# From Natural Language Instructions to Structured Robot Plans

Mihai Pomarlan(✉), Sebastian Koralewski, and Michael Beetz

Institute for Artificial Intelligence, Universität Bremen, Bremen, Germany
pomarlan@uni-bremen.de

**Abstract.** Research into knowledge acquisition for robotic agents has looked at interpreting natural language instructions meant for humans into robot-executable programs; however, the ambiguities of natural language remain a challenge for such "translations". In this paper, we look at a particular sort of ambiguity: the control flow structure of the program described by the natural language instruction. It is not always clear, when more conditional statements appear in a natural language instruction, which of the conditions are to be thought of as alternative options in the same test, and which belong to a code branch triggered by a previous conditional. We augment a system which uses probabilistic reasoning to identify the meaning of the words in a sentence with reasoning about action preconditions and effects in order to filter out non-sensical code structures. We test our system with sample instruction sheets inspired from analytical chemistry.

## 1 Motivation

A current topic of research is the acquisition of knowledge for robotic agents, aimed at enabling them to perform more complex manipulation tasks. One method, explored in previous work [1], is to mine "how to" resources, such as wikiHow, for recipes and instructions for various activities. The benefits of having robotic agents capable to understand natural language instructions are obvious. On one hand, sites like wikiHow already contain a wealth of information about many activities; on the other, it helps usability if a human can instruct a robot as they would another human.

However, the state of the art is still far from agents robustly capable of understanding natural language. Humans, relying on their already rich commonsense knowledge and experience, can tolerate much more ambiguity and underspecification in their communication than machines can. Work on resolving ambiguities and inferring missing information is ongoing [1,2], but has focused so far on instructions with a simple structure that can be represented as a sequence of steps with ambiguous parameters.

In this paper, we are concerned with instructions with more complex structures caused by the presence of (the natural language equivalent of) program flow controls such as conditionals and loops, which create ambiguities of structure.

DOI: 10.1007/978-3-319-67190-1_30

```
pipette naoh analyte
if (color analyte brown) then
  say iron
else if (color analyte white) then
  pour naoh analyte
   if (color analyte clear) then
     say aluminum
   else if (color analyte white) then
     say magnesium
```

```
pipette naoh analyte
if (color analyte brown) then
  say iron
  if (color analyte white) then
    pour naoh analyte
      if (color analyte clear) then
          say aluminum
            if (color analyte white) then
              say magnesium
```

**Fig. 1.** Two possible interpretations for the metal cation identification instruction

Consider the following text, describing a procedure to identify metal cations in a solution: *add three drops of NaOH to the solution. If a brown precipitate appears, say the solution contains Iron. If a white precipitate appears, add five mL of NaOH. If the precipitate disappears, say the solution contains Aluminum. If the precipitate remains, say the solution contains Magnesium.* Two possible ways to interpret this text into a program are given in Fig. 1, and there are other ways as well.

These two programs, though consistent with the operations enumerated in natural language, behave very differently. Nevertheless, a human can tell the second program is wrong, even without chemistry knowledge. The purpose of the procedure described in the example is to identify the metal ions in a solution. Once the ion has been identified, therefore the goal has been reached, the program should be over.

The example above suggests that at least some ambiguity in program structure can be resolved through knowledge of a task's goal, and/or its component actions in terms of preconditions and effects. It is this intuition we examine here.

## 2 Overview

We consider the problem of turning a text written in natural language into a simple structured program (a "code tree") that may contain simple statements, conditionals (if..else if..else..end if structures) and loops. Currently, we support arbitrarily complex conditionals, and loop-while/untils with one instruction in their body.

First, the natural language text is fed into a probabilistic inference system called PRAC [1] which is used to identify the meanings of words and coreference pronouns. PRAC uses Markov logic networks to represent a probability distribution on how various action requests are formulated; the networks are created from training on large text corpora. Interpreting a text is formalized as a probabilistic query: finding the most likely action(s) requested given the natural languge text as evidence. This produces a list of so-called "action cores", action descriptions in terms of roles and values. Based on this list of action cores, the system produces a list of candidate code trees, which are then validated based on a STRIPS-like procedure that checks whether action preconditions are met at all points of the code tree. The STRIPS validation is used as a way to add more knowledge and help disambiguating between structures for the code trees.

## 3 Representing Actions and World States

We use disjunctive normal form expressions (dnf-expressions) to represent world states, action pre- and postconditions, and "ifconditions" (the conditions appearing in an if or loop statement). A "term" is a simple statement about the world state or its negation (for example, $(STATE\ switch\ on)$ and $(NOT\ (STATE\ switch\ on))$ are terms). A "clause" is a conjunction of terms. A dnf-expression is then a disjunction of clauses.

We say that a clause is consistent (with itself) when it doesn't contain both a term and its negation; we will thereafter assume all clauses we work with are self-consistent, except for postcondition clauses. We say two clauses are consistent if there is no term appearing in one clause that appears negated in the other. We say that clause A includes clause B if they are consistent and all terms appearing in B also appear in A. The **world state** is represented by a dnf-expression in which each clause is a possible world. The world state is updated during code tree validation, based on the postconditions of the actions in the code tree. We use open-world semantics: if something is not stated, in a possible world, to be true or false, then it is unknown in that world.

A **precondition** is represented by a dnf-expression. When validating an action in a code tree, we say that the action is valid if its precondition is known to be true in all worlds that are possible when the action is encountered in the code tree: for every possible world W, there exists a clause P in the precondition such that W includes P. If an action is invalid when it is encountered in the code tree, then this counts as an error and the code tree is considered invalid and rejected.

An **ifcondition** is represented by a dnf-expression. When checking an if or loop instruction in the code tree, we say that the instruction is meaningful if its ifcondition is consistent with at least one of the worlds possible when the instruction is met: there exists a possible world W, such that there exists a clause P in the ifcondition, such that W and P are consistent with each other. If there is no such possible world W, then this counts as a warning, which we currently treat as a reason to reject a code tree.

A **postcondition** is represented by what is syntactically a dnf-expression; this allows our actions to have several sets of possible effects, such as different reactions which may be observed between a known and an unknown reagent. However, we interpret a postcondition dnf differently from other expressions above. In particular, clauses are allowed to be inconsistent and their order matters. When performing an update on a possible world, terms about the world are added in the order in which they appear in the postcondition clause, and replace previous contradicting terms. For example, a postcondition branch $(AND\ (NOT\ (STATE\ s\ off))\ (NOT\ (STATE\ s\ on))\ (STATE\ s\ ?new))$ first makes sure the entity s will no longer be in either on or off states, and then puts entity s in state ?new (which can be, for example, on).

A **terminal condition** is a dnf expression we use to represent a goal state, after which no more statements are expected. A terminal condition is considered achieved if there is at least one possible world in which it holds: there is a possible world W, such that there is a clause P in the terminal condition, such that W includes P. The reason why we ask whether a satisfying possible world exists (rather than requiring the terminal condition to hold everywhere) is to guarantee that instructions achieving the terminal condition are the last executed in a code tree.

Note that while our code tree validation procedure is inspired by STRIPS, we go beyond it by using open-world semantics and actions with more sets of possible effects. This is important, since we aim for an approach usable in environments that are only partially known, and where the effects of actions are unpredictable.

## 4 Code Tree Generation and Validation

As our motivating example shows, knowledge about the basic actions in a program can help check that the program's structure makes sense. We chose a STRIPS representation for actions because it is easy to use. Note, we do not do STRIPS planning; we obtain candidate "plans" from interpreting the input natural language instruction, where a plan may not be just a sequence (as would be the case with STRIPS plans) but instead a structured program with branches and loops. A program is considered valid if the procedure described below comes to the conclusion that at every step of the program, preconditions for the current action to perform are met.

Our procedure is to generate all possible code trees from a list of action cores returned by Prac, then discard from this list all code trees that produce errors (preconditions not met) or warnings (meaningless ifconditions). In the future, we will merge the validation and generation steps, for efficiency.

In order to validate a code tree, we first "unroll" all the loops present: a loop is replaced with an IF statement (with ifcondition being the negation of the loop termination condition), where the body of this IF is the body of the loop repeated twice, followed by an assertion of the loop's termination condition. Our STRIPS-like validation, which we will call cs-validation here, is defined as follows:

- sequences without control structures (branches or loops) are cs-valid if they are valid STRIPS sequences (no invalid actions, no action after terminal condition met)
- an IF and its branches are cs-valid if each branch is cs-valid, and if the ifcondition is meaningful (consistent with at least one possible world)
- sequences with control structures are valid if each of their elements are cs-valid

While the validation procedure traverses the code tree, it updates the world state based on postconditions and assertions about the world state. Postconditions and assertions are applied to all possible worlds at a point in a program (all

```
                      "nested"                "sequential"              "else-if"
(if ifcondition-1)  | if ifcondition-1 then  | if ifcondition-1 then | if ifcondition-1 then
(block-1)           |    block-1             |    block-1            |    block-1
(if ifcondition-2)  |     if ifcondition-2 then | if ifcondition-2 then | else if ifcondition-2 then
(block-2)           |       block-2          |    block-2            |    block-2
```

**Fig. 2.** Converting a list of action cores (left) into possible code trees (right)

clauses in the world state dnf expression). Entering an IF branch also affects the world state: for instructions inside the IF body, only possible worlds consistent with its ifcondition are considered.

Our procedure takes as input: the code tree to validate; a domain description (action preconditions and effects); an initial world state; optionally, a terminal condition.

We will now look at structural ambiguities caused by conditionals. Figure 2 shows the three possible structures that are consistent with a list of action cores containing two conditional instructions (for conciseness, the action cores are simplified). More structures are possible in the sequential case: if more statements appear in block-1, there are several ways to split it into statements appearing in, and outside of the IF body, but for ease of exposition, we will focus here on these three cases.

We will refer to the postconditions of block-1 as the changes to the world state done by the postconditions of the actions in block-1, and by the selection of possible worlds made by ifcondition-1. Similarly, preconditions of block-2 will mean here both the preconditions of the actions in block-2, and ifcondition-2.

A "nested" structure can be unambiguously selected when the preconditions of block-2 depend on the postconditions of block-1 to be valid/meaningful. An "else-if" structure can be unambiguously selected when block-1 achieves the terminal condition, or the preconditions of block-2 would be invalidated by the postconditions of block-1. An "else-if" structure can also be explicitly invoked in natural language (for example by statements such as "otherwise, if") and we take this into account when generating candidate code trees: when it is clear from the language that a conditional appears as an ELSE-IF branch of some previous conditional, we mark it as such.

Currently, our approach does not have a way to unambiguously select the sequential case. If the preconditions of block-2 are unaffected by block-1, then both nested and sequential interpretations are still cs-valid; this will be shown in the evaluation, below.

## 5 Evaluation

To test cs-validation as a method to disambiguate code structures we have used several instruction sheets inspired by analytical chemistry. For page count reasons we don't include the action pre-/postconditions here, but these are available on request; they are currently hand-coded, but we will look at more autonomous ways to acquire them. For each instruction sheet, we generate a list of code trees, from which we then remove the code trees that generate errors or warnings during cs-validation. When more code trees remain in the list, we also look at the world state after each code tree is run.

An example instruction sheet is storage: *if the jar is sealed, put it into the fridge. If the jar is empty, then open the drawer. If the lid is there, take it and put it on the jar*. This results in 37 candidate code trees, out of which only the one corresponding to the correct interpretation survives cs-validation. In this case, cs-validation is able to uniquely select among the available options to arrange control structures, and it is able to do so despite the large number of candidates present.

Another example instruction sheet we use is base titration: *put a drop of alizarin into the test solution. Put drops of hydrogen chloride into the solution until it turns yellow. If the drop count is less than five, put two drops of litmus in the solution. If the solution turns red, put more drops of the NaOH in the solution until it turns blue*. There are three possible candidates generated for this instruction sheet, out of which two survive cs-validation (given in Fig. 3). These two candidates result in the same set of possible final world states, and both look like plausible interpretations.

```
pipette alyzarin analyte
loop-until (color analyte yellow)
    pipette hcl analyte
if (drop-count hcl small) then
    pipette litmus analyte
    if (color analyte red) then
        loop-until (color analyte blue)
            pipette naoh analyte
```

```
pipette alyzarin analyte
loop-until (color analyte yellow)
    pipette hcl analyte
if (drop-count hcl small) then
    pipette litmus analyte
if (color analyte red) then
    loop-until (color analyte blue)
        pipette naoh analyte
```

**Fig. 3.** CS-valid code trees for base titration

Another example instruction sheet is the metal cation identification, given in Sect. 1. In this case, 55 candidate code trees are generated from the instruction sheet, however only 2 survive cs-validation, which shows its power to prune the candidate set. The surviving code trees are shown in Fig. 4. This time however the two code trees do not result in the same set of possible final world states: only the correct plan would say nothing when the analyte contains neither iron, aluminum, or magnesium.

Ambiguities result when a nested and a sequential structure both survive cs-validation. For base titration this appears benign, but in general the code trees will not behave the same, and some further disambiguation (e.g. via questions to a human) is necessary. Still, cs-validation significantly reduces the number of candidates to disambiguate.

```
pipette naoh analyte
if (color analyte brown) then
  say iron
else if (color analyte white) then
  pour naoh analyte
  if (color analyte clear) then
    say aluminum
  else if (color analyte white) then
    say magnesium
```

```
pipette naoh analyte
if (color analyte brown) then
  say iron
else
  if (color analyte white) then
    pour naoh analyte
  if (color analyte clear) then
    say aluminum
  else if (color analyte white) then
    say magnesium
```

**Fig. 4.** CS-valid code trees for metal cation identification

## 6 Related Work

There has been substantial work in analyzing the meaning of conditionals in natural language [3,4]. Other work has tackled the ambiguity of sentiment analysis in conditionals [5]. We used an intensional interpretation [3], which matches the procedural one from computer programming: a conditioned action is performed iff its condition is true.

Extracting sequences of procedures (without branching) from text has been shown in [6]. Workflows that branch into parallel tracks that may recombine are extracted from text in [7] using a notion of "trace index". These workflows describe deterministic, possibly parallel actions in known environments. There are also natural language interpretation systems to enable dialog between humans and robotic agents [8–10]. However, they are intended for deterministic environments where the initial state is fully observable, and can handle only simple conditionals– a condition, an action, optionally an else with its action, with no nesting.

## References

1. Nyga, D., Beetz, M.: Cloud-based probabilistic knowledge services for instruction interpretation. In: International Symposium of Robotics Research (ISRR), Italy, Sestri Levante (Genoa) (2015)
2. Misra, D.K., Sung, J., Lee, K., Saxena, A.: Tell me dave: context-sensitive grounding of natural language to manipulation instructions. In: Proceedings of Robotics: Science and Systems, Berkeley, USA, July 2014
3. Abbott, B.: Conditionals in English and fopl. In: Shu, D., Turner, K., (eds.) Contrasting Meanings in Languages of the East and West, pp. 579–606. Peter Lang, Oxford (2010)
4. Rothschild, D.: Conditionals and propositions in semantics. J. Philos. Logic **44**(6), 781 (2015)
5. Narayanan, R., Liu, B., Choudhary, A.: Sentiment analysis of conditional sentences. In: Proceedings of the 2009 Conference on Empirical Methods in Natural Language Processing, vol. 1, pp. 180–189. Association for Computational Linguistics (2009)
6. Dufour-Lussier, V., Le Ber, F., Lieber, J., Nauer, E.: Automatic case acquisition from texts for process-oriented case-based reasoning. Inf. Syst. **40**, 153–167 (2014)

7. Schumacher, P., Minor, M.: Extracting control-flow from text. In: IRI, pp. 203–210. IEEE (2014)
8. Bos, J., Oka, T.: A spoken language interface with a mobile robot. Artif. Life Robot. **11**(1), 42–47 (2007)
9. Misra, D.K., Tao, K., Liang, P., Saxena, A.: Environment-driven lexicon induction for high-level instructions. In: ACL (1), pp. 992–1002 (2015)
10. Eppe, M., Trott, S., Feldman, J.: Exploiting deep semantics and compositionality of natural language for human-robot-interaction. In: 2016 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS), pp. 731–738. IEEE (2016)