Application-Independent and Integration-Friendly Natural Language Understanding

Manfred Eppe[1], Sean Trott[1], Vivek Raghuram[2], Jerome Feldman[1], and Adam Janin[1]

[1] International Computer Science Institute, Berkeley, California, USA
{eppe, seantrott, feldman, janin}@icsi.berkeley.edu
[2] University of California at Berkeley, USA
vivek.raghuram@berkeley.edu

Abstract

Natural Language Understanding (NLU) has been a long-standing goal of AI and many related fields, but it is often dismissed as very hard to solve. NLU is required complex flexible systems that take action without further human intervention. This inherently involves strong semantic (meaning) capabilities to parse queries and commands correctly and with high confidence, because an error by a robot or automated vehicle could be disastrous. We describe an implemented general framework, the ECG2 system, that supports the deployment of NLU systems over a wide range of application domains. The framework is based on decades of research on embodied action-oriented semantics and efficient computational realization of a deep semantic analyzer (parser). This makes it linguistically much more flexible, general and reliable than existing shallow approaches that process language without considering its deeper semantics. In this paper we describe our work from a Computer Science perspective of system integration, and show why our particular architecture requires considerably less effort to connect the system to new applications compared to other language processing tools.

1 Introduction

Natural Language Understanding (NLU) is an important open problem in communication with future and present autonomous systems and other application domains. However, NLU is also very difficult because it requires a computational system to understand the complex cognitive semantics of natural human language. Of course, there exist several robots and other virtual agents that are equipped with some kind of natural language interface, such as Apple's Siri, Microsoft's Corona and Google Now. However, these products are limited in their understanding because they perform relatively shallow language interpretation, often based only on learned or predefined keyword-based input templates that trigger certain actions. In the literature, such shallow approaches are usually referred to as *Natural Language Processing* (NLP). Such systems are appropriate when the results are for human consumption rather than direct action. A much more serious issue is the interpretation of direct action commands for robots, automatic cars, and other autonomous physical systems. False language analysis can have disastrous effects, because the traditional safety layers of autonomous systems can not cover all possible consequences of misunderstanding. Linguists like Goldberg [11] have shown that the shallow language formalisms that underly NLP systems are not capable of capturing the broadness of language because they do not use constructions. They are therefore not sufficiently reliable for safe interaction with machines. In this paper we are looking at a deep level of language understanding that is motivated by decades of research in Cognitive Lingusitics. In literature, this is commonly referred to as *Natural Language Understanding*. However, performing NLU in a scalable manner is also very difficult, and it has been deemed to be intractable for many decades. Some important challenges for NLU are as follows:

P.1 Language is compositional. This means that the meaning of utterances is not only governed by words, but also by the grammatical constructions in which they occur [11]. This allows one to invent completely new word senses on-the-fly, and it makes grammar flexible. For example, the construction

"sneeze the napkin off the table" imposes a transitive cause-motion semantics onto the verb "sneeze", while the verb by itself does not involve any motion. A NLU system should be able to understand such constructions without having a grammatical rule for each possible meaning of a word.

P.2 Language is domain-specific. Integrating a NLU system with an autonomous system currently requires hand-tailored data structures, grammars and vocabulary for each individual application domain, and respective language processing systems have to be interfaced with each application from the ground up.

P.3 Language is ambiguous. One mode of ambiguity is underspecification. This is the case when nouns can be grounded in more than one object in a closed scene. For example, in a situation where a glass of juice and a glass of milk is nearby, and one commands an assistant robot "please bring me the glass", then, given that no other background knowledge is available, the robot will not be able to infer which glass the user referred to.
Another mode of ambiguity stems from anaphora resolution. Natural language often involves pronouns that refer to other words in a sentence. For example, consider a conditional command like "if there is a glass of milk on the table, please bring it to me." It is not trivial for a computational system to infer that the "it" refers to the glass of milk. Embodied Construction Grammar (ECG) can solve this issue because it can identify the "glass of milk" as what linguists call the *head* of a noun phrase, which makes it very likely to be the referent. Consider also the variation "if there is a table under the glass of milk, please bring it to me." Here, "table" is the head, but without further context, it is difficult to determine which antecedent the pronoun refers to. It is less plausible that a user would ask the robot to bring the entire table, so the "glass of milk" could be the higher-ranked antecedent For both kinds of ambiguities, a system integrator needs to implement clarification dialog mechanisms to perform a resolution.[1]

In this paper, we present a framework that addresses these issues. We address **P.1** by using Embodied Construction Grammar [9]. This allows us to define constructions independently from vocabulary. To infer semantics from combinations of words and constructions, we use a Bayesian best-fit algorithm that builds on research results from cognitive linguistics [2]. We address **P.2** by minimizing the effort that is necessary to integrate the NLU system in new applications. To achieve this, ECG helps us again, because its compositionality allows us to provide core definitions of the most common constructions, and the user only needs to add word tokens to the vocabulary. The latter can be done with a workbench GUI. We also provide templates for common action specifications, such as moving or grasping, which are often used in autonomous system and robotic applications. In some cases one also needs to extend the existing core grammar and templates as appropriate. We adress **P.3** by providing all the necessary mechanisms to perform clarification dialogues. This is realized via a world model and a Problem Solver that is equipped with a mechanism to infer ambiguities and to generate disambiguation queries. The system integrator only needs to implement a mapping from the internal situation model, which is defined by the ActSpec templates, to the application's API.
There are still many other open problems in language understanding, such as intention detection, nonverbal communication, metaphoric language, speech recognition and intonation analysis. We do not solve all these problems, but we believe that providing a flexible framework like ours is a good starting point towards sophisticated Natural Language Understanding for a broad range of tasks and users.

2 Preliminaries

Embodied Construction Grammar (ECG) [9] is based on the Neural Theory of Language (NTL) [9] and rooted in research from cognitive linguistics [10]. It provides a precise formalism and technical

[1] There are also much harder cases of reference resolution that require context and background knowledge, e.g., as in Winograd schemas [15]. We leave such problems open for future work.

notation to describe the grammar and meaning of a language. Like other construction grammars [21, 4], ECG consists of a network of rules outlining form and function pairings; meaning is represented with a lattice of embodied *schemas*, which are used by grammatical constructions to bind schema roles to grammatical constituents. As an example, consider the computational analysis of the sentence "John saw the box" found in Figure 1. The figure displays a so-called Semantic Specification. The

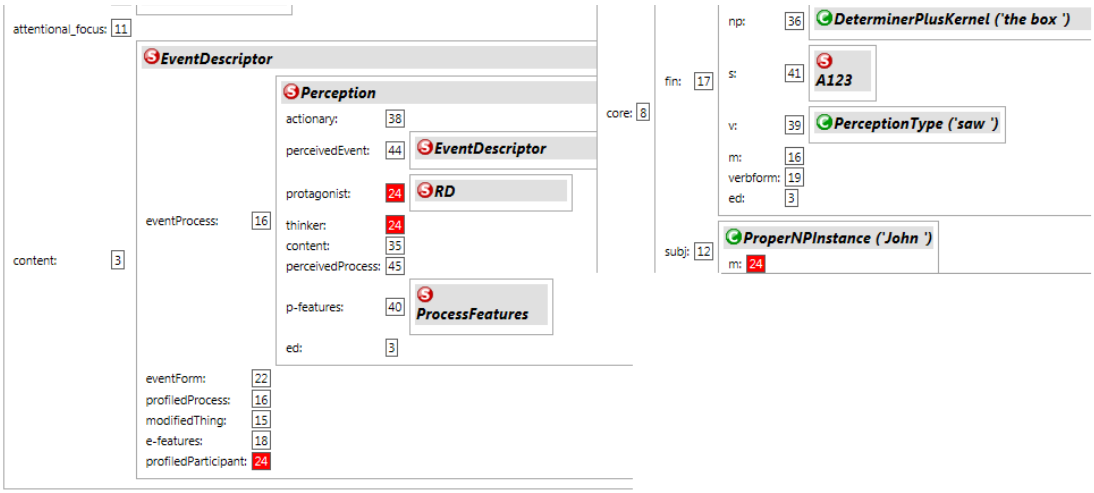

Figure 1: SemSpec excerpt for "John saw the box"

meaning of the sentence is *Perception*, with *John* filling the *protagonist* role.

Central to the theory and motivation behind ECG is that the schemas are embodied and cognitively plausible [9]. The schema inheritance lattice provides a conceptual network rooted in conceptual primitives like perception, causation, motion, and spatial relations. Thus, more abstract or specific schemas can be situated in a network of more embodied schemas; this is congruent with how previous work in linguistics suggests humans conceptualize world knowledge [10]. The combination of semantic and constructional compositionality allows ECG grammars to generalize across domains, and permits greater expressive power while maintaining cognitive plausibility.

In previous work [22, 12, 8], we have demonstrated how ECG can be used to interact with autonomous systems, robotic applications and multi-agent systems. In this work, we extend that work by releasing the framework to the public and providing a detailed description of how novel apps can be integrated in a principled and user-friendly way.

3 System description and implementation

A system overview is depicted in Figure 2. The user sends text or speech input to the system, which is then semantically analyzed by the UI Agent. The UI Agent generates action specifications (ActSpecs) for the Problem Solver, which is connected to the app. The interface between the app's API and the Problem solver is defined within Python code. Note that the NLU side of this interface is largely application independent, due to common linguistic structure. We use shared background data structures and a modular grammar repository which already contains the necessary grammatical constructions for most applications. This way, the framework requires only moderate work for a system integrator to connect it to a specific application.

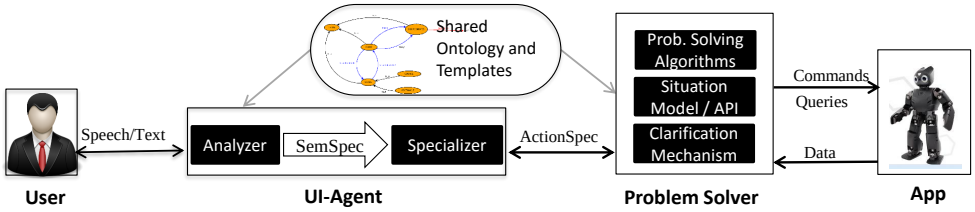

Figure 2: Framework overview

3.1 UI Agent

The Core UI-Agent controls interactions with the user. It receives text or speech as input, and produces an ActSpec as output, which it sends to a Problem Solver. In cases where ambiguities need to be resolved, it also receives clarification ActSpecs from the Problem Solver and conveys them to the user.

3.1.1 Analyzer

The ECG Analyzer takes text as input and uses the chosen grammar to generate a semantic specification (SemSpec) from it. The reasoning within the Analyzer is based on the linguistically motivated Bayesian best-fit approach by [2]. Figure 1 shows the SemSpec for the sentence "John saw the box"; note that the nested boxes with a red "S" denote schemas and their roles (the meaning of an utterance), and the boxes with a green "C" denote constructs and their features (the form of an utterance). In terms of the underlying theory, this SemSpec contains the parameters to produce action or simulated action. Note that the analyzer is application independent and can be used for any purpose.

3.1.2 Specializer and ActSpec templates

The Specializer generates action specifications (ActSpecs) based on the SemSpecs that it obtains from the analyzer. ActSpec templates are essentially Feature Structures that are designed to encode actions. ActSpec templates are instantiated as ActSpec instances and serve two primary functions in our system:[2]

1. They are declarative instructions: While building an ActSpec, the templates act as declarative specifications that govern which aspects of the SemSpecs the core Specializer should extract. They also give instructions on how the output should be formated.
2. They provide a shared semantics: Templates are oriented towards particular tasks, and are shared between the Specializer and Problem Solver; this establishes a shared vocabulary and semantics between a human user and a robot or another application.

There are the following different kinds of templates, which are organized in different .json files:
Mood Templates. These are high level templates, which map what is linguistically known as "mood" of a SemSpec onto a given template. The mood can be one of *command*, *declarative*, *yn_question* or *wh_question*. For example, a Yes/No question like "is the green box near the blue box?" will match to the following "yn_question" template.

```
"yn_question": {
        "predicate_type": "query",
        "return_type": "boolean",
        "eventDescriptor": null}
```

Event Templates. These high level templates correspond roughly to the Event-Descriptor schemas known from ECG [9]. They include event-features, such as modality information like can, temporal information like duration and telicity, and whether or not an event is negated. They also include other embedded templates, which are either parameter templates, descriptor templates, or other

[2]In this paper, we refer to an ActSpec *instance* when using the word ActSpec without further clarification.

event templates. There are the following three Event Descriptor (ED) templates: *EventDescriptor*, *ConditionalED*, *ComplexED*. As an example consider the following excerpt of the *ConditionalED* template (left) that is used for a conditional command, like "Robot1, if the box near the green box is red, push it south!" (right).

```
"ConditionalED": {
    "condition": {"eventDescription": "ed1"},

    "conclusion": {"eventDescription": "ed2"},
    "alternative": {"eventDescription": "ed3"},
    "complexKind": "complexKind",
    "conditionalValue": "conditionalValue", ...}
```

```
"ConditionalED": {
    "condition": {the box near the
                  green box is red}
    "conclusion": {push it south}
    "alternative": {push it north}
    "complexKind": conditional
    "conditionalValue": bounded, ...}
```

Note that mapping sentences to ActSpec instances is recursive. That is, the partial sentences in the *condition*, *conclusion* and *alternative* roles will also be encoded via nested other ActSpecs.

Parameter Templates. This is a very general category with currently 23 different parameter templates. These capture ECG processes and concepts like possession or spatial relations. A simple example is the "Perception" template, which is used for sentences like "John saw the house." It inherits all the roles from "Process" and adds a role for content, such as the object of perception.

Descriptor Templates. We have 8 different templates to describe referents, such as locations or objects. They are important for anaphora resolution by the specializer.

The following excerpt of an *objectDescriptor* template includes information about the gender, the category and other properties of an object.

```
"objectDescriptor": {
        "ontological_category": "ontological_category",
        "referent": "referent",
        "gender": "gender",
        "pointers": {
                "PropertyModifier": {"property": "value"},
                "ScalarModifier": {"property": "value"},
                "TrajectorLandmark": {"decriptor": "locationDescriptor"},
                "EventDescriptor": {"descriptor": "processDescriptor"},
                "MetaphoricalScalarModification": {"property": "value"}...}...}
```

3.2 Problem Solver

The Problem Solver allows for the modular integration of application-specific problem solving algorithms. It is not to be understood as a general AI to solve all kinds of algorithmic problems.

When adding language as a new interaction mode to an app, it often happens that novel interaction patterns are desirable, such as executing conditional commands or answering advanced queries, which may not be supported natively by the app. A system integrator will want to exploit the full potential for these new interaction mechanisms, and the Problem Solver offers a place to implement algorithms for such interaction. This can be all kinds of algorithms, including action planning, motion planning and theorem proving, and one could also connect external tools for these tasks (e.g. [7]). While there are many cases where it is better to do this within the app, such as the Robot Operating System (ROS), which comes with a variety of problem solving algorithms itself, it is also often the case that one can not do so, e.g., if dealing with closed-source software.

Another example is open-source apps that do not provide a coherent internal situation model or another state representation that is compatible with the language interface. Since the Problem Solver already comes with a such a world model, it can save a system integrator the effort of adopting the app's internal situation model and data structures to be compatible with a certain problem solving algorithm. A positive side-effect of this design-choice is that common problem solving algorithms like path planning are often domain-independent, so they can be shared across applications. The standard

object oriented programming inheritance methods offered by Python allow us to realize a hierarchical set of problem solver classes to re-use such algorithms.
There is one core Problem Solver class which contains the basic functionality and interfaces. On top of that, we built a more specialized robot Problem Solver subclass which contains, e.g., a path planning algorithm, and on top of that we built specialized Problem Solver classes for each robot application that share the same path planning algorithm. This is possible due to the commonalities in the situaton model, and ultimately due to the language and domain independent natural language semantics captured by ECG.
Apart from solving app-specific problems, the Problem Solver has several other important functions: Firstly, It also serves as interface to the particular app's API, as a maintenance mechanism to keep track of a coherent world model. That is, for each application it is necessary to implement at least a minimal Problem Solver subclass that translate the internal situation model to the app's API. This little extra effort is necessary anyway when connecting any kind of NLU or NLP system to an app. Secondly, the Problem Solver is crucial for clarification. It uses the situation model to detect ambiguous commands and queries. It then sends the ambiguous ActSpecs, enriched with generated verbal clarification requests, back to the user (via the UI agent) to initiate a clarification dialogue.
Demonstrations of the interaction flow between app, Problem Solver and UI Agent in the implemented system can be found in Section 5.

4 Workflow for connecting a new App

In this section we present the workflow for integrating the NLU system with an application. Illustrative examples of this process can be found in Sec. 5, where we show how our NLU system is integrated with several applications.

4.1 Set up the core system

First, the core system needs to be set up. We provide very detailed steps on how to install the system in the Wiki pages of our GitHub repository [23]. This involves installing the ECG core system, downloading the core grammars, installing the workbench GUI, and optionally one or more of our existing example applications.

4.2 Add app-specific vocabulary and extra grammatical constructions

A system integrator will need to add extra vocabulary that is required for his application. Adding extra grammatical constructions should not be necessary unless you want to use very special language or if you want to experiment with advanced linguistic constructions like metaphors. We give an example for such advanced language requirements in our Wiki pages [23][3], for an application called KARMA [17], which can predict economic states and policies. For both language extension tasks we provide a GUI, the ECG workbench (see Figure 3).
The workbench is a powerful grammar editing tool based on the Eclipse system. The left column in Fig. 3 shows the list of grammatical constructions and schemas that are currently active. Loading and unloading grammar files is done via imports using a preferences file for each particular app.
The middle window shows the grammar editor. The editor is used to add application specific schemas and constructions if necessary. The screenshot shows constructions that are relevant for a grasping action, i.e., an *ApplyForceType* construction and a *ManipulationType* construction. Both have a constraint that assigns an actionary, denoted as *self.m.actionary*. The *self* denotes that the assignment affects the construction itself, and not possible external constitutents, in a manner akin to assigning constituents to classes in object oriented programming. The *m* denotes that the assignment affects the

[3] https://github.com/icsi-berkeley/ecg_framework_code/wiki/Tutorial:-Retargeting, accessed 06/16/16

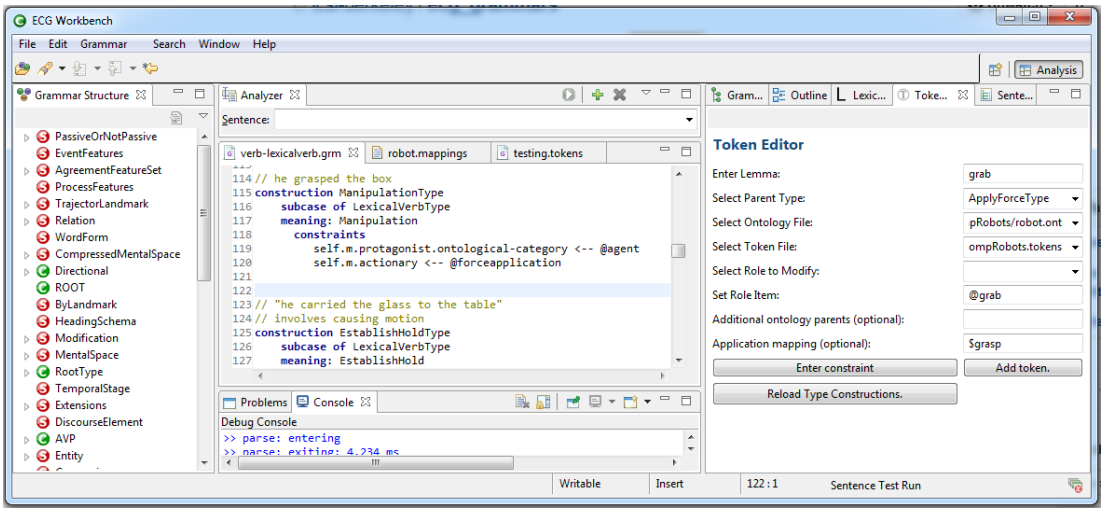

Figure 3: The ECG workbench

meaning part of the construction. The *@forceapplication* is a default value for the constraint which can be overwritten, depending on the lexical tokens that are used in an instance of the construction. On the right of Figure 3 is the Token Tool, a tool that is used to add new word tokens. For each new word, one needs to define the ECG grammatical role that it modifies. This is necessary for the analyzer to reason about the word's function within its grammatical context. For example, if one wants to add a new word "grab", possibly to extend the language scope for a particular application to provide an alternative for saying "grasp", then the word has to be specified as a *ManipulationType* construction and one needs to specify that it modifies the role *self.m.actionary*. In this case, *@grab* would overwrite the *@forceapplication* constraint in the construction. To simplify this process for non-linguists, one would start with an existing similar word, like "grasp", and edit the token entry appropriately. The ECG workbench also contains a Lexicon Viewer that can be used to simplify the process of selecting appropriate parent types and roles to modify for a new token. While it take some effort for a system integrator to appropriately add new tokens, we want to emphasize that this is less work than re-writing whole grammars, which is currently necessary for most NLP systems. It also allows for a flexible use of new words. For example, adding a lemma for "grab" automatically enables the NLU system to understand the word in a transitive, intransitive or another grammatical context.

4.3 Define ActSpec templates

Defining additional ActSpec templates should usually not be necessary. If it is, then one can extend existing .json files described in Sec. 3.1.2 appropriately. As an example, consider metaphoric language where one wants to express an *Action IS Motion* metaphor, as in the sentence "India is stumbling in implementing its economic policy." To capture this metaphor, one would use an ActSpec as follows (template left, excerpt of instance right):

```
Action_is_Motion: {
      mover: {descriptor:
                objectDescriptor}
      actionary: actionary
      aspect: aspect
      frame: frame
      action: {parameters: action}}
```

```
Action_is_Motion: {
      mover: {objectDescriptor: {
                referent: India,
                type: country}}
      actionary: @stumble
      aspect: @progressive
      frame: SelfMotion action: ...}
```

This is part of the general ECG2 constructional treatment of metaphor [17].

4.4 Extend the Problem Solver

Extending the Problem Solver for a new app is always necessary because it contains the mapping to the app-specific API. We provide one *CoreProblemSolver* Python class which contains the minimal structure. This class can be extended hierarchically to maximise the re-usability of code. For example, we also provide a *RobotProblemSolver* class which inherits from *CoreProblemSolver*. The robot problem solver is more specific that the core solver, but general enough to serve as a superclass for all kinds of robotic applications.
To connect a new application, the system designer needs to select an appropriate problem solver superclass and use Python inheritance mechanisms to extend that class. He then needs to define API calls to the app as appropriate, and add app-specific problem solving algorithms if desired. Such domain-specific algorithms could theoretically range from obstacle avoidance to reasoning about past or future events. The extension of the Problem Solver involves some programming, but it also provides considerable flexibility for all kinds of APIs, no matter if they are based on HTTP requests, Sockets, or other protocols.
The situation model that the Problem Solver maintains involves a large dictionary, mapping known objects to key/value pairings. This makes it easy and efficient to perform a search for features like color, location, size, weight, and more complex relations, depending on the domain. An example for a more complex case is deciding whether two objects are *near* to each other, which is highly dependent upon domain context.

4.5 Extend the Specializer

Most novel behavior can be handled by defining novel ActSpec templates, but it is sometimes necessary to understand specialized language or to map it to new application systems. In these cases, one can extend the core grammar and the Specializer appropriately. Editing the grammar is done via the ECG workbench, as described in Sec. 4.2. Extending the Specializer can be done by subclassing the existing *CoreSpecializer*, and adding or overriding existing methods when appropriate. The main methods of the *CoreSpecializer* govern how the Specializer crawls the SemSpec according to the declarative instructions provided by ActSpec templates; these methods should not have to change. However, if the application-specific Problem Solver needs new types of descriptor templates or involves novel constructions, some new Sepcializer code for extracting the right information and formatting it appropriately might be necessary.

5 Application examples

We have developed the framework over several years by adjusting it to the needs of several different applications. Morse, a lightweight robot simulator was the first testing app, and later we also added heavier robot applications based on the Robot Operating System (ROS). We are currently also looking into real-time strategy computer gaming, using the publicly available API of the Starcraft game.

5.1 Morse robot simulation

We provide a tutorial with instructions on how to use our language understanding system in a demo for the Morse robotics environment [23][4]. This demo allows users to control two vehicle robots in a simulated block world with language. Previous work [12, 22] has discussed the Morse system and its abilities in more detail.

1. Identify the vocabulary and language needed for the key actions and queries we wanted the robot to demonstrate. Since the Morse robot model only provides simple locomotion, no grasping or other manipulation, the actions are tailored around motion (moving and pushing). Queries

[4] https://github.com/icsi-berkeley/ecg_robot_code/wiki/Tutorial:-Morse-Demo, accessed 23/06/16

Figure 4: Two vehicle robots in a simulated world with different blocks.

can include questions about object properties and locations. Thus, the vocabulary includes a wide range of motion verbs, which affect different parameters like *speed*, such as "dash", "dart", and "amble", along with adverbs like "quickly", "carefully", and "slowly". We also add force-application verbs like "push". Finally, we add tokens for attributes, such as color ("red", "blue", etc.), size ("huge", "small", "big", etc.), and weight ("heavy", "light", etc.). Note that the size and weight tokens also allow for morphological inflections for comparative ("bigger than") and superlative ("biggest") meanings.

2. Extend the *CoreProblemSolver* to the robotics domain and create a general *RobotProblemSolver*. This inherits the structure of the core solver, and adds desired functionality, such as parsing ActSpecs about motion and pushing, and answering question about object locations. This *RobotProblemSolver* is general enough that it can be repurposed for other applications in the robotics domain, and we later re-use it for the case of ROS (see Sec. 5.2). For the Morse domain, we create a subclass called *MorseProblemSolver*, which makes API calls through a Morse socket using the *pymorse* Python module. This solver is nontrivial, because the Morse robot models only have an API for moving and retrieving information about object locations. Thus, mechanisms had to be added for *pushing*, as well as a basic path planner for obstacle avoidance.

Creating new ActSpec templates or extending the core Specializer is not necessary. This product imports core templates for *commands*, *queries*, and *assertions*, along with events and processes of various kinds, including: *conditionals*, *Motion events*, and *Pushing events*.
For the Morse demo, we focused on actions like moving and pushing, composing these actions into more complex serial commands ("Robot1, move to the green box then move it 4 inches south!", "Robot1, push the green box north before moving to the blue box!"), questions about world states ("is the blue box near the big red box?"), and conditionals that combine questions and commands ("Robot, if the box near the green box is red, push it north!", "Robot1, while you are not near the big red box, move 1 inch north!").

5.2 ROS robotic applications

This applcation builds on previous work on Human-Robot-Interaction [8]. We provide detailed instructions on how to get started with our ROS demo in our Wiki pages [23][5]. A video that shows the system in action can also be found online [6]. We used two different simulated robots, a PR2 and a Darwin-OP. For both robots, we implement illustrative home assistance scenarios that involve object manipulation and navigation. The process of connecting the NLU ramework to the ROS application is as follows:

1. Add tokens for all verbs that can be used to describe the actions of the robots. These are movement verbs like "go", "move", "drive", "walk", "dash" and verbs describing object manipulation

[5] https://github.com/icsi-berkeley/ecg_robot_code/wiki/Tutorial:-Ros-Demo, accessed 16/06/16

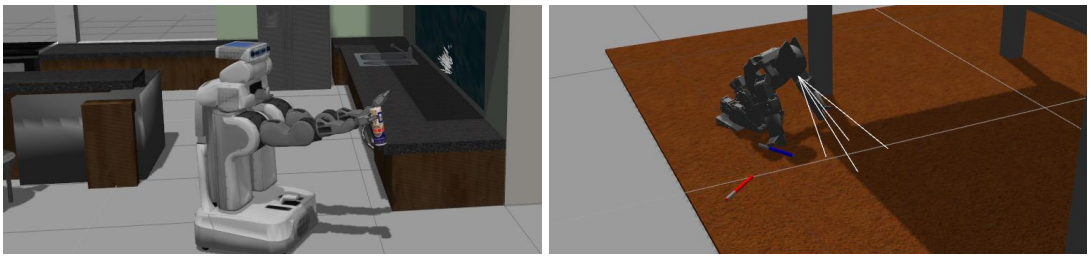

Figure 5: PR2 carrying a can of soda in the kitchen environment (left) and DARwin-OP picking up the blue marker (right)

like “grasp”, “grab”, “pick up”, etc. Also add tokens for words that describe the relevant objects, like “glass”, “marker”, “kitchen”, “table”, and object properties like, “red”, “green”, “large”.

2. Extend the Problem Solver to be able to connect to the ROS API. We were able to use the basic *RobotProblemSolver* described above, and to extend it for ROS-specific API calls. ROS has a sophisticated topic-based messaging paradigm, and we added the relevant Python libraries to the Problem Solver to communicate with this API. It sends simple commands like “moveToXY(x,y)”, “graspObject(objectName)” to ROS, and receives messages that represent the location of objects and other world properties in a similar propositional syntax. These messages are used to update the Problem Solver’s internal propositional situation model. Note that our focus was on language understanding, and not on robot-specific problems like self localization, obstacle avoidance or grasping; we simple used ad-hoc methods to solve these problems.

Adding ActSpec templates, extending the Specializer or extending the grammar is not required. Our ROS app only uses *command* and *query* type templates which are a core part of our system.
In the scenario with the PR2 robot we focus on anaphora resolution. For example, we ask the PR2: “If a soda can is on the kitchen counter, please bring it to the dining table.” The system is able to infer that the “it” refers to the soda can and not to the kitchen counter. A second example is about grounding. There are two text markers on the floor, and we ask the robot “Darwin, please pick up the marker.” The Problem Solver identifies this command as ambiguous and generates a verbal disambiguation query “Which marker?”. After the user answers this question via his input console, the command is clarified and the robot starts acting.

5.3 Real-time strategy computer gaming

The implementation for interacting with the computer game Starcraft is capable of several commands to build different kinds of in-game units while obeying in-game resource management rules. This is achieved with relatively little modification to the core framework:

1. Add game specific vocabulary to the grammar using the Token Tool. These include terms like “marine”, “barracks” and “SCV” (Space Construction Vehicle). This is done by adding new ontology entries for the unit categories and morphology entries and tokens for the words.
2. In order to communicate with the c++ Starcraft API, we extend the Problem Solver so that it is able to communicate with Starcraft. The Starcraft software is running on a different machine, so we also implemented a simple bridge interface to be able to communicate with the Problem Solver over different networks.

With the above changes, a user can tell the system to “build three barracks”, for example (see Fig. 6). The analyzer parses the command, and the information in it ultimately becomes an ActionSpec sent to the Problem Solver. The Problem Solver then communicates to the Starcraft API to build 3

Figure 6: The Starcraft app

barracks. An extra problem solving logic that we implemented for this, is that if not enough resources are available to build all requested barracks, the Problem Solver attempts to build more barracks when resources become available.

6 Related work

Language interfaces for robots and autonomous systems are of high importance in the field of Human-Machine-Interaction (HMI). To relate our work with the state of the art, we investigate those existing approaches that consider deep semantic language *understanding*, not just shallow language processing. This implies that systems which use simple context-free grammars and most learning based systems are out of scope. We look at these frameworks from a Computer Science perspective and investigate not only their linguistic capabilities, but also their integration-friendliness and flexibility.

The control framework by [16] use a Robot Control Language (RCL) to ground natural language in robot actions. The system learns a probabilistic version of Combinatory Categorial Grammar (CCG) [20], and maps the CCG parser's output to RCL. CCG features constructional compositionality, which would make it easy to also deal with conditionals or other rich structure in language. We have not found the use of such structure in the paper, but the fact that their approach is based on learning makes it flexible and adaptive. However, a principled way of separating the langauge understanding side of their toolchain and connecting it to other applictions is not provided.

The work in [3] describes an NLU module for the *DIARC* HRI-System [19]. The architecture is capable of speech recognition, semantic analysis, intention detection, disfluency analysis and reference resolution. For the semantic analysis, the authors use Combinatory Categorical Grammar (CCG) and lambda conversions [20]. Their system differs from ours in that its ambition is to provide a full cognitive computational system, while we focus on application-independent NLU.

The authors of [14, 13] focus on dialog, using a categorical modal-logical semantics and Combinatory Categorical Grammar (CCG) [20]. Their system can perform reference resolution and starts a clarification dialog if the reference is too ambiguous. For disfluency analysis, the authors use contextual knowledge to prime utterances. Verbal feedback with words like "okay" or "fine", can also be handled. The authors present an implementation which they use to perform experiments with impressive results, but do not demonstrate how the system is connected to real or simulated robots in a modular manner.

The CC and Boxer system by [5, 1] has a toolchain similar to ours, starting with CC for syntactic parsing and handing the result over to a tool called Boxer for semantic analysis. The authors use Combinatory Categorial Grammar (CCG) [20]. Their framework is tightly integrated with Prolog, and it therefore offers nice integration possibilities with logic-driven autonomous systems. However, the authors do not provide a principled way to adopt grammars and problem solving mechanisms for integrating the system with novel applications.

7 Conclusion

We describe a framework for language *understanding* that offers deeper analysis and problem solving capabilities than the most existing shallow language *processing* frameworks. This paper is written from a system integration perspective, to provide an overview for researchers who plan to connect NLU as a novel interaction method to their system. For details we refer to the in-depth instructions in our Github repository [23], where all relevant code and data can be downloaded. To give an idea of the effort that it takes to integrate our system with a new application, we provide three examples of system integration in Section 5. Integrating another new systems will most likely require less effort, because much of the vocabulary and functions that we provide for our example applications can potentially be re-used.

As future extensions, we would like to explore more application domains, such as autonomous cars or office software. We also want to make the framework even more modular, by providing a public community hub, where users can upload and access custom grammars, ActSpec templates or problem solvers created by other users. We have also started to integrate the Kaldi speech recognition toolkit [18] into our system. We plan to explore speech further by capturing intonation information about an utterance within the grammar, which can be incorporated into the ECG Analyzer's process for ranking SemSpecs. On the theoretical side, we are interested in developing a more principled situation model representation to allow for more general problem solving algorithms.

Acknowledgment

This work is supported by the Office of Naval Research grant number N000141110416 and a research grant from Google. Manfred Eppe received support from the German Academic Exchange Service (DAAD) via the FITweltweit program.

References

[1] Johan Bos. Wide-Coverage Semantic Analysis with Boxer. In Johan Bos and Rodolfo Delmonte, editors, *Semantics in Text Processing (STEP)*, Research in Computational Semantics. College Publications, 2008.

[2] John Edward Bryant. *Best-Fit Constructional Analysis*. PhD thesis, University of California at Berkeley, 2008.

[3] Rehj Cantrell, Matthias Scheutz, Paul Schermerhorn, and Xuan Wu. Robust spoken instruction understanding for HRI. In *International Conference on Human-Robot Interaction (HRI)*, 2010.

[4] William Croft. *Radical Construction Grammar: Syntactic Theory in Typological Perspective*. Oxford University Press, 2001.

[5] James R. Curran, Stephen Clark, and Johan Bos. Linguistically Motivated Large-scale NLP With C&C and Boxer. *Annual Meeting of the ACL on Interactive Poster and Demonstration Sessions*, (June):33–36, 2007.

[6] Manfed Eppe and Sean Trott. Video – embodied construction grammar for human robot interaction. https://youtu.be/BAqoZEi1IVA, accessed 06/17/16, 2016.

[7] Manfred Eppe, Mehul Bhatt, and Frank Dylla. Approximate Epistemic Planning with Postdiction as Answer-Set Programming. In *International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR)*, 2013.

[8] Manfred Eppe, Sean Trott, and Jerome Feldman. Exploiting Deep Semantics and Compositionality of Natural Language for Human-Robot-Interaction. *International Conference on Intelligent Robots and Systems (IROS)*, 2016.

[9] Jerome Feldman, John Edward Bryant, and E Dodge. Embodied Construction Grammar. In *The Oxfrod Handbook of Computational Linguistics*, pages 38 – 111. Oxford University Press, 2009.

[10] Charles J Fillmore. Frame Semantics and the Nature of Language. In *Conference on the Origin and Development of Language and Speech*, volume 280, pages 20–32, 1976.

[11] Adele Goldberg. *Constructions: A Construction Grammar Approach to Argument Structure*. University of Chicago Press, 1995.

[12] Huda Khayrallah, Sean Trott, and Jerome Feldman. Natural Language For Human Robot Interaction. In *International Conference on Human-Robot Interaction (HRI)*, 2015.

[13] Geert-Jan Kruijff, Pierre Lison, Trevor Benjamin, Henrik Jacobsen, Hendrik Zender, and Ivana Kruijff-Korbayová. Situated dialogue processing for human-robot interaction. *Cognitive Systems*, 2010.

[14] Geert Jan Kruijff, Hendrik Zender, Patric Jensfelt, and Henrik Christensen. Situated dialogue and spatial organization: What, where... and why? *International Journal of Advanced Robotic Systems*, 2007.

[15] Hector Levesque, Ernest Davis, and Leora Morgenstern. The Winograd Schema Challenge. *International Conference on Principles of Knowledge Representation and Reasoning (KR)*, 2012.

[16] Cynthia Matuszek, Evan Herbst, Luke Zettlemoyer, and Dieter Fox. Learning to parse natural language commands to a robot control system. *Int'l Symposium on Experimental Robotics (ISER)*, 2012.

[17] Srini Narayanan. *Knowledge-based Action Representations for Metaphor and Aspect*. PhD thesis, University of California at Berkeley, 1997.

[18] Daniel Povey, Arnab Ghoshal, Gilles Boulianne, Nagendra Goel, Mirko Hannemann, Yanmin Qian, Petr Schwarz, and Georg Stemmer. The kaldi speech recognition toolkit. In *In IEEE 2011 workshop*, 2011.

[19] Matthias Scheutz, Gordon Briggs, Rehj Cantrell, Evan Krause, Tom Williams, and Richard Veale. Novel mechanisms for natural human-robot interactions in the diarc architecture. In *AAAI*, 2013.

[20] Mark Steedman. *The syntactic process*. MIT Press, 2000.

[21] Luc Steels. *Design Patterns in Fluid Construction Grammar*. John Benjamins Publishing, 2011.

[22] Sean Trott, Aurélien Appriou, Jerome Feldman, and Adam Janin. Natural Language Understanding and Communication for Multi-Agent Systems. In *AAAI Fall Symposium*, pages 137–141, 2015.

[23] Sean Trott, Manfed Eppe, Jerome Feldman, and Adam Janin. Ecg nlu framework wiki. https://github.com/icsi-berkeley/ecg_homepage/wiki, accessed 06/17/16, 2016.