Implementing Commitment-Based Interaction

Michael Winikoff

RMIT University
Melbourne, Australia
michael.winikoff@rmit.edu.au

Although agent interaction plays a vital role in MAS, and message-centric approaches to agent interaction have their drawbacks, present agent-oriented programming languages do not provide support for implementing agent interaction that is flexible and robust. Instead, messages are provided as a primitive building block. In this paper we consider one approach for modelling agent interactions: the commitment machines framework. This framework supports modelling interactions at a higher level (using social commitments), resulting in more flexible interactions. We investigate how commitment-based interactions can be implemented in conventional agent-oriented programming languages. The contributions of this paper are: a mapping from a commitment machine to a collection of BDI-style plans; extensions to the semantics of BDI programming languages; and an examination of two issues that arise when distributing commitment machines (turn management and race conditions) and solutions to these problems.

Implementing Commitment-Based Interactions*

Michael Winikoff
School of Computer Science and IT
RMIT University
Melbourne, Australia
michael.winikoff@rmit.edu.au

ABSTRACT

Although agent interaction plays a vital role in MAS, and message-centric approaches to agent interaction have their drawbacks, present agent-oriented programming languages do not provide support for implementing agent interaction that is flexible and robust. Instead, messages are provided as a primitive building block. In this paper we consider one approach for modelling agent interactions: the commitment machines framework. This framework supports modelling interactions at a higher level (using social commitments), resulting in more flexible interactions. We investigate how commitment-based interactions can be implemented in conventional agent-oriented programming languages. The contributions of this paper are: a mapping from a commitment machine to a collection of BDI-style plans; extensions to the semantics of BDI programming languages; and an examination of two issues that arise when distributing commitment machines (turn management and race conditions) and solutions to these problems.

Categories and Subject Descriptors

I.2.11 [**Artificial Intelligence**]: Distributed Artificial Intelligence—*Multiagent systems*; I.2.5 [**Artificial Intelligence**]: Programming Languages and Software

General Terms

Design

Keywords

Commitment Machines, Agent Interaction, Agent Oriented Programming Languages, Belief Desire Intention (BDI)

*The author would to acknowledge the support of the Australian Research Council under grant LP0453486, in collaboration with Agent Oriented Software.

AAMAS'07 May 14–18 2007, Honolulu, Hawai'i, USA.

1. INTRODUCTION

Agents are social, and agent interaction plays a vital role in multi-agent systems. Consequently, design and implementation of agent interaction is an important research topic.

The standard approach for designing agent interactions is *message-centric*: interactions are defined by interaction protocols that give the permissible sequences of messages, specified using notations such as finite state machines, Petri nets, or Agent UML.

It has been argued that this message-centric approach to interaction design is not a good match for intelligent agents. Intelligent agents should exhibit the ability to persist in achieving their goals in the face of failure (robustness) by trying different approaches (flexibility). On the other hand, when following an interaction protocol, an agent has limited flexibility and robustness: the ability to persistently try alternative means to achieving the interaction's aim is limited to those options that the protocol's designer provided, and in practice, message-centric design processes do not tend to lead to protocols that are flexible or robust.

Recognising these limitations of the traditional approach to designing agent interactions, a number of approaches have been proposed in recent years that move away from message-centric interaction protocols, and instead consider designing agent interactions using higher-level concepts such as social commitments [8, 10, 18] or interaction goals [2]. There has also been work on richer forms of interaction in specific settings, such as *teams* of cooperative agents [5, 11].

However, although there has been work on *designing* flexible and robust agent interactions, there has been virtually no work on providing programming language support for *implementing* such interactions. Current Agent Oriented Programming Languages (AOPLs) do not provide support for implementing flexible and robust agent interactions using higher-level concepts than messages. Indeed, modern AOPLs [1], with virtually no exceptions, provide only simple message sending as the basis for implementing agent interaction.

This paper presents what, to the best of our knowledge, is the *second* AOPL to support high-level, flexible, and robust agent interaction implementation. The first such language, STAPLE, was proposed a few years ago [9], but is not described in detail, and is arguably impractical for use by non-specialists, due to its logical basis and heavy reliance on temporal and modal logic.

This paper presents a scheme for extending BDI-like AOPLs to support direct implementation of agent interactions that are designed using Yolum & Singh's *commitment machine* (CM) framework [19]. In the remainder of this paper we briefly review commitment machines and present a simple abstraction of BDI AOPLs which lies in the common subset of languages such as Jason, 3APL, and CAN. We then present a scheme for translating commitment

machines to this language, and indicate how the language needs to be extended to support this. We then extend our scheme to address a range of issues concerned with distribution, including turn tracking [7], and race conditions.

2. BACKGROUND

2.1 Commitment Machines

The aim of the commitment machine framework is to allow for the definition of interactions that are more flexible than traditional message-centric approaches. A *Commitment Machine* (CM) [19] specifies an interaction between entities (e.g. agents, services, processes) in terms of *actions* that change the interaction state. This interact state consists of fluents (predicates that change value over time), but also *social commitments*, both base-level and conditional.

A *base-level social commitment* is an undertaking by debtor A to creditor B to bring about condition p, denoted $\mathsf{C}(A, B, p)$. This is sometimes abbreviated to $\mathsf{C}(p)$, where it is not important to specify the identities of the entities in question. For example, a commitment by customer C to merchant M to make the fluent $paid$ true would be written as $\mathsf{C}(C, M, paid)$.

A *conditional social commitment* is an undertaking by debtor A to creditor B that *should condition q become true*, A will then commit to bringing about condition p. This is denoted by $\mathsf{CC}(A, B, q, p)$, and, where the identity of the entities involved is unimportant (or obvious), is abbreviated to $\mathsf{CC}(q \rightsquigarrow p)$ where the arrow is a reminder of the causal link between q becoming true and the creation of a commitment to make p true. For example, a commitment to make the fluent $paid$ true once $goods$ have been received would be written $\mathsf{CC}(goods \rightsquigarrow paid)$.

The semantics of commitments (both base-level and conditional) is defined with rules that specify how commitments change over time. For example, the commitment $\mathsf{C}(p)$ (or $\mathsf{CC}(q \rightsquigarrow p)$) is *discharged* when p becomes true; and the commitment $\mathsf{CC}(q \rightsquigarrow p)$ is replaced by $\mathsf{C}(p)$ when q becomes true. In this paper we use the more symmetric semantics proposed by [15] and subsequently reformalised by [14]. In brief, these semantics deal with a number of more complex cases, such as where commitments are created when conditions already hold: if p holds when $\mathsf{CC}(p \rightsquigarrow q)$ is meant to be created, then $\mathsf{C}(q)$ is created instead of $\mathsf{CC}(p \rightsquigarrow q)$.

An interaction is defined by specifying the entities involved, the possible contents of the interaction state (both fluents and commitments), and (most importantly) the actions that each entity can perform along with the preconditions and effects of each action, specified as add and delete lists.

A commitment machine (CM) defines a range of possible interactions that each start in some state[1], and perform actions until reaching a final state. A final state is one that has no base-level commitments. One way of visualising the interactions that are possible with a given commitment machine is to generate the finite state machine corresponding to the CM. For example, figure 1 gives the FSM[2] corresponding to the NetBill [18] commitment machine: a simple CM where a customer (C) and merchant (M) attempt to trade using the following actions[3]:

- $sendRequest(C) \Rightarrow request$
- $sendQuote(M) \Rightarrow offer$
  where $offer \equiv promiseGoods \wedge promiseReceipt$ and $promiseGoods \equiv \mathsf{CC}(M, C, accept, goods)$ and $promiseReceipt \equiv \mathsf{CC}(M, C, pay, receipt)$
- $sendAccept(C) \Rightarrow accept$
  where $accept \equiv \mathsf{CC}(C, M, goods, pay)$
- $sendGoods(M) \Rightarrow promiseReceipt \wedge goods$
  where $promiseReceipt \equiv \mathsf{CC}(M, C, pay, receipt)$
- $sendEPO(C) : goods \Rightarrow pay$
- $sendReceipt(M) : pay \Rightarrow receipt.$

The commitment $accept$ is the customer's promise to pay once goods have been sent, $promiseGoods$ is the merchant's promise to send the goods once the customer accepts, and $promiseReceipt$ is the merchant's promise to send a receipt once payment has been made.

As seen in figure 1, commitment machines can support a range of interaction sequences.

2.2 An Abstract Agent Programming Language

Agent programming languages in the BDI tradition (e.g. dMARS, JAM, PRS, UM-PRS, JACK, AgentSpeak(L), Jason, 3APL, CAN, Jadex) define agent behaviour in terms of event-triggered plans, where each plan specifies what it is triggered by, under what situations it can be considered to be applicable (defined using a so-called *context condition*), and a *plan body*: a sequence of steps that can include posting events which in turn triggers further plans. Given a collection of plans and an event e that has been posted the agent first collects all plans types that are triggered by that event (the *relevant* plans), then evaluates the context conditions of these plans to obtain a set of *applicable* plan instances. One of these is chosen and is executed.

We now briefly define the formal syntax and semantics of a Simple Abstract (BDI) Agent Programming Language (SAAPL). This language is intended to be an abstraction that is in the common subset of such languages as Jason [1, Chapter 1], 3APL [1, Chapter 2], and CAN [16]. Thus, it is intentionally incomplete in some areas, for instance it doesn't commit to a particular mechanism for dealing with plan failure, since different mechanisms are used by different AOPLs.

An agent program (denoted by Π) consists of a collection of plan clauses of the form $e : C \leftarrow P$ where e is an event, C is a context condition (a logical formula over the agent's beliefs), and P is the plan body. The plan body is built up from the following constructs. We have the empty step ϵ which always succeeds and does nothing, operations to add ($+b$) and delete ($-b$) beliefs, sending a message m to agent N ($\uparrow^N m$), and posting an event[4] (e). These can be sequenced ($P; P$).

$$\begin{aligned} C &::= b \mid C \wedge C \mid C \vee C \mid \neg C \mid \exists x.C \\ P &::= \epsilon \mid +b \mid -b \mid e \mid \uparrow^N m \mid P; P \end{aligned}$$

Formal semantics for this language is given in figure 2. This semantics is based on the semantics for AgentSpeak given by [12], which in turn is based on the semantics for CAN [16]. The semantics is in the style of Plotkin's Structural Operational Semantics, and assumes that operations exist that check whether a condition

[1]Unlike standard interaction protocols, or finite state machines, there is no designated initial state for the interaction.

[2]The finite state machine is software-generated: the nodes and connections were computed by an implementation of the axioms (available from http://www.winikoff.net/CM) and were then laid out by *graphviz* (http://www.graphviz.org/).

[3]We use the notation $A(X) : P \Rightarrow E$ to indicate that action A is performed by entity X, has precondition P (with ": P" omitted if empty) and effect E.

[4]We use $\downarrow^N m$ as short hand for the event corresponding to receiving message m from agent N.

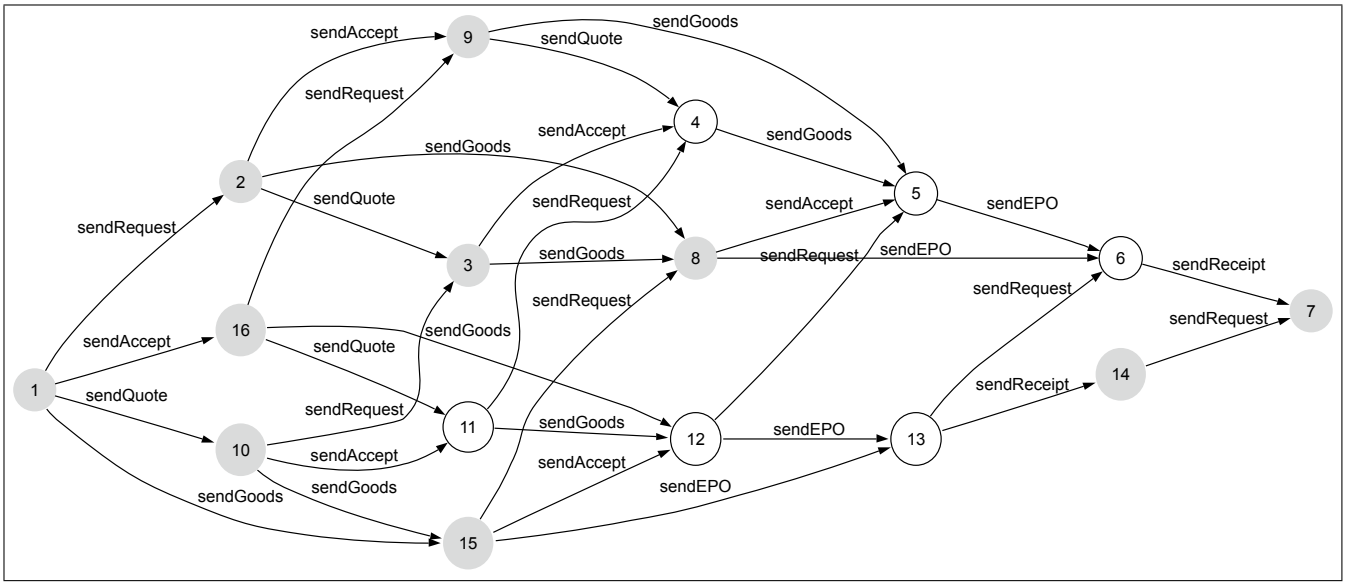

Figure 1: Finite State Machine for NetBill (shaded = final states)

follows from a belief set, that add a belief to a belief set, and that delete a belief from a belief set. In the case of beliefs being a set of ground atoms these operations are respectively consequence checking ($B \models C$), and set addition ($B \cup \{b\}$) and deletion ($B \setminus \{b\}$). More sophisticated belief management methods may be used, but are not considered here.

We define a *basic* configuration $S = \langle Q, N, B, P\rangle$ where Q is a (global) message queue (modelled as a sequence[5] where messages are added at one end and removed from the other end), N is the name of the agent, B is the beliefs of the agent and P is the plan body being executed (i.e. the intention). We also define an *agent* configuration, where instead of a single plan body P there is a *set* of plan instances, Γ. Finally, a complete MAS is a pair $\langle Q, As\rangle$ of a global message queue Q and a *set* of agent configurations (without the queue, Q). The global message queue is a sequence of triplets of the form *sender*:*recipient*:*message*.

A transition $S_0 \longrightarrow S_1$ specifies that executing S_0 a single step yields S_1. We annotate the arrow with an indication of whether the configuration in question is basic, an agent configuration, or a MAS configuration. The transition relation is defined using rules of the form $\dfrac{}{S \longrightarrow S'}$ or of the form $\dfrac{S' \longrightarrow S_r}{S \longrightarrow S'_r}$; the latter are conditional with the top (numerator) being the premise and the bottom (denominator) being the conclusion.

Note that there is non-determinism in SAAPL, e.g. the choice of plan to execute from a set of applicable plans. This is resolved by using selection functions: $\mathcal{S}_{\mathcal{O}}$ selects one of the applicable plan instances to handle a given event, $\mathcal{S}_{\mathcal{I}}$ selects which of the plan instances that can be executed should be executed next, and $\mathcal{S}_{\mathcal{A}}$ selects which agent should execute (a step) next.

3. IMPLEMENTING COMMITMENT-BASED INTERACTIONS

In this section we present a mapping from a commitment machine to a collection of SAAPL programs (one for each role). We begin by considering the simple case of two interacting agents, and assume that the agents take turns to act. In section 4 we relax these assumptions.

Each action $A(X) : P \Rightarrow E$ is mapped to a number of plans: there is a plan (for agent X) with context condition P that performs the action (i.e. applies the effects E to the agent's beliefs) and sends a message to the other agent, and a plan (for the other agent) that updates its state when a message is received from X. For example, given the action $sendAccept(C) \Rightarrow accept$ we have the following plans, where each plan is preceded by "M:" or "C:" to indicate which agent that plan belongs to. Note that where the identify of the sender (respectively recipient) is obvious, i.e. the other agent, we abbreviate $\uparrow^N m$ to $\uparrow m$ (resp. $\downarrow^N m$ to $\downarrow m$). Turn taking is captured through the event $\imath$ (short for "interact"): the agent that is active has an $\imath$ event that is being handled. Handling the event involves sending a message to the other agent, and then doing nothing until a response is received.

C: $\imath : true \leftarrow +accept; \uparrow sendAccept.$
M: $\downarrow sendAccept : true \leftarrow +accept; \imath.$

If the action has a non-trivial precondition then there are two plans in the recipient: one to perform the action (if possible), and another to report an error if the action's precondition doesn't hold (we return to this in section 4). For example, the action $sendReceipt(M) : pay \Rightarrow receipt$ generates the following plans:

M: $\imath : pay \leftarrow +receipt; \uparrow sendReceipt.$
C: $\downarrow sendReceipt : pay \leftarrow +receipt; \imath.$
C: $\downarrow sendReceipt : \neg pay \leftarrow \ldots$ report error $\ldots$.

In addition to these plans, we also need plans to start and finish the interaction. An interaction can be completed whenever there are no base-level commitments, so both agents have the following plans:

$\imath : \neg\exists p.\mathsf{C}(p) \leftarrow \uparrow done.$
$\downarrow done : \neg\exists p.\mathsf{C}(p) \leftarrow \epsilon.$
$\downarrow done : \exists p.\mathsf{C}(p) \leftarrow \ldots$ report error $\ldots$.

An interaction is started by setting up an agent's initial beliefs, and then having it begin to interact. Exactly how to do this depends on the agent platform: e.g. the agent platform in question may offer a simple way to load beliefs from a file. A generic approach that is a little cumbersome, but is portable, is to send each of the agents involved in the interaction a sequence of $init$ messages, each

[5]The "+" operator is used to denote sequence concatenation.

$$\frac{}{\langle Q, N, B, +b\rangle \xrightarrow{\text{Basic}} \langle Q, N, B \cup \{b\}, \epsilon\rangle}$$

$$\frac{}{\langle Q, N, B, -b\rangle \xrightarrow{\text{Basic}} \langle Q, N, B \setminus \{b\}, \epsilon\rangle}$$

$$\frac{\Delta = \{P_i\theta | (t_i : c_i \leftarrow P_i) \in \Pi \wedge t_i\theta = e \wedge B \models c_i\theta\}}{\langle Q, N, B, e\rangle \xrightarrow{\text{Basic}} \langle Q, N, B, \mathcal{S}_{\mathcal{O}}(\Delta)\rangle}$$

$$\frac{\langle Q, N, B, P_1\rangle \xrightarrow{\text{Basic}} \langle Q', N, B', P'\rangle}{\langle Q, N, B, P_1; P_2\rangle \xrightarrow{\text{Basic}} \langle Q', N, B', P'; P_2\rangle}$$

$$\frac{}{\langle Q, N, B, \epsilon; P\rangle \xrightarrow{\text{Basic}} \langle Q, N, B, P\rangle}$$

$$\frac{}{\langle Q, N, B, \uparrow^{N_B} m\rangle \xrightarrow{\text{Basic}} \langle Q + N{:}N_B{:}m, N, B, \epsilon\rangle}$$

$$\frac{Q = N_A{:}N{:}m + Q'}{\langle Q, N, B, \Gamma\rangle \xrightarrow{\text{Agent}} \langle Q', N, B, \Gamma \cup \{\downarrow^{N_A} m\}\rangle}$$

$$\frac{P = \mathcal{S}_{\mathcal{I}}(\Gamma) \quad \langle Q, N, B, P\rangle \xrightarrow{\text{Basic}} \langle Q', N, B', P'\rangle}{\langle Q, N, B, \Gamma\rangle \xrightarrow{\text{Agent}} \langle Q', N, B', (\Gamma \setminus \{P\}) \cup \{P'\}\rangle}$$

$$\frac{P = \mathcal{S}_{\mathcal{I}}(\Gamma) \quad P = \epsilon}{\langle Q, N, B, \Gamma\rangle \xrightarrow{\text{Agent}} \langle Q, N, B, (\Gamma \setminus \{P\})\rangle}$$

$$\frac{\langle N, B, \Gamma\rangle = \mathcal{S}_{\mathcal{A}}(As) \quad \langle Q, N, B, \Gamma\rangle \xrightarrow{\text{Agent}} \langle Q', N, B', \Gamma'\rangle}{\langle Q, As\rangle \xrightarrow{\text{MAS}} \langle Q', (As \cup \{\langle N, B', \Gamma'\rangle\}) \setminus \{\langle N, B, \Gamma\rangle\}\rangle}$$

Figure 2: Operational Semantics for SAAPL

containing a belief to be added; and then send one of the agents a $start$ message which begins the interaction. Both agents thus have the following two plans:

$\downarrow init(B) : true \leftarrow +B.$
$\downarrow start : true \leftarrow \imath.$

Figure 3 gives the SAAPL programs for both merchant and customer that implement the NetBill protocol. For conciseness the error reporting plans are omitted.

We now turn to refining the context conditions. There are three refinements that we consider. Firstly, we need to prevent performing actions that have no effect on the interaction state. Secondly, an agent may want to specify that certain actions that it is able to perform should not be performed unless additional conditions hold. For example, the customer may not want to agree to the merchant's offer unless the goods have a certain price or property. Thirdly, the context conditions of the plans that terminate the interaction need to be refined in order to avoid terminating the interaction prematurely.

For each plan of the form $\imath : P \leftarrow +E; \uparrow m$ we replace the context condition P with the enhanced condition $P \wedge P' \wedge \neg E$ where P' is any additional conditions that the agent wishes to impose, and $\neg E$ is the negation of the effects of the action. For example, the customer's payment plan becomes (assuming no additional conditions, i.e. no P'): $\imath : goods \wedge \neg pay \leftarrow +pay; \uparrow sendEPO$.

For each plan of the form $\downarrow m : P \leftarrow +E; \imath$ we could add $\neg E$ to the precondition, but this is redundant, since it is already checked by the performer of the action, and if the action has no effect then

Customer's plans:
$\imath : true \leftarrow +request; \uparrow sendRequest.$
$\imath : true \leftarrow +accept; \uparrow sendAccept.$
$\imath : goods \leftarrow +pay; \uparrow sendEPO.$
$\downarrow sendQuote : true \leftarrow +promiseGoods;$
$\quad +promiseReceipt; \imath.$
$\downarrow sendGoods : true \leftarrow +promiseReceipt; +goods; \imath.$
$\downarrow sendReceipt : pay \leftarrow +receipt; \imath.$

Merchant's plans:
$\imath : true \leftarrow +promiseGoods;$
$\quad +promiseReceipt; \uparrow sendQuote.$
$\imath : true \leftarrow +promiseReceipt; +goods; \uparrow sendGoods.$
$\imath : pay \leftarrow +receipt; \uparrow sendReceipt.$
$\downarrow sendRequest : true \leftarrow +request; \imath.$
$\downarrow sendAccept : true \leftarrow +accept; \imath.$
$\downarrow sendEPO : goods \leftarrow +pay; \imath.$

Shared plans (i.e. plans of both agents):
$\imath : \neg\exists p.\mathsf{C}(p) \leftarrow \uparrow done.$
$\downarrow done : \neg\exists p.\mathsf{C}(p) \leftarrow \epsilon.$
$\downarrow init(B) : true \leftarrow +B.$
$\downarrow start : true \leftarrow \imath.$

Where
$accept \equiv \mathsf{CC}(goods \rightsquigarrow pay)$
$promiseGoods \equiv \mathsf{CC}(accept \rightsquigarrow goods)$
$promiseReceipt \equiv \mathsf{CC}(pay \rightsquigarrow receipt)$
$offer \equiv promiseGoods \wedge promiseReceipt$

Figure 3: SAAPL Implementation of NetBill

the sender won't perform it and send the message (see also the discussion in section 4).

When specifying additional conditions (P'), some care needs to be taken to avoid situations where progress cannot be made because the only action(s) possible are prevented by additional conditions. One way of indicating preference between actions (in many agent platforms) is to *reorder* the agent's plans. This is clearly safe, since actions are not prevented, just considered in a different order.

The third refinement of context conditions concerns the plans that terminate the interaction. In the Commitment Machine framework any state that has no base-level commitment is final, in that the interaction may end there (or it may continue). However, only some of these final states are desirable final states. Which final states are considered to be desirable depends on the domain and the desired interaction outcome. In the NetBill example, the desirable final state is one where the goods have been sent and paid for, and a receipt issued (i.e. $goods \wedge pay \wedge receipt$). In order to prevent an agent from terminating the interaction too early we add this as a precondition to the termination plan:

$\imath : goods \wedge pay \wedge receipt \wedge \neg\exists p.\mathsf{C}(p) \leftarrow \uparrow done.$

Figure 4 shows the plans that are changed from figure 3.

In order to support the realisation of CMs, we need to change SAAPL in a number of ways. These changes, which are discussed below, can be applied to existing BDI languages to make them "commitment machine supportive". We present the three changes, explain what they involve, and for each change explain how the change was implemented using the 3APL agent oriented programming language. The three changes are:

1. extending the beliefs of the agent so that they can contain commitments;

Customer's plans:
$\imath : \neg request \leftarrow +request; \uparrow sendRequest.$
$\imath : \neg accept \leftarrow +accept; \uparrow sendAccept.$
$\imath : goods \wedge \neg pay \leftarrow +pay; \uparrow sendEPO.$

Merchant's plans:
$\imath : \neg offer \leftarrow +promiseGoods; +promiseReceipt;$
$\uparrow sendQuote.$
$\imath : \neg(promiseReceipt \wedge goods) \leftarrow$
$+promiseReceipt; +goods; \uparrow sendGoods.$
$\imath : pay \wedge \neg receipt \leftarrow +receipt; \uparrow sendReceipt.$

Where
$accept \equiv \mathsf{CC}(goods \rightsquigarrow pay)$
$promiseGoods \equiv \mathsf{CC}(accept \rightsquigarrow goods)$
$promiseReceipt \equiv \mathsf{CC}(pay \rightsquigarrow receipt)$
$offer \equiv promiseGoods \wedge promiseReceipt$

Figure 4: SAAPL Implementation of NetBill with refined context conditions (changed plans only)

2. changing the definition of "$\models$" to encompass implied commitments; and

3. whenever a belief is added, updating existing commitments, according to the rules of commitment dynamics.

Extending the notion of beliefs to encompass commitments in fact requires no change in agent platforms that are prolog-like and support terms as beliefs (e.g. Jason, 3APL, CAN). However, other agent platforms do require an extension. For example, JACK, which is an extension of Java, would require changes to support commitments that can be nested. In the case of 3APL no change is needed to support this.

Whenever a context condition contains commitments, determining whether the context condition is implied by the agent's beliefs ($B \models C$) needs to take into account the notion of implied commitments [15]. In brief, a commitment can be considered to follow from a belief set B if the commitment is in the belief set ($C \in B$), but also under other conditions. For example, a commitment to pay $\mathsf{C}(pay)$ can be considered to be implied by a belief set containing pay because the commitment may have held and been discharged when pay was made true. Similar rules apply for conditional commitments. These rules, which were introduced in [15] were subsequently re-formalised in a simpler form by [14] resulting in the four inference rules in the bottom part of figure 5.

The change that needs to be made to SAAPL to support commitment machine implementations is to extend the definition of $\models$ to include these four rules. For 3APL this was realised by having each agent include the following Prolog clauses:

```
holds(X) :- clause(X,true).
holds(c(P)) :- holds(P).
holds(c(P)) :- clause(cc(Q,P),true), holds(Q).
holds(cc(_,Q)) :- holds(Q).
holds(cc(_,Q)) :- holds(c(Q)).
```

The first clause simply says that anything holds if it is in agent's beliefs (`clause(X,true)` is true if `X` is a fact). The remaining four clauses correspond respectively to the inference rules C1, C2, CC1 and CC2. To use these rules we then modify context conditions in our program so that instead of writing, for example, `cc(m,c, pay, receipt)` we write `holds(cc(m,c, pay, receipt))`.

$$\frac{B' = norm(B \cup \{b\})}{\langle Q, N, B, +b\rangle \longrightarrow \langle Q, N, B', \epsilon\rangle}$$

function norm(B)
  $B' \leftarrow B$
  for each $b \in B$ **do**
    if $b = \mathsf{C}(p) \wedge B \models p$ **then** $B' \leftarrow B' \setminus \{b\}$
    elseif $b = \mathsf{CC}(p \rightsquigarrow q)$ **then**
      if $B \models q$ **then** $B' \leftarrow B' \setminus \{b\}$
      elseif $B \models p$ **then** $B' \leftarrow (B' \setminus \{b\}) \cup \{\mathsf{C}(q)\}$
      elseif $B \models \mathsf{C}(q)$ **then** $B' \leftarrow B' \setminus \{b\}$
      endif
    endif
  endfor
  return B'
end function

$$\frac{B \models P}{B \models \mathsf{C}(P)}\ C1 \qquad \frac{\mathsf{CC}(Q \rightsquigarrow P) \in B \quad B \models Q}{B \models P}\ C2$$

$$\frac{B \models \mathsf{CC}(P \rightsquigarrow Q)}{B \models Q}\ CC1 \qquad \frac{B \models \mathsf{C}(Q)}{B \models \mathsf{CC}(P \rightsquigarrow Q)}\ CC2$$

Figure 5: New Operational Semantics

The final change is to update commitments when a belief is added. Formally, this is done by modifying the semantic rule for belief addition so that it applies an algorithm to update commitments. The modified rule and algorithm (which mirrors the definition of *norm* in [14]) can be found in the top part of figure 5.

For 3APL this final change was achieved by manually inserting `update()` after updating beliefs, and defining the following rules for `update()`:

```
update() <- c(P) AND holds(P)
 | {Deletec(P) ; update()},
update() <- cc(P,Q) AND holds(Q)
 | {Deletecc(P,Q) ; update()},
update() <- cc(P,Q) AND holds(P)
 | {Deletecc(P,Q) ; Addc(Q) ; update()},
update() <- cc(P,Q) AND holds(c(Q))
 | {Deletecc(P,Q) ; update()},
update() <- true | Skip
```

where `Deletec` and `Deletecc` delete respectively a base-level and conditional commitment, and `Addc` adds a base-level commitment.

One aspect that doesn't require a change is linking commitments and actions. This is because commitments don't trigger actions directly: they may trigger actions indirectly, but in general their effect is to prevent completion of an interaction while there are outstanding (base level) commitments.

Figure 6 shows the message sequences from a number of runs of a 3APL implementation of the NetBill commitment machine[6]. In order to illustrate the different possible interactions the code was modified so that each agent selected randomly from the actions that it could perform, and a number of runs were made with the customer as the initiator, and then with the merchant as the initiator. There are other possible sequences of messages, not shown,

[6]Source code is available from http://www.winikoff.net/CM

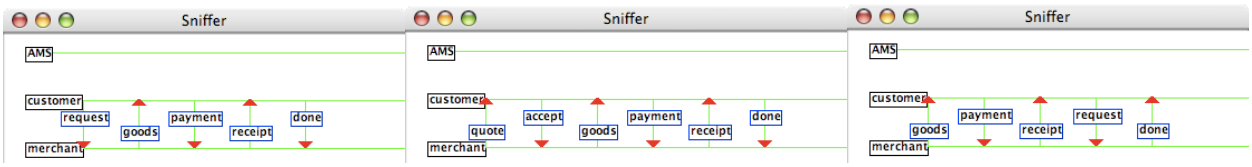

Figure 6: Sample runs from 3APL implementation (alternating turns)

including the obvious one: *request*, *quote*, *accept*, *goods*, *payment*, *receipt*, and then *done*.

One minor difference between the 3APL implementation and SAAPL concerns the semantics of messages. In the semantics of SAAPL (and of most AOPLs), receiving a message is treated as an event. However, in 3APL, receiving a message is modelled as the addition to the agent's beliefs of a fact indicating that the message was received [6]. Thus in the 3APL implementation we have PG rules that are triggered by these beliefs, rather than by any event. One issue with this approach is that the belief remains there, so we need to ensure that the belief in question is either deleted once handled, or that we modify preconditions of plans to avoid handling it more than once. In our implementation we delete these "received" beliefs when they are handled, to avoid duplicate handling of messages.

4. BEYOND TWO PARTICIPANTS

Generalising to more than two interaction participants requires revisiting how turn management is done, since it is no longer possible to assume alternating turns [7].

In fact, perhaps surprisingly, even in the two participant setting, an alternating turn setup is an unreasonable assumption! For example, consider the path (in figure 1) from state 1 to 15 (*sendGoods*) then to state 12 (*sendAccept*). The result, in an alternating turn setup, is a dead-end: there is only a single possible action in state 12, namely *sendEPO*, but this action is done by the customer, and it is the merchant's turn to act! Figure 7 shows the FSM for NetBill with alternating initiative.

A solution to this problem that works in this example, but doesn't generalise[7], is to weaken the alternating turn taking regime by allowing an agent to act twice in a row if its second action is driven by a commitment.

A general solution is to track whose turn it is to act. This can be done by working out which agents have actions that are able to be performed in the current state. If there is only a single active agent, then it is clearly that agent's turn to act. However, if more than one agent is active then somehow the agents need to work out who should act next. Working this out by negotiation is not a particularly good solution for two reasons. Firstly, this negotiation has to be done at every step of the interaction where more than one agent is active (in the NetBill, this applies to seven out of sixteen states), so it is highly desirable to have a light-weight mechanism for doing this. Secondly, it is not clear how the negotiation can avoid an infinite regress situation ("you go first", "no, you go first", ...) without imposing some arbitrary rule. It is also possible to resolve who should act by imposing an arbitrary rule, for example, that the customer always acts in preference to the merchant, or that each agent has a numerical priority (perhaps determined by the order in which they joined the interaction?) that determines who acts.

An alternative solution, which exploits the symmetrical properties of commitment machines, is to not try and manage turn taking.

[7]Consider actions $A_1(C) \Rightarrow p$, $A_2(C) \Rightarrow q$, and $A_3(M) : p \wedge q \Rightarrow r$.

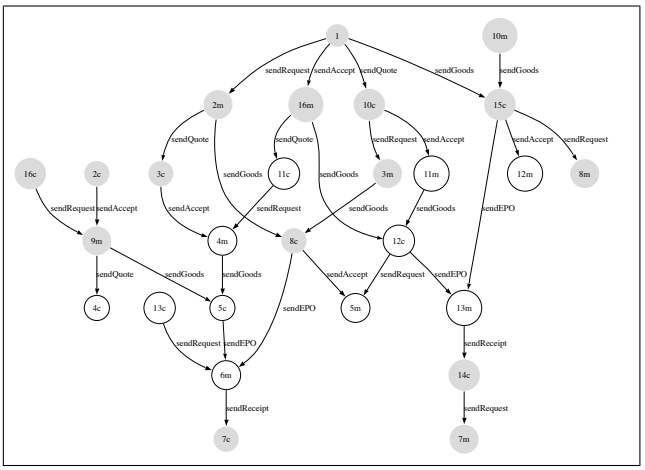

Figure 7: NetBill with alternating initiative

Instead of tracking and controlling whose turn it is, we simply allow the agents to act freely, and rely on the properties of the interaction space to ensure that "things work out", a notion that we shall make precise, and prove, in the remainder of this section.

The issue with having multiple agents be active simultaneously is that instead of all agents agreeing on the current interaction state, agents can be in different states. This can be visualised as each agent having its own copy of the FSM that it navigates through where it is possible for agents to follow different paths through the FSM. The two specific issues that need to be addressed are:

1. Can agents end up in different final states?

2. Can an agent be in a position where an error occurs because it cannot perform an action corresponding to a received message?

We will show that, because actions commute under certain assumptions, agents cannot end up in different final states, and furthermore, that errors cannot occur (again, under certain assumptions).

By "actions commute" we mean that the state resulting from performing a sequence of actions $A_1 \ldots A_n$ is the same, regardless of the order in which the actions are performed. This means that even if agents take different paths through the FSM, they still end up in the same resulting state, because once all messages have been processed, all agents will have performed the same set of actions. This addresses the issue of ending up in different final states. We return to the possibility of errors occurring shortly.

Definition 1 (Monotonicity) *An action is* monotonic *if it does not delete*[8] *any fluents or commitments. A Commitment Machine is*

[8]That is *directly* deletes, it is fine to discharge commitments by adding fluents/commitments.

monotonic if all of its actions are monotonic. (Adapted from [14, Definition 6])

Theorem 1 *If A_1 and A_2 are monotonic actions, then performing A_1 followed by A_2 has the same effect on the agent's beliefs as performing A_2 followed by A_1. (Adapted from [14, Theorem 2]).*

This assumes that both actions *can* be performed. However, it is possible for the performance of A_1 to disable A_2 from being done. For example, if A_1 has the effect $+p$, and A_2 has precondition $\neg p$, then although both actions may be enabled in the initial state, they cannot be performed in either order. We can prevent this by ensuring that actions' preconditions do not contain negation (or implication), since a monotonic action cannot result in a precondition that is negation-free becoming false. Note that this restriction only applies to the original action precondition, P, not to any additional preconditions imposed by the agent (P'). This is because only P is used to determine whether another agent is able to perform the action.

Thus monotonic CMs with preconditions that do not contain negations have actions that commute. However, in fact, the restriction to monotonic CMs is unnecessarily strong: all that is needed is that whenever there is a choice of agent that can act, then the possible actions are monotonic. If there is only a single agent that can act, then no restriction is needed on the actions: they may or may not be monotonic.

Definition 2 (Locally Monotonic) *A commitment machine is* locally monotonic *if for any state S either (a) only a single agent has actions that can be performed; or (b) all actions that can be performed in S are monotonic.*

Theorem 2 *In a locally monotonic CM, once all messages have been processed, all agents will be in the same state. Furthermore, no errors can occur.*

Proof: *Once all messages have been processed we have that all agents will have performed the same action set, perhaps in a different order. The essence of the proof is to argue that as long as agents haven't yet converged to the same state, all actions must be monotonic, and hence that these actions commute, and cannot disable any other actions.*

Consider the first point of divergence, where an agent performs action A and at the same time another agent (call it X_B) performs action B. Clearly, this state has actions of more than one agent enabled, so, since the CM is locally monotonic, the relevant actions must be monotonic. Therefore, after doing A, the action B must still be enabled, and so the message to do B can be processed by updating the recipient agent's beliefs with the effects of B. Furthermore, because monotonic actions commute, the result of doing A before B is the same as doing B before A:

$$\begin{array}{ccc} S & \xrightarrow{A} & S_A \\ \downarrow{\scriptstyle B} & & {\scriptstyle B}\downarrow \\ S_B & \xrightarrow[A]{} & S_{AB} \end{array}$$

However, what happens if the next action after A is not B, but C? Because B is enabled, and C is not done by agent X_B (see below), we must have that C is also monotonic, and hence (a) the result of doing A and B and C is the same regardless of the order in which the three actions are done; and (b) C doesn't disable B, so B can still be done after C.

$$\begin{array}{ccccc} S & \xrightarrow{A} & S_A & \xrightarrow{C} & S_{AC} \\ \downarrow{\scriptstyle B} & & {\scriptstyle B}\downarrow & & {\scriptstyle B}\downarrow \\ S_B & \xrightarrow[A]{} & S_{AB} & \xrightarrow[C]{} & S_{ABC} \end{array}$$

The reason why C cannot be done by X_B is that messages are processed in the order of their arrival[9]*. From the perspective of X_B the action B was done before C, and therefore from any other agent's perspective the message saying that B was done must be received (and processed) before a message saying that C is done.*

This argument can be extended to show that once agents start taking different paths through the FSM all actions taken until the point where they converge on a single state must be monotonic, and hence it is always possible to converge (because actions aren't disabled), so the interaction is error free; and the resulting state once convergence occurs is the same (because monotonic actions commute). □

This theorem gives a strong theoretical guarantee that not doing turn management will not lead to disaster. This is analogous to proving that disabling all traffic lights would not lead to any accidents, and is only possible because the refined CM axioms are symmetrical.

Based on this theorem the generic transformation from CM to code should allow agents to act freely, which is achieved by simply changing $\imath : P \wedge P' \wedge \neg E \leftarrow +E; \uparrow A$ to

$$\imath : P \wedge P' \wedge \neg E \leftarrow +E; \uparrow A; \imath$$

For example, instead of $\imath : \neg request \leftarrow +request; \uparrow sendRequest$ we have $\imath : \neg request \leftarrow +request; \uparrow sendRequest; \imath$.

One consequence of the theorem is that it is not necessary to ensure that agents process messages before continuing to interact. However, in order to avoid unnecessary parallelism, which can make debugging harder, it may still be desirable to process messages before performing actions.

Figure 8 shows a number of runs from the 3APL implementation that has been modified to allow free, non-alternating, interaction.

5. DISCUSSION

We have presented a scheme for mapping commitment machines to BDI platforms (using SAAPL as an exemplar), identified three changes that needed to be made to SAAPL to support CM-based interaction, and shown that turn management can be avoided in CM-based interaction, provided the CM is locally monotonic. The three changes to SAAPL, and the translation scheme from commitment machine to BDI plans are both applicable to any BDI language.

As we have mentioned in section 1, there has been some work on *designing* flexible and robust agent interaction, but virtually no work on *implementing* flexible and robust interactions.

We have already discussed STAPLE [9, 10]. Another piece of work that is relevant is the work by Cheong and Winikoff on their *Hermes* methodology [2]. Although the main focus of their work is a pragmatic design methodology, they also provide guidelines for implementing Hermes designs using BDI platforms (specifically Jadex) [3]. However, since *Hermes* does not yield a design that is formal, it is only possible to generate skeleton code that then needs to be completed. Also, they do not address the turn taking issue: how to decide which agent acts when more than one agent is able to act.

[9]We also assume that the communication medium does not deliver messages out of order, which is the case for (e.g.) TCP.

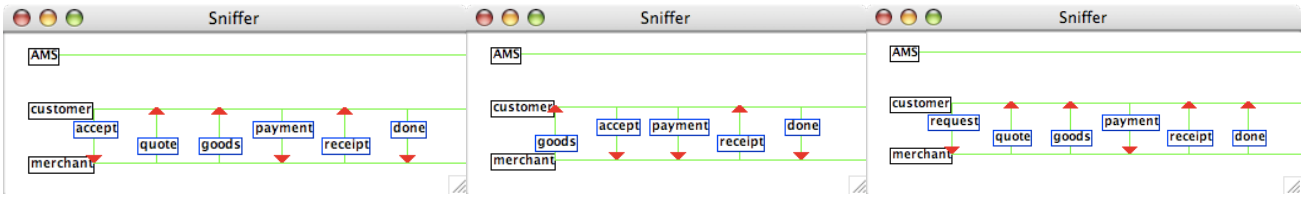

Figure 8: Sample runs from 3APL implementation (non-alternating turns)

The work of Kremer and Flores (e.g. [8]) also uses commitments, and deals with implementation. However, they provide infrastructure support (CASA) rather than a programming language, and do not appear to provide assistance to a programmer seeking to implement agents.

Although we have implemented the NetBill interaction using 3APL, the changes to the semantics were done by modifying our NetBill 3APL program, rather than by modifying the 3APL implementation itself. Clearly, it would be desirable to modify the semantics of 3APL (or of another language) directly, by changing the implementation. Also, although we have not done so, it should be clear that the translation from a CM to its implementation could easily be automated.

Another area for further work is to look at how the assumptions required to ensure that actions commute can be relaxed.

Finally, there is a need to perform empirical evaluation. There has already been some work on comparing *Hermes* with a conventional message-centric approach to designing interaction, and this has shown that using *Hermes* results in designs that are significantly more flexible and robust [4]. It would be interesting to compare commitment machines with *Hermes*, but, since commitment machines are a framework, not a design methodology, we need to compare *Hermes* with a *methodology* for designing interactions that results in commitment machines [13, 17].

6. REFERENCES

[1] R. H. Bordini, M. Dastani, J. Dix, and A. E. F. Seghrouchni, editors. *Multi-Agent Programming: Languages, Platforms and Applications*. Springer, 2005.

[2] C. Cheong and M. Winikoff. Hermes: Designing goal-oriented agent interactions. In *Proceedings of the 6th International Workshop on Agent-Oriented Software Engineering (AOSE-2005)*, July 2005.

[3] C. Cheong and M. Winikoff. Hermes: Implementing goal-oriented agent interactions. In *Proceedings of the Third international Workshop on Programming Multi-Agent Systems (ProMAS)*, July 2005.

[4] C. Cheong and M. Winikoff. Hermes versus prometheus: A comparative evaluation of two agent interaction design approaches. Submitted for publication, 2007.

[5] P. R. Cohen and H. J. Levesque. Teamwork. *Nous*, 25(4):487–512, 1991.

[6] M. Dastani, J. van der Ham, and F. Dignum. Communication for goal directed agents. In *Proceedings of the Agent Communication Languages and Conversation Policies Workshop*, 2002.

[7] F. P. Dignum and G. A. Vreeswijk. Towards a testbed for multi-party dialogues. In *Advances in Agent Communication*, pages 212–230. Springer, LNCS 2922, 2004.

[8] R. Kremer and R. Flores. Using a performative subsumption lattice to support commitment-based conversations. In F. Dignum, V. Dignum, S. Koenig, S. Kraus, M. P. Singh, and M. Wooldridge, editors, *Autonomous Agents and Multi-Agent Systems (AAMAS)*, pages 114–121. ACM Press, 2005.

[9] S. Kumar and P. R. Cohen. STAPLE: An agent programming language based on the joint intention theory. In *Proceedings of the Third International Joint Conference on Autonomous Agents & Multi-Agent Systems (AAMAS 2004)*, pages 1390–1391. ACM Press, July 2004.

[10] S. Kumar, M. J. Huber, and P. R. Cohen. Representing and executing protocols as joint actions. In *Proceedings of the First International Joint Conference on Autonomous Agents and Multi-Agent Systems*, pages 543 – 550, Bologna, Italy, 15 – 19 July 2002. ACM Press.

[11] M. Tambe and W. Zhang. Towards flexible teamwork in persistent teams: Extended report. *Journal of Autonomous Agents and Multi-agent Systems*, 2000. Special issue on "Best of ICMAS 98".

[12] M. Winikoff. An AgentSpeak meta-interpreter and its applications. In *Third International Workshop on Programming Multi-Agent Systems (ProMAS)*, pages 123–138. Springer, LNCS 3862 (post-proceedings, 2006), 2005.

[13] M. Winikoff. Designing commitment-based agent interactions. In *Proceedings of the 2006 IEEE/WIC/ACM International Conference on Intelligent Agent Technology (IAT-06)*, 2006.

[14] M. Winikoff. Implementing flexible and robust agent interactions using distributed commitment machines. *Multiagent and Grid Systems*, 2(4), 2006.

[15] M. Winikoff, W. Liu, and J. Harland. Enhancing commitment machines. In J. Leite, A. Omicini, P. Torroni, and P. Yolum, editors, *Declarative Agent Languages and Technologies II*, number 3476 in Lecture Notes in Artificial Intelligence (LNAI), pages 198–220. Springer, 2004.

[16] M. Winikoff, L. Padgham, J. Harland, and J. Thangarajah. Declarative & procedural goals in intelligent agent systems. In *Proceedings of the Eighth International Conference on Principles of Knowledge Representation and Reasoning (KR2002)*, Toulouse, France, 2002.

[17] P. Yolum. Towards design tools for protocol development. In F. Dignum, V. Dignum, S. Koenig, S. Kraus, M. P. Singh, and M. Wooldridge, editors, *Autonomous Agents and Multi-Agent Systems (AAMAS)*, pages 99–105. ACM Press, 2005.

[18] P. Yolum and M. P. Singh. Flexible protocol specification and execution: Applying event calculus planning using commitments. In *Proceedings of the 1st Joint Conference on Autonomous Agents and MultiAgent Systems (AAMAS)*, pages 527–534, 2002.

[19] P. Yolum and M. P. Singh. Reasoning about commitments in the event calculus: An approach for specifying and executing protocols. *Annals of Mathematics and Artificial Intelligence (AMAI)*, 2004.