Bounded Seed-AGI

Eric Nivel[1], Kristinn R. Thórisson[1,3], Bas R. Steunebrink[2], Haris Dindo[4], Giovanni Pezzulo[5], Manuel Rodríguez[6], Carlos Hernández[6], Dimitri Ognibene[5], Jürgen Schmidhuber[2], Ricardo Sanz[6], Helgi P. Helgason[3], and Antonio Chella[4]

[1] Icelandic Institute for Intelligent Machines, IIIM
[2] The Swiss AI Lab IDSIA, USI & SUPSI
[3] Reykjavik University, CADIA
[4] Università degli studi di Palermo, DINFO
[5] Consiglio Nazionale delle Ricerche, ISTC
[6] Universidad Politécnica de Madrid, ASLAB

Abstract. Four principal features of autonomous control systems are left both unaddressed and unaddressable by present-day engineering methodologies: (1) The ability to operate effectively in environments that are only partially known beforehand at design time; (2) A level of generality that allows a system to re-assess and re-define the fulfillment of its mission in light of unexpected constraints or other unforeseen changes in the environment; (3) The ability to operate effectively in environments of significant complexity; and (4) The ability to degrade gracefully—how it can continue striving to achieve its main goals when resources become scarce, or in light of other expected or unexpected constraining factors that impede its progress. We describe new methodological and engineering principles for addressing these shortcomings, that we have used to design a machine that becomes increasingly better at behaving in underspecified circumstances, in a goal-directed way, on the job, by modeling itself and its environment as experience accumulates. The work provides an architectural blueprint for constructing systems with high levels of operational autonomy in underspecified circumstances, starting from only a small amount of designer-specified code—a seed. Using value-driven dynamic priority scheduling to control the parallel execution of a vast number of lines of reasoning, the system accumulates increasingly useful models of its experience, resulting in recursive self-improvement that can be autonomously sustained after the machine leaves the lab, within the boundaries imposed by its designers. A prototype system named AERA has been implemented and demonstrated to learn a complex real-world task—real-time multimodal dialogue with humans—by on-line observation. Our work presents solutions to several challenges that must be solved for achieving artificial general intelligence.

1 Introduction

Our objective is to design control architectures for autonomous systems meant ultimately to control machinery (like for example robots, power grids, cars, plants, etc.). All physical systems have limited resources, and the ones we intend to build are no exception: they have limited computing power, limited memory, and limited time to fulfill their mission. All physical systems also have limited knowledge about their environment and the tasks they have to perform for accomplishing their mission. Wang [3] merged these two assumptions into one, called AIKR—the assumption of insufficient knowledge and resources—which then forms the basis of his working definition of intelligence: "To adapt with insufficient knowledge and limited resources." We have adopted this definition as one of the anchors of our work, being much in line with Simon's concept of "bounded rationality" [4]. This perspective means that we cannot expect any optimal behaviors from our systems since their behaviors will always be constrained by the amount and reliability of knowledge they can accumulate at any particular point in time. In other words we can only expect from these systems their displaying of a best effort strategy.

The freedom of action entailed by high levels of autonomy is balanced by hard constraints. First, an autonomous system, to be of any value, is functionally bounded by its mission, which imposes not only the requirements the system has to meet, but also the constraints it has to respect. Second, to keep the system operating within its functional boundaries, one has to ensure that some parts of the system will never be rewritten autonomously—for example, the management of motivations shall be excluded from rewriting as this would possibly allow the transgression of the constraints imposed by the designers. In that sense, the system is also bounded, operationally, by its own architecture. Last, any implemented system is naturally bounded by the resources (CPU, time, memory, inputs) and knowledge at its disposal. For these reasons, autonomy, as we refer to it, shall therefore be understood as bounded autonomy.

Representing time at several temporal scales, from the smallest levels of individual operations (e.g., producing a prediction) to a collective operation (e.g., achieving a mission) is an essential requirement for a system that must (a) perform in the real world and (b) model its own operation with regards to its expenditure of resources (as these include time). Considering time values as intervals allows encoding the variable precisions and accuracies needed to deal with the real world. For example, sensors do not always perform at fixed frame rates and so modeling their operation may be critical to ensure reliable operation of their controllers and models that depend on their input. Also, the precision for goals and predictions may vary considerably depending on both their time horizons and semantics. Last, since acquired knowledge can never be certain, one can assume that "truth"–asserting that a particular fact holds—can only be established for some limited time, and for varying degrees of temporal uncertainty.

We target systems that operate continuously in non-resettable environments. We envision learning to be "always on" and inherent in the system's core operation. Due to the complexity of the environment and the unforeseeable nature of future events and tasks for the system, we expect not only that pre-programming of all required operational knowledge will be impossible, but that even pre-programming of substantial amounts of knowledge will be too costly. Therefore the system will need to grow autonomously from a small *seed*, containing its "drives" (i.e., mission goals and constraints) and a relatively small amount of knowledge to bootstrap learning.

Unfortunately, none of the methodologies available in the AI or CS literature are directly applicable for designing systems of this nature. For this reason we have advocated what we call a constructivist AI methodology (CAIM; [5–9]). In the main, our constructivist approach has two key objectives: (a) to achieve bounded recursive self-improvement [1] and generality and, (b) to uncover the principles for—and to actually build—systems that, given a small set of seed information, manage the bulk of the bootstrapping work on their own, in environments and on tasks that may be new and unfamiliar.

The rest of this paper describes AERA [1]: its design principles (sec. 2) to match the reality described above, its core operation (sec. 3), its methods of acquiring and representing knowledge and skills (sec. 4), and the first experimental results (sec. 5).

2 Design Principles

As we cannot assume guarantees for system down-time (after all, we are targeting high levels of operational autonomy), all activities of the system, from low-level (for example, prediction, sub-goaling) to high-level (like learning and planning), must be performed in real-time, concurrently, and continuously. Moreover, we need these activities to be executed in a way that is flexible enough to allow the system to dynamically (re-)allocate its resources depending on the urgency of the situation it faces at any point in time (with regards to its own

goals and constraints), based on the availability of these same resources, over which it may not have complete (or any) control. The approach we chose is to break all activities down into fine-grained elementary reasoning processes that are commensurable both in terms of execution time and scheduling. These reasoning processes are the execution of various kinds of inference programs, and they represent the bulk of the computing. These programs are expected to be numerous and this calls for an architecture capable of handling massive amounts of parallel *jobs*.

A running AERA system faces three main challenges: (1) To update and revise its knowledge based on its experience, (2) to cope with its resource limitation while making decisions to satisfy its drives, and (3) to focus its attention on the most important inputs, discarding the rest or saving them for later processing. These three challenges are commonly addressed by, respectively, learning, planning, and controlling the attention of the system. Notice that all of these activities have an associated cost, have to be carried out concurrently, and must fit into the resource- and knowledge budget the system has at its disposal. That is the reason why they have been designed to result from the fine-grained interoperation of a multitude of lower-level jobs, the ordering of which is enforced by a scheduling strategy. This strategy has been designed to get the maximal global value for the system from the available inputs, knowledge, and resources, given (potentially conflicting) necessities.

We emphasize that AERA has no sub-components called "learning" or "planner" and so on. Instead, learning, planning, and attention are *emergent processes* that result from the same set of low-level processes: These are essentially the parallel execution of fine-grained jobs and are thus reusable and shared system-wide, collectively implementing functions that span across the entire scope of the system's operation in its environment. High-level processes (like planning and learning) influence each other: For example, learning better models and sequences thereof improves planning; reciprocally, having good plans also means that a system will direct its attention to more (goal-)relevant states, and this means in turn that learning is more likely to be focused on changes that impact the system's mission, possibly increasing its chances of success. These high-level processes are dynamically coupled, as they both result from the execution of the same knowledge.

A system must know what it is doing, when, and at what cost. Enforcing the production of explicit traces of the system's operation allows building models of said operation, which is needed for self-control (also called meta-control). In that respect, the functional architecture we seek shall be applicable to itself, i.e., a meta-control system for the system shall be implementable the same way the system is implemented to control itself in a domain. This principle is a prerequisite for integrated cognitive control [10].

Knowledge is composed of states (be they past, present, predicted, desired or hypothetical) and of executable code, called *models*. Models are capable of generating such knowledge (e.g., generating predictions, hypotheses or goals) and are executed by a virtual machine—in the case of AERA, its executive.

Models have a low granularity for two main reasons. First, it is easier to add and replace small models than larger ones because the impact of their addition or replacement in the architecture will be less than replacement of large models. In other words, low model granularity is aimed at preserving system plasticity, supporting the capability of implementing small, incremental changes in the system. Second, low granularity helps compositionality and reuse; small models can only implement limited low-level functions and, if abstract enough, are more likely to be useful for implementing several higher-level functions than coarser models that implement one or more such high-level functions in one big atomic block. We have referred to this elsewhere as the principle of pee-wee granularity [6, 8, 11].

We also need the knowledge to be uniform, that is, encoded using one single scheme regardless of the particular data semantics. This helps to allow execution, planning, and

learning algorithms to be both general and commensurate in resource usage. Thanks to the executable nature of AERA's models, knowledge and skills have a unified representation.

3 Attention & Scheduling

A cornerstone of our approach is that cognitive control results from the continual *value-driven scheduling* of reasoning jobs. According to this view, high-level cognitive processes are grounded directly in the core operation of the machine resulting from two complementary control schemes. The first is top-down: Scheduling allocates resources by estimating the global value of the jobs at hand, and this judgment results directly from the products of cognition—goals and predictions. These are relevant and accurate to various extents, depending on the quality of the knowledge accumulated so far. As the latter improves over time, goals and predictions become more relevant and accurate, thus allowing the system to allocate its resources with a better judgment; the most important goals and the most useful/accurate predictions are considered first, the rest being saved for later processing or even discarded, thus saving resources. In that sense, cognition controls resource allocation. The second control scheme is bottom-up: Resource allocation controls cognition. Shall resources become scarce (which is pretty much always the case in our targeted system–environment–mission triples), scheduling narrows down the system's attention to the most important goals/predictions the system can handle, trading scope for efficiency and therefore survivability—the system will only pay attention to the most promising (value-wise) inputs and inference possibilities. Reciprocally, shall the resources become more abundant, the system will start considering goals and predictions that are of less immediate value, thus opening up possibilities for learning and improvement—in the future even curiosity [2].

Technically, a job in AERA is a request for processing one input by one program (for example, a model). All jobs (e.g., forward and backward chaining, explained in the next section) are assigned a *priority* that governs the point(s) in time when they may be executed. Jobs are small and uninterruptible, but might get delayed and even eventually discarded if they become irrelevant. Jobs' priorities are continually updated, thus allowing high-value new jobs to get executed before less important jobs, and old jobs to become more valuable than newer ones as new evidences constantly accumulates.[1] Thus a job priority depends on the utility value of the program and the expected value of the input. Value-driven scheduling stands at the very heart of our design and underpins our aim of looped-back adaptation and cognition. For detailed explanations of how exactly value, urgency, and priority are calculated in AERA, we refer the interested reader to [1].

4 Model-based Knowledge & Skill Representation

AERA is data-driven, meaning that the execution of code is triggered by matching *patterns* with inputs. Code refers to *models* (which constitute executable knowledge), that have either been given (as part of the bootstrap code) or learned by the system. A model encodes procedural knowledge in the form of a causal relationship between two terms. A model is built from two patterns, left-hand (LT) and right-hand (RT), both possibly containing variables. When an instance of the left-hand pattern is observed, then a *prediction* patterned after the

[1] It is worth noting that some jobs may get delayed repeatedly until their priority drops down to insignificant numbers (for example when the urgency of a goal becomes zero, i.e., when its deadline has expired) and eventually get cancelled. This is likely to happen in situations where either the CPU power becomes scarce or the number of jobs exceeds the available computing power—which is the expected fate of any system limited in both knowledge and resources.

right-hand pattern is produced. Reciprocally, when an instance of the right-hand pattern is observed (such an instance being a *goal*), then a sub-goal patterned after the left-hand pattern is produced. Additionally, when an input (other than a goal or a prediction) matches a RT, an *assumption* is produced, patterned after the LT. Notice that multiple instances of both forward and backward execution can be performed concurrently by a given model, i.e., a model can produce several predictions from several different inputs while producing several goals and assumptions, from several other inputs at the same time. Models also contain two sets of equations, called guards. These are equations meant to assign values to variables featured in the output, from the values held by variables in the input. One set of guards supports forward execution, whereas the other one supports backward execution. In our current implementation, guards are restricted to linear functions. Models form the very core of an AERA system and their operation is detailed in the next subsections.

Our approach to knowledge representation has its roots in a non-axiomatic term logic. This logic is non-axiomatic in the sense that knowledge is established on the basis of a system's experience; that is, truth is not absolute but rather established *to a certain degree* and *within a certain time interval*. In our approach the simplest term thus encodes an observation, and is called a *fact* (or a *counter-fact* indicating the absence of an observation). Terms, including facts, serve as input for (are matched against) the left-hand and right-hand patterns of models. A fact carries a payload (the observed event), a likelihood value in $[0, 1]$ indicating the degree to which the fact has been ascertained and a time interval in microseconds—the period within which the fact is believed to hold (or, in the case of a counter-fact, the period during which the payload has not been observed). Facts have a limited life span, corresponding to the upper bound of their time interval. Payloads are terms of various types, some of which are built in the executive, the most important of these being *atomic state*, *composite state*, *prediction*, *goal*, *command* to I/O, *model*, *success/failure*, and *performance measurement*. Additionally, any type can be defined by the programmer, and new types can be created by I/O devices at runtime. A composite state is essentially a conjunction of several facts, including facts whose payloads are instances of other composite states, thus allowing the creation of structural hierarchies. A composite state is a program with several input patterns, one per fact. Like models, composite states produce forward and backward chaining jobs (explained in the next subsection) when paired with some inputs.

4.1 Chaining and Hierarchy

Motivated by drives, models produce sub-goals when super-goals match their right-hand pattern, and these sub-goals in turn match other models' right-hand pattern until a sub-goal produces a command for execution by I/O devices. In parallel to this top-down flow of data, the hierarchy of models is traversed by a bottom-up data flow, originating from inputs sensed by the I/O devices that match the left-hand patterns of models, to produce predictions that in turn match other models' left-hand patterns and produce more predictions. These bottom-up and top-down flows are referred to as *forward* and *backward chaining*, respectively.

Whenever a model produces a prediction, the executive also produces a corresponding *instantiated model*: This is a term containing a reference to the model in question, a reference to the input that matched its LT and a reference to the resulting prediction. Such a reflection of operation constitutes a first-class input—i.e., an observable of the system's own operation—which is, as any other input, eligible for abstraction (by replacing values with variables bound together by guards) thus yielding a pattern that can be embedded in a model.

When a model M_0 features such an instantiated model M_1 as its LT then, in essence, M_0 specifies a post-condition on the execution of M_1, i.e., M_0 predicts an outcome that is

entailed by the execution of M_1. In case the LT is a counter-evidence of a model's execution (meaning that the model failed to execute because despite having matched an input, its pre-conditions were not met—pre-conditions are explained immediately here below), the post-condition is referred to as a negative post-condition, positive otherwise. Symmetrically, when a model features an instantiated model as its RT, it essentially specifies a pre-condition on the execution of the embedded model instance, i.e., when a condition is matched (LT), the model predicts the success or failure of the execution of a target model (the one an instance of which is the RT). More specifically, what a pre-condition means is "if the target model executes, it will succeed (or fail)." In case the RT is a counter-evidence of a model's successful execution (predicted failure), the pre-condition is referred to as a negative pre-condition, positive otherwise. Control with pre-conditions consists of ensuring that all negative pre-conditions and at least one positive one are satisfied before deciding to let the controlled model operate. This decision is made automatically by the executive by comparing the greatest likelihood of the negative pre-conditions to the greatest likelihood of the positive ones.

Planning concerns observing desired inputs (the states specified by goals) by acting on the environment (i.e., issuing commands) to achieve goals in due time in adversarial conditions, like for example the lack of appropriate models, under-performing models, conflicting or redundant goals, and lack of relevant inputs. Planning is initiated and sustained by the regular injection of drives (as defined by the programmer), thus putting the system under constant pressure from both its drives and its inputs. In our approach, sub-goals derived from goals are simulated, meaning that as long as time allows, the system will run "what if" scenarios to predict the outcome of the hypothetical success of these simulated goals, checking for conflicts and redundancies, eventually committing to the best goals found so far and discarding other contenders. Here again, goals are rated with respect to their expected value. *Simulation* and *commitment* operate concurrently with (and also make direct use of) forward and backward chaining.

4.2 Learning

Learning involves several phases: acquiring new models, evaluating the performance of existing ones, and controlling the learning activity itself. Acquiring new models is referred to as *pattern extraction*, and consists of the identification of causal relationships between input pairs: inputs which exhibit correlation are turned into patterns and used as the LT and RT of a new model. Model acquisition is triggered by either the unpredicted success of a goal or the failure of a prediction. In both cases AERA will consider the unpredicted outcome as the RT of new models and explore buffers of historical inputs to find suitable LTs. Once models have been produced, the system has to monitor their performance (a) to identify and delete unreliable models and, (b) to update the reliability as this control value is essential for scheduling. Both these activities—model acquisition and revision—have an associated cost, and the system must allocate its limited resources to the jobs from which it expects the most value. Last but not least, the system is enticed to learn, based on its experience, about its progress in modeling inputs. The system computes and maintains the history of the success rate for classes of goals and predictions, and the priority of jobs dedicated to acquire new models is proportional to the first derivative of this success rate.

A pattern extractor is a program 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 unpredicted success of a goal or the failure of a prediction. A single *targeted pattern extractor* (TPX) 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 in this case is much closer to guessing than to proving, and guesses are based on the general heuristic "time precedence indicates causality." Models thus built by the TPXs are added to the memory and are subjected to evaluation by other programs called prediction monitors. So their life cycle is governed essentially by their performance.

A TPX accumulates inputs from the target production time until the deadline of the target, at which time it analyses its buffer to produce models if needed: The TPX activity is thus composed of two phases, (a) buffering relevant inputs and, (b) extracting models from the buffer. At the deadline of the target, buffering stops, and the buffer is analyzed as follows, when the target is a goal (the procedure is similar for predictions, see below):

1. If one input is the trace of the execution of one model that predicted the goal's target state, abort—this means that the success was already predicted.
2. Remove any inputs that triggered any model execution.
3. Remove any inputs that were assembled in composite states.
4. Reorder the buffer according to the early deadlines of the inputs.
5. For each input remaining in the buffer create a TPX-extraction job, the purpose of which is to assemble a new model from the input and the target.

Shall the target be a prediction, on the other hand, step 1 would be:

1. If one input is the trace of the execution of one model that predicted a counter-evidence of the prediction's target state, abort, as the failure was already predicted.

Reaching step 5 triggers the second phase of TPX activity, where models are built from inputs found in the buffer. The construction of a new model is performed—by a TPX-extraction job—as follows, when the target is a goal:

1. The target is abstracted[2] and forms the RT of a new model (let's call it M_0). If the input assigned to the TPX-extraction job is synchronized with other inputs (that is, if their time intervals overlap), then all these inputs are assembled into a single new composite state: this new state is chosen as the LT of the model. Otherwise, the input is abstracted and forms the LT of the model. Notice that new states are identified when their parts are needed for the models being built (instead of resulting from blind temporal correlation): Using composite states as models' LT instead of just atomic states fosters the building of structural hierarchies.
2. If, in a model, some variables in the RT (or in the LT) are not present in the LT (or in the RT), then the job attempts to build guards (see below) to bind these to known variables; otherwise, stop.
3. If some variables in a model are still not bound, then if the buffer is still not exhausted, goto step 4; otherwise, goto step 5.
4. The job considers the next older input to build another model (M_1) whose RT is an instance of M_0; the unbounded variables in M_0 are passed from M_1 to M_0 as parameters of M_0. The execution of M_1 allows the execution of M_0: M_1 is a positive pre-condition on M_0. Goto step 2.
5. All models are deleted that hold variables representing deadlines that are unaccounted for, i.e., variables that cannot be computed neither from the LT or RT, nor from the model's parameters list. These models are deleted since they would produce predictions with unbound deadlines, i.e., predictions that cannot be monitored.

As with TPX-accumulation jobs, the priority of a TPX-extraction job is a function of the utility of the model that produced its target and of the incentive of learning said target. It also depends on a decay function.

[2] Here, abstraction means replacing values by variables.

4.3 Rating

In addition to the prioritization strategy, we use two ancillary control mechanisms. These come in the form of two thresholds, one on the likelihood of terms, the other on the reliability of models. When a term's likelihood gets under the first threshold, it becomes ineligible as a possible input for pattern matching; reciprocally, when the reliability of a model gets under the second threshold, it cannot process any input—it is deactivated until said second threshold is increased. These thresholds are a filtering mechanism that operates before priorities are computed (the precise operation of these is beyond the scope of the present paper). If the reliability of a model drops below a threshold THR_1 then it is phased out: In this mode the model can only create forward chaining jobs and produce silent predictions that will not be eligible inputs to the regular models (i.e., models that are not phased out). Silent predictions are still monitored, thus giving the possibility to improve to a model that was recently getting unreliable. If the reliability of a phased out model gets above THR_1, then it is not phased out anymore and resumes its standard operation. When the reliability of a phased out model drops below a second threshold THR_2 ($< THR_1$), the model is deleted as are all the programs that were created to manage its productions; the corresponding jobs are cancelled.

4.4 Reflection

Each time a model predicts, the executive produces a new term, called an *instantiated model*, that references the input, the output, and the model itself. An instantiated model is thus a trace of the execution of a model and, being the payload of a fact, constitutes an (internal) input for the system and can be the target of learning, leading to *self-modeling*.

Even though self-modeling for meta-control has not been leveraged in our demonstrator (described in the next section), this functionality has been implemented and is operational. As we have argued before [2, 6] such functionality is a necessity for a developmental system poised to adapt and make the best use of its resources. With it a system can, for example, model the ways it routinely adopts for achieving some particular goals: This consists of modeling sequences of model execution—these are observable in the form of internal inputs—and, by design, can be modeled using the existing learning mechanisms. The benefit of modeling sequences of execution can be, among others, to enable the system to compile such sequences so as to replace a set of models, which normally have to be interpreted by the executive, with a faster (but also more rigid) equivalent native machine code. Thus *self-compilation* supports a lower-level *re-encoding* of useful and reliable knowledge, which we expect will increase the scalability of AERA.

5 Experimental Results & Conclusion

In our first experiment with AERA, two humans interact for some time, allowing AERA to observe their behavior and interaction; AERA's task is to learn how to conduct the interaction in exactly the same way as the humans do, in either role of interviewer or interviewee. The knowledge given to AERA is represented as a small set of primitive commands and categories of sensory data, along with no more than a few top-level goals such as "pleasing the interviewer" (operationally defined as the interviewer saying "thank you" or asking a new question). A very detailed analysis of this experiment can be found elsewhere [1].

AERA observed real-time interaction between two humans in the simulated equivalent of a videoconference: The humans are represented as avatars in a virtual environment—each human sees the other as an avatar on their screen. Their head and arm movements are

tracked with motion-sensing technology, their speech recorded with microphones. Signals from the motion-tracking are used to update the state of their avatars in real-time, so that everything one human does is translated virtually instantly into movements of her graphical avatar on the other's screen. Between the avatars is a desk with objects on it, visible to both participants. One human is assigned the role of an interviewer, the other the role of an interviewee; the goal of their interaction is collaborative dialogue involving the objects in front of them.

The data produced during their interaction is represented as follows. Body movements are represented as coordinate changes of labeled body parts of the avatars. Each audio signal is piped to two processes: an instance of a speech recognizer (Microsoft SAPI 5.3), and to an instance of the Prosodica prosody analyzer [12]. The speech recognition is augmented with timestamps on the words produced (approximate accuracy of time-stamping typically $\pm$100ms or better), and filtered through a set of 100 allowed words (necessary due to the many false positives produced in live interaction). Words time-stamped with the estimated time-of-utterance are typically output as intermediate hypotheses between 200 and 1000ms of being uttered, with a "final guess" delivered for each audio segment after a 200ms silence is detected. The prosody analyzer produces time-stamped sound-silence boundaries (accuracy 16–32ms) and prosody information in the form of F0 (with update frequency of 6Hz; approximate accuracy of 40ms). This data is the input to AERA, streamed to it in real-time.

The task assigned to the two humans is for the interviewer to ask the interviewee to pick up and move objects around on the table, and to talk about some of their properties. We provide AERA with top-level goals, contained in the system's "seed." Testing consisted of AERA replacing either human and conduct the interaction in an identical manner, in the role of either interviewer or interviewee.

AERA learned everything that it observed in the human-human interactions which is necessary to conduct a similarly accurate and effective interaction. The socio-communicative repertoire acquired autonomously by AERA after an observation period of approximately 20 hours, has been correctly learned, with no mistakes in its subsequent application, including timing of all actions. This repertoire, including skills in either role (interviewer and interviewee), consists of:

- Correct sentence construction, correct word order.
- Effective and appropriate manual and head deictics (gesturing towards object being talked about at the right time, gazing towards it using head direction when mentioned or pointed at).
- Appropriate response generation; answer (as interviewee) and sequence of questions (as interviewer).
- Proper multimodal coordination in both interpretation and production, at multiple time-scales (interview, utterance, and sub-utterance levels).
- Turn-taking skills (avoiding overlaps, avoiding long pauses), and utterance production; presentation of content (answer/question) at appropriate times with regard to the other's behavior.
- Interview skills—doing the interview from first question to last question.

The results from the experiment shows without doubt that AREA correctly acquired and mastered correct usage of all communication methods used by the human interviewer and interviewee in the human-human condition when conversing about the recycling of the various objects' materials. The complete absence of errors in AERA's behaviors, after the observation periods in both conditions, demonstrate that very reliable models have been acquired, and that these form a hierarchy spanning at least two orders of magnitude in time. These models correctly represent the generalized relationships of a non-trivial number of

entities, knowledge which had not been provided to the system beforehand by its designers and was acquired autonomously, given the bootstrap seed initially provided.

The tasks in the experiment require AERA to learn and abstract temporal sequences of continuous events (utterances and multimodal behavior), as well as logical sequences and relationships (word sequences in sentences, meaning of words and gestures) between a number of observed data. These were acquired through a method of generalization using induction, abduction and deduction, allowing AERA to respond in real-time situations that differ from what it has seen before (the humans were not trained actors and did not repeat exactly any of their actions in any of the scenarios).

We have demonstrated an implemented architecture that can learn autonomously many things in parallel, at multiple time scales. The results show that AERA can learn complex multi-dimensional tasks from observation, while provided only with a small ontology, a few drives (high-level goals), and a few initial models, from which it can autonomously bootstrap its own development. This is initial evidence that our constructivist methodology is a way for escaping the constraints of current computer science and engineering methodologies. Human dialogue is an excellent example of the kinds of complex tasks current systems are incapable of handling autonomously. The fact that no difference of any importance can be seen in the performance between AERA and the humans in simulated face-to-face interview is an indication that the resulting architecture holds significant potential for further advances.

References

1. Nivel, E., Thórisson, K.R., Steunebrink, B.R., Dindo, H., Pezzulo, G., Rodriguez, M., Hernandez, C., Ognibene, D., Schmidhuber, J., Sanz, R., Helgason, H.P., Chella, A., Jonsson, G.K.: Bounded recursive self-improvement. Tech. Rep. RUTR-SCS13006, Reykjavik University (2013), http://arxiv.org/abs/1312.6764
2. Steunebrink, B. R., Koutnik J., Thórisson K. R., Nivel E., Schmidhuber J. (2013). Resource-Bounded Machines are Motivated to be Efficient, Effective, and Curious. In K-U Kühnberger, S. Rudolph and P. Wang (eds.), Proc. of the 6th Conf. on AGI (AGI-13), 119-129.
3. Wang, P. (2011). The assumptions on knowledge and resources in models of rationality. International Journal of Machine Consciousness, 3(1):193-218.
4. Simon, H. (1957). A Behavioral Model of Rational Choice. In Models of Man, Social and Rational: Mathematical Essays on Rational Human Behavior in a Social Setting. New York: Wiley.
5. Thórisson, K. R. (forthcoming). Methodology Matters: Constructionism Challenged, Constructivism Challenges.
6. Thórisson, K. R. (2012). A New Constructivist AI: From Manual Construction to Self-Constructive Systems. In P. Wang and B. Goertzel (eds.), Theoretical Foundations of Artificial General Intelligence. Atlantis Thinking Machines, 4:145-171.
7. Thórisson, K. R. (2009). From Constructionist to Constructivist A.I. Keynote, AAAI Fall Symposium Series - Biologically Inspired Cognitive Architectures, Washington D.C., November 5-7, 175-183. AAAI Tech Report FS-09-01, AAAI press, Menlo Park, CA
8. Nivel, E., Thrisson, K. R. (2009) Self-Programming: Operationalizing Autonomy. Proceedings of the Second Conference on Artificial General Intelligence. 2009.
9. Thórisson, K. R., Nivel, E. (2009). Holistic Intelligence: Transversal Skills and Current Methodologies. Proceedings of the Second Conference on Artificial General Intelligence, 220-221.
10. Sanz, R., Hernndez, C. (2012). Towards architectural foundations for cognitive self-aware systems. In Proc. Biologically Inspired Cognitive Architectures. Palermo, 2012. Springer.
11. Thórisson, K. R., Nivel, E. (2009). Achieving Artificial General Intelligence Through Peewee Granularity. Proceedings of the Second Conference on Artificial General Intelligence, 222-223.
12. Nivel, E., Thórisson, K. R. (2008). Prosodica Real-Time Prosody Tracker. Reykjavik University School of Computer Science Technical Report RUTR08002.