Conversations with Mother

Project 103

Student Number:	1229187
Student Name:	Ryan Gibbs
Module Code:	CM3203
Submission Date:	06/05/2016

Abstract

The aim of this project is to ascertain whether controlled English is a suitable, effective and user-friendly interaction method with internet of things devices. The project attempts to answer this question by providing an application which allows users to interact with a smart home device, Mother, using controlled English. The application was designed to make use of the familiarity and powerful reasoning capabilities found in controlled English to provide the user with a system which is not only able to relay information from Mother, but is also able to make inferences through the use of rule statements and accurately model the information provided by Mother.

The main deliverable of this project is an extensible platform prototype, making use of controlled English and the data generated by Mother to allow the user to interact with Mother in a conversational fashion. These interactions could take the form of questions and answers, along with statements to inform the application of new facts.

Acknowledgements

I would like to express my gratitude to Professor Alun Preece for his patient guidance as my supervisor for this project, his advice and enthusiasm have helped tremendously throughout the development.

I would also like to thank Will Webberley, whose existing work and expertise in the development of CENode made this project possible.

Finally, I would like to thank my family for their continued reassurance and support.

Table of Contents

Table of Figures

Introduction

This section will provide an overview of the project including its aims, objectives, the approach to development and a summary of outcomes.

Collaboration between humans and computer systems has taken place in one form or another since computers were first invented. It can be almost universally agreed that there are certain areas in which humans excel and other areas in which we are thoroughly outmatched by automated systems (Cummings, 2014). In the early days of computing this was most evident when computers were used to supplement humans in complex calculations, information storage, or analysis of very large data sets; a famous example of this being the Colossus machines which were used to aid cryptanalysis in the 1940s.

More recently, computer systems have increased in complexity and capability to a point where they are able to more directly collaborate with us on certain tasks. This collaboration is important because it allows us to take advantage of the differing capabilities afforded by humans and machines. In most cases this results in the human being given the role of 'decision maker' and machines being used to support us, either through the use of automation or by providing relevant information (the data to decision process) (Preece, et al., 2014). The degree of automation is determined by the complexity of the tasks, the capability of the machine, and the cognitive workload required to carry out the task. In the SRK taxonomy (Cummings, 2014), tasks are broken down into skill-based, rule-based, and knowledge-based categories with automation being most easily applied to skill-based tasks while knowledge-based tasks lend themselves to human decision making.

There has been a growing trend towards embedding computing in everyday life, beginning in the early 1990s with pervasive and ubiquitous computing, and continuing today with the internet of things (IoT). Both Mother and Cookies are examples of IoT devices designed to be placed in a user's home and blend seamlessly into the background; this concept of 'invisible intelligence' is central to the internet of things.

This invisible intelligence empowers the internet of things in ways that other devices traditionally have not been. IoT devices and services are ever-present and their ability to manipulate the world allows them to perform actions on behalf of the user that would traditionally fall into the domain of the manual. The ubiquity and ease of interaction with these kinds of devices present new opportunities for human-thing collaboration, with variable workload on the part of the device or human operator. The IoT devices could operate in a passive capacity, providing the user with information, in turn allowing them to make decisions; devices could also operate in a more active fashion, performing actions on behalf of the user – sometimes without their explicit intention. This can be seen at some level today, with the first use case exemplified by Mother and its home monitoring. The second use case is more suited to smart-home products such as thermostats or lighting controls, which alter the state of the house for the user without any explicit interaction.

This kind of collaboration between humans and machines allows for benefits in terms of efficiency, information availability, economy and task completion – among others.

The goal of this project is to allow users to interact with an IoT device, Mother, to gather information about the state of their home as measured by the device's sensors, which are themselves IoT devices; having a variety of sensing capabilities, with event feeds generated depending on the applications installed.

Mother (Figure 1) is a smart home device created by Sense (Sen.se, 2016) designed to serve as a central hub for up to 24 simultaneously registered Cookie sensors. While this would fall under a low level of automation, the collaboration between humans and machines is still important as it allows the machine to provide the human with the information needed to inform decision making.

Figure 1: Sense Mother with Cookie sensors

The interaction between the user, Mother and the Cookie sensors is to take the form of controlled natural language (Preece, et al., 2014). Interactions with Mother and Cookies could also flow in the reverse direction, i.e. users telling Mother things, allowing them to task the unit or sensors to measure or collect some data. This type of interaction would also include the passing of some previously unknown information to the device, for example telling Mother "the person X owns the Cookie Y", this would in turn allow for queries making use of person X's name, "where is person X now?".

This type of conversational interaction is beneficial as it allows us to sidestep the GUI (graphical user interface). The GUI, while providing a more accessible method of device interaction since its inception, has become increasingly unwieldy – presenting new design challenges and cognitive workload on the part of the user as devices have evolved. With increasing reliance on mobile computing via smartphones and even smartwatches, the GUI finds itself crammed into smaller and smaller form factors while attempting to accommodate increasing amounts of information and interaction complexity. The use of controlled natural language and a conversational interaction method provides the user with a powerful and expressive means of interaction, removing much of the complexity of the modern GUI while still being familiar and easy to use.

Currently the only way users have of accessing data collected by the Mother hub and its sensors is a web service made available by the Sense platform or a mobile application which presents a pared-down version of the online dashboard. The conversational approach outlined above would allow users to ask Mother about things it is capable of measuring (for example temperature or house occupancy etc.). The queries and responses are to be given in controlled natural language, as this provides common ground for machine and human readability, and is a natural way for humans to interact (Libov, 2016).

The main challenge of this project is to integrate the API and data made available by Mother with a controlled English processing environment. IBM's CEStore or CENode developed by Cardiff University (Webberley & Preece, CENode, 2016) present two options, the latter has been chosen due to the greater availability of local support and the ability to develop an integrated and extensible platform.

The goal of the project is to determine if a conversational interface using controlled natural language provides a feasible, effective, and user-friendly way of interacting with internet of things devices such as Mother and the Cookie sensors, as contrasted with the current web/mobile applications. The project aims to assess whether there is merit in allowing users to have a conversation with an environment that talks back.

Background

This section aims to provide background information about the technologies used in the development of the application, as well as some information on how they work in relation to the project in addition to some reasons behind their inclusion. As discussed previously the aim of the project is to produce a working method of interacting with IoT devices (specifically Mother) via controlled natural language.

Internet of Things & Mother

The internet of things refers to the network of physical objects embedded with sensors, software and networking capability – enabling them to collect and exchange data on behalf of the user. The internet of things encompasses technologies such as smart homes, intelligent interconnected transport, smart cities etc. These devices would allow for the automation of low-level tasks on behalf of the user and the ability to gain new insight into how such devices are used at a fine level of detail. The applications for IoT technologies are varied, including but not limited to media, infrastructure management, environmental monitoring, manufacturing, home automation, medical care, transport, and energy management. The ability of devices in these sectors to communicate, sense and influence their world can be used to improve efficiency, accuracy and economic benefit surrounding the use of such devices.

Currently most interaction with any IoT devices or services is accomplished through either an online or mobile application. These applications typically take the form of a dashboard with readouts and statistics for each device or a simple application designed to operate the basic functions of the device and receive notifications. The appeal of the new approach that is the subject of this project lies in the fact that tasks, both complex and simple, can be accomplished and expressed in a form of natural language. This gives the user a powerful method of interaction that is immediately familiar and easy to use.

Mother is typically used in conjunction with its Cookie sensors throughout a user's home to collect telemetry based on the application currently assigned to a Cookie and its placement in the home. There are a variety of applications available for Mother and its Cookies; keeping track of fitness, monitoring room temperature, tracking medication usage, presence detection, motion detection on object, coffee or water consumption etc. The Cookies have the ability to measure temperature and motion via an integrated thermometer and accelerometer, as well as being capable of presence detection based on connectivity to the Mother hub. This means that data from each Cookie is interpreted differently depending on its application and placement in the home. Since Cookies are versatile, changing role depending on application and placement, a user can make use of them to monitor anything that can be kept track of via any available application on Mother's Senseboard (Figure 2) (Sen.se, 2016).

Data from the Sense API is generated in discrete events with these events being organised into feeds. Feeds are channels in which events of the same kind flow, Cookies have both motion and temperature feeds, for example, which contain all the measured temperature and motion events for that Cookie. Additional feeds are available depending on the applications the Cookie is registered with. The frequency and detail of data events are based on the type of telemetry they contain, in addition to the applications installed via the online dashboard. As an example the temperature feed of a Cookie is updated every 15 minutes, the presence feed every minute, and the motion feed as soon as a motion event occurs and then in gradually increasing increments over the duration of the motion. Each of these event types contain different information with application-specific data included in some types of event such as sleep or fitness tracking.

The availability of data from the Sense API is based on applications (Figure 2) registered with Mother via an online dashboard provided by Sense when logged in with an account. The applications registered with a particular Mother and associated Cookies determine the kind of data that is available and what events are generated by the API. The act of registering a Cookie with an installed application via the dashboard determines how the data collected by Cookies is interpreted and in turn which events will be generated by that Cookie's feeds. Motion data collected from a Cookie, for example, can be used to determine the sleep patterns of a user, if the user has taken medication, or the amount of water consumed in a given period depending on the applications registered against each Cookie and its placement in a user's home.

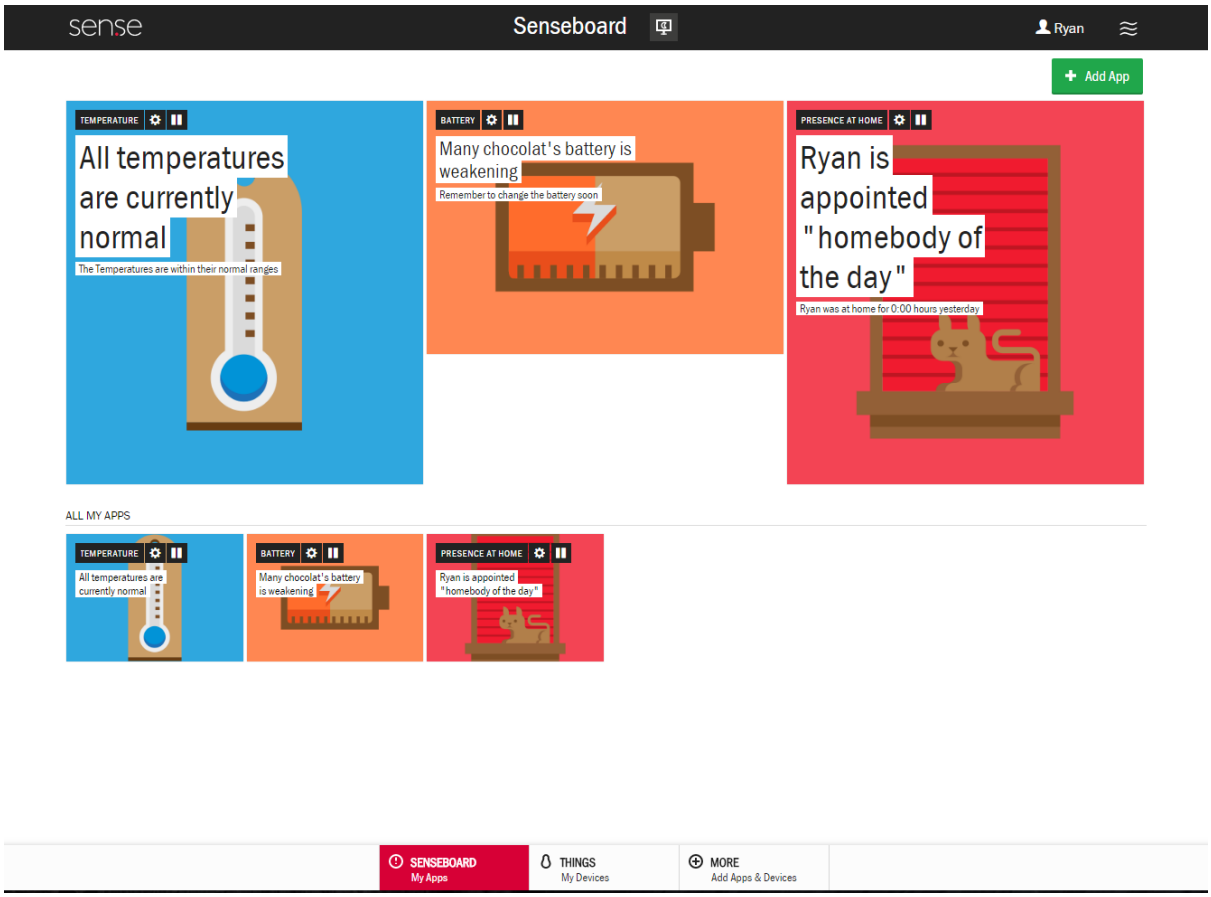

Figure 2: Senseboard with installed applications

Controlled Natural Language

Controlled English (CE) (IBM, 2016) is a human-readable/machine-readable form of controlled natural language developed by IBM in order to enable unambiguous human-human/human-machine/machine-machine communication in a manner which is relatively easy to understand and learn, and is more accessible to users. The appeal of controlled English lies in the conversational nature it affords interactions with services or other software, giving users a method of interaction which is already familiar. This conversational interaction also allows a system to present complex or raw data to the user in an easily digestible form (Preece, Braines, Pizzocaro, & Parizas, 2014).

There are many examples of popular services which allow users to interact in a conversational manner, Google Now, Apple's Siri, and Microsoft's Cortana make it clear that users prefer a method of interaction which is as natural as possible. Indeed, the relatively recent announcement of bot frameworks for the communications platforms Skype (Skype, Microsoft Corporation, 2016) and

Facebook Messenger (David Marcus, Facebook, 2016), combined with the integration of bots into other popular chat platforms such as Slack, shows the popularity of this kind of interaction between user and service. This method of interaction also does not require that a user leave their current context to switch to a separate, dedicated application. The personality imbued upon the digital assistants and services mentioned above also support the conclusion that users enjoy interacting with "friendly things".

User preference is an important factor when using natural forms of interaction such as natural language. Renewed interest in the use of natural language-based interfaces in the mobile computing industry (the digital personal assistants mentioned above, for example) is largely thanks to research conducted in these areas by the artificial intelligence community (Allen, et al., 2001). Users prefer to use these natural interfaces due to their ease of use but also because of their novelty and pleasurable experience (the 'joy of use' phenomenon) (Weiss, Wechsung, Kuhnel, & Moller, 2015).

While natural language interaction remains a computationally hard problem, the use of controlled natural language aims to address this by providing a restricted subset of natural language that is unambiguous, user-friendly and less complex for machines to process. These controlled natural languages are beneficial due to their human and machine processability, meaning they can easily be used as a human-readable method of formal knowledge representation (Kuhn, 2014). This common capability between machine and human communication makes controlled natural language ideal for interaction between human and software agents, including IoT devices such as Mother.

CENode

The project makes use of CENode (Webberley & Preece, CENode, 2016), a JavaScript implementation of the aforementioned controlled English processing environment. This maintains and updates a knowledge base, and reasons over rules and statements submitted in controlled English. Upon instantiation this node is provided a model which conceptualises the entities related to the Mother application, their properties and how they are related to one another. This model also contains rule statements, complete with conditional antecedents and consequents to be evaluated when performing reasoning about the information received. Although this model is interpreted by the node when instantiated, it is also possible to dynamically alter the model during runtime, providing new concepts, properties and relations to the node's knowledge base.

CENode is also capable of automatic interaction with other CENode nodes on a network via implemented policies, given as controlled English statements in the model or defined during runtime. These policies can tell a node to inform all other nodes via HTTP when a statement is added to its knowledge base, for example (Webberley, Preece, & Braines, 2015). This is the basis of the interaction between the constituent parts of the SHERLOCK (Simple Human Experiments Regarding Locally Observed Collective Knowledge) game (Preece, et al., 2015).

CENode makes use of a blackboard architecture and cards (Webberley & Preece, CENode, 2016) to pass information between agents and users (Figure 3). All user-agent/agent-agent interaction is accomplished via these cards which contain controlled English content and attendant data such as timestamps, who the card is from, who the card's intended recipient is etc. A card is intended to act as a container to describe the content (natural versus controlled natural language) and type ("tell" or "ask" cards, for example) of a statement. Indeed, as is described later, communication between the user and node in this application also makes use of these CE card objects.

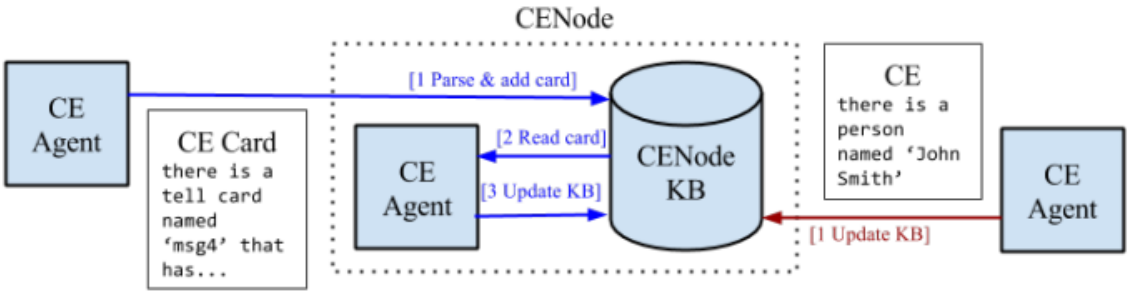

Figure 3: CE Cards used for agent communication (Webberley, Preece, & Braines, 2015)

Approach

This section aims to provide an overview of the originally planned approach to developing the system, as well as the eventual approach that was taken, including reasons behind changes made to the planned implementation.

The development methodology selected for the project makes use of several applicable strategies from the agile methodology, with particular emphasis on sprints with a deliverable minimum viable product, and regular meetings with the supervisor to discuss progress and problems. For the duration of the project, a sprint length of two weeks with regular supervisor meetings was selected. This quick sprint cycle ensures that each sprint is given a specific area of development to focus on and allows for immediate feedback on work to identify areas for improvement or change. Some areas of the agile development methodology are not applicable to a single-person development team and were modified accordingly, for example daily scrum meetings were replaced with regular brainstorming sessions to determine the direction of development and how to proceed.

There were a total of four two-week sprints through the duration of the project. Each sprint had a particular area of focus for development, concentrating on a particular feature or set of features in the application.

- Sprint 1
  The first sprint was focused on initial development, testing with the Sense API and CENode library. The goal of this sprint was to investigate the capabilities of the CENode library and Sense API and to determine how best to approach the problem. The revised system architecture was designed.
- Sprint 2
  After the subscription-based approach was selected, this sprint focused on the development of a simple web application to receive event updates from the Sense API. The goal of this sprint was to provide a functional web endpoint for the subscription mechanism to contact and a client-side webpage for interacting with the server-side instance of CENode. After the client-server interactions surrounding CENode were functional, work was done to alleviate compatibility problems introduced by this new client-server architecture on the part of CENode.
- Sprint 3
  After the web application was functional and able to receive HTTP POST requests from the Sense API, the goal of this sprint was to have the events parsed and added to the CENode knowledge base. This sprint was also the point where it became apparent that extensive modifications would need to be made to CENode to support the data generated by Mother. A large section of this sprint was dedicated to the parsing of CENode knowledge base queries to support the stateful querying of objects.
- Sprint 4
  The focus of this sprint was to enable the correct functioning of complex rule statements and allow the node to infer relationships and properties based on the information provided by the user and Mother.

This development methodology was selected due to its high tolerance for shifting requirements and ability to quickly adapt to changing circumstances in the development. This methodology allowed for some anticipation for the presence of unknown unknowns during the development cycle and afforded enough elasticity to deal with them appropriately.

Requirements

There are several functional and non-functional requirements that are used to judge the success of the implementation. These requirements concern the technical implementation of the solution and are separate from the aim of the project. The requirements are as follows:

- As a user I can submit queries using controlled English via text input
- As a user I can receive output from the application in the form of controlled English text
- As a user I am able to issue queries about the data Mother is capable of collecting
- As a user I receive up-to-date information when submitted a query
- Invalid queries should be handled gracefully
- Loss of connectivity should be handled gracefully
- Data delivered should be accurate

These requirements ensure that the application is able to perform adequately, taking input and giving output in the form of controlled English. The requirements also ensure that the application is able to retrieve information from the Sense API about Mother and that this information is current and correctly integrated into the node's knowledge base, enabling the user to issue queries about the state of things Mother is capable of monitoring. The requirements also make certain that any loss of connectivity or invalid input on the part of the user is handled appropriately and does not disrupt the application.

System Architecture

The initial approach to the project consisted of making use of the Sense API on an as-needed basis to retrieve information relating to the user's query. This would have been done using word recognition to pick out important words from the submitted controlled English statement and use them to derive an appropriate query for the API. The initially proposed system architecture (Figure 4) consisted of an application which would lie between the CENode and Sense API, taking controlled natural language statements or JSON as input, parsing, and then submitting controlled natural language or HTTP requests to either the CENode or API in response.

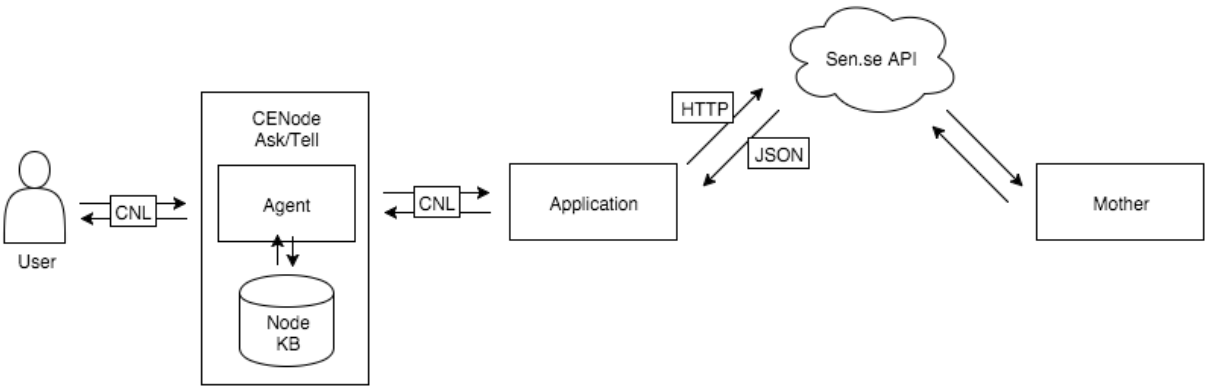

Figure 4: System architecture as initially planned

In actuality, the architecture of the developed system differs significantly from the proposed solution (Figure 5). A new approach to interaction with the API was taken, in favour of the initially planned one described above. This new approach involves the creation of a subscription, a service offered by Sense which sends JSON (ECMA International, 2013) encoded events to a defined web endpoint upon generation via HTTP POST. It is possible, for example, to create a subscription which monitors all feeds generated by every Cookie.

This revised interaction approach shifts the focus of the application to CENode. This push-based subscription approach was selected over the ad-hoc approach that was initially planned, as it allows the node to reason over the most complete set of information available at the time of each query. This is made possible by the fact that any new data events created by the API are immediately pushed to the application and added to the node's knowledge base.

This approach also removes a significant amount of complexity in the system surrounding how and when to pull information from the API since this is taken care of automatically by the subscription mechanism. It is also more favourable in terms of performance as it halves the number of web requests that need to be made by the application to the API.

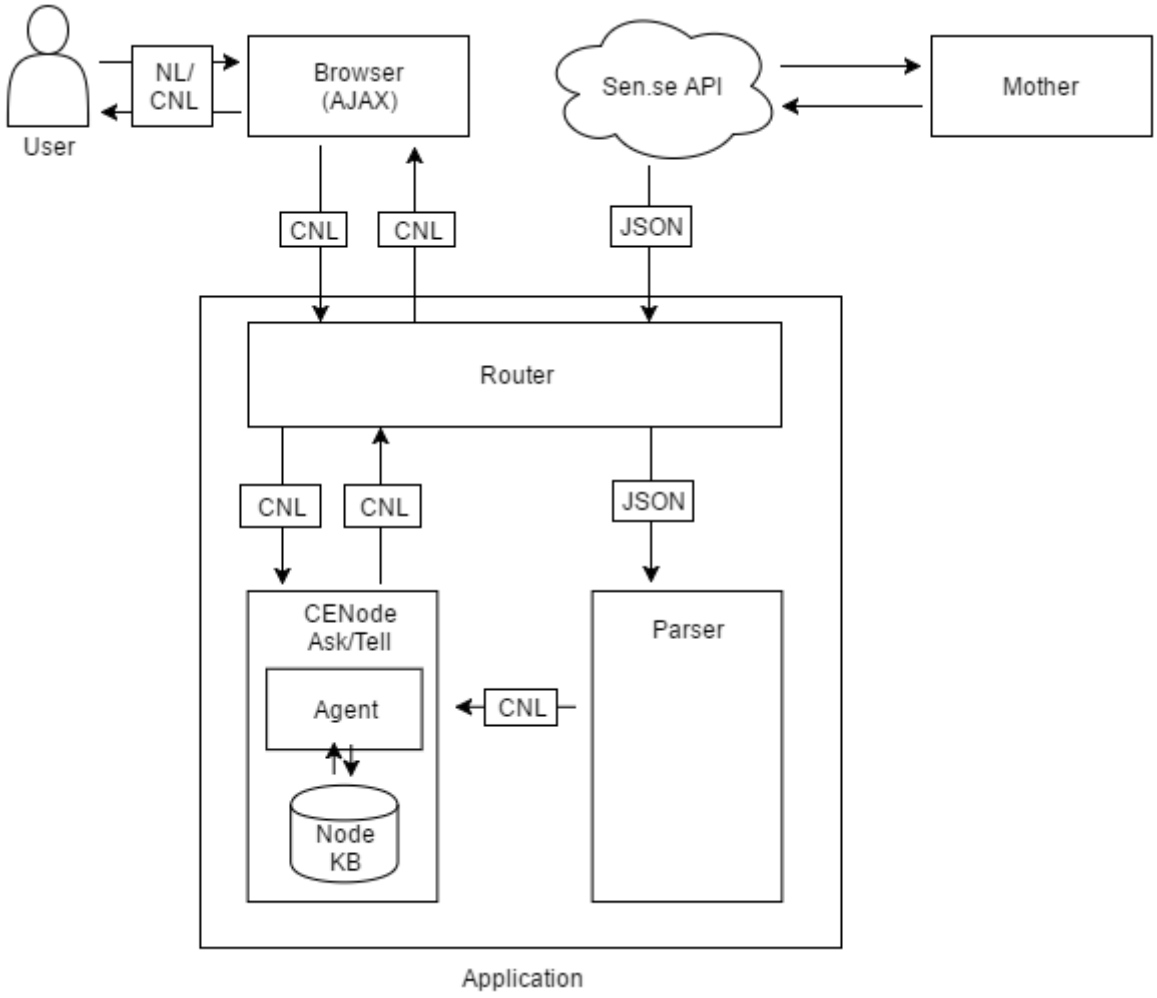

Figure 5: System architecture as developed

Due to the web-based nature of the subscription mechanism, the application requires a web-accessible endpoint to receive events relayed by subscriptions. This requires that the application take the form of a web application, consisting of a server-side environment to receive the subscription events and house the node implementation, and a client-side page for interaction with the node.

The user submits CE queries to a client-side web page which are then sent to the server via AJAX (Mozilla, 2016) and routed to the node for processing. In order to retrieve responses from the system the client-side web page continuously polls the server for all CE card instances, which are

sent via AJAX in the form of an array of JSON-serialised CEInstance objects (Webberley W. M., CENode API Documentation, 2016).

The page then determines if the cards have already been seen by the user. If a card has not been seen by the user, it is parsed, the content field is extracted and is then presented to the user. All responses take the form of controlled English statements which are presented as replies from the system. Statements added to the node's knowledge base are subject to any rule instructions present in the model upon their parsing by the node. Since any JSON is parsed on both sides of the client-server interaction, the application-level protocol is actually CNL.

User interaction with the system takes the form of a conversational interface provided by a web page, where statements and questions by the user, together with responses from the system, are presented as text within a chat-style messaging system. Since this communication is conducted asynchronously via AJAX, the application is able to achieve a consistent real-time chat-style interface without requiring page reloads or navigation.

Mother API

Data is acquired from the Sense platform via a REST-oriented (Rodriguez, 2015) API and returned in JSON format. Users are authenticated with the API via an API key included in the "Authorization" HTTP header with each request. Requests to the API are throttled up to 100 requests per minute from each distinct IP address for an account. Responses from the API are paginated, with each page containing the total number of object and, if applicable, links to the next and previous pages.

The API exposes a number of endpoints, each focusing on a particular concept associated with the Sense platform. There are five concepts in total:

1. Resource
   A client application that is interacting with the platform.
2. Node
   An instance of a resource for a particular user. Any device or application registered to a user's account is a node. Each node produces events.
3. Feed
   A feed is a channel in which events of the same type flow. Cookies have a motion and temperature feed, for example. Feeds are addressed using either a UID or the URL of a node with a feed type. For example, `/feeds/(UID)` or `/node/(UID)/feeds/(type)`.
4. Subscription
   A subscription is a mechanism that notifies a system when events are published to feeds which are subscribed to. A subscription must be provided with a list of feed UIDs or URLs to subscribe to and a callback URL which is used to tell a system via HTTP POST when a subscribed feed is updated.
5. Person
   Members of a household are represented as Persons on a Sense account. There is always at least one person associated with an account. Persons are also nodes.

There are restful endpoints (Sen.se, 2016) associated with each of these concepts for getting, updating and deleting. Endpoints vary by concept and some operations are only supported for certain concepts (for example only subscriptions can be updated or deleted). The application primarily makes use of the node and subscription endpoints to discover nodes and manage subscriptions.

The concepts that the application makes use of are modelled using CE in the domain model, loaded on application start-up. Feed events are modelled as property values attached to any Cookie instances or other entities. A subset of nodes is modelled as Cookies within the application, these modelled Cookies are the first concept instances with which newly generated feed events interact. The application receives feed events and attaches them to the originating Cookies via the node UID; this is modelled as Cookie instances with measured properties as values (Figure 6).

CENode Library & Domain Model

When the application is started, the default domain model (Appendix 3) is loaded which defines concepts and relationships between objects. The default domain model attempts to model the relationships between Cookies and the things they are capable of monitoring. One of the main advantages of this CE-based modelling approach is that users can edit the model during runtime or the existing domain model can be edited before it is loaded on application start-up.

The default model defines only those relationships, properties and entities surrounding simple Cookie measurement such as temperature, motion, and presence detection. The model could be modified to support measurements collected from additional installed Sense applications (sleep patterns or water consumption, for example). The entity relationship diagram below describes the default model.

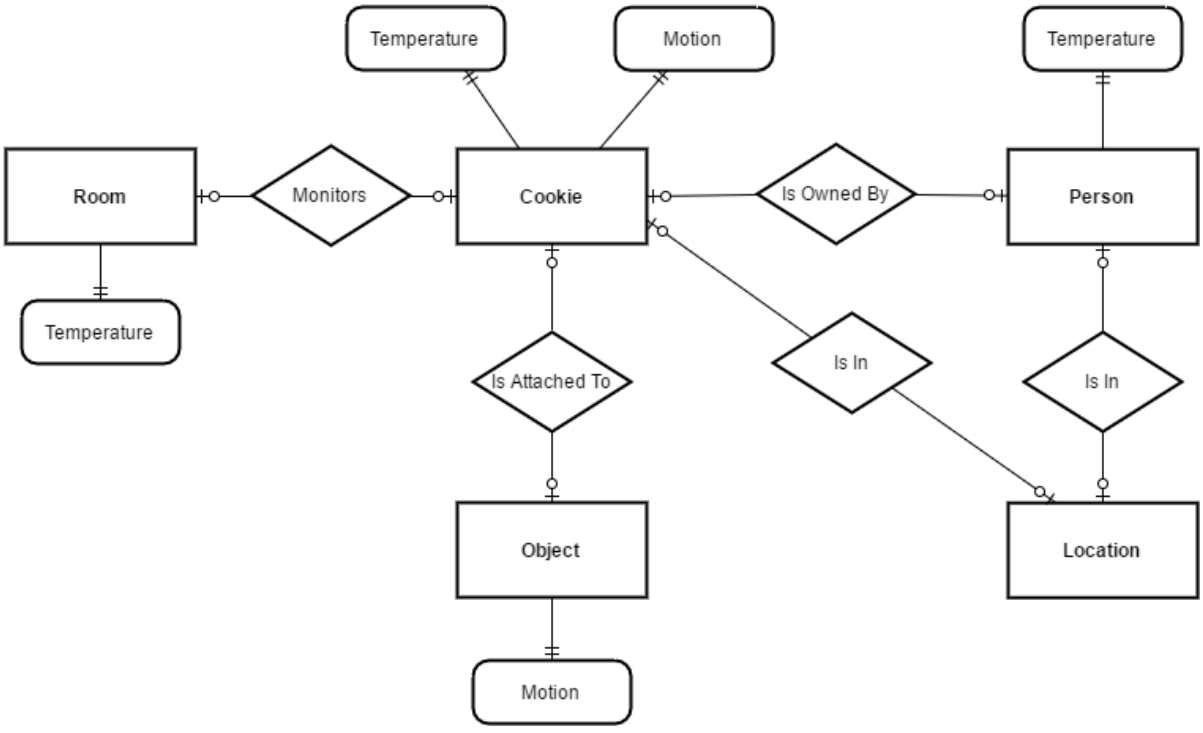

Figure 6: Entity relationships specified in the default model

When these optional relationships are established between a Cookie and any of the corresponding entities, a transitive relation is created between the properties of the Cookie and the properties of the related entity. These transitive relations are possible due to the evaluation of rule statements in the node.

The entity relationship diagram above is intended to describe the model at a conceptual level, in reality the actual CE model differs in several ways. Due to the monotonic nature of the node's knowledge base, the CE model is not capable of cardinality – when querying the model, all relationship targets and property values attached to an object instance over the lifetime of the node are returned. The properties and relationships expressed in the diagram are shown as having only one target as this is how the model is intended to work conceptually. These individual measurements and discrete state are required to effectively model the inherently stateful data generated by Mother. This is discussed further in the Implementation section.

It is this domain model which allows the user to query the application. The domain model allows the node to reason about its world and make inferences based on rules. The default model first lays out concepts that are required, these concepts represent the entities to be involved in the system, for example Cookie, Person, Room etc. These concepts are then expanded to include relationships with other concepts (for example, a Cookie has the "is in" relationship with a location) and any properties the entity has, a Cookie has temperature and motion properties, for example.

The model then defines any rules that are needed to reason about data collected. These rules primarily establish transitive relations between the properties and relations of a Cookie to its attached entity (Person, Room, Object etc.). These rules also work to ensure that any relationships are bi-directional, i.e. if a relationship is established from Cookie A to Object B, Object B also has the inverse relation to Cookie A. For example, the "owns" and "is owned by" relation between Person and Cookie.

Finally, the model defines any object instances that are needed for the application, in this case a set of named Cookie instances for the application to interact with. Ideally these Cookies would be automatically discovered and instantiated from the nodes provided by the Sense API, this is discussed further in the Future Work section.

Implementation

This section aims to provide an overview of the implementation of the project, including challenges faced during development, how these challenges were overcome and how they impacted the design of the system.

The application is implemented in JavaScript, making use of the NodeJS runtime and Express web framework for server-side routing and error handling, along with the Jade templating language and JQuery library to serve client-side web pages with dynamic content.

CENode supports two methods of interaction, a programmatic API which uses a JavaScript instance of a node to interact via function calls, and an HTTP API which uses a self-contained server to interact via HTTP accessible endpoints. JavaScript was chosen as the implementation language due to the greater availability of functionality and documentation for the programmatic API of CENode (Webberley W. M., CENode API Documentation, 2016) as opposed to the HTTP API.

Interaction with the node via the programmatic API is favourable over the HTTP API as the HTTP API currently only supports minimal interaction with the node and its components, and would require a separate running instance of a NodeJS app. This is due to the fact that upon instantiation the node attempts to automatically determine its runtime environment and prepares for the appropriate methods of interaction, for example, if running as a NodeJS app, it will start a new HTTP server and listen on endpoints described in the API documentation.

This requirement of a separate NodeJS server instance in order to run the RESTful (Rodriguez, 2015) version of the node is unfavourable due to the complexity surrounding the management, running, and continued availability of the separate server instance when interacting with the application's own instance. However, this also means that the node's automatic compatibility with existing CENode systems via policy implementations is not possible. Clearly it is desirable to maintain compatibility between this application and other instances of CENode running on the network to enable expected functionality concerning policies. A solution was implemented, as described below, which attempts to alleviate some of these issues.

All information stored in the node's knowledge base takes the form of controlled English statements. As such, any concepts or relationships that are to be modelled must be given in the form of controlled English, either in the model which is loaded when the node is created or later through the interface. New concepts and relationships in the node's knowledge base are stored in the form of a domain model, which itself is defined using `conceptualise` statements (IBM, 2016). These statements allow us to construct a model to reason about the data supplied by the Sense API. Reasoning is described using rule statements, which allow the node to infer new facts based on input, that have not been explicitly entered by the user.

The application was designed and implemented with a view toward extensibility, with a primary focus on enabling the system to support a variety of rules, models and events generated by Mother. The main idea behind this was to enable easier future development and allow the system to easily cope with new event types (depending on Mother's installed applications) by editing the domain model and parser. Since the domain model is constructed entirely using controlled English, it can be easily updated to accommodate any new concepts, relationships or rules that are needed to handle new event types. The parser also makes use of a simple mapping between event types and JSON field names to parse events. This mapping can be easily updated to work with additional event types. This is discussed further in the Future Work section.

The following is a sample of a JSON-serialised presence feed event sent to the application:

```
{
      "profile": "WalkStandard",
      "feedUid": "15cHJBICKx0DBd7facfaZCaYvuv90yxx",
      "gatewayNodeUid": "NZP8tSrF4N0eXd6E2gpRrmJX0pJ3pPb9",
      "dateServer": "2016-02-11T07:52:47.686919",
      "geometry": {
            "type": "Point",
            "coordinates": [51.5199, -3.204]
      },
      "data": {
            "centidegreeCelsius": "2440",
      },
      "signal": "-69",
      "dateEvent": "2016-02-11T07:52:47.000000",
      "type": "temperature",
      "payload": "1",
      "nodeUid": "78m6qJPRmq7tdqyGTAJoNv2Kgk0npNXr"
}
```

Figure 7: Temperature event sample

The application makes use of the `data`, `type` and `nodeUid` fields when parsing the received event to construct a controlled English statement, determining the originating Cookie, type, and data payload of the event. This statement is then submitted to the node for processing.

The mapping between an event type and its attendant data field in the JSON object are described with the following object in the parser:

```
var eventMap = {
  presence: "body",
  temperature: "centidegreeCelsius",
  motion: "avgIntensity"
};
```

This data structure combined with the versatility of the domain model approach allows the application to be easily updated to accommodate new event types and Senseboard applications.

When a new subscription event is received, the application uses the fields in the event JSON to construct a controlled English statement representing the content of that event. As an example the temperature event above would be parsed, and a statement would be constructed as follows:

```
“The Cookie ‘Lively Fruit’ has 2440 as temperature.”
```

This statement has been constructed using the data payload of the event, the event type (temperature in this case) and the unique identifier for the originating node, which is then mapped to a Cookie name (lively fruit in this example). A mapping of node UIDs to Cookie names is created when the application starts by retrieving a list of all nodes from the Sense API and populating the map with nodes which are of type Cookie.

After CE statements have been entered into the node, the knowledge base can be queried. Queries can be asked of concepts or instances present in the node's knowledge base. The most common types of query are "who", "what" and "where" queries. These queries give summaries of objects and return their location respectively. When parsing the "who" and "what" questions, they are treated identically by the node, the only difference lies in the grammar used when querying people as opposed to objects.

The current state of an object can also be obtained by appending the word "now" to the end of each query string (for example, "`where is ryan now?`" would return the most recent recorded location of the Person instance named Ryan). An additional query type exists to return all the named instances of a concept; this is accomplished by asking "`list instances of type X`" where `X` is the concept in question.

When responding to the query types listed above, the node generates brief summaries (known as gists) of concepts and instances. In the case of concepts, these summaries contain descriptions of any relations and properties present on the concept and how these relations interact with other concepts. As an example, the following gist is generated when asked what a Cookie is:

> "`A cookie is a type of locatable thing. An instance of cookie has a value called temperature and has a value called motion and is attached to a type of object and monitors a type of room and belongs to a type of person.`"

In the case of object instances (for example, lively fruit), the node generates a gist describing any property values or relationships that have been attached to the instance over the lifetime of the node. As an example, the following gist is generated when asked what lively fruit is:

> "`lively fruit is a cookie. lively fruit has '2480' as temperature and has '2560' as temperature and has '2440' as temperature and is in the location 'not home'.`"

When CE statements are entered into the node for processing, they are subject to any rules present in the node's knowledge base. Rules are given in the model as CE statements, taking the form of "If P then Q" conditional statements. As an example, the default model defines a rule to set up the "belongs to" relationship between a Cookie and a Person if the inverse relation already exists between a Person and a Cookie:

> "`There is a rule named r1 that has 'If the Person P ~owns~ the Cookie C then the Cookie C ~belongs to~ the Person P' as instruction.`"

When CE statements are added to the node, the statement's subject and object are checked against rule statements to determine if any new relationships or properties should be established. If the following statement were then added to the node:

> "`The person 'ryan' owns the cookie 'lively fruit'.`"

The subject of this statement is an instance of Person named 'ryan', while the object is an instance of Cookie named 'lively fruit'. Upon receiving the subject and object, the node will parse the rule statement above and check if there is a relationship named 'owns' on the subject whose target is of type Cookie. If these conditions are met, the relation is stored in an object for later reference. This process is repeated for each antecedent present in the rule statement.

After all relationships on the subject (including the subject itself) are found, the node checks to make sure the rule's consequent includes a reference to the object of the statement, and that all entities mentioned in the consequent are related to the subject. If all these conditions are satisfied, the node establishes the relationship specified in the rule's consequent on the proper entity. A high level view of how rule statements are enacted upon processed CE is described in the activity diagram below.

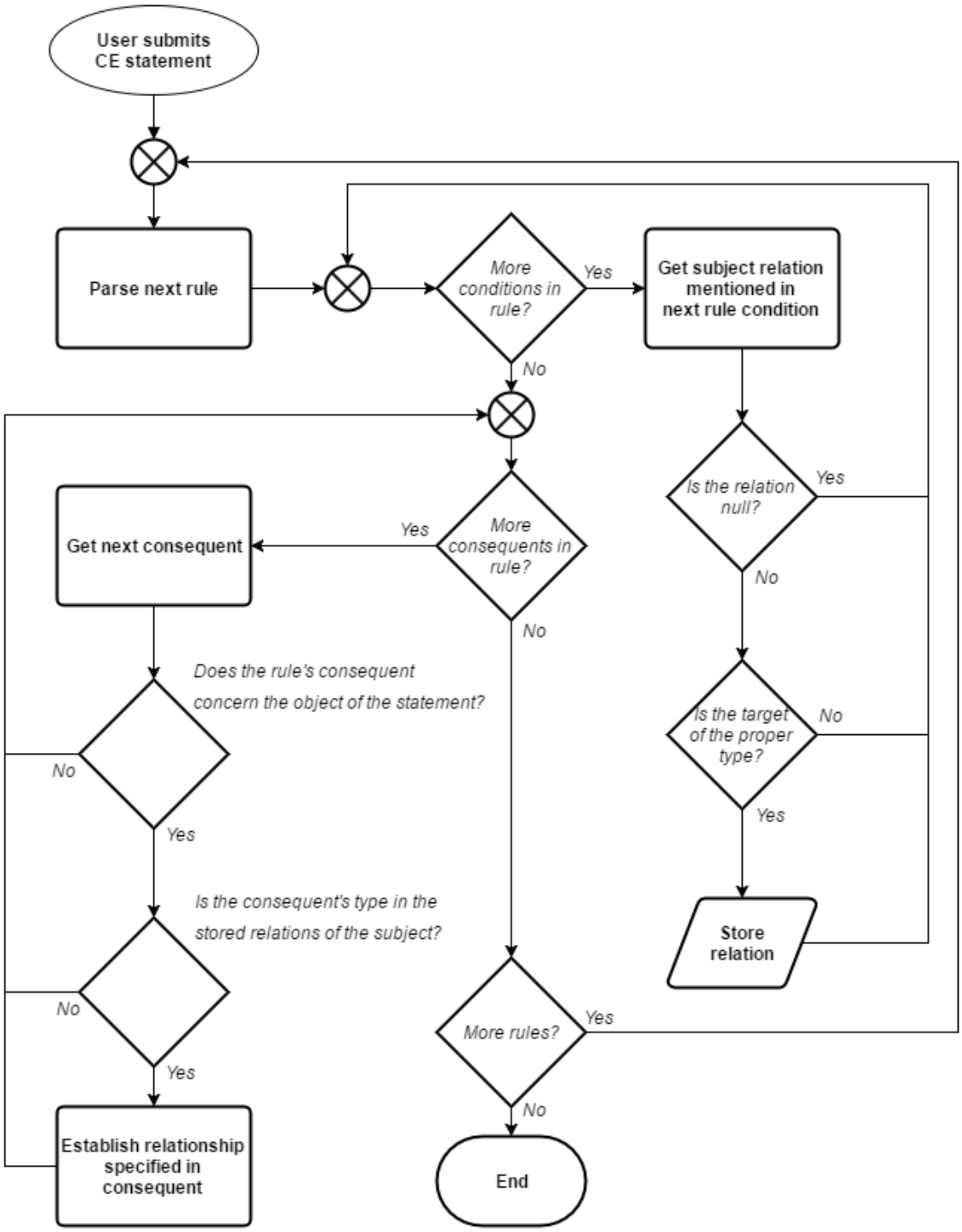

Figure 8: High level activity diagram for enacting rules

Challenges Encountered During Development

The following subsections describe the main problems encountered during development and outline the steps taken to solve them. These problems made up the bulk of development time and accounted for a large portion of the work conducted on the project. Since these problems primarily concern the manner in which CENode processes and reasons about controlled English statements, solutions were necessary in order to develop the system to a state where it is able to perform its function correctly and is ready for user testing.

Problems with Monotonicity

As mentioned earlier (Approach), the implemented system relies on receiving newly generated subscription events from the Sense API. When these events (Figure 7) are received they are parsed and a controlled English statement is constructed to reflect the event. These statements are then submitted to the node for processing. It is during this time that the node applies any rules concerning the subject and object of the CE statement – usually to form or update relationships or properties of any concept instances as needed.

When these constructed sentences are submitted to the node they are parsed and added to the node's knowledge base. This knowledge base is monotonic in nature and as such presented some problems when attempting to ascertain the current state of any instantiated concepts. The node has no concept of discrete state or deletion of facts, as such any information added to the node is permanently available, attached to its instance or concept for the lifetime of the node. This is especially challenging since data collected from Mother is inherently stateful, measuring the current condition of an object at the time of the event generation. The original CENode implementation had no way of ascertaining the current state of an instance, instead returning every fact entered over the lifetime of the node when queried.

This incompatibility is due to the design of the node to accommodate the SHERLOCK application, taking into account potentially conflicting crowd-sourced information. For example, the location of one of the game's persons of interest could be reported differently by different players. The node allows this, adding both conflicting statements to its knowledge base and including the number of times each statement has been entered. This is useful in the context of the SHERLOCK game as it allows us to give each statement a greater probability of truth if there is a proportionally greater number of people claiming it.

This monotonicity also creates problems when evaluating rules, in that the entire history of an object is taken into account when enacting rules on submitted statements. This behaviour is undesirable for the Mother application and was worked around by building a way to ask the node to provide the most recent state of an instance, which is then assumed to be current.

As the source code for this shows (Appendix 1), this is accomplished by iterating over the distinct relationships or values found on an object and making use of the `property` function (Webberley W. M., CENode API Documentation, 2016) to acquire the most recent value or related CEInstance object, then constructing a response string in the same style as the existing gists that are returned by the node. This is how the node is able to respond to questions specifically relating to the current state of an object instance, for example "`What is lively fruit now?`" would return a CE statement giving the most recent state of all values and relationships on the Cookie named Lively Fruit. There has been a similar modification to the parsing of "where" questions to return the most recent recorded location of an object instance when asked.

Compatibility with Policies & Existing Implementations

As mentioned earlier, the need to interact with the node via the programmatic API means that existing implementations would not be able to interact with our node via policies due to the manner in which the node exposes its HTTP endpoints. In order to retain some measure of compatibility with existing CENode implementations that make use of these policies, HTTP endpoints identical to those found in the HTTP API for CENode have been implemented in the application. These endpoints make use of the same HTTP verbs and URLs as those listed in the documentation and simply forward requests to, and return responses from, the appropriate node functions using the programmatic API.

There are however some compatibility issues that remain due to the serialisation of responses for transmission over HTTP. As mentioned earlier, communication with nodes is conducted using CE cards (Figure 3). These cards, and indeed all concept object instances in CENode use JavaScript's `Object.defineProperty` to define relationships on the object (for example, in the case of CE cards passed between nodes the '`is_to`' and '`is_from`' relationships etc.) which means that, in this application, these 'helpers' (Webberley W. M., CENode API Documentation, 2016) are not present when the CE card object is serialised and sent to the client (Figure 9). This is contrasted with the objects from a locally stored node, where the defined properties are still intact (Figure 10, shown faded).

This problem is not present in other implementations of CENode as the node is stored locally and concept instance objects are never serialised, meaning that all defined properties are readily accessible.


```
{
      "name": "Moira49",
      "id": 16,
      "type_id": 10,
      "sentences": [
            "there is a nl card named 'Moira49' … as content"
      ],
      "_values": [{
            "label": "timestamp",
            "type_id": 17,
            "type_name": "1460122719355"
      }, {
            "label": "content",
            "type_id": 0,
            "type_name": "there is a person named 'ryan'."
      }],
      "_relationships": [{
            "label": "is to",
            "target_id": 18,
            "target_name": "Moira"
      }, {
            "label": "is from",
            "target_id": 19,
            "target_name": "1460122711609"
      }],
      "_synonyms": []
}
```


Figure 9: Shortened CE card sample as sent to client

Figure 10: CE card sample from local CENode instance, Helper properties intact, shown faded

This compatibility problem was worked around in the application by taking values from the object's private fields (as denoted by property names beginning with an underscore (Figure 9)) as these are included when an object is serialised. This solution is not perfect however, as it still requires some work on the receiving side to gather the information without the use of the object's defined properties. This is discussed further in the Future Work section.

Parsing & Enacting Complex Rules

Another significant problem to overcome was the evaluation and enacting of multi-antecedent and multi-consequent conditional statements against submitted controlled English. The original CENode implementation supported simple single-antecedent/single-consequent rules to establish relationships between instances of concepts. It was possible, for example, to write a controlled English rule statement:

> "There is a rule named r1 that has 'if the cookie C ~ belongs to ~ the person P then the person P ~ owns ~ the cookie C' as instruction."

This rule would be correctly enacted and establish the corresponding relationship (owns, in this case) on the object (an instance of type person) upon the processing of the following CE statement:

> "The cookie 'lively fruit' belongs to the person 'Ryan'".

This simple rule parsing and enacting would not, however, support more complex rules regarding multiple antecedents or consequents, and likewise would not support evaluation of statements about values as opposed to relationships. This problem was overcome by building a new rule parsing logic and rule enacting function (Appendix 2) that would support these complex rules and correctly apply their consequents to the proper object. In some cases, this is not necessarily the object of the submitted controlled English statement. The primary modifications to the parsing logic for rules allows for multiple antecedents and consequents to be given in the rule's instruction by adding them to an array of 'if' or 'then' conditions in a rule object (Figure 11) after regex parsing of the instruction string.

The new rule enacting logic (Figure 8) takes into account the manner in which controlled English statements are parsed to correctly establish relationships and values on objects mentioned in the rule instruction. When statements are submitted to the node for processing they are evaluated against all rules present in the knowledge base. Parameters passed to the function which enacts these rules (Appendix 2) include the subject and object of a submitted statement, along with the type of property the function should focus on (either relationship or value).

As an example, if the following rule were present in the knowledge base:

> "There is a rule named r2 that has 'if the cookie C ~ belongs to ~ the person P and the cookie C ~ is in ~ the location L then the person P ~ is in ~ the location L' as instruction."
>
> *This rule is intended to infer the presence of a person based on their ownership and presence of a Cookie.*

Then, if the following statements were submitted to the node in the given order, assuming all named instances of Cookie, Person, and Location have already been conceptualised and declared:

> "The cookie 'lively fruit' belongs to the person 'Ryan'."
>
> "The cookie 'lively fruit' is in the location 'home'."

When processing the second statement, the function would receive as input the subject of the statement (lively fruit) and the object (home). The function then iterates over all rule statements, parsing each rule to determine the antecedents, consequents and any concepts or relationship mentioned. Upon parsing the rule statement given above, a rule object would be created as follows:

```
{
      "if": [{
            "concept": "cookie",
            "condition": {
                  "type": "person",
                  "label": "belongs to"
            }
      }, {
            "concept": "cookie",
            "condition": {
                  "type": "location",
                  "label": "is in"
            }
      }],
      "then": [{
            "concept": "person",
            "condition": {
                  "type": "location",
                  "label": "is in"
            }
      }]
}
```

Figure 11: Rule object sample

After confirming that the current rule being evaluated could potentially effect the object of the statement being processed, the function then gathers the most recent targets for all condition labels found in the rule object's '`if`' array on the subject in question. These relationships or values are then added, along with the subject instance itself, to an object, to keep track of any concepts or values which are related to the subject. This ensures that any objects found within this object are definitely related to the subject via the relation mentioned in the rule (i.e. "`~ is in ~`") and are of the proper type (i.e. "`location`").

The function then moves to evaluate any consequents found in the rule object. The function makes use of the collected relations or values on the subject to determine if:

- The object of the CE statement currently being processed is related to the subject via the relationship or value name found in the rule object
- The consequent of the rule applies to the object, either via the concept or condition type properties
- The subject is related to any objects mentioned in the rule's consequent

Only after confirming these conditions, does the function establish the relationship described in the rule object's consequent on its concept.

Continuing with the example given above, after parsing the rule and determining its applicability to the current statement, the function would generate the following for the subject 'lively fruit':

```
{
      "cookie": {CEInstance},
      "person": {CEInstance},
      "location": {CEInstance}
}
```

This example has been shortened, each key in fact contains the entire CEInstance object that is the target of these relationships.

This is the object which keeps track of relationships and values found on the subject which are mentioned in the rule object's '`if`' array. When evaluating the consequent to satisfy the conditions mentioned earlier, we can be sure that any CEInstance objects present must be related to the subject via the proper name and type. It is also this structure which allows for the establishment of relationships mentioned in the rule even though the object of the statement being evaluated may be of a different type. Since this structure stores the entire CEInstance object we are able to establish a relationship or value on that CEInstance as long as it is properly related to the subject as described by the rule, which it must be in order to have been added to the structure.

For instance, the object of the second CE statement given in the earlier example is of type location. In the previous implementation of the rule enacting function we would be restricted to establishing relationships on the instance of that type given (home in our example). In the new implementation we are free to establish relationships or values on any of the CEInstance objects stored in the above structure, allowing us to correctly process and enact rules even if the concepts concerned in their consequents are not necessarily mentioned in the CE statement which satisfies the rule.

From a user's perspective (Figure 12), this means that the application can correctly infer relationships and values depending on the rules present in the model.

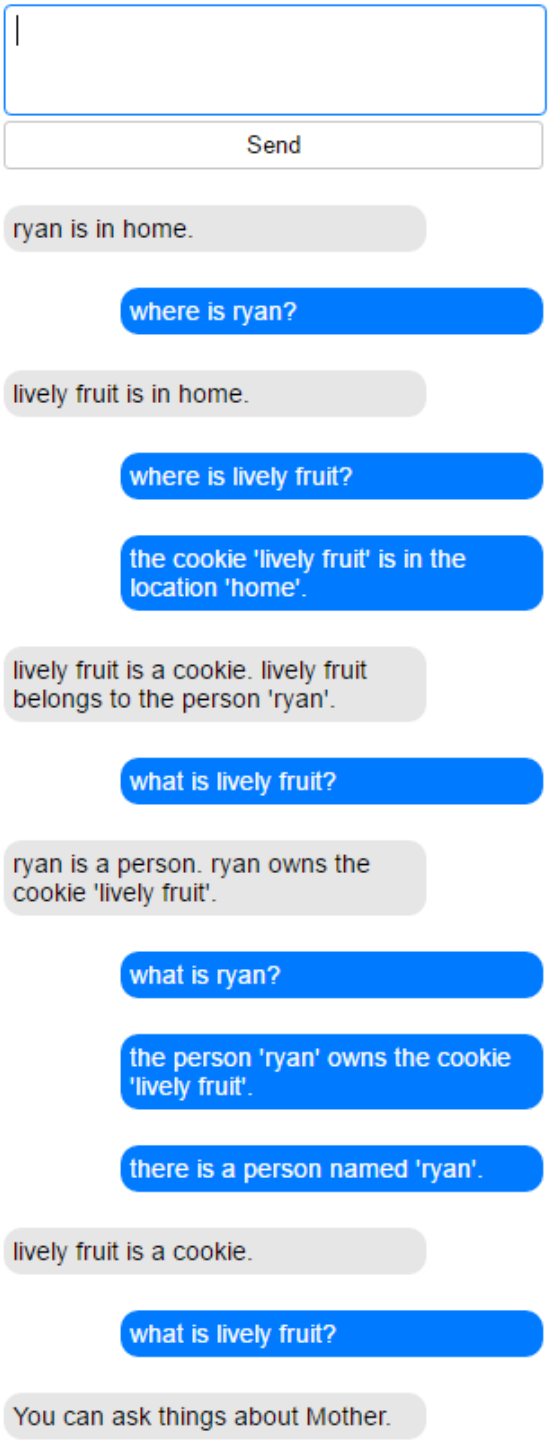

Figure 12: Inferring relationships based on rules using submitted CE (read from bottom up)

This rule enacting logic allows the node to correctly process and enact complex rules with multiple antecedents. There is, however, a caveat in that this only supports the enacting of rules with homogeneous antecedents. This is due to the way in which the subject's relationships and values are collected from a single available subject instance, meaning that the function can currently only ever evaluate rules whose antecedents concern a single concept type (i.e. the one passed as the subject). This is discussed further in the Future Work section.

Results and Evaluation

This section aims to provide information on the results of the project, as well as a critical evaluation against objectives and goals set during the initial plan.

In terms of solution implementation, I believe all of the objectives stated in the initial plan have been met, however without more extensive user testing there has not been a definitive answer to the question posed about the suitability of controlled English as a method of interaction with IoT devices. The system is able to receive controlled English input and process this input using the server-side CENode, output is given to the user in the form of controlled English text. Users are able to ask and have answered questions about Mother and the state of measurements conducted by the Cookies, as well as all information being accurate and up-to-date. Since communication between the server and client is conducted via AJAX, any loss of connectivity on the part of the client is handled gracefully.

With the exception of the potential enhancements mentioned in the Future Work section, I believe the system is able to provide useful information to the user, allowing them to interact with Mother in a natural and convenient way. I believe the system is at a point where a solid extensible foundation exists, and can be worked upon and improved more easily in the future.

A potential scaling problem was identified during testing, all facts about concept instances (though not the concepts or instances themselves) were erased from the node's knowledge base. This is most likely due to hardware limitations (primarily main memory), the continuous input of information into the knowledge base by the Sense API, and the corrective actions of the V8 JavaScript engine used by NodeJS. This issue, although potentially problematic, is not very severe as the instances and concepts stored in the node's knowledge base were not affected; there was only the loss of historical data linked to instances. Given the lack of state stored by the node, and the inability to distinguish past events recorded in the knowledge base from one another, I believe this issue would not disrupt the intended operation of the system.

The agile methodology chosen (Approach) was particularly useful during the development of the application as it allowed enormous flexibility and freedom to adapt to changing requirements. The project underwent several iterations when constructing the server-side components and modifying the CENode implementation. Further thoughts on development methodology are discussed in the Reflection on Learning section.

The decisions to interact with CENode via the programmatic API and make use of JavaScript and NodeJS proved to be advantageous due to the greater availability of documentation for CENode and increased functionality afforded by the programmatic API. The use of the NodeJS server environment also removed a significant amount of complexity and redundant work surrounding the implementation and testing of server routing and error handling first-hand. The use of a JavaScript-based framework for implementation also affords some level of interaction with the CENode library and attendant objects within the server environment that would not normally be present if using a different language. The use of JavaScript also allows for consistency across the application and consistency of business logic between server and client. There are also a wide variety of tools and external libraries (JQuery, for example) available for JavaScript which provide functionality that would be more complicated to implement otherwise.

I have confidence that the application will perform its function in a reasonable and correct manner, and is ready to proceed with user testing. These user tests would take the form of controlled interactions with the system, providing the user with a set of questions to answer and tasks to

accomplish using the application. These tests would range from relatively simple things such as ascertaining the current measurements from a named Cookie, to complex procedures taking advantage of the rules and inferences drawn by the application; ascertaining the temperature of a person by assigning them a particular Cookie and enquiring about the status of the person, for example. A range of questions would be needed to be representative of most (if not all) possible use cases that are of the system. These tests would be conducted with a range of participants, and observations as well as participant feedback being taken into account when assessing the benefits of controlled English conversational interactions with devices.

Walkthrough

This sub-section will outline a set of questions to be given to users when conducting an evaluation of the application. These questions are designed to cover all possible information generated by Cookies and require the user to interact with the system in order to obtain answers.

There are three main use cases for a Cookie sensor, monitoring a person (modelled with the 'belongs to' relationship), monitoring a room (modelled with the 'monitors' relationship), and monitoring an object (modelled with the 'is attached to' relationship). Since these relationships are defined in the default model (Appendix 3), they can be altered or added to at any time, even by providing controlled English statements to modify the model during runtime. In this way, additional rules or relationships can be specified by the user if they deem it necessary.

There are two main types of interaction with the node; the indirect interrogation of entities, making use of rules and inference to draw conclusions about entities. This type of interaction typically requires some input on the part of the user, in order to inform the system of a previously unknown fact (for example, the association of a Cookie with a Person). There are also more simple cases involving the direct interrogation of an entity (a Cookie, for example). This type of interrogation does not require any interaction on the part of the user to assign the Cookie to objects, rooms or people. This interaction is typically used by users who have prior knowledge of a link between a Cookie and Person, and as such do not require inference on the part of the system. The cases outlined below are designed to test interaction across both of these types of interaction.

Case 1: Direct Interrogation

In the first case we can ask the user to directly interrogate a Cookie, obtaining values for temperature and presence that have been directly measured by the Cookie.

If we submit the question:

> "What is lively fruit now?"

This prompts the system to give a complete summary of the current state of the Cookie instance "lively fruit":

> "lively fruit is a cookie. lively fruit is in the location home and has 3750 as temperature."

This is because these values are directly added to the relevant Cookie instance when a subscription event is posted to the application.

Case 2: Monitoring a Person

In the second case we can ask the user to find out a person's location and ambient temperature as follows:

If we submit the question:

> "What is ryan now?"
>
> *Note: The location of the person could also be obtained by asking "Where is ryan now?"*

This prompts the application to give a complete summary of the object instance. The application would respond with:

> "ryan is a person. ryan owns the cookie lively fruit and is in the location home and has 3750 as temperature."

This is because a two-way relationship exists between the person 'Ryan' and the Cookie 'lively fruit'. We can ask the application to clarify the concepts and their relations as follows:

> "What is a person?"
>
> "What is a cookie?"

The application would respond with:

> "A person is a type of locatable thing. An instance of person has a value called temperature and owns a type of cookie."
>
> "A cookie is a type of locatable thing. An instance of cookie has a value called temperature and has a value called battery and has a value called motion and is attached to a type of object and monitors a type of room and belongs to a type of person."

As we can see, an instance of Cookie belongs to an instance of Person and an instance of Person owns an instance of Cookie. To further clarify we can request information about the rules present in the model by using their name. If we were to ask about the rules concerning the relationships between Person, Cookie, Temperature and Presence:

> "What is r1?"
>
> "What is r2?"
>
> "What is r4?"
>
> "What is r5?"
>
> *Note: A list of all instances of type rule could be obtained by asking "list instances of type rule."*

The application would respond with:

> "r1 is a rule. r1 has 'if the cookie C ~ belongs to ~ the person P then the person P ~ owns ~ the cookie C' as instruction."

> "r2 is a rule. r2 has 'if the person P ~ owns ~ the cookie C then the cookie C ~ belongs to ~ the person P' as instruction."

> "r4 is a rule. r4 has 'if the cookie C ~ belongs to ~ the person P and the cookie C ~ is in ~ the location L then the person P ~ is in ~ the location L' as instruction."

> "r5 is a rule. r5 has 'if the cookie C ~ belongs to ~ the person P and the cookie C has the value T as ~ temperature ~ then the person P has the value T as ~ temperature ~' as instruction."

These rules allow us to ensure bi-directional relationships between Cookie and Person, and to infer the properties of any instance of Person as long as they own an instance of Cookie. The establishment of these relationships creates a transitive relation between the data collected by the Cookie and the entity it is assigned to. This means that when a JSON event is sent via subscription (Figure 7), we can parse and submit information from that particular Cookie and draw inferences about any concept instances related to it.

Assuming the default model (Appendix 3), answering these questions about the temperature and location of a person would require interaction with the application to inform the application about a new person (Ryan) and to assign a Cookie to that person to be able to infer properties. This case allows us to evaluate how a user interacts with the system beyond simply asking questions.

Case 3: Monitoring an Object

In the third case we can ask the user to determine if an object has been moved. A typical interaction with the system would proceed as follows:

> "What is money box now?"

To get the current state of an object (a safe in this example). This would prompt the application to list the most recent recorded properties and relationships on the Safe instance:

> "money box is an object. money box is monitored by the cookie pretty joy and has 1548 as motion."

The concepts mentioned here (Object and Cookie) are defined in the default model with relationships similar to those mentioned in the first case earlier. These relationships together with the rules written about them are what allows the system to infer the properties of the 'safe' instance. There is a similar chain of provenance for the inferences made in this case as was presented in case 1:

> "An instance of object has a value called motion and is monitored by a type of cookie."

> "r6 is a rule. r6 has 'if the cookie C ~ is attached to ~ the object O then the object O ~ is monitored by ~ the cookie C' as instruction."

> "r7 is a rule. r7 has 'if the object O ~ is monitored by ~ the cookie C then the cookie C ~ is attached to ~ the object O' as instruction."

```
"r11 is a rule. r11 has 'if the cookie C ~ is attached to ~ the
object O and the cookie C has the value M as ~ motion ~ then
the object O has the value M as ~ motion ~' as instruction."
```

As with the first case, this question requires the user to have more advanced interactions with the application than simply asking questions about Cookies. Part of the benefit of this method of interaction is that it allows the user to tell the application about new facts or relationships which are then incorporated into the knowledge base for processing.

Case 4: Ascertaining the Temperature of a Room

In the fourth case we can ask the user to determine the temperature of a room. The basic pattern of this question follows very closely that of the first two cases. The user is required to input some new fact into the knowledge base informing the node of the existence of a new room and its monitoring by a Cookie. A set of rules in the model similar to those found above ensure the presence of a bi-directional relationship between these objects and infers the temperature of the room based on the temperature of the Cookie.

Although the scope of each of these cases is limited, they provide adequate coverage of functionality in the application. All properties, relationships, concepts, and rules can be modified via the model or during runtime with submitted controlled English statements. For example, it would be possible to modify the knowledge base to include a temperature attribute of an object, as well as motion; new rules could be written to infer the temperature of an object based on its attached Cookie.

Feedback and Findings

Although there was not sufficient time to conduct formal user testing using the procedure outlined above, informal testing was conducted during the development to provide feedback on the use of controlled English as a method of interaction. The findings are outlined below.

Testing was conducted on a non-technical user with no prior knowledge of controlled English, its grammar, or its syntactic requirements. Upon introduction to the system the user was immediately able to recognise and use the input mechanism (Figure 12) to interrogate the application. The user was able to formulate queries and understand output given by the system.

After a brief introductory period, the user became familiar with the types of questions necessary to illicit the desired response from the application ('what' vs 'where' questions for example), though it was not initially obvious to the user that some information could only be gathered by asking what an object was. After a brief period of time and some limited hinting, the user became accustomed to this method of querying to get a summary of an object and its properties.

The user was able to understand and write CE rule statements in order to infer the location of a user based on the presence of their Cookie and found the propositional logic of the rule statements easy to understand and reason about. The user also quickly became acquainted with the syntax and grammar of the controlled English language, for example the need to quote names of instances when defining relationships. The node's predictive feedback was helpful in this regard.

During post-test feedback the user commented that the similarity to plain English was of benefit and that they found the controlled equivalent relatively easy to pick up and understand. The user also expressed a preference for the conversational interface over that of a dashboard or other method.

In the absence of further formal testing, I believe it would also be useful to note my own personal findings regarding the use of CE as an interaction method, collected from observations and experience during development.

Though initially unclear, I found the grammar of the language relatively easy to learn and understand. I believe the conversational nature of interaction making use of controlled English to be of benefit, allowing the simple transfer of variable information in an unambiguous manner.

Although the similarities between controlled English and plain English are indeed of considerable benefit when learning and understanding the language, I believe there could be some cases where these similarities afford syntax or grammar that is not possible in the controlled English environment. This could result in confusion or perhaps even frustration when the CE system fails to understand the user's intent. However, these cases could be avoided by providing instruction or help to the user, emphasising the grammatical and structural limitations placed on the CE language. Indeed, these limitations are a necessary part of the language in some cases, in order to prevent the ambiguity that would normally be present in plain English.

I believe this method of interaction also benefits the user by providing them with an easily configurable way to create domain models and rule statements, which would allow a CE system to reason and make inferences on behalf of the user. The fact that these domain models can be defined using controlled English means that there is a considerably lower barrier to entry when compared to other such systems which may be more complex or require specialised training to operate. The fact that CE was designed to allow machine-to-machine communication also provides an easy method of compatibility with existing systems, for example the incorporation of a CE processing environment into the delivery of data from Mother.

Although this sample is by no means representative and should not be taken as definitive, I believe the feedback and experiences outlined above provides valuable insight into the suitability of controlled natural language as a method of interaction with devices and services. More extensive user testing must be conducted in order to draw any kind of definitive conclusion.

Future Work

This section aims to give information regarding future work and how the project could be expanded following the end of development. Given more time I believe there are several enhancements to be made to the application that would prove useful. This work ranges from relatively simple interface improvements to further improving the way the node processes rules.

Improvements could be made to the chat section of the user interface (Figure 12) to ensure that the chat is more readily understandable, removing the need to read from the bottom of the page upwards or changing the position and order of the chat bubbles to mimic other messaging systems such as Skype, for example.

As the application relies on the presence of a valid subscription registered with the Sense API, it could be improved by ensuring that such a subscription already exists, or if necessary creating it upon start-up. This could make use of the already present subscription manager module to accomplish this. Another improvement that could be made is the periodic automatic detection of Cookies. The application currently automatically detects all nodes (Cookies included) that are registered to a user's Sense account upon start-up (this is how node UIDs (Figure 7) are matched to Cookie names), however this would not detect a Cookie that was newly registered with the account after the application had already started.

Parsing of some event types could be improved by adding some interpretation to the received data during parsing before submission to the node for processing. This would allow us to give more meaningful information to the user to answer questions. This would be particularly useful for event types which are meant to convey qualified states via measured quantities, for example, the motion feed which uses quantified data to measure motion duration and intensity but cannot easily be interpreted directly by the user. Event type parsing could also be improved by adding the ability to parse more application-specific data fields from event JSON. Currently the application parses event types by looking up the type of event in an object to acquire the correct field name for the `data` field in the submitted JSON (Figure 7). This could be improved by including a more comprehensive listing of available field names and event types, it should be noted that this method of acquiring field names is reliant on the consistency of the Sense API.

The application could also be improved by removing the tight coupling that currently exists to a single Mother hub/Sense account via the API token provided by Sense. This could be accomplished via a user account system that would allow each user to enter their own API key and Sense account details (stored securely) that would allow the application to interact more easily with multiple Mother hubs. This would also necessitate some changes to the application structure to ensure that each user has their own linked instance of a CENode to keep track of their individual Mother's information and exposing multiple dynamically generated HTTP endpoints (based on individual user details) to allow for simultaneous communication with multiple subscriptions from the API.

Another improvement that could be made is to ensure compatibility with existing CENode-based systems via policies. As described earlier (Implementation, Compatibility with Policies & Existing Implementations) the current system is not readily compatible with existing nodes via policies due to the complications surrounding the necessary serialisation of node objects. This compatibility could be achieved by using a function within the CENode library to convert and properly construct JSON containing the defined properties when serialising instances of objects. This would then allow for the direct access to these 'helper' properties in a similar manner to existing CENode implementations.

The enacting of rules could also be improved by adding the ability to parse heterogeneous antecedents. This would allow the writing of more complex rule statements which have the ability to infer relations or values based on the satisfaction of a more varied set of conditions.

Another potential enhancement would be the modification of the node's knowledge base to include state. This could be accomplished additively, through the use of some kind of meta-model which attempts to model stateful information on top of the existing monotonic knowledge base; or through changes to the way in which the existing knowledge base stores and retains information. The meta-model approach could make use of multiple nested concepts linked via relationships to create a model of a discrete stateful event, which is then populated with the event's data, within the knowledge base. This approach would be similar to the way in which the current system uses linked objects to store timestamps on CE cards, for example.

Interactions with the node could also be improved by adding some ability to parse direct questions about the properties of an instance, instead of having to ask what it was. For example, instead of asking "What is kitchen now?" in order to get the temperature of the room, you could ask "What is the temperature of kitchen?". This direct questioning about properties is a more natural way to phrase the query and more intuitive to users. Improvements would need to be made to the way in which the node processes natural language queries in order to accommodate this.

More intelligent conversational interactions could also be added to the node by making use of back-chaining to infer the implied presence of undisclosed facts. As an example, if the user asks the application about the temperature of a person and no relationship exists between a Cookie and that person, the system will be unable to answer the question. This interaction could be improved by using back-chaining to infer from rules in the knowledge base that there must be an association between a Cookie and a person to measure the temperature of that person. The system could then tell this to the user or ask if they would like to associate the person with a Cookie. This meta-parsing of rules allows the system make inferences based on the rule statements themselves rather than about the concepts they concern. When back-chaining it would be advantageous to travel as far up the chain of inferences as possible before requesting clarification from the user. This attempt to gain the most general available clarification allows the system to infer the maximum possible amount of information from given facts, perhaps triggering the execution of other rule statements as the chain is followed forward to the point of interest.

Conclusions

This section will outline the points covered in the body of the report and provide a brief summary of conclusions reached in the development.

The aim of this project was to create an application which allowed users to interact with Mother via the medium of controlled natural language and assess whether this method of interaction was viable and user-friendly.

I believe that the development was a success, resulting in an application which performs its function in a correct and appropriate manner. All objectives stated in the initial plan were met and the application is now ready for user testing. Additionally, much of the work done in this project was to get the application to a minimum viable state where it could respond and reason about information collected from Mother in an appropriate manner. I believe the application provides a solid and easily extensible foundation to continue work on this project in the future.

Unfortunately, due to the time constraints on the project and time taken up with development (particularly the issues mentioned in Implementation), there was not time to conduct the controlled testing as outlined (Walkthrough). As a result, no definitive answer is available to the question regarding the suitability of controlled English as a method of interaction.

Feedback from the limited user testing conducted revealed that while controlled English is certainly a technically viable method of interaction, with distinct benefits and advantages over existing information reporting tools, more extensive testing would be needed in order to determine whether its use is preferable over such tools.

While there is evidence to support the use of controlled English as a method of interaction with IoT devices and other services, further user testing must be conducted in order to reach a conclusive answer.

Reflection on Learning

This section will focus on a double-loop approach to the development, attempting to identify areas of success and areas of improvement surrounding development methods and planning. The goal of this section is to identify, with hindsight, techniques which worked well and should be carried on to future developments, and techniques which should be improved upon given experience with this project.

I believe the planning and design stages of the project could have benefitted from more robust research, particularly on the capability of the Sense API (such as the availability of subscriptions) and the then-current capabilities of the CENode library. Although there was limited documentation available concerning the capabilities of the CENode library, there were alternate sources that could have been consulted. This would have allowed for some planning of potential problems.

The agile methodology was particularly useful during the development as it allowed enough elasticity to deal with problems as they arose, where a more rigid development methodology such as waterfall (which places heavy emphasis on the execution of a pre-defined plan) would have proven problematic given the scale of the problems faced and the disruption caused. The agile practice of holding regular sprint meetings with the project supervisor also allowed for the discussion of progress, as well as problems and potential solutions on a frequent basis.

Unnecessary diversions or non-critical developments should be kept out of the project as they detract from the main areas of development and leech development time from critical features. For example, there was a brief period early in development where some development time was wasted attempting to implement OAuth for authentication with the Sense API, it was decided that this was not critical to the application and should not be pursued at this time. The agile focus of providing a minimum viable product at the end of each iteration was helpful in this regard as it allowed me to focus only on what was absolutely necessary for the project.

Although initially daunting, I should not allow myself to be overwhelmed by the perceived scope and seemingly 'unsolvable' problems, as this distracts from the development and potential solutions. Meeting regularly with the project supervisor was particularly helpful during these times as it provided an outside perspective and allowed me to focus on solutions rather than problems.

Although initially planned, test-driven development was not conducted as more time was spent on development than writing and implementing tests. I believe it would have been beneficial to have a suite of tests to give a safety net when developing and ensure that no changes made broke or disrupted parts of the system.

There was also the need for more user testing in order to properly investigate the suitability of controlled English as a method of interaction with devices and services such as Mother. I believe more time should have been planned for testing or perhaps user testing should have been conducted throughout the development process. In the latter case, even if testing was conducted when the system was incomplete the feedback obtained would have been better than the limited feedback available on a complete system. It is difficult to propose a solution to this problem since large amounts of development time were necessarily taken up with solving problems faced and developing the application to a point where user testing was possible.

Glossary

Monotonic – increasing in state, the absence of the rolling back of state.

Conditional statement - an “if p, then q” compound statement (for example, If I throw this ball into the air, it will come down); p is the antecedent, and q is the consequent. A conditional statement asserts that if its antecedent is true, its consequent is also true.

Antecedent – see conditional statement.

Consequent – see conditional statement.

Homogeneous – consisting of things that are very similar or all of the same type.

Heterogeneous – consisting of parts or aspects that are unrelated or unlike each other.

Transitive Relation – Whenever element A is related to element B and element B is related to element C, then element A is also related to element C. For example, measurements are related to sensors, sensors are related to people so measurements are related to people.

Table of Abbreviations

The following abbreviations are used in the body of the document:

API – Application Programming Interface

CE – Controlled English

CNL – Controlled Natural Language

GUI – Graphical User Interface

IoT – Internet of Things

JSON – JavaScript Object Notation

Regex – Regular Expression

SHERLOCK – Simple Human Experiments Regarding Locally Observed Collective Knowledge

Appendices

Appendix 1

Source code for ascertaining the current state of an object

```
else if (t.match(/^(\bwho\b|\bwhat\b) (?:is|are) (?:.*) (now)/i)) {
  t = t.replace(/\?/g, '').replace(/'/g, '').replace(/\./g, '');

  // If we have an exact match (i.e. 'who is The Doctor?')
  var name = t.match(/^(\bwho\b|\bwhat\b) (?:is|are) ([a-zA-Z0-9_ ]*) (?:now)/i);
  var instance;
  if (name) {
    instance = get_instance_by_name(name[2]);
    if (instance != null) {
      //check if text should be 'an' or 'a'.
      var vowels = ["a", "e", "i", "o", "u"];
      var string = "";
      if (vowels.indexOf(instance.type.name.toLowerCase()[0]) > -1) {
        string += instance.name.toLowerCase() + " is an " +
instance.type.name.toLowerCase() + ".";
      } else {
        string += instance.name.toLowerCase() + " is a " +
instance.type.name.toLowerCase() + ".";
      }

      var value_labels = [];
      var relationship_labels = [];
      var values = instance.values;
      var relationships = instance.relationships;

      for (var relationship in relationships) {
        if (relationship_labels.indexOf(relationships[relationship].label) == -1) {
          relationship_labels.push(relationships[relationship].label);
        }
      }

      for (var value in values) {
        if (value_labels.indexOf(values[value].label) == -1) {
          value_labels.push(values[value].label);
        }
      }

      if(value_labels.length > 0 || relationship_labels.length > 0){
        string += " " + instance.name.toLowerCase() + " ";
      }

      for (var i = 0; i < relationship_labels.length; i++) {
        if (i == relationship_labels.length - 1 && value_labels.length < 1) {
          string += (relationship_labels[i] + " the " +
relationships[i].instance.type.name.toLowerCase() + " " +
instance.property(relationship_labels[i].toLowerCase()).name.toLowerCase() + ".");
        } else {
          string += (relationship_labels[i] + " the " +
relationships[i].instance.type.name.toLowerCase() + " " +
instance.property(relationship_labels[i].toLowerCase()).name.toLowerCase() + " and
");
        }
      }

      for (var j = 0; j < value_labels.length; j++) {
        if (j == value_labels.length - 1) {
          string += ("has " + instance.property(value_labels[j].toLowerCase()) + "
as " + value_labels[j].toLowerCase() + ".");
        } else {
          string += ("has " + instance.property(value_labels[j].toLowerCase()) + "
as " + value_labels[j].toLowerCase() + " and ");
        }
```

```
      }

      return [true, string];
    } else {
      return [true, "I don't know who or what that is."];
    }
  }
}
```

Appendix 2

Source code for enacting rules upon submitted CE statements

```
var enact_rules = function (subject_instance, property_type, object_instance) {
  var rules = node.get_instances("rule");
  for (var i = 0; i < rules.length; i++) {
    var rule = node.parse_rule(rules[i].instruction);
    var condition_instances = {};
    if (rule == null) {
      return;
    }
    // if the rule's consequent does not concern the object_instance we've been
given, skip and check next rule,
    // only works for rules with a single 'then' consequent.
    if (typeof object_instance != "string" && rule.then[0].concept !=
object_instance.type.name && rule.then[0].condition.type !=
object_instance.type.name) {
      continue;
    }

    // Get all the object instances mentioned in each 'if' condition that is
related to the subject_instance we've been given.
    // include the subject_instance itself.
    // Does not support mixing condition clause types i.e. 'if' conditions must all
have the same concept property.
    condition_instances[subject_instance.type.name] = subject_instance;
    for (var current in rule.if) {
      current = rule.if[current];
      // if the relationship specified in the condition exists on the
subject_instance and the target of that relationship
      // is the same as the type specified in the condition, add it to the
structure.
      var subject_instance_property =
subject_instance.property(current.condition.label);
      if (subject_instance_property && current.condition.type == "value") {
        condition_instances[current.condition.label] = subject_instance_property;
      } else if (subject_instance_property && subject_instance_property.type.name
== current.condition.type) {
        condition_instances[current.condition.type] = subject_instance_property;
      }
    }

    // for each consequent, if it applies to our object_instance and if our
subject_instance is related to its concept
    // then add the relationship or value specified in the 'then' consequent.
    if (typeof object_instance == "string") {
      for (var consequent in  rule.then) {
        consequent = rule.then[consequent];
        if ((consequent.condition.type == "value") &&
(condition_instances.hasOwnProperty(consequent.concept)) &&
(condition_instances.hasOwnProperty(consequent.condition.type) ||
condition_instances.hasOwnProperty(consequent.condition.label))) {
          if (consequent.condition.type == "value") {

condition_instances[consequent.concept].add_value(consequent.condition.label,
condition_instances[consequent.condition.label], false);
          } else {

condition_instances[consequent.concept].add_relationship(consequent.condition.label
, condition_instances[consequent.condition.type], false);
          }
        }
      }
    } else {
      for (var consequent in  rule.then) {
        consequent = rule.then[consequent];
        if ((consequent.concept == object_instance.type.name ||
```

```
consequent.condition.type == object_instance.type.name) &&
(condition_instances.hasOwnProperty(consequent.concept)) &&
(condition_instances.hasOwnProperty(consequent.condition.type) ||
condition_instances.hasOwnProperty(consequent.condition.label))) {
          if (consequent.condition.type == "value") {

condition_instances[consequent.concept].add_value(consequent.condition.label,
condition_instances[consequent.condition.label], false);
          } else {

condition_instances[consequent.concept].add_relationship(consequent.condition.label
, condition_instances[consequent.condition.type], false);
          }
        }
      }
    }
  }
};
```

Appendix 3

Default model loaded upon application start-up

```
"conceptualise a ~ person ~ P that is a locatable thing.",
"conceptualise a ~ cookie ~ C that is a locatable thing.",
"conceptualise a ~ room ~ R that is a location.",
"conceptualise a ~ object ~ O.",

"conceptualise the person P ~ owns ~ the cookie C and has the value T as ~
temperature ~.",

"conceptualise the cookie C ~ is attached to ~ the object O.",

"conceptualise the cookie C ~ monitors ~ the room R.",

"conceptualise the cookie C ~ belongs to ~ the person P and has the value T as ~
temperature ~ and has the value B as ~ battery ~ and has the value M as ~ motion
~.",

"conceptualise the room R ~ is monitored by ~ the cookie C and has the value T
as ~ temperature ~.",

"conceptualise the object O ~ is monitored by ~ the cookie C and has the value M
as ~ motion ~.",

"there is a rule named r1 that has 'if the cookie C ~ belongs to ~ the person P
then the person P ~ owns ~ the cookie C' as instruction.",

"there is a rule named r2 that has 'if the person P ~ owns ~ the cookie C then
the cookie C ~ belongs to ~ the person P' as instruction.",

"there is a rule named r4 that has 'if the cookie C ~ belongs to ~ the person P
and the cookie C ~ is in ~ the location L then the person P ~ is in ~ the
location L' as instruction.",

"there is a rule named r5 that has 'if the cookie C ~ belongs to ~ the person P
and the cookie C has the value T as ~ temperature ~ then the person P has the
value T as ~ temperature ~' as instruction.",

"there is a rule named r6 that has 'if the cookie C ~ is attached to ~ the
object O then the object O ~ is monitored by ~ the cookie C' as instruction.",

"there is a rule named r7 that has 'if the object O ~ is monitored by ~ the
cookie C then the cookie C ~ is attached to ~ the object O' as instruction.",

"there is a rule named r8 that has 'if the cookie C ~ monitors ~ the room R then
the room R ~ is monitored by ~ the cookie C' as instruction.",

"there is a rule named r9 that has 'if the room R ~ is monitored by ~ the cookie
C then the cookie C ~ monitors ~ the room R' as instruction.",

"there is a rule named r10 that has 'if the cookie C ~ monitors ~ the room R and
the cookie C has the value T as ~ temperature ~ then the room R has the value T
as ~ temperature ~' as instruction.",

"there is a rule named r11 that has 'if the cookie C ~ is attached to ~ the
object O and the cookie C has the value M as ~ motion ~ then the object O has
the value M as ~ motion ~' as instruction.",

"there is a location named 'home'.",
"there is a location named 'not home'.",
"there is a cookie named 'lively fruit'.",
"there is a cookie named 'many chocolat'.",
"there is a cookie named 'pretty joy'."
```

References

Allen, J. F., Byron, D. K., Dzikovska, M., Ferguson, G., Galescu, L., & Stent, A. (2001). Toward Conversational Human-Computer Interaction. *AI Magazine, vol. 22, No. 4*, pp. 27-38.

Cummings, M. M. (2014). Man versus Machine or Man + Machine? In *Intelligent Systems* (pp. 62-69). IEEE 29.5.

David Marcus, Facebook. (2016, April 14). *Messenger Platform at F8*. Retrieved from Newsroom: http://newsroom.fb.com/news/2016/04/messenger-platform-at-f8/

ECMA International. (2013). *The JSON Data Interchange Format.* ECMA International.

IBM. (2016, April 05). *IBM Controlled Natural Language Processing Environment*. Retrieved from IBM developerWorks: https://www.ibm.com/developerworks/community/groups/service/html/communityview?communityUuid=558d55b6-78b6-43e6-9c14-0792481e4532]

Kuhn, T. (2014). A Survey and Classification of Controlled Natural Languages. *Computational Linguistics 40.1*, pp. 127-170.

Libov, J. (2016, January 22). *Futures of Text*. Retrieved from Whoops by Jonathan Libov: http://whoo.ps/2015/02/23/futures-of-text

Mozilla. (2016, April 04). *Ajax*. Retrieved from Mozilla Devloper Network: https://developer.mozilla.org/en-US/docs/AJAX

Preece, A. D., Braines, D., Pizzocaro, D., & Parizas, C. (2014). Human-Machine Conversations to Support Multi-Agency Missions. *ACM SIGMOBILE Mobile Computing and Communications Review*, 75-84.

Preece, A., Gwilliams, C., Parizas, C., Pizzocaro, D., Bakdash, J. Z., & Braines, D. (2014). Conversational Sensing. *SPIE Technology + Applications*. International Society for Optics and Photonics.

Preece, A., Webberley, W., Braines, D., Hu, N., Porta, T. L., Zaroukian, E., & Bakdash, J. Z. (2015). *SHERLOCK: Simple Human Experiments Regarding Locally Observed Collective Knowledge.* US Army Research Laboratory.

Rodriguez, A. (2015, February 09). *RESTful Web Services: The basics*. Retrieved from IBM developerWorks: http://www.ibm.com/developerworks/library/ws-restful/

Sen.se. (2016, April 04). *Mother*. Retrieved from Sen.se: https://sen.se/mother/

Sen.se. (2016, April 04). *Sen.se API*. Retrieved from Sen.se: https://sen.se/api/docs/v2/#endpoints

Skype, Microsoft Corporation. (2016, April 05). *Skype Bots preview comes to consumers and developers*. Retrieved from Skype: http://blogs.skype.com/2016/03/30/skype-bots-preview-comes-to-consumers-and-developers/

Webberley, W. M. (2016, April 06). *CENode API Documentation.* Retrieved from GitHub: https://github.com/flyingsparx/CENode/blob/master/API.md

Webberley, W. M., & Preece, A. (2016, April 05). *CENode.* Retrieved from CENode: http://cenode.io/documentation.pdf

Webberley, W., Preece, A., & Braines, D. (2015). CENode: Enabling Human-Machine Conversations at the Network Edge. *Annual Fall Meeting fo the International Technology Alliance*.

Weiss, B., Wechsung, I., Kuhnel, C., & Moller, S. (2015). Evaluating Embodied Conversational Agents in Multimodal Interfaces. *Computational Cognitive Science 1.1*, pp. 1-21.