A specification-aware modeling of mental model theory for syllogistic reasoning

Yutaro Sugimoto¹ and Yuri Sato²

¹Department of Philosophy, Keio University ²Interfaculty Initiative in Information Studies, The University of Tokyo

sugimoto@abelard.flet.keio.ac.jp, sato@iii.u-tokyo.ac.jp

Abstract

Computational cognitive models can embody the structures and processes proposed in a cognitive theory. However, they do not necessarily reveal the theory’s underlying assumptions and specifications. This study aims to bridge the gap between cognitive theory and its computational implementation, focusing on a case of mental model theory on human reasoning. Using a mathematics-based and statically-typed programming language (Haskell), we provide a specification-aware computational implementation of syllogistic reasoning with mental models.

Keywords: Mental model theory, Type system, Specification, Cognitive modeling, Logical reasoning.

Introduction

The specification problem in cognitive modeling

In cognitive science research, mental representations and processes underlying human cognition have been treated in analogy with computer programs consisting of data structures and algorithms. However, it is not always easy to understand the relationship between the two in cognitive theories employing natural language for description. This occasionally leads to ambiguous formulations of cognitive theories. To remove these ambiguities, researchers have developed computer implementations of these theories. This type of modeling has revealed more detailed structures and processes than the existing formulations using natural language.

Recently, some researchers have questioned whether or not cognitive models should actually reveal theory specifications. McClelland (2009) pointed out that computational cognitive modelings are not full accounts of cognitive theories; implementations are not intended to embody the aspects of a theory’s assumptions or specifications that are loosely defined in a cognitive theory. However, Cooper and Guest (2014) argue that McClelland (2009) downplays the role of modeling in theory specification, and in particular the possibility that a model might embody a set of assumptions (p. 43). To remedy this, they suggest that it is necessary to improve the current situation whereby modeling approaches are not suitable for theory specification, by bridging the gap between computational implementation and cognitive theory. The present study was conceived in a similar vein. We introduce a modeling method consisting of full-fledged descriptions of theory specifications; “type system based modeling.”

A case of mental model reasoning

Cognitive studies involving the mental model theory, which was introduced by Johnson-Laird (1983; 1984), are appropriate examples with which to discuss the specification problem in cognitive modeling. The mental model theory is a cognitive theory of sentence interpretation and inference. The theory was first formulated for categorical syllogisms, but has now been extended to various domains of reasoning (for a review, see Khemlani & Johnson-Laird, 2013). Syllogism with quantificational sentences is one of the most elementary forms of natural language inference, and is important in the analysis of human reasoning (cf. Moss, 2008). The present study considers the domain of syllogisms only, focusing on the recent version (Bucciarelli & Johnson-Laird, 1999) and its corresponding computer implementation (Mental Models & Reasoning Lab, abbreviated as MMRLab.¹).

Natural language description  Syllogistic reasoning processes in mental model theory as described with natural language are briefly illustrated below. The basic idea underlying the theory is that people interpret sentences by constructing mental models corresponding to possibilities and make inferences by constructing counter-models. (1) Mental models consist of a finite number of tokens, denoting the properties of individuals. All A are B has a model illustrated on the leftmost side of Fig.1, where each row represents an individual. A row consisting of two tokens, a and b, refers to an individual that is A and B. The tokens with square brackets, [a], express that the set containing them is exhaustively represented by these tokens and that no new tokens can be added to it. A sequence of tokens without square brackets can be extended with new tokens so that an alternative model is constructed. (2) Some C are not B has a model illustrated on the second from the left side of Fig.1. A row with a single token, b, refers to an individual that is B but not C. A row consisting of two tokens, c and –b, refers to an individual that is C but not B, by using “–” to denote negation. (3) These two models for premises are integrated into a single model, as shown in the second from the right in Fig.1. Two tentative conclusions, Some A are not C and Some C are not A, are extracted. (4) To search counterexamples for them, an alternative model is constructed by adding new tokens (token c), as shown in the rightmost side of Fig.1. Since each token of A is corresponding to tokens of C, Some

Fig 1: Syllogistic reasoning with mental models

<table>
<thead>
<tr>
<th>1st premise model</th>
<th>2nd premise model</th>
<th>Integrated model</th>
<th>Alternative model</th>
</tr>
</thead>
<tbody>
<tr>
<td>[a] b</td>
<td>[a] c</td>
<td>[a] b c</td>
<td>[a] b c</td>
</tr>
<tr>
<td>[a] b</td>
<td>[a] c</td>
<td>[a] b c</td>
<td>[a] b c</td>
</tr>
</tbody>
</table>

¹As described on p.14 of Khemlani and Johnson-Laird (2013), the major part of the program code for syllogistic fragments (MMRLab) still lives on in another implementation of the mental model theory, what is called mReasoner, which can cope with various domains beyond categorical syllogisms.
A are not C is refuted. Instead, Some C are not A survives.

**Modeling implementations** Following the theory informally described above, Johnson-Laird and his colleagues have developed several computational cognitive models, some of which have been published (p. 443 of Khemlani & Johnson-Laird, 2012). Their implementation of syllogisms is found in MMRLab. It is written in Common Lisp and a mental model is represented as a *list*, which is the most basic data structure in Lisp. For example, the 1st and 2nd premises models in Fig. 1 are represented as lists:

```lisp
('{{* A} (B)) {{* A} (B))

{{(C) (~ B)} (C) (~ B) (B) (B))}
```

Their implementation of mental model theory was successful in embodying structures and processes proposed in the theory but it was unable to fully account for its specification. More concretely, it lacked a mathematical formalization of the concept of a “mental model.” This ambiguity has resulted in various misunderstandings of the theory, and most of the controversy around the theory in cognitive science and logic has focused on this point (Braine, 1994; Hintikka, 1987; Stenning & Van Lambalgen, 2008). It is therefore important to understand the nature of the theory by addressing the specification problem.

**Beyond the specification problem**

**Formal modeling** A possible method to bridge the gap between implementation and theory is to convert natural language description theories to formal systems. This method is called *formal cognitive modeling:* structures and processes described in a theory are decomposed at a more abstract (mathematical) level than existing implementations (e.g., Arkoudas & Bringsjord, 2008; Bosse, Jonker, & Treur, 2006). Koralus and Mascarenhas’s (2013) modeling is an example of this approach. They constructed a formal (propositional) inference system, called “the eretotic theory of reasoning”, which essentially corresponds to mental model reasoning. (i) They started by converting from mental models to algebraic structures such as \( p \cup q \), consisting of representational units \( p \) and \( q \) (standing for propositions) and operations \( \cup \) standing for conjunction), which can be translated to logical formulas such as \( p \wedge q \). Here non-standard semantics, not model-theoretic semantics, were given for the interpretations of sentences. (ii) Update (eretotic) rules for adding the new premise, which is treated as an answer to the question in discourse, and operation rules for making inference as refutation were given. (iii) In this system, soundness and completeness for classical propositional semantics were shown via translation.

To our knowledge, Koralus & Mascarenhas’s work is the first major formalization of mental model reasoning. This approach has at least an advantage in analyzing computational complexities of solving tasks rather than predicting human performance data of them (for more details see e.g. Verbrugge, 2009; Isaac, Szymanik, & Verbrugge, 2014). However, this modeling is abstract in that it can be implementation-independent. Therefore it may not address problems caused by a particular representation in a cognitive theory. Thus, certain specifications, such as the mental models definition may not be provided at an appropriate level.

**Specification-aware modeling** An alternative way to bridge the gap between implementation and theory is to use a programming language suitable for specification descriptions, which we call *specification-aware cognitive modeling*. This approach was manifested in the seminal study of Cooper et al. (1996). According to them, (i) ordinary (Lisp and C) implementations themselves do not include the specifications of cognitive theories; (ii) formal theories of specifications, including mathematical descriptions, are not appropriate for bridging the gap between implementation and theory since they are implementation-independent. In our view, Johnson-Laird’s mental model implementation in Common Lisp corresponds to case (i) and the formal modeling approaches of Koralus and Mascarenhas (2013) correspond to case (ii). Alternatively, Cooper et al. (1996) have provided some cognitive modeling implementations written by an executable specification language *Sceptic*, consisting of the declarative (logic programming) language Prolog, with the additional device of control structures. Their implementations include the mental model theory for syllogistic reasoning (for more details, see Cooper, 1992). Here, rewrite rules obey certain mathematical manipulations, such as rewriting from one term to another, as is the case in lambda calculus. This rewriting can be regarded as a specification in itself. Furthermore, an advantage of declarative language is that one can check the logical relationships between input and output representations without specifying a strategy of computation. (cf. sec.3 of Cooper & Guest, 2014).

This study put forward the line of specification-aware modeling. We try to give a “computational” (i.e. executable) semantics for the mental model theory. For the purpose of this research, we consider a reconstruction based on type system (cf. Mitchell, 2003). We refer to this approach as *type system based modeling*. Importantly, we chose a purely-functional programming language *Haskell* that has strong static typing and lazy evaluation strategy (Jones, 2003; Marlow, 2010). In contrast to the Common Lisp language used in Johnson-Laird and his colleagues’ implementation (MMRLab), Haskell is a purely functional programming language with strong static typing. Explicit type annotations are useful for defining mental models within the system, and Haskell encourages to pro-

---

3Recently, Clark (submitted) straightforwardly provided a specification of mental model theory using total and partial truth functions, which correspond to exhaustive and non-exhaustive models.
gram in a declarative style. This property can contribute to reasoning about the program itself (e.g. to give a denotational semantics for it Haftmann, 2010; Vytinioti et al., 2013). It is important not only to guarantee the fact that a program is executable (there is no syntactical error), but also to verify whether a program works as intended by theorists (the aspect of semantics.) Note that this brief paper focuses on a part of the long-term study. Just a relatively weak formal verification (static type-checking based on explicit typing) is available here. To provide a formal verification by a theorem prover or other semantic verifications leave for future work.

**An Implementation of the Mental Model Reasoning System**

An outline of the reasoning system is given in Sugimoto, Sato, and Nakayama (2013). The system is portrayed as a conceptual diagram Fig. 2 that shows translations from one model to another by the following steps: 1. constructing mental models of premises, 2. integrating these premise models into an initial (integrated) model, 3. drawing a tentative conclusion from the initial model, 4. constructing alternative models by falsification, 5. producing a final conclusion.

Steps 1 and 5 involve IO actions. Steps 2, 3 and 4 consist of stateless (no side effects) functions only.

**Mental model construction**

The syllogistic language (input) is considered as a kind of controlled natural language, and its grammar can be defined by BNF+. We define a combinator parser to parse this language instead of using a standard bottom-up parser (MMRLab.) An implementation of the language and the parser is given in Code 1:

```
 data MToken = AToken Atom | BToken Atom | CToken Atom

 type Parser a b = [a] -> [(b,[a])]

 pCop :: Parser String String
 pNeg :: Parser String String
 pNegQuant :: Parser String String

 data Atom = ASymbol | BSymbol | CSymbol

 data MToken = AToken Atom | BToken Atom | CToken Atom

 data MModel = [Indiv]

 type BSymbol = Symbol
 type ASymbol = Symbol
 type Nil = Symbol
 type Neg = Symbol
 type Exh = Symbol
 type Symbol = Char

 data Indiv = [Token]

 type MModel = [Indiv]

 data Token = [Symbol]

 data Atom = ASymbol | BSymbol | CSymbol

 first :: Token -> Atom
 first atom = case atom of
     Atom asymbol -> ASymbol
     Atom bsymbol -> BSymbol
     Atom csymbol -> CSymbol

 token :: Int -> Token -> Atom
 token _ atom = case atom of
     Token _ -> first atom

 satisfy :: (a -> Bool) -> Parser a a
 satisfy p (x:xs) | p x = [(x,xs)]
 satisfy p [] = []

 succeed :: b -> Parser a b
 succeed _ = []

 fail :: Parser a b
 fail _ = []

 succeed r xs = [(r,xs)]

 token :: Int -> Atom
 token _ atom = case atom of
     Token _ -> first atom

 negate :: Symbol -> Symbol
 negate neg = '-'

 asymbol :: Symbol
 asymbol = 'a'

 bsymbol :: Symbol
 bsymbol = 'b'

 csymbol :: Symbol
 csymbol = 'c'

 neg :: Symbol
 neg = '-'
```

**Code 1**: The definition for syllogistic language and its parser

```
data MToken = AToken Atom | BToken Atom | CToken Atom

 type Parser a b = [a] -> [(b,[a])]

 pCop :: Parser String String
 pNeg :: Parser String String
 pNegQuant :: Parser String String

 type ASymbol = Symbol
 type BSsymbol = Symbol
 type CSsymbol = Symbol

 negate :: Symbol -> Symbol
 negate neg = '-'

 asymbol :: Symbol
 asymbol = 'a'

 bsymbol :: Symbol
 bsymbol = 'b'

 csymbol :: Symbol
 csymbol = 'c'

 neg :: Symbol
 neg = '-'
```

**Code 2**: The definition for mental model components

The mental models for syllogistic reasoning are defined in Code 2. A mental model is a class of models s.t. \( m \times n \) matrix (multi-list) of tokens. A row or an individual of a mental model is a finite sequence of tokens (model) where each atom occurs at most once. A column or a (property) of a mental model is a finite sequence of tokens where tokens containing different atoms cannot occur. If square bracketed tokens occur in a column, only negative tokens can be added.

The translation from the syllogistic language to mental model representations is performed by monadic parsing (a recursive descent parsing technique well known in functional programming community. See, e.g. Van Eijck and Unger 2010). The parser for syllogistic language constructs abstract syntax trees and then converts mental model representations by the following compositional semantics (Fig. 2). As an example, let \( X, Y \) denote terms \( A, B, C \), the four types of syllogistic sentences can be translated to mental models as follows:

\[
\begin{align*}
\text{All } X \text{ are } Y & \Rightarrow \forall x (x \in X \Rightarrow x \in Y) \\
\text{Some } X \text{ are } Y & \Rightarrow \exists x (x \in X \land x \in Y) \\
\text{No } X \text{ are } Y & \Rightarrow \forall x (x \in X \Rightarrow \neg x \in Y) \\
\text{Some } X \text{ are not } Y & \Rightarrow \exists x (x \in X \land \neg x \in Y)
\end{align*}
\]

4In Bucciarelli and Johnson-Laird (1999) “exhaustive model” is represented in square brackets, here we use the “*” symbol as used elsewhere (MMRLab.)

5For a treatment of a mental model as a class of models, see Barwise (1993).
Reordering and Switching Since syllogisms have several "figures" according to the order of their premises and term arrangements, the actual integration should occur after the preprocessing of reordering terms and switching premises. This preprocessing consists of the following four patterns: (1) If the term order of $P_1$ is AB and $P_2$ is BC, nothing happens. (2) If the term order of $P_1$ is BA and $P_2$ is CB, integration starts with $P_2$. (3) If the term order of $P_1$ is AB and $P_2$ is CB, the second model is swapped round and added. (4) If the term order of $P_1$ is BA and $P_2$ is BC, the first model is swapped round and the second model added.

Finding a middle atom The function find_middle_atom has a type signature find_middle_atom :: MModel -> MModel -> Symbol. The implementation of this is similar to a set intersection operation for the affirmative tokens (tokens that do not contain negatives.) An example is when two premises are presented as in Fig.1, \{a, a, b, b\} \cap \{c, c, b, b\} = b. The following is an implementation:

```haskell
find_middle_atom :: MModel -> MModel -> Symbol
| null mod1 = []
| not (null mod1) = mid
| otherwise = find_middle_atom (tail mod1) mod2
where mid = find_middle_from_ind (head mod1) mod2
```

Match The function for matching premises $P_1$, $P_2$, and middle atom $a$ has the type signature match :: MModel -> MModel -> Symbol -> MModel. This function recursively calls join_ to join the premises to an integrated model.

```haskell
match :: Symbol -> MModel -> MModel -> MModel
match mid_atom mod1 mod2
| null mod1 = [mid]
| not (null mod1) = mid
| otherwise = remove_lis (find_poslis_in_mod mid_atom mod1) mod2
where mid = find_middle_from_ind (head mod1) mod2
```

Join The recursive function join_ takes a mid atom and two individuals, and joins two individuals together setting the new mid to exhausted if either the first individual or second individual were exhausted. The type signature of join_ is:

```haskell
join_ :: Symbol -> Indiv -> Indiv -> Indiv
join_ :: Symbol -> Indiv -> Indiv -> Indiv
join_mid_atom mid1 mid2
| exhausted = if exhausted = true
| otherwise = remove_lis (find_poslis_in_ind mid_atom mid1 mod1) mod2
else remove_lis (find_poslis_in_ind mid_atom mid1 mod2) mod2
```

Integrating Premises into Initial Model

We give a description of the integration process of premises into an initial model via mid term tokens (Fig. 3-a.) The integration function combine takes two models (premises) $P_1$, $P_2$ and returns an integrated model $M$ with the help of find_a_middle_atom. The type signature for this function is integration :: MModel -> MModel -> MModel. This is implemented as follows:

```haskell
combine :: MModel -> MModel -> MModel
combine mid1 mid2 =
  where mid_atom = find_middle_atom mid1 mid2
```

Figure 3: Process diagrams and their type information: (a) Integration, (b) Drawing a conclusion, (c) Falsification

Drawing a Conclusion from a Model

Drawing a conclusion (Fig.3-b) is a function that takes an integrated (initial) model and dispatches whether it contains negative token or not. Based on the predicates (all_isa, some_isa, no_isa, and sm_not_isa) it then dispatches further and returns corresponding answers (action). conclude has type signature: conclude :: Symbol -> Symbol -> Symbol -> MModel -> [Symbol]. If the predicates return False, then it returns "no valid conclusion". The below is an implementation:

```
conclude :: Symbol -> Symbol -> Symbol
conclude :: Symbol -> Symbol -> Symbol
conclude mod1 mod2
| null mod1 = [mod2]
| not (null mod1) = mod1
| otherwise = find_middle_atom (head mod1) mod2
where mod1 = find_middle_from_ind (head mod1) mod2
```

Note: since possible conclusions have term order: Subj-Obj and Obj-Subj, conclude is executed twice. For simplicity, we omit the second execution of conclude.
The following are sub functions called by conclude:

- **all_isa** takes a model $M$ that has the end terms X, Y and returns True iff all subjects are objects in a model. It then concludes returns the answer “All X are Y.” This has a type signature: all_isa :: Symbol -> Symbol -> MModel -> Bool. For example, for a model $M$:[a] b c, it is given where the end terms are A and C, it returns the answer “Some A are C.”

- **sm_isa** takes a model $M$ that has end terms X, Y and returns True iff at least one individual in the model contains positive occurrences of both subject and object atoms. Then conclude returns the answer “Some X are Y”. This has the type signature: sm_isa :: Symbol -> Symbol -> MModel -> Bool. For example, for a model $M$:[a] [b] c, then sm_isa ([a] [b] c) is given where the end terms are A and C, it returns the answer “Some A are C”.

- **no_isa** takes a model $M$ that has the end terms X, Y and returns True iff no subject object end is term object in any individuals in the model. conclude returns “No X are Y”. This has a functional type: no_isa :: M -> A. For example, if a model $M$:[a] [b] [c], it then returns the answer “No A are C”.

- **sm_not_isa** takes a model $M$ that has the end terms X, Y and returns True iff at least one subject occurs in individuals without an object, then conclude returns “Some X are not Y”. This has a functional type: sm_not_isa :: M -> A. For example, if a model $M$:[a] [b] [c], it is given where the end terms are A and C, it returns the answer “Some A are not C”.

**Falsification**

Once the mental model theory constructs an initial model and draws a tentative conclusion, the theory, according to its rules, tries to construct an alternative model to refute the conclusion (the default assumption). The falsification function (Fig. 3-c) takes a model and dispatches whether or not it contains a negative token. Then, based on the predicates (breaks, add_percentive, moves, and add_negative), it tries to modify the model. If successful, it returns an alternative model and calls conclude recursively. If it fails, this function terminates and conclude outputs a final conclusion. The below is an implementation of falsify:

- **breaks** finds an individual containing two end terms with non-exhaustive mid terms, divides it into two, and then either returns new (broken) model or returns nil. Its type signature is: breaks :: MModel -> MModel. For example: when a model $M$ is a b c, then breaks $M \sim \{a\} b c$. This is implemented as follows:

- **add_affirmative** has the type signature: add_affirmative :: MModel -> MModel. If add_affirmative succeeds, then it returns a new model with added item (added model), else it returns nil if the conclusion is not A-type (“All X are Y”) or if there is no addable subject item. For example: if a given model $M$ is [a] [b] c, then add_affirmative $M \sim [a] [b] [c]$.

- **moves** has the type signature: moves :: MModel -> MModel. If there are exhausted end items not connected to other end items or their negatives (i.e E-type (“No X are Y”) conclusions), and if the other end items are exhausted, or O-type (“Some X are not Y”) conclusions, then it joins them. Otherwise it joins one of each and returns nil if the first end item
cannot be moved, regardless of whether or not a second one can be. E.g., if a given model $M$ is

$$
\begin{align*}
[a] & \not\in b \\
[b] & \in c \\
[c] & \not\in c
\end{align*}
$$

then $\text{moves } M \leadsto
\begin{align*}
[a] & \not\in b \\
[b] & \not\in c \\
[c] & \not\in c
\end{align*}$.

When this function is called by $\text{falsify}$, $\text{neg_breaking}$ (similar procedure to $\text{breaks}$) is also called as an argument.

$$
\text{add_negative :: } [\text{Symbol}] \rightarrow \text{MModel} \rightarrow \text{MModel}
\begin{align*}
\text{add_negative } \text{concl model} &= \\
\text{if falseif model } \text{newmod} \\
\text{then } \text{newmod} \\
\text{else } [\text{Symbol}] \\
\text{where } \text{newmod} = \text{moving concl model}
\end{align*}
$$

Concluding Remarks

We have proposed type system based modeling as a novel approach in (specification-aware) cognitive modeling. To show the advantages of this approach, we have implemented the sylogistic reasoning system with mental models in Haskell (a popular pure functional statically typed programming language). Compared with other approaches, our implementation includes some aspects of theory specification such as mental model definitions and type information for each process. Our type system based modeling sheds light on the ambiguity problem of mental model theory, which has been discussed at the crossroad between cognitive science and logic. In the modeling work done so far, we have analyzed mental model reasoning for sylogistic fragments. However, our work is not limited to this. There is plenty of scope for further work in several kinds of mental model reasoning (Khemlani & Johnson-Laird, 2013; Johnson-Laird, Byrne, & Tabossi, 1989; Johnson-Laird & Byrne, 1989); more generally, various cognitive activities involving propositions taking some semantic values.

Acknowledgements

This study was supported by MEXT-Supported Program for the Strategic Research Foundation at Private Universities and Grant-in-Aid for JSPS Fellows (252291). The authors are grateful to Sangeet Khemlani for helpful comment and Micah Clark for opportunity to read his paper under submission.

References


