CoSaTa: A Constraint Satisfaction Solver and Interpreted Language for Semi-Structured Tables of Sentences

Peter A. Jansen
School of Information, University of Arizona, Tucson, AZ, USA
pajansen@email.arizona.edu

Abstract

This work presents CoSaTa, an intuitive constraint satisfaction solver and interpreted language for knowledge bases of semi-structured tables expressed as text. The stand-alone CoSaTa solver allows easily expressing complex compositional "inference patterns" for how knowledge from different tables tends to connect to support inference and explanation construction in question answering and other downstream tasks, while including advanced declarative features and the ability to operate over multiple representations of text (words, lemmas, or part-of-speech tags). CoSaTa also includes a hybrid imperative/declarative interpreted language for expressing simple models through minimally-specified simulations grounded in constraint patterns, helping bridge the gap between question answering, question explanation, and model simulation. The solver and interpreter are released as open source.[1]

1 Introduction

Performing inference for complex question answering typically requires combining multiple facts from a knowledge base to arrive at a correct answer, where this set of facts can then be used to generate detailed human-readable explanations for the reasoning behind those answers. Combining multiple facts to perform natural language inference is extremely challenging, with contemporary methods generally unable to reliably combine more than two facts together. This is a significant limitation, as even elementary science questions require an average of six (and, as many as 16) atomic facts to answer and explain (Jansen et al., 2018; Xie et al., 2020) – particularly when those explanations include detailed world knowledge. For example,

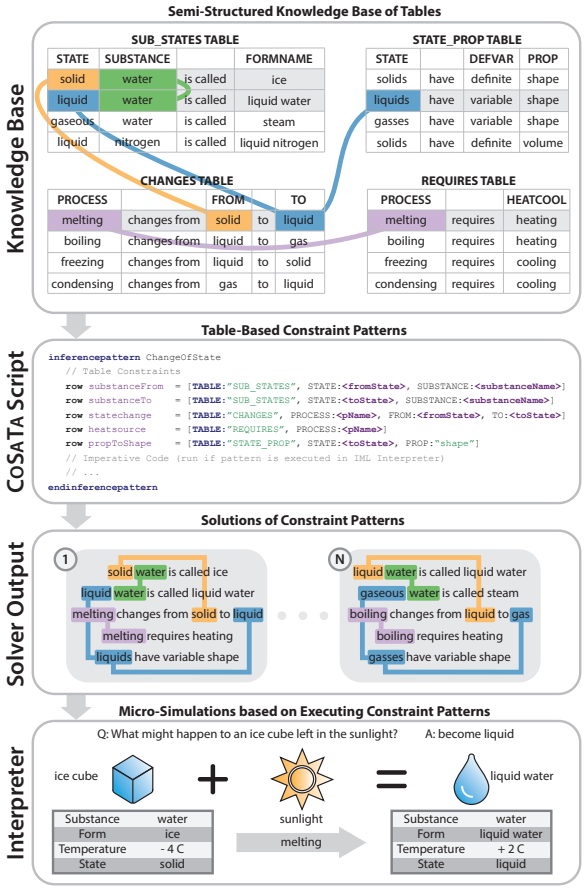

Figure 1: An overview of the proposed system. A semi-structured knowledge base of tables *(top)* serves as input to the CoSaTa scripting language for expressing multi-hop inference patterns as constraints over table rows. A stand-alone constraint solver and full interpreter are provided.

explaining the common process of *an ice cube melting in sunlight* (see Figure 1) can require a large number of facts, especially when those facts are expressed at a fine level of granularity.

Compositional (or *"multi-hop"*) inference solving methods tend to exist on a formality continuum. At one end of the continuum, logical or declarative methods (e.g. Lenat et al., 1990; Forbus, 2019) model a knowledge base as a set of assertions, and inference as sets of axioms and combinatorial rules

[1]Demo: https://youtu.be/t93Acsz7LyE

Proceedings of the 2020 EMNLP (Systems Demonstrations), pages 70–76
November 16-20, 2020.

acting on those assertions. While logical methods provide provably correct inference and detailed explanations, these methods tend to be brittle in practice (MacCartney and Manning, 2007). At the other end of the formality continuum are inference methods that use unstructured text as knowledge (modeled at the level of the word (Fried et al., 2015), sentence (Valentino et al., 2020), or paragraph (Yang et al., 2018)), which is typically combined using connectivity (e.g. Jansen et al., 2017), embedding (e.g. Tu et al., 2020), or other features. Due to the difficulty of combining free text (Jansen, 2018), these methods typically reach peak performance when combining only a small number of facts together – typically two or three.

A middle-ground exists between these two extremes, where semi-structured knowledge bases of text (such as tables) are used to support multi-hop inference (e.g. Sun et al., 2016). This approach offers many practical benefits, such as ease of knowledge base creation (over logical decomposition methods), and providing structure to help infer when combining facts is appropriate (versus free text methods). In spite of these benefits, it is often still challenging to implement inference models that compose (or "hop" between) facts expressed in tables of language data in practice, and practitioners tend to resort to using complex models (such as integer linear programming (e.g. TableILP; Khashabi et al., 2016)) that significantly increase development time and limit interpretability, maintainability, and reuse.

This work presents an easy-to-use scripting language paired with an open source solver and interpreter designed to make compositional inference over semi-structured knowledge bases of tables easy, particularly when those tables express knowledge as lightly structured sentences. The contributions of this work are:

1. The CoSaTa Solver, a stand-alone solver for Constraint Satisfaction over Tables of text. The CoSaTa language allows easily expressing large multi-hop *"inference patterns"* that describe how facts typically connect across tables in a knowledge base to express a larger compositional solution. The optimized multi-threaded solver supports advanced features for dealing with text, including enumerative variable span detection, and robustness to surface form variations with patterns that can match words, lemmas, or parts-of-speech.
2. The CoSaTa Interpreter, a hybrid imperative/declarative interpreted language for modeling simple inferences through minimally-specified simulations grounded in constraint patterns.

2 Language Description

The CoSaTa language includes declarative features for performing *constraint satisfaction over tables*, and imperative features for expressing and executing models. The fundamental unit is the *pattern*, analagous to a *class* in object oriented programming, which (at a minimum) contains a declarative constraint pattern of table rows. The declarative features are sufficient for easily expressing variablized compositional patterns over collections of table rows, and the output of the stand-alone constraint satisfaction solver (e.g. solutions in JSON format) can serve as input to further processing.

The language also supports a suite of imperative features for expressing and executing models. In this paradigm, each pattern is considered a process that, if executed, imparts some change upon a small model of the world (such as *an object warming from heat transfer*) by executing a *pattern code block*. Agents, physical objects, and environments in the model are represented as *objects*, here sets of property-value pairs. A *control script* imports a library of patterns, initializes objects, and executes a small subset of patterns in a particular order to create a simulation. A *state space* keeps a log of each object, it's properties, the patterns executed, and their resulting changes, to form a detailed and human-readable record of a simulation performed to arrive at a particular inference.

2.1 Declarative: Constraints over tables

Table Row Constraints: Each pattern contains one or more *table row constraints*, which collect interconnected sets of facts (table rows) from one or more tables based on satisfying constraints on the content of those facts. Each table row constraint requires: (a) a *name* for the row, (b) a *table* where rows are drawn from, and (c) a list of *variabalized constraint expressions* that specific cells (columns) in a given row must satisfy in order for the entire constraint pattern to be valid. An example pattern with 8 table row constraints surrounding *Changes of State of Matter* is shown in Figure 2.

Constraint Expressions: Constraint expressions for table cells can be expressed as mixtures of

```
// Constraint Pattern: Changing States of Matter
inferencepattern changeStateOfMatter
  // Plain text description
  description = "A substance changing its state of matter"

  // Row definitions
  // e.g. solid/liquid/gas is a kind of state of matter
  row som1 = [TABLE:"KINDOF", HYPONYM:<SOM1>, HYPERNYM:"state of matter"]
  row som2 = [TABLE:"KINDOF", HYPONYM:<SOM2>, HYPERNYM:"state of matter"]

  // e.g. melting/boiling/freezing is a kind of change of state
  row cos  = [TABLE:"KINDOF", HYPONYM:<ChangeOfState>, HYPERNYM:"change of state"]

  // e.g. state of matter is a property of a substance
  row somprop = [TABLE:"PROP-GENERIC", PROPERTY:"state of matter", OBJECT:<obj>]

  // e.g. a boiling point is a kind of phase transition point
  row point = [TABLE:"KINDOF", HYPONYM:<PhaseTransitionPoint>, HYPERNYM:"phase transition point"]

  // e.g. melting means (matter; a substance) changes from a solid to a liquid by increasing heat energy
  row change = [TABLE:"CHANGE", PROCESSNAME:<ChangeOfState>, PROPERTY:"state of matter", OBJECT:<obj>, FROM:<SOM1>, INTO:<SOM2>,
        BY_THROUGH_HOW:<incDec> + "heat energy"]

  // e.g. melting occurs when the temperature of a substance is increased above the substance's melting point, and below it's
       boiling point
  row thresh = [TABLE:"CONDITION-VEC", EVENT:<ChangeOfState>, OBJECT:<obj>, INCREASE_DECREASE:<tempDir>, ABOVE_BELOW1:
       <aboveBelow>, VALUE1:<PhaseTransitionPoint>, ABOVE_BELOW2:<*aboveBelow2>, VALUE2:<*PhaseTransitionPoint2>]

  // e.g. heating means the (temperature; heat energy) of an (object; substance) is increased
  row heatcool = [TABLE:"CHANGE-VEC-PROP", PROCESS_NAME:<heatingOrCooling>, PROPERTY:"heat energy", INCREASE_DECREASE:<incDec>]

endinferencepattern
```

(a) An example listing for a constraint satisfaction pattern that collects 8 facts surrounding Changes of State of Matter.

Row Name	Table Row
som1	a **<solid>** is a kind of "state of matter"
som2	a **<liquid>** is a kind of "state of matter"
cos	**<melting>** is a kind of "change of state"
somprop	"state of matter" is a property of a **<substance>**
point	a **<melting point>** is a kind of "phase transition point"
change	**<melting>** means the "state of matter" of **<substance>** changes from a **<solid>** into a **<liquid>** by **<increasing>** "heat energy"
thresh	**<melting>** occurs when the temperature of a **<substance>** is **<increased>** **<above>** the substance 's **<melting point>**
heatcool	**<heating>** means the "heat energy" of a substance is **<increased>**

(b) An example enumerated solution of the above constraint pattern.

Constraint Variable Name	Value
<aboveBelow>	above
<aboveBelow2>	not populated
<ChangeOfState>	melt
<heatingOrCooling>	heat
<incDec>	increase
<obj>	substance
<PhaseTransitionPoint>	melt point
<PhaseTransitionPoint2>	not populated
<SOM1>	solid
<SOM2>	liquid
<tempDir>	increase

(c) The variable values from this solution.

Figure 2: *(Top)* An example constraint satisfaction pattern expressed in the CoSaTa. *(Bottom)* One of several solutions provided by the CoSaTa solver when evaluating this pattern with a semi-structured knowledge base of tables.

strings (words, lemmas, or part-of-speech tags) and variables. Elements can be combined with simple boolean operations, as well as advanced booleans (e.g. optional elements, enumerative *AND*s that automatically determine variable spans). Example constraint expressions are shown in Table 1.

Inheritance and Composite Patterns: Similar to object oriented programming, patterns can contain their own table row constraints, and/or inherit their table row constraints from one or more other patterns. This enables software engineering practices like problem decomposition into objects to be applied to constraint patterns. For example, the *ChangeOfStateWithSubstanceFromTo* pattern in Table 4 inherits table row constraints from three other patterns: one that describes the general concept of changes of state, another that describes the idea of a substance having a particular melting or boiling point, and a final pattern describing the substance being in it's changed state. Because of this decomposition, these smaller generic patterns are available for reuse in other patterns.

2.2 Imperative: Executable Micro-Models

Objects: Objects are expressed as lists of property-value pairs that can be added, modified, or executed against. For example, an object named *ice cube* might have properties *location:fridge* and *temperature:-4C*.

Pattern Code Block: Each pattern can contain an imperative code block that, if executed, typically imparts changes to the objects or knowledge base suggestive of a particular process having taken place. For example, a *heatTransfer* pattern might have rows that match on any two objects that are touching, while its code block could decrease the temperature of the warmer object, and increase the temperature of the cooler object. Similarly, the code block in Figure 4 changes the state of matter of an object (for example from a solid to a liquid).

Example 1: Single variable assignment
Expression: `ORGANISM: <organismName>`
Cell Text: *"large green plants"*
Variables: `<organismName> = "large green plants"`

Example 2: Boolean AND with variables and strings
Expression: `LOCATION: "in" + <northSouth> + "hemisphere"`
Cell Text: *"in the northern hemisphere"*
Variables: `<northSouth> = "northern"`

Example 3: Matching part-of-speech tags
Expression: `TIME: <month> + "POS:CD"`
Cell Text: *"June 21st"*
Variables: `<month> = "June"`

Example 4: Enumerative AND (multiple adjacent variables)
Expression: `BY_THROUGH: <incDec> + <energy>`
Cell Text: *"increasing heat energy"*
Variables: Two possible enumerations provided for constraint satisfaction:
`<incDec> = "increasing", <energy> = "heat energy"`, or
`<incDec> = "increasing heat", <energy> = "energy"`

Example 5: Optional elements
Expression: `DURATION: <dur1> + *"to" + <*dur2> + "hour"`
Cell Text: *"two to four hours"*
Variables: Two possible enumerations provided for constraint satisfaction:
`<dur1> = "two", <dur2> = "four"`, or
`<dur1> = "two to four", <dur2> = <unpopulated opt>`

Table 1: Example constraint expressions for a given table cell, evaluated against example cell text.

Requirements Specification: Code typically requires certain preconditions to be satisfied for execution to be valid, such as *heatTransfer* requiring two objects that have non-empty *temperature* properties. Patterns have specific functions for verifying that preconditions are met, as well as levels of preconditions (required, or recommended).

Model Control Script: Models take the form of small, easily-composed control scripts that import a library of patterns, initialize the objects required for a model, then sequentially execute a series of patterns that impart changes on those objects. For example, the control script in Figure 3 initializes three objects: an *ice cube*, *freezer*, and *outside environment*. The ice cube begins in the *freezer*, and is then moved to the *outside environment*. *Heat transfer* happens between the *ice cube* and *outside environment* until the *ice cube* meets the conditions for a *Change of State*. The *Change of State* then happens, in this case melting, changing the ice cube's *state of matter* property from *solid* to *liquid*.

3 Solver and Interpreter

Both the constraint satisfaction solver and interpreter are implemented in Scala, with Stanford CoreNLP (Manning et al., 2014) used to provide tokenization, lemmatization, and part-of-speech tags for the knowledge base of tables. The solver pipeline first creates shortlists of rows that may potentially satisfy individual table row constraints for each pattern, then implements a backtrack search (e.g. Davis et al., 1962) to exhaustively find all combinations of table rows that meet the constraints for a given pattern (each unique collection of table rows that satisfies a given pattern is termed a *solution* of that pattern).[2] While some powerful declarative features (such as enumerative ANDs with automatic variable span detection) are expensive to evaluate, nearly all stages of evaluation are multi-threaded for speed, and rely heavily on precomputed look-up tables for evaluating constraint expressions. In practice, the patterns presented in Section 4 below are typically evaluated in between several seconds to a few minutes each.

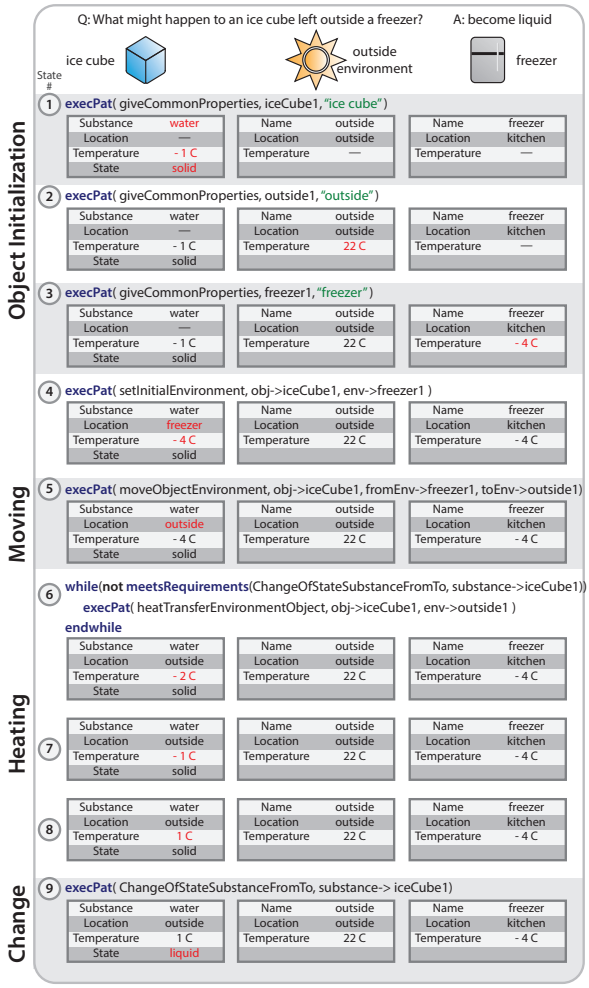

Figure 3: An example micro-model control script *(numbered states and instructions)* with three objects *(ice cube, outside environment, freezer)*, and the resulting state space changes *(red highlights)*. This control script simulates the result of leaving an ice cube outside of a freezer. The full script and CoSaTa interpreter output is provided in the distribution.

[2] Constraint satisfaction solvers are typically formulated to efficiently find a single solution that satisfies the constraints. In contrast, the solver presented in this work finds all possible solutions that can then be used for downstream processing.

```
inferencepattern ChangeOfStateWithSubstanceFromTo
  patterndescription = "Change the state of a substance from " + COS.<SOM1> + " to " + COS.<SOM2>

  // Requirements: This pattern acts on a single object (sub1), that is a kind of substance, and should have a state of matter.
  require instance sub1 = [KINDOF:"substance"]
  shouldhave (sub1."state of matter" != "")

  // Composite requirements: This pattern inherits rows from 3 other patterns
  infpat COS = changeStateOfMatter        // Change of state (fundamentals, requirements, etc.)
  infpat fromSOM = substanceInSOMWithPhaseTransitionPoint // A substance, and its melting/boiling/freezing point
  infpat toSOM = substanceInSOM           // A substance, in a particular state of matter
  rowequiv COS.som1 = fromSOM.subSOM.som  // To be valid, COS.som1 and fromSOM.subSOM.som should be the same fact.
  rowequiv COS.som2 = toSOM.som           // To be valid, COS.som2 and toSOM.som should be the same fact.

  instmap sub1 = COS.substance1         // The substance in this pattern refers to the same substance in the
  instmap sub1 = fromSOM.substance      // changeStateOfMatter (COS), substanceInSOMWithPhaseTransitionPoint (fromSOM),
  instmap sub1 = toSOM.substance        // and substanceInSom (toSOM) patterns.

  // Additional constraints
  // Combined patterns must be talking about the same (melting/boiling/freezing) point, and same material, to be valid.
  musthaveoromit(COS.<PhaseTransitionPoint> == fromSOM.<PhaseTransitionPoint>)
  musthaveoromit(fromSOM.<materialName> == toSOM.<materialName>)
  // The substance object (sub1) should be in state of matter <SOM1>, and have changed temperature above/below <pointTemp>.
  musthave(sub1."state of matter" == COS.<SOM1>)
  musthave(sub1."temperature" CHANGE [direction:COS.<tempDir> threshold:fromSOM.<pointTemp>])

  // Row definitions: All rows in this pattern are inherited from COS, fromSOM, and toSOM.

  // Code: Run the imperative code below if a given enumeration of this pattern is executed.
  // If the object (sub1) recently changed temperature above/below <pointTemp>, and is in state of matter <SOM1>
  if ((sub1."temperature" CHANGE [direction:COS.<tempDir> threshold:fromSOM.<pointTemp>]) && (sub1."state of matter" == COS.
      <SOM1>)) then
    // Set the object (sub1)'s new state of matter to be <SOM2>
    sub1."state of matter" = COS.<SOM2>
    // Add a human-readable explanation to the state space describing what this inference pattern did.
    addExplanationText("Substance (" + sub1."name" + ") made of (" + sub1."material" + ") is within the temperature range to
        change from a (" + COS.<SOM1> + ") to a (" + COS.<SOM2> + ").")
  endif

endinferencepattern
```

Inference Pattern: ChangeOfStateWithSubstanceFromTo

Inherited Pattern 1: COS:changeStateOfMatter

Row Name	Table Row
som1	a **<solid>** is a kind of "state of matter"
som2	a **<liquid>** is a kind of "state of matter"
cos	**<melting>** is a kind of "change of state"
somprop	"state of matter" is a property of a **<substance>**
point	a **<melting point>** is a kind of "phase transition point"
change	**<melting>** means the "state of matter" of **<substance>** changes from a **<solid>** into a **<liquid>** by **<increasing>** "heat energy"
thresh	**<melting>** occurs when the temperature of a **<substance>** is **<increased>** **<above>** the substance 's **<melting point>**
heatcool	**<heating>** means the "heat energy" of a substance is **<increased>**

Inherited Pattern 2: fromSOM:substanceInSOMWithPhaseTransitionPoint

Row Name	Table Row
somHasPoint	a **<solid>** has a **<melting point>**
point1	the **<melting point>** of **<water>** is **<0.0>** **<C>**

fromSOM.subSOM:substanceInSOM (nested)

Row Name	Table Row
som	a **<solid>** is a kind of "state of matter"
propSomTemp	the **<water>** is in the **<solid>** state, called **<ice>**, for temperatures below 0.0 C

Inherited Pattern 3: toSOM:substanceInSOM

Row Name	Table Row
som	a **<liquid>** is a kind of "state of matter"
propSomTemp	the **<water>** is in the **<liquid>** state, called **<water>**, for temperatures between 0.0 C and 100.0 C

Figure 4: *(Top)* An example composite pattern that (i) inherits row constraints from three other simpler patterns, and (ii) includes imperative code that effects the change described by the constraint satisfaction pattern. *(Bottom)* One example solution of this pattern, *melting ice into liquid water*. All other combinations of state changes *(e.g. freezing, boiling)* for all substances described in the knowledge base of semi-structured tables are also enumerated, but not shown here for space.

4 Example Solutions

Here the feasibility of generating constraint patterns (for downstream processing) or executable patterns (for modeling) is empirically demonstrated in the context of generating detailed multi-hop explanations to standardized elementary and middle school science exam questions drawn from the AI2 Aristo Reasoning Challenge (Clark et al., 2018).

4.1 Explanation Regeneration

The explanation regeneration task (Jansen and Ustalov, 2019) requires models to reconstruct large multi-fact gold explanations by selecting a set of interconnected facts from a knowledge base that match gold explanations provided in an explanation corpus. The task is very challenging, and current state-of-the-art models (e.g. Das et al., 2019) achieve nearly all of their performance by evaluat-

# Solutions Combined	Avg. Ceiling Accuracy	Avg. Extra Facts @ Ceiling
Automatically Converted Patterns Only (N=353)		
1	56.0%	7.8
2	67.5%	11.2
3	70.8%	12.6
Automatic and Manually Curated Patterns (N=385)		
1	58.3%	5.9
2	72.3%	7.7
3	78.0%	8.5

Table 2: Ceiling performance of the converted explanatory patterns from the WorldTree V2 corpus evaluated using CoSaTa on the explanation regeneration task.

ing facts independently rather than jointly.

The Worldtree V2 corpus (Xie et al., 2020) includes detailed multi-fact explanations for 4,400 standardized science exam questions grounded in a knowledge base of 63 tables and approximately 10k table rows, as well as a set of 353 semi-automatically authored collections of facts surrounding specific subtopics, such as *changes of state*, *inherited characteristics*, or *seasonal changes in daylight*. Here, those 353 inference patterns were converted to the CoSaTa constraint language using a prototype automatic converter, and all solutions to each pattern were enumerated with the CoSaTa solver. 42 of the automatically converted patterns were selected based on frequency of use for manual curation, where they were further abstracted, decomposed, and debugged. Ceiling performance on the explanation regeneration task was calculated for a shortlist of ranked solutions, in terms of both single solutions, and combinations of up to 3 solutions, with results shown in Table 2. Performance is evaluated in terms of accuracy (proportion of gold rows included in the explanation) and the average number of "extra" facts included in the solutions but not included in the gold explanation. The results show that the pattern solutions enumerated by CoSaTa have a ceiling performance of regenerating up to 58% of gold explanations when using a single solution, and up to 78% when combining up to three solutions. This empirically demonstrates the potential utility of using CoSaTa patterns as input to downstream inference models that are able to accurately select which patterns to combine to generate an explanation. The inference patterns in this experiment and their solutions are included as examples in the distribution.

4.2 Micro-models and Interpreter

Constructing micro-models from scratch requires (i) authoring a knowledge base of semi-structured tables, (ii) authoring patterns that reference those tables (such as those in Figures 2 and 4), and (iii) constructing micro-model control scripts (such as the example in Figure 3) that describe, in a short series of steps, how processes interact with objects and agents to reach a given outcome.

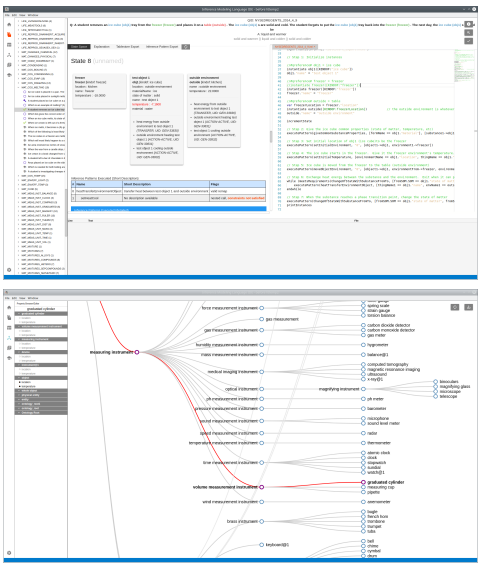

Figure 5: Screenshots of the prototype IDE. *(Top)* The debugger/editor for micro-model control scripts. *(Bottom)* The visual taxonomy editor, a component of the table editor.

To demonstrate this workflow, a series of 23 patterns including imperative code were authored for topics in *heat transfer* and *changes of state*, as well as a supporting semi-structured knowledge base containing several hundred facts across 21 tables including *taxonomic relations*, *locations of common objects*, *processes causing discrete changes*, and *physical properties of substances*. To support this effort, a prototype IDE called *Procession* (shown in Figure 5) was implemented using Electron that integrates a table-editor (including D3-based visual taxonomy editor), Monaco-based code editor, and side-to-side debugger/editor for micro-model control scripts that enables fast debug cycles. These example imperative patterns and the resulting interpreter output of the control scripts are included as examples in the distribution.

5 Conclusion

CoSaTa is an open-source constraint satisfaction solver for easily expressing and evaluating multi-fact compositional patterns in semi-structured tables of text, paired with an interpreted language that allows expressing micro-models. The tool, source, examples, and documentation are available at `http://www.github.com/clulab/cosata/` .

Acknowledgments

Thanks to Sebastian Thiem, who assisted in conducting the ceiling test of the WorldTree V2 patterns, and to Peter Clark for thoughtful discussions. The prototype IDE was developed in part under contract by Soft Design SRL. This work supported in part by the National Science Foundation (NSF Award #1815948, "Explainable Natural Language Inference", to PJ).

References

Peter Clark, Isaac Cowhey, Oren Etzioni, Tushar Khot, Ashish Sabharwal, Carissa Schoenick, and Oyvind Tafjord. 2018. Think you have solved question answering? try arc, the ai2 reasoning challenge. *arXiv preprint arXiv:1803.05457*.

Rajarshi Das, Ameya Godbole, Manzil Zaheer, Shehzaad Dhuliawala, and Andrew McCallum. 2019. Chains-of-reasoning at textgraphs 2019 shared task: Reasoning over chains of facts for explainable multi-hop inference. In *Proceedings of the Thirteenth Workshop on Graph-Based Methods for Natural Language Processing (TextGraphs-13)*, pages 101–117.

Martin Davis, George Logemann, and Donald Loveland. 1962. A machine program for theorem-proving. *Communications of the ACM*, 5(7):394–397.

Kenneth D Forbus. 2019. *Qualitative representations: How people reason and learn about the continuous world*.

Daniel Fried, Peter Jansen, Gustave Hahn-Powell, Mihai Surdeanu, and Peter Clark. 2015. Higher-order lexical semantic models for non-factoid answer reranking. *Transactions of the Association for Computational Linguistics*, 3:197–210.

Peter Jansen. 2018. Multi-hop inference for sentence-level textgraphs: How challenging is meaningfully combining information for science question answering? In *Proceedings of the Twelfth Workshop on Graph-Based Methods for Natural Language Processing (TextGraphs-12)*, pages 12–17.

Peter Jansen, Rebecca Sharp, Mihai Surdeanu, and Peter Clark. 2017. Framing QA as building and ranking intersentence answer justifications. *Computational Linguistics*, 43(2):407–449.

Peter Jansen and Dmitry Ustalov. 2019. Textgraphs 2019 shared task on multi-hop inference for explanation regeneration. In *Proceedings of the Thirteenth Workshop on Graph-Based Methods for Natural Language Processing (TextGraphs-13)*, pages 63–77.

Peter Jansen, Elizabeth Wainwright, Steven Marmorstein, and Clayton Morrison. 2018. WorldTree: A corpus of explanation graphs for elementary science questions supporting multi-hop inference. In *Proceedings of the Eleventh International Conference on Language Resources and Evaluation (LREC-2018)*, Miyazaki, Japan. European Languages Resources Association (ELRA).

Daniel Khashabi, Tushar Khot, Ashish Sabharwal, Peter Clark, Oren Etzioni, and Dan Roth. 2016. Question answering via integer programming over semi-structured knowledge. In *Proceedings of the Twenty-Fifth International Joint Conference on Artificial Intelligence*, pages 1145–1152.

Douglas B Lenat, Ramanathan V. Guha, Karen Pittman, Dexter Pratt, and Mary Shepherd. 1990. Cyc: toward programs with common sense. *Communications of the ACM*, 33(8):30–49.

Bill MacCartney and Christopher D Manning. 2007. Natural logic for textual inference. In *Proceedings of the ACL-PASCAL Workshop on Textual Entailment and Paraphrasing*, pages 193–200.

Christopher D Manning, Mihai Surdeanu, John Bauer, Jenny Rose Finkel, Steven Bethard, and David McClosky. 2014. The stanford corenlp natural language processing toolkit. In *Proceedings of 52nd annual meeting of the association for computational linguistics: system demonstrations*, pages 55–60.

Huan Sun, Hao Ma, Xiaodong He, Wen-tau Yih, Yu Su, and Xifeng Yan. 2016. Table cell search for question answering. In *Proceedings of the 25th International Conference on World Wide Web*, pages 771–782.

Ming Tu, Kevin Huang, and Guangtao Wang. 2020. Select, answer and explain: Interpretable multi-hop reading comprehension over multiple documents. In *Proceedings of the Thirty-Fourth AAAI Conference on Artificial Intelligence*.

Marco Valentino, Mokanarangan Thayaparan, and André Freitas. 2020. Unification-based reconstruction of explanations for science questions. *arXiv preprint arXiv:2004.00061*.

Zhengnan Xie, Sebastian Thiem, Jaycie Martin, Elizabeth Wainwright, Steven Marmorstein, and Peter Jansen. 2020. Worldtree v2: A corpus of science-domain structured explanations and inference patterns supporting multi-hop inference. In *Proceedings of The 12th Language Resources and Evaluation Conference*, pages 5456–5473.

Zhilin Yang, Peng Qi, Saizheng Zhang, Yoshua Bengio, William Cohen, Ruslan Salakhutdinov, and Christopher D Manning. 2018. Hotpotqa: A dataset for diverse, explainable multi-hop question answering. In *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing*, pages 2369–2380.