# Formal Dialogue Model for Language Grounding Error Recovery

Natasha Danas

May 15, 2019

**Abstract**

To enable humans to talk to robots, natural language (NL) commands need to be grounded into a more decidable goal specification, such as linear temporal logic (LTL), which the robot can then execute. While no language grounding technique currently boasts probable approximate correctness (PAC) guarantees, the incorrect groundings of NL commands may not be entirely inaccurate all of the time. If we expand our view from one grounding to a set of grounding variants, formal methods techniques and the human in the loop can enable the robot to perform grounding repair, as long as at least one correct goal specification is in the set. Maximal semantic differencing allows the robot to ask clarifying questions about the grounding variants via maximally independent concrete examples, instead of logical forms the user cannot be expected to understand. The user can then provide feedback about the examples, that clarifies which hidden LTL grounding is the correct goal specification for their NL command. When the grounding technique can meet this 1-correct-in-top-n assumption, and the n is not too large to over burden the maximal semantic differencing or human in the loop, the robot can now repair itself in a wide array of contexts that would otherwise be failures. The human clarifications can be fed back into the grounding model, to enable the robot to learn through conversation with the human indefinitely.

## 1 Introduction

To enable humans to talk to robots, NL commands need to be grounded into a more decidable goal specification, which the robot can then execute. The current state of the art grounds these commands into LTL [6]. A Partially Observable Markov Decision Process (POMDP) then interprets the goal specification into a task instance: concrete examples of the LTL formula being satisfied (positive) or not satisfied (negative) through particular robot behavior. The commands are grounded via a neural network sequence-to-sequence model, that is trained to handle arbitrary NL commands, using data from a crowd-sourced corpus. Crowd-source workers are shown positive and negative task instances, who provide NL commands in return, which are then associated with the hidden LTL formula that produced the task instances. The set of correct NL-LTL groundings is augmented by permuting the domain of the command, such as changing the color of the room.

While the latest, about to be published, sequence-to-sequence model achieves 83% grounding accuracy on commands that it trained on, its grounding accuracy drops to 78% on the held-out commands. However, while no language grounding technique currently boasts PAC guarantees, the incorrect LTL groundings of NL commands may not be entirely inaccurate all of the time. We generate grounding variants and use maximal semantic differencing, enabling the user to repair to a correct LTL goal specification by examining a set of clarifying task instances.

| Example Command | GLTL Expression |
|---|---|
| Go to the green room. | $\Diamond G$ |
| Go into the red room. | $\Diamond R$ |
| Enter blue room via green room. | $\Diamond(G \wedge \Diamond B)$ |
| Go through the yellow or red room, and enter the blue room | $\Diamond((R \vee Y) \wedge \Diamond B)$ |
| Go to the blue room but avoid the red room. | $\Diamond B \wedge \neg\Box R$ |
| While avoiding yellow navigate to green. | $\Diamond G \wedge \neg\Box Y$ |
| Scan for blocks and insert any found into bin. | $\Box((S\mathcal{U}\neg A) \wedge \Diamond A)$ |
| Look for and pick up any non red cubes and put them in crate. | $\Box((S\mathcal{U}\neg N_R) \wedge \Diamond N_R)$ |

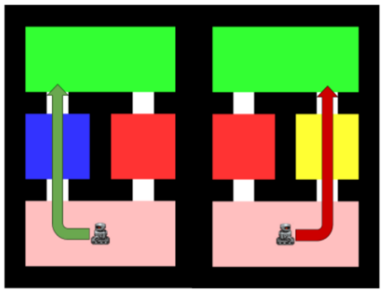


Figure 1: Sample Corpus, followed by a positive and negative task instances of $\Diamond G \wedge \neg\Box Y$.

Within the sequence-to-sequence model, grounding variants are generated by computing the k-most-likely LTL encodings of each token in the originating NL n-token sequence. We product each token encoding together, to produce a large batch of variants.

Standard semantic differencing is used for change impact analysis between only two specifications. For example, semantic differencing can be used to look at the change impact between two firewall policies: what packets are now dropped that used to be accepted, and vice-versa [4]. We define maximal semantic differencing as change impact analysis between any number of specifications. Our implementation enables us turn a set of LTL formulae into a set of task instances that describe each formula most independently from the others. If even one task instance satisfies their NL command, the robot can use the LTL goal specification(s) described by the task instance to repair what would otherwise be a failure.

However, we have made a strong assumption: the possibly correct grounding set contains at least one correct LTL goal specification. We would like to make the set as large as possible, to maximize our chances of repair, without maximal semantic differencing taking too long to report back to the user, or the user being over-burdened with examples. We measure the effectiveness of this approach by answering the following research questions:

1. How does increasing the number of grounding variants increase accuracy?
2. How do the number of grounding variants and grid-world environment affect the time to perform maximal semantic differencing?

We evaluate grounding variant accuracy by computing the position of the correct LTL grounding in the variants, for each NL command. The data is cross-validated by isolating a test set, 20% of all the collected NL commands, outside of the training set. The distribution of correct variant positions is calculated for both data sets. Maximal semantic differencing performance is evaluated over three hand written grid worlds, of varying grid sizes and number of groundings.

## 2 Related Work

Lignos, Hadas Kress-Gazit, and others [8] define a similar formal dialogue to our approach, allowing a user to control a robot through a search-and-rescue environment via NL commands. However, in the case of a grounding error, their dialogue only points out the parts of the NL specification that need to be restated. Instead, we present concrete examples to clarify the groundings, to avoid asking the user to continually restate their intentions.

Boteanu, Hadas Kress-Gazit, and others [2], present a grounding model that can not only synthesize full robot controllers, but verify that the resulting controllers respect hard-coded assumptions about the environment, and the interpretation of the user's stated goals. However, the verification stage is dependent only on the instructions stated by the user, and cannot determine whether the instructions progress towards the user's intended goal. While we have not developed a synthesizer from high level goal specifications to low level controllers, we can do so according to their approach. Our approach will improve accuracy of interpreting the users intentions, putting us one step closer to actually verifying whether the instructions progress towards the correct goal.

Boteanu, Hadas Kress-Gazit, and others [1] use a formal methods approach to perform goal specification repair for unsatisfiable scenarios only, using hard-coded natural language interactions to weaken contradictory assumptions about the environment. In most cases, satisfiability is not enough context to detect semantic errors. That is, even incorrectly grounded goals are often still satisfiable, and possibly equisatisfiable to a correct grounding, for a given environment. Our approach covers this ignored majority of semantic error cases.

## 3 Technical Approach

### 3.1 Grounding Variant Generation

Within the sequence-to-sequence model, grounding variants are generated by computing the k-most-likely LTL encodings of each token in the originating NL n-token sequence. The training of the decoder and encoder, as well as the decoding-encoding algorithm are otherwise unchanged. We product each token encoding together, to produce at most $n^k$ LTL formulae: a majority of which are not grammatically correct, due to the nature of the language model.

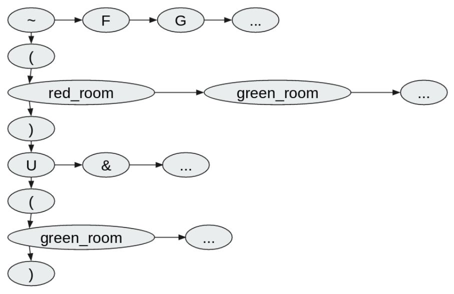


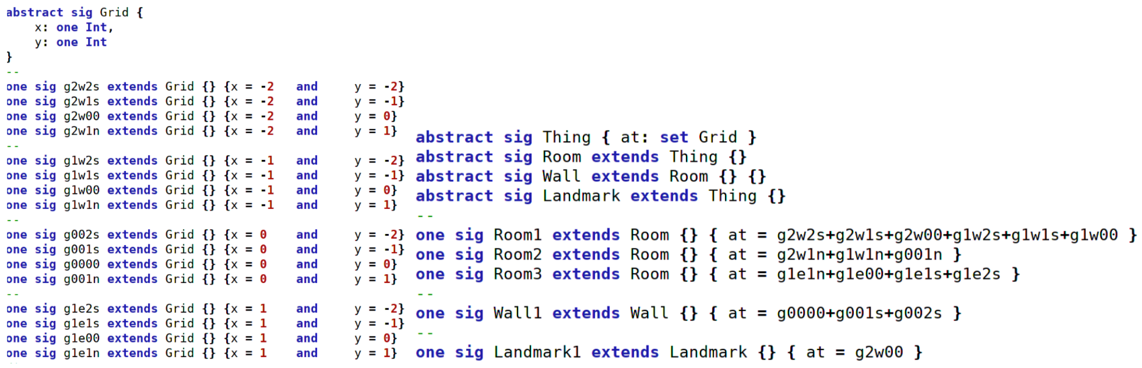
Figure 2: Token variants for "Go to the green room only after you go through the red room." which should be grounded to "F ( red_room ) U ( green_room )"

In fact, only 80% of the top-1 groundings end up being grammatically correct LTL: even when the encoding-decoding algorithm remains unchanged, sequence-to-sequences models do not necessarily embed natural language sequences into valid LTL expressions. Because of this, we avoid token variants that impact the embedding structure, by not varying opening and closing parentheses.

## 3.2 Standard Semantic Differencing

Semantic differencing is a specification modelling technique used within a class of formal methods tools called model finders, such as Alloy [7]. One can use a model finder to specify the robot's environment and behavior, then find models: concrete task instances that satisfy a goal specification. Note, the stochastic or partially-observable features are left out of the environment specification, as these tools are useful for knowledge bases and high-level planning– not for real-time task and motion planning which is left to POMDP-based approaches.

To introduce standard semantic differencing, let us consider two grounding variants for "avoid the blue room until you go to landmark 1": the correct grounding ((¬blue_room) U landmark_1) and the incorrect grounding ((¬landmark_1) U blue_ room).



```
abstract sig Grid {
    x: one Int,
    y: one Int
}
--
one sig g2w2s extends Grid {} {x = -2   and     y = -2}
one sig g2w1s extends Grid {} {x = -2   and     y = -1}
one sig g2w00 extends Grid {} {x = -2   and     y = 0}
one sig g2w1n extends Grid {} {x = -2   and     y = 1}
--
one sig g1w2s extends Grid {} {x = -1   and     y = -2}
one sig g1w1s extends Grid {} {x = -1   and     y = -1}
one sig g1w00 extends Grid {} {x = -1   and     y = 0}
one sig g1w1n extends Grid {} {x = -1   and     y = 1}
--
one sig g002s extends Grid {} {x = 0    and     y = -2}
one sig g001s extends Grid {} {x = 0    and     y = -1}
one sig g0000 extends Grid {} {x = 0    and     y = 0}
one sig g001n extends Grid {} {x = 0    and     y = 1}
--
one sig g1e2s extends Grid {} {x = 1    and     y = -2}
one sig g1e1s extends Grid {} {x = 1    and     y = -1}
one sig g1e00 extends Grid {} {x = 1    and     y = 0}
one sig g1e1n extends Grid {} {x = 1    and     y = 1}
```

```
abstract sig Thing { at: set Grid }
abstract sig Room extends Thing {}
abstract sig Wall extends Room {} {}
abstract sig Landmark extends Thing {}
--
one sig Room1 extends Room {} { at = g2w2s+g2w1s+g2w00+g1w2s+g1w1s+g1w00 }
one sig Room2 extends Room {} { at = g2w1n+g1w1n+g001n }
one sig Room3 extends Room {} { at = g1e1n+g1e00+g1e1s+g1e2s }
--
one sig Wall1 extends Wall {} { at = g0000+g001s+g002s }
--
one sig Landmark1 extends Landmark {} { at = g2w00 }
```

Figure 3: Alloy definition of a 4x4 grid on the left, with the rooms/walls/landmarks on the right.

First we define the grid world, which consists of a set of grid coordinates. Each grid coordinate has an x and y position, represented as an Integer. Then we define the objects in the grid world, which all reside in a subset of the grid coordinates. Finally we define our robot agent, who can move location over time. We have to constrain the robot's starting position, that they can never move into walls, and that at every time step they must stay or move in one direction.

We also express the LTL goals in first order logic, which luckily has universal and existential quantification over time to make up for not supporting native LTL quantification. We can then semantically analyze and differentiate these goals by asking Alloy for task instances that satisfy both (describes commonality), and just one of each instance (describes independence). In this case, the first command (A and B) is satisfiable, the second command (A and not B) is unsatisfiable, and the third command (B and not A) is satisfiable.

By the results of these three commands, we can conclude that $notRoom1UntilLandmark1$ implies $notLandmark1UntilRoom1$ for this specific environment: due to the fact $Landmark1$ is located in $Room1$. For other pairs of LTL goal specifications or other environments, we may

```
sig Time {}
one sig Robot { where: Time -> one Grid } {
    where[first] = g1e2s
    all t:Time | where[t] not in Wall.at
    all t:Time-last | stay[t,t.next] or west[t,t.next] or east[t,t.next] or north[t,t.next] or south[t,t.next]
}
pred stay[t:Time, st:Time] { Robot.where[st] = Robot.where[t] }
pred west[t:Time, st:Time] { Robot.where[st].y = Robot.where[t].y and Robot.where[st].x = minus[Robot.where[t].x, 1] }
pred east[t:Time, st:Time] { Robot.where[st].y = Robot.where[t].y and Robot.where[st].x = add[Robot.where[t].x, 1] }
pred north[t:Time, st:Time] { Robot.where[st].x = Robot.where[t].x and Robot.where[st].y = add[Robot.where[t].y, 1] }
pred south[t:Time, st:Time] { Robot.where[st].x = Robot.where[t].x and Robot.where[st].y = minus[Robot.where[t].y, 1] }
--
pred notRoom1UntilLandmark1 { some u:Time | {
    all t:u.^prev-u | Robot.where[t] not in Room1.at
    Robot.where[u] in Landmark1.at
}}
pred notLandmark1UntilRoom1 {  some u:Time | {
    all t:u.^prev-u | Robot.where[t] not in Landmark1.at
    Robot.where[u] in Room1.at
}}
run {notRoom1UntilLandmark1 and notLandmark1UntilRoom1} for 2 Int, 10 Time
run {notRoom1UntilLandmark1 and (not notLandmark1UntilRoom1)} for 2 Int, 10 Time
run {(not notRoom1UntilLandmark1) and notLandmark1UntilRoom1} for 2 Int, 10 Time
```

Figure 4: Alloy definition of robot agent, two LTL variants, and solver commands.

instead conclude that some goal specifications are unsatisfiable (have no satisfying task instances), are equisatisfiable (same set of task instances), are partially independent (some common and independent task instances), or some goals are completely independent (only independent task instances).

## 3.3 Maximal Semantic Differencing

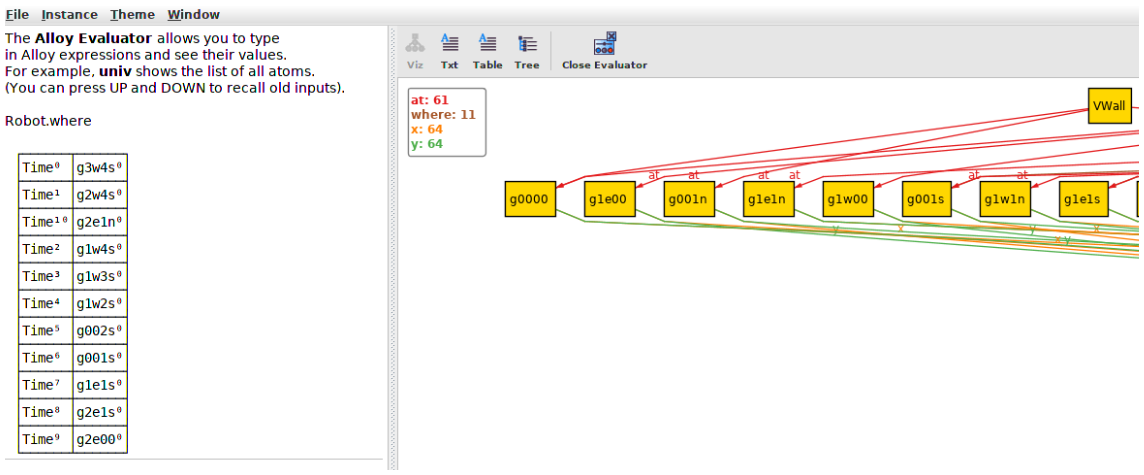
We define maximal semantic differencing as change impact analysis between any number of specifications, as opposed to just two. Our implementation enables us turn a set of LTL formulae into a set of task instances that describe each formula most independently from the others. We do this by extending Alloy to support soft constraints, which can be optionally satisfied unlike the usual hard constraints.

```
// satisfy ~ ( red_room ) U ( green_room ), minimize satisfaction of others
run {
    Robot.where[first] = g3w4s
    notRedRoomUntilGreenRoom
    soft (not eventuallyRedRoomUntilGreenRoom)
    soft (not eventuallyGreenRoomUntilGreenRoom)
    soft (not notRedRoomAndGreenRoom)
    soft (not eventuallyRedRoomAndGreenRoom)
    soft (not notGreenRoomAndGreenRoom)
} for 3 Int, 11 Time, 5 Thing
// satisfy F ( red_room ) U ( green_room ), minimize satisfaction of others
run {
    Robot.where[first] = g3w4s
    soft (not notRedRoomUntilGreenRoom)
    eventuallyRedRoomUntilGreenRoom
    soft (not eventuallyGreenRoomUntilGreenRoom)
    soft (not notRedRoomAndGreenRoom)
    soft (not eventuallyRedRoomAndGreenRoom)
    soft (not notGreenRoomAndGreenRoom)
} for 3 Int, 11 Time, 5 Thing
// satisfy F ( green_room ) U ( green_room ), minimize satisfaction of others
run {
    Robot.where[first] = g3w4s
    soft (not notRedRoomUntilGreenRoom)
    soft (not eventuallyRedRoomUntilGreenRoom)
    eventuallyGreenRoomUntilGreenRoom
    soft (not notRedRoomAndGreenRoom)
    soft (not eventuallyRedRoomAndGreenRoom)
    soft (not notGreenRoomAndGreenRoom)
} for 3 Int, 15 Time, 5 Thing
```

```
// satisfy ~ ( red_room ) & ( green_room ), minimize satisfaction of others
run {
    Robot.where[first] = g3w4s
    soft (not notRedRoomUntilGreenRoom)
    soft (not eventuallyRedRoomUntilGreenRoom)
    soft (not eventuallyGreenRoomUntilGreenRoom)
    notRedRoomAndGreenRoom
    soft (not eventuallyRedRoomAndGreenRoom)
    soft (not notGreenRoomAndGreenRoom)
} for 3 Int, 15 Time, 5 Thing
// satisfy F ( red_room ) & ( green_room ), minimize satisfaction of others
run {
    Robot.where[first] = g3w4s
    soft (not notRedRoomUntilGreenRoom)
    soft (not eventuallyRedRoomUntilGreenRoom)
    soft (not eventuallyGreenRoomUntilGreenRoom)
    soft (not notRedRoomAndGreenRoom)
    eventuallyRedRoomAndGreenRoom
    soft (not notGreenRoomAndGreenRoom)
} for 3 Int, 15 Time, 5 Thing
// satisfy ~ ( green_room ) & ( green_room ), minimize satisfaction of others
run {
    Robot.where[first] = g3w4s
    soft (not notRedRoomUntilGreenRoom)
    soft (not eventuallyRedRoomUntilGreenRoom)
    soft (not eventuallyGreenRoomUntilGreenRoom)
    soft (not notRedRoomAndGreenRoom)
    soft (not eventuallyRedRoomAndGreenRoom)
    notGreenRoomAndGreenRoom
} for 3 Int, 15 Time, 5 Thing
```

Figure 5: Modified Alloy LTL goals and maximal semantic differencing as soft constraints.

Alloy turns each first-order relational logic constraint into propositional logic, via an efficient universe instantiation algorithm in its compiler named Kodkod [10]. To get an intuition for the compilation, quantified formulas can be grounded to quantifier free formulas by simply filling in the quantifiers for each possible combination of atoms in the universe. Additionally, relational algebra and logically connected formulas can be translated into set theory queries. From these grounded formulas, we can produce an equisatisfiable set of propositional formulas, by defining a variable for every possible set theory operation over the universe, and mapping over each operation in the grounded formulas. Kodkod's compact boolean circuit compilation process is an optimization of this naive approach. The resulting propositional logic formulae are then passed to a SAT solver [9], to find a satisfying assignment: the model or task instance.



The **Alloy Evaluator** allows you to type
in Alloy expressions and see their values.
For example, **univ** shows the list of all atoms.
(You can press UP and DOWN to recall old inputs).

Robot.where

| | |
|---|---|
| Time$^{0}$ | g3w4s$^{0}$ |
| Time$^{1}$ | g2w4s$^{0}$ |
| Time$^{10}$ | g2e1n$^{0}$ |
| Time$^{2}$ | g1w4s$^{0}$ |
| Time$^{3}$ | g1w3s$^{0}$ |
| Time$^{4}$ | g1w2s$^{0}$ |
| Time$^{5}$ | g002s$^{0}$ |
| Time$^{6}$ | g001s$^{0}$ |
| Time$^{7}$ | g1e1s$^{0}$ |
| Time$^{8}$ | g2e1s$^{0}$ |
| Time$^{9}$ | g2e00$^{0}$ |

Figure 6: Satisfying model for the first maximal semantic difference in the previous figure.

We enable soft constraint solving by extending the underlying SAT solver to MaxSAT [5], which allows us to define soft propositional constraints, that may or may not be satisfied in the resulting model. Our particular implementation uses Z3 for MaxSAT, a performant theorem prover with a wide array of features [3]. We enable soft constraint expression in Alloy, by defining a new type of quantified formula (keyword soft) in the lexer, parser, and abstract syntax tree. During the compilation process, Kodkod caches each propositionally grounded subformula as a circuit, with a label, which is a propositional variable only true if the circuit is also true. We simply cache the soft circuits and labels seperately, and make sure to write them out as soft propositional clauses when the MaxSAT solver is invoked. Alloy then produces models of the mix of hard and soft constraints as usual.

## 4 Evaluation

### 4.1 Grounding Variant Accuracy

By expanding our view to the top-10 batch generated variants, we recover correct groundings for 0.77% of the training data and 4.45% for test data. This accuracy can be seen as the area under the peaks in the middle of each line plot. However, in order to generate those top-10, we must generate a space of variants two orders of magnitude greater than this: only 2% of the groundings end up being grammatically correct. This results in a performance hit of a few seconds to generate the thousands of mostly junk grounding variants.

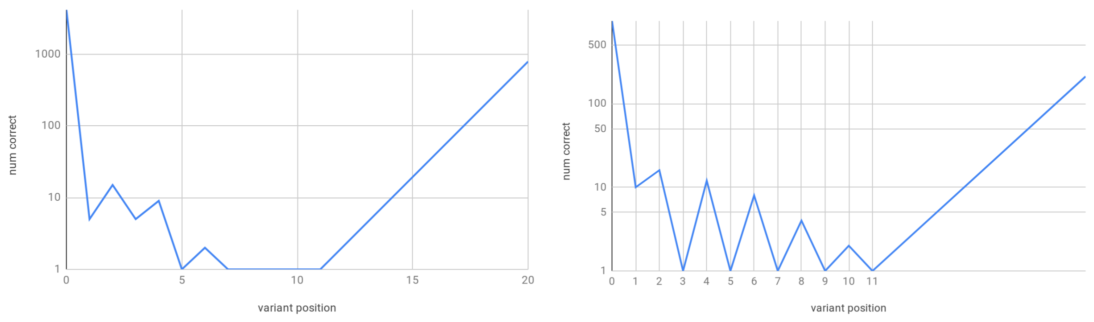


Figure 7: Distribution of positions of correct LTL grounding. Vertical axis is log-scale, the right-most points are the count of still incorrect groundings. Training data on left, test data on right.

For the most part, we are only able to recover from "one-off" semantic errors, where either an LTL operator or primitive (room, landmark, etc) are the only incorrect portion of the grounding. The remaining training errors are more complex, thus this variant generation technique is quite ineffective. However, many of the errors on commands not seen during training are in this class, which is good since we would rather learn over unknown commands rather than known misconceptions.

## 4.2 Maximal Semantic Differencing Performance

We break down maximal semantic differencing performance into three metrics: translation time, UNSAT solve time, and SAT solve time. We pay a cost of one translation time, plus one solve time per variant, depending on whether the query is satisfiable or not.

| Gridworld Size | Trajectory Length (time size) | Number of Variants (queries) | Translation time (average over all queries) | UNSAT Solve time (average over UNSAT queries) | SAT Solve time (average over SAT queries) |
|---|---|---|---|---|---|
| 16 | ~5 | 2 | 166 ms | 202 ms | 528 ms |
| 64 | ~10 | 6 | 1186 ms | 299 ms | 1774 ms |
| 256 | ~20 | 12 | 11 sec | 21 sec | 128 sec |

Figure 8: Performance of Maximal Semantic Differencing for three hand-written specifications.

Quadrupling the size of the grid world (doubling each coordinate space) roughly increases each query by an order of magnitude. UNSAT queries take an order of magnitude less time than SAT queries. Since each query is either UNSAT or SAT, we can view UNSAT solve time as a lower bound, and SAT solve time as an upper bound. So, in order to process 10 variants for an 8x8 grid world, we have to wait 3-20 seconds to produce every maximally independent task instance.

## 5 Conclusion

We expand our view from one NL-LTL grounding to a set of grounding variants, produced by searching through the less likely token encodings in the sequence-to-sequence language model. We perform maximal semantic differencing in a modified model finding process within Alloy, that allows the robot to ask clarifying questions about the grounding variants via maximally independent concrete examples, instead of logical forms the user cannot be expected to understand. While we have yet to produce an full end-to-end implementation, these two ends enable the user to provide feedback about the examples, that clarifies which hidden LTL grounding is the correct goal specification for their NL command. For a reasonably sized 8x8 grid, with 3-20 seconds of the user's time, the robot can now repair itself in 1% of the training cases and 5% of the test cases, which would otherwise be a failure.

While Alloy is great for usability, its age limits it's ability to access the latest in solver performance. Working directly with a theorem prover (such as Z3) or a model checker with model finding capabilities should reduce time to perform maximal semantic differencing. Also, enabling incremental solving of multiple semantic differencing queries will reduce average query time by an order of magnitude, as the SAT solver will not need to re-load and re-solve much of the search problem shared between queries. We may also be able to implement maximal semantic differencing directly into an MDP planner.

The weakest end of this system is currently the variant generation. We will first finish implementing and evaluating beam search, as an alternative to the current batch searching method. Additionally, we will compare the sequence-to-sequence model to other language grounding models. We aim to implement and evaluate against a seq-to-seq extension, and a language model that is more likely to produce grammatically correct variants.

Essentially, this approach allows us to filter incorrect groundings via environment context and some user input. This variant generation and filtering to correct grounding approach becomes more powerful as we develop more sound ways to filter variants through contexts. We will attempt to filter based on spoken context via paraphrasing. By training a duplicate language model in reverse direction, we can embed each variant back into natural language. Due to the differences in the embedding spaces, this will not result in an identity function, and the discrepancies from the identity may be enough to filter even more variants before performing semantic differencing.

## Research Artifacts

- Sequence-to-Sequence Grounding Variant Modifications:
  https://github.com/transclosure/ltl-amdp
- Alloy Soft Clause Modifications:
  https://github.com/transclosure/org.alloytools.alloy/tree/add-objectives
- Maximal Semantic Differencing Specifications:
  https://github.com/transclosure/logic/tree/master/collabrobo

## References

[1] Adrian Boteanu, Jacob Arkin, Siddharth Patki, Thomas Howard, and Hadas Kress-Gazit. Robot-initiated specification repair through grounded language interaction. *arXiv preprint arXiv:1710.01417*, 2017.

[2] Adrian Boteanu, Thomas Howard, Jacob Arkin, and Hadas Kress-Gazit. A model for verifiable grounding and execution of complex natural language instructions. In *2016 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 2649–2654. IEEE, 2016.

[3] Leonardo De Moura and Nikolaj Bjørner. Z3: An efficient smt solver. In *International conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340. Springer, 2008.

[4] Kathi Fisler, Shriram Krishnamurthi, Leo A Meyerovich, and Michael Carl Tschantz. Verification and change-impact analysis of access-control policies. In *Proceedings of the 27th international conference on Software engineering*, pages 196–205. ACM, 2005.

[5] Zhaohui Fu and Sharad Malik. On solving the partial max-sat problem. In *International Conference on Theory and Applications of Satisfiability Testing*, pages 252–265. Springer, 2006.

[6] Nakul Gopalan, Dilip Arumugam, LL Wong, and Stefanie Tellex. Sequence-to-sequence language grounding of non-markovian task specifications. In *Robotics: Science and Systems*, 2018.

[7] Daniel Jackson. *Software Abstractions: logic, language, and analysis*. MIT press, 2012.

[8] Constantine Lignos, Vasumathi Raman, Cameron Finucane, Mitchell Marcus, and Hadas Kress-Gazit. Provably correct reactive control from natural language. *Autonomous Robots*, 38(1):89–105, 2015.

[9] Robert Nieuwenhuis, Albert Oliveras, and Cesare Tinelli. Solving sat and sat modulo theories: from an abstract davis–putnam–logemann–loveland procedure to dpll (t). *Journal of the ACM (JACM)*, 53(6):937–977, 2006.

[10] Emina Torlak and Daniel Jackson. Kodkod: A relational model finder. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 632–647. Springer, 2007.