OntoAgent: Implementing Content-Centric Cognitive Models

Jesse English DRJESSEENGLISH@GMAIL.COM
Sergei Nirenburg ZAVEDOMO@GMAIL.COM
Language-Endowed Intelligent Agents Lab, Rensselaer Polytechnic Institute, Troy, NY 12180, USA

Abstract

We describe the major architectural features of OntoAgent, an agent architecture that supports content-centric modeling. Content-centric modeling places the emphasis on maintaining and dynamically expanding non-toy knowledge bases that are integrated with a combination of native and imported modules for perception, reasoning, and action. The paper motivates the rationale for and utility of this architecture, and describes the infrastructure support provided by the OntoGraph API.

1. Introduction

This paper describes OntoAgent, an architecture for implementing social agents, and OntoGraph, its knowledge management infrastructure. This introduction serves two purposes. First, we briefly motivate our choice to make OntoAgent content-centric and explain why we must use large amounts of knowledge. Second, we briefly describe what features in the OntoAgent environment support the agent's ability to make decisions in bounded time, which is a core requirement of agent architectures.

1.1 The Centrality of Content

Our longstanding interest has been developing social intelligent agents. Like other AI agents, such agents must feature perception, reasoning and action capabilities. The main distinguishing feature of social agents is the ability to communicate with other agents in natural language. We believe that language understanding and generation can only reach a human level of quality if the agent understands the content of texts and dialog turns. (In this paper, we will discuss only language understanding, as this suffices to motivate our agent architecture design decisions.) Understanding content means associating elements of language with elements of a model of the real world.[1] This, in turn, requires developing formal world models, ontologies, and semantic lexicons – knowledge resources that connect words and phrases of language with particular ontological concepts. Moreover, to approach human levels of functioning, it is essential that the above knowledge resources have broad coverage. To be a full-fledged member of a human-agent team, the agent must understand what the speaker or writer intends to convey, and represent this content in an expression (a *text meaning representation*, or TMR) in a metalanguage congruent with that of the ontological world model. Over the years, we have made significant progress on this task (see McShane and Nirenburg, in press, for the most up-to-date book-size report). Details of that work are not important

[1] This decision means, among other things, parting ways with surface string matching approaches to language processing that started with Weisenbaum's Eliza and Colby's Parry and have continued, albeit in a much more sophisticated form, till this day.

for this paper. What is germane here is our conclusion that language processing in social agents requires the involvement of the agents' general reasoning capabilities.

Reasoning is required for treating many phenomena regularly occurring in language inputs. Here, we briefly illustrate this point on the example of just two such phenomena – ellipsis and unknown words:

a) In the interests of saving time and lowering the cognitive effort of generating language, speakers typically omit much detail of a situation or task in the expectation that hearers can recreate the elided meaning using their stored knowledge.[2] To support this reasoning capability, our agents use scripts (represented as fillers of the HAS-EVENT-AS-PART property of all EVENTs in the ontology), which both aid in language understanding and guide action-related decisions.
b) Language inputs routinely include words, word senses, and (more rarely) syntactic constructions unknown to particular readers/hearers. Still, those interlocutors are capable of deriving substantial meaning from them. To recreate this capability, our agents use the content of ontological concepts underlying the known words in the input to construct expectations about the meanings of the unknown words.

In addition to the above, language-endowed agents also have the option to behave in the face of imperfect analyses the same way humans do: they can extract as much of the speaker's meaning as they can and then decide either to ask the speaker for clarifications or wait in the expectation that downstream utterances will provide that clarification. Once the agent understands the clarification, it updates its knowledge base so that the newly learned knowledge can be used in the future.

Such clarification dialogs can be viewed as instances of agent learning while executing tasks. Learning is a necessary functionality in a full-service agent architecture. In our approach, learning by instruction is used not only when clarifications are needed but also in other types of communication – recovering from errors, explaining, negotiating and advising as well as training for tasks and general instruction. Indeed, lifelong learning by instruction in natural language,[3] applicable both during task execution and via dedicated instruction sessions, is the main method of agent learning in our approach. This kind of learning must be bootstrapped by the availability of a good quality, broad coverage language understanding system that relies on a non-toy amount of knowledge. See Section 5.2 below for a brief discussion of this functionality in OntoAgent.

1.2 Mitigating Time-related Issues

Cognitive system developers have argued that systems relying on large amounts of knowledge are not realistic: "There are tradeoffs between the amount of knowledge that can be stored about a situation, the accuracy with which it can be stored, and the efficiency and accuracy with which it can be retrieved in the future." (Laird 2012, p. 32) This opinion expresses the practical concern for building real-time applications. It is also motivated theoretically by invoking the notion of perfect rationality: "it is not possible for a knowledge-rich agent embedded in a complex and dynamic environment with non-trivial novel tasks to achieve perfect rationality." (ibid.) The latter statement invokes Newell's (1990) knowledge level, at which "an agent is not described using specific data structures, representations of knowledge, and algorithms" but instead "using the content of the

[2] This observation was the seed for Roger Schank's work on scripts in the 1970s and was included in the description of the knowledge level by Newell (1990).

[3] In a large subset of domains combining language understanding with visual scene interpretation is also very helpful. While originally our work did not address such integration, the current version of OntoAgent overtly includes this functionality.

knowledge and the principle of rationality… whereby an agent selects actions to achieve its goals on the basis of the available knowledge. Achieving the knowledge level requires perfect rationality, which is computationally infeasible except when an agent has simple goals or limited bodies of knowledge." (op. cit. p.8). We do not want to forgo developing real-time applications. On the other hand, we cannot jettison the non-toy content underlying our agents. The question is: How to deal with this tradeoff? In this paper, we will just say that we accepted that agents, just like people, will make mistakes. As our agents operate in teams, they will rely on being corrected by teammates.

We want our agents to be able to make decisions in bounded time but not because their knowledge is limited and their goals are simple. One useful strategy for reducing decision-making time is deemphasizing operating from first principles. Much of the problem of assuring that all system decisions are made in bounded time is constraining heuristic search in problem spaces. Our solution is, whenever possible, to avoid search: Our agents pursue goals using stored plans. When no known plan leads to a goal, the agent's first choice is to ask for help (see above on learning by being told). Only if for some reason that cannot happen, and then only if they first assess that they have sufficient time to do so, they may opt to engage in reasoning from first principles – for example, to start generating plans on the spot through heuristic search.[4] (In some set-ups, to find a solution from an impasse, another alternative is to engage in learning by reading.)

We hypothesize that the above is the default pattern of human behavior in most situations: A social agent would prefer to ask somebody how to do something rather than explore independently. This might be viewed as a facet of the cognitive miser theory (Stanovich 2009), which posits that the well-known principle of least effort is at work with respect to cognitive effort as well as other human endeavors. This type of avoiding exploration to economize effort is only viable for language-endowed, content-centric social agents.

One additional means our agents employ to speed up processing is operationalizing the observation that hearers often understand speakers' intended meanings well before the latter finish their utterances. (We routinely interrupt one another in conversations not only because we don't have manners, but also to enhance the time efficiency of the conversation.) As the agent processes language input incrementally, it recognizes at what stage of the process the partial understanding results become *actionable*, at which time it can suspend language analysis and proceed to goal selection and downstream operations.

In addition to the above, we also seek ways to speed up processing on the knowledge representation and system engineering levels, for example, moving toward using construction-based instead of exclusively word-based lexicons. All of the above choices and operations demonstrate our attention to the need to make the agent decide in bounded time. Still, our approach is, at base, content-centric, and we are not prepared to compromise on that issue. As a result, agent applications built on the basis of our approach will have more problems with bringing response times closer to those of humans than systems that use limited amounts of knowledge.

The content-centric approach of relying on large amounts of stored knowledge about complex events saves cognitive effort – and speeds up agent processing – because a) the default method for choosing next steps does not involve search; and b) the main processing operations are checking the preconditions of available next steps and running preference heuristics for their selection. The speed of these operations relies primarily on efficient access to a variety of large knowledge bases.

[4] In our current agent applications reasoning from first principles is restricted to dealing with impasses. Since our conceptual and architectural approach readily accommodates this type of operation, incorporating (or importing) such a module in any application that may require it will not require modification to the overall system, just the reconciliation of representation formats.

Thus, in our approach the availability of content and the efficient management of knowledge base access and modification operations are the main engineering prerequisites for reducing computational complexity and, therefore, response time in agents.

The OntoAgent and OntoGraph environments have been developed to support efficient operation of agents of the kind described above.

2. OntoAgent Basics

OntoAgent is a platform for developing intelligent agent systems. It consists of (a) a network of processing modules, (b) a content module (comprised of several non-toy knowledge bases), and (c) an infrastructure module that supports system functioning, system integration, and system development activities. OntoAgent fosters developing embodied or simulated intelligent agent models and testing their performance in application systems. The latest version of the system, OntoAgent II, supports the integration of various types of perception, reasoning, and action modalities. Supported channels of perception include language understanding, actual and simulated visual perception, and simulated interoception (i.e., the perception of bodily signals). Supported action modalities include actual and simulated robotic motor action, natural language utterance generation, and simulated physiological action.

The orientation at human-level language processing necessitates solutions to many problems – such as, for example, the many facets of ambiguity resolution required for language analysis and percept interpretation – that can be safely downplayed by approaches that operate with small-scale knowledge resources. Cognitive system applications typically operate with knowledge resources that offer limited coverage of both language and the world. While its main purpose is supporting full-blown content-centric systems, for limited-coverage applications OntoAgent can be configured with just a subset of its content. For example, while the language understanding module of OntoAgent produces its best results when it has access to the situation model component of agent memory, it can also successfully generate actionable, if less precise, interpretations of language inputs without such access, relying exclusively on long-term memory components, such as the ontology and the semantic lexicon. Similarly, if speed of language analysis is more important than the depth and quality of output, the language interpretation service of OntoAgent can be configured to use only a lightweight subset of its disambiguation and inference-making capabilities.

In this paper we briefly describe the infrastructure developed to support the maintenance and functioning of the agent's knowledge resources: the long-term semantic memory; its repository of recalled past events (its episodic memory); and a situation model containing active concept instances in the agent's working memory and their interconnections. The agent's long-term memory is "first-person" in that it covers a particular agent's beliefs about the world as well as its own goal and plan inventory, its individual attitudes, biases, and heuristic decision rules and its individual memories.

OntoAgent operates at a level of abstraction that supports interoperability across the various perception, reasoning, and action modules by standardizing input and output signals for use by the in-house modules. These signals are interoperable meaning representations (XMRs), in which *X* is a variable describing the particular type of meaning representation. The types of XMRs are:

1. Perception meaning representations, for example, TMRs encoding the meaning of language inputs and outputs and VMRs for representing the meaning of visually perceived scenes or events;
2. Action meaning representations (AMRs) that encode the meaning of robotic motor actions; and

3. Mental meaning representations (MMRs) that encode mental actions and are used, e.g.,, to update the agent's situation model, to signal attention needs (e.g., to instantiate a goal instance), and to trigger any of the several truth maintenance, grounding and learning actions. (See English and Nirenburg, 2019, for details and examples.)

XMRs are encoded using the same representation format as both the representation substrate of the agent's memory system (the ontology, the episodic memory, and the situation model) and the OntoAgent decision mechanisms. All in-house components of OntoAgent are designed to accommodate XMRs. But results of each imported module must undergo semantic interpretation and be represented in XMRs. Conversely, all externally developed action modules must receive inputs in their native format, which requires a native format generation step from AMRs. OntoAgent imports results of vision processing and thus must add a dedicated visual perception interpreter for each imported vision system. On the action side, OntoGen, the language generation module of OntoAgent, is being developed in-house and generates text that is then sent to the imported speech synthesizer. A dedicated physical action generator was built to convert AMRs into procedure calls of a particular robotic system for a recent robotic application (Nirenburg et al. 2018).

3. Processing Services in OntoAgent

OntoAgent is implemented as a suite of services, each comprising a set of specific executables. The service-based infrastructure allows for architecture components to be individually developed and scaled up at different rates. As OntoAgent is intended to support embodied as well as simulated agents, a given agent system conforming to this architecture may require components that traditionally belong to either cognitive or robotic architectures.

Services in OntoAgent may be producers and/or consumers of knowledge. A **producer** of knowledge will generate data (such as an XMR). A **consumer** of knowledge will read from memory and/or input signals in order to operate. In practice, most services will be both. In what follows we present a necessarily very brief overview of the services in the latest version of OntoAgent, illustrated in Figure 1.

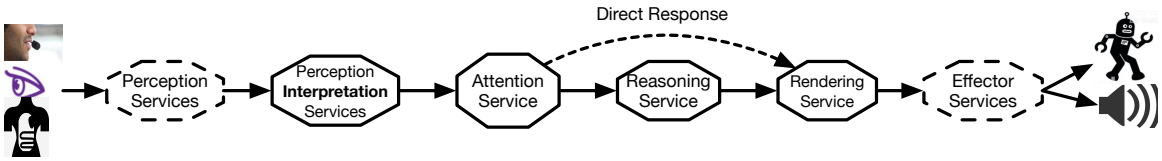

Figure 1. A top-level schematic view of the services in OntoAgent (much detail is omitted). Services in dashed-line octagons are expected to be imported.

P**erception** services are responsible for the initial processing of perceptual input. OntoAgent can work with both actual or simulated perception services. Currently, OntoAgent perception covers speech-to-text conversion, results of simulated agent's physiology (Nirenburg, McShane, and Beale, 2008) and a limited set of outputs from actual and simulated vision systems.

Interpretation services are responsible for semantically and situationally (pragmatically) interpreting the output of each of the perception services and generating representations of the meaning of the inputs in a uniform metalanguage of XMRs (cf. the discussion of OntoGraph in Sections 3 and 4 below). Each of the perception services has its own dedicated interpretation service. OntoSem

(McShane, Nirenburg, and Beale, 2016; McShane and Nirenburg, in press) is one such interpretation service that operates on input text and generates a TMR; OntoViz, on which we will report separately, is our proof of concept vision understanding system, interpreting visually categorized and temporally anchored tokens into VMRs.

The **attention** service is a dispatcher and prioritizer service. On obtaining an input from a perception interpreter or from the reasoning service[5] it decides whether this input a) warrants the creation of a new goal instance; b) warrants the generation of an "direct" response action (this mechanism simulates reactive functioning); or c) should be ignored (this simulates not paying attention to certain object and event instances in the world).

If the decision is to ignore the input, the latter is simply stored in the situation model. If the decision is to generate a **direct** response (in Kahneman's (2011) terms, a System 1, "fast thinking" process), a dedicated rule generates an AMR sent directly to the appropriate rendering service (verbal, physical or physiological) bypassing the reasoning service. For example, if the visual interpretation service generates a VMR signal describing an object starting to fall, the system will use a rule (provided specifically to cover this eventuality) to directly trigger the physical action generator to send a fall-prevention action specification to the motor action rendering service.

If in response to an interpreted input signal the attention service decides to engage the agent's reasoning service (in Kahneman's terms, a System 2, "slow thinking" process), the attention service selects an instance of a goal from the agent's goal inventory. This goal instance is added to the agent's goal agenda after which the attention service prioritizes the agenda to select the goal instance for the agent to pursue.[6] The prioritizer will operate even in the absence of input signals, as long as the agenda is not empty. The heuristics guiding the goal selection (prioritization) process involve a large number of diverse parameters from different parts of the agent's memory. Supporting efficient mustering of heuristic knowledge is a core functionality of OntoAgent as a content-centric model.

The **reasoning service** selects a plan for attaining the selected goal instance. At this time, while OntoAgent II includes several methods of recovering from planning impasses, it does not offer a full dynamic planning option. Each goal in the agent's goal inventory is associated with a set of plans that can lead to the goal. The first task of the reasoning service is to select one plan out of this set to pursue. The reasoning service has several methods of doing this. A detailed description of planning in OntoAgent is outside the scope of this paper. (Suffice it to say that the reasoning service takes advantage of the richness of the OntoAgent knowledge resources that allows it to bypass the need for planning from first principles.) Plans are hierarchically organized sequences of component events. Each of them has (a) a set of preconditions that must be checked before the action is executed, and (b) a set of effects that specify how the agent's situation model should be modified as a result of the successful performance of the action.

The **rendering** service is responsible for converting individual steps of OntoAgent plans into commands for the effectors, such as a speech generator or a robotic arm. Depending on the capabilities of the effectors, this step may itself involve planning because a single step in an OntoAgent plan may have to be unpacked into a sequence of primitive actions in the external effector service,

[5] The agent's own reasoning generates MMRs that may serve as a trigger to action, without need for an external stimulus, when the attention service becomes aware of them.

[6] This is a natural place to incorporate parallelism in the system: after all the prioritizer may be allowed to select more than one goal for processing at the same time. We will not address this issue in this paper, only note that this option is under consideration, though we will incorporate parallelism only for the situations in which people habitually demonstrate it.

such as the fine motor control system of a robotic arm. The input to a renderer is an XMR. Its output is in the language required by the corresponding effector services.

Effector services include as speech generation from text, the motor control system of a robot or the regulator of a physiological model of an agent. These services manipulate **effectors**: for language generation, a speaker; for robotic functioning, its arms, hands, wheels, etc. As an illustration, consider that when the reasoning service produces an action meaning representation (AMR) encoding the command "move to the next room", the rendering service will have to convert that into something like "move forward 8 yards, turn 90 degrees clockwise, and move forward 3 yards", and the motor action service effector will further convert that into "set wire 7 to hot for 16.9 seconds, set wire 9 to hot for 1.2 seconds, set wire 7 to hot for 6.4 seconds."

4. Memory Management

Agent memory modeling is at the core of content-centric modeling. The agent's memory in OntoAgent includes: (a) a situation model (SM) that contains the elements of the world and of the agent's internal state that are present and activated at the time of processing; (b) a long-term semantic (LTS) memory of types of entities known to the agent; and (c) an episodic (LTE) memory that includes, among other properties, spatial and temporal information about each remembered event, state, and object instance, including a time stamp indicating when the knowledge was added to agent memory.

The **memory management** component of OntoAgent facilitates access, update and management of the knowledge elements (e.g., ontological concepts, concept instances, lexicon entries, etc.) stored in the agent's memory. This component also controls what processing services in the system have access and modification privileges to particular memory spaces. System-level support for this functionality is provided by OntoGraph, described in Section 6.

Content-centric modeling emphasizes joint use of elements from different memory modules in a variety of operations, notably, decision-making by the agent. To facilitate this, the agent's memory store is organized into memory spaces. This organization, which is standard in database management, helps overall knowledge indexing and contextualizes knowledge elements. All agent services have unrestricted access to all the components of the agent's memory, although in practice, each process only uses a subset of spaces. The content-oriented memory spaces in the current version of OntoAgent (in alphabetical order) are listed below:

- **AGENDA:** part of SM, AGENDA contains elements related to agenda processing; this includes the agenda, goal/plan/step instances, and miscellaneous supporting elements
- **ENV:** part of SM, ENV contains object instances in the current state of the spatial environment; something that is present in the environment but may be stored in another space may be cross-referenced here (such as a known agent, who might be stored in long-term episodic memory)
- **GOALS:** part of LTS, GOALS contains the inventory of goal types known to the agent
- **I(X)MR:** shared by LTE and SM, I(X)MR contains all of the input XMR signals interpreted from perception; in practice, each XMR is contained in a dynamic subspace with its own index (for example, TMR#123)
- **LEX:** part of LTS, LEX contains the entries from the agent's semantic lexicon (at this time, only English is covered)
- **LTE:** part of LTE, the LTE space contains long-term episodic memory elements that are not XMR signals

- **ONT:** part of LTS, ONT contains ontological concepts known to the agent
- **OPT:** part of LTS, OPT contains the entries from the agent's semantic opticon (connecting types of visually observable objects and events with the ontology)
- **O(X)MR:** shared by LTE and SM, O(X)MR contains all of the output XMR signals generated by attention and reasoning; in practice, each XMR is contained in a dynamic subspace with its own index (for example, AMR#456)
- **SELF:** part of SM, SELF contains the agent's current view of itself (@SELF.AGENT.1), specifically, the status of its effectors, active background executables, and current physical and mental states
- **SM:** part of SM, the SM space contains the remainder of the current situation model for the agent, holding instances that do not belong in ENV or ???; this space is used to store observed events, states, and non-local (physically) objects
- **SYS:** part of LTS, SYS contains all of the executables known to the agent (such as heuristic rules or direct response rules)
- **???:** part of SM, ??? contains as yet ungrounded instances that are expected to be co-referenced or grounded; once grounded they are moved to another space, as appropriate; this space is also used to store elements generated during counterfactual reasoning

Dividing memory into spaces is very important in a content-centric architecture. Knowledge in OntoAgent is common for all its services. OntoAgent-based application systems are configured to include different inventories of the available services. The knowledge stratum of OntoAgent is designed to support any application system configuration. So it is critical that all services have well-defined expectations about where to find knowledge required for its operation. For example, any service expects to find all of the known ontological concepts in the ONT space, or all of the known goals in the GOALS space. Conversely, any service that generates knowledge stores it in its appropriate space, thus assuring this new data available to the rest of the system.

In addition to the content-oriented spaces defined above, OntoAgent also maintains infrastructure-oriented spaces whose purpose is to support indexing and storage. For example, OntoAgent uses the IO space to index input and output XMRs and the EXE space both for bookkeeping and to store entities that are not part of its "conscious" functioning, such as direct response signals or states in a simulated physiological model. OntoAgent also allows system developers to declare new spaces to cover any additional needs of specific applications. Providing convenient access to the creation and maintenance of memory spaces has been one of the objectives in the development of OntoGraph (see Sections 6 and 7 below).

5. Selected Functionalities and "Self-Improvement" Services in OntoAgent

Implementation of a particular OntoAgent-based agent involves, minimally, implementing a subset of the services guaranteeing throughput. Several different configurations have been used in application systems we developed to-date. In what follows, we briefly describe a subset of functionalities that the current version of OntoAgent offers to all services; these can be adopted as needed by individual implementations.

5.1 Service functionalities

Situation model consolidation: grounding. Grounding is the process of identifying object or event instances in memory referring to the same ontological instance and merging the information into a single instance. It is an extension of the reference resolution task in language understanding. The most frequent type of grounding in a robotic application is associating an object instance mentioned in an utterance with the visually perceived object instance. Nirenburg et al. (2018) and Wood (2019) include descriptions of the OntoAgent approach to the cross-module grounding of objects and events from visual perception and language understanding. Another type of grounding connects elements of XMRs with remembered object and event instances stored in LTE. This allows the agent's to reason using all the knowledge about a concept instance available in its long-term episodic memory, not just that part of this knowledge that is conveyed by an XMR. Work on enhancing OntoAgent's approach to and coverage of grounding phenomena is ongoing. In architectural terms, grounding is a memory consolidation process commonly controlled by the attention service.

Operating Modes. Each service in OntoAgent can operate in one or both the **signaled** or **cycled** control mode. In a **signaled** mode, the module expects and anticipates signals to trigger processing and might otherwise be idle. A service defines which signals it expects (typically a type of XMR, optionally with specified content). When such a signal is received, the service uses it as input for executing appropriate processing elements. In a **cycled** mode, a service operates on a continuous cycle, repeatedly polling relevant areas of memory looking for updates. As a result, OntoAgent-based systems may employ a mixed pipeline and blackboard control architecture.

Signal Priority. Signals sent to a service can be assigned priorities by their senders; priority can suggest that the receiving service handle the signal at its convenience, as soon as possible, or even request that the service interrupt its current task. A common use for attaching priority to signals is in a multi-scheduler agent – one scheduler may need to overwrite the resource needs of another in a given situation, and can suggest this to the renderer and effector services.

5.2 "Self-Improvement" services

Figure 1 illustrates a basic, task-oriented configuration of OntoAgent. Such a configuration does not fully support content-centric modeling. The latter assumes the long-term perspective where agents are supposed to persist beyond particular task runs and be amenable to perform a variety of tasks. The overall objective is to model the constant accumulation of specific memories, facts and beliefs and life-long learning that is characteristic of humans. Accordingly, OntoAgent adds services to support these functionalities.

Episodic memory consolidation. Functionality for manipulating memory into, out of, and within its LTE is offered by OntoGraph, and is intended to emulate human memory management processes, such as memory consolidation and forgetting. and combining elements inside episodic memory (which is an extension of the notion of grounding). Nirenburg and Wood (2017) describes an example of the application of this service for consolidating in LTE of two different instances of a complex event. The result of this consolidation was then used (by the knowledge learning service described next) to learn an improved ontological representation of this complex event. This, in turn, improved the agent's ability to attain the goal for which this complex event was listed as the default plan.

Knowledge learning. OntoAgent is content-centric, so maintenance of knowledge is a core objective in it. Agents are supplied with initial stores of knowledge, including ontology, lexicon, opticon, goal inventories, etc. As an agent experiences its environment and operates in it, it prompts itself to learn additional knowledge. This is typically done by reasoning over available knowledge. Triggers for learning include, among others, TMRs of definitional and descriptive utterances or

following up on the results of "emergency" recovery processing that a service when encountering any of a number of anomalies due to knowledge lacunae. Our past work along these lines is described in Nirenburg, Oates, and English (2007) and McShane, Blissett and Nirenburg (2017). An experiment in learning complex events (ontological scripts) "hands-on", when accompanied by instruction in natural language by a human teammate, is reported in Nirenburg and Wood (2017). A dovetailing direction of work, which involves the mixed-initiative acquisition of structured knowledge about low-density languages in support of NLP applications, is described in McShane et al. (2002).

6. An Example of Agent Operation

We illustrate a small subset of task-related operation in an OntoAgent-based application system for assembling furniture in collaboration with a human. The system integrates native services of OntoAgent with perception and action capabilities of a robotic system (Roncone, Mangin, and Scassellati, 2017). The immediate purpose of this example is to illustrate how the system generates and uses knowledge. Due to space constraints, we show only those properties of the knowledge entities that are directly relevant to the presentation of the example. Additionally, the timing of the steps is not discussed, and all effectors are assumed to be available at the time they are needed.

The "native" services in this system include memory management, vision interpretation, text interpretation (OntoSem), attention (dispatcher and prioritizer), reasoning, and visual and verbal rendering services. Services imported from the robotic system include a vision preprocessor service ("camera") that identifies objects and their properties (position, color, shape, size), a speech-to-text service ("microphone"), and a motor action service ("hand"). At the start of the example run the agent's agenda is empty.

Step 1. The camera service processes a scene and outputs 1) an instance of a human, with a particular facial features profile; 2) a red screwdriver on a table directly in front of the agent; and 3) a green screwdriver on a shelf far to the left. The vision interpretation service converts the input into a set of VMRs and stores it in the agent's SM. The attention service notices the augmentation of the SM, disregards grounding in the SM space – since the latter does not (yet) contain anything but the current input – but carries out grounding against the LTE because the agent always attempts to recognize any humans that it sees. So, the system retrieves from the LTE the remembered instance of HUMAN, @LTE.JAKE.1, that matches on the facial feature profile. It is then unified with the just generated VMR by adding the current spatial coordinates supplied by the camera service. At the end of this step, the following VMRs are present:[3]

```
@VMR.JAKE.1
  LOCATION [4, 0, 5]
  <a set of properties from @LTE.Jake.1>
@VMR.TABLE.1
  LOCATION [0, 0, 2]
@VMR.SHELF.1
  LOCATION [-10, 0, 8]
@VMR.SCREWDRIVER.1
  COLOR    RED
  ON-TOP-OF  @VMR.TABLE.1
  LOCATION [0, 3, 2]
@VMR.SCREWDRIVER.2
```

[3] The LOCATION property values in this example are simplified encodings of relative distances and directions of objects from the agent as the source, in feet)

```
  COLOR    GREEN
  ON-TOP-OF  @VMR.SHELF.1
  LOCATION [-10, 5, 8]
```

Step 2. The attention service does not instantiate any goals for the agent because in this particular system the agent does not initiate activity. (This functionality was included in other OntoAgent-based systems, e.g., in the MVP project (Nirenburg, McShane, and Beale, 2008).)

Step 3. Jake says, "Let's build a chair." The microphone service converts the audio input into text and sends it to the OntoAgent language interpretation service, OntoSem. OntoSem makes use of all the components of the OntoAgent memory (LTS, LTE and SM) to carry out a large number of operations such as lexical and referential ambiguity resolution to generate a text meaning representation for this input.[4] OntoSem works incrementally. Having consumed the lexical token *Let's*, it retrieves it's meaning from the lexicon: an instance of the REQUEST-ACTION speech act (say, @TMR.REQUEST-ACTION.13 – which assumes that the agent has twelve instances of the ontological concept REQUEST-ACTION recorded in its LTE) that refers to an as-yet-unspecified action (say, @TMR.ACTION.37) whose AGENT case role filler is a set whose members are co-referential with the fillers for the AGENT (speaker) and the BENEFICIARY (hearer) case roles of @TMR.REQUEST-ACTION.13. Ontological knowledge establishes that fillers of these roles must be constrained to HUMANs or ARTIFICIAL-INTELLIGENT-AGENTs. So, the system knows about agents of @TMR.ACTION.37 even before it establishes what kind of action it actually is!. So, the system identifies @LTE.JAKE.1 as the speaker. To detect the filler (or fillers!) of the beneficiary role of @TMR.REQUEST-ACTION.13, the system inspects the SM to establish what humans or intelligent agents are present there. It so happens that, in addition to the speaker, it is only the agent itself, @SELF.AGENT.1. OntoSem next consumes the remainder of the input, in the process concretizing @TMR.ACTION.37 to @TMR.BUILD-CHAIR.5. The resulting TMR (presented here in a simplified display format) is recorded in SM:

```
@TMR.REQUEST-ACTION.13
  TMR-ROOT True
  AGENT    @LTE.JAKE.1
  THEME    @TMR.BUILD-CHAIR.1
  BENEFICIARY  @LTE.JAKE.1 @SELF.AGENT.1
@TMR.BUILD-CHAIR.5
  AGENT    @TMR.SET.1
  THEME    @TMR.CHAIR.87
@TMR.SET.1
  ELEMENTS @LTE.JAKE.1 @SELF.AGENT.1
```

Step 4. The attention service detects the above TMR. Since the agent's model of self contains the information that the speaker has authority over the agent, the attention service decides that attention is due to this TMR. As a result, the goal of having built a chair, which is represented as

[4] The operation and capabilities of OntoSem have been described in great detail in many publications. See McShane and Nirenburg (in press) for the latest detailed presentation. A shorter treatment available in McShane and Nirenburg (2019). Here, we trace its operation in a rather superficial way. (Readers uninterested in details of language analysis may skip to the end of this step.) The difficult issues that OntoSem tackles include lexical and referential ambiguity resolution, reconstructing meanings that are intended but are not overtly stated in language inputs, processing implicatures and non-literal language, and interpreting sentence fragments. Research in these areas is ongoing. Architecturally, the most interesting facet of the interaction between OntoSem and OntoAgent is that, for its last, "deepest" stage of language understanding, OntoSem uses the complete set of OntoAgent knowledge resources – including the ontology, the episodic memory, and the situation model.

the state listed in the EFFECT property of BUILD-CHAIR.5, the existence of CHAIR.87. So, @AGENDA.CHAIR.87 is added to the AGENDA space of SM.

Step 5. At this point, the prioritizer subservice of the attention service prioritizes each goal instance on the agenda, based on a variety of parameters such as standing priorities, expected costs, resource availability, etc. In the example, the choice is easy: only one goal instance is present.

Step 6. The reasoning service receives the goal instance @AGENDA.CHAIR.87. The goal has a single plan associated with it, which happens to be the complex event BUILD-CHAIR.[7] the HAS-EVENT-AS-PART slot of the BUILD-CHAIR event contains a sequence (actually, a hierarchical transition network, HTN) of subevents (plan steps), the first of which (illustrated below) is for the agent to affix two dowels together using a screwdriver (the precondition and effect are glossed):

```
@AGENDA.AFFIX.1
  AGENT          @SELF.AGENT.1
  THEME          @???.DOWEL.1
  DESTINATION    @???.DOWEL.2
  INSTRUMENT     @???.SCREWDRIVER.1
  PRECONDITION   <the AGENT is holding the INSTRUMENT>
  EFFECT         <a BRACE exists, made of the THEME and DESTINATION>
```

The precondition states that the agent of AFFIX, @SELF.AGENT.1, must be holding the INSTRUMENT of AFFIX, an unspecified screwdriver, @???.SCREWDRIVER.1. The description of the current state of the agent in the SELF space does not state that this condition is met. As a result, a subgoal is created describing the (counterfactual) state of @SELF.AGENT.1 holding @???.SCREWDRIVER.1, the unspecified screwdriver that is the INSTRUMENT of the parent AFFIX event. The reasoning service then searches the LTS for a suitable plan, selects @ONT.PICK-UP (its EFFECT satisfies the subgoal):

```
@ONT.PICK-UP
  AGENT    @ONT.ROBOT
  THEME    @ONT.PHYSICAL-OBJECT
  EFFECT   <the AGENT is holding the THEME>
```

and instantiates it with the parameters already specified for the parent goal: [6]

```
@AGENDA.PICK-UP.1
  AGENT  @SELF.AGENT.1
  THEME  @???.SCREWDRIVER.1
```

Step 7. In order for a plan to be executable, all of its props must be grounded to known objects. In our example, in order to pick up a screwdriver, the agent must select a particular screwdriver to pick up. The SM contains two screwdrivers (see Step 1 above). The reasoning service must at this point decide which one to pick up. In the example, the decision is made on the basis of the single parameter of relative distance to the screwdrivers. @???.SCREWDRIVER.1 is subsequently grounded to the nearest screwdriver instance: @???.SCREWDRIVER.1 becomes @ENV.SCREWDRIVER.1 everywhere it is referenced in the environment, and the plan is updated on the agenda:

```
@AGENDA.PICK-UP.1
  AGENT  @SELF.AGENT.1
  THEME  @ENV.SCREWDRIVER.1
```

[7] Notice that this version of the reasoning service does not need to use the presence of an instance of BUILD-CHAIR in the input TMR to choose a plan.

The attention service keeps the current goal in focus (no other goals contend for the agent's attention), so the reasoning component proceeds to the next processing step, which is to determine a primitive action that can be the first one in the complex event (plan) PICK-UP. The content of the HAS-EVENT-AS-PART slot of PICK-UP is retrieved from the ontology module of the agent's LTS and instantiated on the agenda as follows (once again, represented in a simplified way):

```
@AGENDA.MOVE.1
    PRECONDITION   <AGENT is within arm's length of DESTINATION>
    AGENT          @SELF.AGENT.1
    DESTINATION    @AGENDA.PICK-UP.1.THEME.LOCATION
    INSTRUMENT     @SELF.HAND.1
@AGENDA.GRASP.1
    PRECONDITION <INSTRUMENT is within grasp range of THEME>
    AGENT        @SELF.AGENT.1
    THEME        @AGENDA.PICK-UP.1.THEME
    INSTRUMENT   @SELF.HAND.1
```

The first subevent has a precondition: the agent must be within "arm's length" of the destination – in this case, we know this to be true, as the agent selected the nearby screwdriver as the object to grasp. Had the agent selected the other screwdriver, another step of subgoaling would be required to move (the agent) closer to the screwdriver.

Step 8. The current goal is still in focus. So, the reasoning service next processes the first action of the selected plan. It does so by converting this action into an AMR:

```
@AMR.MOVE.1
  AGENT        @SELF.AGENT.1
  DESTINATION  @ENV.SCREWDRIVER.1.LOCATION
  INSTRUMENT   @SELF.HAND.1
```

Step 9. The rendering service concretizes the AMR to include specific data and converts the format to conform with expectations of the effector service (a robotic arm movement system), producing the following signal:

```
@AMR.MOVE.1
  AGENT      @SELF.HAND.1
  DIRECTION  <forward>
  DISTANCE   <2 feet>
```

Step 10. The effector receives the above AMR. Since the effector (the robotic arm) is not in use at the time, there is no need for scheduling (prioritizing) its use. As a result the effector service proceeds to convert the above AMR into a sequence of robotic commands (not illustrated in detail in this paper). Executing these commands will cause the arm to move forward a short distance.

Step 11 and Beyond. The agent has now completed the very first step in its plan; the attention service records this and, if the attention service does not force a change in what goal to pursue, proceeds to the next step in the plan, which will activate the grasping mechanism of the hand.

7. Infrastructure for Memory Management: OntoGraph

The memory management service in OntoAgent requires support for multiple data views and flexible retrieval across knowledge of different types and provenance. To respond to this need we created OntoGraph, a knowledge base API that imposes a single interoperable metasyntactic format on all knowledge elements in the system; connects elements stored in different components of the

agent's memory; implements efficient interaction with underlying datastores; and facilitates the development of ergonomic developer and knowledge engineering environments. OntoGraph extends the functionality found in industry standard database systems in order to a) present a view of data that is amenable for use in a graph database, b) natively support inheritance and allow knowledge to c) be organized dynamically along various dimensions. These three core functionalities are detailed below.

The OntoGraph API represents a graph database. The nodes in OntoGraph are frames, which are collections of properties with their value sets and unique identifiers. Some of the properties are relations, which point to other frames. This enables any data stored in OntoGraph to be viewed and queried as directed networks of frames. OntoGraph has built-in graph querying capabilities which support determining similarities and associations across elements of the agent's memory. In particular, the OntoGraph API supports the representation of inheritance hierarchies: each frame in OntoGraph can specify one or more frames to inherit from. Inheritance is implemented by making all property values from the ancestry tree locally available in each frame. OntoGraph further allows for a variety of inheritance-management strategies and views. For example, locally defined fillers override inherited ones by default, but non-default views can optionally show inherited values as well. Figure 2 illustrates inheritance views.

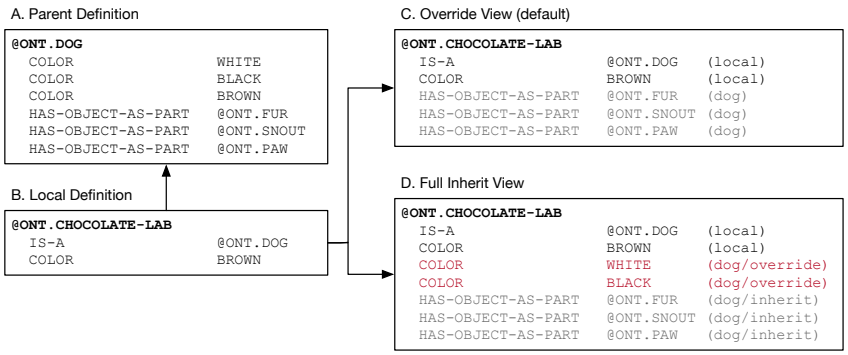

Figure 2: An example of inheritance views.

OntoGraph organizes frames into spaces (see Figure 3). Each space can be used for organization or indexing, and their inventory and use can be defined either at the OntoAgent model level (see above) or in a specific application. OntoGraph supports a variety of functionalities: e.g., frames can be generated in spaces, moved between spaces, and removed from spaces. Relations can also cross space boundaries if desired. In addition to the above primary functionalities, the OntoGraph API also defines a large collection of convenience methods, such as: *inverse relations*: all relations in the graph are directed, but search and lookup can follow incoming (e.g., inverse) relations if specifically requested; *frame consumption*: one frame can fully absorb the property values of another, removing the consumed frame from the graph – a useful tool when combining memory elements; *ancestry lookups*: detecting parents, ancestors, children, descendants, and siblings are common operations in the API; and *query pipelines*: search results for property values, ancestry, and graphing paths can be pushed through a pipeline of transformations and subqueries to produce specific results.

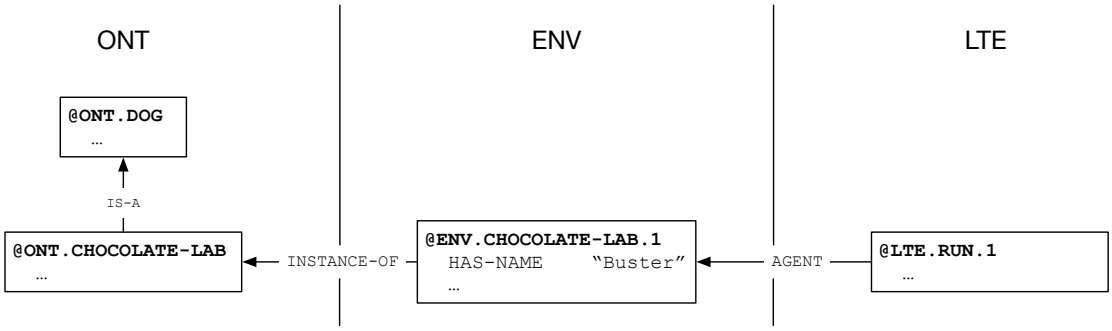

Figure 3. A view of connected frames in different spaces. The agent's ontology (ONT) knows that chocolate Labradors are types of dogs. Its situation model (ENV) contains an instance of a chocolate Labrador, whom the agent recognizes as the agent of a particular running event. The running happened at an earlier time and is stored in the agent's episodic memory (LTE).

8. OntoGraph Implementation

OntoGraph is implemented as an API with a host of functions supporting the above core behaviors. While some existing knowledge base systems, e.g., SCONE (Fahlman 2014), share many high-level functionalities with OntoGraph, we wanted the OntoGraph architecture to both natively support OntoAgent, and importantly to allow a greater flexibility of data storage: having an API that wraps various implementations of industry-standard data storage options does this, while offloading the heavy work of low-level data management to database systems that are better suited to it.

An OntoGraph implementation, called a driver, is a wrapper around an existing datastore (such as a relation database, graph database, NOSQL database, etc.) that conforms to OntoGraph's API. At the time of writing, such wrappers have been developed for SQLite (https://sqlite.org/index.html) and PostgreSQL (https://www.postgresql.org), with additional ones to be added on an as-needed basis to provide the best fit for a specific application. The API functions in a standard manner irrespective of the underlying datastore. This allows developers to effortlessly configure different agent system components to use different databases.

A critical element addressed in the implementation of OntoGraph was usability. In contrast to other knowledge bases – for example, SCONE, which is written in LISP, or OWL (https://www.w3.org/2001/sw/wiki/OWL), a semantic web language – the OntoGraph API is written in Python in such a way that using it would feel like using common Python builtins. For example, the OntoGraph Frame object has near-identical behavior to Python's default dict object, though it adds automatic inheritance, relation crawling, and data persistence, among other operations. This emphasis on usability promises faster ramp-up times for both developers and knowledge acquirers. Implementation in Python also allowed for a variety of other benefits: property values (frame slot fillers) can be nearly any data type handled by any modern programming language, and they can, importantly, include executable code as well. With its object-oriented approach, OntoGraph allows for individual frames to extend the Frame class, thus giving knowledge elements custom programmatic capabilities.

To enhance the efficiency of domain specialists and knowledge acquirers, OntoGraph has a built-in extension called OntoLang. OntoLang is a custom language used for acquiring and editing knowledge as well as for writing knowledge base queries. OntoLang is essentially a reskinning of

OntoGraph's API functionality, and its expressions are, in fact, parsed into those same API actions. What it contributes is a visualization for simplifying the knowledge acquisition task, and is designed for use by knowledge engineers who are not programmers.

OntoGraph is an evolving system, with a variety of features planned to improve its functionality, including the expansion of contextual markers, the support of additional custom data types, the expansion of functional elements beyond procedural attachments, and more. We plan to release a version of OntoGraph for the use by the community in the near future.

9. Discussion

Our work shares goals and issues with many other efforts. Systems and architectures such as DIARC (Scheutz et al., 2007), Icarus (Choi and Langley, 2018), Rosie (Mohan and Laird, 2014), Arcadia (Bridewell and Bello, 2016) and many others all offer salient points of comparison and have, to greater or smaller degree, addressed integration and infrastructure issues. Fundamental comparison of these and other systems is not feasible in this space. We will briefly review several concerns relating to integration of perception, reasoning and action functionalities raised in Scheutz, Harris, and Schermerhorn's (SHS, 2013) discussion of integrating cognitive and robotic architectures.

SHS correctly criticize approaches integrating existing robotic architectures into cognitive architectures for treating time in abstract agent operation cycles instead of the real-world time intervals with which robotic architectures operate. The above example of OntoAgent performance was indeed presented without a reference to real time, though it notes the presence of effector availability check in Step 10. The integration approach described in this paper facilitates the timing of operations on the basis of availability of resources (such as effectors). Since the attention service of OntoAgent operates over both external (percepts) and internal (thoughts) inputs, it facilitates asynchronous operation of services and overcomes the rigidity of the traditional "sense-think-act" cycles that SHS correctly criticize as stifling for a comprehensive agent system. SHS additionally point out that robotic systems may do things of which cognitive systems might not be aware. As we already mentioned, the OntoAgent approach to integration licenses a direct connection between perception and action (though this will require adding a rule set for connecting raw input signals with particular effector signals), which will model the "subconscious" behavior that SHS describe. Finally, SHS point out that robotic systems are not well equipped to deal with goals. In the OntoAgent approach, the perception and action components, indeed, are not expected to relate to "conscious" goal-oriented processing, but the latter is connected to the outputs of perception and inputs of action components through interpreters and generators, offering a clear division of labor. The development of interpreters and generators is a central part of work in OntoAgent. One of the directions of our future work is developing parameter sets for assessing how difficult it would be to develop an interpreter for a particular candidate perception system for OntoAgent and a generator for a particular action system.

To sum up, in Figure 4 we present the content-centric view of the architecture that is "native" to OntoAgent. Our intent is to juxtapose it against the process-centric view in Figure 1. Content-centric modeling concentrates on indexing, integrating and managing knowledge elements that are produced by system services; supplying knowledge elements to these services that assure their normal operation; and generating new knowledge for future use by these services. The organization of knowledge in spaces helps with indexing and enhances efficiency. In this paper we discussed only

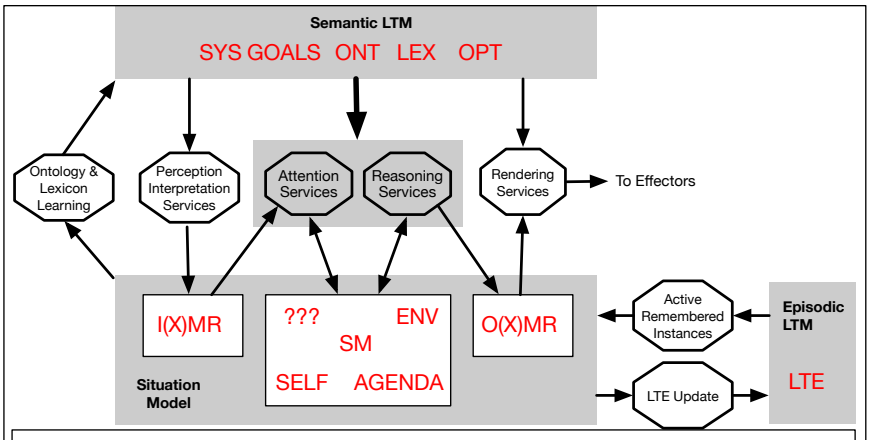

Figure 4. The content-centric view of OntoAgent architecture. Spaces within the three parts of the agent's memory are marked in red. Bringing into SM the knowledge about known active concept instances involves grounding against LTE and supports task-oriented processing. Note the three services that were not included in Figure 1. LTE update and ontology and lexicon learning services are integral parts of OntoAgent.

the content-oriented spaces in the OntoAgent environment. The latter also uses a number of infrastructure-oriented spaces that help with a variety of "bookkeeping" tasks behind the scenes. OntoAgent also allows system developers to define and seamlessly incorporate into the infrastructure new spaces as needed.

This paper does not describe the content of OntoAgent's long-term semantic memory in any detail. The two core static knowledge resources in OntoAgent – the ontology and the ontological semantic lexicon – are most comprehensively described in Nirenburg and Raskin (2004), McShane, Nirenburg and Beale (2005), and McShane and Nirenburg (in press). The particulars of agent models are described in McShane (2014). We will describe the opticon and the management of heuristic rules in future reports.

Acknowledgements

This research was supported in part by Grant N00014-17-1-2218 from the U.S. Office of Naval Research. Any opinions or findings expressed in this material are those of the authors and do not necessarily reflect the views of the Office of Naval Research. Many thanks also to the anonymous referees for their excellent critique.

References

Bridewell, W., and Bello, P. 2016. A theory of attention for cognitive systems, Advances in Cognitive Systems, 4:1-16/

Choi, D., and Langley, P. 2018. Evolution of the ICARUS cognitive architecture. *Cognitive Systems Research*, 48: 25-38. Elsevier.

English, J., and Nirenburg, S. 2019. XMRs: Uniform semantic representations for intermodular communication in cognitive robots. *ACS Workshop on Cognitive Robotics*. Cambridge, MA, August.

Fahlman, S. 2014. Scone User's Guide. http://www.cs.cmu.edu/~sef/scone/Scone-User.pdf

Kahneman, D. 2011. *Thinking: Fast and slow*. New York: Farrar, Straus and Giroux.

McShane, M. 2014. Parameterizing mental model ascription across intelligent agents. *Interaction Studies, 15(3)*: 404-425.

McShane, M., Blissett, K., and Nirenburg, I. 2017. Treating unexpected input in incremental semantic analysis. *Proceedings of the Fifth Annual Conference on Advances in Cognitive Systems.*

McShane, M., Nirenburg, S., Cowie, J., and Zacharski, R. 2002. Embedding knowledge elicitation and MT systems within a single architecture. *Machine Translation* 17(4): 271–305.

McShane, M., Nirenburg, S., and Beale, S. 2005. An NLP lexicon as a largely language independent resource. *Machine Translation,* 19(2): 139–173.

McShane, M., Nirenburg, S., and Beale, S. 2016. Language understanding with Ontological Semantics. *Advances in Cognitive Systems* 4: 35-55.

McShane, M., and Nirenburg, S. 2019. Context for language understanding by intelligent agents. *Applied Ontology* 14: 415-449.

McShane, M., and Nirenburg, S. (in press). *Linguistics for the Age of AI*. MIT Press.

Mohan, S., and Laird, J. E. 2014. Learning goal-oriented hierarchical tasks from situated interactive instruction. In *AAAI*, pp. 387–394.

Nirenburg, S., McShane, M., and Beale, S. 2008. A simulated physiological/cognitive "double agent". In Beal, J., Bello, P., Cassimatis, N., Coen, M. and Winston, P. (Eds.), Papers from the AAAI Fall Symposium "Naturally Inspired Cognitive Architectures". AAAI Technical Report FS-08-06. Menlo Park, CA: AAAI Press.

Nirenburg, S., McShane, M., Beale, S., Wood, P., Scassellati, B., Mangin, O., and Roncone, A. 2018. Toward human-like robot learning. *Proceedings of the Twenty Third International Conference on Natural Language and Information Systems* (pp. 73-82).

Nirenburg, S., Oates, T., and English, J. 2007. Learning by reading by learning to read. *Proceedings of the International Conference on Semantic Computing*. San Jose, August.

Nirenburg, S., and Wood, P. 2017. Toward human-style learning in robots. *AAAI Fall Symposium on Natural Communication with Robots*.

Nirenburg, S., McShane, M. & English, J. (submitted). Content-centric computational cognitive modeling. Submitted to Advances in Cognitive Systems 2020.

Roncone, A., Mangin, O., and Scassellati, B. 2017. Transparent role assignment and task allocation in human robot collaboration. *Proceedings of International Conference on Robotics and Automation*. Singapore.

Scheutz, M, Schermerhorn, P., Kramer, J., and Anderson D. 2007. First steps toward natural human-like HRI. *Autonomous Robots*, 22: 411-423.

Scheutz, M., Harris, J., and Schermerhorn, P. 2013. Systematic integration of cognitive and robotic architectures. *Advances in Cognitive Systems* 2: 277-296.

Stanovich, K.E. 2009. What Intelligence Tests Miss: The Psychology of Rational Thought. Yale University Press.

Sun, R. 2017. The CLARION cognitive architecture: Towards a comprehensive theory of the mind. In: S. Chipman (ed.), *The Oxford Handbook of Cognitive Science*. Oxford University Press, New York.

Wood, P. 2019. Cross-Modal Instance Grounding for Intelligent Agent Systems. Unpublished MSc Thesis, Department of Computer Science. Rensselaer Polytechnic Institute.