Computer Science and Artificial Intelligence Laboratory

Technical Report

MIT-CSAIL-TR-2014-005 April 9, 2014

Moebius Language Reference, Version 1.2

Gary C. Borchardt

massachusetts institute of technology, cambridge, ma 02139 usa — www.csail.mit.edu

Möbius

Language Reference

Version 1.2

Gary C. Borchardt

MIT Computer Science and Artificial Intelligence Laboratory

1. Introduction

Möbius is a representation and interface language based on a subset of English. It is designed for use as a means of encoding information and as a means of conveying information between software components and other software components, between software components and humans, and between data repositories and their users—human or machine.

Central to the design of Möbius is the view that language and representation are one and the same thing—the illusive two sides of the Möbius strip.

Following is a sample Möbius expression:

```
travel(
  subject:vehicle "V507", adverb:named_heading north, on:road "Colton Ave",
  from:time "2007-09-27T10:08:45.418", to:time "2007-09-27T10:09:00.612").
```

This expression illustrates several aspects of Möbius:

- It can be used to capture simple English.
- It can include basic syntactic information.
- It can include basic semantic information.

In addition, the following are all acceptable Möbius expressions:

```
(blue, green, brown, black)          (:name "John Hill", :age "21")

f(g(x), h(x, y))                  =(x, +("2.0 m", *("1.5 m/s", "3.4 s")))

"V507 travels north on Colton Ave from 10:08:45.418 to 10:09:00.612."
```

As these examples illustrate, Möbius can also be used in a simplified manner to encode lists, records, function applications, mathematical expressions and equations, strings, and other quantities.

Möbius also has the ability to express patterns for matching. Following is another Möbius expression:

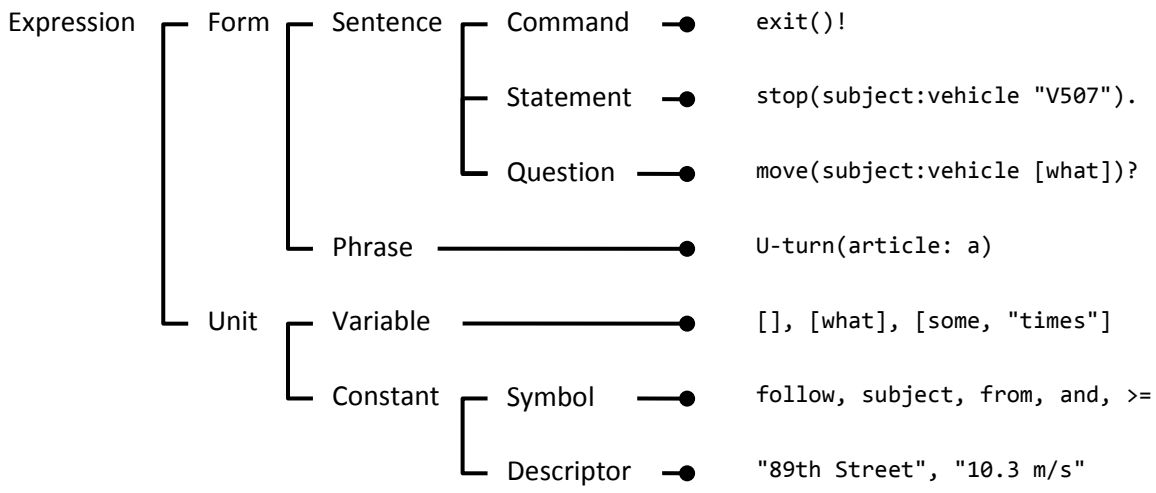
```
travel(
  subject:vehicle [which, those, "vehicles"], adverb:named_heading north,
  on:road [the, "major roads"], from:time [>=("2007-09-27T10:00:00")],
  to:time [<("2007-09-27T11:00:00")])?
```

In this expression, the portions in square brackets are matching variables, including some more programming-language-like rather than English-like constructs to specify how these variables can be matched to other Möbius expressions.

Möbius was initially developed in 2006–2007 and has been in use and gradual refinement since that time. Sections 5, 6 and 7 of this report describe three applications of Möbius to date. One application of the language has been to provide support for distributed, collaborative interpretation of English requests. This application is also described in [Katz *et al.*, 2007]. A second application has been that of English "wrapping" of database tables to provide support for language-based entry and retrieval of data, reasoning, and question answering. A third application has been to depict timestamped information and models of "what happens" during events within the IMPACT reasoning system, using the *transition space* representation described in [Borchardt, 1992] and [Borchardt, 1994]. In this application, Möbius is used to encode *relative attribute value expressions* (or RAVEs), which serve as the basic building blocks for constructing transition space specifications.

2. Language Syntax

Möbius encodes information in the form of Expressions. The following examples illustrate the hierarchy of Expression types:



Forms have an optional Function (`exit` , `stop` , `move` and `U-turn` in the above examples) and zero or more Arguments, each of which has an optional Label (`subject:vehicle` and `article:` in the above examples) and a mandatory Value (`"V507"` , `[what]` and `a` in the above examples). Labels have either or both a Syntactic Relation (`subject` and `article` in the above examples) and a Semantic Category (`vehicle` in the above examples); several Arguments in a

Form may have the same Label. Variables contain zero or more Tags (`what` , `some` and `"times"` in the above examples).

More precisely, Möbius Expressions adhere to the following syntax, where < … > indicates a defined quantity, … | … indicates alternatives, { … }~ indicates zero or one occurrence, and { … }* indicates zero or more occurrences:

<Expression> ::= <Form> | <Unit>
<Form> ::= <Sentence> | <Phrase>
<Sentence> ::= <Command> | <Statement> | <Question>
<Command> ::= {<Function>}~({<Argument>{, <Argument>}*}~)!
<Statement> ::= {<Function>}~({<Argument>{, <Argument>}*}~).
<Question> ::= {<Function>}~({<Argument>{, <Argument>}*}~)?
<Phrase> ::= {<Function>}~({<Argument>{, <Argument>}*}~)
<Function> ::= <Unit>
<Argument> ::= {<Label>}~ <Value>
<Label> ::= <Syntactic_Relation>: | :<Semantic_Category> |
<Syntactic_Relation>:<Semantic_Category>
<Syntactic_Relation> ::= <Unit>
<Semantic_Category> ::= <Unit>
<Value> ::= <Expression>
<Unit> ::= <Variable> | <Constant>
<Variable> ::= [{<Tag>{, <Tag>}*}~]
<Tag> ::= <Expression>
<Constant> ::= <Symbol> | <Descriptor>
<Symbol> ::= *(one or more characters—alphanumeric or in `~@#$%^&*-_=+\|'<>/)*
<Descriptor> ::= "*(zero or more characters with " and \ escaped as \" and \\)*"

3. Language Elements

3.1 Descriptors

Descriptors are used to depict quantities that may appear in unlimited variations within an application: numbers, addresses, times, names, coordinate locations and so forth, including arbitrary character strings. In many situations, Descriptors and other Möbius Expressions are intended to be aligned with language, and in these cases, the character sequence of a Descriptor should be “readable” in the sense that its meaning is readily apparent from inspection and its contents would not appear out of place if inserted directly into a natural language statement. In addition, some usage conventions are adopted to simplify the operation of code fragments that extract information from Descriptors or compare Descriptors

to other Descriptors. Times, for example, are typically depicted as a combined date and time of day expression using the ISO 8601:2004 “extended format” in either the “complete representation” (possibly extended to specify milliseconds), or an abbreviation of the complete representation to achieve reduced granularity by omitting lower-order components. Following are examples of times depicted in this manner:

```
"2014-02-05T21:06:13.548"
"2014-02-05T21:06:13"
"2014-02-05T21:06"
"2014-02-05T21"
"2014-02-05"
"2014-02"
"2014"
```

Numerical quantities with units are typically depicted using fixed prefixes and/or suffixes, as, for example:

```
"$220.45"
"$0.99"

"34.77 m/s"
"-5 m/s"
```

As well, Descriptors can be used to depict free natural language expressions (generated, or yet-to-be-parsed), or, as the need arises, natural language intermixed with Möbius, as illustrated in the following examples:

```
"How fast is vehicle 509 going at 10:20:08?"
"At 2007-09-27T10:20:08, the speed of V509 is [what]?"

"The most recent speed of V509 at 10:20:08 is 30.5 m/s."
```

3.2 Symbols

Symbols complement Descriptors by depicting quantities that are limited within an application, such as might be defined, instance by instance, in a lexicon or hierarchy. One use of symbols is to depict natural language tokens: nouns, verbs, adjectives and so forth, as in the following examples:

```
vehicle, road, named_heading, U-turn
travel, meet, follow, stop
short, long, medium-length
quickly, slowly
from, to, between
and, or
the, a
```

Another use of Symbols is to name a part of speech

```
noun, verb, adjective, adverb, preposition, conjunction, pronoun, determiner,
article, interjection
```

or a verb argument relationship

```
subject, object, indirect_object
```

as a means of filling the Syntactic Relation slot of a Möbius Label.

Also, Symbols are used to characterize restrictions within Möbius Variables, and in this usage, the Symbols conform more to the style of programming language tokens:

```
<, <=, =, <>, >=, >
v=, ^=, @=, ~
```

As is the case with Descriptors, some usage conventions are adopted to simplify the operation of code segments that manipulate Symbols. One convention is to use the underscore character _ in place of a space character for multi-word natural language tokens, as in the above examples `named_heading` and `indirect_object`. Hyphens and capitalization are used where they would normally appear in natural language tokens, as in the examples `medium-length` and `U-turn`.

3.3 Variables

Variables are used for pattern matching in Möbius. The simplest Variable is `[]`, containing no Tags. This Variable will match any Möbius Expression. A range of Tags and Tag varieties exist, several of which are illustrated in the following examples:

```
[what, "times", >=("2011-06-09T12:00:00")]
[("122 St", "122nd St", "122nd Street")]
[those, "vehicles"]
[something, specified]
[some, literal, "expression"]
```

The following paragraphs describe specific functions associated with particular Tags and varieties of Tags. Within a particular application of Möbius, there may be additional varieties of Tags that are introduced or varieties listed here that are excluded.

`what`, `which`, `some`, `something`. A *reported* Variable includes the Tag `what` or `which` (used interchangeably) and is inserted within a Question, Statement or Command to indicate that the result of a matching operation concerning that Question, Statement or

Command should be presented to the requester as simply the sub-Expression matching the reported Variable, not the entire Expression matched to the Question, Statement or Command. Thus, the response to a request `move(subject:vehicle [which])?` might be one or more vehicles such as `"V246"`, rather than a set of Expressions such as `move(subject:vehicle "V246")`. An *unreported* Variable does not include the Tag `what` or `which` and is used in all other Variable contexts. Unreported Variables may optionally include a non-functioning Tag `some` or `something` for readability.

`"times"`, `"vehicles"`, `"expression"`, and other Descriptors. A *consistent* Variable includes one or more Descriptors as Tags. These Tags serve as names or labels for the Variables. Within a pattern Expression or set of related pattern Expressions that are to be associated with a specific match, all Variables that share a common Descriptor Tag must be mapped to the same value. A *free* Variable contains no Tags that are Descriptors and is not constrained in this manner.

`>=("2011-06-09T12:00:00")`, `("122 St", "122nd St", "122nd Street")` and other Phrases. A *restricted* Variable includes one or more Phrases as Tags. These Phrases limit the range of values that may be matched to the Variable. A one-argument Phrase Tag with a Function `<`, `<=`, `=`, `<>`, `>=` or `>` is treated as an arithmetic restriction if its argument can be interpreted as a number or time and a lexicographic restriction otherwise. For example, the restriction `<("8.0 m/s")` will limit matching values to be Descriptors that indicate a speed less than 8.0 meters per second (e.g., `"0.25 m/s"`), whereas a restriction `<("N")` will limit matching values to be Descriptors that lexicographically precede `"N"` (e.g., `"Chicago Ave"`). A parallel set of restrictions with Function `v`, `v=`, `=`, `v^`, `^=` or `^` operates on a generalization/specialization hierarchy of Symbols and Descriptors. For example, `v=(vehicle)` will match any Symbol or Descriptor that is equal to or a specialization of the Symbol `vehicle`. A Phrase such as `("122 St", "122nd St", "122nd Street")`, with no Function and an arbitrary number of Arguments, specifies that the indicated Expressions are the only values that may match this Variable.

`that`, `those`, `the`. A *referential* Variable includes a Tag `that`, `those` or `the`. `that` and `those` are used interchangeably to constrain a consistent Variable to match only those values most recently matched in previous matching cycles by Variables containing any of its Descriptor Tags. Thus, a Variable `[those, "vehicles"]` would be constrained to match no values other than the values most recently matched by a Variable with Descriptor Tag `"vehicles"` in previous matching cycles. The Tag `the` is similar, but constrains a Variable to match only values that have been explicitly “assigned” to the Descriptor Tag or Tags for that Variable—regardless of whether the most recent

matches involving Variables with those Descriptor Tags may have resulted in subsets of the assigned sets of values in previous match cycles. A *non-referential* Variable does not contain a Tag `that`, `those` or `the` and is unconstrained by previous match cycles or assigned value sets for Descriptor Tags.

`specified`, `unspecified`. A *specified* Variable includes the Tag `specified` and can only be matched to a quantity that has a finite, listable set of possible match values in the current match cycle. For example, a specified Variable might be matched to a Form or Constant, or it might be matched to another Variable that currently has three possible match values. An *unspecified* Variable includes the Tag `unspecified` and can only be matched to a quantity that does not have a finite, listable set of possible match values in the current match cycle. For example, an unspecified Variable might be matched to another Variable that is restricted to match Descriptors in a lexicographic range, but has not yet been constrained to match a particular, listable subset of these values. A Variable that is *neither specified nor unspecified* contains neither the Tag `specified` nor the Tag `unspecified` and is not constrained in these ways. The Tags `specified` and `unspecified` are typically used within pattern-action rules defined in Möbius.

`literal`. A *literal* Variable contains the Tag `literal` and matches Expressions at face value, treating other Variables as constants rather than as specifications of sets of matching values. A *non-literal* Variable does not contain the Tag `literal` and will match in the usual manner by identifying common match values that can instantiate both itself and the candidate matched Expression. The Tag `literal` is also typically used within pattern-action rules defined in Möbius.

3.4 Phrases

Phrases are used to depict relationships between quantities. In those cases where a Phrase is intended to be aligned with language, the Function of the Phrase will typically be a Symbol depicting a noun, verb, adjective or adverb. The Arguments of a Phrase vary with context. For a Phrase with a verb as its Function, the Argument Labels may have Syntactic Relations that are verb argument relationships, like `subject` and `object`, or the Syntactic Relations may be Symbols depicting conjunctions or prepositions, with the values of these Arguments being other Phrases or Units, or the Syntactic Relations may be Symbols that name parts of speech, like `adverb`, with the values of these Arguments being Symbols or Expressions that depict the corresponding part of speech. The following example illustrates several varieties of Arguments for a Phrase with a verb as its function:

```
move(subject:vehicle "V507", adverb: quickly, on:road "Elm St")
```

Semantic Categories, if provided within the Labels of Arguments, are typically Symbols naming semantic classes in a predefined set or hierarchy.

Phrases intended to be aligned with language and containing a noun, adjective or adverb as their Function will have similar Arguments, except that no Syntactic Relations depicting verb argument relationships will appear. Following are examples of these types of Phrases:

```
speed(article: the, of:vehicle "V509")
associated(with:vehicle "V246")
```

In some cases, the Syntactic Relation of a Phrase Argument will be omitted, as, for example, when the Argument names the quantity that is the Function of the Phrase, or it introduces a list of quantities. Following are examples of these cases, also illustrating Phrases that have no Unit as Function:

```
vehicle(article: the, :vehicle "V246")
(:road "Elm St", and:road "98th St")
(:vehicle "V246", or:vehicle "V507", or:vehicle "V509")
```

The Semantic Category of a Phrase Argument is also omitted in some situations. A usage convention regarding the inclusion and omission of Semantic Category specifications is described in Section 4.3, Language Templates.

The primary consideration taken in the construction of Phrases intended to be aligned with language is that these Phrases should produce readable English when subjected to a simple language generation algorithm that respects the positioning of verb arguments relative to verbs, adjective modifiers relative to nouns, and so forth. For example, the Expression

```
exit(
  subject:vehicle "V002034",
  object:intersection intersection(
    article: the, of: (:road "D410", and:road "Highway 10")))
```

is well-formed in this sense, as a simple generation algorithm will produce the following English:

```
"V002034 exits the intersection of D410 and Highway 10".
```

3.5 Questions

Questions are used to depict relationships whose status is being requested, or specific instances of which are being requested. Arguments within Questions follow the same forms as Arguments within Phrases that have verbs as Functions. A yes/no Question will have no Variables within it, as, for example:

```
be(subject:vehicle "V507", on:road "Elm St", at:time "2009-05-23T14:00:00")?
```

A positive response to such a Question will be a Statement of the same form, indicating a match:

```
be(subject:vehicle "V507", on:road "Elm St", at:time "2009-05-23T14:00:00").
```

A negative response will be the absence of such a Statement, indicating no match.

A Question that requests a list of quantities will contain one or more Variables, as, for example:

```
be(
  subject:vehicle [some, "vehicles"], on:road "Elm St",
  at:time "2009-05-23T14:00:00")?

be(
  subject:vehicle [what], on:road "Elm St", at:time "2009-05-23T14:00:00")?
```

A response to the first Question might be:

```
be(subject:vehicle "V507", on:road "Elm St", at:time "2009-05-23T14:00:00").
be(subject:vehicle "V509", on:road "Elm St", at:time "2009-05-23T14:00:00").
be(subject:vehicle "V514", on:road "Elm St", at:time "2009-05-23T14:00:00").
```

while a response to the second Question might be:

```
"V507"
"V509"
"V514"
```

3.6 Statements

Statements are used to depict relationships that are asserted to be true. Arguments within Statements follow along the lines of Arguments within Questions and Arguments within Phrases that have verbs as Functions. Typically, a Statement will contain no Variables, as, for example:

```
turn(
  subject:vehicle "V002034", adverb:turning_direction left,
  from:time "2010-12-02T14:52:35", to:time "2010-12-02T14:52:39").
```

In some cases, however, Variables do appear within Statements (within rule definitions, for example).

Statements are used to represent a wide range of assertions in the Möbius language. Contents of datasets are typically rendered as Statements, as are the responses to many Questions.

3.7 Commands

Commands are used to specify instructions to be carried out by a receiving party. Arguments within Commands follow along the lines of Arguments within Questions and Statements and Arguments within Phrases that have verbs as Functions.

Following are examples of several Commands currently supported for processing by the IMPACT system:

```
open(object:database "blue")!
clear(object:table "INFERRED_EVENTS_")!
focus(
  on:vehicle "V246", from:time "2009-10-15T12:00:00",
  to:time "2009-10-15T12:15:00")!
initialize(object: event_summary(article: the))!
```

4. Language Usage

4.1 Grounding in Language

Möbius is designed to encode information in a manner that is closely aligned with language. Most Möbius Expressions adhere to this intent. The principal exception is the Variable structure, with its restriction Expressions (e.g., `>("10.0 m/s")`) that take the form of programming language or mathematical constructs.

Aligning Möbius Expressions with language provides them with intrinsic meaning. When a simple language generation algorithm is applied to Möbius Expressions—placing verbs between subjects and objects, adjectives before nouns they modify, and so forth—then the resulting English expressions produced by this algorithm articulate the meaning of the original Möbius Expressions. This correspondence has many benefits:

- It makes the language easy to learn.
- It encourages consistency of use by application developers.
- It can facilitate joint development by teams of application developers.
- It can assist in the debugging of information content and communications.
- It provides the language with substantial coverage of content.
- It simplifies interactions with human users.
- It enables the language to capture both precision and ambiguity.

Given the correspondence of Möbius Expressions to English language expressions, it is not necessary—or even advisable—to prescribe a universal mechanism by which Möbius requests

are “evaluated” to yield Möbius responses. As a convention, a Möbius response will typically echo the instigating Möbius request by listing zero or more instantiations of the request, each instantiation replacing any Variables in the request with value Expressions. The exception to this convention concerns the handling of reported Variables—Variables that include the Tag `what` or `which`—for which the response is a list of instantiations of that Variable only, rather than instantiations of the entire request Expression. Within the boundaries established by these conventions on request–response format, however, the manner in which a component arrives at its response to a request may vary. Depending on the situation, a system might perform a calculation, retrieve information from a data source, or consult with other systems or humans in order to formulate a response to a received request. What matters, given the grounding of Möbius in language, is that the response be appropriate—interpreted in human language—given the nature of the request and the extent of that component’s knowledge and capabilities.

4.2 Keeping It Simple

An attempt has been made to keep Möbius as simple as possible. One aspect of the language that facilitates this is the distinction between Descriptors and Symbols. Descriptors can be used for anything that is not tidy—values with infinite variations or having inconsistent format—whereas Symbols can be used for small sets of values specific to the application at hand.

Syntactic Relations and Semantic Categories can be included or omitted within the Labels of Möbius Arguments, depending on several factors including their utility in clarifying the syntactic construction and meaning of Möbius Expressions and their utility in facilitating the matching of Möbius Expressions. Where the intent is clear from context, nouns, verbs, adjectives and adverbs can be placed in the Function position of Forms rather than as the first labeled Argument to those Forms. Prepositions and conjunctions, being relatively few in number, are used directly in the Syntactic Relation position, saving an extra level of nesting. In general, terms with multiple meanings are tolerated, as long as their inclusion does not interfere with matching (e.g., the Symbol `object` may appear as a Syntactic Relation to indicate a direct object or elsewhere to indicate the top-most semantic category). Also, for compactness, multi-word constructions are allowed in place of nouns (e.g., `named_heading` or `horizontal_speed`), verbs (e.g., `pick_up` or `not_move`), prepositions (e.g., `in_back_of` or `away_from`), and so forth.

4.3 Language Templates

An extremely useful information encoding technique facilitated by Möbius is that of creating and applying *language templates*. A language template is an expression that, despite having an arbitrary level of syntactic complexity in itself, remains constant as it envelops a fixed number of other language expressions. In Möbius, the constant and variable portions within a language

template are distinguished through the inclusion and omission of Semantic Categories within Argument Labels. The following Statement is an example of a language template that has been applied to several contained Expressions:

```
make(
  subject:vehicle "V507", object: U-turn(article: a),
  at:intersection intersection(
    article: the, of: (:road "95th St", and:road "Chicago Ave")),
  from:time "2008-05-12T09:55:48.817", to:time "2008-05-12T09:55:53.319").
```

The language template itself is that portion of the outermost Expression that extends to inner levels in those places where there is no Semantic Category within an Argument's Label. Where a Semantic Category is specified in an Argument's Label, the value position of that Argument is an enveloped Expression, not part of the language template. For the Expression above, the language template is:

```
make(
  subject:vehicle [], object: U-turn(article: a), at:intersection [],
  from:time [], to:time []).
```

In the notation of language templates, the "slots" for insertion of enveloped Expressions are depicted as empty Variables (`[]`). In the original Expression listed above, the enveloped Expressions are

```
"V507"
intersection(article: the, of: (:road "95th St", and:road "Chicago Ave"))
"2008-05-12T09:55:48.817"
"2008-05-12T09:55:53.319"
```

The second of these Expressions, in turn, can itself be viewed as a language template

```
intersection(article: the, of: (:road [], and:road []))
```

applied to two enveloped Expressions

```
"95th St"
"Chicago Ave"
```

Language templates are used in the encoding of relative attribute value expressions (RAVEs) and also for English wrapping of database tables.

4.4 Support for Matching

Matching of Möbius Expressions to other Möbius Expressions is made simpler by adopting several usage conventions. Typically, nouns are depicted in the singular (e.g., `road`), and verbs

are depicted in the infinitive (e.g., `enter`), except when necessary to distinguish meanings or to prepare an Expression for algorithmic language generation. Also, the article "a"/"an" is typically depicted as simply `a` .

A powerful form of matching between Möbius Expressions allows for both the reordering of Arguments and the omission of Arguments in a pattern Expression, relative to the Expression to which it is to be matched. Allowing for reordering and omission of Arguments has several consequences. First, it is often not the case that the Arguments of a Möbius Expression need to be arranged in any particular order. However, ordering does matter when an Expression is to be generated in English. For example, listing a verb's object before its subject can force a language generator to produce a passive rather than active voice rendering of a statement. Likewise, ordering prepositional phrases in an intuitive manner within an Expression can help a language generator produce natural-sounding English renderings. Second, when allowing for the omission of Arguments in a pattern Expression gives rise to unintended matches, it is sometimes necessary to introduce additional Arguments in both the pattern Expression and in the Expressions to which that pattern Expression is to be matched, so that the unintended matches do not succeed. Finally, since the omission of Arguments in a pattern Expression can lead to a match, negation of verbs is specified directly within the Symbols depicting verbs (e.g., `move` versus `not_move`), thereby assuring the absence of a match between positive and negative forms of a statement.

5. Using Möbius in Distributed Interpretation of English Requests

One application of the Möbius language has been to represent English requests in various stages of partial interpretation by collaborating software systems. This application is described in [Katz *et al.*, 2007], regarding distributed interpretation of requests submitted to mobile phones working in collaboration with a central language server and supporting resources. Language interpretation depends on access to relevant knowledge, and in a distributed environment, this knowledge is also distributed. Individual components in this environment can form partial interpretations of user-submitted requests, then relay the partially interpreted requests to other components, which can complete the interpretation of the requests. Once a request has been fully interpreted, it can be fulfilled, also in a collaborative manner.

As an encoding of language, Möbius can represent both the ambiguity inherent in many user-submitted natural language requests and the precision resulting from subsequent interpretation of those requests. The following sections describe several types of ambiguities that can arise within natural language requests, plus corresponding ways in which Möbius encodings of the requests can be transformed to reflect the resolution of the ambiguities from

interpretation of the requests. Some of these varieties of ambiguity were handled in our application of distributed language interpretation in mobile phone environments; other varieties described here are amenable to this approach to encoding and intercommunication, but have not yet been addressed in our applications.

5.1 Interpreting References

Names, times and places can be specified in varying degrees of precision within user-submitted natural language requests. A request to “Call Karen at 8 o’clock.” can be initially encoded as

```
call(object:human "Karen", at:time "8 o’clock"))!
```

by a component responsible for parsing submitted natural language requests. This component might also replace the reference `"8 o’clock"` with a specific time such as `"2014-01-21T20:00:00"`, based on the current date and time. On the other hand, the reference to `"Karen"`, might have to be resolved by a different component that has access to that user’s contact list. That component might replace `"Karen"` with a more specific reference like `"Karen Jones"`.

A similar situation can arise when pronouns are to be resolved. Suppose a submitted request is “Where does he live?”. A component responsible for parsing this request might initially encode the request as

```
live(subject:human he, at:address [what])?
```

If a record of items mentioned in recent requests and responses is maintained by a separate component, however, that second component will have to resolve the reference `he` to a more specific reference like `"John Nelson"`.

Separately, descriptive references can also be interpreted and replaced. A request “Where does Kevin Brown’s brother live?” might initially include a sub-expression `brother(of:human "Kevin Brown")` which can be subsequently replaced by `"David Brown"`.

5.2 Interpreting Abstract Terms

Distributed interpretation of natural language requests can involve replacement of abstract terms with more specific terms, possibly in different ways by different components. For example, a request “When did John most recently contact me?” might initially be encoded as:

```
contact(subject:human "John", object:human me, adverb: most_recently,
  at:time [what])?
```

then be relayed to a number of associated components, which might each form their own interpretation of the original request. Components responsible for maintaining records of phone calls or e-mails might replace `contact` with `call` and `e-mail` respectively, while a component that maintains a calendar might replace `contact` with `meet`.

Similarly, interpretation of requests may involve the replacement of other parts of speech with more specific terms. A request "Did Sally send me a message yesterday?" might initially incorporate a noun `message` that is subsequently replaced with `e-mail_message` and `text_message` by components responsible for maintaining records of these types of messages. Or, interpretation of a request "Take a high-resolution picture." might involve replacement of `high-resolution` with `"20 Mpixel"` by a component familiar with the capabilities of an associated camera.

5.3 Interpreting Ambiguous Expressions

Interpretation of requests can also involve larger-scale replacement of expressions in order to resolve ambiguities. [Katz *et al.*, 2007] argues that it is appropriate for components in a distributed processing environment to form interpretations of requests that are consistent with their own capabilities to respond to requests. For example, if a request "Is Mark Smith at MIT?" is received, this request might be initially encoded as

```
be(subject:human "Mark Smith", at:object "MIT")?
```

leaving the precise semantic category of `"MIT"` unspecified. A component that has access to information about physical locations of people might re-express this request as a request about presence at a facility:

```
be(subject:human "Mark Smith", at:facility "MIT")?
```

while a component that has access to employment information might re-express this as a request about employment:

```
employ(subject:organization "MIT", object:human "Mark Smith")?
```

In other cases, syntactic ambiguity may require multiple encodings of a request to be maintained temporarily, pending selection and further interpretation by other components. For example, if a request "Reschedule my meeting at 4pm." is received, this might initially be encoded in three forms:

```
reschedule(object:event meeting(determiner: my, at:time "4pm"))!
reschedule(object:event meeting(determiner: my), at:time "4pm")!
reschedule(object:event meeting(determiner: my), to:time "4pm")!
```

These encodings capture three possible interpretations of the original request, depending on whether 4 PM is taken to be the current time of the meeting, the time at which the rescheduling is to be done, or the new time of the meeting.

A component with access to the user's calendar might subsequently make a determination that, if there is already a meeting scheduled at 4 PM, the first encoding will be selected. Otherwise, that component might discard the first encoding, then ask the user which of the remaining two interpretations was intended.

5.4 Adding Sub-Expressions

Components in a distributed language interpretation environment can also add language expressions to encoded requests in order to specify missing details. For example, suppose a submitted request is to "Call Diane Taylor.". An initial encoding of the request might be as follows:

```
call(object:human "Diane Taylor")!
```

A component that has information about the current network connectivity of the calling device might then add one of the following arguments to the encoded request:

```
... using:network VOIP_network(article: a)
... using:network cellular_network(article: a)
```

A separate component with information about Diane Taylor's schedule might then add one of the following arguments to the encoded request:

```
... at:phone_number home_phone_number(article: a)
... at:phone_number cell_phone_number(article: a)
... at:phone_number office_phone_number(article: a)
```

That same component or a separate component with access to contact phone numbers might then replace this argument with an argument specifying a literal phone number.

6. Using Möbius to Wrap Information in Database Tables

Database tables and sets of columns within database tables often depict relationships that can be expressed in natural language. Constructing a natural language interface to data in tables of this sort can involve an intermediate step of "wrapping" the tables or column sets in natural language, so that natural language statements can be used to insert data into the tables and natural language queries can be used to extract data from the tables.

In other cases, database tables are created for the express purpose of storing and searching through large sets of natural language statements. These tables can also be interfaced to language by wrapping, associating the tables or sets of their columns with natural language expressions.

Möbius can be used to facilitate both of these wrapping applications. Sections 6.1 through 6.3 describe three encoding schemes used within the IMPACT reasoning system for associating database tables or sets of database table columns with natural language expressions. These encoding schemes pair individual table columns to fields or slots within natural language patterns. Section 6.4 describes several mechanisms used within IMPACT to map data values that appear in these columns to corresponding natural language tokens.

6.1 Matched Expression Encoding

In the "matched expression" encoding scheme, explicitly specified language expressions are associated with a table or set of columns. These expressions contain one or more variables which are associated with individual columns of the table. Möbius Statements containing Variables are used to encode ways of entering information into a table, and Möbius Questions containing Variables are used to encode ways of retrieving information from a table. If a table has both Statements and Questions associated with it in a matched expression encoding, then natural language can be used both to write to the table and read from the table. If only Questions are associated with the table, natural language can only be used to read information from the table.

As an example, a table that records timestamped instances of vehicles being on particular roads might have five columns: "VEHICLE", "TIME", "ROAD_NAME_VARIANT", "ROAD" (canonical name) and "ROAD_MULTILINESTRING" (geometric description). As part of creating functionality to answer questions like "When is V234 on Indiana Ave?" and "Which cars are on 89th Street at 10 AM?", the first three columns of this table could be wrapped with the following Möbius Question:

```
be(subject:vehicle [some, "vehicle 1"], on:road [some, "road 1"],
  at:time [some, "time 1"])?
```

The Variables `[some, "vehicle 1"]`, `[some, "road 1"]` and `[some, "time 1"]` would then be associated with the columns "VEHICLE", "ROAD_NAME_VARIANT" and "TIME", respectively. If the wrapping were to include only the above Question and not a corresponding Möbius Statement as a matched expression encoding, then the wrapping would be read-only for this table.

6.2 Templated Expression Encoding

The templated expression encoding makes use of language templates to distribute the information content of language expressions among the columns of a database table. One designated column of the table is used to store a language template—chosen independently for each row of the table—and the remaining columns of the table are used to store values that fill the slots in these templates, either sequentially as the slots appear within the templates or organized into particular categories by semantic type or roles played in the surrounding expressions.

As an example, suppose a table is to be used to store statements of observed events such as "Human 7263 bends over from 2013-10-16T22:48:18.100 to 2013-10-16T22:48:18.700." and "Human 9240 takes Object 252 from Human 9209 from 2013-10-16T23:18:06.900 to 2013-10-16T23:18:08.700." These two events can be encoded as Möbius Statements as follows:

```
bend_over(subject:human "Human 7263", from:time "2013-10-16T22:48:18.100",
  to:time "2013-10-16T22:48:18.700").

take(subject:human "Human 9240", object:tangible_object "Object 252",
  from:human "Human 9209", from:time "2013-10-16T23:18:06.900",
  to:time "2013-10-16T23:18:08.700").
```

Treating each Statement as the application of a language template to one or more arguments, the language templates can be extracted

```
bend_over(subject:human [], from:time [], to:time []).

take(subject:human [], object:tangible_object [], from:human [], from:time [],
  to:time []).
```

and stored in one column of the table, and the arguments

```
"Human 7263"
"2013-10-16T22:48:18.100"
"2013-10-16T22:48:18.700"

"Human 9240"
"Object 252"
"Human 9209"
"2013-10-16T23:18:06.900"
"2013-10-16T23:18:08.700"
```

can be stored in other columns, arranged sequentially or by semantic type or role played.

6.3 RAVE Encoding

The RAVE encoding scheme is used to store relative attribute value expressions, or RAVEs, as described in Section 7. RAVEs can be used to express a range of language-based comparisons involving either constant values or attributes applied to one or two objects and possibly measured at particular time points. As such, any RAVE can be encoded using a subset of the following nine fields, each field represented as a database table column:

predicate
primary attribute
primary object 1
primary object 2
primary time
secondary attribute
secondary object 1
secondary object 2
secondary time

Specific encodings for particular types of RAVEs (Level 1 RAVEs, Level 2 RAVEs and Level 3 RAVEs) are described in Section 7.

6.4 Relating Data Values and Language Tokens

The encoding schemes described above specify ways in which larger natural language expressions may be decomposed into language templates or patterns combined with data values maintained in database table columns. Using these schemes, individual Variables or RAVE fields in the language templates or patterns are associated with individual database table columns, and it is necessary to specify additionally how particular natural language tokens, as values of these Variables or RAVE fields, may be matched or converted to and from individual data values within the database table columns.

The IMPACT system provides a two-level mechanism for relating natural language tokens and database table values. The top level of this mechanism specifies whether a constant is assumed on either side of the association between a Variable or RAVE field, on the one hand, and a table column, on the other. Specifying a constant value on one side of such an association can simplify the language requests that are used or produce a more compact database representation.

The bottom level of the association mechanism specifies how individual natural language tokens are related to individual database values. Possibilities are:

- the natural language token is specified as corresponding to a Descriptor, in which case double quotes are added to surround an individual database table value,
- the natural language token is specified as corresponding to a Symbol, in which case the underscore character is substituted for all whitespace characters within an individual database table value,
- additionally, the natural language token may be specified with "null conversion", in which case an empty text string as a database table value will be translated to the Möbius Symbol `null`,
- also additionally, the natural language token may be specified as adding a particular prefix (e.g., `"$"`) or suffix (e.g., `" m/s"`) to the corresponding database table value.

The above-described capabilities are currently implemented within the IMPACT system. Additional association and conversion capabilities may be envisioned as well: number formatting, timestamp conversions, and so forth.

7. Using Möbius to Describe Events

7.1 Relative Attribute Value Expressions

Möbius can be used to assert that an event has occurred. For example:

```
turn(
  subject:vehicle "V507", adverb:turning_direction left,
  at:intersection intersection(
    article: the, of: (:road "96th St", and:road "Beaufort Ave")),
  from:time "2007-09-27T09:55:16.177", to:time "2007-09-27T09:55:29.120").
```

Language goes deeper, though, and can also express what happens during an event, in terms of individual states and changes of the participants. This section describes the use of Möbius to encode states and changes in the *transition space* representation, as described in [Borchardt, 1992] and [Borchardt, 1994]. The following are examples of using Möbius to encode states and changes in this manner:

```
equal(
  subject:attribute turning_direction(
    article: the, of:vehicle "V507", at:time "2007-09-27T09:55:21.804"),
  object:value left).

exist(
  subject:attribute speed(article: the, of:vehicle "V507"),
  at:time "2007-09-27T09:55:21.804").
```

```
appear(
  subject:attribute be(
    subject:vehicle "V507",
    at:intersection intersection(
      article: the, of: (:road "96th St", and:road "Beaufort Ave"))),
  between: (
    :time "2007-09-27T09:55:16.177", and:time "2007-09-27T09:55:17.865")).
```

These statements are called *relative attribute value expressions*, or RAVEs. RAVEs encode simple pairwise comparisons expressible in language: comparisons of time-varying quantities to reference quantities, reference quantities to reference quantities, time-varying quantities to other time-varying quantities, or time-varying quantities to themselves at different times. RAVEs are composed of four types of elements: objects, attributes, times, and predicates. In the above examples, the objects are:

```
"V507"
left
intersection(article: the, of: (:road "96th St", and:road "Beaufort Ave"))
```

the attributes are:

```
turning_direction(article: the, of:vehicle [])
speed(article: the, of:vehicle [])
be(subject:vehicle [], at:intersection [])
```

the times are:

```
"2007-09-27T09:55:21.804"
"2007-09-27T09:55:16.177"
"2007-09-27T09:55:17.865"
```

and the predicates are:

```
equal
exist
appear
```

The Möbius encoding of RAVEs makes use of a two-level nesting of language templates. The outermost language template encodes the predicate and specifies each argument as having a Semantic Category of either `attribute`, `value` or `time`. The outermost template for the first RAVE example above is:

```
equal(subject:attribute [], object:value []).
```

Within each argument labeled `attribute`, a second use of language templates appears. Here, an attribute such as

```
turning_direction(article: the, of:vehicle [])
```

is applied to one or two objects such as

```
"V507"
```

plus a possible time Argument such as

```
"2007-09-27T09:55:21.804".
```

7.2 Level 1 RAVEs

There are four predicates at the lowest level of the transition space representation. These are:

```
equal
not_equal
exceed
not_exceed
```

This lowest level of the transition space representation depicts direct comparisons between quantities. The quantities can be either attribute applications or values, and the comparisons may be at specified times, or independent of time. RAVEs at the lowest level of representation in the transition space representation are called “Level 1 RAVEs”.

The outermost language template of a Level 1 RAVE encodes the predicate and the types of quantities being compared, attributes or values. Examples of these types of language templates are as follows:

```
not_equal(subject:value [], object:value []).
exceed(subject:attribute [], object:attribute []).
not_exceed(subject:value [], object:attribute []).
```

An example of a Level 1 RAVE comparing a value to a value is as follows:

```
exceed(subject:value "15.2 deg/s", object:value "3.7 deg/s").
```

Attributes are based on English expressions like

the position of a vehicle
the speed of a vehicle
the turning direction of a vehicle
the distance between a vehicle and a vehicle
an object being an instance of an object

a vehicle being operational
a vehicle being on a road
a vehicle being at an intersection

and these are encoded as language templates in a manner that parallels their English form:

```
position(article: the, of:vehicle [])
speed(article: the, of:vehicle [])
turning_direction(article: the, of:vehicle [])
distance(article: the, between: (:vehicle [], and:vehicle []))
be(subject:object [], noun: instance(article: a, of:object []))
be(subject:vehicle [], adjective: operational)
be(subject:vehicle [], on:road [])
be(subject:vehicle [], at:intersection [])
```

Attributes apply to either one or two objects.

When there is no reference to time, the attribute's language template is simply applied to its objects:

```
equal(
  subject:attribute road_type(article: the, of:road "95th St"),
  object:value "local road").
```

For Level 1 RAVEs, the outermost language template does not reference time. Comparisons that involve times do so by inserting a time Argument within an attribute application, as in the following example:

```
exceed(
  subject:attribute speed(
    article: the, of:vehicle "V507", at:time "2007-09-27T10:12:12.511"),
  object:attribute speed(
    article: the, of:vehicle "V507", at:time "2007-09-27T10:12:11.948")).
```

When a RAVE encoding scheme is used to store a Level 1 RAVE in a database table, all nine fields of the encoding scheme representation are used. If any of the attribute, object or time values are not specified by a RAVE, the corresponding field values are considered to be empty Symbols and are stored as empty text strings in the database.

7.3 Level 2 RAVEs

There are two predicates at the second level of the transition space representation:

```
exist
not_exist
```

Each attribute is assumed to have a null value in its range. When an attribute applied to one or two objects does not equal the null value at a particular time, the attribute is said to exist or be present at that time. If it does equal the null value, the attribute is said not to exist or not to be present at that time. Existence and non-existence of attributes at particular times is encoded in Level 2 RAVEs, which are defined in terms of corresponding Level 1 RAVEs.

Examples of Level 2 RAVEs are:

```
exist(
  subject:attribute be(subject:vehicle "V507", in_front_of:vehicle "V509"),
  at:time "2007-09-27T09:57:41.367").

not_exist(
  subject:attribute turning_rate(article: the, of:vehicle "V509"),
  at:time "2007-09-27T09:57:55.436").
```

These RAVEs utilize the following outermost language templates:

```
exist(subject:attribute [], at:time []).
not_exist(subject:attribute [], at:time []).
```

In turn, these two RAVEs correspond to the following Level 1 RAVEs:

```
not_equal(
  subject:attribute be(
    subject:vehicle "V507", in_front_of:vehicle "V509",
    at:time "2007-09-27T09:57:41.367"),
  object:value null).

equal(
  subject:attribute turning_rate(
    article: the, of:vehicle "V509", at:time "2007-09-27T09:57:55.436"),
  object:value null).
```

Level 2 RAVEs that involve attributes with function Symbol `be` can be rendered in an alternate form that utilizes `be` or `not_be` at the outermost level:

```
be(subject:vehicle "V507", in_front_of:vehicle "V509",
  at:time "2007-09-27T09:57:41.367").
not_be(subject:vehicle "V507", in_front_of:vehicle "V509",
  at:time "2007-09-27T09:57:41.367").
```

When encoded in a database table using a RAVE encoding scheme, Level 2 RAVEs utilize five of the nine fields: predicate, primary attribute, primary object 1, primary object 2, and primary time. If the second object value is not specified by a RAVE, the corresponding field value is considered to be an empty Symbol and is stored as an empty text string in the database.

7.4 Level 3 RAVEs

There are ten predicates at the third level of the transition space representation:

```
appear
not_appear
disappear
not_disappear
change
not_change
increase
not_increase
decrease
not_decrease
```

These predicates cover the range of relative changes possible for boolean, qualitative and quantitative attributes, given the presence of a null value in the range of each attribute.

Following is an example of a Level 3 RAVE:

```
disappear(
  subject:attribute be(
    subject:vehicle "V509",
    at:intersection intersection(
      article: the, of: (:road "83rd St", and:road "Clinton Ave"))),
  between: (
    :time "2007-09-27T09:57:59.375", and:time "2007-09-27T09:58:01.064")).
```

This RAVE uses the following outermost language template:

```
disappear(subject:attribute [], between: (:time [], and:time [])).
```

Level 3 RAVEs are also defined in terms of Level 1 RAVEs. For Boolean attributes, the predicates `appear`, `not_appear`, `disappear` and `not_disappear` are used and are defined in terms of the attribute equaling or not equaling the null value at the beginning and end of the change interval. For example, the following Level 3 RAVE:

```
appear(
  subject:attribute be(subject:vehicle "V507", on:road "Colton Ave"),
  between: (
    :time "2007-09-27T09:57:57.687", and:time "2007-09-27T09:57:58.250")).
```

is defined as being equivalent to the following Level 1 RAVEs:

```
equal(
  subject:attribute be(
    subject:vehicle "V507", on:road "Colton Ave",
    at:time "2007-09-27T09:57:57.687"),
  object:value null).
```

```
not_equal(
  subject:attribute be(
    subject:vehicle "V507", on:road "Colton Ave",
    at:time "2007-09-27T09:57:58.250"),
  object:value null).
```

For qualitative attributes, which have multiple unordered, non-null values, the predicates `appear`, `not_appear`, `disappear` and `not_disappear` can be used, and the predicates `change` and `not_change` can also be used in those cases where the predicate `not_disappear` would apply. As an example, the following Level 3 RAVE

```
change(
  subject:attribute position(article: the, of:vehicle "V507"),
  between: (
    :time "2007-09-27T09:57:36.303", and:time "2007-09-27T09:57:36.865")).
```

is defined as being equivalent to the following Level 1 RAVEs:

```
not_equal(
  subject:attribute position(
    article: the, of:vehicle "V507", at:time "2007-09-27T09:57:36.303"),
  object:value null).

not_equal(
  subject:attribute position(
    article: the, of:vehicle "V507", at:time "2007-09-27T09:57:36.865"),
  object:value null).

not_equal(
  subject:attribute position(
    article: the, of:vehicle "V507", at:time "2007-09-27T09:57:36.865"),
  object:attribute position(
    article: the, of:vehicle "V507", at:time "2007-09-27T09:57:36.303")).
```

Finally, for quantitative attributes, which have multiple ordered, non-null values, the predicates `appear`, `not_appear`, `disappear`, `not_disappear`, `change` and `not_change` can be used, and four additional predicates can also be used in those cases where the predicate `not_disappear` would apply: `increase`, `not_increase`, `decrease` and `not_decrease`. For example, the following Level 3 RAVE

```
decrease(
  subject:attribute speed(article: the, of:vehicle "V507"),
  between: (
    :time "2007-09-27T09:57:35.177", and:time "2007-09-27T09:57:35.740")).
```

is defined as being equivalent to the following Level 1 RAVEs:

```
not_equal(
  subject:attribute speed(
    article: the, of:vehicle "V507", at:time "2007-09-27T09:57:35.177"),
  object:value null).

not_equal(
  subject:attribute speed(
    article: the, of:vehicle "V507", at:time "2007-09-27T09:57:35.740"),
  object:value null).

exceed(
  subject:attribute speed(
    article: the, of:vehicle "V507", at:time "2007-09-27T09:57:35.177"),
  object:attribute speed(
    article: the, of:vehicle "V507", at:time "2007-09-27T09:57:35.740")).
```

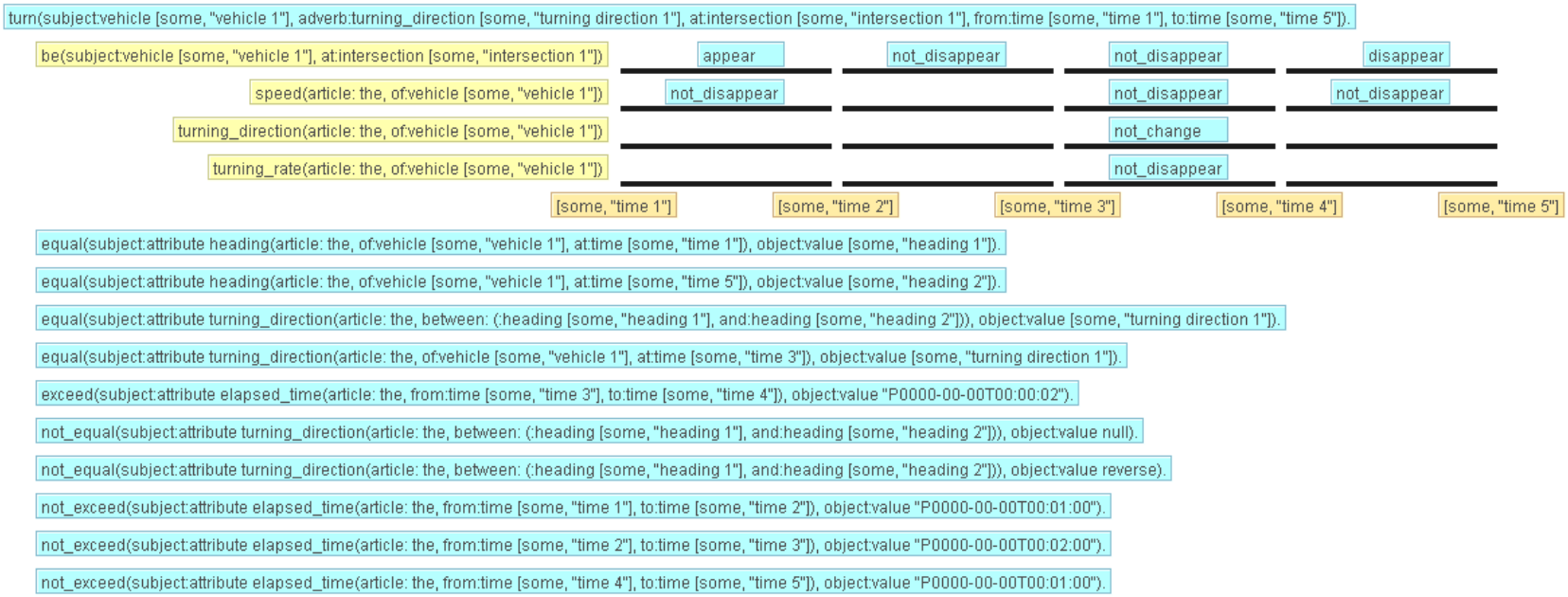
Level 3 RAVEs that involve attributes with function Symbol `be` can also be rendered in an alternate form that utilizes `become`, `not_become`, `cease_to_be`, or `not_cease_to_be` at the outermost level:

```
become(subject:vehicle "V507", on:road "Colton Ave", between:
  (:time "2007-09-27T09:57:57.687", and:time "2007-09-27T09:57:58.250")).
not_become(subject:vehicle "V507", on:road "Colton Ave", between:
  (:time "2007-09-27T09:57:57.687", and:time "2007-09-27T09:57:58.250")).
cease_to_be(subject:vehicle "V507", on:road "Colton Ave", between:
  (:time "2007-09-27T09:57:57.687", and:time "2007-09-27T09:57:58.250")).
not_cease_to_be(subject:vehicle "V507", on:road "Colton Ave", between:
  (:time "2007-09-27T09:57:57.687", and:time "2007-09-27T09:57:58.250")).
```

When encoded in a database table using a RAVE encoding scheme, Level 3 RAVEs utilize six of the nine fields: predicate, primary attribute, primary object 1, primary object 2, primary time (later time) and secondary time (earlier time). If the second object value is not specified by a RAVE, the corresponding field value is considered to be an empty Symbol and is stored as an empty text string in the database.

7.5 Event Models

Level 1, Level 2 and Level 3 RAVEs can be used together to describe "what happens" during an event. An *event model* provides such a description for an abstract event, with times and possibly objects specified as Variables. Following is an example of an event model for turning left or right at an intersection, using a graphical notation that condenses the underlying Möbius Expressions:


turn(subject:vehicle [some, "vehicle 1"], adverb:turning_direction [some, "turning direction 1"], at:intersection [some, "intersection 1"], from:time [some, "time 1"], to:time [some, "time 5"]).
be(subject:vehicle [some, "vehicle 1"], at:intersection [some, "intersection 1"])
speed(article: the, of:vehicle [some, "vehicle 1"])
turning_direction(article: the, of:vehicle [some, "vehicle 1"])
turning_rate(article: the, of:vehicle [some, "vehicle 1"])
appear
not_disappear
not_disappear
disappear
not_disappear
not_disappear
not_disappear
not_change
not_disappear
[some, "time 1"]
[some, "time 2"]
[some, "time 3"]
[some, "time 4"]
[some, "time 5"]
equal(subject:attribute heading(article: the, of:vehicle [some, "vehicle 1"], at:time [some, "time 1"]), object:value [some, "heading 1"]).
equal(subject:attribute heading(article: the, of:vehicle [some, "vehicle 1"], at:time [some, "time 5"]), object:value [some, "heading 2"]).
equal(subject:attribute turning_direction(article: the, between: (:heading [some, "heading 1"], and:heading [some, "heading 2"])), object:value [some, "turning direction 1"]).
equal(subject:attribute turning_direction(article: the, of:vehicle [some, "vehicle 1"], at:time [some, "time 3"]), object:value [some, "turning direction 1"]).
exceed(subject:attribute elapsed_time(article: the, from:time [some, "time 3"], to:time [some, "time 4"]), object:value "P0000-00-00T00:00:02").
not_equal(subject:attribute turning_direction(article: the, between: (:heading [some, "heading 1"], and:heading [some, "heading 2"])), object:value null).
not_equal(subject:attribute turning_direction(article: the, between: (:heading [some, "heading 1"], and:heading [some, "heading 2"])), object:value reverse).
not_exceed(subject:attribute elapsed_time(article: the, from:time [some, "time 1"], to:time [some, "time 2"]), object:value "P0000-00-00T00:01:00").
not_exceed(subject:attribute elapsed_time(article: the, from:time [some, "time 2"], to:time [some, "time 3"]), object:value "P0000-00-00T00:02:00").
not_exceed(subject:attribute elapsed_time(article: the, from:time [some, "time 4"], to:time [some, "time 5"]), object:value "P0000-00-00T00:01:00").

In this event model, the vehicle first enters the intersection (being "at" the intersection appears), then it remains at the intersection while possibly stopping and restarting, then it turns, and then it exits the intersection. Additional constraints on the event activity are listed beneath the changes: the turning must be in the specified direction, and various time bounds have been placed on the intervals of the event. Event models such as this have been used within IMPACT in support of event recognition, summarization, and related reasoning operations.

8. References

Borchardt, G. C. (1992). "Understanding causal descriptions of physical systems." In *Proceedings of the AAAI Tenth National Conference on Artificial Intelligence*, 2–8.

Borchardt, G. C. (1994). *Thinking between the Lines: Computers and the Comprehension of Causal Descriptions*, MIT Press.

Katz, B., Borchardt, G., Felshin, S., and Mora, F. (2007). "Harnessing Language in Mobile Environments." In *Proceedings of the First IEEE International Conference on Semantic Computing (ICSC 2007)*, 421–428.