Robot Program Construction via Grounded Natural Language Semantics & Simulation

Robotics Track

Mihai Pomarlan
University of Bremen- FB10
Bremen, Germany
pomarlan@uni-bremen.de

John Bateman
University of Bremen- FB10
Bremen, Germany
bateman@uni-bremen.de

ABSTRACT

Robots acting in semi-structured, human environments need to understand the effects of their actions and the instructions given by a human user. Simulation has been considered a promising reasoning technique to help tackle both problems. In this paper, we present a system that constructs an executable robot program from a linguistic semantic specification produced by parsing a natural language sentence; in effect, our system grounds the semantic specification into the produced robot plan. The plan can then be run in a simulated environment, which allows one to infer more about the plan than was present in the initial semantic specification. Our system allows modeling how actions can be modified by subclauses, which we showcase by a transport action. Simulation runs allow discovery of better parameters, either locally for a subtask or such that the entire task is better performed; simulation reveals these parameterizations may differ.

KEYWORDS

cognitive robotics, language grounding, human-robot interaction, robotic agent languages

ACM Reference Format:
Mihai Pomarlan and John Bateman. 2018. Robot Program Construction via Grounded Natural Language Semantics & Simulation. In *Proc. of the 17th International Conference on Autonomous Agents and Multiagent Systems (AAMAS 2018), Stockholm, Sweden, July 10–15, 2018,* IFAAMAS, 8 pages.

1 INTRODUCTION

Many robots today are not lumbering machines toiling in an industrial hall; rather, they are aimed at becoming domestic assistants, caregivers, teaching aids and companions. These applications bring with them new challenges. A robot must understand the effects of its actions on the world, if it is to robustly act in an environment not specifically designed to be robot fool-proof. It also must understand the humans it shares the environment with, when they offer commands, requests or advice, despite the fact that human communication leaves a lot of information unspecified. Though different, these challenges have been met with related approaches stemming from the simulation theory of cognition: simulation is used both to provide an environment where the robot can "play with", so as to understand, its own actions[12], as well as a means to ground natural language understanding in an active representation [7, 11, 13] of robot behavior, which allows different kinds of reasoning to uncover what a natural language instruction leaves unsaid.

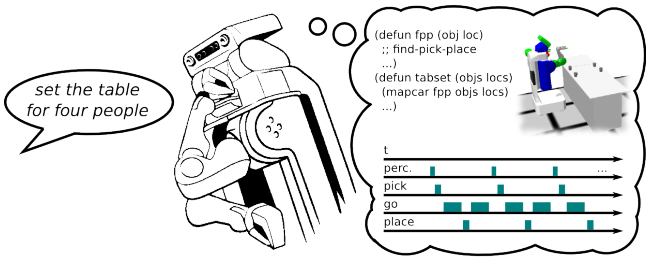

Figure 1: A deeper understanding of an instruction involves constructing a program to implement it, as well as observing what might happen in a more or less typical run.

In this paper we present a system that takes a "semantic specification" (semspec) obtained from parsing a natural language instruction and converts it into a program to be run by a (simulated) robot. Program creation starts from a few basic building blocks which are then combined based on the information in the semspec. If more interpretations (semspecs) can be created from the same NL utterance, then more programs can be tried in simulation. In that case, the simulation becomes an extra filter for interpretation, because "bad" interpretations, resulting from failures in NL parsing and semspec construction, may result in programs that don't make sense: don't run or behave in recognizably bad ways. Unlike some previous approaches, our system accomodates the creation of program steps with no correspondent in the initial semspec (such as when translating a short instruction into a longer sequence of actions). Further, we allow elements in the subclauses of a semspec to modify the function corresponding to the main verb either by describing (possibly functional) arguments to it or by replacing it with a function template to which the original function is an argument; we use this mechanism to provide the resulting program with failure handling constructs. Our semantics of language is therefore richer in describing how behaviors can modify each other and what reactions are known to recover from failure events. Simulation of the resulting program can then be used to infer properties about it beyond the description of the semspec, such as time taken. We use simulation to learn new parametrizations for robot actions so as to improve performance.

There is an apparent similarity between the system we present and a task planner, in that both take a usually high-level description of a task and output a program to perform it. We would like to emphasize however that our focus is not on searching for plans; most tasks a service robot would need to perform are, at the planning

Proc. of the 17th International Conference on Autonomous Agents and Multiagent Systems (AAMAS 2018), M. Dastani, G. Sukthankar, E. André, S. Koenig (eds.), July 10–15, 2018, Stockholm, Sweden.

level, simple, and sometimes may even be described explicitly by the human user via a list of actions for the robot to perform. Our system is better thought of as a kind of translator, from natural to a programming language, where its purpose is also, through simulation, to analyze the properties of the resulting program, and thus gain more understanding of the original sentence.

The contributions of this paper are as follows:

- composing robot programs from a linguistic semantic specification (conformant to the Generalized Upper Model)
- grounding natural language understanding into an executable and expressive representation (the CRAM plan language)
- presenting a natural language understanding through simulation pipeline, where the simulation is used to improve robot program performance

2 BACKGROUND

In this section we review some previously existing components and how we use them in our system.

2.1 Generalized Upper Model

The Generalized Upper Model (GUM) [5], an extension of the Penman Upper Model [4], is a linguistic ontology which defines general concepts and relations for representing the semantics of a natural language sentence. The representations of semantics (which we will refer to in this paper as "semspecs") can be used either for reasoning in a natural language understanding system or to guide natural language generation, as has been done with the KPML system [2], which can generate English and German sentences from a semspec. The GUM has also been extended with an ontology for spatial concepts and relations, GUM-Space [1, 3].

In this paper a semspec will be a set of key-value pairs; a key is unique in a semspec, and the order of keys does not matter. Keys are concepts or relations from the GUM and its spatial extension. The values can be literals (symbols or numbers), semspecs, designators (see section 2.2), or (in our system) function objects. For example, a sentence such as "set the table" would have the semspec:

```
[(:dispositivematerialaction . set) (:actee . table)]
```

whereas for "set the table for two using the tray" would be:

```
[(:dispositivematerialaction . set) (:actee . table)
 (:client . [(:quantity . 2)])
 (:manner . [(:affectingspatialaction . use)
             (:actee . tray)])]
```

Semspecs have been used for linguistic analysis and generation; in that context, they are instances of feature structures. We depart here from the feature structure formalism in that we allow function objects to be values; this is because our semspecs are intended to describe an executable robot program, which they do through a repeated process of interpretation and program construction. The final, executable output of interpretation is also a semspec, in which the non-GUM keys :fn and :args appear, where :fn contains a function object to be applied to the argument list in :args.

2.2 Cognitive Robot Abstract Machine

The Cognitive Robot Abstract Machine (CRAM) [6] is a set of tools for the development of cognition-enabled robot programs. Of interest for us here is the CRAM plan language, a concurrent reactive plan language in the vein of RPL [15]. The CRAM plan language is implemented on top of Common Lisp and offers several features of interest to our system, such as control structures for failure handling and concurrency, as well as a way to specify arguments to a robot task at an appropriate level of abstraction through a mechanism called designators. In general a designator is a list of key-value pairs, where the keys are unique and the order in which they appear does not matter; designators may be used to specify an object, a location, or an action/motion. For example, an object designator indicating some cup in an environment would be:

```
(an object (type cup))
```

note that there is no identifier or pose associated for the cup here. A designator for a location near a cup would be:

```
(a location (near (an object (type cup))))
```

and finally a designator for an action to move the robot base near the cup:

```
(a motion (type going)
          (target (a location (near (an object
                                    (type cup))))))
```

CRAM also has a library of basic action programs, such as perceiving, picking up, and placing an object, navigating with the robot base, using the end effector to press an object etc. These basic programs take designators as parameters, and the designators get "referenced", i.e. have more concrete data attached to them (such as an actual cup name or an actual pose) lower in the hierarchy of subroutines called by the CRAM plans. The referencing process is done by appropriate reasoners, depending on the designator type and its keys; e.g. locations are often generated using information about occupancy and costmaps.

Structurally, designators and semspecs (see section 2.1) are very similar. The difference is in how our system treats them: a semspec is something to be interpreted, and the output of an interpretation step is a (possibly executable) semspec; designators are arguments to pass to functions. A value in a key-value pair for a semspec may be a designator, but no value in a designator is a semspec.

Finally, CRAM has the ability to log execution traces [25]. The logs include information about the task tree (what tasks were run, which were the subtasks, what were the arguments and the results such as completion/failure) as well as information about what the robot perceived. One can then analyze the execution of a plan and, for example, rate it according to some metric (as we do here) or answer questions pertaining to the events that happened during the execution (such as how many cups were put on a tray), or try to find the cause of a failure.

2.3 Light-weight simulation

CRAM allows programs to be written and tested in special "projection" environments [17]. Such an environment is a simulation containing a model of the robot and the obstacles and objects around it; the Bullet physics engine is used to update the state of the world. However, unlike in full-fledged simulations such as in the Gazebo simulator, the physics engine is only turned on "on demand", and the intermediate states traversed while a robot performs an action are skipped over (e.g., moving the robot to a location teleports it there, and arms assume desired configurations instantly).

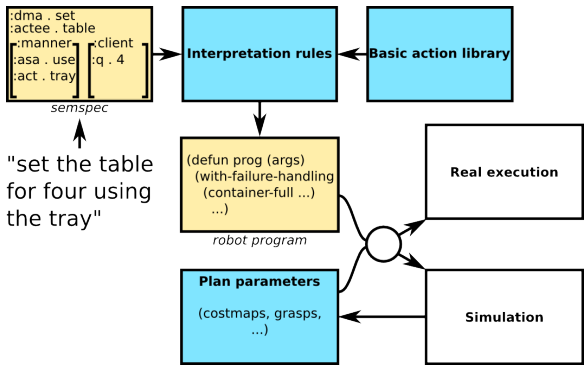

Figure 2: System overview: interpretation rules convert a semspec into robot code, which can be run in a real or simulated environment.

Light-weight simulation, or projection, loses the ability to verify robot trajectories and cannot directly offer information such as the time taken by a task. However, it is much faster to perform and still allows relevant reasoning queries to be answered by the simulator. One can test, for example, whether certain poses are reachable for the robot (using an IK solver), whether certain objects are visible (off-screen rendering), whether arrangements of objects are stable (by turning the physics engine on for a limited time). Projection is therefore particularly good at verifying significant events during a program execution, such as object perception, pick-up, and placement. It offers therefore a fast way to check whether a plan "makes sense" in the world. Also, as we show in section 4.2, it can be used to improve parameterization for better plan performance.

3 PROGRAM COMPOSITION

Our approach to translating a semspec into a robot program written in the CRAM language is rule-based: starting from a semspec which is a proper feature structure, an executable robot program is obtained by the application of transformation patterns called interpretation rules. Feature structures are defined as mappings from feature identifiers to feature values, which can be simple (numbers or strings), nested feature structures, or variables, to support unification. We allow function objects to be values inside our intermediary semspecs (which are therefore not proper feature structures). This is necessary because the desired output of interpretation in our case is a program, so there must be at least one rule translating a semspec into executable code. Further, it is more feasible and generalizable to have rules for assembling parts of a program given parts of the semspec, rather than attempt to go in one step from a feature structure semspec to executable code. Because of this, our interpretation rules may contain function code, which means we must commit to a language to write them in. Since we use CRAM to supply our basic building blocks, and CRAM is Lisp-based, so are our rules. The resulting robot program may be run on a real robot, or in simulation, and may make use of other robot knowledge such as costmaps for navigation and reachability maps for manipulation, grasping parameterizations etc. Simulated experience may update these parameter stores. An overview of our system is given in figure 2.

3.1 Semspec interpretation rules

An interpretation rule is a pair (denoted by ":-") containing an antecedent pattern and a consequent result, together with a scope restriction (denoted by ":@"). The effect of a rule on a semspec is that if the antecedent unifies with the semspec, then a new semspec is produced with the structure of the consequent. The scope restriction is used to constrain where the semspec targeted by the rule is allowed to appear inside a larger semspec. An example rule is below:

```
((:-
    [(:a . x) (:old-key . ?y) ...]
    [(:new-key . ?y) ...])
 (:@ :c :d))
```

We use "..." in the antecedent to specify that other keys may be present, but do not restrict matching; an antecedent with no "..." is matched only to semspecs containing the exact keys in the antecendent and no others. In the consequent, the "..." will get replaced with the key-values matched to "..." in the antecedent. Values in the antecedents and consequents are literals, unless the name is preceded by "?", in which case they are variables. Variables in the consequent are replaced with the values they are bound to by the unification between the antecedent and a semspec. A key-value pair in the antecedent where the value is *nil* will make the antecedent match only semspecs in which that key is absent.

Special handling must be taken for extending unification to function objects. A pattern describing a function object unifies with a semspec if the function name and signature unify. A name unifies if the name in the pattern is either a symbol which matches the one in the semspec, or a variable which then gets bound to the name in the semspec. Function signatures unify when they have the same number of mandatory and keyword parameters, and each parameter name in the pattern unifies with (is either a variable or is equal to) the parameter name in the semspec. The body of a function never appears in rule antecedents and does not matter for unification.

Consequents are usually patterns into which to place values taken from the semspec a rule was applied to. For convenience we also allow function calls on the values taken from the semspec, and use the results for the new semspec, as seen in the idiom interpretation rule for setting the table:

```
(:-
  [(:dispositivematerialaction . set) (:actee . table)
   (:client . (:quantity . ?n)) ...]
  [(:affectingspatialaction . transport)
   (:actee . ,(get-obj tableset ?n))
   (:placement . ,(get-loc tableset ?n))
   ...])
```

where get-obj and get-loc are auxiliary functions returning lists of designators (object and location respectively) and the "," denotes that the following expression is to be evaluated first before usage in a new semspec. More importantly, consequents can also construct new function objects to place in semspecs, as seen in the example below:

```
(:-
  [(:a . arithmetic-op) ...]
  [(:a . (defun op (a b &key (f +))
           (f a b)))
   ...])
(:-
  [(:a . (defun op (?a ?b &key ?f) ...))
```

```
   (:manner . /))
   ...]
 [(:a . (defun div (?a ?b)
          (with-failure-handling
            ((zero-division-error (e)
               (print "Zero divisor!")
               (return)))
            (op ?a ?b :?f /))))
   ...])
```

The first rule produces a semspec where the "arithmetic-op" symbol is replaced with a function object named op that takes two mandatory and one key parameter. The second rule modifies the op function object, first by supplying parameters to it (as in this case a function object parameter) to change its internal operation, but also surrounds it with an appropriate template to handle the new failure mode introduced by the parametrization. "with-failure-handling" is a CRAM plan language construct that catches errors by type, executes some handling code, and has the option to either exit or retry its body.

The scope restriction is a list of keys representing a "path" through a semspec, and is used to modify how semspecs appearing as values in other semspecs are interpreted. A starred key in a scope restriction means any number of repetitions of that key, including 0. Rules without scope restrictions are applicable everywhere. For a rule to be applicable to a semspec, the semspec must unify with the antecedent and it must occur in a place compatible with the scope restriction.

Scope restrictions are useful because the location where a semspec occurs should influence its interpretation: a top-level semspec is something to execute, and its interpretation eventually results in a function object and argument list. A semspec appearing as the value of a :manner key however is something to modify another action with; in this case, the interpretation result may be several function objects, semantically annotated with their roles to slot in the modified action, and/or a template to insert the modified action in. We provide an example sketch below.

```
((:-
   [(:action . use) (:actee . tray)]
   [(:fn . (defun use (o)
             (do-something o)))
    (:args . (an object (type tray)))]))

((:-
   [(:action . use) (:actee . tray)]
   [(:precede . (defun find ()
                  (perceive (an object (type tray)))))])
 (:@ :thing* :manner))
```

The first rule triggers on a top-level semspec, and replaces an instruction to use the tray with some complete and parameterized program. The second rule triggers on a semspec appearing as a value in a :manner key, either in the top-level semspec or some lower one, and produces a function object that may be used later on in the interpretation process to construct the final program. For the second rule scope restriction, we use the fact that all keys are members of an ontology and "thing" is the topmost concept.

3.2 Types of rules

We have found that rules necessary for program creation may have one of a few purposes, and therefore rules belong to one of a few categories, which we list below.

- Idiom interpretation are rules that convert a short phrase to a longer description of the action it signifies (e.g. the equivalent of going from "set the table" to "transport these objects to these locations"); these rules operate entirely inside the feature structure formalism (they never generate function objects)
- Fusion rules construct function objects and designators based on information in subclauses (e.g. a :manner subclause of the semspec being used to change the value of the main verb); they may generate function objects and put them as values in semspec keys
- Execution preparation rules rewrite a semspec to an executable form– one with :fn and :args keys

A rule's category also indicates when it may be used. Our system attempts to modify a semspec using idiom interpretation rules until no such rule is applicable; then it tries the fusion rules until none apply, and finally the execution preparation rules until an executable semspec is found. We are exploring how to organize the rules in a richer, more linguistically motivated hierarchy (e.g., rules about constructing designators from fusing information from adjectives into the designator, versus rules to construct actions), and what implications this may have for rule ordering.

3.3 Rule ordering

Several rules may apply to a semspec or one of its parts, and in general the order in which rules are applied matters. Some ordering information, as described above, is given by the rule type, however there may be many rules with the same type. In this case, our system uses a preference ordering based on "specificity", which we describe below, where a more specific rule is preferred to a less specific one. However, the specificity ordering is not strict, and we break ties in the specificity ordering with a user-defined score for each rule. In the future we will allow several interpretations for a semspec, and use the simulator to verify which interpretations make sense; this also offers an opportunity to learn rule preference scores so as to discourage combinations that seem reliably to perform badly.

Specificity refers to both the scope restriction and antecedent of a rule. A rule with a more specific scope restriction is preferred to a rule with a less specific scope restriction. If the scope restrictions are equally specific, a rule with a more specific antecedent is preferred. Specificity in antecedents is given by the subsumption relation induced by unification: if an antecedent may, through some substitution of its variables, unify with another, it is less specific.

Specificity of scope restrictions is defined thusly. Longer scope restrictions are more specific (therefore the empty scope restriction is the least specific). For scope restrictions of the same length, one is more specific if all its keys are subclasses (in the GUM ontology) of the corresponding keys in the other. For scope restrictions of same length and same string of keys, the one with no starred keys is preferred. Note that two scope restrictions can be of same specificity if they are the same length and both use (or don't use) starred keys or use keys that are not GUM subclasses of each other.

3.4 Program parametrization

Robot plans require a lot of information; for example, a grasping action needs to know what object to grasp, which gripper to use and

with what force, a grasping pose around the object, potentially some auxiliary poses to describe the motion before and after the grasp, etc. The CRAM plans we use take arguments specified very abstractly via designators, but at some level of execution those abstract descriptions need to be made concrete, e.g., when selecting an actual pose such that it is near some object.

We distinguish between arguments, which are passed explicitly to CRAM plans, and parameters, which are not, but are still necessary to inform a robot action. While the distinction is somewhat vague from a logical point of view (e.g., why should, or shouldn't, gripper force be an argument?), it is motivated by the fact that the CRAM plans we use as building blocks have particular function signatures containing only part of the information needed to execute them. A grasping plan takes as argument the object to grasp; any other information is stored in dynamic variables (Lisp's version of global variables). This allows us to write rules that parameterize basic plans by incorporating them in a closure, as in this example:

```
(:-
  [(:dispositivematerialaction . (defun grasp (?o) ...))
   (:manner . gently) ...]
  [(:dispositivematerialaction .
     (defun grasp* (?o)
       (let ((*grasp-strength* (* *grasp-strength*
                                  *softness*)))
         (grasp ?o))))
   ...])
```

4 EVALUATION

4.1 Qualitative evaluation

To get an appreciation of what our system can model, we will look at a concrete example, the transport action. This is generically defined by the pair of rules below:

```
((:-
   [(:affectingspatialaction . transport) ...]
   [(:affectingspatialalaction .
      (defun transport (obj loc &key precede follow)
        (precede obj)
        (go-to (a location (near loc)))
        (follow obj loc))
    ...]))

((:-
   [(:affectingspatialaction .
      (defun transport (?obj ?loc &key ?p ?f) ... )
    (:manner .
      ((:precede . (defun ?precede (?obj) ...))
       (:follow . (defun ?follow (?obj ?loc) ...))
       (:circumstance .
         (defun ?circumstance (&key ?actee ?placement
                                    ?action)
           ...)))))
    ...]

   [(:affectingspatialalaction .
      (defun transport* (?obj ?loc)
        (?circumstance
          :?actee ?obj
          :?placement ?loc
          :?action (lambda (ob lc)
                     (transport ob lc
                                :?p ?precede :?f ?follow))))
    ...]))
```

The first rule converts a semspec where the affecting spatial action key is associated to a symbol value (as might be produced by a parser from a natural language instruction) into a semspec where the affecting spatial action is associated to a function object. The transport action is described very generically: it involves the robot moving near a destination location (using the CRAM plan for moving the robot base, go-to). Before this movement, some preparation happens, involving some object(s). After this movement, some follow-up happens involving the same object(s) and destination location(s). Both the preparation and followup are also arguments to the transport function.

The second rule creates a new transport function by merging information present in a manner key. The expected information in the manner key are function objects describing the preparation, follow-up, and a circumstance. The circumstance describes an "environment" to run the action in.

The next rule produces the manner for the default transport action: if the original semspec has no manner (the user specified nothing about how to do the transport), then the assumed default is that the robot will try to pick an object and carry it to the destination. There may be more objects and destinations however, so the circumstance is the operation of mapping the transport action over the lists of objects and destinations: ie., transport is to be repeated for each object-destination pair. The rule constructs functions from basic CRAM plans perceive (which takes an object designator as parameter and updates internal CRAM variables such as the object's pose), pick and place (basic object manipulation actions).

```
((:-
   [(:affectingspatialaction . transport)
    (:manner nil) ...]
   [(:affectingspatialalaction . transport)
    (:manner
      ((:precede .
         (defun find-pick (o)
           (go-to (a location (near o)))
           (perceive o)
           (pick o)))
       (:follow .
         (defun place-object (o l)
           (place o l)))
       (:circumstance .
         (defun for-all (&key actee placement action)
           (map action actee placement)))))
    ...]))
```

The user however may suggest that the robot use the tray to carry the objects, which requires a different modification for the transport action. The preparation now involves finding the tray and putting objects on it, while the followup consists in taking the objects from the tray and putting them at the destinations; we comment out these steps in the example for clarity. The circumstance also changes: the tray can carry several objects, but it might not be able to carry all of them. We use a failure signal to indicate when this happens, and the failure signal includes information about what objects are not on the tray yet. Note that the failure signal triggered in the preparation function is a low level one: it simply says that a location designator could not be resolved. In the context of preparing to use the tray, this is interpreted to mean that no more locations are available on the tray, so the failure signal propagates upward with a different semantics (tray full). This second failure signal then gets handled in

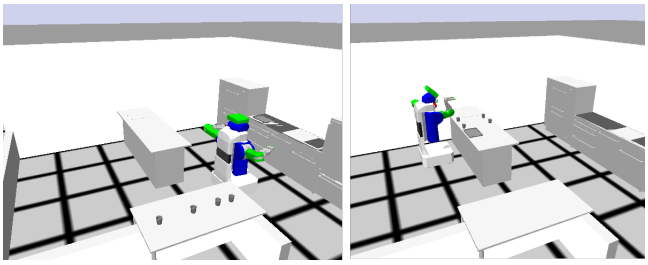

Figure 3: The "table-setting" task: moving four cups from a table to a particular arrangement on another. Left: start state; some randomization in cup and tray placement. Right: goal state; cups must be at definite positions.

the circumstance by carrying the objects on the tray, then retrying the action with the remaining objects and locations. Note that this is a scope-restricted rule (operating on a semspec inside a :manner key).

```
((:-
   [(:dispositivematerialaction . use)
               (:actee . tray)]
   [(:precede .
      (defun find-pick (o)
        ;;; pick-place tray near objects, then
        (let ((picked 0))
          (with-failure-handling
            ((desig-not-resolved (e)
               (fail container-full picked)))
            (map
              (lambda (o)
                ;;; pick-place object o on tray, then
                (setf picked (1+ picked)))
              o)))))
    (:follow .
      (defun place-objects (o l)
        ;;; place tray near l
        ;;; pick-place objects from tray to l
        ))
    (:circumstance .
      (defun use-tray (&key actee placement action)
        (with-failure-handling
          ((container-full (n)
             (go-to (a location
                       (near (subseq placement 0 n))))
             (place-objects (subseq actee 0 n)
                            (subseq placement 0 n))
             (setf actee (subseq actee n))
             (setf placement (subseq placement n))
             (retry)))
          (action actee placement))))))])
 (:@ :thing* :manner))
```

4.2 Quantitative evaluation

We generate a program for the robot to move four cups from one table to another using a tray, and we run this program several times in a light-weight simulation environment. Initial poses for cups and tray are randomized. When resolving a designator for an "on tray" location, poses are sampled from a uniform distribution over the tray surface and validated by checking that placing a cup there would not collide with other, already present cups. Resolving a "near location(s)" designator samples a uniform distribution around the first target location, and validates samples by checking that no collisions with objects already present, or with objects that may be placed at the target locations, occur.

Table 1: Distance travelled by the robot base vs. numbers of cups in tray transfers

	Avg	Stdev	Min	Max	Count
3+1	20.4	1.3	18.4	23.7	41
2+2	18.0	1.5	14.3	21.3	211
2+1+1	20.9	2.7	14.9	25.7	88

Because the light-weight simulator does not provide an indication of how much time an action takes (the robot teleports to a destination configuration), we use the distance travelled by the robot base as a proxy for execution time. We motivate this choice by observing that in a real execution the robot base movement would be slow, in part for safety reasons, and would therefore dominate the execution time.

We ran the program 340 times, and measured the total distance travelled by the robot base, as well as the number of cups that were loaded on the tray for each transport action. Because of the cup poses were obtained by sampling, the robot is not always able to place three cups on the tray while maintaining enough space between them; however, if we wanted to optimize tray use in terms of number of cups carried when running the program in the real world, we could simply use the cup-on-tray locations found in one of the simulation runs where three cups were put on the tray at a time. The simulator therefore, even a simple one like the light-weight simulator we currently use, is a means to try out and find better parameterizations for subtasks in a program.

The light-weight simulation also revealed an interesting pattern: when the robot has to carry four cups and cannot fit all of them on the tray, then attempting to carry two then two is usually better than attempting to carry three then one. This is because of where the cup destinations are located; when the robot carries three cups, it will need an extra action of going around the destination table. If it only tries to carry two then two cups, the go-around-the-table action is not needed. We show statistics over the simulation runs in terms of distance travelled by the robot base in table 1.

This is a very simple example, but it shows optimizing subtasks according to some intuitive looking metrics (number of objects placed on the tray) does not always result in optimality for the complete task (distance travelled or execution time). Instead, one might use the simulator to find parameter combinations that perform better for the global task. Exploratory simulations such as these may happen when the robot is otherwise idle [22].

5 RELATED WORK

Interpreting natural language to a form usable by robots has been a very active area of research; we only present some approaches here.

Embodied Construction Grammar (ECG) is a formalism introduced by Bergen[7] for linguistic analysis in which a natural language statement is mapped to constructions, which are then used to parameterize a "mental simulation". The simulation is performed over an "active" representation of the semantics of the statement, also refered to as a schema, which is a Petri net. Note that ECGs

were defined as a tool to analyze language in general, but have also found a home in robotics. Eppe[10] uses ECGs to create a so-called "n-tuple", which then selects and parameterizes a robot plan. Initially, ECG constructions and schemas were defined by hand, however Dodge[9] improves coverage of ECGs using data from FrameNet. We will also look at such data driven approaches to define plan composition rules in the future.

Misra[16] presents a system for interpreting commands which is trained on pairs (natural language instruction, execution trace of a human user performing the instruction in a simulator). We use a library of basic robot plans, however such a system would be useful to extend that library. Another system enabling spoken dialog between a robot and human is presented by Bos[8]: an instruction or question is interpreted into a first-order logic statement. A theorem checker verifies its consistency with known context data about the environment (by attempting to prove its negation) while a model finder attempts to find a consistent history of the instruction execution/question answer. Roughly speaking, the first order logic statement corresponds to the robot plan our approach constructs from a semspec, while the model finder plays a similar role to the simulator generating a trace of the execution of that plan. The difference is that we use simulation-based, rather than purely logical reasoning.

Matuszek[14] shows a system to parse a natural language instruction into a robot control language. It supports sequencing, looping-until, and other control structures. The correspondence between a semspec/parse and a robot plan we describe here is more complex however: it allows inferring actions and conditions not explicitly mentioned in the natural language instruction, and allows actions to be alterable in terms of the manner in which they are executed.

Nyga[18, 19] uses Markov logic networks to interpret a natural language statement into a probabilistic action core, which is a list of role-value pairs. He formalizes the interpretation problem as finding the most likely action core given the evidence provided by the natural language instruction. This approach can disambiguate actions, do reference resolution, and select appropriate tools for an action given other action parameters (such as quantity of material to manipulate). Pomarlan[20] combines it with a reasoning through simulation approach, however in that work the robot program being executed is selected based on, rather than constructed from, the action core.

Learning parameters for robot plans has been investigated by Stulp[21]. This paper defines the concept of Action Related Place: a collection of robot base poses associated with a probability of success for some manipulation action. The learning can be done on either simulated or real executions of a task. Similar ideas are investigated by Winkler[24], where parameters related to a manipulation action are richer than just the base pose. Welke[23] uses collections of poses to ground symbols for spatial relations during task planning; likely locations of objects in an environment are learned.

6 CONCLUSIONS

We have presented a system that converts a semantic description of a natural language sentence into an executable robot program, through the application of interpretation rules to the semantic description. The system can handle modifications of actions as specified in manner clauses. The generated programs include failure handling, for example when the manner of execution specified for an action has new failure modes compared to the default manner. The programs can then be run in a simulated environment or on a real robot.

Interpreting a sentence into a program allows the robot to have a deeper understanding of what the sentence means, because simulation allows it to reason about sequences of events and interactions between subtasks that are not described in the original semantic description. The robot can also "explore" parameter settings, to be used for later program executions. The results of these explorations depend on both the programs being simulated, the tasks they are simulated for, as well as the environments they are simulated in, and are therefore much richer than what could be obtained from reasoning about a linguistic description of a task alone.

We have used a very simple simulator for this paper. While the light-weight simulator is fast and captures some important queries, such as reachability, or collision checking, it is nonetheless fairly limited in the kinds of phenomena and interactions it can handle. For future work, we are looking at a physics simulator that can also model deformable bodies and fluid dynamics. We will also look at defining a richer ontology of interpretation rules so as to have a more principled way to order their application to a semantic description.

We have clear guidelines for how to extend the coverage of our interpretation rules. For semspec keys, we aim for full coverage of the action part of the GUM. For semspec values, we aim for full coverage of a kitchen actions grammar being provided by colleagues from our university, developed by studying large corpora and extended with user surveys on the correctness of generated sentences and variations thereof. In terms of basic action concepts implemented in the rules, we aim to cover a set of basic action concepts emerging from an ontology of image schemas; we note that the set of basic actions is unlikely to extend indefinitely.

7 ACKNOWLEDGEMENTS

The research reported in this paper has been (partially) supported by the German Research Foundation DFG, as part of Collaborative Research Center (Sonderforschungsbereich) 1320 "EASE - Everyday Activity Science and Engineering", University of Bremen (http://www.ease-crc.org/). The research was conducted in subproject P1: Embodied Semantics for the Language of Action and Change.

REFERENCES

[1] John Bateman. 2010. Situating Spatial Language and the Role of Ontology: Issues and Outlook. *Language and Linguistics Compass* 4, 8 (2010), 639–664. https://doi.org/10.1111/j.1749-818X.2010.00226.x

[2] John A. Bateman. 1997. Enabling Technology for Multilingual Natural Language Generation: The KPML Development Environment. *Nat. Lang. Eng.* 3, 1 (March 1997), 15–55.

[3] John A. Bateman, Joana Hois, Robert Ross, and Thora Tenbrink. 2010. A linguistic ontology of space for natural language processing. *Artificial Intelligence* 174, 14 (2010), 1027 – 1071. https://doi.org/10.1016/j.artint.2010.05.008

[4] John A. Bateman, Robert T. Kasper, Johanna D. Moore, and Richard Whitney. 1990. *A General Organization of Knowledge for Natural Language Processing: The Penman Upper Model*. Technical Report. USC/ISI.

[5] J. A. Bateman, B. Magnini, and G. Fabris. 1995. The Generalized Upper Model knowledge base: organization and use. In *Towards Very Large Knowledge Bases: Knowledge Building and Knowledge Sharing*, N.J.I. Mars (Ed.). IOS Press.

[6] Michael Beetz, Dominik Jain, Lorenz Mösenlechner, Moritz Tenorth, Lars Kunze, Nico Blodow, and Dejan Pangercic. 2012. Cognition-Enabled Autonomous Robot Control for the Realization of Home Chore Task Intelligence. *Proc. IEEE* 100, 8 (2012), 2454–2471.

[7] Benjamin K. Bergen and Nancy Chang. 2003. *Embodied Construction Grammar in Simulation-Based Language Understanding*. Technical Report. Östman, J.-O., & Fried, M. (eds): CONSTRUCTION GRAMMAR(S): COGNITIVE AND CROSS-LANGUAGE DIMENSIONS. JOHN BENJAMIN PUBL CY.

[8] Johan Bos and Tetsushi Oka. 2007. A spoken language interface with a mobile robot. *Artificial Life and Robotics* 11, 1 (2007), 42–47.

[9] Ellen Dodge, Sean Trott, Luca Gilardi, and Elise Stickles. 2017. Grammar Scaling: Leveraging FrameNet Data to Increase Embodied Construction Grammar Coverage. In *Proceedings of the AAAI Spring Symposium Series*.

[10] Manfred Eppe, Sean Trott, and Jerome Feldman. 2016. Exploiting deep semantics and compositionality of natural language for Human-Robot-Interaction. In *Intelligent Robots and Systems (IROS), 2016 IEEE/RSJ International Conference on*. IEEE, 731–738.

[11] Germund Hesslow. 2012. The current status of the simulation theory of cognition. *Brain Research* 1428 (2012), 71–79.

[12] Lars Kunze and Michael Beetz. 2017. Envisioning the qualitative effects of robot manipulation actions using simulation-based projections. *Artificial Intelligence* 247, Supplement C (2017), 352 – 380. Special Issue on AI and Robotics.

[13] Carol Madden, Michel Hoen, and Peter Ford Dominey. 2010. A cognitive neuroscience perspective on embodied language for human-robot cooperation. *Brain and Language* 112, 3 (2010), 180 – 188. Embodied Language Processing: Neuroimaging, Behavioural, and Neurocomputational Perspectives.

[14] Cynthia Matuszek, Evan Herbst, Luke Zettlemoyer, and Dieter Fox. 2013. *Learning to Parse Natural Language Commands to a Robot Control System*. Springer International Publishing, Heidelberg, 403–415.

[15] Drew Mcdermott. 1993. *A Reactive Plan Language*. Technical Report. Yale.

[16] Dipendra Kumar Misra, Jaeyong Sung, Kevin Lee, and Ashutosh Saxena. 2014. Tell Me Dave: Context-Sensitive Grounding of Natural Language to Manipulation Instructions. In *Proceedings of Robotics: Science and Systems*. Berkeley, USA.

[17] Lorenz Mösenlechner and Michael Beetz. 2013. Fast Temporal Projection Using Accurate Physics-Based Geometric Reasoning. In *Proceedings of the IEEE International Conference on Robotics and Automation (ICRA)*. Karlsruhe, Germany, 1821–1827.

[18] Daniel Nyga and Michael Beetz. 2012. Everything Robots Always Wanted to Know about Housework (But were afraid to ask). In *2012 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*. Vilamoura, Portugal.

[19] Daniel Nyga and Michael Beetz. 2015. Cloud-based Probabilistic Knowledge Services for Instruction Interpretation. In *International Symposium of Robotics Research (ISRR)*. Sestri Levante (Genoa), Italy.

[20] Mihai Pomarlan, Daniel Nyga, Mareike Picklum, Sebastian Koralewski, and Michael Beetz. 2017. Deeper Understanding of Vague Instructions through Simulated Execution (Extended Abstract). In *Proceedings of the 2017 International Conference on Autonomous Agents & Multiagent Systems (AAMAS '17)*. International Foundation for Autonomous Agents and Multiagent Systems.

[21] Freek Stulp, Andreas Fedrizzi, Lorenz Mösenlechner, and Michael Beetz. 2012. Learning and Reasoning with Action-Related Places for Robust Mobile Manipulation. *Journal of Artificial Intelligence Research (JAIR)* 43 (2012), 1–42.

[22] David Vernon, Michael Beetz, and Giulio Sandini. 2015. Prospection in cognition: the case for joint episodic-procedural memory in cognitive robotics. *Frontiers in Robotics and AI* 2, 19 (2015), 1–14.

[23] Kai Welke, Peter Kaiser, Alexey Kozlov, Nils Adermann, Tamim Asfour, Mike Lewis, and Mark Steedman. 2013. Grounded spatial symbols for task planning based on experience. In *Humanoids*.

[24] Jan Winkler, Asil Kaan Bozcuoglu, Mihai Pomarlan, and Michael Beetz. 2017. Task Parametrization through Multi-modal Analysis of Robot Experiences (Extended Abstract). In *Proceedings of the 2017 International Conference on Autonomous Agents (AAMAS '17)*. Sao Paulo, Brazil.

[25] Jan Winkler, Moritz Tenorth, Asil Kaan Bozcuoglu, and Michael Beetz. 2014. CRAMm – Memories for Robots Performing Everyday Manipulation Activities. *Advances in Cognitive Systems* 3 (2014), 47–66.