A Human-Centric API for Programming Socially Interactive Robots

J. P. Diprose & B. Plimmer
Dept. of Computer Science
University of Auckland
New Zealand
jdip004@aucklanduni.ac.nz
beryl@cs.auckland.ac.nz

B. A. MacDonald
Dept. of Electrical & Computer Engineering
University of Auckland
New Zealand
b.macdonald@auckland.ac.nz

J. G. Hosking
College of Engineering & Computer Science
Australian National University
Australia
john.hosking@anu.edu.au

***Abstract*— Whilst robots are increasingly being deployed as social agents, it is still difficult to program them to interact socially. This is because current programming tools either require programmers to work at a low level or lack features needed to create certain aspects of social interaction. High level, domain specific tools with features designed specifically to meet the requirements of social interaction have the potential to ease the creation of social applications. We present a domain specific application programming interface (API) that is designed to meet the requirements of social interaction. The Cognitive Dimensions Framework was used as a design tool during the design process and the API was validated by implementing an exemplar application. The evaluation of the API showed that programmers with no robotics knowledge were positively impressed by the notation and that its organization, domain specific interfaces and object oriented nature positively affected several Cognitive Dimensions.**

Keywords— application programming interfaces, api, usability, design, cognitive dimensions, human robot interaction, social robot interaction, humanoid robot.

I. Introduction

There are many applications for social robots, that is robots that interact with humans in a human-like way [1]. Examples include: companions for the aged [2], interactive theatre [3] and robotic butlers [4]. Despite there being much work in social robotics, it is still a challenge to create such applications. There are two reasons for this: first, the tools used to create social robot scenarios lack support for many social interaction requirements; second, tools that do support social interaction requirements often express them at too low an abstraction level.

Social robot applications are ideally created by combining social primitives to form higher level social interactions; as illustrated in Figure 1. Examples of social primitives include: speaking to people, performing gestures and understanding human speech and gesture. These primitives can be used to build higher level social interactions including: dialogue, joint attention and displaying expression. The end goal of this layered approach to robot programming is to have domain specific end-user programming languages for social robotics.

Without support for social interaction, programmers are unable to adequately express its nuances. For example, a common task is for a robot to speak to a specific person by vocalising words and gazing at them [1]. To realize this requires synthesising speech, specifying who is being spoken to, making gestures (gaze) and synchronising gestures with speech. If tools don't support these requirements, the programmer is unable to express the nuances of social interaction.

In addition, supporting the requirements of social interaction, but at lower levels of abstraction than necessary is problematic as it takes the programmer excessive work to attain goals. For example, some tools realise important requirements of making robot speech and gesture, but express them at low abstraction levels. Rather than simply specifying who the robot is to speak to and what it should say, a programmer must give detailed commands for speech synthesis to make it speak, analyse results

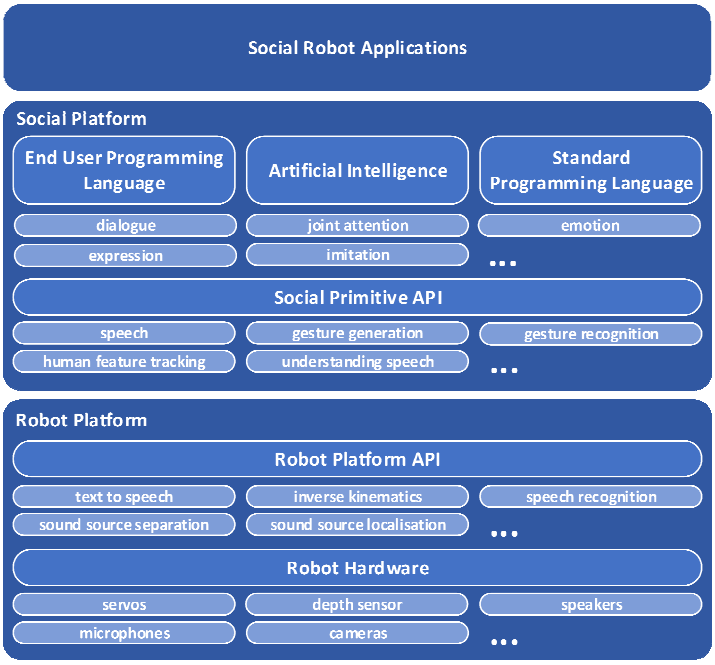

FIGURE 1. HIERARCHY OF SOCIAL ROBOT SYSTEMS.

from a face detection algorithm to find where the person's head is and use joint control to make the robot gaze at that head.

This paper addresses the problem of designing a social primitive API that supports social interaction requirements and expresses them at a high abstraction level. It is an attempt to make social robotics programming accessible to programmers with or without robotics-programming experience and could be used as a platform to create end-user programming languages for authoring higher level interactions such as dialogue, joint attention and showing expression.

We start by discussing how related work fails to meet social primitive requirements and/or express them at an appropriate abstraction level (Section II). We then describe our approach to designing an API based on a set of social primitives and requirements that are important for expressing social interaction (Section III). Next, we describe our API (Section IV), its evaluation and results obtained (Section V). Lastly, lessons learnt from the evaluation are discussed in Section VI.

II. PRIOR APPROACHES

A number of systems have been designed to address problems related to programming robot social interactions. They include visual tools to support end-user programming; and software frameworks, API's and textual domain specific languages designed for programmers. We overview several of these tools analysing their limitations, summarised in Table 1.

A. Choregraphe & NAOqi

Choregraphe is an end-user programming environment for Nao robots using two visual programming views, a key-frame animation editor, simulator and a code editor [5]. A flow diagram is used to combine algorithms together to produce specific behaviour; e.g., one can make Nao respond verbally to a word detected by its speech recogniser. A visual timeline is used to organise animation timing and is often used together with the key-frame animation editor to create individual animations. NAOqi [6] is the software framework that Choregraphe is built on. It can be used with a variety of programming languages including: C++, Python and Java.

B. Interaction Composer

Interaction Composer is a programming environment supporting collaboration between programmers and end users to create social interaction scenarios [7]. Programmers perform low level tasks, such as face recognition; while interaction designers (akin to end-users) use Interaction Composer, a visual programming environment, to create the higher level dialogue and interaction sequences, such as a greeting scenario. The programmer developed modules are visually represented as "blocks" which the scenario designers use in the visual programming environment. It also has textual versions of each of its visual programming modules.

C. TiViPE

Lourens et al. [8] present an API for programming a Nao robot paired with a visual programming environment TiViPE. They use the environment to create a social scenario of a robot shaking a child's hand. TiViPE's social interaction functions are: control of the robot's LEDs, audio, joint control and serial or parallel command execution. As with Interaction Composer [7], Lourens et al. argue programmers and scenario designers can collaboratively create social robot interaction using TiViPE; the scenario designer decides what behavioural "blocks" are needed, and the programmer creates them using the API.

D. Behaviour Markup Language

Behaviour Markup Language (BML) is a textual domain specific language for specifying the actions of Embodied Conversational Agents [9], which includes robots [10]. It allows the definition of behaviours by providing XML interfaces that control speech, gesture, gaze and body movement [9].

E. Limitations

There are two limitations with these tools. First, most realise social interaction requirements at a low-mid abstraction level. Second, all omit abstractions to realise important social interaction elements. A detailed analysis and comparison of the four programming tools and our own[1] is given in Table 1.

Choregraphe [5] and NAOqi [6] realise many social interaction requirements with low to mid-level abstractions. E.g., to make a robot point at a person, one must combine multiple low level abstractions (inverse kinematics and face detection) rather than using one abstraction, e.g. a target attribute for a gesture. They overlook some social interaction requirements, in particular, abstractions to specify: who is being spoken to; synchronise gestures with speech; and understand who is speaking to whom. Interaction Composer realises far fewer social requirements, but those that are realised are represented at a high abstraction level. For example, it is possible to specify gesture targets and synchronise them with speech.

TiViPE Nao API [8] realises the fewest requirements, making it the least equipped tool for specifying social interaction. It has abstractions for programming speech synthesis, body language, facial expressions and running multiple gestures at once. The last three of these are represented by mid-level abstractions, suggesting its abstraction level could be raised.

BML [9] implements all requirements for making gestures and at a high abstraction level. However, it fails to realise many requirements for making speech and all requirements of the human feature model, understanding speech and recognising gestures. Note that no tools implement any of the requirements for recognising gestures, including our own (due to time constraints). Adding gesture recognition is a future goal of ours.

[1] Our prototype doesn't implement all of the social interaction requirements, but it implements more than any of the other tools.

TABLE 1. ABSTRACTION LEVEL OF SOCIAL ROBOT PROGRAMMING TOOLS. (3) HIGH LEVEL, APPROPRIATELY DESIGNED ABSTRACTION; (2) MID LEVEL, ABSTRACTION NOT APPROPRIATE FOR REQUIREMENT; (1) LOW LEVEL, HAVE TO COMBINE MULTIPLE INAPPROPRIATE ABSTRACTIONS TO REALISE REQUIREMENT; (0) NO ABSTRACTION TO REPRESENT REQUIREMENT.

Social Primitives	Requirements (from [1], [11])	Choregraphe & NAOqi		Interaction Composer		TiViPE		BML		Our API	
		Abst lvl	Comment	Abst lvl	Comment	Abst lvl	Comment	Abst lvl	Comment	Abst lvl	Comment
Making Speech	Synthesise voice	3	tts.say(text)	3	talk(string **text**) (tab. I [7])	3	say (fig. 2. [8])	3	speech tag (fig. 5. [9])	3	robot.say_to(**text**, audience)
	Specify how words are being said	3	speed, vol, pause ([12])	0		0		0		0	
	Specify who is being spoken to	0		0		0		0		3	robot.say_to(text, **audience**)
Making Gesture	Body Language	3	sit down, stand up, wipe forehead (Choregraphe)	3	pointing, emphasis, big, small (sec. III c. 1 [7])	2	move or movem (fig. 2. [8])	3	gesture tag & body tag postures (fig. 5. [9]), body part movements (fig. 4. [9])	3	wave, point-left, point-right, hips
	Facial Expressions	2	ALLeds API	0		2	ledto or ledset (fig. 2. [8])	3	face tag (fig. 2. [9])	3	red-eyes, blue-eyes
	Synchronise with speech	0		3	gesture tag in text (sec. IV c. 1 [7])	0		3	mark tag (p100 code example [13])	3	"<wave> hello </wave>"
	Targeted at objects	1	joint control & face detection (gaze), inverse kinematics & face detection (point)	3	pointing reference (sec. III c. 1 [7])	0		3	target attribute of gaze & gesture tags (fig. 4 & 5 [9])	3	robot.say_to("<point **target={0}**>get him</point>", person, person1)
	Multiple gestures at once	2		0		2	[a|b] & c | d & e (sec. 2.1 [8])	3	pointing, sitting & gazing (fig. 5. [9])	0	
Human feature model	Body part model	1	ALFaceDetection	0		0		0	No abstractions to represent body parts of **sensed** people.	3	person.head, person.torso, person.left_hand, person.right_hand…
Understanding human speech	Verbal commands	2	ALSpeechRecognition	2	isSpeechResult(string result) (tab. I [7])	0		0		3	person.**said_to**(meaning, object)
	Continuous natural language	0		0		0		0		0	
	How they said it	0		0		0		0		0	
	Who the speaker is	1	ALSoundDetection & ALFaceDetection	0		0		0		3	**person**.said_to(meaning, object)
	Who they are speaking to	0		0		0		0		3	person.said_to(meaning, **object**)
Recognising gestures	What the gesture is	0		0		0		0		0	
	How they gestured	0		0		0		0		0	
	Who is gesturing	0		0		0		0		0	
	Who they were gesturing to	0		0		0		0		0	

As we have described, most of these systems fail to define primitives at a high level (Choregraphe [5], Interaction Composer [7] & TiViPE [8]). All fail to implement key requirements of social interaction. Both of these limitations hinder the ability of these tools to specify social interaction with ease. To alleviate these problems we need a programming tool with primitives set at a high abstraction level that meet more of the requirements of social interaction than current tools. In the next section we describe the approach we took to create our API.

III. OUR APPROACH

The most important trade-off when designing an API is between two design decisions: the expressability vs the usability of the API [14]. On the one hand, enough features need to be included in the API so that it can be used to produce solutions for a particular problem domain [14]. On the other hand it should be simple enough to learn and use [14].

To ensure enough features were included in the API, we undertook a two-step process. First, we examined the social robot literature for primitives that could be used to build social interaction [11]. We found that a number of social primitives must be supported to enable robots to interact socially with humans. These are the same requirements we used to compare the programming tools in Table 1; more detailed reports on these can be found in [11] and [1]. Second, to give the API a context, it was designed and implemented alongside an exemplar use case: a multiplayer game show (game shows are commonly used scenarios to explore social robot interaction [15], [16]). In our scenario a Nao robot hosts a quiz and two human players compete against each other by answering Nao's questions.

To ensure the API was simple enough to learn and use, the Cognitive Dimensions Framework [17] was used as a self-reflection tool during design and implementation. This was a useful aid when making design decisions that affected usability.

IV. OUR API

The API is a high-level interface for the social primitives described in column one of Table 1. It implements functionality as listed in Table 1 and is composed of a number of important classes that perform different tasks: Environment, Object (with subclasses Robot and Person), Query and StateMachine. The rest of this section overviews these.

A. Environment

The *Environment* class encapsulates the objects in the robot's environment, including the robot itself. These are represented by two attributes: *objects* and *robot*. The former references a list of the objects in the robot's environment and is automatically updated by the underlying platform. The latter references a *Robot* instance encapsulating the actions of a robot. The class and attribute names were chosen to support role expressiveness.

B. Object (Robot, Person)

The *Object* class encapsulates functions and attributes common to all objects in the environment. The most important functions include: *distance_to(obj)*, which finds the distance between two objects; and standard functions for querying the spatial relationships of objects, including, *left_of(obj)*, *right_of(obj)*, *infront_of* and *behind(obj)* which return whether an object (caller) is left-of, right-of, in front or behind another object (obj) respectively (Table 2). *Object* has two principle subclasses already defined for programmers[2]: *Robot* and *Person*. Instances of are automatically populated by the underlying framework into the *robot* and *objects* attributes of the *Environment* class respectively. We have implemented several functions for the Robot and Person classes that support the social primitives described in column one of Table 1.

TABLE 2. EXAMPLES OF OBJECT FUNCTIONS.

distance_to	
Parameters	obj (Object)
Example	`person.distance_to(robot)` `>> 1.1`
left_of	
Parameters	obj (Object)
Example	`person.left_of(robot)` `>> True`
Explanation	`person is to the left of robot`
behind	
Parameters	obj (Object)
Example	`person.behind(robot)` `>> False`
Explanation	`person is not behind robot`

The most important function for the *Robot* class is *say_to*; (Table 3) which makes the robot speak and gesture to a person or a group of people. Once the robot has made eye contact with a person specified by the *audience* parameter (specifies who is being spoken to) it begins synthesizing the text in the *text* parameter. If more than one person is supplied by the *audience* parameter, whenever a new sentence is reached, the robot changes its gaze to another person. As well as being designed with the Cognitive Dimensions principles in mind, the high level *say_to* was designed to fulfil the requirements of robot speech and gesture from column 1 of Table 1. These include: synthesise voice, specify who is being spoken to, synchronise gestures with speech and the ability to gesture.

TABLE 3. ROBOT SAY_TO FUNCTION.

say_to	
Parameters	text (String), audience (Object, Query)

Examples	`robot.say_to('Hello', people)`
	`robot.say_to('<wave> Hello </wave>', people)`
	`robot.say_to('<point target={0}> Who is that? </point>', people, person1)`

The text supplied to the text parameter can optionally be marked up with gesture tags to make the robot gesture in time with its speech. This fulfils most gesture making requirements, including: body language, facial expressions, synchronise with speech and target gesture at an object. The following list has example gestures, including both body language (wave, hands on hips, point) and facial expressions (red-eyes, blue-eyes):

- Wave: "<wave> Hello human </wave>"
- Hands on hips: "<hips> I am angry with you </hips>"
- Point arm to right: "<point-right> look at that over there </point-right>"
- Point arm to left: "no that <point-left> thing looks more interesting </point-left>"
- Change eye colour to red: "<red-eyes> I am the start of the robopocalypse </red-eyes>"
- Change eye colour to blue: "<blue-eyes> maybe not </blue-eyes>"

The function name was chosen to reinforce role expressiveness; *say_to(text, audience)* suggests the robot is able to say something (*text*) to one or more people (*audience*). The gesture markup language syntax was chosen to closely map to the act of synchronising gestures with speech, one of the social interaction requirements. To achieve this, tags surround the text: opening tags specify a gesture start *"<wave>"* and closing tags *"</wave>"* when it stops. Gesture tags are specified by the name of the gesture to keep the notation terse. Other systems such as Interaction Composer [7] and BML [9] use a more diffuse syntax, e.g. *"<gesture type='wave'> <gesture>"*.

The last relevant function for the Robot class is *associate_utterances_with_meaning*. This function associate's utterances people say with higher level meanings to enable verbal commands. An example is shown in Table 4, two

[2] New objects can be supported by sub-classing *Object*.

meanings are created: greet and insult. Different synonyms for these are created by associating a set of utterances people could say with those meanings. For example, 'hello' and 'hi' are both greetings, while 'stupid robot' and 'shut up' are insulting.

TABLE 4. ROBOT ASSOCIATE_UTTERANCES_WITH_MEANING FUNCTION.

associate_utterances_with_meaning	
Parameters	utterances (list), meaning (Enum)

Example	`meanings = Enum('greet', 'insult')`
	`robot.associate_utterances_with_meaning(['hello', 'hi'], meanings.greet)`
	`robot.associate_utterances_with_meaning(['stupid robot','shut up'], meanings.insult)`

The most significant function for the Person class is *said_to* (Table 5). It is used to find out if a specific person (who the speaker is) said an utterance with a particular meaning (verbal commands) to another object, such as the robot (who they are speaking to). It returns a Boolean indicating if this is true or not. Realising this on a mobile robot uses sound source localisation, tracking and separation to isolate an audio stream for each person; each audio track is then processed individually by a separate speech recogniser. This fulfils three requirements of understanding human speech: verbal commands, who the speaker is and who they are speaking to.

TABLE 5. PERSON FUNCTIONS.

said_to	
Parameters	meaning (Enum), other (Object)
Example	`person.said_to(meanings.greet, robot)` `>> False`

C. Query

The *Query* class is used to filter objects from the environment. Objects can be filtered by type (e.g. *Person* objects) or by distance (e.g. objects closer than 2m); sorted by an attribute (e.g. closest object); and selected (e.g people who said "yes" to the robot). It uses syntax similar to Microsoft's LINQ [18] called Python-ASQ [19]; examples of queries are shown in Table 6.

D. StateMachine

Dialogue is a cooperative process of communication that shares information between two or more individuals [1]. It is a higher form of interaction that emerges when social interaction primitives from both the *Robot* and *Person* classes are combined (Figure 1). A dialogue management system is needed to create social applications from the social primitives. We use an event driven state machine for this purpose (Table 7), which is popular with other programming tools, including Interaction Composer [7] and Robot Behaviour Description Language [20].

TABLE 6. QUERY FUNCTIONS.

query	
Parameters	iterable (Iterable)
Example	`q = query(env.objects)`
of_type	
Parameters	class (Class)
Examples	`ppl = q.of_type(Person)`
where	
Parameters	predicate (lambda)
Examples	`ppl.where(lambda p: p.distance_to(env.robot)) < 2)` `ppl.where(lambda p: p.said_to(meanings.greet, env.robot)`
order_by	
Parameters	predicate (lambda)
Example	`ppl.order_by(lambda p: p.distance_to(env.robot))`
order_by_descending	
Parameters	predicate (lambda)
Example	`ppl.order_by_descending(lambda p: p.distance_to(env.robot))`

TABLE 7. STATEMACHINE EXAMPLE.

```
sm = StateMachine(env)

class Listen(State):
    def create_transitions(self, next_state):
        q = people.where(lambda p: p.said_to(meanings.greet, robot))
        event = QueryEvent(q)
        self.add_transition(event, next_state, id = 'greeted')

class Respond(State):
    def create_transitions(self, next_state):
        self.next_state = next_state

    def execute(self, e):
        if e.id == 'greeted':
            robot.say_to('Hello, nice to meet you!', people)
        return Transition(self.next_state)

listen = Listen() #Define states
respond = Respond()
listen.create_transitions(respond) #Define transitions
respond.create_transitions(listen)
sm.add_state(listen, first = True) #Add to StateMachine
sm.add_state(respond)
sm.start() #Start
```

In the *StateMachine*, dialogue is represented across a number of states, by the class *State*. States contain social interaction primitives, such as *robot.say_to* commands, that are run when a state is run by the state machine. To make the state machine transition between states, the programmer combines *Query*, *Event (QueryEvent)* and *Transition* classes. For example, one could write a query that searches for a person that insults the robot. The *Query* is supplied to a *QueryEvent*, which fires when one or more people are returned by the *Query*. When the *QueryEvent* fires, the state machine transitions. In this new state, the robot interacts with the specific person who insulted it, for example, the robot could say to its insulter "you nasty human, you should be more careful - haven't you seen the Terminator?"

In summary, our API allows social interactive primitives to be programmed with the *Environment*, *Object (Robot, Person)* and *Query* classes. Dialogue is programmed with the *StateMachine*

and its associated classes. In the next section we describe the evaluation of our API.

V. EVALUATION

We evaluated our API by a usability study where programmers used our API to create a social application and then reflected on their experience. There were 9 participants in the study (P3 - P11) [3]. All were expert programmers with 3 to 10 years programming experience, except one, who withdrew due to a lack of object oriented programming experience. Five of the participants were male, three were female and the majority had no experience programming robots (one had six months experience working on a robotics related research project).

The specific tool used to evaluate the usability of our API was the cognitive dimensions questionnaire optimised for users [21]. The questionnaire is designed to present Cognitive Dimensions (CDs) in a way that end users of notations can readily understand [21]. The goal, is to enable end users, rather than designers, to evaluate a system with the CDs Framework [21].

A. Method

Before participants started the study, they completed a background questionnaire concerning their programming experience (summarised above). The study itself consisted of four phases:

1. Play game show with researcher and robot (5 minutes).
2. Read API documentation (20-30 minutes).
3. Complete a set of tasks (30-40 minutes).
4. Reflect on experience by completing a Cognitive Dimensions Questionnaire Optimised for Users (30 minutes).

Participants first interacted with the robot to understand the types of interactions Nao was capable of. This interaction was a multiplayer game show (Figure 2) where the Nao robot acts as the game show's host. Nao interacts with two teams of people autonomously, asking aloud a multiple choice question for each round of the game (making speech). As Nao speaks to people, it gazes at them and makes gestures synchronised with its speech (making gestures). Each team has a button to press to answer a question. A team's verbal response is recognized (understanding human speech): If the answer is correct then Nao increases the team's score, otherwise Nao does one of two things: give the other team a chance to answer or subtract points from the team that got the question wrong (dialogue). After a set number of questions Nao announces the winner and loser of the show.

After interacting with the robot, participants spent 20-30 minutes reading the API documentation; class documentation and an example program. This overviews the API's most important classes, what their salient functions and attributes do and how they are used. In the example program, the robot greets a person and responds positively or negatively based on the user's response. The example is provided with step-by-step explanations of how each part works.

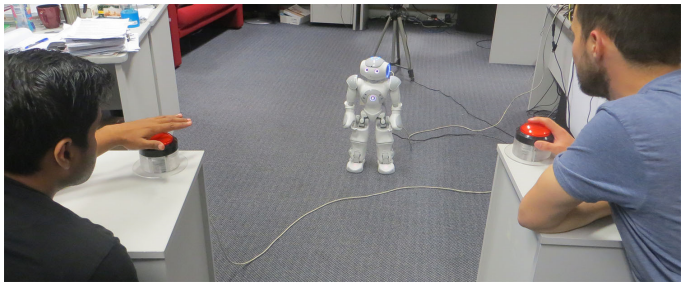

FIGURE 2. GAME SHOW SETUP.

The participants then conduct a series of tasks to convert the example program into a new scenario, a number guessing game. Here, Nao asks a person to guess what number Nao is thinking of. The person responds with a number from one to three. If the response matches Nao's number, Nao tells them they were correct, otherwise Nao tells them they were wrong. Participants were observed while they completed the tasks; during this time the researcher took notes and asked questions if there was a need to clarify why they programmed in a particular way.

At the conclusion of the tasks, participants were given the questionnaire [21] to reflect on their experience using our API to program social interaction.

B. Results

Results analysis consisted of classifying questionnaire responses by whether they were positive, equivocal or negative; based on how Blackwell & Green analysed responses in [21]. This was further broken down into general positive and negative responses and specific positive and negative responses. General responses just indicate whether the notation was acceptable with respect to a particular dimension; "yes", "no", "easy" and "hard" are examples of this [21]. Specific responses show how specific usability features perform against a particular dimension [21].

The general responses indicate an overall positive impression of the notation. Participants responded with 49 general positive comments and only 3 general negative comments. Specific reasons why participants had a positive impression include: its object oriented nature (P6); it is clear, concise and the class definitions are well thought out (P6, P8, P10); the notation is easy to understand (P10); and it allows programmers to express emotional emphasis and empathy on the robot (P8). Individual dimensions with the most general positive comments include role expressiveness (9), visibility & juxtaposability (8), closeness of mapping (6) and progressive evaluation (6).

The specific responses provide formative feedback about how specific usability features perform with respect to a particular

[3] P1 & P2 were pilot testers; their results were not included in the analysis.

dimension. Participants' positive and negative responses were fairly even, with 51 specific positive and 53 specific negative comments. The specific positive comments focused on a number of factors, including: the programming environment; the APIs: organisation, object oriented nature, and its "well thought out" domain specific nature; and being able to test interactions with the robot. The following paragraphs discuss these factors in the context of the Cognitive Dimensions Framework.

1) Visibility & Juxtaposability. Unsurprisingly participants stated that the programming environment (Eclipse) benefited visibility & juxtaposability. They didn't state how Eclipse specifically increased notation visibility, just that "programming in an IDE is convenient & familiar". In terms of juxtaposability, Eclipse allows a programmer to compare different parts of the notation side-by-side "using multiple windows." Participants also noted that the organisation of the API benefited visibility & juxtaposability, for example, when asked how easy it is to find various parts of the notation P10 responded that it was "simple because the organisation of the notation is clear and concise." A social application is organised so that general things such as setting up the environment and global queries are at the top, whereas defining states are further down (P8).

2) Closeness of Mapping. Participants indicated that the domain specific aspects of the notation had a close mapping to the programs they created, for example P4 stated that the notation was "pretty close in some parts (e.g. robot.say_to)." These parts of the API had a positive effect on the diffuseness & terseness of the notation (discussed next).

3) Diffuseness & Terseness. Participants' responses here indicate that the domain specific aspects of the API had a positive effect on the terseness of the notation. For example, P4 stated that the API lets you say what you want reasonably briefly because the notation "is domain specific". This is likely because domain specific languages have a close mapping to the problem domain they describe, allowing programmers to express what they want with fewer primitives than a non-domain specific language. Similarly, P6 said that the API was "Brief & concise as the API is well-written" and P8 said the notation lets you say what you want reasonably briefly because "Each element (class definition) was well thought out."

4) Role Expressiveness. The notations object oriented nature had a positive effect on role expressiveness. For example, P3 stated that it was easy to tell how each part of the API fits into the overall scheme of things because "the structures in this API are similar to those in any OO language."

5) Progressive Evaluation. The notation performed well with respect to two aspects of progressive evaluation: the ease of stopping and testing a notation and checking progress made when programming a solution. First, only a "basic structure is needed to run" a program, making it easy to stop in the middle to check your work (P4). Second, participants found it is easy to test their progress because they could directly interact with the robot to see if it was behaving how they wanted, for instance P7 stated that it was possible to test the progress she had made by "test[ing the] interaction directly with the robot" (P7).

The number of specific negative comments almost equalled the specific positive comments; however, over 60% of the specific negative responses (31 of 53) were related to one aspect of the API: the *StateMachine*. The purpose of the *StateMachine* and its associated classes and functions are to perform dialogue management; a higher level aspect of social interaction than the social primitives. The dimensions with the most specific negative responses for the *StateMachine* include: hard mental operations (5), diffuseness & terseness (6), error proneness (5) and premature commitment (4); these are discussed below.

1) Hard Mental Operations. Specific negative responses indicated programming state changes to perform dialogue management required much mental effort (5). E.g. P4 said "Probably moving between *States* in the state machine & passing arguments to the state" required the most mental effort. Other participants had similar views, but that the notation was easily grasped if this was understood. P3 commented he had "Some difficulty with queries/events/ State changes at first, but once that was figured out it was all fairly simple."

2) Diffuesness & Terseness. Responses about diffuseness & tersness indicated that the code required to transition the state machine was diffuse (6). For example, p4 responded that "many similar/grouped actions/events" took a lot of space to describe. Users have to instantiate several classes (*Query*, *QueryEvent*) and call several methods to create an event based state transition, which is likely the reason why this part of the notation is diffuse.

3) Error Proneness. Most responses here related to dialogue management via the state machine (5). Users reported they misnamed state id's, made mistakes due to copy and pasting queries, events and state transitions, left query & event declarations unused.

4) Premature Commitment. Lastly, the responses about premature commitment indicated participants thought that using the state machine forced them to think ahead and make decisions about dialogue before they needed to (4). For example, P3 commented "You would need to have an idea of what States you need in the app, and how you move between them. This would be easier to sketch out first rather than doing it within the API."

The remaining 22 specific negative comments were largely related to minor usability issues such as inappropriate function names and easily fixable inconsistencies.

In summary, the evaluation demonstrates users had an overall positive impression of the notation and that they specifically appreciated the programming environment, the API's organisation, its object oriented nature and domain specific interfaces. The part of the API that received the majority of specific negative comments was the method of managing dialogue and is an area for future improvement.

VI. Discussion and conclusions

We have described our API for programming robot social interactions. Our API overcomes many of the disadvantages of existing tools for programming social interaction as it provides high level, domain specific interfaces for programming social interaction. It also supports a broader range of social interaction requirements than other existing tools. These include requirements for making speech, making gestures, modelling the human body and understanding human speech.

The evaluation demonstrates that users had an overall positive impression of the notation, as the vast majority of general responses were positive. Specific factors that users thought benefited the notation include the programming environment, the API's organisation, its object oriented nature, it's "well thought out" domain specific interfaces and being able to test interactions directly with the robot. These positively affected: visibility & juxtaposability, closeness of mapping, diffuseness & terseness, role expressiveness and progressive evaluation.

The majority of specific negative responses related to one of notational aspect, its means to express dialogue, a higher level aspect of social interaction than social interaction primitives. This negatively affected hard mental operations, diffuseness and terseness, error proneness and premature commitment. This shows a better language for managing robot dialogue is needed; for both programmers and end users. For end users, a possible solution is to represent dialogue with a visual language.

Our API is a first step in a more general study of tools for programming human robot interaction. Our intention is to extend the API to target other aspects of human robot interaction. By examining other scenarios such as social interaction, a fetch & carry task and a robot guide scenario, we expect to create a more general framework to program human robot interaction that has a much higher abstraction level and better support for the requirements of human robot interaction than existing tools such as Choregraphe [5], Interaction Composer [7] and BML [9].

Acknowledgement

The authors thank the reviewers for their helpful feedback, the University of Auckland PhD Scholarship programme for financial support, the participants of the user study, Chandan Datta and Adam Roughton for advice and the University of Auckland HCI Research group for their feedback on the paper (particularly Andrew Luxton-Reilly).

References

[1] T. Fong, I. Nourbakhsh, and K. Dautenhahn, "A survey of socially interactive robots," *Robot. Auton. Syst.*, vol. 42, no. 3–4, pp. 143–166, Mar. 2003.

[2] C. D. Kidd, W. Taggart, and S. Turkle, "A sociable robot to encourage social interaction among the elderly," in *Proc. 2006 IEEE ICRA*, Florida, USA, 2006, pp. 3972–3976.

[3] N. Mavridis and D. Hanson, "The IbnSina Center: An augmented reality theater with intelligent robotic and virtual characters," in *Proc. 18th IEEE RO-MAN*, Toyama, Japan, 2009, pp. 681–686.

[4] K. Dautenhahn, S. Woods, C. Kaouri, M. L. Walters, K. L. Koay, and I. Werry, "What is a robot companion - friend, assistant or butler?," in *Proc. 2005 IEEE/RSJ IROS*, Edmonton, Canada, 2005, pp. 1192–1197.

[5] E. Pot, J. Monceaux, R. Gelin, and B. Maisonnier, "Choregraphe: a graphical tool for humanoid robot programming," in *Proc. 18th IEEE RO-MAN*, Toyama, Japan, 2009, pp. 46–51.

[6] Aldebaran Robotics, "NAOqi modules API's." [Online]. Available: http://www.aldebaran-robotics.com/documentation/naoqi/.

[7] D. F. Glas, S. Satake, T. Kanda, and N. Hagita, "An interaction design framework for social robots," in *Proc. Robotics: Science and Systems*, Sydney, Australia, 2012, vol. 7, p. 89.

[8] T. Lourens and E. Barakova, "User-Friendly Robot Environment for Creation of Social Scenarios," in *Foundations on Natural and Artificial Computation*, Springer, 2011, pp. 212–221.

[9] S. Kopp, B. Krenn, S. Marsella, A. N. Marshall, C. Pelachaud, H. Pirker, K. R. Thórisson, and H. Vilhjálmsson, "Towards a Common Framework for Multimodal Generation: The Behavior Markup Language," in *Intelligent Virtual Agents*, Springer, 2006, pp. 205–217.

[10] A. Holroyd and C. Rich, "Using the Behavior Markup Language for human-robot interaction," in *Proc. 7th ACM/IEEE HRI*, Boston, USA, 2012, pp. 147–148.

[11] J. P. Diprose, B. Plimmer, B. A. MacDonald, and J. G. Hosking, "How People Naturally Describe Robot Behaviour," in *Proc. ACRA*, Victoria Univ., Wellington, New Zealand, 2012.

[12] Acapela Group, "Text Tag Documentation, Acapela TTS For Mobile."

[13] H. Vilhjálmsson, N. Cantelmo, and J. Cassell et. al., "The Behavior Markup Language: Recent Developments and Challenges," in *Intelligent Virtual Agents*, Springer, 2007, pp. 99–111.

[14] D. Roberts and R. Johnson, "Evolving Frameworks: A Pattern Language for Developing Object-Oriented Frameworks," in *Pattern Languages of Program Design 3*, Boston, MA, USA: Addison-Wesley, 1997.

[15] *Furhat - a robot that plays quiz games*. 2013.

[16] I. Kruijff-Korbayová, G. Athanasopoulos, and A. Beck et. al., "An Event-Based Conversational System for the Nao Robot," in *Proc. of the Paralinguistic Information and its Integration in Spoken Dialogue Systems Workshop*, Springer, 2011, pp. 125–132.

[17] T. R. G. Green and M. Petre, "Usability Analysis of Visual Programming Environments: A 'Cognitive Dimensions' Framework," *J. Vis. Lang. Comput.*, vol. 7, no. 2, pp. 131–174, Jun. 1996.

[18] Microsoft Corporation, "LINQ (Languge Integrated Query System)." [Online]. Available: http://msdn.microsoft.com/en-us/library/vstudio/bb397926.aspx.

[19] R. Smallshire, "asq - A Python implementation of LINQ and parallel LINQ to objects." [Online]. Available: https://code.google.com/p/asq/.

[20] C. Datta, B. A. MacDonald, C. Jayawardena, and I.-H. Kuo, "Programming Behaviour of a Personal Service Robot with Application to Healthcare," in *Social Robotics*, Springer, 2012, pp. 228–237.

[21] A. F. Blackwell and T. R. Green, "A Cognitive Dimensions questionnaire optimised for users," in *Proc. 12'th Annu. Psychology of Programming Interest Group*, 2000, pp. 137–152.