A Computational Framework for Practical Social Reasoning

Citation for published version:
Wallace, I & Rovatsos, M 2015, 'A Computational Framework for Practical Social Reasoning' Computational Intelligence, vol. 31, no. 1, pp. 69-105. DOI: 10.1111/coin.v31.1

Digital Object Identifier (DOI):
10.1111/coin.v31.1

Link:
Link to publication record in Edinburgh Research Explorer

Document Version:
Peer reviewed version

Published In:
Computational Intelligence

Computational Intelligence, Volume ??, Number 000, 2012

A Computational Framework for Practical Social Reasoning

IAIN WALLACE

School of Mathematics and Computer Science, Heriot-Watt University, Scotland

MICHAEL ROVATSOS

School of Informatics, University of Edinburgh, Scotland

This paper describes a framework for practical social reasoning designed to be used for analysis, specification and implementation of the social layer of agent reasoning in multiagent systems. Our framework, called the *Expectation-Strategy-Behaviour* (ESB) framework, is based on (i) using sets of update rules for social beliefs tied to observations (so-called *expectations*), (ii) bounding the amount of reasoning to be performed over these rules by defining a reasoning *strategy*, and (iii) influencing the agent's decision making logic by means of *behaviours* conditioned on the truth status of current and future social beliefs. We introduce the foundations of ESB conceptually and present a formal framework, and an actual implementation of a reasoning engine which is specifically combined with a general (BDI-based) practical reasoning programming system. We illustrate the generality of ESB through select case studies which show that it is able to represent and implement different typical styles of social reasoning. The broad coverage of existing social reasoning methods, the modularity that derives from its declarative nature, and its focus on practical implementation make ESB a useful tool for building advanced socially reasoning agents.

Key words: Agent-Oriented Software Engineering, Multiagent Systems, Practical Reasoning Systems, Social Reasoning

Address correspondence to Michael Rovatsos, School of Informatics, The University of Edinburgh, Informatics Forum, 10 Crichton Street, Edinburgh EH8 9AB, United Kingdom, Tel.: +44-131-651-3263, Fax: +44-131-650-6899, Michael.Rovatsos@ed.ac.uk

1. INTRODUCTION

Social reasoning, which we define as reasoning about the interactions between agents in a multiagent system, is a central concern of multiagent systems research, as the nature of such systems emphasises the aspect of interaction in collectives of intelligent agents. Unsurprisingly, the range of methods applied to social reasoning is therefore very broad, and includes (Shoham and Leyton-Brown, 2009; Wooldridge, 2009):

- methods for offline design of interaction mechanisms based on a principled analysis of the interaction problem in hand (as in game-theoretic approaches, mechanism design, and agent communication languages),
- methods that restrict the range of behaviours of agents to ensure certain global system properties are maintained (as in plan coordination and teamwork-based approaches, norms and social laws), and
- methods for online reasoning about other agents while interaction is unfolding (as in adversarial search, planning and learning, opponent modelling, and trust and reputation mechanisms).

Quite naturally, this means that different social reasoning approaches focus on different techniques, e.g. logic-based specification and (manual or automated) verification of formal properties, game-theoretic analysis, adaptation of algorithms from such areas as search, planning, and machine learning to social reasoning problems, and the design of novel, “bespoke” formalisms and algorithms.

This has resulted in a great deal of fragmentation in the area, making different methods from different areas hard to compare, combine, and reuse in a context different from the original setting they were developed for. From an engineering point of view, this limits the possibilities of using existing approaches for new multiagent systems. Moreover, many of these areas do not provide a means for using their theoretical models directly in the *implementation* of agent-based systems. Instead, they restrict themselves to specifying and analysing properties of systems, and leave the translation of these properties to actual implementations to the human expert.

In this paper, we present a framework for describing social reasoning mechanisms in a uniform way which allows agents to reason about specifications of these reasoning mechanisms at run-time. This framework, called the *Expectation-Strategy-Behaviour* (ESB) framework, attempts to capture what we view as the "greatest common denominator" between many existing methods:

(i) The ability to maintain a set of evolving beliefs about properties regarding the "social state" of the system,
(ii) a facility to trigger behaviours based on the state of these beliefs and predictions about their evolution, and
(iii) provisions to limit this analysis in a rational way depending on available computational resources.

ESB presents the opportunity to analyse, formalise, specify and implement social reasoning methods on several levels. Firstly, as a conceptual tool that provides a unifying method for looking at different types of social reasoners. Secondly, as a formal specification language which allows designers to capture and analyse social reasoning mechanisms in a precise way. Thirdly, as a computational mechanism for generating behaviour according to such specifications in implemented agents in a modular, extensible, and reusable way.

This paper presents a combination and extension of different aspects of ESB that have been presented before in Wallace and Rovatsos (2009, 2011). We tie together the existing work presenting a more uniform formal presentation, expanding on the general context of the work, provide further details of the implemented reasoning system and present a more complex example on norm reasoning.

The remainder of this paper is structured as follows: We start by discussing desiderata for a general practical social reasoning system in Section 2. This is followed by an introduction of the ESB framework at a conceptual level in Section 3. We deliberately keep sections 2 and 3 informal, explaining the general ESB principles. Section 4 provides the formal definition of the ESB model. The ESB reasoning engine implemented is described in Section 5, followed by an evaluation based on several case studies in Section 6. Sections 7 and 8 close with an overview of related work and concluding remarks.

2. DESIDERATA

The purpose of ESB is to aid the implementation of *practical* social reasoning agents that can make inferences about the social state of affairs and act accordingly at run-time. In other words, we emphasise the requirement of being able to use a framework for social reasoning from the agent's point of view in actual agent programming, rather than only for the human expert to reason "externally" about specifications of social reasoning methods. We call it "practical" in this sense, as it is directed toward action (as opposed to theoretical reasoning *about* a system). To achieve this at a relatively generic level, it is necessary to provide:

- a template for social reasoning that fits the requirements of many existing social reasoning methods,
- a description of the computational mechanism that is needed to process specifications of these methods in the suggested "meta-model", and
- a means to adapt the amount of run-time social reasoning to the computational resources that are available in a concrete system.

This is precisely what we set out to achieve with the research described in this paper: To capture various different styles of social reasoning, we develop the abstract *ESB framework*, a generic meta-model/template for concrete social reasoning methods. To be a useful and flexible technique in agent development, we want to be able to program ESB agents. We develop the formal model for the ESB framework with a clear syntax and semantics, which is suitable for boundedly rational run-time processing. This allows us to build an *ESB interpreter*, a reasoning engine for processing ESB specifications. The interpreter can be integrated with a BDI interpreter to (i) show how practical social reasoning and general practical reasoning interact and can be meaningfully combined and (ii) to be able to present example implementations of existing social reasoning methods for evaluation purposes.

In the development of ESB, we follow the example of BDI in many ways. Firstly, just as the BDI architecture acts as a meta-model for *general* practical reasoning (i.e. reasoning about goals and actions), ESB can be used as a conceptual template for *social* practical reasoning. Secondly, following the example of BDI logics, the formal ESB framework allows us to construct formal specifications of concrete social reasoning mechanisms and to verify those automatically. Thirdly, the ESB reasoning engine allows these formal specifications to be used directly for agent implementation much in the same way as BDI-based programming languages provide similar functionality.

3. CONCEPTUAL FRAMEWORK

3.1. High-level overview

Before moving on to a more detailed introduction of the framework, it is worth describing the main ideas behind it along very general lines to provide some intuition and to motivate our approach.

Generally speaking, multiagent systems involve distributed representations of the global state of affairs, where the local views agents have of parts of this state are modelled as *beliefs*. The intuition underlying the design of ESB is that any "hidden" property of the system that concerns other agents and/or interactions with them that cannot be observed directly could be viewed as a *social belief.* While there is no precise definition of what a social belief is, the most common examples include the mental states of other agents (their beliefs, intentions, preferences, capabilities etc.), the execution state of a protocol-based or plan-based interaction, social laws, norms and commitments, etc.

We define *expectations* as rules that govern how social beliefs are updated under different circumstances based on different observations. *Strategies* are then "expectation expansion" functions that specify how an agent "maps out" the evolution of expectations over time to make decisions, determining how the agent's social reasoning is bounded. Finally, we introduce *behaviours* as rules that determine how social beliefs affect other beliefs or the agents' actual behaviour in the system.

To develop a practical mechanism that can be used by agents at run-time, our *computational* tools for ESB provide the following features:

- A specification language that can be used by the designer to specify the properties of a social reasoning mechanism using expectations, strategies and behaviours.
- An algorithm that automatically generates a state transition model that behaves in compliance with these properties. Adding this model to the agent's specification will ensure that the resulting agent design satisfies the properties of the social reasoning mechanism in question.
- A method for verifying the guards of behavioural rules that specify how different states of social beliefs impact on real behaviour, and to modify behaviour according to those rules that fire in a given situation.

Put simply, the work described in this paper enables agent designers to specify an update mechanism for social beliefs and rules for how different states of social belief affect behaviour. Then, the generic ESB machinery allows such specifications to be directly executed in an implementation. We introduce a simple example in the following section that illustrates this further.

3.2. A Simple Example

Consider the following very simple example of a trust mechanism in a multiagent system: Assume a reasoning agent maintains trust values for other agents and updates them based on its own interaction experience and reports from other agents. For the sake of simplicity, we assume a discrete range of trust "levels" $\{++,+,0,-,--\}$ and use operators like $<$ for comparisons between them.

The non-observable social belief here is the actual trustworthiness of another agent A which we denote by t_A, as estimated by the reasoning agent R. Here he semantics of "trustworthiness" is only important as far as it affects R's behaviour, though how it is modelled may of course be based on some underlying theory of trust.

In this trust mechanism, expectations can be used to describe update rules between different states of t_A. For example, R may want his trust mechanism to satisfy the following constraints:

$E1$ "When interacting with A for the first time, let $t_A = 0$. If the interaction has a positive outcome, increment t_A, else decrement t_A by two levels."
$E2$ "If B is a witness with $t_B > 0$ and B reports a negative experience with A, decrement t_A by one level."

Even in this simple mechanism, we can observe a few simple properties that will be discussed at a more general level later. Firstly, as the clauses suggest, social beliefs are often contingent statements that depend on the context, i.e. they rely on some *condition*. Secondly, as the belief they concern (trust) cannot be verified itself through observation, an explicit *test* has to be specified, i.e. an observable event that triggers belief change (a positive/negative interaction experience, or B's report in the above example). Finally, to capture the belief change mechanism, a *response* to different outcomes of a test has to be defined, that modifies some belief (here, changes to the value of t_A).

In isolation, these update rules, which we call expectations in our framework, are of course not very useful for practical decision making. Rather, in a sensible agent design, different states, or projections about future states of their values would trigger behaviours, for example:

$B1$ "If $t_A < 0$, don't send A any sensitive information."

$B2$ "If $t_A = --$, don't interact with A."

$B3$ "If $t_A = --$ and t_A cannot ever increase again, discard A as an interaction partner."

$B4$ "If $t_A > t_B$, prefer A over B."

$B5$ "If $t_A > t_B$ always in the future, discard B."

Some of these rules refer only to the current situation, simply conditioning certain behaviours on the belief state. More complex rules however depend on the evolution of beliefs, effectively representing *meta-reasoning* that considers the dynamics of R's own belief update mechanism. $B3$ and $B5$ illustrate this. Given that the expectations allow third party witness reports only to decrease trust values in the example, rule $B2$ effectively means that once t_A has dropped to $--$, it will not increase again, as that agent will not be interacted with. So one could apply rule $B3$ to stop considering A as an interaction partner ("discard" them) and save time. Similarly, if rule $B4$ is applied, as t_B won't increase again, rule $B5$ will also become true – B can be safely ignored.

This kind of meta-reasoning requires inference over the entire set of expectation rules, and, if temporal dynamics with cycles are present, can cause unbounded reasoning complexity. To enable rational, bounded decision making and generate responsive behaviour in real-world applications, one has to restrict the scope of possible future changes to expectations. Example strategies of how to do this might be the following:

$S1$ "Only consider trust value changes up to the next ten interactions or twenty witness reports."

$S2$ "Ignore positive interactions or positive witness reports when making predictions about future trust values."

$S3$ "Only make predictions about high-value interactions, ignoring low-value interactions."

These are just examples for how to limit the projection space, either by imposing a depth limit ($S1$), taking a pessimistic "worst-case" point of view in evaluating future developments ($S2$), or pruning the search space based on domain-dependent criteria ($S3$). Effectively, they restrict the "horizon" over which temporal operators like "never again" or "always in the future" will be evaluated, thus bounding the set of future states that have to be considered in the guards of behaviour rules like $B3$ and $B5$ above. Depending on the time and space available to an agent when making decisions, different levels of reasoning accuracy are possible, and they reflect the amount of effort that goes into social reasoning activities in actual agent implementations.

At an informal level, this example suggests that ESB is appropriate for capturing trust and reputation mechanisms. In Section 6, we will show that ESB achieves broad coverage of existing social reasoning methods by presenting concrete case studies where we apply it to reasoning about intentions in communication, reasoning about norms, and opponent modelling in games.

3.3. Expectations, strategies, and behaviours

We now present a mostly informal overview of the ESB framework, which already uses some of the notation which will be formally introduced in Section 4 in full detail. To begin with, we introduce our notion of *expectation*: An *expectation*

$$(\mathbf{Exp}\, N\; A\; C\; \Phi\; T\; \rho^{+}\; \rho^{-})$$

is a rule specifying the way in which A will update a belief regarding statement Φ. The rule has identifier N, and specifies that Φ would be believed only under condition C. It also specifies how the expectation set might change after performing *test* T, by specifying a *response* ρ^{+} if the test is successful (i.e. confirms the expectation), and an alternative response ρ^{-} if the test fails.

Going back to the examples on p.7, we could encode $E1$ as

$$\Big(\mathbf{Exp}\; \underbrace{E_1}_{N}\; \underbrace{R}_{A}\; \underbrace{\mathit{first\_interaction}(A)}_{C}\; \underbrace{\mathit{trust}(A,0)}_{\Phi}\; \underbrace{\text{“positive interaction”}}_{T}$$
$$\underbrace{\text{“replace E1 with E3”}}_{\rho^{+}}\; \underbrace{\text{“replace E1 with E4”}}_{\rho^{-}}\Big)$$

where the responses ρ replace the expectation with $E3$ and $E4$, which are similar expectations with the appropriately adjusted trust values (one level up and two levels lower, respectively). An expectation for $E2$ would be conditioned on another agent

B being trusted, expect that B would make for an appropriate witness, and the tests would involve B informing R of the untrustworthiness of A. The positive response would be to replace $E1$ with an expectation corresponding to the respective lower trust level. The negative response is void in this example, so nothing would be required here. We deliberately only present one illustrative expectation here, rather than the complete system, as this is a simple example to introduce the ideas. Further details of expectations systems can be found throughout later sections of the paper.

Expectations allow us to express context-dependent rules for how to update social belief, and under which circumstances the update will have what consequences. Thereby, we assume that the test T is something that can be observed in the environment, and the response ρ is a modification to the set of all expectations held by the agent.

We should emphasise that introducing the notion of "expected belief" contained in an expectation as something *different* from beliefs (in the ordinary sense of the word) is important for several reasons. Methodologically, it allows us to single out "social" beliefs from other beliefs used by the general (practical) reasoning mechanism of the agent, so that any influence on non-social beliefs has to be made explicit via some behaviour rule (as we shall see below). Also, it encourages the designer to think about the context, triggering test and update rules associated with an expected belief when mentioning it in an expectation. Computationally, as we describe below, labelling beliefs as "expectation-related" serves as a mechanism for including them in the model generation phase of the ESB reasoning engine.

3.4. Reasoning about expectations

The overall ESB reasoning process is relatively easy to describe at a high level, since an agent's social reasoning state is simply defined by the agent's active set of expectations (defined formally below). Each processing cycle consists of two phases:

Expectation update Based on observed events and/or belief changes outside the ESB reasoning mechanism (e.g. at BDI level), expectations in the active set are made current (or not, according to their conditions), which leads to a new set of expected beliefs – those directly used for behaviour generation.

Behaviour generation The guards of behaviour rules are verified against the model of dynamic expectation changes derived from current expectations. Those behaviour rules that "fire" effect a change to the agent's non-ESB reasoner. Usually this will be a belief change, but in theory it could directly lead to physical action, the adoption of some intention, etc.

For update, the agent looks at the *active* expectations that are currently present. For those active expectations whose conditions C apply (called *current* expectations), the presence of test condition (or event) T will trigger the response ρ^+, if $\neg T$ is believed then the negative response ρ^- is triggered.

As far as behaviour generation is concerned, it is easiest to imagine the reasoning model as a graph among possible sets of expectations whose edges represent transitions caused by expectation updates (similar to state-transition models used in temporal/dynamic logics). Behaviour rules of the format "`if` ϕ `then` ψ" will cause the ESB engine to verify whether property ϕ holds in this state-transition model. If the verification of ϕ succeeds, the ESB engine will attempt to make ψ true, e.g. by revising the agent's (non-ESB) belief. A behaviour rule like

$B5$ "If $t_A > t_B$ always in the future, discard B",

for example, will traverse the expectation graph checking whether t_B can ever exceed t_A in all future states. If this check fails, it will remove a fact that B is a potential interaction partner from the agent's belief base. If a strategy like

$S2$ "Ignore positive interactions or positive witness reports when making predictions about future trust values."

is used during this verification of $t_A > t_B$, only edges in the graph would be followed that are caused by negative witness reports or negative interactions, i.e. the agent would only take negative information into account.

It is important to highlight here that this reasoning cycle does not describe any particular "style" of social reasoning, and this is intentional. By sticking to a general structure which translates any set of rules that could describe an update mechanism for social beliefs into a model within which arbitrary behaviour-triggering properties can be proven, ESB provides a framework for social reasoning that is as generic as possible, within the limitations of a propositional formalism. Moreover, the notion of strategies allows any potential limitation on reasoning resources to be taken into account in a concrete ESB implementation.

4. FORMAL FRAMEWORK

4.1. Expectations

To define the syntax of expectations, assume a set of *agent names* $\mathscr{A} = \{A, A', \ldots\}$, a set of *expectation names* $\mathscr{N} = \{N, N', \ldots\}$, and a propositional logical language $\mathscr{L}$ (with atoms $\{p, q, \ldots\}$, constants $\top$ and $\bot$, and the usual connectives). An *expectation* is a structure

$$(\mathbf{Exp}\; N\; A\; C\; \Phi\; T\; \rho^{+}\; \rho^{-})$$

where $N \in \mathscr{N}$, $A \in \mathscr{A}$, $C \in \mathscr{L}$, $\phi \in \mathscr{L}$, $T \in \mathscr{L}$ and $\rho^{+}, \rho^{-} \in \wp(\mathscr{N}) \times \wp(\mathscr{N})$. The intended meaning of such an expectation held by agent A is as follows:

"If N is an active expectation, whenever condition C holds, adopt belief Φ. When verification of T occurs, respond according to ρ^{+} if T is true, and according to ρ^{-} otherwise."

Essentially, this means that expectations are event-condition-action rules, with the only difference that the actions taken refer to the rulebase itself: although the agent could "respond" in many different ways in theory, in ESB this is restricted to activating and deactivating other (or the same) expectation(s). To express this, we define $\rho = (A, D)$ for both responses $\rho \in \{\rho^{+}, \rho^{-}\}$, such that $add(\rho) = A/del(\rho) = D$ are the sets of expectations that are added to/removed from the set of active expectations after update.

To keep the framework for reasoning about expectations simple, we deliberately do not introduce a novel logic of expectations. Instead, we use expectations as atomic rules that affect non-social beliefs, and resort to a BDI logic for capturing the semantics of expectations. Specifically, we use $\mathscr{LORA}$ (Wooldridge, 2000), a multi-modal propositional BDI logic which combines modalities for belief, desire and intention with aspects of dynamic logic and CTL temporal logic (note that in the concrete implementation presented below, we introduce additional "syntactic sugar" to allow first-order-like specifications of finite sets of expectations using variables, see Section 5.2). Additionally, to represent dynamically changing expectation sets, we introduce special propositions $\{\alpha_E | E \subseteq_{fin} \mathscr{N}\}$ that are distinct from all other atoms in $\mathscr{L}$ and are used to *reify* what subset of all possible expectations is active. We take these to be finite subsets of $\mathscr{N}$ and to be mutually exclusive, such that α_E is only true for the maximal (with respect to set inclusion) set of active expectations at any given point in time. This allows us to introduce expectation syntax simply as shorthand notation for $\mathscr{LORA}$ formulas in the following way:

Assume agent A is reasoning about a set of (potential) expectations $\mathbf{EXP} = \{e_1, \ldots, e_m\}$, where $e_i = (\mathbf{Exp}\ N_i\ A\ C_i\ \Phi_i\ T_i\ \rho_i^+\ \rho_i^-)\ 1 \leqslant i \leqslant m$. The semantics of this set $\mathbf{EXP}$ is defined by rewriting it as a set of $\mathscr{LORA}$ statements:

$$\alpha_E \Leftrightarrow \bigwedge_{i \in \{1,\ldots,m\}} \mathbf{A}\Box(current_i\ \mathscr{W}\ update_i)$$

where

$$current = (\mathbf{Bel}\ A\ C_i) \Rightarrow (\mathbf{Bel}\ A\ \Phi_i)$$

$$update = \Big((\mathbf{Bel}\ A\ T_i) \wedge \mathbf{A} \bigcirc \alpha_{EXP \setminus del(\rho_i^+) \cup add(\rho_i^+)}\Big) \vee \Big((\mathbf{Bel}\ A\ \neg T_i) \wedge \mathbf{A} \bigcirc \alpha_{EXP \setminus del(\rho_i^-) \cup add(\rho_i^-)}\Big)$$

This reformulation of expectation sets in $\mathscr{LORA}$ requires some explanation. To start with, we should describe some of the more complex $\mathscr{LORA}$ notation used: $\bigcirc\ \varphi$ ($\Box\ \varphi$) means φ will be true in the next state (in all future states) along a temporal path; $\varphi\ \mathscr{W}\ \psi$ is a weak "until" operator, denoting that until ψ becomes true (which might never happen), φ will hold true; $\mathbf{A}\ \varphi$ is universal path quantification used for the branching-time side of $\mathscr{LORA}$, denoting that φ will be true on all paths from the current state; finally, $(\mathbf{Bel}\, A\ \varphi)$ is a normal KD45 belief modality for agent A.

Given this, each of the rules defines when exactly each α_E holds true, i.e. when the agent has set E as its current set of active expectations. This is defined as a conjunction over all $e \in E$, where the formula for each e corresponds to

"belief in the condition $C(e)$ will lead to adopting the expected belief $\Phi(e)$ ("*current*") until ("*update*") the agent either comes to believe $T(e)$ and modifies e according to ρ^+, disbelieves $C(e)$, or comes to believe $\neg T(e)$ and modifies E according to ρ^-."

We use the notation $X(e)$ to refer to component X of expectation e in a set of expectations (e.g. $C(e)$ refers to the condition of e, etc).

What is referred to as "modification" of the set of active expectations here is what necessitates introduction of the α_E propositions, as we have to be able to express a dynamic relationship between different such sets. The $\mathscr{LORA}$-level definition of expectation sets effectively means that these create a number of logical constraints on the belief dynamics of the agent at the meta-level, using BDI-level belief as an "object language". This has several implications: On the positive side, it allows us to seamlessly integrate ESB specifications with the BDI-level reasoner, and to avoid introduction of yet another logical language and proof theory. On the negative side, we cannot reason compositionally about the contents of $C(e)$, $\Phi(e)$ and $T(e)$ across different expectations, as capturing the dynamics of how sets of expectations are updated requires the reification performed above, at least to be able to do efficient, resource-bounded reasoning over them. Below, we will return to the issue of exponential blowup in specification length that results from the power set construction over expectations above.

To summarise, this defines a relatively straightforward (though restricted) way for managing expectations. In what follows, for convenience we will often write EXP_A for the set of *active* expectations, i.e. the subset of expectations for which α_E is true, and EXP_C for *current* expectations $e \in EXP_A$ for which $C(e)$ is true i.e. active expectations for which the agent will adopt the expected belief $\Phi(e)$ (obviously, $EXP_C \subseteq EXP_A \subseteq \mathbf{EXP}$). Note that, in our examples below, the annotation of expectations with the identifier of the agent who holds them is not needed. However we maintain it in the expectation notation throughout the paper as it is necessary to maintain the mapping to the $\mathscr{LORA}$ semantics, which require beliefs to be ascribed to the agent that holds the respective expectation.

4.2. Strategies

As explained above informally, strategies are used to bound reasoning over expectations by constraining the way in which future expectation updates are considered. To define them formally, it is easiest to think of the different potential values of EXP_A as nodes in an *expectation graph*, and of the connections between different such sets that are triggered by the expectations contained in EXP_A as edges among them. Then, a strategy simply becomes a restriction on the paths considered in this graph when verifying a given behaviour.

More formally, let $G = (\mathscr{V}, \mathscr{E})$ an *expectation graph*, where $\mathscr{V} = \wp(\mathbf{EXP})$. To define the set of edges $\mathscr{E} \subseteq \mathscr{V} \times \mathscr{V}$, we first introduce the notation

$$\begin{aligned} \mathit{update}(E,E^+,E^-) := E \cup \big(\bigcup_{e \in E^+} \mathit{add}(\rho^+(e))\big) \setminus \big(\bigcup_{e \in E^+} \mathit{del}(\rho^+(e))\big) \\ \cup \big(\bigcup_{e \in E^-} \mathit{add}(\rho^-(e))\big) \setminus \big(\bigcup_{e \in E^-} \mathit{del}(\rho^-(e))\big) \end{aligned}$$

where

$$\begin{aligned} E^+ &= \{e | e \in EXP_A, (\mathbf{Bel}\, A\ T(e))\} \\ E^- &= \{e | e \in EXP_A, (\mathbf{Bel}\, A\ \neg T(e))\} \end{aligned}$$

This allows us to determine the set of active expectations $\mathit{update}(E,E^+,E^-)$ that results from applications of the responses of an arbitrary combination of updates following from positive test outcomes for some expectations E^+ and negative outcomes for the tests of expectations E^-. That is, $\mathit{update}(E,E^+,E^-)$ is one potential successor state of E brought about by one particular combination of test outcomes. With this, we can define the edges of the expectation graph as

$$\mathscr{E} = \Big\{ (E,E') \mid E' = \mathit{update}(E,E^+,E^-), E^+ \cup E^- \subseteq E, E^+ \cap E^- = \emptyset \Big\}$$

This means that all pairs of active expectation sets (E,E') will be connected if E' is a potential successor state of E for some active expectation tests succeeding and others failing. Note that this does not require a complete specification of all test outcomes in E; due to the condition $E^+ \cup E^- \subseteq E$ all transitions are included for any subset of tests $T(e)$ for $e \in E$. This enables us to track transitions when only some outcomes of $T(e)$ become known.

With this structure in mind, a strategy is a way to restrict this graph in some way, either according to some particular style of reasoning (e.g. “optimistic”, i.e. only considering positive test outcomes), or so as to respect limitations regarding reasoning resources (e.g. restricted to some depth limit regarding future expectation states). This is useful as behaviours rely on checking conditions on the accessible portion of the graph as we shall see below.

A *strategy* **STR** $\subseteq \mathscr{E}^*$ is therefore formally just a subset of the paths that can be generated from $\mathscr{E}$. How the set of paths considered following a particular strategy is restricted is deliberately not specified. This is a concern for specific ESB implementations, but could be in terms of forbidden states, sequences of states, transitions etc. In principle any graph operation could be used that restricts the set of paths considered.

It should be remarked that this graph-theoretic description of the expectation update system is closely related to the constructions of the previous section, in particular to the reification of expectations as atomic objects which allows us to reason about their dynamics without being concerned with their content. As shown in the following section, behaviours bring the two layers together again and explain how meta-level reasoning about expectations affects actual agent belief.

4.3. Behaviours

Conceptually, behaviours in ESB are rules that express how the agent's overall reasoning and actions are affected by its social reasoning. These rules depend on the system of expectations an agent holds, which reflect the agent's reasoning dynamics about social "facts". In our formal framework, more specifically, behaviours are condition-action rules mapping conditions on the expectations to actions external to the social reasoner, where actions take the form of adding or removing beliefs to influence (say) a BDI practical reasoner (e.g. by restricting the set of plans from the plan library that can currently be used).

The set of expectations does not only describe the current social reasoning state, as represented by active expectations, but also possible future belief states. These belief dynamics are encoded in the expectation graph structure and behaviours may include conditions such as "Φ will always be expected" and "it is possible to expect Φ" in the future.

To enable this, we assume that the preconditions of behaviour rules φ are expressed in CTL (Gabbay et al., 2000) using the usual syntax

$$\begin{aligned} \varphi ::= & \top \mid \bot \mid p \mid \neg\varphi \mid \varphi \wedge \varphi \mid \varphi \vee \varphi \mid \varphi \rightarrow \varphi \mid \\ & \mathbf{A} \bigcirc \varphi \mid \mathbf{E} \bigcirc \varphi \mid \mathbf{A} \Box \varphi \mid \mathbf{E} \Box \varphi \mid \mathbf{A}\, [\varphi\, \mathscr{U}\, \varphi] \mid \mathbf{E}\, [\varphi\, \mathscr{U}\, \varphi] \end{aligned}$$

where p is an atom in $\mathscr{L}$, $\mathbf{E}\varphi$ denotes existential path quantification, and $\varphi\, \mathscr{U}\, \psi$ is a "strong" until operator (which requires that ψ eventually become true). We write $\varphi \in \mathscr{L}_{CTL}$ for formulae that are in this superset of $\mathscr{L}$ (and subset of $\mathscr{LORA}$).

As regards postconditions, as stated above, at the conceptual level it would be possible for ESB to impact the agent's behaviour in different ways depending on which behaviours are currently satisfied. To keep things simple, however, we choose to restrict the effects of ESB behaviours to belief revision operations, as this provides a clean interface to other (non-social) reasoning and does not interfere with the procedural nature of such activities as deliberation and means-ends reasoning. Thus, we only consider

$$\mathit{add\_belief}(\varphi) \text{ and } \mathit{del\_belief}(\varphi)$$

as possible *actions* of a behaviour rule, where $\varphi \in \mathscr{L}$. Note that while behaviour conditions allow temporal expressions over future expectation states to be taken into consideration, actions only affect the current beliefs of the agent, i.e. any impact of ESB reasoning on the agent's overall state is instantaneous. Also, we are not concerned with providing a semantics for these belief revision operations at the non-ESB level, but simply assume that any sufficiently expressive reasoning framework will have basic facilities for adding and removing beliefs whose semantics will be well-defined (however, in a $\mathscr{LORA}$ context, what we specify does satisfy the normal constraints on agent beliefs postulated by the logic).

With this, we can define the behaviours of an ESB reasoner as a set

$$\mathbf{BEH} \subseteq \Big\{(\phi, a) | \phi \in \mathscr{L}_{CTL}, a \in \{add\_belief(\psi), del\_belief(\chi)\}, \psi, \chi \in \mathscr{L}\Big\}$$

of such condition-action pairs (ϕ, a), and write $\phi \ \triangleright \ a$ when referring to such rules to distinguish them from other types of rules.

The semantics of such behaviours, given an expectation set **EXP** and a strategy **STR** is defined as follows, given a set of active expectations EXP_A. If $G = (\mathscr{V}, \mathscr{E})$ is the expectation graph obtained from the set of all expectations **EXP** available in principle, we define a slightly modified semantics for CTL entailment with respect to $\mathscr{EXP} = \wp(\mathbf{EXP})$ as state set and $\mathscr{E}$ as described by the expectation graph as transition function (i.e. for all $E, E' \in \wp(\mathbf{EXP})$ we write $E \rightarrow E'$ iff $(E, E') \in \mathscr{E}$). Defining $\mathscr{M} = (\mathscr{EXP}, \rightarrow)$ as a model in the CTL sense, the modification that has to be performed is with regard to any rule that involves path operators or temporal operators, e.g.

$$\mathscr{M}, E \models \mathbf{A}\Box\phi \text{ iff } (\forall E \rightarrow E_1 \rightarrow \cdots \rightarrow E_n \in \mathbf{STR}\ \forall i)\mathscr{M}, E_i \models \phi$$

where the "$\in$ **STR**" restricts the set of paths to those specified by the strategy and is the only difference between this entailment rule and the standard corresponding CTL rule. For the base case, we define:

$$\mathscr{M}, E_i \models \phi \text{ iff } \phi = \Phi(e) \text{ for some } e \in E_i, \phi \in \mathscr{L}$$

i.e. a formula $\phi \in \mathscr{L}$ is entailed in a particular expectation state E_i if it is equivalent to one of the expected beliefs of that set. Adapting the other semantic rules for modal operators is done by performing the same modification to each of the standard CTL semantic rules that involves a statement about paths on the right-hand side.

Finally, we say that a behaviour $\phi \ \triangleright \ a$ is *satisfied*, if its precondition is entailed by the current expectation state in the model $\mathscr{M}$, i.e. iff

$$\mathscr{M}, EXP_C \models \phi$$

where, as before, EXP_C are those elements $e \in EXP_A$ whose condition $C(e)$ is satisfied. Note that this implies a difference between how the *current* expectation state is treated, as opposed to future potential expectation states: Since the applicability of conditions $C(e')$ for future states cannot be anticipated in the present state (they don't depend on expectation updates defined in **EXP** but on ESB-external events), the validity of expectations in a (potentially temporal) formula ϕ must be interpreted as "contingent" on the conditions. For example, if $\mathscr{M}, \{e, e'\} \models \mathbf{E} \bigcirc \Phi(e'')$ holds, this would mean that, given that $\{e, e'\}$ are the current expectations whose conditions $C(e)$ and $C(e')$ are satisfied, then there exists at least one immediate successor (expectation) state, in which e'' would be *active*. Whether or not e'' would then become a *current* expectation, however, would depend on the value of $C(e'')$ at the time.

As stated above, the connection to the non-ESB part of the reasoner is made through fairly high-level constraints on the beliefs of the agent: Let

$$\mathbf{BEH}_C = \{(\phi_1, a_1), \dots, (\phi_n, a_n)\}$$

be the *current* behaviours, i.e. all behaviours satisfied in the above sense, then the belief base of the agent (at the non-ESB level) will satisfy

$$(\mathbf{Bel}\, A\ \psi_1 \wedge \cdots \wedge \psi_n)$$

and

$$\neg(\mathbf{Bel}\, A\ \chi_1 \vee \cdots \vee \chi_n)$$

where $a_i = (add\_belief(\psi_i), del\_belief(\chi_i))$.

4.4. Putting it all together

With these definitions, we can define an *ESB reasoner* as a tuple

$$R = \langle \mathbf{EXP}, \mathbf{STR}, \mathbf{BEH}, B, E_0 \rangle$$

where given some set of current beliefs $B \subseteq \mathscr{L}$ (which we assume to be deductively closed) and a *set of initial expectations* $E_0 \subseteq \mathbf{EXP}$ we can view the operations defined above as the specification of a belief revision function

$$(B, EXP_A) \xrightarrow{R} (B', EXP'_A)$$

that is based on conducting the following steps to determine the current expectation given beliefs B and active expectations EXP_A:

(1) Calculate the set of current expectations based on existing beliefs:

$$EXP_C = \{e \in EXP_A \mid B \models_{\mathscr{L}} C(e)\}$$

(2) Determine which behaviours are satisfied given current expectations:

$$\mathbf{BEH}_C = \{(\phi, a) \mid \mathscr{M}, EXP_C \models_{\mathbf{STR}} \phi\}$$

(3) Compute new beliefs based on current beliefs and satisfied behaviours:

$$B' = \{\varphi \mid (\phi, a) \in \mathbf{BEH}_C, a = (\mathit{add\_belief}(\psi), \mathit{del\_belief}(\chi)),$$
$$B \backslash \{\chi\} \cup \{\psi\} \models_{\mathscr{L}} \varphi\}$$

(4) Update set of active expectations:

$$EXP'_A = \mathit{update}(EXP_A, E^+, E^-) \text{ where}$$
$$E^+ = \{e \in EXP_A | B' \models_{\mathscr{L}} T(e)\} \text{ and}$$
$$E^- = \{e \in EXP_A | B' \not\models_{\mathscr{L}} T(e)\}$$

For clarity, in this description we use subscripts $\models_{\mathscr{L}}$ and $\models_{\mathbf{STR}}$ to distinguish between propositional entailment in the language of ordinary beliefs $\mathscr{L}$ and CTL entailment subject to the model restriction induced by **STR**, respectively. Steps (1) and (2) result from checking expectation conditions and behaviour conditions. In step 3., we re-build the ($\mathscr{L}-$)deductive closure of the (non-ESB) belief base B based on executing belief additions/deletions caused by those behaviours that fire. Finally, in step 4., we use the *update* function as defined when we introduced the semantics of expectations and calculate the sets of active expectations E^+/E^- whose tests have succeeded/failed according to the contents of B.

At this level of specification, the above gives an abstract account of the overall ESB reasoning cycle, which leaves a few details underspecified. Firstly, it is clear that in any real implementation of ESB, the procedure for updating beliefs will not be implemented as described in step 3. In most knowledge-based agent architectures, deductive inferences are made on demand, and no explicit update of a deductively closed set of formulae is ever performed. Any concrete implementation of ESB (such as our ESB-RS system presented in the following section) will have to provide specific implementation-level methods for dealing with these issues. Also, the definition does not deal with the problem of potentially creating a conflicting knowledge base which is a problem inherent to any design of any belief revision mechanism and ESB does not claim to preempt.

Secondly, this model of ESB reasoning lacks a notion of time, in particular with regard to the ways in which test outcomes are determined in practice. If we followed step 4. exactly as described, an arbitrary number of consecutive ESB updates could be performed without waiting for any opportunity for real belief updates from the non-ESB part of the agent. In practice, implementations should perform a single such step, and then hand control over to non-ESB reasoning components of the agent until one or more new test outcomes become available, and thereby effectively *suspend* the verification of $B \models_{\mathscr{L}} T(e)$ statements for some period of time between each ESB update. In the next section, we present a concrete implementation of a system that realises exactly this functionality, the ESB reasoning system ESB-RS.

5. AN ESB REASONING ENGINE

Previously, we have claimed that the main benefit of ESB lies in its ability to capture social reasoning mechanisms in such a way that they can be directly used by agents for reasoning at run-time. To show how expectations, strategies and behaviour rules can be used directly as executable specifications in a concrete agent design, we now present the ESB Reasoning System ESB-RS.

ESB-RS is a fully implemented prototypical ESB interpreter based on algorithms for model generation (to construct expectation graphs) and the NuSMV (Cimatti et al., 1999) model checking system (to check the preconditions of behaviours). As ESB is not a standalone agent architecture, ESB-RS is designed for integration with a specific BDI reasoning engine, in our case Jason (Bordini et al., 2007). Figure 1 illustrates the overall setup of an integrated ESB-RS+Jason agent, highlighting the interactions between the ESB and BDI components of the system. These occur via the BDI-level belief revision function (BRF), since expectations are updated based on the agent's perceptions and beliefs, and, conversely, the control of BDI-level behaviour through ESB is exerted through beliefs which act as guards on plans to carry out actions.

[Figure 1 about here.]

At a high level, ESB-RS execution can be described as follows:

Initialisation The expectation graph is generated from the expectation and strategy specifications. This is only done once, as after this initial step the agent can simply track the current expectation state in the graph during execution.

Reasoning cycle In each reasoning step, the beliefs from the BDI component, the expectation graph and the strategy specification are used to create the reduced strategy graph (i.e. the sub-graph accessible from the current state). The strategy graph is then used by the behaviour condition checker to select the applicable behaviour actions, which are used to update the agent's BDI-level beliefs accordingly.

In the following sections we describe ESB-RS at a more implementation-specific level, detailing how the general principles of the formal model translate into a concrete computational design.

5.1. Model generation

We have already described above how an expectation graph is built and how it can be restricted based on a strategy specification. To be able to make use of modern model checking technology, we need to map these operations to a concrete "model generation" procedure. This procedure builds a finite-state machine (FSM) representation of the expectation graph, which will then be used as a model (in the model-checking sense) to verify behaviour's preconditions. Importantly, we want to construct the FSM model in the simplest possible way so as to obtain an executable process model that satisfies the constraints set out by the ESB specification on which it is based. With this regard, since ESB operates at the meta-reasoning level, it is sufficient to encode the different "fields" (condition, test, responses) of an expectation as variables controlled by state transitions in the FSM. More specifically, for every expectation:

$e.\mathtt{Condition} \in \{\mathtt{True}, \mathtt{False}, \mathtt{DC}\}$: If $e \notin EXP_A$, the value of its condition is DC ("don't care"). Else, $e \in EXP_C$ iff $e.\mathtt{Condition} = \mathtt{True}$.

$e.\mathtt{Phi} \in \{\mathtt{True}, \mathtt{False}\}$: If $e.\mathtt{Condition} \neq \mathtt{DC}$ we have $e.\mathtt{Phi} = e.\mathtt{Condition}$, else $e.\mathtt{Phi} = \mathtt{False}$ ($\Phi(e)$ is not believed for non-active expectations).

$e.\mathtt{Test} \in \{\mathtt{Tp}, \mathtt{Tn}, \mathtt{NA}\}$ – The outcome of a test is either positive (Tp) or negative (Tn) or "not applicable" (NA). It is not-applicable whenever $e.\mathtt{Condition} = \mathtt{DC}$.

$e.\mathtt{Tp.add}$, $e.\mathtt{Tp.del}$ and $e.\mathtt{Tn.add}$, $e.\mathtt{Tn.del}$: The add- and delete-sets for each response of expectations are used to define the transition relation of the FSM, and contain names of other expectation objects.

The full model generation algorithm, originally described in (Wallace and Rovatsos, 2011), basically consists of pairwise checking of all expectations to determine whether they are connected in the expectation graph G and in principle yields an FSM structure that is isomorphic to G. However, the use of the model-checking level of modelling here brings major advantages: Firstly, we don't have to consider all subsets of expectations and explicitly construct the powerset of **EXP**. It is sufficient to capture the dynamics of how the state of an expectation could be altered by any other expectation (e.g. "if $e \in e'.\texttt{Tn.del}$ and $e'.\texttt{Test} = \texttt{Tn}$ then $e.\texttt{Condition} = \texttt{DC}$") and how different elements of an expectation affect each other (e.g. "if $e.\texttt{Condition} \neq \texttt{DC}$ then $e.\texttt{Test} \in \{\texttt{Tp,Tn}\}$"). These impose constraints on possible system states (like $e.\texttt{Test} \in \{\texttt{Tp,Tn}\}$) which will automatically be converted into multiple possible states whenever necessary by the model checker. Secondly, given an initial state, the model checking system will automatically only generate *accessible* portions of the expectation graph, so that the designer does not have to care about mutually exclusive combinations of test outcomes or contradictory responses of different expectations in the current expectation state.

5.2. ESB specifications in ESB-RS

To alleviate some of the practical issues surrounding writing purely propositional expectation rules (such as potentially large sets of rules) ESB-RS provides some additional syntactic sugar. By supporting variables in the various terms it allows for writing more general expectations "templates", covering a wide set of more specific circumstances as instances of these templates. While not a full first-order extension, this still proves effective in greatly speeding up the task of writing sets of expectations.

For implementation, supporting this is simple: it is done by grounding all possible variables when the condition C is checked and applying the resulting matches to the same variables when Φ is added to the belief base, or the tests carried out. As variables effectively represent shorthand for sets of expectations, there may be multiple instantiations. If there is more than one valid variable binding for C, then Φ is added to the belief base once for each binding. The tests are also considered for each binding, exactly as if there were instead a set of ground expectations.

As an example, consider an agent that has an active expectation

$$(\mathbf{Exp}\, N\, A\; trust(X)\; tradeWith(X)\; T\; \rho^{+}\; \rho^{-})$$

with beliefs $(\mathbf{Bel}\, A\; trust(alice)), (\mathbf{Bel}\, A\; trust(bob))$, the rule above means the expectations include:

$$(\mathbf{Exp}\, N\, A\; trust(alice)\; tradeWith(alice)\; T\; \rho^{+}\; \rho^{-})$$

$$(\mathbf{Exp}\, N\, A\; trust(bob)\; tradeWith(bob)\; T\; \rho^{+}\; \rho^{-})$$

There is one instance of the expectation for each value of X.

5.3. Strategies

Strategies are implemented as a set of constraints on the transition relation, which can be conditioned on either the edges (the tests that trigger transitions) or the states (the expectations that define these). Each constraint is a Boolean next-expression in NuSMV syntax which must be satisfied in all states in the strategy graph, so that the expectation graph is reduced to those states satisfying these constraints. For constraints, we use the following syntax:

$$
\begin{array}{rcl}
\texttt{Constraint} & ::= & \texttt{expr} \rightarrow \texttt{next(expr)} \mid \texttt{next(expr)} \\
\texttt{expr} & ::= & \texttt{c = a} \mid \texttt{c != a} \mid \texttt{expr|expr} \mid \texttt{expr \& expr} \mid \texttt{!expr} \\
\texttt{c} & ::= & \texttt{name} \mid \texttt{condition} \mid \texttt{phi} \mid \texttt{test} \mid \texttt{Tp.add} \mid \texttt{Tp.del} \mid \texttt{Tn.add} \mid \texttt{Tn.del} \\
\texttt{a} & ::= & \texttt{expectation} \mid \texttt{True} \mid \texttt{False} \mid \texttt{DC} \mid \texttt{Tp} \mid \texttt{Tn} \mid \texttt{NA}
\end{array}
$$

where

`next(expr)` refers to the value of `expr` in the next state.

`expectation` refers to the name given to an expectation tuple.

This means the transitions from a given state are constrained to those where the *next* state meets all the strategy constraints (which may not just be constraints on the current state, but also on the relation between the current and *next* state). The atoms in the constraints are the conditions and tests of each expectation. These allow restrictions on the states and transitions respectively, and can be combined using the standard propositional logical operators.

For implementation, the specification of the FSM used for model checking behaviour conditions is split into a global *main* module and one *sub-module* per expectation. The main module controls the update for each sub-module and handles carrying out the tests and thus effectively controls the changing sets of active expectations. For example, a main-module constraint could be written as

```
next(Test != Tn)
```

to remove negative response transitions from the strategy graph, by requiring that the `Test` of no successor state can return a negative test result. Note that for ease of implementation it is possible to specify that all tests must follow a certain form, or to refer to specific expectations.

An example expectation sub-module constraint to ignore a particular expectation (i.e. force the state to "don't care") could look as follows:

```
(name = Exp3) -> next(ExpC = DC)
```

This example states that if the `name` of an expectation is `Exp3` then in successor states the condition must be "don't care" – the expectation is not active. This is only one possible way the transition relation in the state transition model the model checker uses could be specified by the strategies and was chosen for ease of implementation and to be minimally constraining.

The difference between main-module-level and sub-module specifications constraints used to specify a strategy is that the former concern test results (and thus responses), whereas the latter modify the state of the condition of an expectation.

5.4. Behaviours

The language of behaviour conditions imposes limits on the power of an ESB agent. The more complex the conditions that can be checked, the more complex the forms of reasoning that can be captured. With this in mind, we aim to provide the most general language for conditions within the constraints of the model checking infrastructure used, to ensure fast, tractable reasoning while achieving maximum expressiveness. Recalling that a behaviour is specified in terms of a condition and an action, in ESB-RS, the action is simply a set of beliefs to add (or remove if present) to the agent's belief base. This can then be used however the designer sees fit by practical reasoner plans. For example, it could act as the guard on a plan to request a joint action (this is exactly the case in one of the case studies presented in the next section).

For implementation purposes the conditions are split into two parts: The CTL component, which is a NuSMV syntax CTL condition to check, and a Jason condition, which is a Jason-syntax expression evaluated on the agent's belief base. This allows for behaviours to account for future reasoning (through the CTL condition) and the current state of the agent's belief base. So in the abstract case, for implementation a behaviour is split up as:

$$\texttt{belief formula} \wedge \texttt{CTL formula} \rhd \texttt{belief}$$

This addition of a condition on the *belief base* to the condition on the *expectations* previously described serves principally to ease implementation and efficiency: Instead of adding an action belief and then checking it in the practical reasoner, the condition can include this check within it so that this (cheap) check can be done before a (more expensive) model checking operation is performed. This does not modify the general ESB functionality, but allows for more efficient runtime behaviour.

6. CASE STUDIES

To evaluate the practical usefulness of the ESB framework, we have conducted several case studies which deliberately focus on very different types of social reasoning: joint intentions in communication, reasoning about norms, and opponent modelling. On the one hand, these methods are representative of classic coordination mechanisms in multiagent systems research. On the other hand, they emphasise different aspects, such as hard-coded, simple and relatively static social conventions (joint intentions), belief revision and choice over possible social behaviours (norm-autonomous agents), and reasoning about others' mental states in strategic or adversarial domains (opponent modelling). Also, indirectly, our abstract trust and reputation example above illustrates the application of ESB to a fourth category of social reasoning, namely methods based on long-term observation of behaviour and testimonies received from third parties (though this has not been taken to the level of full implementation). This evaluation suggests coverage of a broad range of social reasoning methods in which the ESB framework can be successfully applied. In the following sections, we look at each of these case studies individually.

6.1. Joint intentions in communication

Joint Intentions (JI) (Cohen and Levesque, 1991) are one of the oldest coordination frameworks in multiagent systems research, and are based on extending the notion of "intention" (as choice with commitment) to teams of collaborating agents. Roughly speaking, the foundation of JI is the notion of a joint persistent goal (JPG) – a joint mental state which requires that every agent is trying to achieve a joint goal, and commits to notifying every other agent in the team when the goal either becomes unachievable or has been achieved.

To re-implement JI-based reasoning and communication using ESB, we use the description of the Request Conversation Protocol (RCP) for establishing a JPG by Kumar et al Kumar et al. (2002), as shown in Figure 2 (with edges as communication acts and nodes capturing the "social state"). Once agent X requests an action of Y, X holds a so-called *persistent weak achievement goal* (PWAG) toward Y. The statement

$pwag(X,Y,A,Q)$ denotes that agent X has a persistent goal to achieve A, if "relevance condition" Q holds, and will have a persistent goal to notify Y that A is achieved, or becomes impossible or irrelevant (i.e. $\neg Q$ holds). If Y agrees, Y now also holds a PWAG toward X to achieve the goal, which creates a JI (a JPG all participants are aware of). X has a goal for Y to do some action, and Y has a goal to do this action relative to X's goal (we omit the case where X cancels the request).

[Figure 2 about here.]

[Table 1 about here.]

Translating this to ESB, four expectations are needed to form and act on JIs in the RCP from an agent's point of view as shown in Table 1:

ExpJI If X holds a PWAG and Y holds a similar PWAG relative to it, the JI regarding A is established. If A becomes true, or Q ceases to be present, or failure occurs (indicated by a "*fail*" fact here), an **ExpNotify** expectation is raised which will trigger notification of the other agent.

ExpReq If Y receives `request`(X,Y,A,Q) from X, Y expects that X holds $pwag(X,Y,A,Q)$.

ExpAgree If X receives `agree`(Y,X,A,Q) from Y, X will also expect that Y expects $pwag(Y,X,A,[Q \wedge pwag(X,Y,A,Q)])$, given that this is a response to a previous `request` message (for clarity, **ExpReq** captures an agent receiving a request). This relevancy condition for Y's PWAG also includes that X still has the commitment toward the goal, since it makes no sense for an agent to perform a requested action if the requester is no longer committed to it.

ExpNotify Given that a JI exists this expectation will cause a belief *must-Notify*(X,Y)

to become true, which is supposed to block all plans at the BDI level except those that cause a notification message to be sent to Y (this can be done by placing guards on the respective plans).

This is an example of a very simple type of social reasoning, effectively based on a conjunction of separate $pwag(\ldots)$ facts depending on the sequence of messages exchanged. As can be seen from the expectations, hardly any tests and responses are required, as the only dynamic belief concerns determining when notification of the other agent is necessary. In fact, this ESB implementation of JI is particularly "condition-heavy" in the sense that it makes very little use of tests (only $T(\text{ExpJI})$ is non-trivial), and expected beliefs are simply switched "on" and "off" depending on context conditions that capture most of the semantics here. Alternatively, one could have opted for a "response-heavy" design where each expectation corresponds to a state in the RCP diagram, and ExpJI expectations are explicitly added and removed from EXP_A depending on the situation. The choice of design here depends on what part of the dynamics the user wants to make explicit for behaviour checking (as the expectation graph is determined by tests and operations on EXP_A). In our formalisation above, for example, one would not be able to make decisions contingent on future communicative behaviour of X or Y, as the messages are not used as tests. The reasoning agent could only anticipate different outcomes for $A \vee \neg Q \vee fail$. Note that the ESB version of JI reconstructs the agent's reasoning that follows logically from the "mentalistic" semantics of the communication acts – whether or not these are correct in the actual system depends on whether agents' internal reasoning mechanism implements them correctly.

As regards *strategies*, in this simple set of expectations, there is no real need to choose a constrained strategy, as the number of transitions in the expectation graph is very small. We shall see cases in which strategies become more useful in more complex examples below.

In terms of behaviours (*condition* $\rhd$ *action* pairs), only two behaviours are required for an agent to act upon joint intentions, that is:

$$\mathit{ji}(\mathit{self}, Y, A, Q) \ \rhd \ \mathit{add\_belief}(\mathit{haveJI}(Y, A, Q))$$

$$\mathit{mustNotify}(\mathit{self}, Y, A, Q) \ \rhd \mathit{add\_belief}(\mathit{triggerNotify}(Y, A, Q))$$

i.e. the agent simply adds a corresponding belief at the BDI level, where the different names for beliefs (*haveJI*, *triggerNotify*) are only chosen to illustrate that at the BDI level beliefs might be taken from a different language of primitives and that behaviours act as an explicit interface between ESB and the general practical reasoning level. What these belief revision operations effectively achieve would be that a guard like *haveJI*$(\ldots)$ on all plans for joint action would ensure that no such action is initiated without a joint intention having been previously formed. Similarly, *triggerNotify*$(\ldots)$ would trigger an intention to notify the other agent of the required fact. While this may seem a very obvious and simple case of social reasoning, it shows how ESB can be used to make the reasoning pattern explicit and generate actual social behaviour from a declarative specification of the mechanism in concrete implementations.

Moreover, although the above is all that is strictly necessary to express joint intentions, ESB provides an easy route to simply and generically query the social reasoning mechanism. An example of this is determining when to request a joint action, and when to agree to one. We could postulate that we should request this if it is possible in the future to hold a JI toward A, and it is desired:

$$(\mathbf{desire}\,A) \wedge \mathbf{E}\Diamond(ji(self,Y,A)) \rhd add\_belief(requestJI(A))$$

The condition is similar for agreeing to a request, the only difference being the requirement that a request has been received:

$$(\mathbf{desire}\,A) \wedge \texttt{request}(self,Y,A,Q) \wedge\ \mathbf{E}\Diamond(ji(self,Y,A) \rhd add\_belief(agreeJI(A))$$

While these behaviours might not seem to add much in our concrete set of expectations, they would help avoid engaging in RCPs if, for example, additional knowledge was available about agents who will never agree to a request.

6.2. Norm-autonomous agents

The second case study we use to illustrate the broad applicability of our framework is taken from the area of normative systems, which is usually concerned with formulating and reasoning about system-level rules for agent behaviour to achieve certain social objectives. Within this space, we deliberately choose the Normative Agent architecture (NoA) (Kollingbaum, 2005) as an example of a framework which relies less on strict regimentation of agent behaviour and allows agents to make decisions about which norms to adhere to autonomously. This allows us to move from a purely "socio-centric" social reasoning mechanism like Joint Intentions toward more "agent-centric" frameworks, and thus to illustrate that ESB is also useful to capture this kind of social reasoning.

NoA provides the usual concepts of *obligations*, *permissions* and *prohibitions*, and uses these within the general practical reasoning cycle of the agent, operating on explicitly represented plans and goals. Thereby, it enables the agent to deliberate over conflicting prohibitions and permissions, and to detect inconsistencies with obligations. The architecture assumes that plans explicitly state their effects (so that obligations can trigger plan selection, as obligations refer to states of affairs that should be achieved), and it also makes use of explicit plan and goal representations to be able to apply prohibitions selectively (e.g. when only certain instances of an abstract plan need to be prohibited, NoA is able to derive appropriate constraints that still permit application of the same plan for other instantiations).

Without going into too much detail on the general NoA architecture, we illustrate its translation to ESB here using the blocks world example presented by Kollingbaum (Kollingbaum, 2005, p. 69). The example features two norms for a block moving robot:

```
obligation (                prohibition (
    robot,                      robot,
    achieve clear(X),           achieve on(X,"c"),
    not clear(X),               TRUE,
    clear(X)                    FALSE
)                           )
```

The first of these states that `robot` is obliged to achieve the state `clear(X)` (this is called the “action specification” in NoA). It becomes active if `not clear(X)`, and ceases to be relevant (“expires”) if `clear(X)`. The second norm prohibits `robot` from achieving `on(X,"c")` for a specific block `c`. This norm is always active and never expires.

In ESB, each norm corresponds to two expectations: a "norm expectation" to handle the deontic content of the norm by asserting appropriate beliefs concerning the action specification of the norm, and the "complement" which is needed to store knowledge of the norm while it is not active so that it can be re-activated (this is not needed for norms that never expire, like the prohibition above).

The expectations for the two norms above are shown in Table 2.

[Table 2 about here.]

As regards the obligation, $\mathit{achieve}(X)$ is the belief added when the OblClear(C) norm is adopted, and $\mathit{clear}(X)$ controls addition/removal of OblClear(X) and its complement OblClearCompl(X). The prohibition, on the other hand, can be expressed pretty trivially as it simply establishes a permanent social belief $\neg \mathit{allowed}(\mathit{on}(X,c))$.

Note that the ESB style adopted for implementing this example is more "response-heavy" than in our Joint Intentions example. In fact, OblClear(X) has no expectation condition at all, and the only conditions we use are $\neg \mathit{relaxOC}$ and $\neg \mathit{relaxPO}$ to represent domain-dependent situations in which the obligation/prohibition would be relaxed (dropped) – they are simply placeholders in our example.

To translate these beliefs to BDI-level behaviour, we need a mapping from the different types of action specifications provided by NoA into ESB, as shown in Table 3. Apart from obligations, the norms translate to actual BDI plan selection policies by means of a predicate $\mathit{allowed}(\ldots)$ which can be used as a guard on plan selection rules (note that this predicate does not have a built-in semantics for ESB, it is introduced as a special belief at the BDI level). For obligations, things are more complicated, as plan selection policies need to be modified – this is discussed in more detail in the next section.

[Table 3 about here.]

6.2.1. *Controlling permissions, prohibitions, and obligations.* NoA uses explicit representations of the effects of plans. Given that deliberation normally heavily relies on plan selection, this is necessary as obligations, permissions and prohibitions might refer to states of affairs (rather than to actions/plans directly), and an agent needs to know what the effects of a plan would be before selecting it in order to be able to check how that plan relates to existing norms. In the Blocks World domain, an `unstack` plan, for example, could be described as follows:

```
plan unstack(X,Y)
     precondition(on(X,Y)),
     effects(ontable(X), not on(X,Y),clear(Y) )
 (
    achieve clear(X);
    primitive doMove(X,Table);
 )
```

This plan should only be selected if it is motivated by an obligation and the effects are permitted by the norms according to some deliberation strategy. In ESB-RS, this requires modification of plans at the BDI (in our case, Jason) level, to make the effects explicit, so that the plan would look like this:

```
+!unstack(X,Y): block(X) & block(Y) & on(X,Y)
```

```
                  & effects(unstack(X,Y))
    <- !achieve(clear(X));
       !move(X,table).
```

where:

```
effects(unstack(X,Y)):- effect(unstack(X,Y)) & effect(ontable(X))
                       &  effect(clear(Y)) & effect(notA(on(X,Y))).
```

is the statement that makes the effects of the plan explicit, and ensures that the plan itself, its actions, sub-goals, and post-conditions are admissible, based on the following predicate:

```
effect(X) :-  not ¬allowed(X)
 | (¬allowed(X) & allowed(Y) & moreSpecific(Y,X))
 | (¬allowed(X) & allowed(Z) & sameScope(Z,X)
                             & moreRecent(Z,X))
 | (¬allowed(X) & allowed(W) & intersection(W,X)
                             & moreRecent(W,X)).
```

This (plan-independent, general) rule also enables the agent to deal with contradictory `allowed` propositions regarding the same action/fact at different levels of abstraction: it will make `effect`(X) true either if X is not explicitly disallowed itself, or if (i) a more specific instance, (ii) a more recent and equally specific, or (iii) a more recent, semantically overlapping instance of X *is* allowed, while X itself is disallowed. For states of affairs, an explicit `notA` predicate is used. This is required for implementation. The reason for this is that it is not required to check if P is in the belief base (i.e. ¬P is true), but rather the predicate is used when matching effect specifications of plans. `notA` in a plan effect specification indicates explicitly that not P is achieved as an effect.

The functions above to check the scope (`moreSpecific`, `sameScope`, `in-tersection`) work by comparing the variables that are unified for the effects' parameters. These are implemented as simple support plans in Jason that recurse over the parameters of the effects statements being compared to determine the difference in the sets of ground variables. For example, a more specific set of effects would have more ground variables than a less specific one.

As before with the *relaxOC* and *relaxPO* facts, this just captures one possible method of conflict resolution, and is only shown as an illustration of how the ESB-RS implementation of NoA norms can be used by the BDI-level practical reasoner to implement norm-autonomous behaviour (effectively specifying how the agent chooses to interpret the norms at a local reasoning level).

To trigger plan selection, as a result of norms requiring the achievement of a goal or the selection of a plan, two small extensions to the BDI reasoner are required. Firstly, to select a particular plan it is simply adopted as an intention (and the above mechanism ensures that plan is allowed). Secondly, if particular effects are required, an internal action is triggered to select a plan based on the above effect information. This process is described in more detail in Wallace (2010).

The fact that these interventions at the BDI reasoning level are required in ESB mirrors the use of a "bespoke" BDI reasoner in the original NoA architecture, that can reason explicitly about plan effects. While not strictly part of ESB, we include it here to show that aspects of practical reasoning that are clearly non-social (such as plan selection, in this case) cannot be dealt with by ESB in a straightforward way, at least not in combination with an off-the-shelf practical reasoning system like Jason – wherever a social reasoning method affects general practical reasoning, system implementation will have to cross the boundary between the two.

6.2.2. *Translation from NoA to ESB.* To summarise, we briefly recap the steps to translate a NoA specification into ESB, given the additional generic ESB machinery described above.

In a first step, plans are translated into plans in the Jason BDI component of ESB-RS. Preconditions become guards, and the effects are specified as separate "effects" predicates, as per the previous section. The actions in the NoA plan form the body of the Jason plan.

Then, for each norm, the activation condition becomes the test of the complement expectation (the condition is that the norm is not relaxed, and Φ is set to *true*). The response to the test of this expectation is to add the norm-expectation. This has a *true* condition, and the Φ term is as set out in Table 3 depending on the type of norm. The deactivation condition of the norm becomes the test of this expectation, and the responses swap the expectation for its complement.

The process of this translation from a NoA specification into ESB-RS is summarised in Figure 3. Note that this translation algorithm presents a naive, generic method of translation. As shown in the example above certain optimisations can be performed where pairs of expectations can be contracted.

[Figure 3 about here.]

6.2.3. *Discussion.* The sets of expectation structures and support plans described above provide a general way to transform the key features of NoA specifications into an ESB representation.

Although the method of implementation is different, the case study has shown that the key properties of a complex social reasoning system can be preserved when it is specified in ESB. The examples given here only come from a trivial Blocks World scenario presented by Kollingbaum (Kollingbaum, 2005), but a more complex “three-party Letter of Credit” protocol described in the same work was also implemented. We were able to see that the ESB-NoA implementation behaved the same in terms of norm activation and plans chosen (Wallace, 2010).

Considering the “practical vs. social reasoning divide” in the NoA implementation, more work is required on the practical reasoner side than would be desirable. This is in part due to the inability to reason over expectation content. Assuming that norms are captured with expectations and that the conflict resolution function must act on these expectations, conflict detection at least must happen at the BDI level as it cannot occur in strategies or behaviours. It is also not surprising that work was required on the practical reasoning side as the NoA scheme is fairly close to practical reasoning in its implementation – it represents a BDI extension to handle social influences.

Not considered in the ESB-RS NoA implementation is the notion of adding new norms previously unknown to the system. This would require an extension to the current ESB implementation to allow for adoption of new expectations, but it is easy to sketch how this might be done: When new expectations are added it would be necessary to re-generate the FSM specification for the model checker and to run behaviour-condition style checks on the new expectations to ensure they caused no problems.

A benefit brought to NoA from ESB-RS is that it situates NoA style normative reasoning in an existing agent infrastructure (that of Jason). The NoA architecture presented in (Kollingbaum, 2005) does not situate the reasoner in an agent infrastructure. Combined with the modularity shown in the ESB case studies this presents a strong case for the argument that a framework for general purpose social reasoning can ease implementation of such systems.

6.3. Opponent modelling in games

As an example of simple opponent modelling using ESB, we present a portion of the reasoning rules for a Rummy (Rummy.com, 2008) playing agent (previously introduced in (Wallace and Rovatsos, 2009)). Card games, particularly Rummy, can provide a nice simple example domain as the strategies that the opponents might use are known (for simple games) but depend on the cards they hold – which are unknown. For the purposes of this example it is only important to understand that players are trying to collect either runs of cards in a suit, e.g. $\{2\clubsuit,3\clubsuit,4\clubsuit\}$, or sets, e.g. $\{2\clubsuit,2\diamondsuit,2\heartsuit\}$ (given a card X, other cards that could be part of a set/run with that card are referred to as a "member of X's sets/runs").

The example involves only two players, where the ESB agent A reasons about its opponent B picking up a $2\diamondsuit$. The example considers only relatively naive reasoning on A's part: that B will either be collecting a run of diamonds around 2, or a set of 2s (A assumes B won't hedge her bets by collecting cards for both a run and a set). The expectations to enable this kind of reasoning are shown in Table 4. Note that the use of variables in ESB-RS specifications in Section 5.2 allows us to write these rules at a more general level (e.g. for any card value rather than just for 2s in a single set of expectations). For ease of readability, we have simplified the description of expectations in places. For example, we say "member of a run" instead of a disjunction of all the cards in that run. We keep the example specific to 2s here to be able to show a relatively concise expectation graph below.

[Table 4 about here.]

Figure 4 shows the accessible expectation graph generated from the expectations in Table 4 and initial state S2. Vertices are labelled with the expected belief Φ of each expectation tuple, as this makes consideration of the relevant behaviours easier. The initial state S2 is highlighted using double lines. Edges represent transitions specified by the responses. A loop from a state indicates that the ρ^+, ρ^- responses of some expectations do not cause a change in state. It is worth noting that this example is different from the previous ones in that a lot more reasoning is encoded in the responses, creating a richer graph.

[Figure 4 about here.]

A simple strategy that can be applied here is to consider the expectation graph to a depth of one from the current state and to ignore loopbacks (shown by the bold states in Figure 4, the strategy is applied assuming a current state of S5). This could be appropriate, because responses are triggered by observations (which update the status of tests) and these must come from observing cards played or picked up. Near the end of the game, trying to predict actions too far ahead is meaningless. In terms of implementation, this strategy would require slightly more control over strategy than that presented in this paper. Although it is possible to express a condition that must hold in the next state in CTL (so check a depth of one) only adopting this strategy at the end of the game implies an ability to change strategy according to some beliefs of the agent (which would be influenced by expectations, if required). It is easy to see how the single strategy approach could be extended to a set of strategies preconditioned on agent beliefs, but this is not a functionality of the current ESB-RS implementation. The strategy example still serves to illustrate the principle here, however.

The richer nature of the graph hints at the benefits of being to be able to use more complex CTL formulae to encode more sophisticated strategies. To give a game-theory inspired example, consider an agent wishing to adopt a minimax strategy for reasoning. Such a strategy would assume that the opponent will pick the worst scoring option for the agent in each move, and so for its own move the agent will attempt to maximise this minimum. To apply this concept to our running example, we can model "moves", which are the observed results of expectation$T$ tests. The ESB agent's moves are not represented in the expectation graph for simplicity, but its behaviour should maximise utility given worst-case play by the opponent. We restrict the expectation graph by applying this principle before evaluating the preconditions of behaviours. This requires making assumptions about the evaluation function the opponent uses, which (to keep things simple) might look something like this:

- A pickup scores -1, as the opponent has gained a useful card.
- A discard scores $+1$ as the opponent has lost a card.
- Ignoring a card scores 0 as it tells us nothing.

Based on this, the strategy is to build a tree of the graph to a certain maximum depth, and then restrict it to only those paths which score lowest (adding the scores down each path to gain a total at the leaves). The agent would then only consider behaviours whose conditions apply in this minimax-restricted view of future possible strategies.

Again, this is a hypothetical example to illustrate the potential of using advanced strategies in ESB, the ESB-RS implementation would need some extension to support it. The model used to check behaviour conditions would need some small changes to support restriction to a certain depth, namely adding a counter defined to be incremented on subsequent states from the initial state. Similarly, it would need a domain-specific extension to score certain expectations according to their properties (as above) and use these scores to constrain the model as per the depth. This is a limitation of ESB as currently envisaged, and, once more, is caused by the limitation that reasoning over expectations is limited to the meta level and not based on inference over expectation content. Ideally in this case we would score for minimax based on the properties of an expectation, without requiring the domain-specific knowledge of each expectation's score.

7. RELATED WORK

While the multiagent literature abounds in proposals for specific social reasoning mechanisms and frameworks, there are relatively few attempts to model social reasoning as a general reasoning process allowing for enough generality to cover different approaches, and to support this model by a practical computational architecture.

Along these lines, the work that is probably most closely related to ESB is the expectation-based framework introduced by Cranefield in (Cranefield, 2006, 2007) and later refined in (Cranefield et al., 2011). Like ESB, the authors use the concept of expectations to track dynamically changing beliefs about the environment. They introduce a hybrid Metric Interval Temporal Logic hyMITL$^{\pm}$ to capture social expectations such as norms and commitments in electronic institutions. The original aim of this work was to build a monitoring tool using hyMITL$^{\pm}$ that could be used as a compliance monitor based on the evolution of the system and observation of agent behaviour. More recently, their concept of expectations as first-class constructs has been further developed Cranefield et al. (2011), extending the formalism to capture nesting of expectations and describing an approach to model-check the resulting language.

As in ESB, no first-order concepts of specific social reasoning constructs like commitments or norms are built into the logic. Instead, the logic is argued to be capable of capturing the relevant properties that occur in many social reasoning frameworks. Due to its focus on detecting violations of expectations, monitoring is works by checking conditions that must be met at certain times, hence the focus on time in the logic, which is not a focus in our formalism. The structure of expectations in the hiMITL$^{\pm}$-based approach is also similar to ESB. An expectation is held under some condition and then monitored until it is fulfilled or violated according to some other specification. In contrast to ESB, the social expectations in that work do not say anything about how expectations influence decision making and how the interaction of an agent's social and practical reasoning should be managed.

This work is extended to an implementation also utilising the Jason BDI engine in Ranathunga et al. (2011). Superficially, this seems similar to work on ESB-RS. Although the terminology is similar to that used here and in previous ESB papers, the intent and construction of the system is different. Again the authors' goal is the monitoring of agent's expectations with respect to norms. This is in contrast to the ESB attempt to provide a generic framework to capture and implement social reasoning schemes. Their approach to implementation also differs. They extend Jason via the agent environment and additional internal actions for the reasoner, to allow external plug-in expectation monitoring software. The agent reasoning is still all part of the BDI plans and the interface from the external monitor is via beliefs. This belief-based interface is similar to the approach in ESB-RS, however here we present a split in the reasoning, with social reasoning separated from general practical reasoning. This is central to our approach, and a key difference.

The definition of expectations used in the above work follows from that of Castelfranchi in Castelfranchi (2005). These authors consider an expectation to be an expected belief and a goal related to it. In ESB we talk about an expectation as being the tuple containing the expected belief and the manner in which it affects other expectations (the responses). This is because we view them as tightly interlinked – expectations are closely related to other expectations and the overall dynamics of the system. Castelfranchi also considers the dynamics of expectations, though in terms of mental attitudes such as surprise and relief, relating to the actual outcome of expected beliefs. Instead, in ESB we are interested in how responses, which capture expectation change, affect future social beliefs and actions.

Also mentioning expectations, though restricted to interaction protocols, is the work of Alberti et al. Alberti et al. (2006). They present a tool that combines different expectations to make inferences over sets of them. This would be complementary to our ESB approach, but we focus on the interface to practical reasoning instead. It is similar to ESB in that they talk about expectations in the light of events actually occurring, but they do not link them to agent behaviour – specification and verification is the focus.

Another approach that uses expectation-based modelling and reasoning is presented by Nickles et al. Nickles et al. (2004). While seemingly similar, however, the "expectation networks" presented there are only loosely related to our concept of expectations. Expectation networks focus on communication between agents, and are used as a means to capture communication patterns probabilistically. On top of the probabilistic model, occurrence of these patterns is subject to logical conditions identifying certain social states of the system (or beliefs). The motivation for this approach is that communication is necessary for agents to interact and so any social relationships between agents can be described in terms of the communication structures that emerge among them. Whilst this might be true, there are types of agent reasoning that cannot be captured in this way, e.g. reasoning about mental attitudes of other agents and inferred states of others, which requires revising beliefs based on the observations of others' behaviour as suggested by ESB. In this respect, the work described in (Nickles et al., 2004) could be seen as a far more specific form of social reasoning concerned only with identification of observable communication patterns and the prediction of future communication using on past experience.

Dennis et al. Dennis et al. (2008) propose to create an intermediate agent description language that can be used to capture different commonly used BDI programming languages. The authors observe that there are many different BDI implementations that share common constructs, and a common representation might allow easier comparison between approaches and transfer of concepts. This goal is certainly similar to that addressed by ESB, which tackles the same problem as far as it concerns social reasoning, albeit at a higher level of abstraction. However, the language Dennis et al. propose is intended to generalise over existing agent programming languages – we suggest that ESB can be used to implement many abstract social reasoning methods for which no practical computational framework has been presented at all, and that it allows the combination of different such methods in the same implementation.

At the intersection between social reasoning and general practical reasoning, Dignum et al. have developed ideas similar to ours, albeit in a more specific manner. Their B-DOING architecture (Dignum et al., 2000, 2002b,a) extends BDI to include obligations and norms as motivations for an agent's actions beyond the normal beliefs and desires of the BDI model. A logical system is presented to describe the semantics of their approach and how these concepts may be captured and integrated by way of preference orderings over goals. In contrast to the meta-level ESB reasoning, the object-level reasoning of B-DOING agents is based on the assumption that these basic social motivations for action are sufficient to capture an agent's social concerns. A claim about *how* social reasoning should be done in the general case is implicit to this approach. Quite differently, we do not wish to promote any particular *style* of reasoning, but propose a generic BDI-compatible social reasoning "plugin" that allows many such styles.

The focus of their work is also more toward the logical representation of agent reasoning than the computational concerns of ESB, which arise from our desire to bridge the gap from theory to implementation.

Later work (van der Vecht et al., 2009) is more closely related to the ESB approach, as it focuses on more generalised decision making based on the influence that events processed by the agent have on the agent's practical reasoning cycle. They propose an architecture for dynamically modifying the ways in which the agent chooses to be influenced by others, i.e. the suggested method allows for meta-reasoning about social influence, and thus aims at a similarly general level of social reasoning as ESB does. Also, the suggested meta-reasoning can be used to determine what type of reasoning and "how much of it" is appropriate in a particular situation. This is similar to the goal of bounding social reasoning in ESB. However, while the paper gives examples of a rule-based controller for this type of mechanism in a particular domain, no general language and computational engine is given that could aid designers in implementing agents with these capabilities more generally.

Sindlar et al. Sindlar et al. (2009, 2011) propose the use of abduction to infer mental states of other agents given assumptions about their internal structure and observations of agent behaviour in a BDI context. The underlying idea of tracking an agent's assumptions about hidden properties of the system based on observations is similar to the basic idea behind expectations in ESB, however the authors apply this idea in a much more specific way: Their focus is exclusively on BDI agents (as the agents that are *being* modelled, not the modelling agent herself), and on producing explanations for observed behaviour, whereas ESB focuses on bounding reasoning (through strategies) and on social decision making and acting (through behaviours). Thus, abduction-style reasoning to infer mental states could be something that might be implemented in ESB, but there are many other instances of ESB that could be very different in style.

8. CONCLUSION

In this paper, we have presented a computational framework for practical social reasoning based on making the update mechanisms regarding social beliefs explicit, by connecting them to observations. Our framework allows for defining how sets of such belief-observation-update rules, which we call *expectations*, are reasoned over, taking into account the reasoning resources available to the agent, by specifying *strategies* that define the set of future expectation states considered when making inferences. Furthermore, it specifies how decision making and actual *behaviour* generation can be influenced at a general practical reasoning level by using behaviours that are conditioned on properties regarding current and future expectations.

The ESB framework makes the following contributions: From an *analysis* point of view, we introduce the conceptual machinery necessary to capture practical social reasoning mechanisms at the desired level of generality, i.e. using abstractions of expectations, strategies, and behaviours as metaphors for structuring social reasoning process. These abstractions allow for an analysis of existing social reasoning mechanisms.

We claim that our conceptual framework is general enough to cover most existing social reasoning abstractions, but specific enough to allow us to focus on the computational processes that are specific to social reasoning. This also allows for a direct comparison of different social reasoning methods using our framework.

From an *engineering* point of view, we present a formal model and specification language that allows designers to write declarative specifications of social reasoning rules like those presented in the examples above in a modular, extensible, and domain-independent way. This language supports designers in implementing socially intelligent agents by allowing them to formulate the "theory" of a given social reasoning mechanism in a uniform language. Moreover, following a declarative paradigm enables the reuse of specifications, and eases the task of combining different specifications.

The central contribution from an *implementation* point of view is the "transduction" of these specifications to executable process models. We have shown that such deductive synthesis can be achieved within the limits of ESB, and that simple algorithms can be used to prune the models generated from a specification to avoid unnecessary complexity (Wallace and Rovatsos, 2009). Using model checking methods, we obtain a high degree of expressiveness in terms of behaviour checking, and we illustrate how an ESB reasoning engine can be seamlessly integrated with BDI-style agent programming languages.

Overall, the significance of our contribution lies in enabling agent designers to implement a broad range of (combinations of) social reasoning methods in a uniform, generic framework without having to "embed" the behavioural properties that should follow from the method in the general practical reasoning mechanism of the agent in an implicit way. Instead, they are provided with the "social reasoning middleware" that can be directly coupled to the remaining agent program, and provides an additional level of abstraction. This, in turn, is expected to simplify the implementation of such social reasoning algorithms, which is still the exception rather than the rule. And, in theory, it should have similar benefits as the introduction of generic practical reasoning schemes such as BDI: The abstractions provided by BDI, desires, plans, intentions, goals, commitments, intention reconsideration, belief revision, etc. enabled a standardisation that provided an additional, generic layer for modelling architectures that previously were largely developed in an *ad hoc*, incomparable fashion. In the long term, we would hope that something similar can be achieved by ESB and other such social reasoning frameworks.

In the future, we would like to develop more practical and usable tools for ESB so that more large-scale, complex specifications of social reasoning mechanisms can be tested, and we can learn more about and thus improve the process of engineering the social reasoning layer of multiagent systems in real-world applications.

REFERENCES

ALBERTI, MARCO, MARCO GAVANELLI, EVELINA LAMMA, FEDERICO CHESANI, PAOLA MELLO, and PAOLO TORRONI. 2006. Compliance verification of agent interaction: A logic-based software tool. Applied Artificial Intelligence, **20**(2-4):133–157.

BORDINI, RAFAEL H., JOMI FRED HÜBNER, and MICHAEL WOOLDRIDGE. 2007. Programming Multi-Agent Systems in AgentSpeak using Jason. Wiley.

CASTELFRANCHI, CRISTIANO. 2005. Mind as an anticipatory device: For a theory of expectations. *In* Brain, Vision, and Artificial Intelligence. *Edited by* M. De Gregorio, V. Di Maio, M. Frucci, and C. Musio, Volume 3704 of *Lecture Notes in Computer Science*. Springer Berlin / Heidelberg, pp. 258–276. ISBN 978-3-540-29282-1.

CIMATTI, ALESSANDRO, EDMUND CLARKE, FAUSTO GIUNCHIGLIA, and MARCO ROVERI. 1999. NUSMV: a new Symbolic Model Verifier. *In* Proceedings Eleventh Conference on Computer-Aided Verification (CAV'99). *Edited by* N. Halbwachs and D. Peled, Number 1633 in Lecture Notes in Computer Science. Springer, pp. 495–499.

COHEN, PHILLIP R., and HECTOR J. LEVESQUE. 1991. Teamwork. Nous, **25**(4).

CRANEFIELD, STEPHEN. 2006. A rule language for modelling and monitoring social expectations in multi-agent systems. *In* Proceedings of Coordination, Organizations, Institutions, and Norms in Multi-Agent Systems, AAMAS 2005, Volume 3913 of *Lecture Notes in Computer Science*, Springer, pp. 246 – 258.

CRANEFIELD, STEPHEN. 2007. Modelling and monitoring social expectations in multi-agent systems. *In* Proceedings of Coordination, Organizations, Institutions, and Norms in Agent Systems II AAMAS 2006, Volume 4386 of *Lecture Notes in Computer Science*, pp. 308 – 321.

CRANEFIELD, STEPHEN, MICHAEL WINIKOFF, and WAMBERTO VASCONCELOS. 2011. Modelling and monitoring interdependent expectations. *In* Coordination, Organization, Institutions and Norms in Agent Systems, 12th International Workshop.

DENNIS, LOUISE A., BERNDT FARWER, RAFAEL H. BORDINI, MICHAEL FISHER, and MICHAEL WOOLDRIDGE. 2008. A common semantic basis for BDI languages. *In* Proceedings of Programming Multi-Agent Systems 5th International Workshop, ProMAS 2007, Volume 4908 of *Lecture Notes in Computer Science*, pp. 124 – 139.

DIGNUM, FRANK, DAVID KINNY, and LIZ SONENBERG. 2002a. From Desires, Obligations and Norms to Goals. Cognitive Science Quarterly, **2**(3-4):407–430.

DIGNUM, FRANK, DAVID KINNY, and LIZ SONENBERG. 2002b. Motivational attitudes of agents: on desires, obligations, and norms. *In* Second International Workshop of Central and Eastern Europe on Multi-Agent Systems, CEEMAS 2001, Volume 2296 of *Lecture Notes in Artificial Intelligence*, pp. 83 – 92.

DIGNUM, FRANK, DAVID MORLEY, ELIZABETH A. SONENBERG, and LAWRENCE CAVEDON. 2000. Towards socially sophisticated BDI agents. *In* Proceedings of the Fourth International Conference on MultiAgent Systems, Springer, pp. 111 – 18.

GABBAY, DOV M., MARK A. REYNOLDS, and MARCELLO FINGER. 2000. Temporal Logic: Mathematical Foundations and Computational Aspects, Volume 2. Oxford Science Publications.

KOLLINGBAUM, MARTIN. 2005. Norm-governed Practical Reasoning Agent. Ph. D. thesis, University of Aberdeen, Dept. of Computing Science.

KUMAR, SANJEEV, MARCUS J. HUBER, PHILIP R. COHEN, and DAVID R. MCGEE. 2002. Toward a formalism for conversation protocols using joint intention theory. Computational Intelligence, **18**(2):174 – 228. ISSN 08247935.

NICKLES, MATTHIAS, MICHAEL ROVATSOS, WILFRIED BRAUER, and GERHARD WEISS. 2004. Towards a Unified Model of Sociality in Multiagent Systems. International Journal of Computer & Information Science, **5**(2).

RANATHUNGA, S., S. CRANEFIELD, and M. PURVIS. 2011. Integrating expectation handling into Jason. *In* Proceedings of the 9th International Workshop on Programming Multi-Agent Systems (ProMAS), pp. 105–120.

Rummy.com 2008. The Rules of Rummy. http://rummy.com/rummyrules.html. last checked 29/09/2008.

SHOHAM, YOAV, and KEVIN LEYTON-BROWN. 2009. Multiagent Systems – Algorithmic, Game-Theoretic, and Logical Foundations. Cambridge University Press.

SINDLAR, MICHAL P., MEHDI M. DASTANI, FRANK DIGNUM, and JOHN-JULES CH. MEYER. 2009. Mental state abduction of BDI-based agents. *In* Declarative Agent Languages and Technologies VI, Volume 5397 of *Lecture Notes in Computer Science*, pp. 161–178.

SINDLAR, MICHAL P., MEHDI M. DASTANI, FRANK DIGNUM, and JOHN-JULES CH. MEYER. 2011. Programming Mental State Abduction. *In* Proc. of 10th Int. Conf. on Autonomous Agents and Multiagent Systems (AAMAS 2011). *Edited by* K. Tumer, P. Yolum, L. Sonenberg, and P. Stone, Taipei, Taiwan, pp. 301–308.

VAN DER VECHT, BOB, FRANK DIGNUM, and JOHN-JULES. CH. MEYER. 2009. Autonomy and coordination: Controlling external influences on decision making. *In* 2009 IEEE/WIC/ACM International Conference on Web Intelligence and Intelligent Agent Technology, Volume 2, pp. 92 –95.

WALLACE, IAIN. 2010. Social Reasoning in Multi-Agent Systems with the Expectation-Strategy-Behaviour Framework. Ph. D. thesis, School of Informatics, University of Edinburgh.

WALLACE, IAIN, and MICHAEL ROVATSOS. 2009. Bounded Social Reasoning in the ESB Framework. *In* Proceedings of the Eighth International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS 2009), pp. 1097–1104.

WALLACE, IAIN, and MICHAEL ROVATSOS. 2011. Executing specifications of social reasoning agents. *In* Declarative Agent Languages and Technologies VIII, Volume 6619 of *Lecture Notes in Artificial Intelligence*, Toronto, ON, Canada, pp. 112 – 129.

WOOLDRIDGE, MICHAEL. 2000. Reasoning about Rational Agents. Intelligent robotics and autonomous agents. The MIT Press.

WOOLDRIDGE, M. 2009. An Introduction to Multiagent Systems, 2nd edition. John Wiley & Sons, Chichester, England.

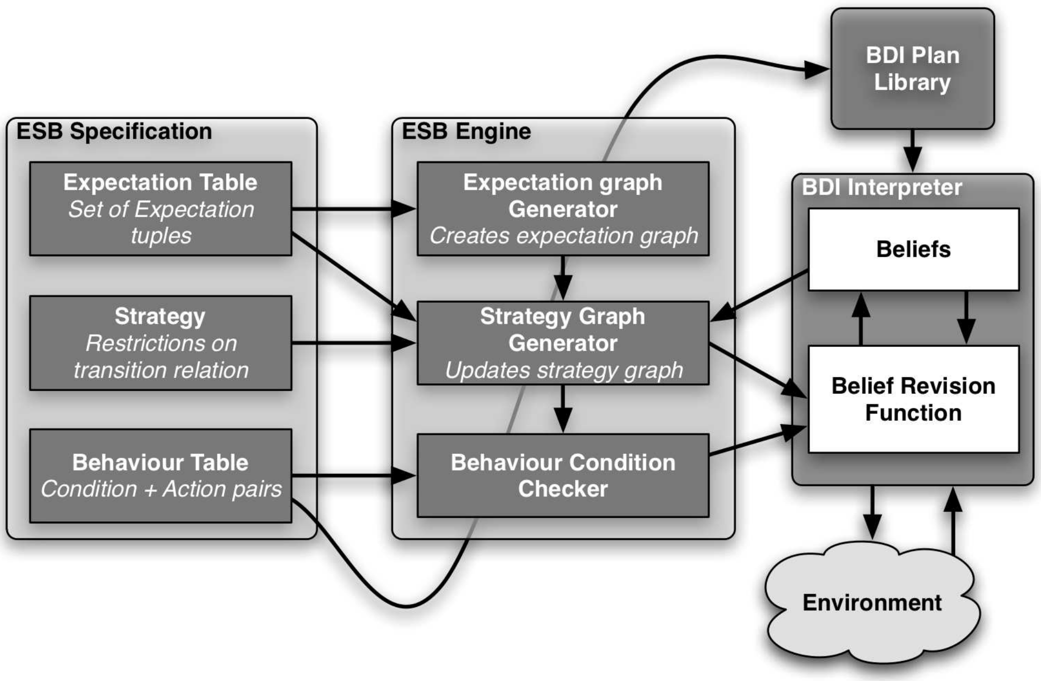

FIGURE 1. Conceptual Overview of an ESB-RS Agent

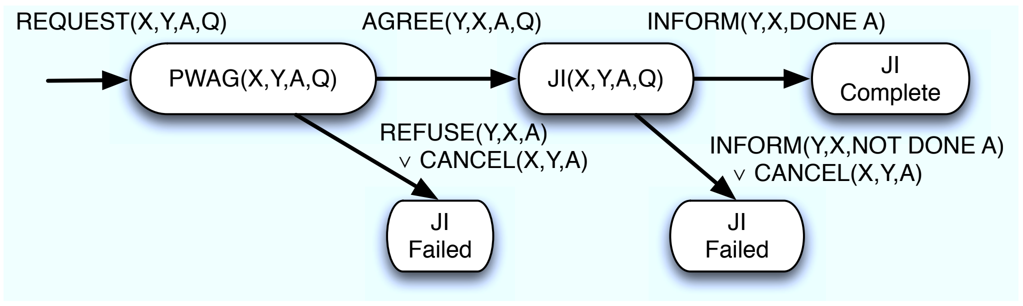

FIGURE 2. The Request Conversation Protocol (adapted from Kumar et al. (2002))

```
Require: $NoA_P$, Set of NoA plans
Require: $NoA_N$, Set of NoA norms
Ensure: $BDI_P$, Set of BDI plans and rules that capture $NoA_P$
Ensure: $EXP_N$, Set of ESB-RS expectation capturing $NoA_N$

  for all $P \in NoA_P$ do
      Create a BDI plan $P' \in BDI_P$
      Name($P'$) = Name($P$)
      Precondition($P'$) = precondition($P$) & effects(Name($P'$))
      Body($P'$) = Body($P$)

      Create a rule effects(Name($P'$))
      effects(Name($P'$)) = effect($E_1$) & effect($E_2$)... & effect(notA($N_1$) & effect(notA($N_2$)...
      Where $E_1, E_2 \ldots E_N$ are achieved by effects(P)
            $N_1, N_2 \ldots N_N$ are explicitly not achieved by effects(P)
  end for
  for all $N \in NoA_N$ do
      Create an expectation $E_{1N} \in EXP_N$
      $C(E_{1N})$ = true
      $\Phi(E_{1N})$ = ActivationCondition($N$), mapped as per Table 3
      $T(E_{1N})$ = ExpirationCondition($N$)
      $\rho^+(E_{1N})$ = add( $\{E_{2N}\}$), del($\{E_{1N}\}$)
      $\rho^-(E_{1N})$ is empty

      Create an expectation $E_{2N} \in EXP_N$
      $C(E_{2N})$ $\neg$relax, where this is a belief specific to relaxing this norm.
      $\Phi(E_{2N})$ = true
      $T(E_{2N})$ = ActivationCondition($N$)
      $\rho^+(E_{2N})$ = add( $\{E_{1N}\}$), del($\{E_{2N}\}$)
      $\rho^-(E_{2N})$ is empty
  end for
```

FIGURE 3. Pseudocode describing a translation from a NoA specification into ESB-RS.

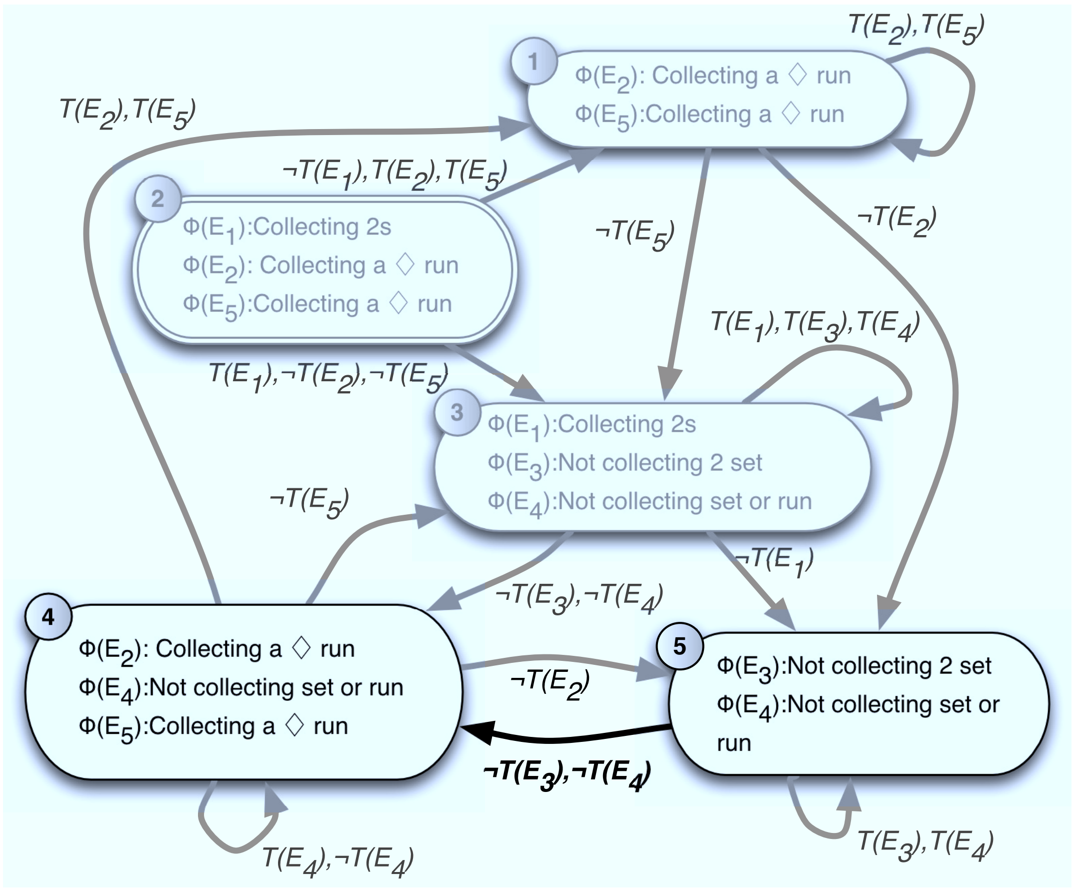

FIGURE 4. Accessible Expectation Graph, showing the states reachable from the initial state S2. Bold states represent states accessible with a simple depth-1 strategy from state S5.

TABLE 1. JI Expectations. All are in the initial set $\mathbf{EXP}_A$ of both agents. X,Y are both variables, but the example in the text assumes X requests a JI with Y.

	ExpJI(A,Q)	**ExpAgree**(A,Q)
C	$pwag(X,Y,A,Q) \wedge$ $pwag(Y,X,A,$ $[Q \wedge pwag(X,Y,A,Q)])$	$\texttt{agree}(Y,X,A,Q) \wedge$ $pwag(X,Y,A,Q)$
Φ	$ji(X,Y,A,Q)$	$pwag(Y,X,A,$ $[Q \wedge pwag(X,Y,A,Q)])$
T	$A \vee \neg Q \vee fail$	*false*
ρ^+	$add(\text{ExpNotify}(X,Y,A,Q))$	-
ρ^-	$del(\text{ExpNotify}(X,Y,A,Q))$	-

e	**ExpReq**(A,Q)	**ExpNotify**(A,Q)
C	$\texttt{request}(X,Y,A,Q)$	$ji(X,Y,A,Q)$
Φ	$pwag(X,Y,A,Q)$	$mustNotify(X,Y)$
T	*false*	*false*
ρ^+	-	-
ρ^-	-	-

TABLE 2. Obligation-to-Clear and Prohibition-On expectations in the blocks' world example

	OblClear(X)	OblClearCompl(X)	ProhibitionOn(X)
C	$true$	$\neg relaxOC$	$\neg relaxPO$
Φ	$achieve(clear(X))$	$true$	$\neg allowed(on(X,c))$
T	$clear(X)$	$\neg clear(X)$	$false$
ρ^+	$add($OblClearCompl$(X))$, $del($OblClear$(X))$	$add($OblClear$(X))$, $del($OblClearCompl$(X))$	-
ρ^-	-	-	-

TABLE 3. Translation of Action Specifications into ESB Representation. S represents a state of affairs, P a plan (or single action). Duals for negative cases (e.g. $\mathit{prohibit}(\neg \mathit{perform}(P))$ for $\mathit{obligation}(\mathit{perform}(P))$) are omitted for brevity.

Norm	ESB Φ belief
$\mathit{permission}(\mathit{achieve}(S))$	$\mathit{allowed}(S)$
$\mathit{prohibition}(\mathit{achieve}(S))$	$\neg \mathit{allowed}(S)$
$\mathit{obligation}(\mathit{achieve}(S))$	$\mathit{achieve}(S)$, select a plan with $\mathit{allowed}(S)$ in its effects
$\mathit{permission}(\mathit{perform}(P))$	$\mathit{allowed}(P)$
$\mathit{prohibition}(\mathit{perform}(P))$	$\neg \mathit{allowed}(P)$
$\mathit{obligation}(\mathit{perform}(P))$	select plan P

TABLE 4. Expectations in the Rummy example: the labels of initial expectations (1,2,5) are shown in bold face; tests are broken down into $+(\ldots)$ and $-(\ldots)$ events that would confirm/disappoint the expectation

N	C	Φ	T	ρ^+	ρ^-
1	*B* picked up 2♢	*B* collecting 2s	*B* picks up an available 2 and has discarded none	remove({2,5}) add({3,4})	remove({1})
2	*B* picked up 2♢	*B* collecting ♢ run	*B* picks up member of 2♢ run and has discarded none	remove({1,3,4}) add({5})	remove({2,5}) add({3,4})
3	*B* discarded 2♣	*B* not collecting 2s	*B* ignores a 2	-	remove({1,3}) add({2,5})
4	*B* ignored a 2	*B* not after 2s for run or set	*B* ignores a 2	-	remove({1,3}) add({2,5})
5	*B* picked up 3♢	*B* collecting ♢ run	*B* picks up another member of 3♢ run and has discarded none	remove({1,3,4}) add({2})	remove({2,5}) add({1,3,4})