part of the system: it is actually the very core of a model-based and model-driven system. The memory is responsible for most of the computation occurring in the system.

Design Principle 2*: Reflectivity.* A system must know what it is doing, when and at what cost. By enforcing explicit traces of the system's operation allows the building of models of said operation, which is needed for self-control (also called meta-control). In that respect, the architecture shall be applicable to itself, i.e. a control system for the system shall be implementable the same way the system is in the domain. In other words, this principle must be followed if one wants to implement an Integrated Cognitive Control ([8]).

Design Principle 3*: Have all operations controllable in a uniform way using mechanisms as simple as possible.* More elaborate control schemes shall be learned by a control system, according to the second principle. To keep things simple eases the learning of early basic control models that in turn constitute the base for more complex ones. In other words, too complex innate control mechanisms impose too steep a learning curve for the system.

Design Principle 4*: Keep things short and uniform.* This means that (a) the granularity of the encoding of knowledge shall be low and, (b) all primitive operations of the system shall focus on one task and take as little time as possible, keeping in mind that higher-level operations result from the coupling of a multitude of said primitive operations. This principle aims at preserving plasticity – the capability of implementing small, incremental changes in the system: this is one of the key requirements for the architecture as it underpins our entire research avenue.

Design Principle 5*: Account for time at all stages of computation and at all scales* - from the scale of an individual operation (e.g. performing a reduction) to the scale of a collective operation (e.g. achieving a goal). This is an essential requirement for a system that (a) has to perform in the real world and, (b) has to model its own operation with regards to its expenditure of resources. Additionally, time values shall be considered as intervals to encode the variable precisions and accuracies to be expected in the real world: for example, sensors are not always performing at fixed frame rates and therefore the ability to model their operation is critical to ensure the reliable operation of their controllers and the models that depends on their input. Also, the precision for goals and predictions may vary considerably depending on both their time horizons and semantics.

8.2 Functional Description

8.2.1 Overview

The architecture is composed of low-level, system-wide functions that interoperate to implement the higher-level cognitive processes mentioned earlier, in a bottom-up fashion.

All the processes described below are entirely domain-dependent.

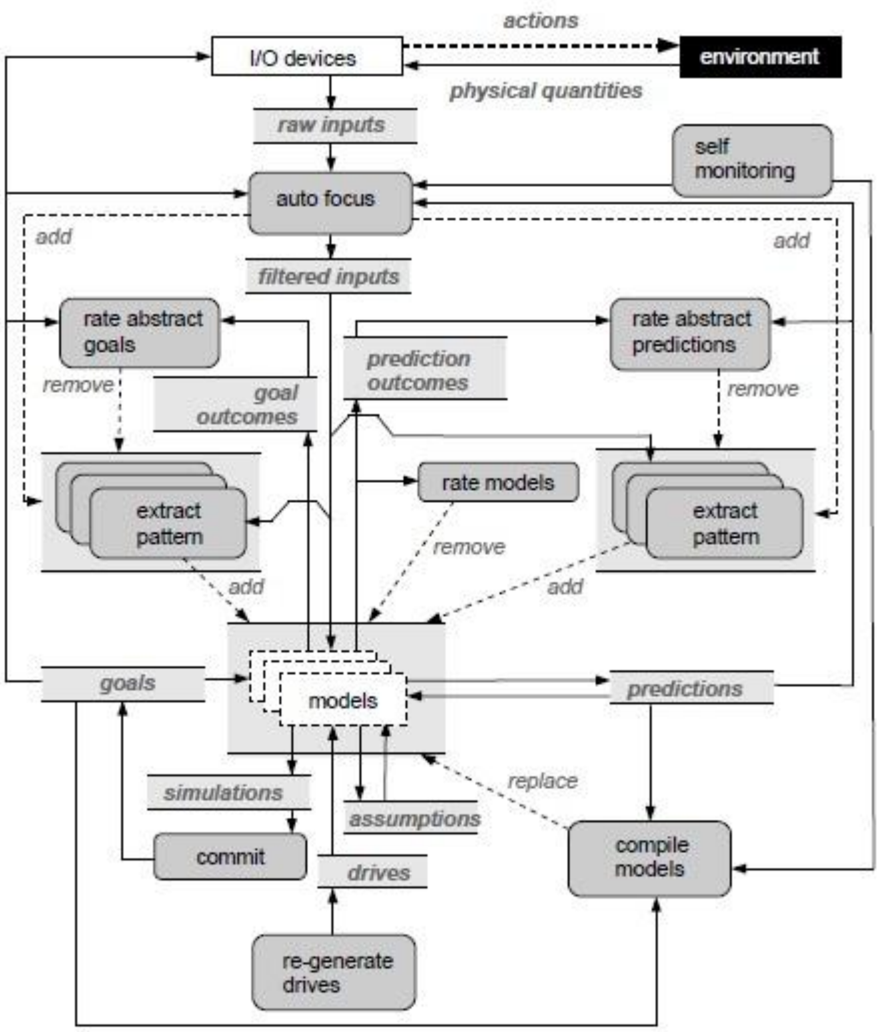
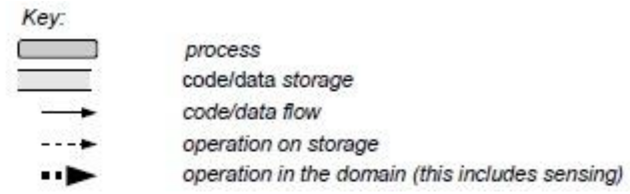


Figure 16 – Low-Level Processes



See text for details.

N.B.: control signals (i.e. data and emitting processes that control the activation of models and saliency of data) are not depicted for clarity.

8.2.2 Auto Focus

The process called "auto-focus" is responsible for directing the system's attention with regards to its internal inputs. These inputs are goals, predictions, goal successes, prediction failures and reflective inputs, i.e. traces of the execution of models and performance reports (see section 7.2.9)

The auto-focus acts on the system in the following ways:

- it filters the raw inputs: inputs that are irrelevant to goals and predictions are discarded.

Relevancy is defined as sharing at least one value with at least one goal or one prediction.

 for each new goal or prediction, it generates a pattern extractor whose task is to find explanations for the success of a goal or the failure of a prediction (see section 7.2.3).

8.2.3 Pattern Extraction

A pattern extractor is a process that is generated dynamically upon the creation of a goal or a prediction. Its main activity is to produce models, i.e. explanations for the success of a goal or the failure of a prediction: these are signals that trigger the building of models.

One Targeted Pattern Extractor is responsible for attempting to explain either the success of one given goal, or the failure of one given prediction. Said goal or prediction is called the TPX's target. Under our assumption of insufficient knowledge, explaining is more akin to guessing rather than to proving, and guesses are based on the heuristic “time precedence indicates causality”.

A TPX processes a buffer of inputs (see figure 7 below) that share at least one value with its target. This buffer is time-bound, i.e. inputs are discarded after a time limit called the time-horizon, which is domain-dependent and represented as a system-wide parameter (for example, in the domain of socio-communication we set this value to 5s which roughly matches the attention of a human). When the target goal succeeds or when the target prediction fails, the TPX is signaled resulting in an attempt to produce a model from the most relevant input in the buffer.

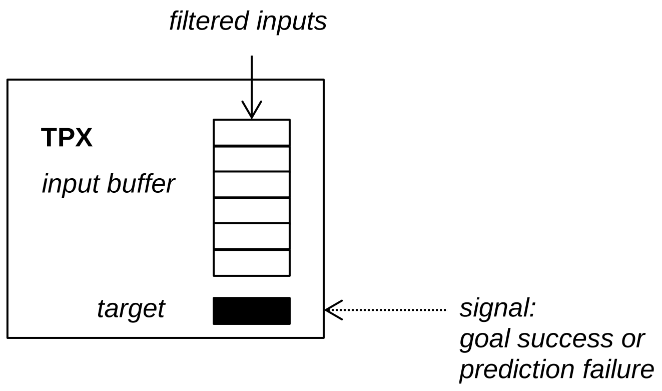


Figure 17 – Targeted Pattern Extractor

The filtered inputs are produced by the Attention Control process (called Auto-Focus in AERA). Signaling results from the catching of reflective inputs produced by the executive.

In order to find the relevant input, a TPX proceeds as follows (we assume the target is a goal, knowing that the procedure is similar for predictions):

1 if one input is a trace of the execution of one model that solved the goal, abort – a trace of model execution is called a reduction marker and contains (a) a reference to the executed model, (b) a reference to its input and (c) a reference to its production. This step ensures that no two identical models can be produced;

2 remove any inputs that triggered any model execution;

3 remove any inputs that were assembled in composite states;

4 from what remains, choose the younger input as the left-hand term of a new model (the right-hand term being the target itself).

Shall the target be a prediction, step 1 would become:

1 if one input is the trace of the execution of one model that predicted the failure, abort;

The construction of new models is performed as follows (assuming the target is a goal):

1 if the input chosen as the left-hand term of the new model is synchronized with other inputs, then all these inputs are assembled into one new composite state: this new state is chosen as the left-hand term of the model; Notice that new states are identified when they are parts needed for the models being built – instead of resulting from blind temporal correlation;

2 the chosen left-hand term is abstracted and so is the target: the two terms of the model (let's call it M0) now share some variables;

3 if some variables in the right-hand term are not present in the left-hand term, then the next input is chosen as the left-hand term of a new model M1 whose right-hand term is an instance of M0: the variable missing in M0 is passed from M1 to M0 as a parameter; when executed, M1 allows the execution of M0;

4 repeat 3 as long as necessary; if the buffer is exhausted, delete all models found so far and abort.

If the target was a prediction, then the signaling event would be a failure. Technically this means that M1 mentioned above would have its right-hand term being the failure of the execution of M0. In other words, the execution of M1 would inhibit the execution of M0.

In models, pattern-matching is guarded by pre-conditions (called guards): these are arithmetical/logical operations to be executed on the values held by an input and, mapped to the variables in the model. We need to induce these guards from the inputs – we limit ourselves to the induction of invertible linear functions of the form y=a*x+b where a, b, x and y are variables in the model. Such induction relies on a heuristic that ascribes some semantics to some of the variables found in models. The algorithm proceeds as follows:

1 induction is attempted only in the case of a model explaining the change of a transient, continuous value (for example, the position of an object in space); transient values are identified as such when found among a stream of regular samples of the same quantity (for example, the successive values of the aforementioned object position, stored in the TPX's buffer);

2 right-side time variables are then assumed to follow their right-side counterpart by a constant sampling period estimated as the difference between the timestamps of two successive samples of the same quantity;

3 then for each left-side numerical variable x1 (occurring at t1), try to find a counterpart x0 (i.e. a prior having occurred at t0) in the right-side term and in the model's parameters;

4 if successful, find a match for the variable c - as in the expression x1=x0+(t1-t0)*c – in the right-side and parameters of the model; otherwise abort and delete the model;

5 if successful, build two guards (one is the expression above, the other its inverted form) and insert them in the model; otherwise abort and delete the model.

The method used to dynamically generate new models may give the reader the impression that these models are completely abstracted and that, having been built on the ground of *one* evidence – say, e0 -, they will be used by the system to process *any* evidence matching the structure of e0. It is not so and this deserves furthermore an explanation.

We define the *level of abstraction* of a model by the precision of the constraints it applies on the values matching one of its terms (right- or left-side). Let's consider for example one term of a model, expressed as a structure S containing variables x_i. When an input I matches S, the relevant values v_i contained in a are mapped to the x_i if and only if the v_i are within a certain range: the wider the range the more abstract the model.

Each model maintains a range for each dimension of the observables – a dimension is encoded a s a class of relation between values (an object of the class ont in Replicode) – examples of such dimensions are position, essence, belonging_to. Depending on the semantics of a dimension, the associated range is either discrete (in {0,1}) or continuous (in [0,1]).

At construction time, a model initializes its ranges with the values observed in the input its right-side term has been derived from. Each time an input matches the structure of the right-side term and is within the known ranges, the model issues a prediction, and at due time, an assessment of the prediction outcome is performed and the model rated accordingly. Now, when an input matches the structure of the right-side term but is out of the ranges, then the model will produce a prediction *silently*, that is without publishing it: if the prediction fails, the model is not rated negatively as the prediction was a mere trial. If on the contrary the prediction succeeds, then (a) the model is rated positively and, (b) the ranges are enlarged to accommodate the values held by the input in question.

In other words, the more the model performs well on new evidences, the more abstract it gets. Conversely, if no further evidences produce good predictions, then the model remains constrained by its initial input, i.e. encodes a particular case instead of more a general one.

TPX can only generate models if the terms on both sides share at least one variable. It follows that TPX cannot generate models encoding procedures such as “alarm sound at t0 -> flashing light at t1” since the two terms share no variable. We developed another kind of Pattern Extractor called Cross Pattern Extractor (XPX). An XPX works exactly as a TPX except that the inputs in its buffer are not constrained by the requirement of sharing values with the target. Instead, the XPX buffer is defined as the intersection of all TPX buffers (see figure below).

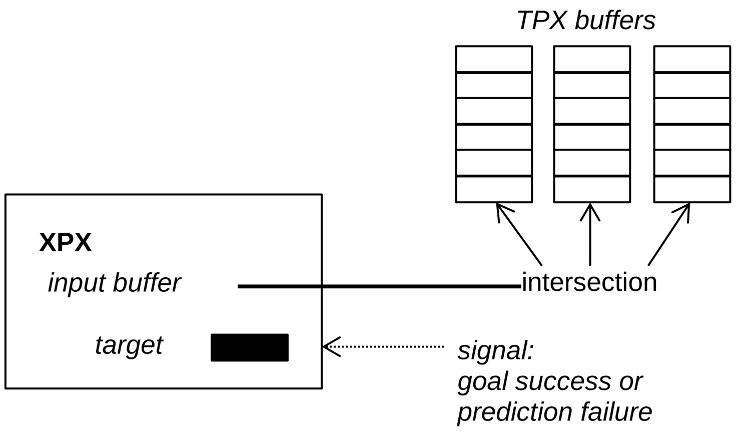


Figure 18 – Cross Pattern Extractor

The input buffer of an XPX results from the time-ordered shuffling of common inputs found in the TPX' buffers. A dedicated time-horizon is defined system-wide for constraining the XPX's buffers.

The rationale for the XPX design is as follows:

- value-sharing is too constraining to allow capturing causality between structurally unrelated events, but
- removing constraints on inputs makes the Pattern Extractor blind, i.e. calls for more trials and errors which are time consuming (affects negatively the learning speed) and resource consuming (the computational power needed to produce these trials and to evaluate them), therefore
- we need to relax the constraints on the buffers but reasonably so, and, in fine
- we define the admissible inputs for an XPX as having to be relevant to the *global current* activity of the system, i.e. to the set of targets it currently pursues, which translates technically into the intersection of the TPX buffers.

8.2.4 Rating Models

To rate models is to keep track of their performance. A model's performance is the number of times its predictions came true, divided by the number of times the model produced a prediction. In other words, the performance of a model is its success rate with regards to predictions.

The success rate of a model is interpreted by the executive as its activation value, which means that models with low success rates are likely to fall under the activation threshold of the group the models live in, and thus, be deactivated.

We distinguish two states for a model: weak and strong:

- a model is weak when the number of time it produced predictions is below a system-wide threshold. When the success rate of a model gets below the activation threshold mentioned above, the model is deactivated but still receives inputs silently: this is meant to give the model a chance of improving. If the success rate continues to decrease, after crossing a threshold, the model is deleted.

- a model is said to be strong when its success rate is high enough (i.e. above a system-wide threshold) while a significant number of evidences (i.e. an evidence count above another system-wide threshold). The success rate of a strong model getting under the activation threshold is interpreted as a context switch: the system knows the model is usually a good performer and assumes that the failures of said models are due to the inadequacy of the model in the current context of the domain. The model is deactivated but not rated negatively; instead it still receives inputs silently. In case the model starts to perform well again, the system will assume the domain has switched back to a context the model is attuned to, and will re-activate the model.

8.2.5 Rating Abstract Goals and Predictions

Besides evaluating the performance of its individual components (rating models), the system has to evaluate its own global performance. Here, by evaluating the system's global performance we mean evaluating how good the system is at achieving a class of goals and how reliable the system is with regards to a class of predictions. Classes of goals and predictions are actually abstractions of goals and predictions, that is, goals and predictions where values have been replaced by variables – in other words, patterns of goals and predictions.

The assessment of the system's global performance proceeds as follows:

- the system maintains a dictionary of abstract goals and predictions: each time a new goal / prediction is generated, it is abstracted and added to the dictionary if not already present;
- each time a goal or prediction outcome (that is success or failure) is received, the success rate of its corresponding abstraction is updated: the success rate is the number of successes divided by the number of times an instance of the abstract goal / prediction has been produced;
- if the success rate of an abstract goal / prediction gets above a certain threshold, any pattern extractor targeting an instance of said goal / prediction will be removed from the system. More accurately, this control is exerted upon the success rate being high enough and derivative of the success rate being under another threshold: this indicates that the system makes no significant progress in learning how to achieve a goal or to make reliable predictions.

8.2.6 Simulation and Commitment

The system has to make meaningful decisions in the following cases:

- several goals defeat each other,
- several possibilities are available for achieving one goal.

Simulation is a global process that, for each goal currently pursued by the system, evaluates its consequences with regards to other goals. Simulation is akin to a breadth-first parallel search in model- and state-space.

When a goal is committed to, the system, via backward chaining, produces all possible sub-goals, which in turn triggers the production of sub-sub-goals and so on. Simulation is a process that performs as follows:

- at some point in time (a system-wide parameter called simulation time horizon) the generation of sub-goals is frozen and simulated predictions of the success of the sub-goals generated so far are injected in the system;
- the simulated predictions trigger other simulated predictions (forward chaining), some of which indicating the possible success or failure of other goals;
- for each committed goal, the system gathers the potential outcomes for all other goals;
- for each committed goal, the system eliminates the sub-goals that triggered the failure of more important goals, and from what remains, selects the best sub-goals and commits. Goal importance is defined as the product of its saliency (i.e. its chances of success) and its urgency (i.e. its deadline).

The simulation process is described in full details in [1].

8.2.7 Compaction

The purpose of compaction is to increase the performance of the system by replacing some of its models by C++ code generated on the fly, i.e. to compile part of the system's memory, at the cost of reducing the system's plasticity. Compaction results form learning patterns of coupling between models, and as such, results from the identification of internal patterns of activity. Compaction belongs naturally to WP2 and the full details of its operation are given in D17.

Compaction targets two patterns of internal activity:

- goal production: the system identifies predictable sequences of goals producing sub-goals involving a fixed set of underlying models. In other words, the system identifies stable abduction paths. The models involved are removed from the system and replaced by one compiled plan generator – a plan being defined as a timely sequence of goal production.
- prediction production: the system identifies predictable sequences of predictions entailing other predictions involving a fixed set of models. In other words, the system identifies stable deduction paths. The models involved are removed from the system and replaced by one compiled prediction generator – a component that generates a timely sequence of predictions.

By stable path we mean a set of strong models, constant over time, that produces the same outputs given the same inputs. Such paths are learned by the system, offline – intuitively, during "sleep". The system builds models of the inputs/outputs of each model, based on the reflective data generated by the system itself while acting in the domain.

In both cases, shall a model be shared between several paths (deductive or abductive), said model is left in place to avoid breaking paths that are not compiled. More over, models removed following compaction are not destroyed: since paths are context dependent, shall said context switch to another one where the compiled paths are not stable anymore, then the initial models are re-injected and the compiled code removed. This is meant to avoid for the system to have to learn again individual models that it already knows.

8.2.8 Drive Re-Generation

A drive is a goal targeting a state that is *not* observable. For example, such a non-observable state could be "stay in good condition to ensure a normal operation": no input like "self is in good shape" will ever be received from the environment. It constitutes a top-level goal for which initial hand-crafted models are given to the system (in the Masterplan). Each of these models states a possible way to satisfy (or dis-satisfy) the drive. In our example, such a possible way could be expressed as "maintain a sufficient level of energy", which is actually an observable state.

Once a drive is satisfied, the system re-generates it within a time-frame: this time-frame is domain dependent as it depends on the semantics of the drive, which depends on the domain. The purpose of the drive re-generation is two-fold:

- it ensures a perpetual and periodic "pressure" on the system backward chaining activity. In other words, when sub-goals fail, their super-goals will be injected again, as a result of the renewed injection of drives, thus giving the system another chance to achieve the sub-goals in question – assuming the context in the domain allows for it;
- its ensure that the continuous "needs" of the system are given attention to. For example and as mentioned earlier, such a "need" can be "maintain a sufficient level of energy", in the context of a continuous depletion of said "energy".

Technically, the system contains a set of (Replicode) programs whose responsibility is to re-generate the drives upon success, failure or absence thereof at arbitrary time-horizons. These programs belongs to the Masterplan.

8.2.9 Self-Monitoring

Self-monitoring is the production by the system of data describing the computation of the system. These data are generated automatically by the executive; they are:

- notifications of the productions of models,
- notifications of the data used for assembling composite states,
- periodic samples of the average time consumed by reduction tasks (the lower the better),

- periodic samples of the average time ahead of time-bound tasks (the higher the better).

Such data constitute internal inputs for the system itself but also external inputs for meta-systems, i.e. systems in charge of controlling the former one. For example, compaction is a process carried out by such a meta-system, and makes use of the notifications mentioned in the list above.

Periodic samples of computation times and time ahead possibly indicate congestion, i.e. insufficient resources with regards to the tasks at hand. It is the responsibility of a meta-system to take contingency measures. These are:

- increase the thresholds for model activation: fewer models will be executed, as the standards for model performance are higher;
- increase the saliency threshold in the groups the models live in. Goals, predictions and assumptions will be more likely to fall under said threshold, and this limits the depth of the chaining;
- reduce the simulation time horizon: the system will spend less resource evaluating the consequences of its actions since the search is less deep;
- deactivate drives of low priority: this is achieved by deactivating the programs that re-generate them. It is up to the developers to define priorities for the drives (as they are domain-dependent). Priorities are defined as the activation values of the re-generating programs;
- increase the threshold that controls the rating of abstract goals and predictions (section 7.2.5): this lowers the expectations on the system's capabilities to achieve goals and predict correctly. As a result, fewer models will be learned – since the system's performance will be deemed “good enough” - at the cost of having less reliable models.

8.2.10 Mapping of the Architectural Template to Functions

The Architectural Template (introduced section 5.1) is an abstraction of the functional view of the system. This section describes how the functions detailed above reify the four processes of the template.

Model Acquisition – Model acquisition is performed by the pattern extraction processes. Model acquisition is controlled by the rating of abstract goal and predictions.

Model Revision – Model revision stems from the rating of models. This rating is triggered by the injection of the outcomes of the predictions produced by models.

Attention Control – The control of the system's attention is two-fold. First, the attention of the system in the domain is driven by models, as these control the sensors – one may think of the visual attention for example, which controls the visual sensors. Second, the selection of relevant inputs from all the inputs provided by the sensors is performed by the auto-focus process and is driven by the system's current activities – the goals it pursues and the predictions it produced.

Compaction -Compaction results from learning of patterns of model coupling. Compaction is actually implemented by models that are learned from reflective data produced by the system while interacting in the domain. Compaction is implemented by a separate system, according to the principle of Integrated Cognitive Control.

8.2.11 Cognition

This section describes how the four main cognitive activities (Memory, Attention, Learning and Planning) result from the inter-operation of the processes described in the functional view of the system.

Memory – The memory of the system holds several kinds of data: facts (filtered inputs), goals, predictions, assumptions, notifications (reflective data), simulations and models – the latter constitute a procedural memory. Episodes (from which models are built) are stored in the buffers of each pattern extractor. Facts, goals, etc. are stored as long as they hold – this is determined by the upper bound of the time interval specified for each fact. Models are stored until their success rate drops under a threshold, at which point they are deleted. The system is data-driven and thus there is no “recalling” activity. Instead, data that hold are presented by the executive to models either once (transient data) or repeatedly (persistent data) until either their lifetime (resilience) expires or they are disproved by new data observed in the environment.

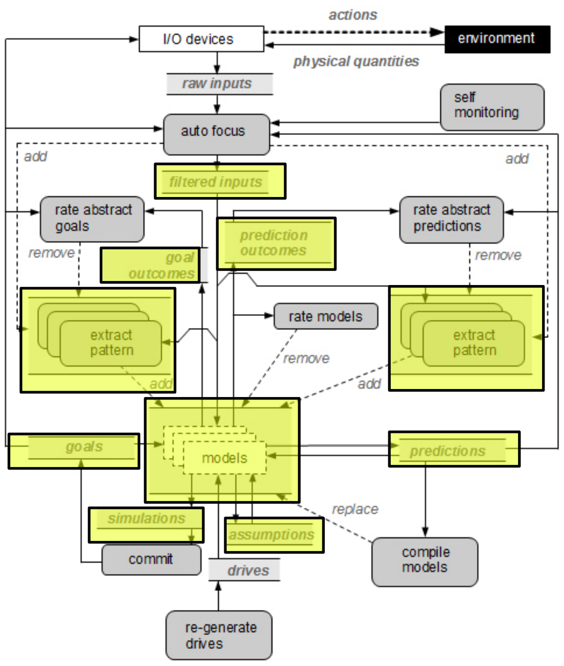


Figure 19 – Memory

Attention – As mentioned in the previous section, attention is two-fold: it is implemented by models (attention in the domain) and by the auto-focus (attention with regards to the system's current activity).

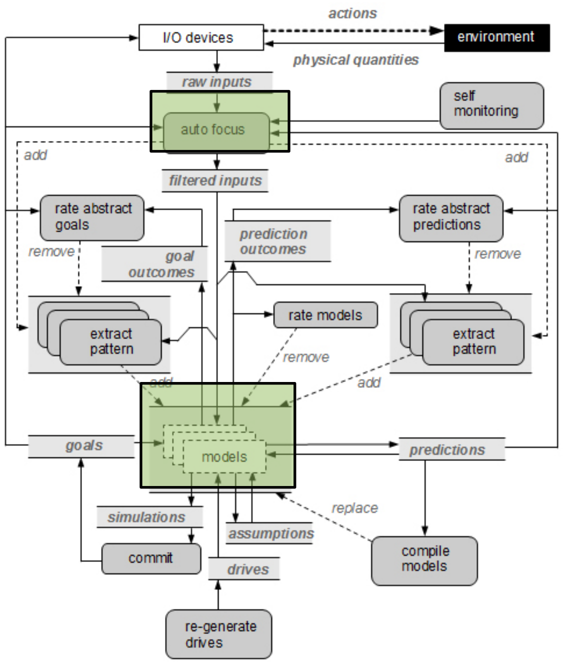


Figure 20 – Attention

Learning – Learning stems from the acquisition of models and the revision thereof. It is controlled on the basis of the evaluation of the system's global performance, i.e. its ability to achieve goals and to predict correctly. Both the acquisition and the revision of models are fixed processes.

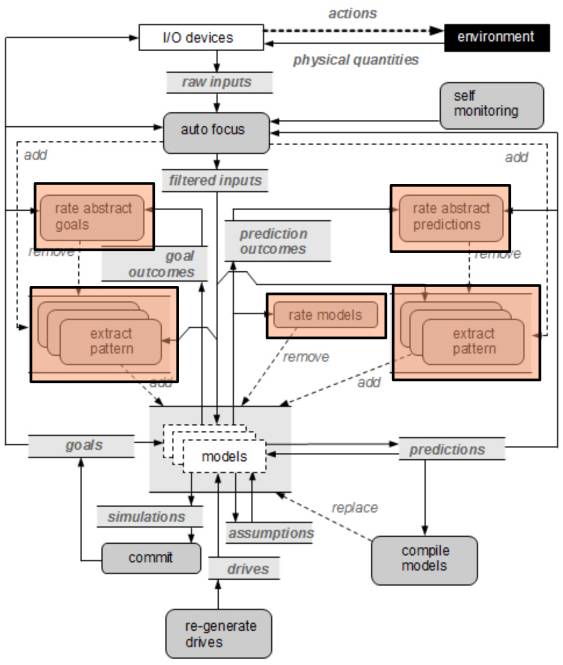


Figure 21 – Learning

Planning – Planning is the generation (by models) of sequences of goals to reach a target state in the domain. It results from the commitment to one or several sub-goals, given a super-goal. This commitment is performed by the analysis of simulated impacts on both the environment and the system itself. This analysis is a fixed process.

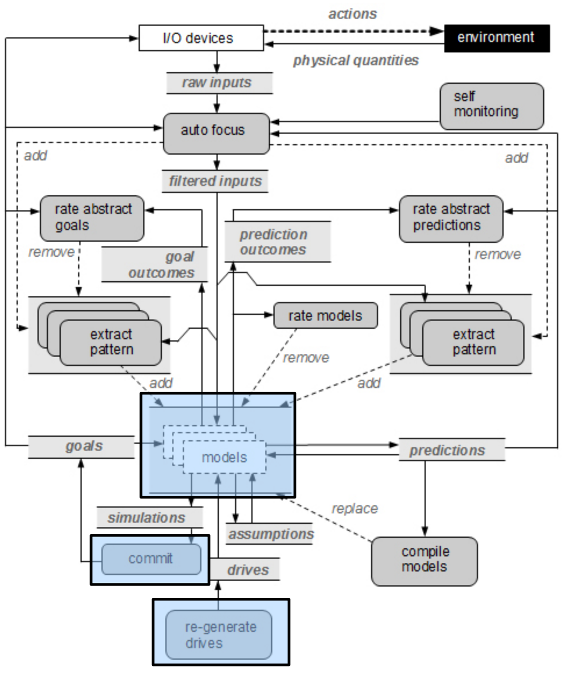


Figure 22 – Planning

8.3 Structural Description

This section maps the functional description of the architecture to an implementation based on the Replicode executive. The structure described here is an extension of the structure presented in section 6.3.1.

8.3.1 Overview

The system is composed of (a) a multitude of models, (b) one instance of auto-focus and, (c) a collection of pattern extractors (see figure 22 below).

The auto-focus is a component that implements the auto-focus process described in section 7.2.2. For performance reasons, this component is also responsible for rating the abstract goals and predictions, a process described in section 7.2.5.

There is a one-to-one correspondence between a pattern extraction process (section 7.2.3) and a pattern extractor (a component).

Models are projected onto two groups called the primary and secondary groups. Models active in the primary group receive the filtered inputs and produce predictions and goals. Models that are not active in the primary group are models whose success rate is not high enough. These models are kept active in the secondary group only: they receive inputs and produce – silently – predictions (they are not injected in the system). This is meant to allow weak models to improve, and to detect context switches as mentioned in section 7.2.4,.

Drive re-generation programs live in a separate group: this allows to deactivate some drives by increasing the activation threshold of the group: (see section 7.2.8).

The monitoring of the system's performance is performed by the executive and does not appear in the apparent structure of the system.

Model rating, simulation and commitment are performed by “reasoning units”, described in section 7.4.2 below.

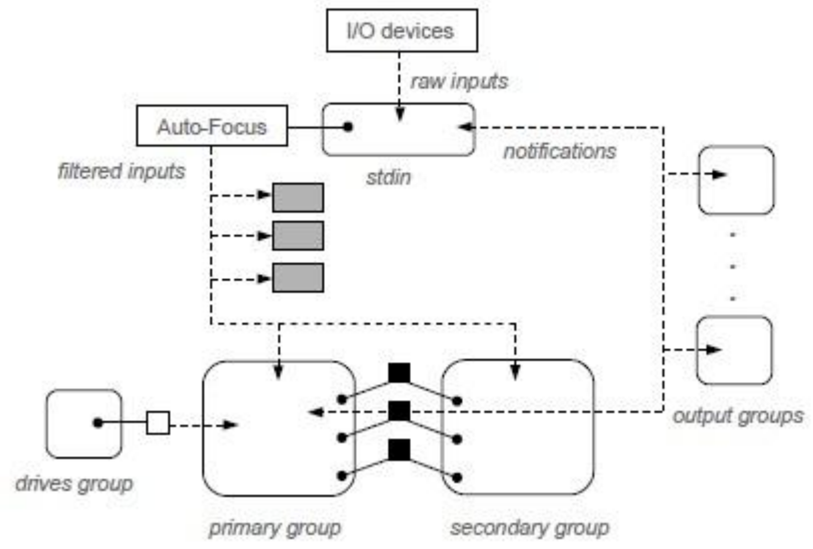
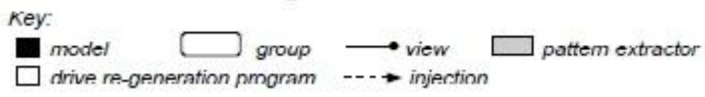


Figure 23 – Structure



The output groups are either input groups (stdin) of other systems (arranged accordingly to the principle of Integrated Cognitive Control), or other groups of the same system if one needs to partition the memory, i.e. isolate some models from others.

Compaction is performed by a control system (following ICC) and as such has no counterpart in terms of components.

8.3.2 Reasoning Units

In the main, reasoning is performed by primitive units (see figure 23 below). A reasoning unit is composed of a model, one or several views and a controller (one per view).

The controller is a lightweight thread that performs forward and backward chaining, i.e. deduction and abduction. The controller also manages a collection of monitors, which are also lightweight threads and are responsible for:

- prediction monitors: detect the occurrence of a predicted fact within the time interval specified for the prediction; at the deadline of the prediction (upper bound of the interval), the outcome (success or failure) is assessed and the model is rated accordingly.
- goal monitors: detect the occurrence of the target fact and asses the outcome of the goal; collect simulation results and commit to appropriate sub-goals.

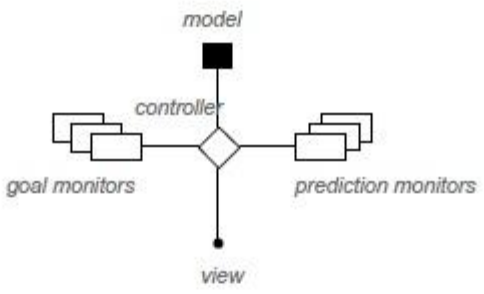


Figure 24 – Reasoning Unit

Each time a model is produced, a computation unit is instantiated and parametrized by the model. The implementation of the system consist essentially in a huge collection of computation units, supported by an instance of the auto-focus and a collection of pattern extractors.

8.4 Open Issues and Future Work

This section describe our approach towards selected short-term developments.

In its current state, the architecture does not have any meta-learning abilities: learning is a fixed process, and planning is also a fixed process. This does not constitute however a significant limitation with regards to the objectives of the project. To develop furthermore the meta-learning abilities is left for the long-term, as we give precedence to industrial applications (which require improved dependability, accuracy and robustness) over high-level cognitive feats that we feel will not be required until significant impact in the industry has been achieved.

Learning is triggered by the production of goals and predictions and relies entirely on observation: this limits the acquisition of knowledge to apparent phenomena. In other words, the system lacks curiosity and is not able to produce hypotheses and to experiment. For example, we have experimented “motor babbling” to bootstrap learning in the prototype. However this relied entirely on ad-hoc hand-crafted goals injected in the system and said code has been removed from the final system. The question of how to generate such goals in a principle way is still open.

The system strives to achieve multiple goals stemming from its internal (user-defined) drives. However there is no guarantee that it will *actually* achieve these goals. This “best effort” policy is deemed good enough for the project but constitutes a severe limitation in the context of mission-critical applications: in such cases we need to guarantee that the system (a) does actually performs as intended and (b) does not violate constraints imposed by the designers. In other words, significant work remains to be done to bound the operation of the system in a dependable, deterministic way by imposing top-down (allonomic) constraints to control the bottom-up (autonomic) development of the system.

Acknowledgments

This work was supported by the European Project HUMANOBS – Humanoids that Learn Socio-Communicative Skills By Observation (grant number 231453), and by research grants from RANNIS, Iceland.

9 References

1 - Replicode Specification V2.0 - available at:

https://projects.humanobs.org/projects/language/documents

2 –Wang, P. *Rigid Flexibility: The Logic of Intelligence*. Springer, Dordrecht. 2006.

3 – Nivel, E. and Thórisson, K. R. Self-Programming: Operationalizing Autonomy. *Proceedings of the Second Conference on Artificial General Intelligence*. 2009.

4 – Specification of the Interaction Scenario V1.0 - available at:

https://projects.humanobs.org/projects/wp7/documents

5 - Demiris, Y. and Khadhouri, B. Hierarchical attentive multiple models for execution and recognition (HAMMER). Robotics and Autonomous Systems Journal, 54:361–369. 2005.

6 – Wolpert, D. M., Gharamani, Z., and Jordan, M. An internal model for sensorimotor integration. Science, 269:1179–1182. 1995.

7 – Pezzulo, G. and Calvi, G. Schema-based design and the AKIRA Schema Language: An Overview. In M.V. Butz and O. Sigaud, G. Pezzulo, G. Baldassarre (Eds.), *Anticipatory Behavior in Adaptive Learning Systems: Advances in Anticipatory Processing*. Springer, LNAI 4520. 2007

8 – Sanz, R. An Integrated Control Model of Consciousness. *Proceedings of the conference Toward a Science of Consciousness*. 2002.

9 – Model Proposer V1.0 – available at:

https://projects.humanobs.org/projects/wp2/documents

10 - H. Dindo and D. Zambuto. "A Probabilistic Approach to Learning a Visually Grounded Language Model through Human-Robot Interaction", in 2010 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS), Taipei, Taiwan, October 18-22, 2010

10 Annex 1: Masterplan of the Prototype

This Annex gives the compilable code of the Masterplan. In addition to their code, each model and state is also summarized in pseudo-code for readability. This annex also includes the learned models and states that have been integrated into the Masterplan – for such models/states, an explicit mention is made to differentiate them from the innate models/states.

11 Ontology

```
; domain-dependent classes.

!class (vec3 x:nb y:nb z:nb)
!class (speech_context (_obj {speaker:ent listener:ent}))



; device functions.

!dfn (grab_hand); arg0: a hand, arg1: deadline.
!dfn (release_hand); arg0: a hand, arg1: deadline.
!dfn (move_hand :); arg0: a hand, arg1: target position, arg2: deadline.
!dfn (speak :); arg0: a word, arg1: deadline.
!dfn (look_at :); arg0: a vec3, arg1: deadline.
!dfn (point_at :); arg0: a hand, arg1: target position, arg2: deadline.

; application ontology.

; attributes.

essence:(ont 1) [[SYNC_FRONT now 1 forever root nil]]; usage: (mk.val x essence "sphere" 1)
part_of:(ont 1) [[SYNC_FRONT now 1 forever root nil]]
position:(ont 1) [[SYNC_FRONT now 1 forever root nil]]; usage: (mk.val x position (vec3 0 0 0) 1)
color:(ont 1) [[SYNC_FRONT now 1 forever root nil]]; usage: (mk.val x color "green" 1)
attachment:(ont 1) [[SYNC_FRONT now 1 forever root nil]]; usage: (mk.val self_right_hand attachment
a_thing 1)
role:(ont 1) [[SYNC_FRONT now 1 forever root nil]]

speaking:(ont 1) [[SYNC_FRONT now 1 forever root nil]]; usage: (mk.val an_actor speaking "a_word" 1)
listening:(ont 1) [[SYNC_FRONT now 1 forever root nil]]; usage: (mk.val an_actor listening "a_word" 1)
pointing:(ont 1) [[SYNC_FRONT now 1 forever root nil]]; usage: (mk.val an_actor pointing (vec3 0 0 0) 1)
knowing:(ont 1) [[SYNC_FRONT now 1 forever root nil]]; usage: (mk.val an_actor knowing something 1)

action:(ont 1) [[SYNC_FRONT now 1 forever root nil]]
action_target:(ont 1) [[SYNC_FRONT now 1 forever root nil]]
most_salient:(ont 1) [[SYNC_FRONT now 1 forever root nil]]

; values.

actor:(ont 1) [[SYNC_FRONT now 1 forever root nil]]
```

```
interviewer:(ont 1) [[SYNC_FRONT now 1 forever root nil]]
interviewee:(ont 1) [[SYNC_FRONT now 1 forever root nil]]
hand:(ont 1) [[SYNC_FRONT now 1 forever root nil]]
head:(ont 1) [[SYNC_FRONT now 1 forever root nil]]


; initial application objects.

self:(ont 1) [[SYNC_FRONT now 1 forever root nil]]
(mk.val self essence actor 1) [[SYNC_FRONT now 1 forever root nil]]
self_right_hand:(ent 1) [[SYNC_FRONT now 1 forever root nil]]
self_right_hand_is_a_hand:(mk.val self_right_hand essence hand 1) [[SYNC_FRONT now 1 forever root nil]]
self_right_hand_belongs_to_self:(mk.val self_right_hand part_of self 1) [[SYNC_FRONT now 1 forever root nil]]
self_head:(ent 1) [[SYNC_FRONT now 1 forever root nil]]
(mk.val self_head essence head 1) [[SYNC_FRONT now 1 forever root nil]]
(mk.val self_head part_of self 1) [[SYNC_FRONT now 1 forever root nil]]

programmer:(ont 1) [[SYNC_FRONT now 1 forever root nil]]

human1:(ent 1) [[SYNC_FRONT now 1 forever root nil]]
(mk.val human1 essence actor 1) [[SYNC_FRONT now 1 forever root nil]]
human1_right_hand:(ent 1) [[SYNC_FRONT now 1 forever root nil]]
(mk.val human1_right_hand essence hand 1) [[SYNC_FRONT now 1 forever root nil]]
(mk.val human1_right_hand part_of human1 1) [[SYNC_FRONT now 1 forever root nil]]
human1_head:(ent 1) [[SYNC_FRONT now 1 forever root nil]]
(mk.val human1_head essence head 1) [[SYNC_FRONT now 1 forever root nil]]
(mk.val human1_head part_of human1 1) [[SYNC_FRONT now 1 forever root nil]]

human2:(ent 1) [[SYNC_FRONT now 1 forever root nil]]
(mk.val human2 essence actor 1) [[SYNC_FRONT now 1 forever root nil]]
human2_right_hand:(ent 1) [[SYNC_FRONT now 1 forever root nil]]
(mk.val human2_right_hand essence hand 1) [[SYNC_FRONT now 1 forever root nil]]
(mk.val human2_right_hand part_of human2 1) [[SYNC_FRONT now 1 forever root nil]]
human2_head:(ent 1) [[SYNC_FRONT now 1 forever root nil]]
(mk.val human2_head essence head 1) [[SYNC_FRONT now 1 forever root nil]]
(mk.val human2_head part_of human2 1) [[SYNC_FRONT now 1 forever root nil]]

top_level:(ont 1) [[SYNC_FRONT now 1 forever root nil]]; to be targeted by a goal (top-level goal).
```

12 Physics

```
; initial high-level patterns (HLP) for manipulating objects.
; all models/states are learned during "motor" babbling".
; the models/states are given here in a compilable form.
;
; abstraction is fully automatic: HLPs are built from examples.
; the provided examples will die at the first update period (upr).



; hand_pos[H P T]:[(H position P)T,(H essence hand)T] - LEARNED.
```

```
some_object_0:(ent 1) |[]

some_object_pos_0:(mk.val some_object_0 position (vec3 0 0 0) 1) |[]
f_some_object_pos_0:(fact some_object_pos_0 0us 1 1) |[]

some_object_is_a_hand_0:(mk.val some_object_0 essence hand 1) |[]
f_some_object_is_a_hand_0:(fact some_object_is_a_hand_0 0us 1 1) |[]

hand_pos:(cst [f_some_object_pos_0 f_some_object_is_a_hand_0] [attention] 0us 1) [[SYNC_FRONT now
0 forever stdin nil 1]]

; ----------------------------------------------------------------------

; same_hand_pos[H X P T]:[(icst hand_pos_cst X P)T,(X position P)T] - LEARNED.

some_hand_1:(ent 1) |[]
some_object_1:(ent 1) |[]

i_hand_pos_1:(icst hand_pos [some_hand_1 (vec3 0 0 0) 0us] 1) |[]
f_i_hand_pos_1:(fact i_hand_pos_1 0us 1 1) |[]

some_object_pos_1:(mk.val some_object_1 position (vec3 0 0 0) 1) |[]
f_some_object_pos_1:(fact some_object_pos_1 0us 1 1) |[]

same_hand_pos:(cst [f_i_hand_pos_1 f_some_object_pos_1] [attention] 0us 1) [[SYNC_FRONT now 0
forever stdin nil 1]]

; ----------------------------------------------------------------------

; grab[H X T1 T0]:[(grab_hand H T1)T0 -> (H attachment X)T1] - LEARNED.

some_hand_2:(ent 1) |[]
some_object_2:(ent 1) |[]

cmd_grab_2:(cmd grab_hand [some_hand_2 100000us] 1) |[]
f_cmd_grab_2:(fact cmd_grab_2 0us 1 1) |[]

attached_2:(mk.val some_hand_2 attachment some_object_2 1) |[]
f_attached_2:(fact attached_2 100000us 1 1) |[]

grab:(mdl [f_cmd_grab_2 f_attached_2] [attention] 0us 1) [[SYNC_FRONT now 0 forever stdin nil 1]]

; ----------------------------------------------------------------------

; req_grab[H X P T T0 T1]:[(icst same_hand_pos H X P T0)T -> (imdl grab H X T1 T)T] - LEARNED.

some_hand_3:(ent 1) |[]
some_object_3:(ent 1) |[]

i_same_hand_pos_3:(icst same_hand_pos [some_hand_3 some_object_3 (vec3 0 0 0) 0us] 1) |[]
f_i_same_hand_pos_3:(fact i_same_hand_pos_3 0us 1 1) |[]
```

```
i_grab_3:(imdl grab [some_hand_2 some_object_2 100000us 0us] 1) |[]
f_i_grab_3:(fact i_grab_3 0us 1 1) |[]

req_grab:(mdl [f_i_same_hand_pos_3 f_i_grab_3] [attention] 0us 1) [[SYNC_FRONT now 0 forever stdin nil
1]]

; ----------------------------------------------------------------------

; release[H X T1 T0]:[(release_hand H T1)T0 -> |(H attachment X)T1] - LEARNED.

some_hand_4:(ent 1) |[]
some_object_4:(ent 1) |[]

cmd_release_4:(cmd release_hand [some_hand_4 100000us] 1) |[]
f_cmd_release_4:(fact cmd_release_4 0us 1 1) |[]

attached_4:(mk.val some_hand_4 attachment some_object_4 1) |[]
not_f_attached_4:(|fact attached_4 100000us 1 1) |[]

release:(mdl [f_cmd_release_4 not_f_attached_4] [attention] 0us 1) [[SYNC_FRONT now 0 forever stdin nil
1]]

; ----------------------------------------------------------------------

; req_release[H X T1 T]:[(H attachment X)T -> (imdl release H X T1 T)T] - LEARNED.

some_hand_5:(ent 1) |[]
some_object_5:(ent 1) |[]

attached_5:(mk.val some_hand_5 attachment some_object_5 1) |[]
f_attached_5:(fact attached_5 0us 1 1) |[]

i_release_5:(imdl release [some_hand_5 some_object_5 1000000us 0us] 1) |[]
f_i_release_5:(fact i_release_5 0us 1 1) |[]

req_release:(mdl [f_attached_5 f_i_release_5] [attention] 0us 1) [[SYNC_FRONT now 0 forever stdin nil 1]]

; ----------------------------------------------------------------------

; move[H P T1 T0]:[(move_hand H P T1)T0 -> (icst hand_pos H P T1)T1] - LEARNED.

some_hand_6:(ent 1) |[]

cmd_move_6:(cmd move_hand [some_hand_6 (vec3 0 0 0) 100000us] 1) |[]
f_cmd_move_6:(fact cmd_move_6 0us 1 1) |[]

i_hand_pos_6:(icst hand_pos [some_hand_6 (vec3 0 0 0) 100000us] 1) |[]
f_i_hand_pos_6:(fact i_hand_pos_6 100000us 1 1) |[]

move:(mdl [f_cmd_move_6 f_i_hand_pos_6] [attention] 0us 1)  [[SYNC_FRONT now 0 forever stdin nil 1]]

; ----------------------------------------------------------------------
; link[H X P T1 T]:[(H attachment X)T -> (icst same_hand_pos H X P T1)T1] - LEARNED.

some_hand_7:(ent 1) |[]
some_object_7:(ent 1) |[]

attached_7:(mk.val some_hand_7 attachment some_object_7 1) |[]
f_attached_7:(fact attached_7 0us 1 1) |[]

i_same_hand_pos_7:(icst same_hand_pos [some_hand_7 some_object_7 (vec3 0 0 0) 100000us] 1) |[]
f_i_same_hand_pos_7:(fact i_same_hand_pos_7 100000us 1 1) |[]

link:(mdl [f_attached_7 f_i_same_hand_pos_7] [attention] 0us 1)  [[SYNC_FRONT now 0 forever stdin nil 1]]
```

13 Speech

```
; states referred to by speech in/out porgrams - HAND-CODED.

; all of these models/composite states have actually been learned while observing demonstrations of the interview.
; nevertheless, we need to name them in order to reference them in the speech in/out programs.
; thus, these learned states have been purged from memory and hand-coded - which gave the opportunity to assign labels to them.
; be aware that the same overhead would also be incurred shall the requirements for dialogue include things like "move your hand", "grab something", etc.


; ----------------------------------------------------------------------

; stand_col_ess_pos[X P E C T]:[(X color C)T,(X essence E)T,(X position P)T]

a_thing_0:(ent 1) |[]

col_0:(mk.val a_thing_0 color "black" 1) |[]
f_col_0:(fact col_0 0us 1 1) |[]

ess_0:(mk.val a_thing_0 essence "cube" 1) |[]
f_ess_0:(fact ess_0 0us 1 1) |[]

pos_0:(mk.val a_thing_0 position (vec3 0 0 0) 1) |[]
f_pos_0:(fact pos_0 0us 1 1) |[]

stand_col_ess_pos:(cst [f_col_0 f_ess_0 f_pos_0] [attention] 0us 1) [[SYNC_FRONT now 0 forever stdin nil 1]]

; ----------------------------------------------------------------------

; stand_col_ess_pos_sln[X P E C T]:[(X color C)T,(X essence E)T,(X position P)T,(X most_salient nil)T]

a_thing_1:(ent 1) |[]

col_1:(mk.val a_thing_1 color "black" 1) |[]
f_col_1:(fact col_1 0us 1 1) |[]
```

```
ess_1:(mk.val a_thing_1 essence "cube" 1) |[]
f_ess_1:(fact ess_1 0us 1 1) |[]

pos_1:(mk.val a_thing_1 position (vec3 0 0 0) 1) |[]
f_pos_1:(fact pos_1 0us 1 1) |[]

sln_1:(mk.val a_thing_1 most_salient nil 1) |[]
f_sln_1:(fact sln_1 0us 1 1) |[]

stand_col_ess_pos_sln:(cst [f_col_1 f_ess_1 f_pos_1 f_sln_1] [attention] 0us 1) [[SYNC_FRONT now 0
forever stdin nil 1]]

; ----------------------------------------------------------------------

; stand_pos_sln[X P T]:[(X position P)T,(X most_salient nil)T]

a_thing_2:(ent 1) |[]

pos_2:(mk.val a_thing_2 position (vec3 0 0 0) 1) |[]
f_pos_2:(fact pos_2 0us 1 1) |[]

sln_2:(mk.val a_thing_2 most_salient nil 1) |[]
f_sln_2:(fact sln_2 0us 1 1) |[]

stand_pos_sln:(cst [f_pos_2 f_sln_2] [attention] 0us 1) [[SYNC_FRONT now 0 forever stdin nil 1]]

; ----------------------------------------------------------------------

; taken_col_ess[H X P E C T]:[(X color C)T,(X essence E)T,(icst same_hand_pos H X P T)T]

a_thing_3:(ent 1) |[]
cube_3:(ent 1) |[]
a_hand_3:(ent 1) |[]

col_3:(mk.val a_thing_3 color "black" 1) |[]
f_col_3:(fact col_3 0us 1 1) |[]

ess_3:(mk.val a_thing_3 essence cube_3 1) |[]
f_ess_3:(fact ess_3 0us 1 1) |[]

i_same_hand_pos_3:(icst same_hand_pos [a_hand_3 a_thing_3 (vec3 0 0 0) 0us] 1) |[]
f_i_same_hand_pos_3:(fact i_same_hand_pos_3 0us 1 1) |[]

taken_col_ess:(cst [f_col_3 f_ess_3 f_i_same_hand_pos_3] [attention] 0us 1) [[SYNC_FRONT now 0
forever stdin nil 1]]

; ----------------------------------------------------------------------

; taken_col_ess_sln[H X P E C T]:[(X color C)T,(X essence E)T,(X most_salient nil)T,(icst same_hand_pos
H X P T)T]
```

```
a_thing_4:(ent 1) |[]
cube_4:(ent 1) |[]
a_hand_4:(ent 1) |[]

col_4:(mk.val a_thing_4 color "black" 1) |[]
f_col_4:(fact col_4 0us 1 1) |[]

ess_4:(mk.val a_thing_4 essence cube_4 1) |[]
f_ess_4:(fact ess_4 0us 1 1) |[]

sln_4:(mk.val a_thing_4 most_salient nil 1) |[]
f_sln_4:(fact sln_4 0us 1 1) |[]

i_same_hand_pos_4:(icst same_hand_pos [a_hand_4 a_thing_4 (vec3 0 0 0) 0us] 1) |[]
f_i_same_hand_pos_4:(fact i_same_hand_pos_4 0us 1 1) |[]

taken_col_ess_sln:(cst [f_col_4 f_ess_4 f_sln_4 f_i_same_hand_pos_4] [attention] 0us 1) [[SYNC_FRONT
now 0 forever stdin nil 1]]

; ----------------------------------------------------------------------

; taken_sln[H X P T]:[(X most_salient nil)T,(icst same_hand_pos H X P T)T]

a_thing_5:(ent 1) |[]
a_hand_5:(ent 1) |[]

sln_5:(mk.val a_thing_5 most_salient nil 1) |[]
f_sln_5:(fact sln_5 0us 1 1) |[]

i_same_hand_pos_5:(icst same_hand_pos [a_hand_5 a_thing_5 (vec3 0 0 0) 0us] 1) |[]
f_i_same_hand_pos_5:(fact i_same_hand_pos_5 0us 1 1) |[]

taken_sln:(cst [f_sln_5 f_i_same_hand_pos_5] [attention] 0us 1) [[SYNC_FRONT now 0 forever stdin nil 1]]


; speech-in grammar - HAND-CODED.


; parsing programs.

speech_in1:(pgm |[] []
   []
      (ptn (mk.val speaker: speaking verb: ::) |[])
      (ptn (mk.val listener: listening sp: ::) |[])
   []
      (= listener sp)
   |[]
[]
   (inj []
      sc:(speech_context speaker listener 1)
      [SYNC_FRONT now 1 1 stdin nil]
   )
```

```
  (inj []
    (mk.val sc action verb 1)
    [SYNC_FRONT now 1 1 stdin nil]
  )
1) |[]
```

```
i_speech_in1:(ipgm speech_in1 |[] RUN_ALWAYS 0us NOTIFY 1) [[SYNC_FRONT now 0 forever stdin nil
1]]
```

```
speech_in2:(pgm |[] []
  []
    (ptn sc:(speech_context speaker: ::) |[])
    (ptn (mk.val actor: speaking "a" ::) |[])
  |[]
  []
    (= speaker actor)
[]
  (inj []
    (mk.val sc action_target nil 1)
    [SYNC_FRONT now 1 1 stdin nil]
  )
1) |[]
```

```
i_speech_in2:(ipgm speech_in2 |[] RUN_ALWAYS 0us NOTIFY 1) [[SYNC_FRONT now 0 forever stdin nil
1]]
```

```
speech_in2a:(pgm |[]  []
  []
    (ptn sc:(speech_context speaker: ::) |[])
    (ptn (mk.val actor: speaking "the" ::) |[])
  |[]
  []
    (= speaker actor)
[]
  (inj []
    (mk.val sc action_target most_salient 1)
    [SYNC_FRONT now 1 1 stdin nil]
  )
1) |[]
```

```
i_speech_in2a:(ipgm speech_in2a |[] RUN_ALWAYS 0us NOTIFY 1) [[SYNC_FRONT now 0 forever stdin
nil 1]]
```

```
speech_in2b:(pgm |[] []
  []
    (ptn sc:(speech_context speaker: ::) |[])
    (ptn (mk.val actor: speaking "it" ::) |[])
  |[]
  []
    (= speaker actor)
```

```
[]
   (inj []
      (mk.val sc action_target most_salient 1)
      [SYNC_FRONT now 1 1 stdin nil]
   )
1) |[]

i_speech_in2b:(ipgm speech_in2b |[] RUN_ALWAYS 0us NOTIFY 1) [[SYNC_FRONT now 0 forever stdin
nil 1]]


speech_in3:(pgm |[] []
   []
      (ptn sc:(speech_context speaker: ::) |[])
      (ptn (mk.val actor: speaking col: ::) |[])
      oc:(ptn (mk.val _sc: action_target ::) |[])
   |[]
   []
      (= speaker actor)
      (= sc _sc)
[]
   (inj []
      (mk.val oc color col 1)
      [SYNC_FRONT now 1 1 stdin nil]
   )
1) |[]

i_speech_in3:(ipgm speech_in3 |[] RUN_ALWAYS 0us NOTIFY 1) [[SYNC_FRONT now 0 forever stdin nil
1]]


speech_in4:(pgm |[] []
   []
      (ptn sc:(speech_context speaker: ::) |[])
      (ptn (mk.val actor: speaking ess: ::) |[])
      oc:(ptn (mk.val _sc: action_target ::) |[])
   |[]
   []
      (= speaker actor)
      (= sc _sc)
[]
   (inj []
      (mk.val oc essence ess 1)
      [SYNC_FRONT now 1 1 stdin nil]
   )
1) |[]

i_speech_in4:(ipgm speech_in4 |[] RUN_ALWAYS 0us NOTIFY 1) [[SYNC_FRONT now 0 forever stdin nil
1]]


speech_in5:(pgm |[] []
   []
```

```
    (ptn sc:(speech_context speaker: ::) |[])
    (ptn (mk.val actor1: speaking "there" ::) |[])
    (ptn (mk.val actor2: pointing pos: ::) |[])
    oc:(ptn (mk.val _sc: action_target nil ::) |[])
  |[]
  []
    (= speaker actor1)
    (= speaker actor2)
    (= sc _sc)
[]
  (inj []
    (mk.val oc position pos 1)
    [SYNC_FRONT now 1 1 stdin nil]
  )
1) |[]

i_speech_in5:(ipgm speech_in5 |[] RUN_ALWAYS 0us NOTIFY 1) [[SYNC_FRONT now 0 forever stdin nil
1]]


speech_in6:(pgm |[] []
  []
    (ptn sc:(speech_context speaker: listener: ::) |[])
    (ptn (mk.val sc1: action "put" ::) |[])
    (ptn oc:(mk.val sc2: action_target nil ::) |[])
    (ptn (mk.val oc1: color col: ::) |[])
    (ptn (mk.val oc2: essence ess: ::) |[])
    (ptn (mk.val oc3: position pos: ::) |[])
  |[]
  []
    (= sc sc1)
    (= sc sc2)
    (= oc oc1)
    (= oc oc2)
    (= oc oc3)
[]
  (inj []
    v:(var 1)
    [SYNC_FRONT now 1 1 stdin nil]
  )
  (inj []
    i_cst:(icst stand_col_ess_pos [v col ess pos] 1)
    [SYNC_FRONT now 1 1 stdin nil]
  )
  (inj []
    f_i_cst:(fact i_cst now 1 1)
    [SYNC_FRONT now 1 1 stdin nil]
  )
  (inj []
    listener_goal:(mk.goal f_i_cst listener 1)
    [SYNC_FRONT now 1 1 stdin nil]
  )
  (inj []
    f_listener_goal:(fact listener_goal now 1 1)
    [SYNC_FRONT now 1 1 stdin nil]
  )
  (inj []
    speaker_goal:(mk.goal f_listener_goal speaker 1)
    [SYNC_FRONT now 1 1 stdin nil]
  )
  (inj []
    (fact speaker_goal now 1 1)
    [SYNC_FRONT now 1 1 stdin nil]
  )
1) |[]

i_speech_in6:(ipgm speech_in6 |[] RUN_ALWAYS 0us NOTIFY 1) [[SYNC_FRONT now 0 forever stdin nil
1]]


speech_in6a:(pgm |[] []
  []
    (ptn sc:(speech_context speaker: listener: ::) |[])
    (ptn (mk.val sc1: action "put" ::) |[])
    (ptn oc:(mk.val sc2: action_target most_salient ::) |[])
    (ptn (mk.val oc1: color col: ::) |[])
    (ptn (mk.val oc2: essence ess: ::) |[])
    (ptn (mk.val oc3: position pos: ::) |[])
  |[]
  []
    (= sc sc1)
    (= sc sc2)
    (= oc oc1)
    (= oc oc2)
    (= oc oc3)
[]
  (inj []
    v:(var 1)
    [SYNC_FRONT now 1 1 stdin nil]
  )
  (inj []
    i_cst:(icst stand_col_ess_pos_sln [v col ess pos] 1)
    [SYNC_FRONT now 1 1 stdin nil]
  )
  (inj []
    f_i_cst:(fact i_cst now 1 1)
    [SYNC_FRONT now 1 1 stdin nil]
  )
  (inj []
    listener_goal:(mk.goal f_i_cst listener 1)
    [SYNC_FRONT now 1 1 stdin nil]
  )
  (inj []
    f_listener_goal:(fact listener_goal now 1 1)
    [SYNC_FRONT now 1 1 stdin nil]
  )
  (inj []
    speaker_goal:(mk.goal f_listener_goal speaker 1)
    [SYNC_FRONT now 1 1 stdin nil]
  )
  (inj []
    (fact speaker_goal now 1 1)
    [SYNC_FRONT now 1 1 stdin nil]
  )
1) |[]
```

i_speech_in6a:(ipgm speech_in6a |[] RUN_ALWAYS 0us NOTIFY 1) [[SYNC_FRONT now 0 forever stdin nil 1]]

```
speech_in6b:(pgm |[] []
  []
    (ptn sc:(speech_context speaker: listener: ::) |[])
    (ptn (mk.val sc1: action "put" ::) |[])
    (ptn oc:(mk.val sc2: action_target most_salient ::) |[])
    (ptn (mk.val oc1: position pos: ::) |[])
  |[]
  []
    (= sc sc1)
    (= sc sc2)
    (= oc oc1)
[]
  (inj []
    v:(var 1)
    [SYNC_FRONT now 1 1 stdin nil]
  )
  (inj []
    i_cst:(icst stand_pos_sln [v pos] 1)
    [SYNC_FRONT now 1 1 stdin nil]
  )
  (inj []
    f_i_cst:(fact i_cst now 1 1)
    [SYNC_FRONT now 1 1 stdin nil]
  )
  (inj []
    listener_goal:(mk.goal f_i_cst listener 1)
    [SYNC_FRONT now 1 1 stdin nil]
  )
  (inj []
    f_listener_goal:(fact listener_goal now 1 1)
    [SYNC_FRONT now 1 1 stdin nil]
  )
  (inj []
    speaker_goal:(mk.goal f_listener_goal speaker 1)
    [SYNC_FRONT now 1 1 stdin nil]
  )
  (inj []
    (fact speaker_goal now 1 1)
    [SYNC_FRONT now 1 1 stdin nil]
  )
1) |[]
```

```
i_speech_in6b:(ipgm speech_in6b |[] RUN_ALWAYS 0us NOTIFY 1) [[SYNC_FRONT now 0 forever stdin
nil 1]]


speech_in7:(pgm |[] []
  []
    (ptn sc:(speech_context speaker: listener: ::) |[])
    (ptn (mk.val sc1: action "take" ::) |[])
    (ptn oc:(mk.val sc2: action_target nil ::) |[])
    (ptn (mk.val oc1: color col: ::) |[])
    (ptn (mk.val oc2: essence ess: ::) |[])
  |[]
  []
    (= sc sc1)
    (= sc sc2)
    (= oc oc1)
    (= oc oc2)
[]
  (inj []
    v:(var 1)
    [SYNC_FRONT now 1 1 stdin nil]
  )
  (inj []
    i_cst:(icst taken_col_ess [v col ess] 1)
    [SYNC_FRONT now 1 1 stdin nil]
  )
  (inj []
    f_i_cst:(fact i_cst now 1 1)
    [SYNC_FRONT now 1 1 stdin nil]
  )
  (inj []
    listener_goal:(mk.goal f_i_cst listener 1)
    [SYNC_FRONT now 1 1 stdin nil]
  )
  (inj []
    f_listener_goal:(fact listener_goal now 1 1)
    [SYNC_FRONT now 1 1 stdin nil]
  )
  (inj []
    speaker_goal:(mk.goal f_listener_goal speaker 1)
    [SYNC_FRONT now 1 1 stdin nil]
  )
  (inj []
    (fact speaker_goal now 1 1)
    [SYNC_FRONT now 1 1 stdin nil]
  )
1) |[]

i_speech_in7:(ipgm speech_in7 |[] RUN_ALWAYS 0us NOTIFY 1) [[SYNC_FRONT now 0 forever stdin nil
1]]
```

```
speech_in7a:(pgm |[] []
   []
      (ptn sc:(speech_context speaker: listener: ::) |[])
      (ptn (mk.val sc1: action "take" ::) |[])
      (ptn oc:(mk.val sc2: action_target most_salient ::) |[])
      (ptn (mk.val oc1: color col: ::) |[])
      (ptn (mk.val oc2: essence ess: ::) |[])
   |[]
   []
      (= sc sc1)
      (= sc sc2)
      (= oc oc1)
      (= oc oc2)
[]
   (inj []
      v:(var 1)
      [SYNC_FRONT now 1 1 stdin nil]
   )
   (inj []
      i_cst:(icst taken_col_ess_sln [v col ess] 1)
      [SYNC_FRONT now 1 1 stdin nil]
   )
   (inj []
      f_i_cst:(fact i_cst now 1 1)
      [SYNC_FRONT now 1 1 stdin nil]
   )
   (inj []
      listener_goal:(mk.goal f_i_cst listener 1)
      [SYNC_FRONT now 1 1 stdin nil]
   )
   (inj []
      f_listener_goal:(fact listener_goal now 1 1)
      [SYNC_FRONT now 1 1 stdin nil]
   )
   (inj []
      speaker_goal:(mk.goal f_listener_goal speaker 1)
      [SYNC_FRONT now 1 1 stdin nil]
   )
   (inj []
      (fact speaker_goal now 1 1)
      [SYNC_FRONT now 1 1 stdin nil]
   )
1) |[]

i_speech_in7a:(ipgm speech_in7a |[] RUN_ALWAYS 0us NOTIFY 1) [[SYNC_FRONT now 0 forever stdin
nil 1]]


speech_in7b:(pgm |[] []
   []
      (ptn sc:(speech_context speaker: listener: ::) |[])
      (ptn (mk.val sc1: action "take" ::) |[])
      (ptn oc:(mk.val sc2: action_target most_salient ::) |[])
   |[]
   []
      (= sc sc1)
      (= sc sc2)
[]
   (inj []
      v:(var 1)
      [SYNC_FRONT now 1 1 stdin nil]
   )
   (inj []
      i_cst:(icst taken_sln [v] 1)
      [SYNC_FRONT now 1 1 stdin nil]
   )
   (inj []
      f_i_cst:(fact i_cst now 1 1)
      [SYNC_FRONT now 1 1 stdin nil]
   )
   (inj []
      listener_goal:(mk.goal f_i_cst listener 1)
      [SYNC_FRONT now 1 1 stdin nil]
   )
   (inj []
      f_listener_goal:(fact listener_goal now 1 1)
      [SYNC_FRONT now 1 1 stdin nil]
   )
   (inj []
      speaker_goal:(mk.goal f_listener_goal speaker 1)
      [SYNC_FRONT now 1 1 stdin nil]
   )
   (inj []
      (fact speaker_goal now 1 1)
      [SYNC_FRONT now 1 1 stdin nil]
   )
1) |[]

i_speech_in7b:(ipgm speech_in7b |[] RUN_ALWAYS 0us NOTIFY 1) [[SYNC_FRONT now 0 forever stdin
nil 1]]

; speech-out grammar - HAND-CODED.


; goal translation programs.


speech_out1:(pgm |[] []
   []
      (ptn (mk.goal (fact (mk.goal (fact (icst stand_col_ess_pos [(var ::) col: ess: pos:] ::) ::) listener: ::) ::)
self ::) |[])
      (ptn (mk.val h1: part_of l: ::) |[])
      (ptn (mk.val h2: essence head ::) |[])
      (ptn (mk.val h3: position l_pos: ::) |[])
   |[]
   []
      (= listener l)
```

```
    (= h1 h2)
    (= h1 h3)
[]
  (cmd look_at [l_pos now] 1)
  (cmd speak ["Put a" now] 1)
  (cmd speak [col (+ now 500000us) ] 1)
  (cmd speak [ess (+ now 1000000us) now] 1)
  (cmd look_at [l_pos (+ now 1500000us)] 1)
  (cmd point_at [pos (+ now 2000000us)] 1)
  (cmd speak ["there" (+ now 2000000us)] 1)
1) |[]

i_speech_out1:(ipgm speech_out1 |[] RUN_ALWAYS 0us NOTIFY 1) [[SYNC_FRONT now 0 forever
attention nil 1]]


speech_out2:(pgm |[] []
  []
    (ptn (mk.goal (fact (mk.goal (fact (icst taken_col_ess [(var ::) col: ess:] ::) ::) listener: ::) ::) self ::) |[])
    (ptn (mk.val h1: part_of l: ::) |[])
    (ptn (mk.val h2: essence head ::) |[])
    (ptn (mk.val h3: position l_pos: ::) |[])
  |[]
  []
    (= listener l)
    (= h1 h2)
    (= h1 h3)
[]
  (cmd look_at [l_pos now] 1)
  (cmd speak ["Take a" now] 1)
  (cmd speak [col (+ now 500000us) ] 1)
  (cmd speak [ess (+ now 1500000us) now] 1)
1) |[]

i_speech_out2:(ipgm speech_out2 |[] RUN_ALWAYS 0us NOTIFY 1) [[SYNC_FRONT now 0 forever
attention nil 1]]


speech_out3:(pgm |[] []
  []
    (ptn (mk.goal (fact (mk.goal (fact (icst stand_pos_sln [: pos:] ::) ::) listener: ::) ::) self ::) |[])
    (ptn (mk.val h1: part_of l: ::) |[])
    (ptn (mk.val h2: essence head ::) |[])
    (ptn (mk.val h3: position l_pos: ::) |[])
  |[]
  []
    (= listener l)
    (= h1 h2)
    (= h1 h3)
[]
  (cmd look_at [l_pos now] 1)
  (cmd speak ["Put it" now] 1)
  (cmd look_at [l_pos (+ now 500000us)] 1)
```

```
    (cmd point_at [pos (+ now 500000us)] 1)
    (cmd speak ["there" (+ now 1000000us)] 1)
1) |[]

i_speech_out3:(ipgm speech_out3 |[] RUN_ALWAYS 0us NOTIFY 1) [[SYNC_FRONT now 0 forever attention nil 1]]
```

14 Goals

```
; models/states for motivating the agent.
;
; abstraction is fully automatic: HLPs are built from examples.
; the provided examples will die at the first upr.



; utilities.

; self_speak[S T1 T0]:[(cmd speak S T1)T0-> (self speaking S)T1] - HAND CODED (could be learned: yet another case of babling).

cmd_speak_0:(cmd speak ["something" 100000us] 1) |[]
f_cmd_speak_0:(fact cmd_speak_0 0us 1 1) |[]

speaking_0:(mk.val self speaking "something" 1) |[]
f_speaking_0:(fact speaking_0 100000us 1 1) |[]

self_speak:(mdl [f_cmd_speak_0 f_speaking_0] [stdin] 0us 1) [[SYNC_FRONT now 1 forever stdin nil 1]]

; ---------------------------------------------------------------------

assume_goal:(pgm |[] []
    []
        (ptn (fact (mk.goal (fact g:(mk.goal target: actor: ::) ::) ::) ::) |[])
    |[]
    |[]
[]
    (inj []
        (mk.asmp g nil 1 1)
        [SYNC_FRONT now 1 1 stdin nil]
    )
1) |[]

i_assume_goal(ipgm assume_goal |[] RUN_ALWAYS 0us SILENT 1) [[SYNC_FRONT now 0 forever attention nil 1]]

confirm_goal:(pgm |[] []
    []
        (ptn a:(mk.asmp (mk.goal target: actor: ::) ::) |[])
        (ptn f:(fact ::) |[])
    []
        (= target f)
    |[]
[]
    (set [a.vw.res 0])
1) |[]

i_confirm_goal:(ipgm confirm_goal |[] RUN_ALWAYS 0us SILENT 1) [[SYNC_FRONT now 0 forever attention nil 1]]



; ----------------------------------------------------------------------

; interviewer_thanks_interviewee[IR IE T]:[(speech_context IR IE) (IR speaking "thanks you")T,(IR role interviewer)T,(IE role interviewee)T] - HAND_CODED.

some_interviewer_1:(ent 1) |[]
some_interviewee_1:(ent 1) |[]

speech_context_1:(speech_context some_interviewer_1 some_interviewee_1 1) |[]
f_speech_context_1:(fact speech_context_1 0us 1 1) |[]

actor_thanks_1:(mk.val some_interviewer_1 speaking "tank you" 1) |[]
f_actor_thanks_1:(fact actor_thanks_1 0us 1 1) |[]

actor_is_interviewer_1:(mk.val some_interviewer_1 role interviewer 1) |[]
f_actor_is_interviewer_1:(fact actor_is_interviewer_1 0us 1 1) |[]

actor_is_interviewee_1:(mk.val some_interviewee_1 role interviewer 1) |[]
f_actor_is_interviewee_1:(fact actor_is_interviewee_1 0us 1 1) |[]

interviewer_thanks_interviewee:(cst [f_speech_context_1 f_actor_thanks_1 f_actor_is_interviewer_1 f_actor_is_interviewee_1] [stdin] 0us 1) [[SYNC_FRONT now 0 forever stdin nil 1]]

; ----------------------------------------------------------------------



; as an interviewee, the system has learned to comply to the interviewer's requests: the goal is to have the interviewer thnak the interviewee.
; as an interviewer: same goal as above (in that case, interviewer=self).
; for both roles: model to trigger the pursuit of the goal "interviewer thanks interviewee"; also used for automatic monitoring of the success of the top-level goal.

; top_level_model[A T]:[(icst interviewer_thanks A T)T -> (top_level)T]- HAND-CODED.

some_interviewer_2:(ent 1) |[]
some_interviewee_2:(ent 1) |[]

interviewer_thanks_interviewee_2:(icst interviewer_thanks_interviewee [some_interviewer_2 some_interviewee_2 0us] 1) |[]
f_interviewer_thanks_interviewee_2:(fact interviewer_thanks_interviewee_2 0us 1 1) |[]

f_top_level_2:(fact top_level 0us 1 1) |[]

top_level_model:(mdl [f_interviewer_thanks_interviewee_2 f_top_level_2] [stdin] 0us 1) [[SYNC_FRONT now 1 forever stdin nil 1]]

; ----------------------------------------------------------------------
; program for killing the top level goal when satisfied.

top_level_control:(pgm |[] []
  []
    (ptn (fact (mk.pred top_level ::) ::) |[])
    (ptn self_goal:(fact (mk.goal top_level self ::) ::) |[])
  |[]
  |[]
[]
  (set [self_goal.vw.res 0])
1) |[]

i_top_level_control:(ipgm top_level_control |[] RUN_ALWAYS 0us NOTIFY 1) [[SYNC_FRONT now 0
forever attention nil 1]]

; ----------------------------------------------------------------------

; (goal (goal (icst stand_col_ess_pos something "red" "cube" (100 100 0)) IE) self)

some_thing_2:(ent 1) |[]

red_cube_100_100_0:(icst stand_col_ess_pos [some_thing_2 "red" "cube" (vec3 100 100 0) 2000000us] 1)
|[]
f_red_cube_100_100_0:(fact red_cube_100_100_0 2000000us 1 1) |[]

one_step_manipulation_task:(mk.goal f_red_cube_100_100_0 human2 1) |[]
f_one_step_manipulation_task:(fact one_step_manipulation_task 0us 1 1) |[]

g_self_f_one_step_manipulation_task:(mk.goal f_one_step_manipulation_task self 1) |[]
f_g_self_f_one_step_manipulation_task:(fact g_self_f_one_step_manipulation_task 2000000us 1 1)
[[SYNC_FRONT now 0 forever stdin nil]]

; ----------------------------------------------------------------------

; 2_steps_object_manipulation[X H IE P1 P0 T3 C E T2 T1 T0]:[(goal (icst taken_col_ess H X P0 E C T1)T1
IE)T0 -> (goal (icst stand_pos_sln X P1 T3)T3 IE)T2] - HAND-CODED.

some_thing_3:(ent 1) |[]
some_hand_3:(ent 1) |[]
some_actor_3:(ent 1) |[]

i_taken_col_ess_3:(icst taken_col_ess [some_hand_3 some_thing_3 (vec3 0 0 0) "cube" "red" 100000us] 1)
|[]
f_i_taken_col_ess_3:(fact i_taken_col_ess_3 100000us 1 1) |[]

g_interviewee_f_i_taken_col_ess_3:(mk.goal f_i_taken_col_ess_3 some_actor_3 1) |[]
f_g_interviewee_f_i_taken_col_ess_3:(fact g_interviewee_f_i_taken_col_ess_3 0us 1 1) |[]

i_stand_pos_sln_3:(icst stand_pos_sln [some_thing_3 2000000us] 1) |[]
f_i_stand_pos_sln_3:(fact i_stand_pos_sln_3 2000000us 1 1) |[]

g_interviewee_f_i_stand_pos_sln_3:(mk.goal f_i_stand_pos_sln_3 some_actor_3 1) |[]
```

```
f_g_interviewee_f_i_stand_pos_sln_3:(fact g_interviewee_f_i_stand_pos_sln_3 500000us 1 1) |[]

two_steps_object_manipulation:(mdl [f_g_interviewee_f_i_taken_col_ess_3
f_g_interviewee_f_i_stand_pos_sln_3] [stdin] 0us 1) [[SYNC_FRONT now 1 forever stdin nil 1]]

; ---------------------------------------------------------------------

; (goal (goal (icst stand_pos_sln X T1) IE)T1 self)

some_thing_4:(ent 1) |[]

i_stand_pos_sln:(icst stand_pos_sln [some_thing_4 2000000us] 1) |[]
f_i_stand_pos_sln:(fact i_stand_pos_sln 2000000us 1 1) |[]

g_interviewee_put_sln_object_somewhere:(mk.goal f_i_stand_pos_sln human2 1) |[]
f_g_interviewee_put_sln_object_somewhere:(fact g_interviewee_put_sln_object_somewhere 0us 1 1) |[]

g_self_has_interviewee_put_sln_object_somewhere:(mk.goal  f_g_interviewee_put_sln_object_somewhere
self 1) |[]
f_g_self_has_interviewee_put_sln_object_somewhere:(fact
g_self_has_interviewee_put_sln_object_somewhere 0us 1 1) [[SYNC_FRONT now 0 forever stdin nil]]
```

15 Control

```
; initial code for controlling the agent (programmer's side).

; command parser for role switching.

switch_role:(pgm []
  (ptn r:(ont ::) |[])
  (ptn s:(val ::) |[])
[]
  []
    (ptn (mk.val programmer speaking "switch" ::) |[])
    (ptn (mk.val programmer speaking "to" ::) |[])
    (ptn (mk.val programmer speaking s ::) |[])
    (ptn old_role:(fact (mk.val self role ::) ::) |[])
    (ptn h1_role:(fact (mk.val human1 role ::) ::) |[])
    (ptn h2_role:(fact (mk.val human2 role ::) ::) |[])
  |[]
  |[]
[]
  (inj []
    new_role:(mk.val self role r 1)
    [SYNC_FRONT now 1 1 stdin nil]
  )
  (inj []
    (fact new_role now 1 1)
    [SYNC_FRONT now 1 forever stdin nil]
  )
  (inj []
    top_level_goal:(mk.goal top_level self 1)
    [SYNC_FRONT now 1 1 stdin nil]
  )
  (inj []
    (fact top_level_goal now 1 1)
    [SYNC_STATE now 1 forever stdin nil]
  )
  (set [old_role.vw.res 0])
  (set [h1_role.vw.res 0])
  (set [h2_role.vw.res 0])
1) |[]

s_interviewee:(val "interviewee" 1) |[]
s_interviewer:(val "interviewer" 1) |[]

i_switch_to_interviewee:(ipgm switch_role [interviewee s_interviewee] RUN_ALWAYS 0us NOTIFY 1)
[[SYNC_FRONT now 0 forever stdin nil 1]]
i_switch_to_interviewer:(ipgm switch_role [interviewer s_interviewer] RUN_ALWAYS 0us NOTIFY 1)
[[SYNC_FRONT now 0 forever stdin nil 1]]



switch_to_no_role:(pgm |[] []
  []
    (ptn (mk.val programmer speaking "switch" ::) |[])
    (ptn (mk.val programmer speaking "to" ::) |[])
    (ptn (mk.val programmer speaking "observer" ::) |[])
    (ptn self_role:(fact (mk.val self role ::) ::) |[])
    (ptn self_goal:(fact (mk.goal top_level self ::) ::) |[])
  |[]
  |[]
[]
  (set [self_role.vw.res 0])
  (set [self_goal.vw.res 0])
  (inj []
    h1_interviewer:(mk.val human1 role interviewer 1)
    [SYNC_FRONT now 1 1 stdin nil]
  )
  (inj []
    (fact h1_interviewer 0us 1 1)
    [SYNC_STATE now 1 forever stdin nil]
  )
  (inj []
    h2_interviewee:(mk.val human2 role interviewee 1)
    [SYNC_FRONT now 1 1 stdin nil]
  )
  (inj []
    (fact h2_interviewee 0us 1 1)
    [SYNC_STATE now 1 forever stdin nil]
  )
1) |[]

i_switch_to_no_role:(ipgm switch_to_no_role |[] RUN_ALWAYS 0us NOTIFY 1) [[SYNC_FRONT now 0
forever stdin nil 1]]

; -----------------------------------------------------------------------

second_goal:(pgm |[] []
  []
    (ptn (mk.success g_self_f_one_step_manipulation_task ::) |[])
  |[]
  |[]
[]
  (inj []
    f_g_self_has_interviewee_put_sln_object_somewhere
    [SYNC_STATE now 1 1 stdin nil]
  )
1) |[]

first_goal:(pgm |[] []
  []
    (ptn (mk.val self role interviewer ::) |[])
  |[]
  |[]
[]
  (inj []
    f_g_self_f_one_step_manipulation_task
    [SYNC_STATE now 1 1 stdin nil]
  )
  (inj []
    (ins second_goal |[] RUN_ONCE 0us NOTIFY)
    [SYNC_FRONT now 0 forever stdin nil 1]
  )
1) |[]

i_first_goal:(ipgm first_goal |[] RUN_ONCE 0us NOTIFY 1) [[SYNC_FRONT now 0 forever stdin nil 1]]
```