5-2019

# Processing Narratives by Means of Action Languages

Craig Olson
*University of Nebraska at Omaha*

# PROCESSING NARRATIVES BY MEANS OF ACTION LANGUAGES

A Thesis

Presented to the

Department of Computer Science

and the

Faculty of the Graduate College

University of Nebraska at Omaha

In Partial Fulfillment
of the Requirements for the Degree

Master of Science in Computer Science

University of Nebraska at Omaha

by

Craig Olson

May 2019

Thesis Committee:

Dr. Yuliya Lierler, Chair

Dr. Parvathi Chundi

Dr. Ryan Schuetzler

# PROCESSING NARRATIVES BY MEANS OF ACTION LANGUAGES

Craig Olson, MA
University of Nebraska at Omaha, 2019
Advisor: Dr. Yuliya Lierler

In this work we design a narrative understanding system Text2ALM that can be used in Question Answering domains. System Text2ALM utilizes an action language $\mathcal{ALM}$ to perform inferences on complex interactions of events described in narratives. The methodology that Text2ALM follows in its implementation was originally outlined by Yuliya Lierler, Daniela Inclezan, and Michael Gelfond in 2017 via a manual process, and this work serves as a proof of concept in a large-scale environment. Our system automates the conversion of a narrative to an $\mathcal{ALM}$ model containing facts about the narrative. We make use of the VerbNet lexicon that we annotated with interpretable semantics in $\mathcal{ALM}$. Text2ALM also utilizes Text2DRS system developed by Gang Ling at UNO in 2018. These resources are used to produce an $\mathcal{ALM}$ program with a system description containing information on the narrative's entities, events, and their relations, as well as a history of the narrative's events. The $\mathcal{ALM}$ logic is used in tandem with a basic commonsense library of $\mathcal{ALM}$ modules to generate a formal structure capturing the narrative's properties. The CALM system designed by researchers at Texas Tech in 2018 and is used by Text2ALM to process the $\mathcal{ALM}$ program. The effectiveness of this approach is measured by the system's ability to correctly answer questions from the QA bAbI tasks published by Facebook Research in 2015. The Text2ALM system matched or exceeded the performance of state-of-the-art machine learning methods in six of the seven tested tasks.

# DEDICATION

*To my family, friends, and mentors*

# TABLE OF CONTENTS

# TABLE OF EXAMPLES

# TABLE OF FIGURES

# TABLE OF TABLES

# 1 INTRODUCTION

*Question Answering* (QA) systems have been growing in popularity. QA systems are used to answer a variety of questions from information accessible to the system. QA systems are popular research experiments because top-notch systems must successfully combine the fields of natural language processing, knowledge representation and reasoning, information retrieval, and machine learning. Contributions in these fields can be measured by testing and implementing research proposals in a QA domain. Advancements in QA systems are in high demand due to the growing commercial demands in services like Amazon's Alexa, the Google Assistant, and Apple's Siri, just to name a few.

The domain and scope of QA systems can vary depending on the requirements. This thesis focuses on QA systems in the domain of narrative texts. A *narrative text* is a sequence of sentences retelling a story in past tense. Prior research has been published in the field of QA systems for narrative text, some of which can be seen in (Kočiský et al., 2017; Labutov, Yang, Prakash, & Azaria, 2018; Lierler, Inclezan, & Gelfond, 2017). Our research focuses on using the actions occurring within the narrative to provide QA capabilities.

The sentences in a narrative text contain action verbs occurring in chronological order. *Action verbs* are verbs that express either physical or mental acts performed by a sentence's subject or clause. In the scope of this thesis, we focus our attention on physical action verbs. Examples of physical actions verbs are *to go*, *to give*, and *to put*. All these

verbs describes specific changes to the physical environment, such as changes to the subject's location or possessions.

Sentences in an action-based narrative describe a series of events with conjoining event-specific information. After each event, the narrative's reader has a picture of the action that occurred and which properties related to the state of the narrative were changed. A reader can easily answer questions related to the effects of each action in a narrative, however it is often a challenging task for an artificial intelligence (AI) agent to answer the same questions.

For instance, consider the simple example narrative in Example 1 that will be referred to as the *JS Discourse.*

```
John travelled to the hallway.                    (1)
Sandra journeyed to the hallway.                  (2)
```

*Example 1: JS Discourse*

Now envision these simple questions related to the narrative:

```
Is John inside the hallway at the end of the story?          (3)
Is Sandra inside the hallway at the end of the story?        (4)
Is the hallway empty at the end of the story?                (5)
```

*Example 2: JS Discourse Example Questions*

Humans answer these questions due to their ability to combine the information in the narrative with commonsense knowledge. The actions in the narrative describe changes to the environment and can be coupled with the reader's commonsense knowledge to change the reader's mental picture of the narrative's state. For example, after reading sentence (1), a human knows that *John* is the subject of the sentence and *travelled* is an action verb that describes an action performed by *John*. A human also knows that *travelled* describes the act of motion, and specifically that *John's* location

changes from an arbitrary initial location to a new destination, the *hallway*. Likewise, a human reader could reason similarly about sentence (2). After sentence (2), a human reader understands that *Sandra's* location is also the *hallway* and can therefore answer question (4) accurately.

The ability to combine facts from the narrative with a human's commonsense knowledge on actions and their effects convert narrative-based Question Answering from being a simple human task to a difficult task for modern AI agents. The field of *Knowledge Representation and Reasoning* (KRR) is a subfield of artificial intelligence dedicated to representing information about the world in a format usable by an AI agent to reason about and solve complex tasks. In this work we test methods of knowledge representation and reasoning in QA systems.

Lierler, Inclezan, and Gelfond (2017) outline a methodology for designing QA systems to make inferences based on complex interactions of events in narratives. Their process utilizes an action language $\mathcal{ALM}$ (Inclezan & Gelfond, 2015) and an extension of the VerbNet lexicon (Palmer, 2018; Schuler, 2005). $\mathcal{ALM}$ enables a system to structure knowledge regarding complex interactions of events and implicit background knowledge in a straight-forward and modularized manner. The represented knowledge is then used to derive inferences about the text. The proposed methodology also assumes the extension of the VerbNet lexicon with interpretable semantic annotations in $\mathcal{ALM}$. The VerbNet lexicon provides semantic information by mapping narrative events to VerbNet classes. The methodology also specifies the use of several other Natural Language Processing (NLP) resources to produce $\mathcal{ALM}$ system descriptions for input discourses, such as LTH (Johansson & Nugues, 2007), Stanford CoreNLP (Manning et

al., 2014), PropBank (Palmer, Gildea, & Kingsbury, 2005), and SemLink (Bonial, Stowe, & Palmer, 2013). The process attempts to align with cognitive processes humans perform when reading a narrative and answering questions on it. The described methodology is exemplified with two sample narratives that were completed manually. The authors translated those narratives to a $\mathcal{ALM}$ programs by hand and wrote the supporting $\mathcal{ALM}$ modules to capture knowledge as needed.

The goal of the current work is to create an automated system based on the ideas of Lierler et al. (2017). The automated system developed in this thesis is titled Text2ALM. It serves as a proof of concept to test the feasibility, effectiveness, challenges, and limitations on this approach for automated large-scale narrative-based QA systems. The system performs commonsense reasoning task by expressing implicit knowledge with action languages.

The structure of this thesis report is the following. Section 2 provides a background, where we introduce the key concepts and resources for the Text2ALM system. This section contains relevant definitions and examples for each resource. Section 3 explains Text2ALM system by diving deeper into its four main tasks:

1. Entity and Relation Extraction
2. Creation of $\mathcal{ALM}$ Program
3. $\mathcal{ALM}$ Model Generation and Interpretation
4. Question Answering Reasoning

Section 4 provides the details on the system's evaluation and related work. Section 5, Section 6, and Section 7 contain suggestions for future work, conclusions, and acknowledgements, respectively.

# 2 BACKGROUND

We rely on several prior research endeavors by the KR and NLP communities to accomplish the development of the Text2ALM system. The resources utilized are:

1. $\mathcal{ALM}$: An Action Language (Inclezan & Gelfond, 2015)
2. CALM: An $\mathcal{ALM}$ Solver (Wertz, Chandrasekan, & Zhang, 2018)
3. CoreALMLib: An $\mathcal{ALM}$ Knowledge Base (Inclezan, 2016)
4. VerbNet: An English Verb Lexicon (Palmer, 2018; Schuler, 2005)
5. Text2DRS: An Entity and Relation Framework System (Ling, 2018)
6. bAbI Project: QA Benchmarking Tests (Weston et al., 2015)

We now introduce the resources in more detail.

## 2.1 LANGUAGE $\mathcal{ALM}$

A key concept related to this work is the action language $\mathcal{ALM}$ (Inclezan & Gelfond, 2015). *Action languages* are formal languages used to capture knowledge using the effects of actions. Action languages provide convenient syntactic constructs to represent knowledge about dynamic domains. The knowledge can then be compiled into a *transition diagram* where nodes are possible system states and edges are actions. The effects of an action are represented by the differences between the two system states connected by the arc. The resulting transition diagram represents both direct and indirect effects of actions.

For instance, consider a sample expression "*a causes p*" in a toy action language where *a* is an action and *p* is a property. Intuitively, the expression means that the execution of action *a* result in property *p* being true. Assuming *p* is the only property in the domain, the transition diagram given in Figure 1 depicts possible behaviors in the domain. The execution of action *a* at the state $s_1$, namely, when property p does not hold,

causes the transition to the state *p*. The execution of action *a* at the state $s_2$ doesn't change property *p*.

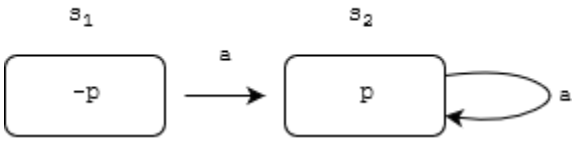


*Figure 1: Example Transition Diagram*

The *JS Discourse* is a dynamic domain where *John* and *Sandra's* location populate the transition diagram and the events in the narrative are possible actions. The transition diagram capturing the possible states of the *JS Discourse* is given in Figure 2. State $s_1$ in Figure 2 designates the state where the location of *John* and *Sandra* is the *hallway*. Likewise, state $s_2$ characterizes the state where *John's* location is the *hallway*, but *Sandra's* location is not the *hallway*.

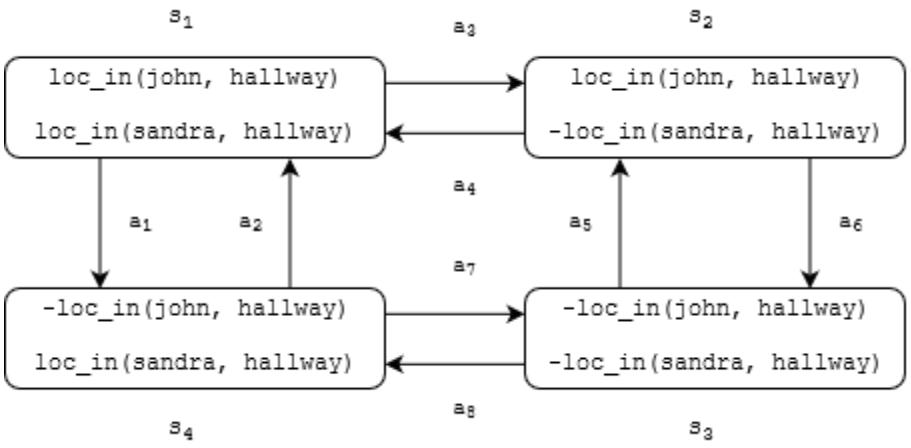


*Figure 2: JS Discourse Transition Diagram*

Scenarios of a dynamic domain correspond to *trajectories* in the domain's transition diagram. Trajectories are sequences of alternating states and actions. A trajectory captures the sequence of events, starting with the initial state associated with time step 0. Each arc is associated with the time step incrementing by 1. A sample

trajectory for the transition diagram in Figure 2 is $<s_1, a_3, s_2, a_6, s_3>$. This trajectory captures the following scenario:

- *John* and *Sandra* are in the *hallway* at the initial time step 0.
- *Sandra* leaves the *hallway* at time step 0. This is denoted by arc $a_3$.
- The result is that *Sandra's* location is not the *hallway* at time step 1.
- *John* leaves the *hallway* at time step 1. This is represented by arc $a_6$.
- The result is that *John's* location is also not the *hallway* at time step 2.

Language $\mathcal{ALM}$ is a sophisticated action language with an ability to capture the commonalities of similar actions (Inclezan & Gelfond, 2015). This section illustrates the syntax and semantics of $\mathcal{ALM}$ using the sample *JS Discourse* dynamic domain. We first define an $\mathcal{ALM}$ system description in Section 2.1.1. We then present an $\mathcal{ALM}$ history in Section 2.1.2. We conclude our $\mathcal{ALM}$ overview with a definition of an $\mathcal{ALM}$ model in Section 2.1.3.

### 2.1.1 Language $\mathcal{ALM}$ System Description

In language $\mathcal{ALM}$, we describe a dynamic domain via a *system description* populated with statements in sprits of *"a causes p"*. The system description captures a transition diagram specifying the behavior of a given domain. An $\mathcal{ALM}$ system description consists of a *theory* and a *structure*. A theory is comprised of a hierarchy of *modules*. A module contains declarations of sorts, attributes, and properties of the domain, together with axioms describing the properties. The properties that can be changed by actions are called *fluents* and modeled by functions in $\mathcal{ALM}$. The structure defines instances of the sorts. We now illustrate the introduced concepts on the $\mathcal{ALM}$ formalization of the *JS Discourse* domain. The resulting formalization will depict the transition diagram in Figure 2.

#### 2.1.1.1 LANGUAGE $\mathcal{ALM}$ THEORY

We start by defining a system description named *JS_discourse*:

```
system description JS_discourse                                    (1)
```

*Example 3: JS Discourse System Description*

Next, we describe the theory. The *JS Discourse* theory contains a single module for implicit knowledge about moving. Example 4 define the name of the theory and the name of the module:

```
theory JS_discourse_theory                                         (2)
  module JS_discourse_module                                       (3)
```

*Example 4: JS Discourse Theory*

After the theory and module are named, we declare the sorts, attributes, fluents, and axioms. Let us first consider the sorts. Assume that there are two base sorts named *universe* and *actions*. The *universe* sort is the root of the sort hierarchy for all objects. The *actions* sort is the root of the sort hierarchy for all actions represented in our theory. We begin the sort hierarchy by defining the sorts *points* and *things* inheriting from *universe*. The *points* sort denotes locations and the *things* sort denotes objects. We then define an *agents* sort that inherits from the *things* sort and represents agents that move. Next, we create the *move* sort as a sub-sort of the global *actions* sort. The *move* sort has three related attributes: *actor*, *origin*, and *destination*. The *actor* is an *agent* and the *origin* and *destination* are *points*. Example 5 declares the sorts in the *JS Discourse* domain.

```
sort declarations                                                    (4)
  points, things :: universe                                         (5)
  agents :: things                                                   (6)
  move :: actions                                                    (7)
    attributes                                                       (8)
      actor : agents                                                 (9)
      origin : points                                               (10)
      destination : points                                          (11)
```

*Example 5: JS Discourse Sort Declarations*

We then move to the module's fluents. A single fluent, *loc_in*, is defined for the module and maps *things* to *points*. The fluent *loc_in* is defined in Example 6.

```
function declarations                                                (12)
  fluents                                                            (13)
    basic                                                            (14)
      loc_in : things -> points                                      (15)
```

*Example 6: JS Discourse Function Declarations*

The final section of a module defines the axioms. We focus on three kinds of axioms in this work:

- Dynamic Causal Laws
- Executability Conditions
- State Constraints

Statements of two kinds, dynamic causal laws and executability conditions, are sufficient to capture the rules of our running example. Example 7 presents these statements. A dynamic causal law in lines (18) – (21) states that if an action *move* occurs, then the actor's location is set to the destination. The executability conditions in lines (22) – (28) state that it is impossible for a *move* action to occur if:

- The actor's location doesn't match the action's origin, and
- The destination is the same location as the actor's current location.

```
axioms                                                                (16)
  dynamic causal laws                                                 (17)
    occurs(X) causes loc_in(A) = D                                    (18)
      if instance(X, move),                                           (19)
      actor(X) = A,                                                   (20)
      destination(X) = D.                                             (21)
  executability conditions                                            (22)
    impossible occurs(X) if instance(X, move),                        (23)
      actor(X) = A,                                                   (24)
      loc_in(A) != origin(X).                                         (25)
    impossible occurs(X) if instance(X, move),                        (26)
      actor(X) = A,                                                   (27)
      loc_in(A) = destination(X).                                     (28)
```

*Example 7: JS Discourse Axioms*

Comprehensively, Examples 3 - 7 define the module for the *JS Discourse*.

#### 2.1.1.2 LANGUAGE $\mathcal{ALM}$ STRUCTURE

The system description's structure identifies the specific events and objects from the narrative. Considering our *JS Discourse*, we have the actors *John* and *Sandra*, the location *hallway*, and the action *go*. A structure representing this information instantiates *John*, with the instance name *j*, and *Sandra*, with the instance name *s*, as sort *agents*, and the *hallway*, with the instance name *h*, as sort *points*. The structure defines *go* as function that is an instance of *move*, where the first argument is move's *actor* and the second argument is move's *destination*. Example 8 gives this structure.

```
structure john_and_sandra                                             (29)
  instances                                                           (30)
    j, s in agents                                                    (31)
    h in points                                                       (32)
    go(X,P) in move                                                   (33)
      actor = X                                                       (34)
      destination = P                                                 (35)
```

*Example 8: JS Discourse Structure*

This concludes the system description of the *JS Discourse.* The semantics of this structure is captured by the transition diagram in Figure 2.

### 2.1.2 LANGUAGE $\mathcal{ALM}$ HISTORY

The final component of an $\mathcal{ALM}$ program is the history. The history is a particular scenario described by observations about the values of fluents and events that occur. In the case of narratives, a history describes the known sequence of events by stating the occurrence of specific actions at different time steps. The *JS Discourse* history consists of the events of:

- First *John* moving to the *hallway*.
- Then *Sandra* moving to the *hallway* too.

Example 9 presents the *JS Discourse* history. Line (37) states an instance of the action *go* happened at time step 0 with *John (j)* as the *actor* and the *hallway (h)* as the *destination*. Likewise, line (38) states an instance of action *go* at time step 1 with *Sandra (s)* as the *actor* and the *hallway (h)* as the *destination.*

```
history                                                    (36)
  happened(go(j, h), 0).                                   (37)
  happened(go(s, h), 1).                                   (38)
```

*Example 9: JS Discourse History*

We have now defined all parts of the *JS Discourse* $\mathcal{ALM}$ program in Lines (1) – (38) in Examples 3 - 9. Appendix A contains the combined elements of the *JS Discourse* program.

### 2.1.3 LANGUAGE $\mathcal{ALM}$ MODEL

Language $\mathcal{ALM}$ programs formulate the transition diagrams and trajectories through the transition diagram can be "compatible" with a history or not. Intuitively, compatible trajectory means that we can find a path through the transition diagram corresponding to the history's events. For example, the history in Example 9 is compatible to the *JS Discourse* because the events correspond to the

trajectory <$s_3$, $a_5$, $s_2$, $a_4$, $s_1$>. through the diagram in Figure 2. Compatible trajectories of a history are *models*. An $\mathcal{ALM}$ model is composed of facts and their associated time steps. For example, the model for the *JS Discourse* $\mathcal{ALM}$ program contains the following facts:

```
happened(go(j, h), 0), happened(go(s, h), 1),                    (1)
loc_in(j, h, 1), loc_in(j, h, 2),                                (2)
loc_in(s, h, 2)                                                  (3)
```

*Example 10: JS Discourse $\mathcal{ALM}$ Model*

Line (1) contains facts representing which actions occurred and when. The fact *happened(go(j, h), 0)* states that the event of *John* going to the *hallway* occurred at time step 0. The second fact can be read similarly. Lines (2) and (3) contain facts related to *John* and *Sandra's* locations. The fact *loc_in(j, h, 1)* can be read as "the location of *John* is the *hallway* at time step 1" and likewise for *John's* location time step 2 and for *Sandra's* location at time step 2. The model indicates that the action of *John* travelling to the *hallway* occurs at time step 0, so his location becomes the *hallway* after the action. The same can be said for the action of *Sandra's* journey to the *hallway* during time step 1.

## 2.2 SYSTEM CALM

System *CALM* is an $\mathcal{ALM}$ implementation developed by researchers Wertz, Chandrasekan, and Zhang at Texas Tech University (2018). System CALM uses an $\mathcal{ALM}$ program to produce a model of a dynamic domain. The *JS Discourse* $\mathcal{ALM}$ program in Section 2.1 uses the CALM syntax. System CALM requires two additional components to the $\mathcal{ALM}$ program:

- Specify the computational task
- Specify the max steps

### 2.2.1 COMPUTATIONAL TASKS

The CALM system can solve computational tasks in temporal projection and planning. For our work, temporal projection is sufficient. Temporal projection is the process of determining the effect of a given sequence of actions executed in a given initial situation. In the case of a narrative, the initial situation is often unknown, but the sequence of actions and their effects are known. To perform temporal projection, we insert the following statement in the $\mathcal{ALM}$ program prior to the history:

```
temporal projection                                                    (1)
```

*Example 11: Temporal Projection*

### 2.2.2 MAX STEPS

Additionally, our $\mathcal{ALM}$ program must designate the max number of steps that can be performed in the trajectory. In temporal projection problems, this information denotes the final state's time step. Max steps equals the number of events plus one because we consider each event as having its own step and an additional time point for the final state. To define the max steps for the *JS Discourse* program, we simply count the two events and add one. The max steps is stated in the following manner:

```
max steps 3                                                            (2)
```

*Example 12: JS Discourse Max Steps*

Appendix B contains the $\mathcal{ALM}$ program input to CALM for the *JS Discourse*.

## 2.3 KNOWLEDGE BASE COREALMLIB

The *CoreALMLib* is an $\mathcal{ALM}$ library of generic commonsense knowledge for modeling dynamic domains developed by Daniela Inclezan (2016). The library foundation is the Component Library (CLib) (Barker, Porter, & Clark, 2001), which is a library of general, reusable, composable, and interrelated components of knowledge.

CLib consists of knowledge from linguistic and ontological resources, such as VerbNet, WordNet, FrameNet, a thesaurus, and an English dictionary. CLib was successfully used in a challenge problem for DARPA's Rapid Knowledge Formation project, where the goal was to provide a software environment in which a biologist can build a knowledge base from information in a Cell Biology textbook. The CoreALMLib was created by translating portions of CLib to obtain descriptions of 123 action classes grouped into 43 reusable modules. The modules contain action classes and are organized into a hierarchy of modules.[1]

The root of the module hierarchy is the *entity_event_and_action* module. This module declares the sort hierarchy of basic entities and events. Figure 3 presents the sort hierarchy declared in this module.

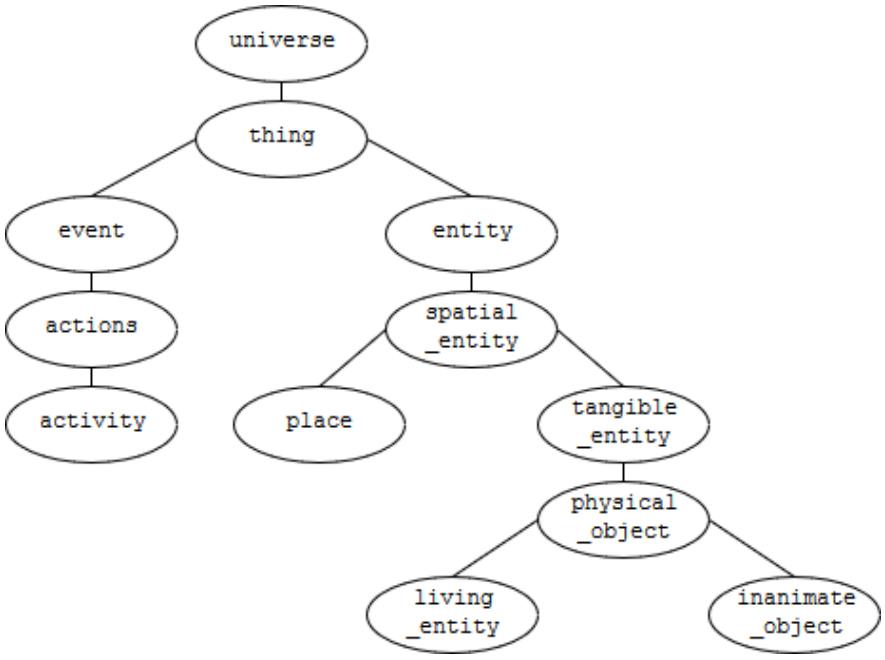


*Figure 3: entity_event_and_action Sort Hierarchy*

[1] The CoreALMLib is available online and the hierarchy can be viewed at https://ceclnx01.cec.miamioh.edu/~inclezd/coreALMlib/ModuleHierarchy.htm.

The *event* sort, unlike other sorts declared in the module, also declares attributes associated with it and sorts inheriting from *event*. The attributes support mapping between defined sorts and roles associated with events. For example, *agent, object,* and *destination* are three attributes associated with *event.* The sort declarations for *event,* with some of *event*'s attributes, *actions*, and *activity* from the *entity_event_and_action* module are presented in Example 13:

```
sort declarations                                                    (1)
  event :: thing                                                     (2)
    attributes                                                       (3)
      agent : entity -> booleans                                     (4)
      object : entity -> booleans                                    (5)
      destination : spatial_entity -> booleans                       (6)
      ...                                                            (7)
  actions :: event                                                   (8)
  activity :: event                                                  (9)
```

*Example 13: entity_event_and_action Sort Hierarchy*

The *entity_event_and_action* module has thirteen branches inheriting from the root, each for a different category of action classes. For example, one branch begins with the *motion* module. The *container_motion* and *locomotion* modules inherit from the *motion* branch to represent knowledge on specific aspects of movement. Consider the CoreALMLib module *locomotion* presented in Example 14. The module contains commonsense axioms about events related to an agent's movement. The *locomotion* module inherits from the *motion* module, and implements more specific classes of movement, such as *carry*, *go_to*, *leave*, and *walk*. Import statements, like the statement in line (2), denotes the dependence on knowledge in external modules. Line (2) suggests that we inherit module *motion* from theory *motion.* Appendix C contains the *motion* module. The *locomotion* module declares a new *event* sort named *locomotion* extending from the *move* sort in line (6). The *move* sort is declared in the *motion* module as a

subsort of *actions.* Therefore, the *locomotion* event sort inherits all the attributes associated with events. Additionally, the *locomotion* module contains a single fluent, *held_by*, to define the objects held by an agent. This fluent is stated in lines (12) – (15). Lines (16) – (37) list state constraints, dynamic causal laws, and executability conditions for the actions.

```
theory locomotion                                                    (1)
  import motion.motion from CoreALMLib                               (2)
  module locomotion                                                  (3)
    depends on motion                                                (4)
    sort declarations                                                (5)
      locomotion :: move                                             (6)
      carry :: locomotion                                            (7)
      go_to :: locomotion                                            (8)
      leave :: locomotion                                            (9)
      walk :: locomotion                                            (10)
    function declarations                                           (11)
      fluents                                                       (12)
        basic                                                       (13)
          held_by : tangible_entity *                               (14)
                    tangible_entity -> booleans                     (15)
    axioms                                                          (16)
      state constraints                                             (17)
        false if instance(X, locomotion),                           (18)
          -defined_agent(X).                                        (19)
        false if instance(X, carry),                                (20)
          -defined_agent(X).                                        (21)
        false if instance(X, carry),                                (22)
          agent(X, A),                                              (23)
          -instance(A, tangible_entity).                            (24)
        false if instance(X, go_to),                                (25)
          -defined_destination(X).                                  (26)
        false if instance(X, leave),                                (27)
          -defined_origin(X).                                       (28)
        object(X, Y) if instance(X, locomotion),                    (29)
          agent(X, Y).                                              (30)
        agent(X, Y) if instance(X, locomotion),                     (31)
          object(X, Y).                                             (32)
      executability conditions                                      (33)
        impossible occurs(X) if instance(X, carry),                 (34)
          object(X, O),                                             (35)
```

```
        agent(X, A),                                    (36)
        -held_by(O, A).                                 (37)
```

*Example 14: CoreALMLib Locomotion Module*

## 2.4 VERB LEXICON VERBNET

*VerbNet* is the largest on-line verb lexicon currently available for English (Palmer, 2018; Schuler, 2005). The verb lexicon is hierarchical, domain-independent, and has a broad-coverage with mappings to other lexical resources, such as FrameNet (Baker, Fillmore, & Lowe, 1998) and WordNet (Miller, Beckwith, Fellbaum, Gross, & Miller, 1990). The lexicon is organized into a hierarchical set of verb classes aiming to achieve syntactic and semantic coherence between members of a class. Each verb class is characterized by a set of verbs, thematic roles, syntactic frames, and semantic predicates.

For example, the verb *run* is defined by VerbNet class *run-51.3.2*. *Run-51.3.2* has 97 members: *bolt*, *bound*, *climb*, *crawl*, *frolic*, etc. The class has three thematic roles, *agent*, *theme*, and *location*, and five syntactic frames that define syntactic uses of the verb class. The class *run-51.3.2* has two subbranches, *run-51.3.2-1* and *run-51.3.2-2* which define more specific instances of run.

## 2.5 SYSTEM TEXT2DRS

System *Text2DRS***,** developed by Gang Ling (2018), converts a narrative to a *discourse representation structure (DRS)* in Neo-Davidsonian style (Kamp & Reyle, 1993). The DRS encodes information about the entities and their relations in the narrative. Text2DRS relies on semantic role labeling and coreference resolution to create the DRS. Semantic role labeling is the process of assigning labels to words or phrases in a sentence to denote their semantic role in the sentence. The DRS states the result of the semantic role labeling by listing entities, events, and properties from the narrative.

Details on the Text2DRS system can be found in (Ling, 2018). To exemplify the Text2DRS system at work, consider the *JS Discourse* as an input. System Text2DRS outputs the following DRS:

```
% r1, r2, r3, e1, e2                                                    (1)
% ================================================                      (2)
entity(r1). entity(r2). entity(r3).                                     (3)

property(r1, "John"). property(r2, "hallway").                          (4)
property(r3, "Sandra").                                                 (5)

event(e1).                                                              (6)
event(e2).                                                              (7)

eventType(e1, "51.3.2-1"). eventType(e2, "51.3.2-1").                   (8)

eventTime(e1, 0). eventTime(e2, 1).                                     (9)

eventArgument(e1, "Theme", r1).                                        (10)
eventArgument(e1, "Destination", r2).                                  (11)
eventArgument(e2, "Theme", r3).                                        (12)
eventArgument(e2, "Destination", r2).                                  (13)
```

*Example 15: JS Discourse Text2DRS Output*

This DRS denotes the three subjects in the narrative and labels *John* as entity *r1*, the *hallway* as entity *r2*, and Sandra as entity *r3* in lines (3) – (5). Lines (6) – (8) list the two events that occurred in the narrative. Event *e1* matches the event "travelled" and event *e2* matches the event "journeyed". These events are instances of the VerbNet class run-51.3.2-1, stated in line (8). Lines (9) – (11) are the conditions associated with the two events. *John* travelling to the *hallway* is as an event occurring at time step 0, with John fulfilling the thematic role *theme* and the hallway fulfilling the thematic role *destination*. Likewise, the event of *Sandra* journeying to the *hallway* is denoted as an event occurring at time step 1, with Sandra as the thematic role *theme* and the hallway as the thematic role *destination*.

The inner workings of Text2DRS rely on the semantic role labeler LTH (Johansson & Nugues, 2007) that annotates a given sentence with the semantic roles stemming from the PropBank lexicon (Palmer et al., 2005). PropBank is a linguistic verb resource that systematizes knowledge about verbs with respect to their predicate-argument structure. PropBank is organized into sets of roles associated with verbs. Text2DRS utilizes the SemLink (Bonial et al., 2013) resource to map the LTH's PropBank annotations to the thematic roles of VerbNet. Table 1 presents

- The first sentence of the JS Discourse,
- Its PropBank semantic role labels,
- Its VerbNet thematic role labels.

Example 16 provides a SemLink entry which gives the mappings from PropBank to VerbNet for the verb *travel.*

| **Sentence** | John | travelled | to the hallway. |
|---|---|---|---|
| **PropBank Annotations** | A0 | Travel.01 | A1 |
| **VerbNet Annotations** | Theme | Run-51.3.2-1 | Destination |

*Table 1: JS Discourse Sentence, PropBank, and VerbNet Annotations*

```
<predicate lemma="travel">
  <argmap pb-roleset="travel.01" vn-class="51.3.2-1">
    <role pb-arg="0" vn-theta="Theme" />
    <role pb-arg="1" vn-theta="Destination" />
  </argmap>
</predicate>
```

*Example 16: SemLink Entry for Travel*

## 2.6 bAbI Project

The *bAbI project* is a Facebook AI Research project whose goal is the advancement of automatic text understanding and reasoning (Weston et al., 2015). The bAbI project contributes to the field of KRR by providing data sets to evaluate progress in various reasoning tasks. Part of the project is the *QA bAbI tasks*. The QA bAbI tasks provide benchmarking abilities for QA systems in twenty reasoning tasks. Each task focuses on unique aspects of text and reasoning. The tasks contain simple narratives and ask questions during the narrative. The questions were developed so a human could potentially achieve 100% accuracy when answering the questions. Each task contains 2000 questions split into a training set of 1000 questions and a testing set of 1000 questions.

Examples 17 and 18 provide example narratives from the Three Supporting Facts and Agent's Motivations QA bAbI tasks. In the Three Supporting Facts task, a question is given whose answer requires information from three separate sentences. This tests a system's ability to combine related information in an effective manner. In Example 17, the system must combine the information about *John* and the *apple* from sentences (1), (2), and (3) to answer the question "Where was the apple before the kitchen?" in line (5) with the answer "office".

```
John picked up the apple.                                    (1)
John went to the office.                                     (2)
John went to the kitchen.                                    (3)
John dropped the apple.                                      (4)
Where was the apple before the kitchen? A: office            (5)
```

*Example 17: BAbI Three Supporting Facts Example*

Example 18 is from the Agent's Motivations task. This task tests a system's ability to answer questions about why an agent performed a specific action. In Example

18, the system must combine knowledge about the actions *go to the kitchen* and *get food* to the cause of *John* being hungry, and then extend this knowledge to *Daniel*. Line (5) asks "Where does Daniel go?". A system requires the ability to learn that *Daniel* would go to the *kitchen* because that was the action *John* took when he was also hungry. Line (6) directly tests a system's ability to understand that *John's* motive for going to the *kitchen* was his *hunger*.

```
John is hungry.                                          (1)
John goes to the kitchen.                                (2)
John grabbed the apple there.                            (3)
Daniel is hungry.                                        (4)
Where does Daniel go? A: kitchen                         (5)
Why did John go to the kitchen? A: hungry                (6)
```

*Example 18: BAbI Agent's Motivations Example*

In this thesis we consider tasks 1, 2, 3, 5, 6, 7, and 8 from the QA bAbI dataset. These tasks are selected because they contain action-based narratives that the system Text2ALM is designed for.

## 3 TEXT2ALM SYSTEM IMPLEMENTATION

QA system Text2ALM is based on the methodology proposed by Lierler et al. (2017). This approach utilizes:

- The action language $\mathcal{ALM}$ (Inclezan & Gelfond, 2015)
- The VerbNet lexicon (Palmer, 2018; Schuler, 2005) annotated with information on relevant $\mathcal{ALM}$ modules encoding commonsense knowledge about verbs

The represented knowledge is then used to derive inferences about the text. System Text2ALM serves as a proof of concept for this methodology. It enables us to identify successes, challenges, and limitations of the approach.

We explain the Text2ALM system implementation through its four main tasks:

1. Text2DRS Processing - Entity, Event, and Relation Extraction
2. DRS2ALM Processing - Creation of $\mathcal{ALM}$ Program
3. CALM Processing - $\mathcal{ALM}$ Model Generation and Interpretation
4. QA Processing

The Text2ALM system architecture was designed around these four tasks. Figure 4 models the Text2ALM system architecture:

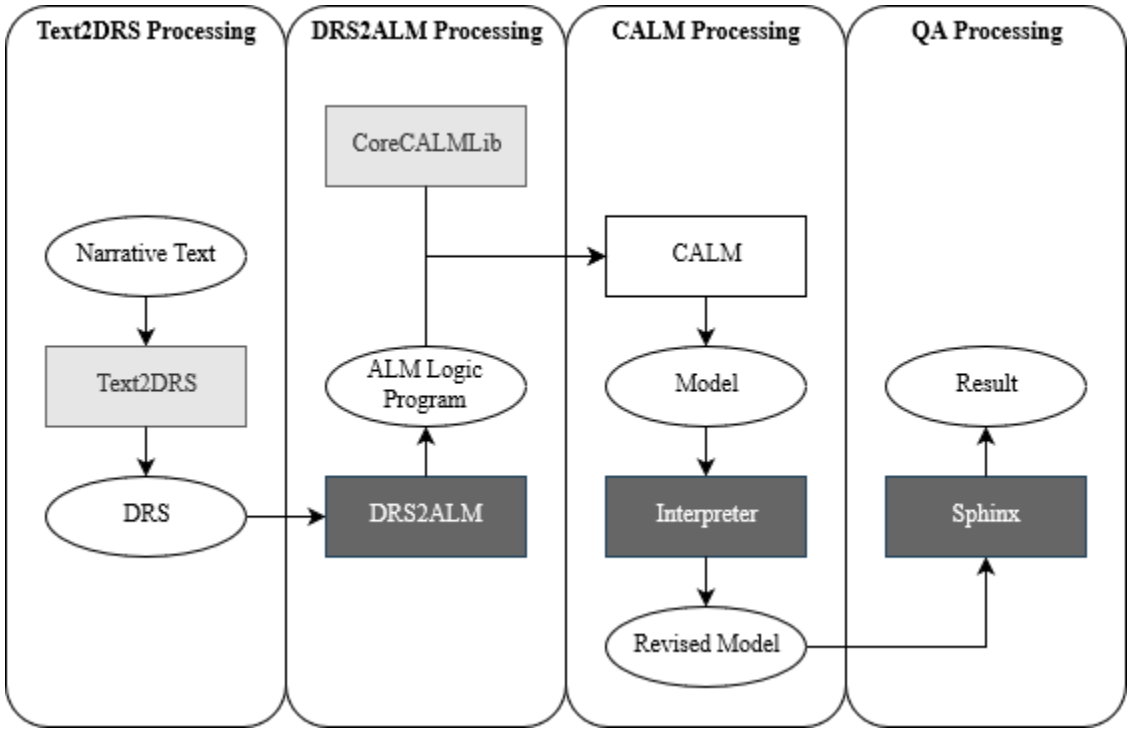


*Figure 4: Text2ALM System Architecture*

Each process is represented by its own column. Ovals identify inputs and outputs. Rectangles stand for systems or resources. White rectangles denote existing, unmodified resources. Grey rectangles are used for existing, but modified resources. Black rectangles signify newly developed resources.

We now explain how each process accomplishes its task. We rely on two examples to outline the tasks: the *JS Discourse* and an altered version of the *JS Discourse*

titled the *JSB Discourse.* The *JSB Discourse* is a slightly more complex version of the *JS Discourse* where we add one more sentence to the narrative:

```
John travelled to the hallway.                    (1)
Sandra journeyed to the hallway.                  (2)
John got the ball there.                          (3)
```

*Example 19: JSB Discourse*

The *JSB Discourse* is slightly more intricate because it introduces a new class of action, possession. The additional sentence results in the same DRS as the *JS Discourse* presented in Example 15, but with the following additions to the given lines:

- Line (1): `r4, e3`
- Line (3): `Add entity(r4)`
- Line (5): `property(r4, "ball")`
- Line (7): `event(e3)`
- Line (8): `eventType(e3, "get-13.5.1-1")`
- Line (9): `eventTime(e3, 2)`
- New Line: `eventArgumentRole(e3, "Agent", r1)`
- New Line: `eventArgumentRole(e3, "Theme", r4)`

The complete DRS for the *JSB Discourse* is offered in Appendix D.

## 3.1 TEXT2DRS PROCESSING - ENTITY, EVENT, AND RELATION EXTRACTION:

The first step of converting a narrative to an $\mathcal{ALM}$ model is Entity, Event, and Relation extraction. This extraction is conducted by the Text2DRS system (Ling, 2018) which produces a DRS. The DRS maps the narrative's action verbs to their corresponding VerbNet classes. The DRS also annotates actions with the thematic roles in each VerbNet class. The thematic roles communicate their role in a narrative's events and how their properties were affected.

### 3.1.1 Extensions To Text2DRS

Extensions were made to the original Text2DRS system. The changes include:

1. Update VerbNet (Palmer, 2018; Schuler, 2005) and SemLink (Bonial et al., 2013)
2. Alter the DRS output
3. Improve system efficiency
4. Additions to SemLink

First, we updated Text2DRS to use the most recent edition of VerbNet, version 3.3 (Palmer, 2018; Schuler, 2005). VerbNet 3.3 extends earlier VerbNet versions by covering more English verbs and altering attributes of existing verb classes. System Text2DRS was updated to get the most recent VerbNet classes and thematic roles. Text2DRS relies on SemLink library (Bonial et al., 2013) to identify attributes related to VerbNet. Therefore, we also updated SemLink to be compatible with VerbNet 3.3.

Second, we altered the DRS output. Originally the Text2DRS system output *eventType* arguments in the format of *eventType(<eventID>, <VerbNet Class ID>).* For readability and testability purposes, the output was updated to the format of *eventType(<eventID>, <VerbNet Class Name-ID>).* This change made the DRS easier to read for a human, and enabled users to verify the mapping between narrative event to VerbNet class more easily. For example, Line (8) in the *JS Discourse* DRS in Example 15 changes from "*eventType(e1, "51.3.2-1"). eventType(e2, "51.3.2-1")*" to "*eventType(e1, "run-51.3.2-1"). eventType(e2, "run-51.3.2-1")*".

Third, we improved the system's efficiency. System Text2DRS relies on two natural language processing systems: LTH (Johansson & Nugues, 2007) and Stanford CoreNLP (Manning et al., 2014). The initial version of Text2DRS relied on JAR

executables to utilize these systems, which required substantial bootstrapping time for each narrative. The solution was to modify the system design to a client-server architecture. We create server instances of the LTH and Stanford CoreNLP tools, which can be accessed remotely by the Text2DRS client. This enabled Text2DRS to avoid the bootstrapping time for each narrative, and we can use both tools in parallel. System Text2DRS's execution time for a narrative decreased substantially due to these changes.

Fourth, we added new links to the SemLink library (Bonial et al., 2013). The SemLink library maps PropBank roles to VerbNet classes, but the library does not contain mappings for every PropBank verb. We identified several verbs where the Text2DRS sub-system lacked SemLink mappings. For example, the verbs *hand.02* and *pass.01* often annotated events for a transfer of possession. However, SemLink does not have mappings for either of these. Therefore, new entries were inserted into the SemLink library to translate PropBank annotations to VerbNet annotations for these verbs. The additions were written by taking information from similar verbs and PropBank documentation. Example 20 displays the additions to SemLink for *hand.02* and *pass.01*.

```
<predicate lemma=”hand”>                                          (1)
  <argmap pb-roleset=”hand.02” vn-class=”13.1-1”>                 (2)
    <role pb-arg=”0” vn-theta=”Agent” />                          (3)
    <role pb-arg=”1” vn-theta=”Theme” />                          (4)
    <role pb-arg=”2” vn-theta=”Recipient” />                      (5)
  </argmap>                                                       (6)
</predicate>                                                      (7)
<predicate lemma=”pass”>                                          (8)
  <argmap pb-roleset=”pass.01” vn-class=”13.1”>                   (9)
    <role pb-arg=”0” vn-theta=”Agent” />                         (10)
    <role pb-arg=”1” vn-theta=”Theme” />                         (11)
    <role pb-arg=”2” vn-theta=”Recipient” />                     (12)
  </argmap>                                                      (13)
  <argmap pb-roleset=”pass.01” vn-class=”11.1”>                  (14)
    <role pb-arg=”0” vn-theta=”Agent” />                         (15)
    <role pb-arg=”1” vn-theta=”Theme” />                         (16)
    <role pb-arg=”2” vn-theta=”Destination” />                   (17)
  </argmap>                                                      (18)
  <argmap pb-roleset=”pass.01” vn-class=”17.1”>                  (19)
    <role pb-arg=”0” vn-theta=”Agent” />                         (20)
    <role pb-arg=”1” vn-theta=”Theme” />                         (21)
    <role pb-arg=”2” vn-theta=”Destination” />                   (22)
  </argmap>                                                      (23)
</predicate>                                                     (24)
```

*Example 20: SemLink Hand.02 and Pass.01 Additions*

## 3.2 DRS2ALM Processing - Creation of $\mathcal{ALM}$ Program

The second processing stage in the Text2ALM system is translating the DRS to an $\mathcal{ALM}$ program. Information from the DRS is systematically used to create an $\mathcal{ALM}$ system description, a history, and indicate the max steps. The system description contains the narrative’s entities, event types, and commonsense knowledge associated with the events. The history represents the narrative’s order of events. First this section describes the process for automatically creating the system description. Then we outline how to derive the max steps and history from the DRS. Lastly, we describe the process for creating the Text2ALM system’s knowledge base.

### 3.2.1 DRS2ALM System Description

We automatically generate the two parts of a system description: the theory and the structure. We now explore how the theory and structure are produced from the DRS.

#### 3.2.1.1 DRS2ALM Theory

The purpose of the an $\mathcal{ALM}$ theory is to represent the knowledge relevant for a domain. The Text2ALM system gathers relevant knowledge by selecting the modules containing information related to the narrative's events. Given a narrative, the Text2DRS system will produce a respective DRS. The *eventType* arguments in such a DRS define a VerbNet class associated with each event captured by the narrative. In Section 3.2.5 we describe the details of the so called CoreCALMLib that consists of the $\mathcal{ALM}$ modules, which intuitively encode the commonsense knowledge about actions corresponding to VerbNet classes. In an $\mathcal{ALM}$ theory that we generate, we import the CoreCALMLib modules associated with each VerbNet classes in the narrative. For example, expressions from *JSB Discourse* DRS result in the following import statements in the $\mathcal{ALM}$ theory:

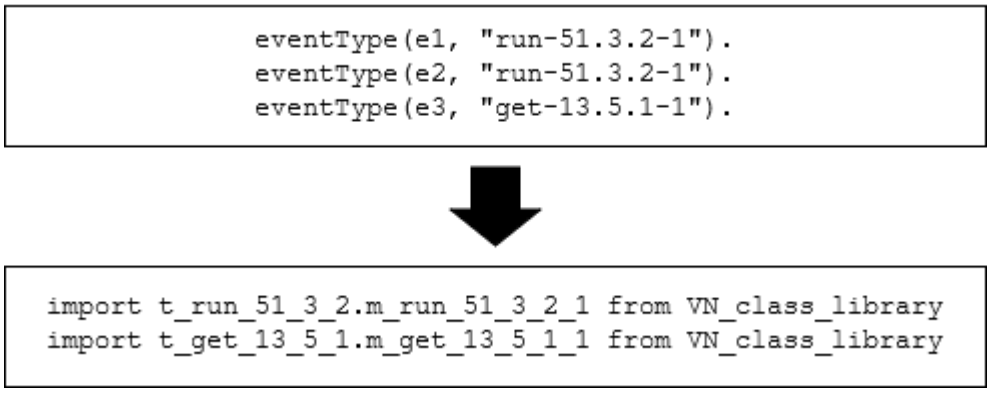


*Figure 5: Text2ALM Module Importing*

After we import the modules containing the implicit knowledge for the events, the theory defines a new module for the sorts unique to the narrative. Each *property* in the DRS is defined as a sort inheriting from a more general sort. For example, the *properties*

from the *JSB Discourse* are *property(r1, "John")*, *property(r2, "hallway")*, *property(r3, "Sandra"),* and *property(r4, "ball")*. From a sort hierarchy perspective, we consider *John*, *hallway, Sandra,* and *ball* as sorts unique to the narrative. Intuitively, the sort, such as *hallway,* is true of all objects that have a property *being_a_hallway*.

Next, we must choose the parents for the new sorts. We rely on the DRS *eventArgument* fields for this task. An *eventArgument* identifies the VerbNet thematic roles that an entity plays in the narrative. We grouped thematic roles of VerbNet into four "parent" sorts originally defined by the CoreALMLib: *spatial_entity*, *place*, *living_entity*, and *entity*. We identify the roles an entity plays throughout the DRS and choose the entity sort's parent from the roles' categories. Table 2 lists the thematic roles associated with each sort.

| **Parent Sort** | **living_entity** | **place** | **spatial_entity** | **entity** |
|---|---|---|---|---|
| **Associated Thematic Roles** | Actor<br>Agent<br>Beneficiary<br>Cause<br>Co-Agent<br>Co-Theme<br>Experiencer<br>Participant<br>Patient<br>Recipient<br>Theme<br>Undergoer | Location<br>Place | Destination<br>Initial_location<br>Source | Instrument<br>Material<br>Pivot<br>Product<br>Stimulus<br>Time<br>Trajectory<br>Topic<br>Value<br>Result<br>Attribute<br>Duration<br>Extent<br>Final_time<br>Frequency<br>Goal<br>Initial_time |

*Table 2: VerbNet Thematic Roles to CoreALMLib Sorts*

Throughout a discourse, the entity, such as *John,* may be associated with different roles. We use a prioritized sort order if an entity is associated with thematic roles categorized with different sorts. Priorities on the parent sorts are defined as follows, where >> is transitive and denotes the fact that the left argument has a higher priority than the right argument:

```
living_entity >> place >> spatial_entity >> entity
```

#### 3.2.1.2 DRS2ALM STRUCTURE

The structure is the second part of the system description. The structure defines instances of entities and events. We automate this process by considering the *property, eventType*, and *eventArgument* arguments in the DRS.

First, we produce instances of a narrative's entities. The *property* arguments are used to define an instance of every entity. Figure 6 shows the mapping between *properties* and entity instances for the *JSB Discourse.*

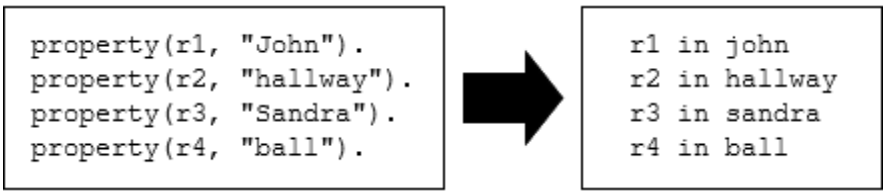


*Figure 6: DRS Properties to Structure Entities*

Next, we create instances of the events from the DRS *eventType* and *eventArgument* fields. An event instance is declared for every *eventType*. Attributes for the event are defined by finding all *eventArguments*. For each *eventArgument* associated with the event, we define an attribute corresponding to the *eventArgument's* VerbNet thematic role. The attribute is the name of the thematic role prepended with *vn_*. Figure 7 shows this process for the *JSB Discourse.*

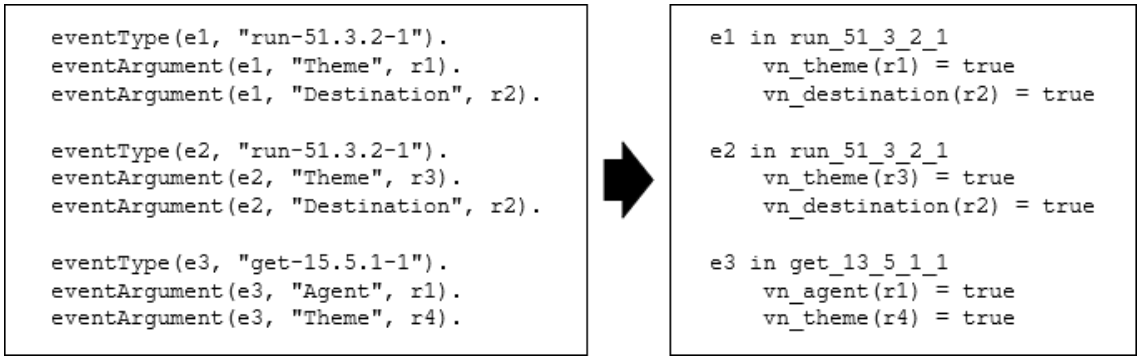


*Figure 7: DRS EventTypes and EventArguments to Structure Events*

### 3.2.2 DRS2ALM HISTORY

Once we generate the system description, we then produce a history. The history declares the order in which events happened in the narrative. The order of events is defined by the *eventTime* arguments in the DRS. For every argument in the format *eventTime(eventNumber, timeStep)*, we include a fact *happened(eventNumber, timeStep)* in the history. Figure 8 shows the mapping between *eventTime* arguments and the history for the *JSB Discourse*.

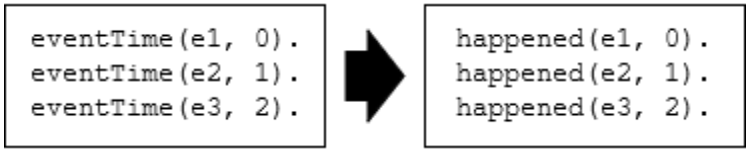


*Figure 8: DRS EventTimes to History Events*

### 3.2.3 DRS2ALM MAX STEPS

Finally, we state the max steps in the trajectory. The max steps value is equal to the number of events plus one for a final state. This is calculated by counting the number of *event* arguments in the DRS and adding one. In the *JSB Discourse*, there are three events, so the max steps value is 3 + *1*, or *max steps = 4*.

### 3.2.4 DRS2ALM PROGRAM EXAMPLE

To outline the conversion from DRS to an $\mathcal{ALM}$ program, consider the *JSB Discourse* and its Text2DRS output in Appendix D. Example 21 depicts the $\mathcal{ALM}$ program generated for the narrative. The elements from Figures 5, 6, 7, and 8 were combined to form this $\mathcal{ALM}$ program.

```
system description example                                          (1)
  theory example                                                    (2)
    import t_run_51_3_2.m_run_51_3_2_1 from VN_class_library        (3)
    import t_get_15_5_1.m_get_15_5_1_1 from VN_class_library        (4)
    module example                                                  (5)
      depends on m_run_51_3_2_1                                     (6)
      sort declarations                                             (7)
        john, sandra, ball :: living_entity                         (8)
        hallway :: place                                            (9)
  structure example                                                (10)
    instances                                                      (11)
      r1 in john                                                   (12)
      r2 in hallway                                                (13)
      r3 in sandra                                                 (14)
      r4 in ball                                                   (15)
      e1 in run_51_3_2_1                                           (16)
        vn_theme(r1) = true                                        (17)
        vn_destination(r2) = true                                  (18)
      e2 in run_51_3_2_1                                           (19)
        vn_theme(r3) = true                                        (20)
        vn_destination(r2) = true                                  (21)
      e3 in get_15_5_1_1                                           (22)
        vn_agent(r1) = true                                        (23)
        vn_theme(r4) = true                                        (24)
temporal projection                                                (25)
max steps 4                                                        (26)
history                                                            (27)
  happened(e1, 0).                                                 (28)
  happened(e2, 1).                                                 (29)
  happened(e3, 2).                                                 (30)
```

*Example 21: JSB Discourse $\mathcal{ALM}$ Program*

Note that a peculiarity in the automatically generated program is that the *ball* sort extends from *living_entity* in line (8). This is the result of the mappings between VerbNet

thematic roles to CoreALMLib sorts previously given in Table 2. Even though we want the *ball* to inherit the same attributes and interact with the same axioms as *living_entity*, the name is counter-intuitive. A piece of future work is to re-evaluate the CoreALMLib sort hierarchy and simplify the sort hierarchy to more closely align with VerbNet.

### 3.2.5 Creating CoreCALMLib

Besides the $\mathcal{ALM}$ program that is unique to each narrative, the Text2ALM system needs a commonsense knowledge base to use for all narratives. The knowledge base's goal is to be a library of implicit knowledge to reason about actions and their effects. The CoreALMLib (Inclezan, 2016) served as a promising starting point for such a knowledge base. However, the CoreALMLib required systematic changes to work within the Text2ALM framework. We titled the resulting library of $\mathcal{ALM}$ modules the Core***C***ALMLib. Changes between the CoreALMLib and CoreCALMLib are categorized into four groups:

1. Syntactic changes
2. Fluent extractions
3. VerbNet extensions
4. Axiom changes

#### 3.2.5.1 CoreCALMLib Syntactic Changes

Syntactic differences exist between $\mathcal{ALM}$ modules and the required format for modules in the CALM system. The CALM system requires the user to specifically split axioms into types and label the types as *state constraints, function definitions, executability conditions,* and *dynamic causal laws*. We went through each module and manually grouped the axioms under their corresponding label. As a byproduct, we translated a substantial knowledge base into a format that can be used in other CALM

systems. Figure 9 displays the differences between a set of axioms in the *changing_possession* CoreALMLib module and the same module in the CoreCALMLib.

```
axioms
    false if instance (X, transfer), -defined_object(X).
    occurs(X) causes -possesses(D, O) if instance (X, transfer),
                            donor(X, D),
                            object(X, O).
    impossible occurs(X) if instance(X, transfer),
                            donor(X, D),
                            object(X, O),
                            -possesses(D, O).
```

```
axioms
  state constraints
    false if instance (X, transfer), -defined_object(X).
  dynamic causal laws
    occurs(X) causes -possesses(D, O) if instance (X, transfer),
                            donor(X, D),
                            object(X, O).
  executability conditions
    impossible occurs(X) if instance(X, transfer),
                            donor(X, D),
                            object(X, O),
                            -possesses(D, O).
```

*Figure 9: CoreALMLib to CoreCALMLib Syntactic Changes*

#### 3.2.5.2 CORECALMLIB FLUENT EXTRACTIONS

We extraction the declarations of fluents on the original CoreALMLib. The CoreALMLib library defines fluents to describe properties relevant to the action classes in a module. However, the CoreALMLib library contains instances where fluents with the same name are declared in multiple modules. Yet semantically, since the fluents have the same name, they are assumed to be the same fluents across all the modules. We find that this approach is counterintuitive from the point of view of knowledge base design.

Therefore, we extract all fluent declarations from the CoreALMLib modules and create new modules, whose purpose are to declare fluents. We organize all fluents based

on the properties they capture. For example, the *location_fluents* module encapsulates the related fluents *location* and *is_at*. This module is given in Example 22.

```
module location_fluents                                                (1)
  depends on entity_event_and_action                                   (2)
  function declarations                                                (3)
    fluents                                                            (4)
      basic                                                            (5)
        location : tangible_entity * spatial_entity -> booleans        (6)
        is_at : entity * spatial_entity -> booleans                    (7)
```

*Example 22: Location_Fluents Module*

These fluent modules are contained in a newly created sub-library, titled CALMFluents. The modifications of CoreALMLib into CoreCALMLib include the following. At each branch from the *entity_event_and_action* root we add a corresponding fluent module, which imports the fluents necessary for that branch from CALMFluents. As a result, the ALMFluents library consists of 14 distinct collections of modules, where:

- One of collection contains fluent declarations that are used across the CoreCALMLib. For example, module presented in Example 22 is a part of this collection.
- The remainder correspond to collections of all fluent declarations per each existing branch.

For instance, fluents related to the *motion* branch are contained in the following module of CALMFluents:

```
theory motion_fluents                                                    (1)
  import entity_event_and_action.entity_event_and_action from            (2)
    CoreALMLib                                                           (3)
  import fluents.abuts_fluents from CALMFluents                          (4)
  import fluents.accessible_fluents from CALMFluents                     (5)
  import fluents.blocked_fluents from CALMFluents                        (6)
  import fluents.closed_fluents from CALMFluents                         (7)
  import fluents.confined_fluents from CALMFluents                       (8)
  import fluents.contained_fluents from CALMFluents                      (9)
  import fluents.content_fluents from CALMFluents                       (10)
  import fluents.encloses_fluents from CALMFluents                      (11)
  import fluents.held_by_fluents from CALMFluents                       (12)
  import fluents.is_inside_fluents from CALMFluents                     (13)
  import fluents.location_fluents from CALMFluents                      (14)
  import fluents.possession_fluents from CALMFluents                    (15)
  import fluents.restrained_fluents from CALMFluents                    (16)
  import fluents.shut_out_fluents from CALMFluents                      (17)

  module motion_fluents                                                 (18)
    depends on entity_event_and_action, abuts_fluents,                  (19)
      accessible_fluents, blocked_fluents,                              (20)
      closed_fluents, confined_fluents,                                 (21)
      contained_fluents, content_fluents,                               (22)
      encloses_fluents, held_by_fluents,                                (23)
      is_inside_fluents,                                                (24)
      location_fluents, possession_fluents,                             (25)
      restrained_fluents, shut_out_fluents                              (26)
    function declarations                                               (27)
      fluents                                                           (28)
        basic                                                           (29)
          is_above : spatial_entity * spatial_entity -> booleans        (30)
          is_along : spatial_entity * spatial_entity -> booleans        (31)
          is_behind : spatial_entity * spatial_entity -> booleans       (32)
          is_below : spatial_entity * spatial_entity -> booleans        (33)
          is_beside : spatial_entity * spatial_entity -> booleans       (34)
          is_between : spatial_entity * spatial_entity -> booleans      (35)
          is_contained : tangible_entity -> booleans                    (36)
          is_in_front_of : spatial_entity *                             (37)
            spatial_entity -> booleans                                  (38)
          is_near : spatial_entity * spatial_entity -> booleans         (39)
          is_on : spatial_entity * spatial_entity -> booleans           (40)
          is_opposite : spatial_entity * spatial_entity -> booleans     (41)
          is_over : spatial_entity * spatial_entity -> booleans         (42)
          is_under : spatial_entity * spatial_entity -> booleans        (43)
```

*Example 23: Motion_Fluents Module*

We depict these changes in the following visualizations presented in Figures 10 and 11. Figure 10 displays a subset of the original CoreALMLib hierarchy containing the branches utilized in the *JSB Discourse* and Figure 11 displays the CoreCALMLib fluent hierarchy for the corresponding subset. The rectangle in Figure 11 denotes the difference.

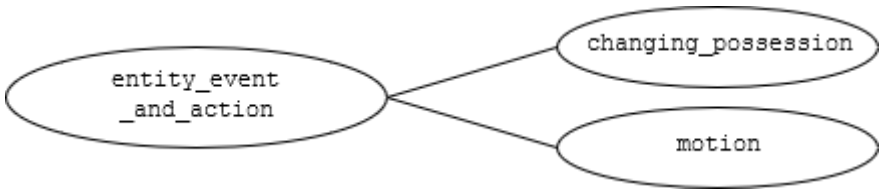


*Figure 10: CoreALMLib Hierarchy*

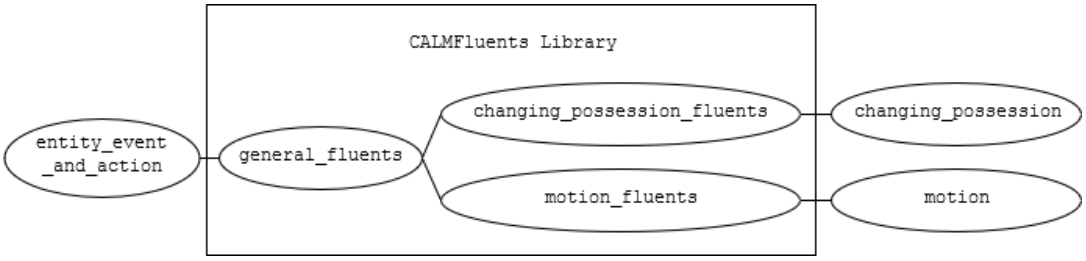


*Figure 11: CoreCALMLib Fluent Hierarchy*

This method of declaring fluents has two key benefits. First, fluents are modularized and can therefore easily be imported into other modules if the need arises. Second, the modules are clearer now that a knowledge engineer must no longer consider whether fluents defined in separate branches are instances of the same global fluent or if they should be considered distinct.

#### 3.2.5.3 CoreCALMLib VerbNet Extensions

Knowledge bases CoreALMLib and CoreCALMLib differ in the fact that CoreCALMLib is compatible with the VerbNet lexicon. We organize CoreCALMLib so that every VerbNet class has a corresponding $\mathcal{ALM}$ module. For that we add VerbNet modules, which link information from VerbNet classes to the CoreALMLib knowledge base. Each VerbNet class module defines a sort for that verb class inheriting from an existing action sort in the CoreALMLib. The thematic roles from the VerbNet lexicon are connected to the general semantic roles already used in the CoreALMLib modules through state constraints. The new VerbNet theories are stored within the sub-library VN_class_library.

We design VN_class_library to mirror the class hierarchy in VerbNet. Subclasses in VerbNet also have their own modules which inherit from its VerbNet parent class. For example, the VerbNet class *run-51.3.2* has the subclasses *run-51.3.2-1, run-51.3.2-2*, and *run-51-3.2-2-1*. Therefore, the corresponding $\mathcal{ALM}$ theory is structured in the following manner:

```
theory t_run_51_3_2                                    (1)
  module m_run_51_3_2                                  (2)
    <implement module>                                 (3)
  module m_run_51_3_2_1                                (4)
    depends on run_51_3_2                              (5)
    <implement module>                                 (6)
  module m_run_51_3_2_2                                (7)
    depends on run_51_3_2                              (8)
    <implement module>                                 (9)
  module m_run_51_3_2_2_1                             (10)
    depends on run_51_3_2_2                           (11)
    <implement module>                                (12)
```

*Example 24: t_run_51_3_2 Theory Outline*

Prior to describing the process of creating the modules in VN_class_library, we have to introduce the notion of a training set. In Section 2.6, we reviewed a dataset called QA bAbI. By *training set (training data)* we understand 700 questions formed by extracting 100 questions from each QA bAbI task 1, 2, 3, 5, 6, 7, and 8. For instance, if to concatenate narratives in Example 17 and 18, we obtain a three question training set containing eight declarative sentences. The training set contained a total of 3580 declarative sentences.

We are now ready to explain the methodology of creating the modules in VN_class_library. First, we extract all VerbNet classes used in the training set. Then, we extend the CoreALMLib hierarchy to include new VerbNet modules. For every VerbNet class, we identify the components of CoreALMLib associated with this VerbNet class.

Our primary resource to identify the associated CoreALMLib modules was CoreALMLib's searching capabilities[2]. CoreALMLib provides searching over the library by linking WordNet senses to CoreALMLib actions classes and fluents. We identify the WordNet senses associated with the VerbNet classes and link the VerbNet modules to the action class recommended by CoreALMLib.

To exemplify the process of creating an $\mathcal{ALM}$ module for a VerbNet class, consider the VerbNet class *run-51.3.2-1*. The VerbNet class *run_51.3.2-1* is a subclass of the VerbNet class *run-51.3.2*, so we first define a theory named after the root class, *t_run_51_3_2*, to contain all modules inheriting from the root.

```
theory t_run_51_3_2                                                    (1)
```

*Example 25: t_run_51_3_2 Theory*

Next, we import the CoreALMLib action class associated with the root VerbNet class *run-51.3.2*. Searching CoreALMLib for a WordNet sense matching "run" returns no matches. In addition to verb "run", class *run-51.3.*2 contains verb "go". The WordNet sense *go#1* is defined as "change location; move, travel, or proceed, also metaphorically" and this definition captures this VerbNet class. CoreALMLib states that *go#1* corresponds to the *locomotion* action class in the *locomotion* module. Therefore, we import this module into the *t_run_51_3_2* theory:

```
import locomotion.locomotion from CoreALMLib                           (2)
```

*Example 26: t_run_51_3_2 Importing*

Then, we define the module for the base VerbNet class *run-51.3.2*, named *m_run_51_3_2*. The module depends on the *locomotion* CoreALMLib module. We

[2] Search capabilities can be found at https://ceclnx01.cec.miamioh.edu/~inclezd/coreALMlib/SearchByVerb.htm

declare a new “action” sort for class *run-51.3.2* extending from the *locomotion* action sort. Example 27 displays the result of this process.

```
module m_run_51_3_2                                    (3)
  depends on locomotion                                (4)
  sort declarations                                    (5)
    run_51_3_2 :: locomotion                           (6)
  axioms                                               (7)
    <axiom definitions>                                (8)
```

*Example 27: m_run_51_3_2 Module Outline*

A module for a VerbNet subclass inherits from its superclass’s module in the same manner that a VerbNet subclass inherits information, such as thematic roles, from their parent class. To illustrate this hierarchy, consider the *run-51.3.2-1* class. It is a subclass of *run-51.3.2* so we declare a new module named *m_run_51_3_2_1* in theory *t_run_51_3_2*. The module depends on the module of its VerbNet parent, *m_run_51_3_2*, and declares a new action sort *run_51_3_2_1* inheriting from its parent’s sort. Example 28 displays the *m_run_51_3_2_1* module.

```
module m_run_51_3_2_1                                  (9)
  depends on m_run_51_3_2                             (10)
  sort declarations                                   (12)
    run_51_3_2_1 :: run_51_3_2                        (12)
  axioms                                              (13)
    <axiom definitions>                               (14)
```

*Example 28: m_run_51_3_2_1 Module Outline*

Finally, we define the axioms in the VerbNet modules. We use state constraints to map VerbNet thematic roles to the attributes associated with the respective actions from the CoreALMLib. The state constraints were created manually by identifying the CoreALMLib attribute most similarly representing the thematic role through trial and error. For example, the VerbNet class *run-51.3.2* has the thematic roles *theme*, *initial_location*, *destination*, and *trajectory*. We identified the CoreALMLib action

attributes *agent*, *origin*, *destination*, and *path* to convey similar information as these thematic roles, respectively. Example 29 presents the resulting state constraints for the *m_run_51_3_2* module.

```
axioms                                                              (1)
  state constraints                                                 (2)
    agent(X, Y) if instance(X, run_51_3_2),                         (3)
                        vn_theme(X, Y).                             (4)
    origin(X, Y) if instance(X, run_51_3_2),                        (5)
                        vn_initial_location(X, Y).                  (6)
    destination(X, Y) if instance(X, run_51_3_2),                   (7)
                        vn_destination(X, Y).                       (8)
    path(X, Y) if instance(X, run_51_3_2),                          (9)
                        vn_trajectory(X, Y).                       (10)
```

*Example 29: m_run_51_3_2 Axioms*

This concludes the introduction of the *t_run_51_3_2* theory, which contains modules for the root VerbNet class *run-51.3.*2 and its subclass *run-51.3.2-1*. The full *t_run_51_3_2* theory is given in Appendix E.

We repeat this process to expand VN_class_library to all VerbNet classes identified in the training set. Figure 12 presents a snippet of the module hierarchy obtained within VN_class_library. The module *m_run_51_3_2* relies on knowledge represented in the *locomotion* CoreALMLib module. The oval nodes in the figure represent original CoreALMLib modules.

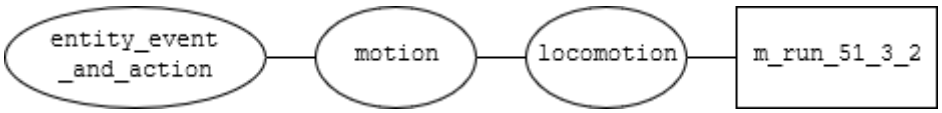


*Figure 12: Example Module Hierarchy*

Figure 13 presents another snippet of the hierarchy with modules for the VerbNet classes *escape-51.1, meander-47.7, get-13.5.1* and *give-13.1*. Like Figure 12, the oval nodes represent original CoreALMLib modules.

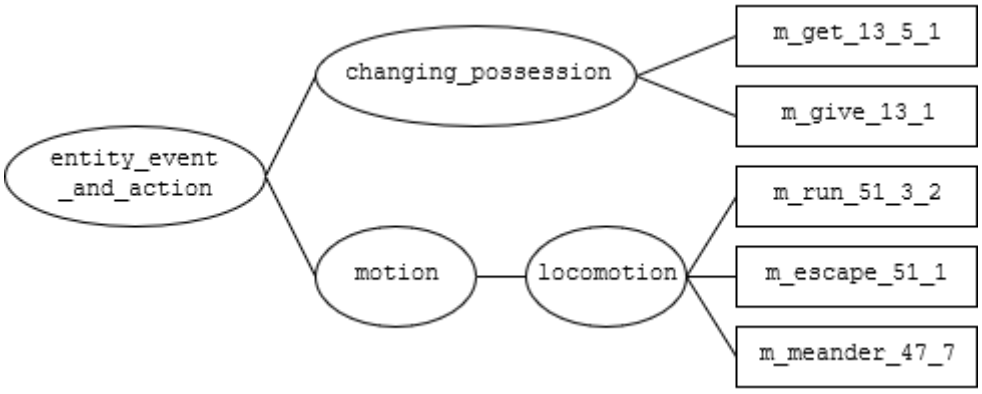


*Figure 13: Extended Example Module Hierarchy*

Related to the new VerbNet modules, we define additional attributes to the *actions* sort defined in CoreALMLib's root module, *entity_event_and_action*. The attributes match the names of the VerbNet thematic roles, with *vn_* prepended to the thematic role name. Example 30 displays a portion of the new attributes of *actions* that are added to Example 13 between lines (8) – (9).

```
actions :: events                                                    (1)
  attributes                                                         (2)
    vn_actor : entity -> booleans                                    (3)
    vn_destination : spatial_entity -> booleans                      (4)
    vn_source : spatial_entity -> booleans                           (5)
    vn_theme : entity -> booleans                                    (6)
    ...
```

*Example 30: Actions Attributes*

### 3.2.5.4 CoreCALMLib Axiom Changes

A final category of modifications made to CoreALMLib were changes to axioms about actions. We found gaps in the CoreALMLib knowledge base and we made axiom adjustments to represent the missing knowledge. To illustrate the case of missing

knowledge in the CoreALMLib, consider the *JSB Discourse.* When *John gets* the *ball*, it is important to know that the location of the *ball* is the same location as *John* after he picks up the ball. The verb *get* is defined by an action declared in the *changing_possession* module of CoreALMLib. We extend this module with the following axiom that encodes the desired knowledge in terms of fluents of CoreALMLib.

```
state constraints                                                    (1)
  location(A, Q) if held_by(A, B), location(B, Q),                   (2)
    instance(A, tangible_entity).                                    (3)
```

*Example 31: New State Constraints*

Appendix G lists all axiom adjustments made to the CoreALMLib to form the CoreCALMLib. The few number of axiom adjustments had significant effects to improve training results. This shows that the CoreALMLib provides a solid foundation of knowledge that can be adapted for a system's domain.

## 3.3 CALM Processing - $\mathcal{ALM}$ Model Generation and Interpretation

Once we have a narrative's $\mathcal{ALM}$ program importing relevant CoreCALMLib entries, we call the CALM system to calculate a model. Let file titled *JSB_Discourse.tp* contain the *JSB Discourse* $\mathcal{ALM}$ program. We can invoke the CALM system on this program with the command:

```
java -jar calm.jar JSB_Discourse.tp
```

CALM generates the model containing facts about the narrative.

Then we perform post-processing on the model to make the output easier for a human to read. The model states its facts using the entity identifiers from the DRS. For example, the model of the *JSB Discourse* contains a fact *loc_in(r1, r3, 1)*, where the

identifiers *r1* and *r2* correspond to *John* and *hallway*, respectively. Instead of the fact *loc_in(r1, r3, 1),* Text2ALM outputs the fact *loc_in(john, hallway, 1).*

## 3.4 Question Answer Processing

After a model is created, the system requires the following steps for QA capabilities:

1. Translate the question to a collection of fluents representing the desired information.
2. Query the model for the required fluents.
3. Derive an answer from the query's results.

These steps are performed by the Sphinx subsystem. The Sphinx system was developed to answer questions from the bAbI QA tests and then compare Text2ALM's answer with the expected answer.

The questions in the bAbI QA tasks use a set of specific syntactic formats. Therefore, Sphinx utilizes regular expressions to identify the question and details related to the question. We identified the following question formats:

1. Where is <entity>?
2. Where is the <entity>?
3. Where was the <entity> before the <location>?
4. What did <entity> give to <entity>?
5. Who <received/gave> the <entity>?
6. Who did <entity> give the <entity> to?
7. Is <entity> in the <location>?
8. How many objects is <entity> <carrying/holding>?
9. What is <entity> <carrying/holding>?
10. Who gave the <entity> to <entity>?

Once the Sphinx system matches the bAbI question with one of these formats, the question is translated to specific atoms to query in the model. The Sphinx system extracts the information from the model by finding the search atoms via regular expressions. The found atoms are processed to answer the question.

Consider a question of the form 8 as an example. Sphinx queries the model for the atom *held_by(<entity>, <>, <question time point>),* where <> matches arbitrary expressions. The system then counts found atoms. The count is given as the answer. If the question was instead of form 9, Sphinx searches the model for the same atoms *held_by(<entity>, <>, <question time point>)* and the answer is the list of entities matching to <>. Appendix G states the search atoms and outlines how the system derives an answer for all question formats.

## 4 TEXT2ALM EVALUATION AND RELATED WORK

We use Facebook AI Research's bAbI dataset (Weston et al., 2015) to evaluate system Text2ALM. Section 2.6 reviewed the structure of bAbI. This dataset enables us to compare Text2ALM applied to a benchmark QA problem with modern machine learning approaches and another answer set logic approach to this problem. The testing data of QA bAbI is composed of 1000 questions per its twenty tasks. We use all questions from tasks 1, 2, 3, 5, 6, 7, and 8 to evaluate the Text2ALM system. These tasks are selected because they contain action-based narratives that the system Text2ALM is designed for. Prior to presenting our test results, we discuss related work.

## 4.1 Related Work

Many modern QA systems predominately rely on machine learning techniques. However, recently there has been more work related to the design of QA systems combining advances of natural language processing and knowledge representation and reasoning, subfields of AI. The Text2ALM system is a representative of the latter approach. Other approaches include the work by (Clark, Dalvi, & Tandon, 2014) and (Mitra & Baral, 2016). Mitra and Baral (2016) use a provided training dataset of narratives, questions, and answers to learn the knowledge needed to answer similar questions. Their approach posted nearly perfect test results on the bAbI tasks. However, this approach doesn't scale well to narratives that utilize other action verbs which are not present in the training set, including synonymous verbs. For example, if their system is trained on bAbI training data that contains verb *travel* it will process the *JS Discourse* correctly. Yet, if we alter the *JS Discourse* by exchanging *travel* with a synonymous word *stroll*, their system will fail to perform any inference on this altered narrative. The Text2ALM system will process this narrative with no mistakes.

Another relevant QA approach is the work by Clark, Dalvi, and Tandon (2014). This approach uses VerbNet to build a rule base of preconditions and effects of actions utilizing the semantic annotations that VerbNet provides for its classes. In our work, we can view $\mathcal{ALM}$ modules associated with VerbNet classes as machine interpretable alternatives to these annotations. Clark et al. (2014) use the first and most basic action language STRIPS (Fikes & Nilsson, 1971) for inference. STRIPS allows more limited capabilities than $\mathcal{ALM}$ in modeling complex interactions between events and modeling such common sense property of fluents as *inertia axiom*. This axiom states that unless

there was a cause for a fluent to change its status, it keeps the same value across time. For instance, if we are aware that *John is in a hallway* in the initial situation and we never learn any new facts of *John* it is reasonable for us to believe that *John* remains in the *hallway* at the later time points. While this behavior is naturally expressed in $\mathcal{ALM}$, STRIPS is not designed for modeling such a commonsense axiom.

## 4.2 TEXT2ALM EVALUATION RESULTS

Table 3 displays the accuracy of the Text2ALM system by comparing the system's results with the state-of-the-art machine learning approach AM+NG+NL MemNN described by Weston et al. (2015) and an approach by Mitra and Baral (2016). Weston et al. (2015) compared results from 8 machine learning approaches, and the AM+NG+NL MemNN method performed best. We cannot compare our results with the methodology by Clark et al. (2014) because their system is not available and there are no results posted for these datasets. The results are given by providing the accuracy of the systems.

| bAbI Task | AM+NG+NL MemNN (Weston et al., 2015) | Inductive Rule Learning (Mitra & Baral, 2016) | Text2ALM |
|---|---|---|---|
| 1 – Single Supporting Facts | **100** | **100** | **100.0** |
| 2 – Two Supporting Facts | **100** | **100** | **100.0** |
| 3 – Three Supporting Facts | **100** | **100** | **100.0** |
| 5 – Three Argument Relations | 98 | **100** | 22.0 |
| 6 – Yes/No Questions | **100** | **100** | **100.0** |
| 7 – Counting | 85 | **100** | 96.1 |
| 8 – Lists/Sets | 91 | **100** | **100.0** |

*Table 3: Text2ALM bAbI Test Results*

Text2ALM matches the Memory Network approach by Weston et al. (Weston et al., 2015) at 100% accuracy in tasks 1, 2, 3, and 6 and performed above this methodology for tasks 7 and 8. This suggests that the methodology used to develop Text2ALM can be just as accurate, if not more so, than state-of-the-art machine learning methods. When compared to the methodology by Mitra and Baral (2015), the Text2ALM system matched the results for tasks 1, 2, 3, 6, and 8, but was outperformed in tasks 5 and 7.

The Text2ALM system also used a smaller training size to match the results. Recall that our training set comprised of 100 questions per QA bAbI task. In our understanding, Mitra and Baral used all 1000 training questions per task. For the AM+NG+NL MemNN approach, the researchers reported using training sets of varying sizes. Table 4 displays the number of questions in the training set for each task to achieve greater than 95%.

| bAbI Task | Number of Questions in the Training Set to Achieve 95% |
|---|---|
| 1 – Single Supporting Facts | 250 |
| 2 – Two Supporting Facts | 500 |
| 3 – Three Supporting Facts | 500 |
| 5 – Three Argument Relations | 1000 |
| 6 – Yes/No Questions | 500 |
| 7 – Counting | Failed to Achieve 95% |
| 8 – Lists/Sets | Failed to Achieve 95% |

*Table 4: Number of Questions in Training Sets for AM+NG+NL MemNN Approach*

Comprehensively, the Text2ALM system's results were comparable to the industry-leading results with one outlier, task 5. We investigated this task to identify causes and found that there was one phrase that the narratives regularly used that the Text2DRS system failed to represent correctly. The problem phrase was any sentence in the format of *"Entity1 handed the Object to Entity2."*, such as *"Fred handed the football to Bill."* The problem stemmed from a semantic parsing error by the LTH component in

the Text2DRS sub-system. The LTH system parsed the *"handed the Object to Entity2"* phrase as a single argument in the format of *"the Object belonging to Entity2"*, instead of two arguments, *"the Object"* and *"Entity2"*. This parse error prevented the Text2DRS system from adding crucial *eventArguments* to the DRS stating that *Entity2* plays the thematic role of *destination* or *beneficiary* in the phrase. Without these thematic roles, the Text2ALM system did not encode that possession of the object was passed from *Entity1* to *Entity2*. Since the same sentence structure and verb phrases are often repeated in the narratives in a bAbI task, the error occurred in many of the test cases.

# 5 Future Work

We conclude our work by listing future research directions in four areas:

- Expanding Text2ALM narrative processing capabilities
- Expanding Text2ALM QA ability
- Developing Text2ALM web interface
- Supporting additional reasoning tasks

## 5.1.1 Expanding Text2ALM Narrative Processing Capabilities

The Text2ALM system performed well in the basic domain comprised of the QA bAbI tasks. However, there are several areas of further research to explore to make the system handle more complex sentence structures and narratives.

The first research area to improve Text2ALM's narrative processing capabilities is mitigating the impact of semantic role labeling errors. Task 5 test results show that incorrect semantic role labeling can have a significantly negative impact on Text2ALM's ability to create a valid model for the narrative and answer questions. This problem stems from the system's dependence on semantic roles to map to VerbNet thematic roles in

SemLink. Further research on limiting this dependence or combining the results from several semantic role labelers could be impactful.

A second area of research to handle more advanced narratives is the topic of choosing the best matching PropBank role set in SemLink. SemLink provides mappings between PropBank arguments to VerbNet classes and VerbNet thematic roles per PropBank role set. However, SemLink often provides multiple VerbNet classes and mappings for a single PropBank role set. For example, PropBank role set *run.02* has mappings to both VerbNet class *run-51.3.2.3-1* an *meander-47.5.1-1* and the Text2DRS system chooses whichever one is listed first in SemLink. This straightforward solution was efficient for our testing, but it remains to be seen if a more complex solution would be required for larger systems.

A third topic of research in narrative processing is choosing the best sort for an entity in the $\mathcal{ALM}$ system description. Text2ALM declares new sorts for the narrative's entities based on the entities' VerbNet thematic roles in the narrative. This methodology worked well in our tests, but further testing in larger systems is required to prove the validity of this method. An alternative to sort identification is sort generalization by reducing the amount of entity sorts in the CoreCALMLib and using a generic sort for all entities in the narrative. This method eliminates requirement for sort identification but may reduce reasoning ability that CoreCALMLib is ability to perform because of the sort hierarchy of entities.

A final area of future work related to narratives is handling instances where no SemLink match exists. There are also instances were no matches exists for a PropBank role set in SemLink, which then inhibits Text2ALM's ability to support the action with

commonsense knowledge in CoreCALMLib. Further research on SemLink extensions or an alternative to SemLink would alleviate this restriction.

### 5.1.2 Expanding Text2ALM's Question Answering Ability

Text2ALM utilizes the fact that the questions in the bAbI QA Tasks follow specific formats and only certain questions are asked to create a QA subsystem that relies on regular expressions for each case. If this system is going to be used in additional QA fields, then further research is required on representing generic questions and answers in $\mathcal{ALM}$ domains. Additionally, our approach should be tested on other KRR datasets, such as the ProPara dataset (Mishra, Huang, Tandon, Yih, & Clark, 2018). Conducting tests on the ProPara dataset would enable us to compare the results of Text2ALM to the approach by Clark, Dalvi, and Tandon (2014).

### 5.1.3 Text2ALM Web Interface

System Text2ALM's current implementation is a Python project that executes via the command line. The long-term vision for Text2ALM is to convert the project to a web interface. Moving Text2ALM over to a web interface will improve its accessibility and impact.

### 5.1.4 Perform Backwards Induction Reasoning from the Model

An area of interest we discovered is that the derived CALM model may sometimes not contain atoms that could be argued as reasonable. For example, the if input narrative is "The monkey is in the tree. The monkey grabs the banana.", the CALM model will contain fluents that the monkey's location is the tree starting at time point 1, the monkey is holding the banana starting at time point 2, and the banana's location is the

tree starting at time point 2. However, the model would not contain the atom that the banana's location is the tree when the monkey grabs it. This may seem minor, but this restriction would inhibit the system from answering the question "Where was the banana when the monkey grabbed it?" This inability stems from the initial state described in the narrative's system description is empty and knowledge about the state is added after actions occur.

## 6 Conclusion

Lierler, Inclezan, and Gelfond (2017) outline a methodology for designing QA systems to make inferences based on complex interactions of events in narratives. Their process utilizes an action language $\mathcal{ALM}$ (Inclezan & Gelfond, 2015) and an extension of the VerbNet lexicon (Palmer, 2018; Schuler, 2005) to formalize knowledge. The VerbNet lexicon provides semantic information by mapping narrative events to VerbNet classes. The system performs commonsense reasoning task by expressing implicit knowledge with action languages.

To explore the feasibility of this methodology, we built the Text2ALM system to take an action-based narrative as input and output a model encoding the narrative's facts. Text2ALM accomplishes four key tasks. First, the Text2DRS system accepts a text file containing a narrative text and processes the text to output a DRS representation of the narrative. Then, the DRS is sent to the DRS2ALM sub-system to convert the information in the DRS to an $\mathcal{ALM}$ program containing a system description and history. After the $\mathcal{ALM}$ program is created, the CALM system uses the program's system description and history with commonsense knowledge represented in the CoreCALMLib to generate a model. Overall, these first three steps convert a narrative text to a model containing the

narrative's facts. The final task is translating a given question to search fluents representing the information we want to extract. The model is then queried for the search fluents to generate an answer.

We tested the system over Tasks 1, 2, 3, 5, 6, 7, and 8 from the bAbI QA tasks. Text2ALM matched the results of the state-of-the-art machine learning method (Weston et al., 2015) in Tasks 1, 2, 3, and 6, and outperformed this method in Tasks 7 and 8. Text2ALM also matched the results of another answer set programming approach (Mitra & Baral, 2016) in Tasks 1, 2, 3, 6, and 8, but did not perform as well in Tasks 5 and 7. However, we expect our approach to generalize to narratives of diverse lexicon.

## 7 Acknowledgements

First and foremost, I would first like to thank my Thesis Committee Chair Dr. Yuliya Lierler for her insights and guidance throughout this process. Her mentorship was vital during research, development, and writing. I would also like to thank Dr. Parvathi Chundi and Dr. Ryan Schuetzler for being members of my Thesis Committee. I value their helpfulness and feedback for this work. I am also grateful for fellow University of Nebraska Omaha researchers Nicholas Hippen, Brian Hodges, and Joseph Meyer for their support. Lastly, I appreciate the willingness of Daniela Inclezan, Michael Gelfond, Edward Wertz, and Yuanlin Zhang to answer my questions on their prior work and provide direction on this endeavor. I would not have been able to complete this Thesis without the support from all these people. Thank you.

# 8 REFERENCES

Baker, C. F., Fillmore, C. J., & Lowe, J. B. (1998). The Berkeley FrameNet Project. In *36th Annual Meeting of the Association for Computational Linguistics and 17th International Conference on Computational Linguistics* (Vol. 1, pp. 86–90). San Mateo, CA.

Barker, K., Porter, B., & Clark, P. (2001). A Library of Generic Concepts for Composing Knowledge Bases. *Proceedings of the 1st International Conference on Knowledge Capture - K-CAP*, 14–21. https://doi.org/10.1145/500742.500744

Bonial, C., Stowe, K., & Palmer, M. (2013). SemLink. University of Colorado. Retrieved from https://verbs.colorado.edu/semlink/

Clark, P., Dalvi, B., & Tandon, N. (2014). What Happened? Leveraging VerbNet to Predict the Effects of Actions in Procedural Text.

Fikes, R. E., & Nilsson, N. J. (1971). Strips: A new approach to the application of theorem proving to problem solving. *Artificial Intelligence*, *2*(3–4), 189–208. https://doi.org/10.1016/0004-3702(71)90010-5

Inclezan, D. (2016). CoreALMlib: An ALM library translated from the Component Library. *Theory and Practice of Logic Programming*, *16*(5–6), 800–816. https://doi.org/10.1017/S1471068416000363

Inclezan, D., & Gelfond, M. (2015). Modular Action Language ALM. *Theory and Practice of Logic Programming*, *16*(2), 189–235. https://doi.org/10.1017/S1471068415000095

Johansson, R., & Nugues, P. (2007). Language Technology at LTH. Lund: Lund University. Retrieved from http://nlp.cs.lth.se/

Kamp, H., & Reyle, U. (1993). *From Discourse to Logic* (Vol. 42). https://doi.org/10.1007/978-94-011-2066-1

Kočiský, T., Schwarz, J., Blunsom, P., Dyer, C., Hermann, K. M., Melis, G., & Grefenstette, E. (2017). The NarrativeQA Reading Comprehension Challenge. https://doi.org/10.1080/13506280444000193

Labutov, I., Yang, B., Prakash, A., & Azaria, A. (2018). Multi-Relational Question Answering from Narratives: Machine Reading and Reasoning in Simulated Worlds. *Proceedings of the 56th Annual Meeting of the Association for Computational Linguistics (Long Papers)*, 833–844. Retrieved from http://aclweb.org/anthology/P18-1077

Lierler, Y., Inclezan, D., & Gelfond, M. (2017). Action Languages and Question Answering. *IWCS 2017 - 12th International Conference on Computational Semantics-Short Papers*, (6).

Ling, G. (2018). *From Narrative Text to VerbNet-Based DRSes: System Text2DRS*. University of Nebraska at Omaha. Retrieved from https://www.unomaha.edu/college-of-information-science-and-

technology/natural-language-processing-and-knowledge-representation-lab/_files/papers/Text2Drses_system_description.pdf

Manning, C. D., Surdeanu, M., Bauer, J., Finkel, J., Bethard, S., & McClosky, D. (2014). The Stanford CoreNLP Natural Language Processing Toolkit. *Proceedings of 52nd Annual Meeting of the Association for Computational Linguistics: System Demonstrations*, 55–60. https://doi.org/10.3115/v1/P14-5010

Miller, G. A., Beckwith, R., Fellbaum, C., Gross, D., & Miller, K. (1990). WordNet: An On-line Lexical Database. *International Journal of Lexicography*, 245–244.

Mishra, B. D., Huang, L., Tandon, N., Yih, W., & Clark, P. (2018). Tracking State Changes in Procedural Text: A Challenge Dataset and Models for Process Paragraph Comprehension, (January). https://doi.org/10.18653/v1/N18-1144

Mitra, A., & Baral, C. (2015). Learning to Automatically Solve Logic Grid Puzzles, (September), 1023–1033.

Mitra, A., & Baral, C. (2016). Addressing a Question Answering Challenge by Combining Statistical Methods with Inductive Rule Learning and Reasoning, 2779–2785.

Palmer, M. (2018). VerbNet. Boulder: University of Colorado. Retrieved from https://verbs.colorado.edu/verb-index/vn3.3/

Palmer, M., Gildea, D., & Kingsbury, P. (2005). The Proposition Bank: An Annotated Corpus of Semantic Roles. *Computational Linguistics*, *31*(1), 71–106. https://doi.org/10.1162/0891201053630264

Schuler, K. K. (2005). *VerbNet: A Broad-Coverage, Comprehensive Verb Lexicon*. University of Pennsylvania. Retrieved from http://verbs.colorado.edu/~kipper/Papers/dissertation.pdf

Wertz, E., Chandrasekan, A., & Zhang, Y. (2018). CALM: a Compiler for Modular Action Language ALM.

Weston, J., Bordes, A., Chopra, S., Rush, A. M., Merri, B. Van, Joulin, A., & Mikolov, T. (2015). Towards AI-Complete Question Answering: A Set of Prerequisite Toy Tasks.

# 9 APPENDICES

## APPENDIX A - JS DISCOURSE $\mathcal{ALM}$ PROGRAM

```
system description JS_discourse                                    (1)
  theory JS_discourse_theory                                       (2)
    module JS_discourse_module                                     (3)
      sort declarations                                            (4)
        points, things :: universe                                (5)
        agents :: things                                           (6)
        move :: actions                                            (7)
          attributes                                               (8)
            actor : agents                                         (9)
            origin : points                                       (10)
            destination : points                                  (11)
      function declarations                                       (12)
        fluents                                                   (13)
          basic                                                   (14)
            loc_in : things -> points                             (15)
      axioms                                                      (16)
        dynamic causal laws                                       (17)
          occurs(X) causes loc_in(A) = D                          (18)
            if instance(X, move),                                 (19)
            actor(X) = A,                                         (20)
            destination(X) = D.                                   (21)
        executability conditions                                  (22)
          impossible occurs(X) if instance(X, move),              (23)
            actor(X) = A,                                         (24)
            loc_in(A) != origin(X).                               (25)
          impossible occurs(X) if instance(X, move),              (26)
            actor(X) = A,                                         (27)
            loc_in(A) = destination(X).                           (28)
  structure john_and_sandra                                       (29)
    instances                                                     (30)
      j, s in agents                                              (31)
      h in points                                                 (32)
      go(X,P) in move                                             (33)
        actor = X                                                 (34)
        destination = P                                           (35)
history                                                           (36)
  happened(go(j, h), 0).                                          (37)
  happened(go(s, h), 1).                                          (38)
```

## APPENDIX B - JS DISCOURSE COMPLETE $\mathcal{ALM}$ PROGRAM

```
system description JS_discourse                                    (1)
  theory JS_discourse_theory                                       (2)
    module JS_discourse_module                                     (3)
      sort declarations                                            (4)
        points, things :: universe                                 (5)
        agents :: things                                           (6)
        move :: actions                                            (7)
          attributes                                               (8)
            actor : agents                                         (9)
            origin : points                                       (10)
            destination : points                                  (11)
      function declarations                                        (12)
        fluents                                                    (13)
          basic                                                    (14)
            loc_in : things -> points                              (15)
      axioms                                                       (16)
        dynamic causal laws                                        (17)
          occurs(X) causes loc_in(A) = D                           (18)
            if instance(X, move),                                  (19)
            actor(X) = A,                                          (20)
            destination(X) = D.                                    (21)
        executability conditions                                   (22)
          impossible occurs(X) if instance(X, move),               (23)
            actor(X) = A,                                          (24)
            loc_in(A) != origin(X).                                (25)
          impossible occurs(X) if instance(X, move),               (26)
            actor(X) = A,                                          (27)
            loc_in(A) = destination(X).                            (28)
  structure john_and_sandra                                        (29)
    instances                                                      (30)
      j, s in agents                                               (31)
      h in points                                                  (32)
      go(X,P) in move                                              (33)
        actor = X                                                  (34)
        destination = P                                            (35)
temporal projection                                                (36)
max steps 3                                                        (37)
history                                                            (38)
  happened(go(j, h), 0).                                           (39)
  happened(go(s, h), 1).                                           (40)
```

## APPENDIX C – CORECALMLIB MOTION MODULE

```
module motion
  depends on entity_event_and_actions
  sort declarations
    move :: actions
  function declarations
    fluents
      basic
        location : spatial_entity * place -> booleans
        is_near : spatial_entity * spatial_entity -> booleans
        abuts : spatial_entity * spatial_entity -> booleans
        is_beside : spatial_entity * spatial_entity -> booleans
        is_opposite : spatial_entity * spatial_entity -> booleans
        is_above : spatial_entity * spatial_entity -> booleans
        is_below : spatial_entity * spatial_entity -> booleans
        is_behind : spatial_entity * spatial_entity -> booleans
        is_in_front_of : spatial_entity * spatial_entity -> booleans
        is_inside : spatial_entity * spatial_entity -> booleans
        is_over : spatial_entity * spatial_entity -> booleans
        is_under : spatial_entity * spatial_entity -> booleans
        is_on : spatial_entity * spatial_entity -> booleans
        encloses : spatial_entity * spatial_entity -> booleans
        is_along : spatial_entity * spatial_entity -> booleans
        is_at : spatial_entity * spatial_entity -> booleans
        is_outside : spatial_entity * spatial_entity -> booleans
        is_between : spatial_entity * spatial_entity -> booleans
        is_blocked : spatial_entity -> booleans
        is_held : tangible_entity -> booleans
        is_restrained : tangible_entity -> booleans
  axioms
    is_near(X, Y) if is_near(Y, X).
    -is_near(X, Y) if -is_near(Y, X).
    abuts(X, Y) if abuts(Y, X).
    -abuts(X, Y) if -abuts(Y, X).
    is_beside(X, Y) if is_beside(Y, X).
    -is_beside(X, Y) if -is_beside(Y, X).
    is_opposite(X, Y) if is_opposite(Y, X).
    -is_opposite(X, Y) if -is_opposite(Y, X).
    -is_above(Y, X) if is_above(X, Y), X != Y.
    -is_below(Y, X) if is_below(X, Y), X != Y.
    -is_behind(Y, X) if is_behind(X, Y), X != Y.
    -is_in_front_of(Y, X) if is_in_front_of(X, Y), X != Y.
    -is_inside(Y, X) if is_inside(X, Y), X != Y.
    -is_over(Y, X) if is_over(X, Y), X != Y.
```

```
-is_under(Y, X) if is_under(X, Y), X != Y.
-is_on(Y, X) if is_on(X, Y), X != Y.
-encloses(Y, X) if encloses(X, Y), X != Y.
false if instance(X, move),
         -defined_object(X).
false if instance(X, move),
         object(X, O),
         -instance(O, tangible_entity).
false if instance(X, move),
         origin(X, Or),
         -instance(Or, spatial_entity).
false if instance(X, move),
         destination(X, D),
         -instance(D, spatial_entity).
false if instance(X, move),
         away_from(X, Aw),
         -instance(Aw, spatial_entity).
false if instance(X, move),
         toward(X, T),
         -instance(T, spatial_entity).
false if instance(X, move),
         path(X, P),
         -instance(P, spatial_entity).
false if instance(X, move),
         object(X, Y),
         origin(X, Y).
false if instance(X, move),
         object(X, Y),
         destination(X, Y).
false if instance(X, move),
         object(X, Y),
         away_from(X, Y).
false if instance(X, move),
         object(X, Y),
         toward(X, Y).
false if instance(X, move),
         object(X, Y),
         path(X, Y).
false if instance(X, move),
         destination(X, D),
         -instance(D, place),
         -instance(D, tangible_entity).
-origin(X, Or1) if instance(X, move),
                   origin(X, Or),
                   Or1 != Or.
```

```
-destination(X, D1) if instance(X, move),
                       destination(X, D),
                       D1 != D.
occurs(X) causes location(O, D) if instance(X, move),
                                   object(X, O),
                                   destination(X, D),
                                   instance(D, place).
occurs(X) causes location(O, P) if instance(X, move),
                                   object(X, O),
                                   destination(X, D),
                                   -instance(D, tangible_entity),
                                   location(D, P).
occurs(X) causes is_near(O, Y) if instance(X, move),
                                   object(X, O),
                                   destination(X, D),
                                   is_near(D, Y),
                                   instance(D, place).
occurs(X) causes is_above(O, Y) if instance(X, move),
                                   object(X, O),
                                   destination(X, D),
                                   is_above(D, Y),
                                   instance(D, place).
occurs(X) causes is_below(O, Y) if instance(X, move),
                                   object(X, O),
                                   destination(X, D),
                                   is_below(D, Y),
                                   instance(D, place).
occurs(X) causes is_along(O, Y) if instance(X, move),
                                   object(X, O),
                                   destination(X, D),
                                   is_along(D, Y),
                                   instance(D, place).
occurs(X) causes is_at(O, Y) if instance(X, move),
                                   object(X, O),
                                   destination(X, D),
                                   is_at(D, Y),
                                   instance(D, place).
occurs(X) causes is_beside(O, Y) if instance(X, move),
                                   object(X, O),
                                   destination(X, D),
                                   is_beside(D, Y),
                                   instance(D, place).
occurs(X) causes is_along(O, Y) if instance(X, move),
                                   object(X, O),
                                   destination(X, D),
```

```
                                    is_along(D, Y),
                                    instance(D, place).
occurs(X) causes is_between(O, Y) if instance(X, move),
                                     object(X, O),
                                     destination(X, D),
                                     is_between(D, Y),
                                     instance(D, place).
occurs(X) causes is_behind(O, Y) if instance(X, move),
                                     object(X, O),
                                     destination(X, D),
                                     is_behind(D, Y),
                                     instance(D, place).
occurs(X) causes is_in_front_of(O, Y) if instance(X, move),
                                     object(X, O),
                                     destination(X, D),
                                     is_in_front_of(D, Y),
                                     instance(D, place).
occurs(X) causes is_inside(O, Y) if instance(X, move),
                                     object(X, O),
                                     destination(X, D),
                                     is_inside(D, Y),
                                     instance(D, place).
occurs(X) causes is_on(O, Y) if instance(X, move),
                                     object(X, O),
                                     destination(X, D),
                                     is_on(D, Y),
                                     instance(D, place).
occurs(X) causes is_opposite(O, Y) if instance(X, move),
                                     object(X, O),
                                     destination(X, D),
                                     is_opposite(D, Y),
                                     instance(D, place).
occurs(X) causes is_outside(O, Y) if instance(X, move),
                                     object(X, O),
                                     destination(X, D),
                                     is_outside(D, Y),
                                     instance(D, place).
occurs(X) causes -encloses(O, Y) if instance(X, move),
                                     object(X, O),
                                     destination(X, D),
                                     -encloses(D, Y),
                                     instance(D, place).
occurs(X) causes is_over(O, Y) if instance(X, move),
                                     object(X, O),
                                     destination(X, D),
```

```
                                    is_over(D, Y),
                                    instance(D, place).
occurs(X) causes is_under(O, Y) if instance(X, move),
                                    object(X, O),
                                    destination(X, D),
                                    is_under(D, Y),
                                    instance(D, place).
occurs(X) causes -location(O, L) if instance(X, move),
                                    object(X, O),
                                    location(O, L).
occurs(X) causes -is_near(O, Y) if instance(X, move),
                                    object(X, O),
                                    is_near(O, Y),
                                    destination(X, D),
                                    -is_near(D, Y),
                                    instance(D, place).
occurs(X) causes -abuts(O, Y) if instance(X, move),
                                    object(X, O),
                                    abuts(O, Y),
                                    destination(X, D),
                                    -abuts(D, Y),
                                    instance(D, place).
occurs(X) causes -is_above(O, Y) if instance(X, move),
                                    object(X, O),
                                    is_above(O, Y),
                                    destination(X, D),
                                    -is_above(D, Y),
                                    instance(D, place).
occurs(X) causes -is_below(O, Y) if instance(X, move),
                                    object(X, O),
                                    is_below(O, Y),
                                    destination(X, D),
                                    -is_below(D, Y),
                                    instance(D, place).
occurs(X) causes -is_along(O, Y) if instance(X, move),
                                    object(X, O),
                                    is_along(O, Y),
                                    destination(X, D),
                                    -is_along(D, Y),
                                    instance(D, place).
occurs(X) causes -is_at(O, Y) if instance(X, move),
                                    object(X, O),
                                    is_at(O, Y),
                                    destination(X, D),
                                    -is_at(D, Y),
```

```
                                    instance(D, place).
occurs(X) causes -is_between(O, Y) if instance(X, move),
                                    object(X, O),
                                    is_between(O, Y),
                                    destination(X, D),
                                    -is_between(D, Y),
                                    instance(D, place).
occurs(X) causes -is_behind(O, Y) if instance(X, move),
                                    object(X, O),
                                    is_behind(O, Y),
                                    destination(X, D),
                                    -is_behind(D, Y),
                                    instance(D, place).
occurs(X) causes -is_in_front_of(O, Y) if instance(X, move),
                                    object(X, O),
                                    is_in_front_of(O, Y),
                                    destination(X, D),
                                    -is_in_front_of(D, Y),
                                    instance(D, place).
occurs(X) causes -is_inside(O, Y) if instance(X, move),
                                    object(X, O),
                                    is_inside(O, Y),
                                    destination(X, D),
                                    -is_inside(D, Y),
                                    instance(D, place).
occurs(X) causes -encloses(O, Y) if instance(X, move),
                                    object(X, O),
                                    encloses(O, Y),
                                    destination(X, D),
                                    -encloses(D, Y),
                                    instance(D, place).
occurs(X) causes -is_on(O, Y) if instance(X, move),
                                    object(X, O),
                                    is_on(O, Y),
                                    destination(X, D),
                                    -is_on(D, Y),
                                    instance(D, place).
occurs(X) causes -encloses(O, Y) if instance(X, move),
                                    object(X, O),
                                    encloses(O, Y),
                                    destination(X, D),
                                    -encloses(D, Y),
                                    instance(D, place).
occurs(X) causes -is_opposite(O, Y) if instance(X, move),
                                    object(X, O),
```

```
                                    is_opposite(O, Y),
                                    destination(X, D),
                                    -is_opposite(D, Y),
                                    instance(D, place).
occurs(X) causes -is_outside(O, Y) if instance(X, move),
                                    object(X, O),
                                    is_outside(O, Y),
                                    destination(X, D),
                                    -is_outside(D, Y),
                                    instance(D, place).
occurs(X) causes -is_over(O, Y) if instance(X, move),
                                    object(X, O),
                                    is_over(O, Y),
                                    destination(X, D),
                                    -is_over(D, Y),
                                    instance(D, place).
occurs(X) causes -is_under(O, Y) if instance(X, move),
                                    object(X, O),
                                    is_under(O, Y),
                                    destination(X, D),
                                    -is_under(D, Y),
                                    instance(D, place).
impossible occurs(X) if instance(X, move),
                        object(X, O),
                        origin(X, Or),
                        instance(X, place),
                        -location(O, Or).
impossible occurs(X) if instance(X, move),
                        object(X, O),
                        origin(X, Or),
                        instance(Or, tangible_entity),
                        location(Or, P),
                        -location(O, P).
impossible occurs(X) if instance(X, move),
                        object(X, O),
                        is_held(O),
                        -defined_agent(X).
impossible occurs(X) if instance(X, move),
                        object(X, O),
                        is_held(O),
                        agent(X, A),
                        -held_by(O, A).
impossible occurs(X) if instance(X, move),
                        object(X, O),
                        is_restrained(O).
```

```
impossible occurs(X) if instance(X, move),
                        path(X, P),
                        is_blocked(P).
```

## APPENDIX D – JSB DISCOURSE DRS

```
% r1, r2, r3, r4, e1, e2, e3                                    (1)
% ================================================              (2)
entity(r1). entity(r2). entity(r3). entity(r4).                 (3)

property(r1, "John"). property(r2, "hallway").                  (4)
property(r3, "Sandra"). property(r4, "ball").                   (5)

event(e1).                                                      (6)
event(e2).                                                      (7)
event(e3).                                                      (8)

eventType(e1, "51.3.2-1"). eventType(e2, "51.3.2-1").           (9)
eventType(e3, "13.5.1-1")                                      (10)

eventTime(e1, 0). eventTime(e2, 1). eventTime(e3, 2).          (11)

eventArgument(e1, "Theme", r1).                                (12)
eventArgument(e1, "Destination", r2).                          (13)
eventArgument(e2, "Theme", r3).                                (14)
eventArgument(e2, "Destination", r2).                          (15)
eventArgument(e3, "Agent", r1).                                (16)
eventArgument(e3, "Theme", r4).                                (17)
```

## APPENDIX E - VN_CLASS_LIB T_RUN_51_3_2 THEORY

```
theory t_run_51_3_2
  import locomotion.locomotion from CoreALMLib
  module m_run_51_3_2
    depends on locomotion
    sort declarations
      run_51_3_2 :: locomotion
    axioms
      state constraints
        agent(X, Y) if instance(X, run_51_3_2),
                       vn_theme(X,Y).
        origin(X, Y) if instance(X, run_51_3_2),
                       vn_initial_location(X,Y).
        destination(X, Y) if instance(X, run_51_3_2),
                       vn_destination(X,Y).
        path(X, Y) if instance(X, run_51_3_2),
                       vn_trajectory(X,Y).
  module m_run_51_3_2_1
    depends on m_run_51_3_2
    sort declarations
      run_51_3_2_1 :: run_51_3_2
    axioms
      state constraints
        agent(X, Y) if instance(X, run_51_3_2_1),
                       vn_theme(X,Y).
        origin(X, Y) if instance(X, run_51_3_2_1),
                       vn_initial_location(X,Y).
        destination(X, Y) if instance(X, run_51_3_2_1),
                       vn_destination(X,Y).
        path(X, Y) if instance(X, run_51_3_2_1),
                       vn_trajectory(X,Y).
  module m_run_51_3_2_2
    depends on m_run_51_3_2
    sort declarations
      run_51_3_2_2 :: run_51_3_2
    axioms
      state constraints
        agent(X, Y) if instance(X, run_51_3_2_2),
                       vn_theme(X,Y).
        origin(X, Y) if instance(X, run_51_3_2_2),
                       vn_initial_location(X,Y).
        destination(X, Y) if instance(X, run_51_3_2_2),
                       vn_destination(X,Y).
        path(X, Y) if instance(X, run_51_3_2_2),
```

```
                    vn_trajectory(X,Y).
      destination(X, Y) if instance(X, run_51_3_2_2),
                    vn_result(X,Y).
      origin(X, Y) if instance(X, run_51_3_2_2),
                    vn_source(X,Y).
module m_run_51_3_2_2_1
  depends on m_run_51_3_2_2
  sort declarations
    run_51_3_2_2_1 :: run_51_3_2_2
  axioms
    state constraints
      agent(X, Y) if instance(X, run_51_3_2_2_1),
                    vn_theme(X,Y).
      origin(X, Y) if instance(X, run_51_3_2_2_1),
                    vn_initial_location(X,Y).
      destination(X, Y) if instance(X, run_51_3_2_2_1),
                    vn_destination(X,Y).
      path(X, Y) if instance(X, run_51_3_2_2_1),
                    vn_trajectory(X,Y).
      destination(X, Y) if instance(X, run_51_3_2_2_1),
                    vn_result(X,Y).
      origin(X, Y) if instance(X, run_51_3_2_2_1),
                    vn_source(X,Y).
```

# APPENDIX F - CORECALMLIB AXIOM ADJUSTMENTS

## NEW AXIOMS:

**Module:** location_fluents
**Axiom(s):**

```
-location(X, Y) if location(X, Z), Y != Z.
```

**Module:** changing_possession
**Axiom(s):**

```
possesses(B, A) if held_by(A, B).
-possesses(B, A) if -held_by(A, B).
held_by(B, A) if instance(B, tangible_entity), possesses(A, B).
-held_by(B, A) if instance(B, tangible_entity), -possesses(A, B).
location(A, P) if held_by(A, B), location(B, P),
  instance(A, tangible_entity).
-location(A, P) if held_by(A, B), -location(B, P),
  instance(A, tangible_entity).
impossible occurs(X) if instance(X, transfer), object(X, O),
  recipient(X, R), location(R, A), location(O, B), A != B.
```

**Module:** letting_go_and_taking_hold
**Axiom(s):**

```
impossible occurs(X) if instance(X, take_hold), object(X, O),
  agent(X, A), location(A, Q), location(O, R), Q != R.
```

**Module:** motion
**Axiom(s):**
Convert uses of `place` to `spatial_entity`.

## ALTERED AXIOMS:

**Module:** changing_possession
**Original Axiom(s):**

```
occurs(X) causes -possesses(R, O) if instance(X, transfer),
  recipient(X, R), object(X, O).
```

**New Axiom(s):**

```
occurs(X) causes possesses(R, O) if instance(X, transfer),
  recipient(X, R), object(X, O).
```

## APPENDIX G - BABI QUESTION TO TEXT2ALM ANSWER PROCESS

**Note:** The question_time_point is already known and can be at any point in the narrative.
**Question Format:** Where is <entity>?
**Atom(s) to Query:** location(<entity>, <>, <question_time_point>).
**Answer:** The variable that matches to <>.

**Question Format:** Where is the <entity>?
**Atom(s) to Query:** location(<entity>, <>, <question_time_point>).
**Answer:** The variable that matches to <>.

**Question Format:** Where was the <entity> before the <location>?
**Atom(s) to Query:** location(<entity>, <location>, *) and location(<entity>, <>, *).
**Answer:** Extract all atoms that match. Sort the atoms descending by time point. Find the first instance that matches location(<entity>, <location>, *) and traverse the list of matches until the location no longer matches <location>. This must be the location of the entity before <location>.

**Question Format:** What did <entity1> give to <entity2>?
**Atom(s) to Query:** event_object(<event>, <>), event_agent(*, <entity1>), and event_recipient(*, <entity2>).
**Answer:** Find all matches to atoms 2 and 3. Sort them in descending order by event number. Then find the first match where the desired event_agent and event_recipient are associated with the same event. This gives us the event number we want. Then find the most recent match for atom 1 using the event number in the <event> field. The answer is the variable that matches to <>.

**Question Format:** Who <received/gave> the <entity>?
If Received:
**Atom(s) to Query:** event_object(*, <entity>), event_recipient(<event>, <>).
**Answer:** Find all matches to atom 1. Sort them in descending order by event number. Then plug the event numbers in to the <event> field in the second atom and continue through all event numbers until a match is found. The answer is the first variable that matches to <>.
If Gave:
**Atom(s) to Query:** event_object(*, <entity>), event_agent(<event>, <>).
**Answer:** Find all matches to atom 1. Sort them in descending order by event number. Then plug the event numbers in to the <event> field in the second atom and continue through all event numbers until a match is found. The answer is the first variable that matches to <>.

**Question Format:** Who did <entity1> give the <entity2> to?
**Atom(s) to Query:** event_object(*, <entity2>), event_agent(*, <entity1>), and event_recipient(<event>, <>).
**Answer:** Find all matches to atoms 1 and 2. Sort them in descending order by event number. Then find the first match where the desired event_object and event_agent are associated with the same event. This gives us the event number we want. Then find the most recent match for atom 3 using the event number in the <event> field. The answer is the variable that matches to <>.

**Question Format:** Is <entity> in the <location>?
**Atom(s) to Query:** location(<entity>, <location>, <question_time_point>).
**Answer:** Yes if there is a match. No if there is no match.

**Question Format:** How many objects is <entity> <carrying/holding>?
**Atom(s) to Query:** held_by(<>, <entity>, <question_time_point>).
**Answer:** Count the number of <> matches.

**Question Format:** What is <entity> <carrying/holding>?
**Atom(s) to Query:** held_by(<>, <entity>, <question_time_point>).
**Answer:** List the matches to <>.

**Question Format:** Who gave the <entity1> to <entity2>?
**Atom(s) to Query:** event_object(*, <entity1>), event_agent(<event>, <>), and event_recipient(*, <entity2>).
**Answer:** Find all matches to atoms 1 and 3. Sort them in descending order by event number. Then find the first match where the desired event_object and event_recipient are associated with the same event. This gives us the event number we want. Then find the most recent match for atom 3 using the event number in the <event> field. The answer is the variable that matches to <>.