Eliciting Expectations for Monitoring Social Interactions

Michael Winikoff[1] and Stephen Cranefield[2]

[1] School of Computer Science and Information Technology
RMIT University
Melbourne, Australia
michael.winikoff@rmit.edu.au
[2] Department of Information Science
University of Otago
Dunedin, New Zealand
scranefield@infoscience.otago.ac.nz

Abstract. The use of computers to mediate social interactions (e.g. blogs, chatting, facebook, second life) creates the possibility of providing software to support social awareness in a range of ways. In this paper we focus on monitoring expectations and consider how a user who is not a programmer or logician might specify expectations to be monitored. We propose a novel approach where the user provides a collection of scenarios, and then candidate formulae are induced from the scenarios. The approach is applied to examples and appears to be promising.

1 Introduction

In recent years, advances in Web application development, the increased availability of broadband internet access, and changing end-user engagement with the Web have caused a growing trend for the internet to be used as a medium for social interaction. The popularity of blogs has increased enormously over the past few years, as have other types of online information and opinion sharing applications such as as photo and video sharing, wikis, chat and short message services, social networking sites such as Facebook, and virtual worlds such as Second Life.

However, while these "Web 2.0" applications provide the computational and communications infrastructure to *enable* interaction, they generally provide no support for users to maintain an awareness of the *social context* of their interactions (other than basic presence information indicating which users in a "buddy list" are online).

In contrast, researchers in the field of multi-agent systems (MAS) have adopted, formalised and created computational infrastructure allowing concepts from human society such as trust, reputation, expectation, commitment and narrative to be explicitly modelled and manipulated in order to support social awareness amongst open communities of interacting autonomous software agents [1]. This awareness helps agents carry out their interactions efficiently and helps preserve order in the society, e.g. the existence of reputation, recommendation and/or sanction mechanisms discourages anti-social behaviour.

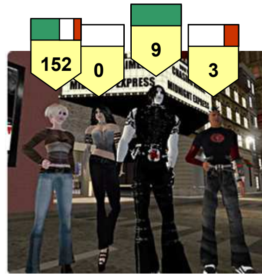

Fig. 1. Possible depiction of reputation information in Second Life

There is therefore an opportunity for the techniques developed in MAS research to be applied in the context of electronically mediated human interaction as resuable Web-based services. For example, a service to maintain users' reputations could usefully be "mashed up" with wikis (e.g. consider Wikipedia) and virtual worlds. Figure 1 illustrates how reputations might be visualised within an extended Second Life client: coloured bars[3] record the percentage of positive, neutral and negative ratings, based on the indicated number of encounters.

To maintain reputations, it is necessary to aggregate information about the social behaviour of participants in a society. This can be done by collecting opinions from other participants, or by directly comparing participants' observed behaviour to some pre-existing expectations. This paper focuses on the latter process: the monitoring of expectations on user behaviour that have been modelled formally and publicised for a given community. In previous work, we have investigated the use of temporal logic to model expectations with a rich temporal structure, e.g. "Once payment is made, the service-providing agent is committed to sending a report to the customer once a week for 52 weeks or until the customer cancels the order", and developed tools that enable the detection of fulfilment and violations of expectations modelled in this way [2, 3].

This type of service would be a useful component of a reputation mechanism, but is also valuable in its own right as it allows users or the managers of communities to be notified when observed behaviour deviates from expectations. However, these expectations must first be defined formally by application or scenario designers, community managers, or even by individual users who wish to invoke the monitoring service. While the use of temporal logic has great theoretical advantages, it is unlikely that managers and users of an online community would be comfortable with this notation. Therefore, in this paper we address the problem of how can we assist users, including non-programmers and non-logicians, to specify expectations in temporal logic? Clearly, it is important to avoid logical symbols, and to provide facilities for "animating" formulae so a specifier can ensure that what they said is what they meant. One approach is to use a graphical syntax for temporal logic. Another is to derive formulae from alternative specification notations such as interaction protocols. However, both of these approaches have problems, and so we propose a novel third approach (inspired by programming-

[3] From left to right the bars are green, white and red.

by-demonstration): the user provides example scenarios, and candidate formulae are derived from these. This approach avoids requiring from the user familiarity with logic (rendered graphically or otherwise), or with alternative notations (such as interaction protocols). The notation used to provide scenarios is extremely simple, and, we believe, well within the reach of non-programmers.

The remainder of this paper is organised as follows. Section 2 briefly introduces Linear Temporal Logic (LTL) and gives finite trace semantics. Section 3 presents motivating examples and in Section 4 we discuss two approaches for specifying formulae: graphical syntax and alternative notations. In Section 5, the core of the paper, we present our alternative method; and we conclude in Section 6.

2 Logical Preliminaries

In this section we briefly review temporal logic and give some basic definitions. The logic that we use is LTL (Linear Temporal Logic). Note that since we are concerned with models that are traces, i.e. that are non-branching ("sticks" rather than trees), we do not need to distinguish between "all paths" and "some paths" as is done in more sophisticated logics, such as CTL.

The language we use is defined by the following grammar, where $\Box$ is "always", $\Diamond$ is "eventually", $\bigcirc$ is "next", and U is "until"; and where p denotes a proposition, and $\top$ is "true". We define typical abbreviations ($\phi \vee \psi \equiv \neg((\neg\phi) \wedge (\neg\psi))$ and $\bot \equiv \neg\top$). Note that some of the connectives can be viewed as derived rather than primitive (specifically, $\top$, $\Diamond$ and $\Box$), but we define them directly for clarity.

$$\phi \ ::= \ \top \mid p \mid \neg\phi \mid \phi_1 \wedge \phi_2 \mid \bigcirc\phi \mid \Diamond\phi \mid \Box\phi \mid \phi_1 \,\mathsf{U}\, \phi_2$$

Semantics for LTL are normally given in terms of an *infinite* model, but for our purposes we need to define semantics over *finite* traces. We do this, following [4], by defining $\phi \,\mathsf{U}\, \psi$ as holding with respect to model $\mathcal{M}$ and index i if i denotes a state in $\mathcal{M}$ and the usual conditions apply. This allows the usual definitions of $\Diamond\phi \equiv \top \,\mathsf{U}\, \phi$ and $\Box\phi \equiv \neg\Diamond\neg\phi$ to be used.

Our semantics are given in Figure 2. We use $\mathcal{M}$ to denote a sequence of states of length $|\mathcal{M}|$ where each state is a set of propositions. We use $\mathcal{M}_i$ to denote the ith element of $\mathcal{M}$, and we use $i \in \mathcal{M}$ as shorthand for $1 \leq i \leq |\mathcal{M}|$. One interesting point is the additional case on $\Box$: because $\mathcal{M}, i \models \phi \,\mathsf{U}\, \psi$ requires $i \in \mathcal{M}$ and $\Box\phi$ is defined as $\neg(\top \,\mathsf{U} (\neg\phi))$, then $\Box\phi$ must also be true outside of the boundaries of the trace. Roughly speaking, this can be thought of as viewing $\Box p$ as really being short for $p \,\mathsf{U}\, end$ where end is true in the last state in the trace.

3 Motivating Examples

In this section we present a number of examples of expectations that can be usefully checked. These examples include original examples that we have developed, as well as examples that appear in the literature. Note that we have focussed on literature in the area of agents rather than the literature on software specification (e.g. [5]). The reason

$$
\begin{array}{rll}
\mathcal{M}, i \models \top & & \\
\mathcal{M}, i \models p & \text{iff} & i \in \mathcal{M} \text{ and } p \in \mathcal{M}_i \\
\mathcal{M}, i \models \neg\phi & \text{iff} & \mathcal{M}, i \not\models \phi \\
\mathcal{M}, i \models \phi_1 \wedge \phi_2 & \text{iff} & \mathcal{M}, i \models \phi_1 \text{ and } \mathcal{M}, i \models \phi_2 \\
\mathcal{M}, i \models \bigcirc\phi & \text{iff} & i \in \mathcal{M} \text{ and } \mathcal{M}, i+1 \models \phi \\
\mathcal{M}, i \models \Diamond\phi & \text{iff} & i \in \mathcal{M} \text{ and } \exists k \text{ such that } i \leq k \leq |\mathcal{M}| : \mathcal{M}, k \models \phi \\
\mathcal{M}, i \models \Box\phi & \text{iff} & \forall k \text{ such that } i \leq k \leq |\mathcal{M}| : \mathcal{M}, k \models \phi \\
 & & \text{or } i \notin \mathcal{M} \\
\mathcal{M}, i \models \phi_1 \mathsf{U}\, \phi_2 & \text{iff} & i \in \mathcal{M} \text{ and} \\
 & & \exists k \text{ such that } i \leq k \leq |\mathcal{M}| : \mathcal{M}, k \models \phi_2 \text{ and} \\
 & & \forall j \text{ such that } i \leq j < k : \mathcal{M}, j \models \phi_1
\end{array}
$$

Fig. 2. Semantics for Linear Temporal Logic over Finite Traces

is that it is not clear that properties that we would want to check in software systems are the same as those we would want to check in social interactions.

For each of the examples below we give an informal description, a formula that captures formally the desired property, and a citation (if the property comes from the literature). Each of the formulae ϕ gives a rule that should be followed and, for actual use, would be a sub-formula of a larger formula that gave the context in which the rule should apply, e.g. ($\Box\phi$).

- No noise during lectures ($startLecture \rightarrow (\neg noise) \mathsf{U}\, endLecture$)
- It will rain until midnight ($rain \mathsf{U}\, midnight$) [6]
- an agent that makes a booking cannot leave the interaction without paying ($booking \rightarrow \Diamond pay$) [7]
- bids placed (in an auction) must be larger than preceding bids ($bid(N) \wedge \bigcirc\Diamond bid(M) \rightarrow M > N$) [7]
- once an auction is opened is must eventually end (*openAuction* $\rightarrow \Diamond$*endAuction*) [8]
- Once an order is placed, no more orders may be placed until payment is made (for the first order) ($order \rightarrow \bigcirc((\neg order) \mathsf{U}\, pay)$) [3]

Clearly the first (no noise during lectures) and last (no second order until payment) are closely related; furthermore, the formulae for the auction being eventually ended and payment being eventually made are identical in structure.

4 Eliciting Expectations

In this section we explore the issues of how to specify expectations and how to represent them. Loosely speaking, there are two overall approaches:

1. Specify expectations using an alternative (graphical) syntax for temporal logic

2. Specify expectations in a different formalism, and derive temporal logic expressions from this formalism

Regardless of which of these approaches is used, it would be naive to expect a user to always specify correctly what they meant, and so it is important to have a means of applying a formula to provided examples to see whether it behaves as the user intended. This can be seen as a (very limited) form of specification animation [9], and can be easily done by the same mechanism that will be used at deployment time to check actual traces. Another approach that may assist users in checking that what they have specified is correct is to render the formula in a limited subset of natural language (see Section 5.1).

In the next section we will explore a novel third approach inspired by programming-by-demonstration [10, 11]: instead of having the user specify the expectation, the user provides a number of examples, and the formula is derived from these examples. But first, we consider the two existing approaches.

The first existing approach, which retains the power and precise semantics of logic but without the unfriendly syntax (to non-logicians), is to use a graphical rendition/visualisation of temporal logic.

A number of visualisations have been proposed in previous work. One of the better-known ones is GIL [12]. Another interesting approach that uses 3D visualisation is that of Del Bimbo *et al.* [13]. Space precludes an exhaustive survey, see [14, section 4] for a brief survey with additional references.

Figure 3 (left side) shows the formula $order \rightarrow ((\neg order) \mathsf{U}\, pay)$ rendered in the notation of [13]. A few observations can be made. Firstly, the observant reader will note that the formula is missing the $\bigcirc$: this is because the graphical notation does not include this connective. On the other hand, the notation does distinguish between something holding on all paths, or on some path, a distinction which we do not need. Although the notation is graphical, it still retains the use of the implication symbol, which undermines it's accessibility to those not familiar with symbolic logic.

The right side of Figure 3 shows the formula $(\neg order) \mathsf{U}\, pay$ in GIL (based on the work of Dillon *et al.* [12, equation 2]). The top dashed line denotes a search that finds the first state where $order \vee pay$ holds. We then search from there (second dashed line) to the end of the trace resulting in an interval (the solid line) where pay holds at the start of the interval. The intuition is that the first state in which $order \vee pay$ holds is in fact one in which pay holds, so there are no preceding states in which $order$ holds but pay doesn't. Two observations about this notation are that even a basic formula is somewhat complex to understand, and that logical symbols are used. The GIL notation is inspired by timing diagrams, and is appropriately used to capture timing constraints by engineers familiar with timing diagrams, but we do not feel that it is likely to be usable by a user who is not familiar with logic and with timing diagrams.

We thus conclude that although the use of a graphical syntax is promising, more work is needed to develop a syntax that is really usable by a wide range of users.

Another approach is to specify expectations in an alternative formalism, and derive the logical expectations from this alternative formalism. In a sense this approach merely defers the question: instead of having to develop a usable graphical syntax for temporal logic, we now need to develop a usable (typically graphical) notation, and then develop

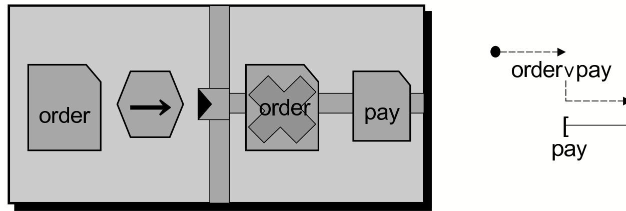

Fig. 3. Logical formulae in graphical form

a mapping from it to temporal logic. The attractiveness of this approach comes from the prior existence of a wide range of possible alternative notations including design notations such as the Unified Modeling Language (UML).

Given that we are interested in specifying properties of events in time, two classes of notations that suggest themselves as natural fits are those for specifying interaction protocols and those for specifying processes.

Although it is certainly feasible to derive a temporal logic formula corresponding a certain protocol, it is less clear how to specify arbitrary temporal logic formulae using a process diagram or interaction protocol. For instance, what would a UML sequence diagram for $\phi \cup \psi$ look like?

Having considered the two existing approaches, and found issues with both of them, in the next section we propose our new alternative approach.

5 Specifying Expectations by Examples

An alternative to having the user specify expectations directly, is to use examples to indirectly specify the desired expectations. In this approach, which is inspired by programming-by-demonstration (PBD) [10, 11], the user provides a number of example traces, indicating for each trace whether it is desirable or undesirable. The system takes these positive and negative examples and induces a formula.

Unfortunately, given a finite (and usually fairly small) number of examples, there will be *many* formulae which could be used, and so the best that the system could do is to try and find a collection of plausible candidates and present them to the user. The user then needs to be able to determine which of the alternative formulae matches their intention.

Clearly, presenting the user with a collection of logical formulae to peruse is not practical, and so we need to render them in a user-friendly format, such as one of the graphical formalisms discussed earlier, or an alternative format. Both these options are feasible, although generating a graphical representation of a formula, complete with layout information, is not trivial. However, here we consider a third option: (constrained) natural language. Note that although it is difficult to use natural language as an input, generating natural language as an output is considerably easier. That is, it is feasible to use (constrained) natural language because in our approach it plays the role of output, not input.

Thus the overall process is:

1. User provides examples (both positive and negative)
2. System induces candidate formulae and presents them to the user in natural language
3. If there are too many candidates, then the user provides more examples (return to step 1)
4. The user selects the formula that matches their intention; if there is no such formula then the system is asked to find more candidates or the user removes some of the examples and then asks the system to re-generate candidates (i.e. return to step 2).

The remainder of this section focuses on the operation of the system (step 2) and considers the two things it needs to do, namely inducing candidate formulae from examples, and rendering them in constrained natural language.

5.1 Rendering Temporal Logic in Constrained Natural Language

It is fairly straightforward to render a temporal logic formula in constrained natural language by replacing each connective with its English equivalent. For example, $order \rightarrow ((\neg order) \mathsf{U}\, pay)$ would become "order implies that not order until pay". However, the sorts of English sentences generated by this naive process do not tend to be easy to understand.

While a generic solution is clearly difficult, we observe that many of the expectations being specified are simple in structure, and that many cases can be handled through a collection of patterns (see Figure 4). While this approach is *ad hoc* in nature, it is simple, and, we believe, sufficient for our purposes. Where patterns do not apply, the default is to replace each connective with an English equivalent (i.e. $\bigcirc$ is "next", $\Box$ is "always", $\Diamond$ is "eventually", $\wedge$ is "and", etc.). In Figure 4 the phrase "left inclusive" reflects the asymmetrical nature of the until connective: $p \,\mathsf{U}\, q$ holds if p holds from every state *including the current state*, until, but *not* including the state in which q holds. If we view the sequence of states as progressing from left to right then the initial state (on the left) is included, but not the final state (on the right), hence "left inclusive".

Formula	**English**
$\phi \rightarrow ((\neg\psi) \mathsf{U}\, \tau)$	"ψ must not be between ϕ and τ (left inclusive)"
$\phi \rightarrow (\psi \,\mathsf{U}\, \tau)$	"ψ must be between ϕ and τ (left inclusive)"
$\phi \rightarrow \Diamond\psi$	"ϕ is followed (now or later) by ψ"
$\phi \rightarrow \bigcirc\Diamond\psi$	"sometime after ϕ, ψ"
$\phi \rightarrow \bigcirc\psi$	"just after ϕ, ψ"
$\phi \rightarrow \psi$	"when ϕ then ψ"
$\bigcirc\Diamond\phi$	"later ϕ"

Fig. 4. Translating Temporal Linear Logic to Natural Language

Applying these patterns to the examples discussed in Section 3 we obtain the following[4]:

- "noise must not be between startLecture and endLecture (left inclusive)" (*startLecture* $\rightarrow$ $(\neg$*noise*$)$ U *endLecture*)
- "rain until midnight" (*rain* U *midnight*)
- "booking is followed (now or later) by pay" (*booking* $\rightarrow \Diamond$*pay*)
- "when bid(N) and later bid(M) then M>N" ($bid(N) \wedge \bigcirc\Diamond bid(M) \rightarrow M > N$)
- "open-auction is followed (now or later) by end-auction" (*open-auction* $\rightarrow \Diamond$*end-auction*).
  Interestingly, this highlights that it is acceptable for open and end auction to occur simultaneously, which may not have been the user's intention.
- "just after order, not order until pay" ($order \rightarrow \bigcirc((\neg order) \mathsf{U}\, pay)$)

5.2 Inducing Candidate Formulae

Our aim in this section is to explore the feasibility of inducing candidate formulae from examples. We leave developing a detailed (efficient) mechanism for induction — perhaps based on inductive logic programming [15] — for future work. In order to investigate the feasibility of this approach the key question is this: given a reasonable number of examples, how many candidate formulae will be found? If the number of candidate formulae is very large then the approach is not feasible.

In order to explore this we have developed software that performs induction using a generate-and-test method: it generates all formulae (up to a certain specified complexity, specified in terms of the number of connectives in the formula, termed its *depth*) and for each formula tests whether it satisfies all of the examples provided. A number of logical equivalences are used to reduce the number of generated formulae, for example, $\Box\Box\phi \equiv \Box\phi$ and so we don't generate formulae of the form $\Box\Box\phi$. The generation algorithm, which incorporates the equivalences, can be found in the Prolog language in appendix A.

Scenarios can be specified in a number of ways. The simplest is to give a collection of models, where each model is a list of states, and where each state is a list of propositions that are true in that state. However, by slightly varying the specification we can extract some additional information from the user.

Before we discuss these variations, note that the notation used in the remainder of this section to represent scenarios (e.g. $\{o\} \rightarrow \{p\} \rightarrow \{o\}$) is a concise formal notation. In an actual tool we would expect that scenarios would be depicted graphically (as in Figure 5). From the user's perspective a model can be thought of as the trace of an interaction, with the occurrence of events of interest modelled by propositions.

The first variation is to have the user specify for each pair of adjacent states in a model whether they must occur immediately next to each other, or whether there could be intervening states that are not shown. For example specifying[5] $\{p\} \rightarrow \{q\}$ would

[4] Recall that if none of the patterns of Figure 4 apply then we replace each connective with an English equivalent.

[5] We use a set of propositions to denote a state.

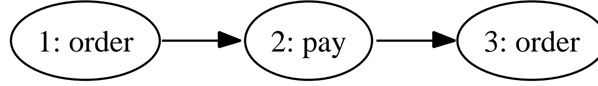

Fig. 5. Scenario in user-friendly notation

mean that the second state (in which q is true) must be the next state after the first state (in which p is true). Alternatively, $\{p\} \leadsto \{q\}$ specifies that the first state is followed by the second state, but that there may be intervening states (not shown), i.e. it is shorthand for $\{p\} \rightarrow \ldots \rightarrow \{q\}$ where "..." stands for zero or more unspecified states. In fact, in some situations we don't want to have unspecified states, but want to specify some properties of the intermediate states. For example, we may want to say that a number of states can be inserted, so long as no additional orders are in those states. We denote this as $\overset{c}{\leadsto}$ where c is a condition that must be true of the additional states. We treat $\leadsto$ as being shorthand for $\overset{\top}{\leadsto}$, i.e. the condition is true, so any inserted state will do.

In effect, using this variation allows a single scenario to be expanded into a number of scenarios by replacing $\leadsto$ with $\rightarrow$ followed by zero or more repetitions of "$s \rightarrow$", where s is some state that satisfies the condition c. In our experiments we replace $\leadsto$ with either "$\rightarrow$" or "$\rightarrow s \rightarrow$" (i.e. just zero or one). For example, $\{book\} \overset{\neg pay}{\leadsto} \{\neg pay\}$ is expanded in our experiments to the scenarios $\{book\} \rightarrow \{\neg pay\}$ and $\{book\} \rightarrow s \rightarrow \{\neg pay\}$ for all states s which satisfy $\neg pay$.

The second variation is to have the user specify explicitly what is false, and anything that is unspecified can be either true or false. So $\{p, \neg q\}$ would denote a state in which p is true, q is false[6], and we don't know or care about any other propositions. Thus, given an alphabet of propositions such as p, q, r a state $\{p, \neg q\}$ is equivalent to two possible states: $\{p, \neg q, r\}$ and $\{p, \neg q, \neg r\}$.

In effect, using this variation allows a single scenario to be expanded into a number of scenarios by filling in missing propositions. For any state that does not contain a proposition r or its negation we construct two scenarios: one where the r is added to the state, and another where $\neg r$ is added. If there are any incompletely specified states remaining then this process is repeated.

Let us now consider for example the fomula $order \rightarrow \bigcirc((\neg order) \mathsf{U}\, pay)$. We assume that the user has specified the following scenarios (where p abbreviates "pay" and o abbreviates "order"):

1. $\{o\} \overset{\neg o, \neg p}{\leadsto} \{p\} \leadsto \{o\}$ is ok
2. $\{o\} \overset{\neg p}{\leadsto} \{o\} \leadsto \{p\}$ is not ok
3. $\{\neg o\} \rightarrow \{\neg o\}$ is ok

For the second variation we assume that the user explicitly specified negations in the first two scenarios as follows:

1. $\{o\} \rightarrow \{p, \neg o\} \rightarrow \{o\}$ is ok
2. $\{o\} \rightarrow \{o, \neg p\} \rightarrow \{p\}$ is not ok

[6] In fact we would write $\neg q$ as a crossed out q.

The following table summarises the number of candidate formulae generated at different depths with the different variations. Generating all four cases with depths 1, 2, and 3 took 3.4 seconds[7]. Generating the depth 4 figures (for all four cases) took 5 minutes and 16 seconds.

"One order at a time"	depth 1	depth 2	depth 3	depth 4
A: Base case	1	19	390	8976
B: Variation 1 ($\rightarrow$ vs. $\rightsquigarrow$)	0	3	117	3261
C: Variation 2 (explicit negation)	1	9	103	1849
D: Both Variations	0	0	3	53

The single option returned for the case A and for case C at depth 1 is $\bigcirc\neg o$ ("next not order"), which is clearly too simplistic, and would be rejected by the user. For case B and depth 2 the three candidate formulae are: $\Box\Diamond\neg p$ ("always eventually not pay"); $\Box(p \mathsf{U} \neg p)$ ("always pay until not pay"); and $\Diamond\Box\neg p$ ("eventually always not pay"). Again, none of these capture the user's intention.

For case D at depth 3 the candidate formulae are: $o \rightarrow \bigcirc((\neg o) \mathsf{U} p))$ ("just after order, not order until pay"); $(\bigcirc((\neg p) \mathsf{U} o)) \rightarrow \neg o$ ("when next not pay until order then not order")[8]; and $(\bigcirc((\neg o) \mathsf{U} p)) \mathsf{U} \neg o$ ("(next (not order until pay)) until not order")[9]. The first of these three candidates is precisely the user's intention.

Let us now consider the expectation that bookings must eventually be paid for ($booking \rightarrow \Diamond pay$). We assume that the user simply specifies a scenario where booking is followed by paying as being ok, and a second scenario where booking is not followed by payment as being not ok:

1. $\{book\} \rightarrow \{pay\}$ ok
2. $\{book\} \rightarrow \{\}$ not ok

For the variations we assume that payment is required to not hold until the final state (first scenario) or anywhere (second scenario):

1. $\{book, \neg pay\} \overset{\neg pay}{\rightsquigarrow} \{pay\}$ ok
2. $\{book, \neg pay\} \overset{\neg pay}{\rightsquigarrow} \{\neg pay\}$ not ok

Results are in the table below (on the left of the "/"), and took 2 seconds for depths 1-3 (all cases), and approx. 50 seconds for depth 4 (all cases). None of the formulae at depth 1 match the user's intent, and let us suppose that the user does not have the patience to wade through 50 formulae to find the one that matches their intent for depth 2. In this situation (according to the process discussed at the start of this section), the user would specify additional scenarios. One scenario that might well be added is saying

[7] Average of three runs with B-Prolog 7.0 on a 2.16 GHz Intel Core 2 Duo Mac Powerbook Pro with 1 Gig of 667 MHz RAM running Mac OS X 10.4.11.

[8] In fact using the operator $\phi \mathsf{R} \psi \equiv \neg((\neg\phi) \mathsf{U} (\neg\psi))$ and the equivalence $\neg(\phi \mathsf{U} \psi) \equiv (\neg\phi) \mathsf{R} (\neg\psi)$, as well as reversing the implication this can be rewritten as $o \rightarrow \bigcirc(p \mathsf{R} \neg o)$, i.e. "after order, pay releases not order", which is equivalent to the previous expectation.

[9] Brackets added to clarify. Note that if o holds in the first state but not the second then this is equivalent to $\bigcirc((\neg o) \mathsf{U} p)$, so is related to the desired expectation.

that it's also ok if there have been no orders ($\{\neg book\}$ is ok). This results (1.3 seconds for depths 1-3, 29.7 seconds for depth 4) are on the right side of the "/" in the table below.

"Bookings are paid for"	depth 1	depth 2	depth 3	depth 4
A: Base case	4 / 0	102 / 12	2209 / 290	49,090 / 6,641
B: Variation 1 ($\rightarrow$ vs. $\rightsquigarrow$)	2 / 0	59 / 5	1408 / 148	33,157 / 3,823
C: Variation 2 (explicit negation)	4 / 0	86 / 8	1615 / 140	34,048 / 2,918
D: Both Variations	2 / 0	50 / 3	1045 / 71	22,983 / 1,634

The three candidate formulae at depth 2 (case D) are:

1. $book \rightarrow \Diamond pay$ ("book is followed (now or later) by pay") which is the user's intention;
2. $book \rightarrow ((\neg pay) \cup pay)$ ("pay must not be between book and pay (left inclusive)") which is actually equivalent to the previous, but is somewhat more convoluted; and
3. $(\Box\neg pay) \rightarrow (\neg book)$ ("when always not pay, then not book") which is also equivalent[10].

The key point here is that even though some of the English (and corresponding formulae!) are obfuscated, the desired formulae are clearly understandable, and there are only a few other alternatives. It thus appears that inducing candidate formulae from example scenarios is feasible both in terms of efficiency, and in terms of the number of candidate formulae not being too large. Of course, we have only considered two example formulae, but, from the examples we have seen, these seem to be typical of sorts of formulae that are often specified.

6 Discussion

We have discussed the issue of eliciting expectations to be checked over (traces of) electronically-mediated social interactions. For the intended use by non-logicians and non-programmers it is important that the means of specifying desired expectations not require expertise with logics or formal notations which rules out existing work on logic and most work on graphical notations.

Our solution is to use a process where the user gives examples, rather than some representation of a formula, and candidate formulae are derived from the examples. We have shown for a few examples that this approach is feasible, i.e. that the desired formula is able to be derived, and that its English rendition is readable.

One issue when expectations are applied to interactions that involve humans is that humans will need to know what expectations will apply to a given interaction. This can be done by having signs of some sort that graphically or in natural language communicate norms and other expectations, analogous to a "no smoking" sign. Note that communicating expectations to software entities is relatively simple - library code can be provided that processes expectations in some suitable format (e.g. XML).

[10] Reversing the implication yields $book \rightarrow \neg\Box\neg pay$.

Another issue that arises in applying this work is that propositions need to be "grounded" in reality. That is, when monitoring a condition, the system needs to know that a proposition such as p corresponds to a certain action being performed, and needs to have a mechanism for detecting when the action has been performed. For example, the bank may inform the system when deposits are made into an account.

Using scenarios, rather than formulae, to specify desired properties of software systems (as opposed to social interactions) has been explored previously. Alfonso *et al.* [16] present a graphical notation for scenarios, but their notation, although considerably simpler than graphical renditions for logic, still seems to be too complex for use by non-programmers. Furthermore, they do not consider inducing formulae from a collection of scenarios: rather, each scenario corresponds to a desired (or, for an "anti-scenario", prohibited) sequence of events.

The vision of Harel [17] where specifications are derived by "playing in" behaviour is related to our work, but instead of having the system generalise from provided scenarios, Harel extends the language of scenarios and requires the playing in process to include generalisation, i.e. the generalisation is done by the human, and the notation used is significantly more complex than our simple traces.

Monitoring conditions that are specified using design notations can be seen as related to work that has been done on using design artifacts (such as interaction protocols) for debugging [18], where the idea is to monitor a running (purely software, i.e. no humans) system and detect violations of expectations. There has also been work on using temporal logic to represent interaction protocols (specified as finite state machines) [8], but this work doesn't replace logic with protocols: the conditions to be checked are still specified (directly) in temporal logic.

The induction process appears to be related to work in the area of formal concept analysis [19], where concepts are placed in lattices, which allows for the determination of (e.g.) whether concept A is more specific than concept B, or finding objects that have certain common attributes. The existence of well-defined relationships is of potential use in the induction process, for instance, being able to determine whether a formula is more specific than another formula, but it needs to be stressed that the notion of a concept is much simpler than the notion of a formula: a concept is simply a set of objects and their associated attributes. Thus, applying formal concept analysis ideas in our context would require significant further work.

Areas for future work include developing an efficient mechanism for inducing candidate formulae, and performing more extensible experimentation with more complex examples, as well as an assessment with human subjects (which would require implementing a usable software tool, not just a prototype).

Another interesting topic for investigation concerns the situation where the system induces a large number of candidate formulae. In this situation the process discussed in section 5 has the user provide additional example scenarios. However, an alternative is to have the system somehow determine which scenarios would provide valuable additional information, and propose these to the user, who might then select an additional scenario from those proposed by the system, indicating whether the scenario represents desirable or undesirable behaviour.

An open question is what sorts of expectation are of use in practice. In order to answer this question we need to gain more experience in using expectations to monitor computer-mediated social interactions. Although work has been done in collecting and classifying the sorts of formulae used in software verification [5], it is not clear to what extent properties for monitoring social interactions are similar to those for specifying software systems. Nonetheless, the two examples that we have considered would fall into the taxonomy of Dwyer *et al.* [5] as Response ($b \rightarrow \Diamond p$) and Absence ($o \rightarrow \bigcirc((\neg o) \cup p)$), which together account for just under 60% of their collected specifications.

References

1. Dignum, F., van Eijk, R.M.: Special issue on agent communication. J. Automomous Agents and Multiagent Systems 14(2) 119–206 (2007)
2. Cranefield, S.: A rule language for modelling and monitoring social expectations in multi-agent systems. In: Boissier, O.; Padget, J.; Dignum, V.; Lindemann, G.; Matson, E.; Ossowski, S.; Sichman, J.; Vzquez-Salceda, J. (eds.) COIN 2005. LNCS, vol. 3913, pp. 246–258. Springer, Heidelberg (2006). Springer (2006) 246–258
3. Cranefield, S., Winikoff, M.: Verifying social expectations by model checking truncated paths. In: Workshop on Coordination, Organizations, Institutions, and Norms (COIN). (2008)
4. Barringer, H., Goldberg, A., Havelund, K., Sen, K.: Program monitoring with LTL in EAGLE. In: Workshop on Parallel and Distributed Systems: Testing and Debugging (PADTAD). (2004)
5. Dwyer, M.B., Avrunin, G.S., Corbett, J.C.: Patterns in property specifications for finite-state verification. In: International Conference on Software Engineering (ICSE), pp. 411-420. ACM Press (1999)
6. Verdicchio, M., Colombetti, M.: A logical model of social commitment for agent communication. In: Proceedings of the 2nd International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS 2003), pp. 528–535. ACM Press (2003)
7. Sierra, C., Thangarajah, J., Padgham, L., Winikoff, M.: Designing institutional multi-agent systems. In: Padgham, L., Zambonelli, F., (eds.) AOSE 2006. LNCS, vol. 4405, pp. 84–103. Springer, Heidelberg (2007)
8. Endriss, U.: Temporal logics for representing agent communication protocols. In: Dignum, F., van Eijk, R., Flores, R. (eds.) Agent Communication II. LNAI, vol. 3859, pp. 15–29. Springer, Heidelberg (2006)
9. Artikis, A., Pitt, J., Sergot, M.J.: Animated specifications of computational societies. In: Proceedings of the First International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS), pp. 1053–1061. ACM Press (2002)
10. Cypher, A., Halbert, D.C., Kurlander, D., Lieberman, H., Maulsby, D., Myers, B.A., Turransky, A., eds.: What What I Do: Programming by Demonstration. The MIT Press (1993)
11. Lieberman, H., ed.: Your Wish is My Command: Programming by Example. Morgan Kaufman (2001)
12. Dillon, L., Kutty, G., Moser, L., Melliar-Smith, P., Ramakrishna, Y.: Graphical specifications for concurrent software systems. In: 14th International Conference on Software Engineering (ICSE), pp. 214–224. ACM Press (1992)
13. Del Bimbo, A., Rella, L., Vicario, E.: Visual specification of branching time temporal logic. In: 11th International IEEE Symposium on Visual Languages (VL), pp. 61–68. IEEE Computer Society (1995)

14. Autili, M., Inverardi, P., Pelliccione, P.: Graphical scenarios for specifying temporal properties: an automated approach. Automated Software Engineering 14(3) 293–340 (2007)
15. Muggleton, S.: Inductive Logic Programming. first edn. Academic Press (1992)
16. Alfonso, A., Braberman, V., Kicillof, N., Olivero, A.: Visual timed event scenarios. In: International Conference on Software Enginering (ICSE), pp. 168–177. IEEE Computer Society (2004)
17. Harel, D.: Can programming be liberated, period? IEEE Computer 41(1), 28–37 (2008)
18. Poutakidis, D., Padgham, L., Winikoff, M.: Debugging multi-agent systems using design artifacts: The case of interaction protocols. In: Proceedings of the First International Joint Conference on Autonomous Agents and Multi Agent Systems (AAMAS), pp. 960–967. ACM Press (2002)
19. Wormuth, B., Becker, P.: Introduction to Formal Concept Analysis. http://www.wormuth.info/ICFCA04/materials.html (2004)

A Generating Formulae

The formula generation algorithm is based on the one used by the `lbtt` tool[11].

```
% gentop - generate and, or, implies at the top level
gentop(N,Xs,and(P1,P2), N3) :-
    N>0, N1 is N-1,
    gen(N1,Xs,P1,N2), gen(N2,Xs,P2,N3), \+ P1=P2.
gentop(N,Xs,or(P1,P2), N3) :-
    N>0, N1 is N-1,
    gen(N1,Xs,P1,N2), gen(N2,Xs,P2,N3),
    \+ P1=P2, \+ P1=not(_), \+ P1=notatom(_).
gentop(N,Xs,implies(P1,P2), N3) :-
    N>0, N1 is N-1,
    gen(N1,Xs,P1,N2), gen(N2,Xs,P2,N3),
    \+ P1=P2, \+ P1=not(_), \+ P1=notatom(_).
gentop(N,Xs,F,N1) :- gen(N,Xs,F,N1).

gen(N,Xs,atom(X),N) :- member(X,Xs).
gen(N,Xs,notatom(X),N) :- member(X,Xs).
gen(N,Xs,always(P),N2) :-
    N>0, N1 is N-1, gen(N1,Xs,P,N2),
    \+ P=always(_), \+ P=next(_).
gen(N,Xs,eventually(P),N2) :-
    N>0, N1 is N-1, gen(N1,Xs,P,N2),
    \+ P=eventually(_), \+ P=next(_).
gen(N,Xs,until(P1,P2),N3) :-
    N>0, N1 is N-1,
    gen(N1,Xs,P1,N2), gen(N2,Xs,P2,N3),
    \+ P1=P2, \+ P2=eventually(_).
gen(N,Xs,until(P1,and(P2,P3)),N4) :-
```

[11] http://www.tcs.hut.fi/Software/lbtt/doc/html/The-formula-generation-algorithm.html

```
    N>1, N1 is N-2,
    gen(N1,Xs,P1,N2), gen(N2,Xs,P2,N3), gen(N3,Xs,P3,N4),
    \+ P2=P3.
gen(N,Xs,until(or(P1,P2),P3),N4) :-
    N>1, N1 is N-2,
    gen(N1,Xs,P1,N2), gen(N2,Xs,P2,N3), gen(N3,Xs,P3,N4),
    \+ P1=P2.
gen(N,Xs,next(P),N2) :- N>0, N1 is N-1, gen(N1,Xs,P,N2).
gen(N,Xs,always(or(P1,P2)),N3) :-
    N>1, N1 is N-2,
    gen(N1,Xs,P1,N2), gen(N2,Xs,P2,N3), \+ P1=P2.
gen(N,Xs,eventually(and(P1,P2)),N3) :-
    N>1, N1 is N-2,
    gen(N1,Xs,P1,N2), gen(N2,Xs,P2,N3),  \+ P1=P2.

% holds(F,M) - does formula F hold in model M?
holds(atom(X),[M|_]) :- member(X,M).
holds(notatom(X),[M|_]) :- \+ member(X,M).
holds(not(X), M) :- \+ holds(X,M).
holds(always(_P), []).
holds(always(P), [M]) :- holds(P,[M]).
holds(always(P), [M,M1|Ms]) :-
    holds(P,[M,M1|Ms]), holds(always(P), [M1|Ms]).
holds(eventually(P), [M]) :- holds(P,[M]).
holds(eventually(P), [M,M1|Ms]) :-
    holds(P,[M,M1|Ms])  ; holds(eventually(P),[M1|Ms]).
holds(until(_P1,P2), [M]) :- holds(P2,[M]).
holds(until(_P1,P2), [M,M1|Ms]) :- holds(P2,[M,M1|Ms]).
holds(until(P1,P2), [M,M1|Ms]) :-
    holds(P1,[M,M1|Ms]), holds(until(P1,P2), [M1|Ms]).
holds(next(P), [_|Ms]) :- holds(P,Ms).
holds(or(P1,P2),M) :- holds(P1, M) ; holds(P2,M).
holds(and(P1,P2),M) :- holds(P1, M) , holds(P2,M).
holds(implies(P1,P2),M) :- holds(P2, M) ; (\+ holds(P1,M)).
```