Available online at www.sciencedirect.com
SCIENCE DIRECT®
The Journal of Systems and Software 76 (2005) 65–76
The Journal of Systems and Software
www.elsevier.com/locate/jss

Effects of introducing survival behaviours into automated negotiators specified in an environmental and behavioural framework

Peter Henderson, Stephen Crouch *, Robert John Walters, Qinglai Ni [1]

Declarative Systems and Software Engineering, School of Electronics and Computer Science, University of Southampton, Southampton SO17 1BJ, UK

Received 15 December 2003; received in revised form 15 April 2004; accepted 15 June 2004
Available online 26 August 2004

Abstract

With the rise of distributed e-commerce in recent years, demand for automated negotiation has increased. In turn, this has engendered a demand for ever more complex algorithms to conduct these negotiations. As the complexity of these algorithms increases, our ability to reason about and predict their behaviour in an ever larger and more diverse negotiation environment decreases. In addition, with the proliferation of internet-based negotiation, any algorithm also has to contend with potential reliability issues in the underlying message-passing infrastructure. These factors can create problems for building these algorithms, which need to incorporate methods for survival as well as negotiation.

This paper proposes a simple yet effective framework for integrating survivability into negotiators, so they are better able to withstand imperfections in their environment. An overview of this framework is given, with two examples of how negotiation behaviour can be specified within this framework. Results of an experiment which is based on these negotiation algorithms are provided. These results show how the stability of a negotiation community is affected by incorporating an example survival behaviour into negotiators operating in an environment developed to support this framework.

Keywords: E-commerce; Automated negotiation; Negotiation framework; Pseudocode

1. Introduction

1.1. Background

The choice of algorithm used to carry out automated negotiation on behalf of a client is a significant problem in distributed e-commerce (Bichler et al., 1998; Burg, 2002; Cranor and Resnick, 1997; Farhoodi and Fingar, 1997; Fingar et al., 2000; Henderson, 2002). Furthermore, predicting how well a given algorithm will perform in a given environment is difficult.

The ability of an algorithm to succeed in an automated negotiation environment is dependent on its ability to survive in that environment. Automated negotiators, on their own terms, must be able to make sense of and conduct negotiation on the web in which there are no guarantees of the reliability of the underlying message-passing infrastructure. As the web increases in size and interconnectivity, this will become an even greater problem. In some cases, offers sent may not be received at all, but equally problematic is that offers received are out of date. Suppose a negotiator receives an offer that has spent an inordinate amount of time in transit. Despite replying promptly in sending an accept to this offer, the negotiator finds their acceptance rejected because the offer's sender has already sold their

* Corresponding author. Tel.: +44 23 8059 7684; fax: +44 23 8059 3045.
E-mail addresses: p.henderson@ecs.soton.ac.uk (P. Henderson), s.crouch@ecs.soton.ac.uk (S. Crouch), r.j.walters@ecs.soton.ac.uk (R.J. Walters), q.ni@reading.ac.uk (Q. Ni).
[1] Present address: School of Plant Sciences, University of Reading, Reading, Berkshire RG6 6AS, UK.

0164-1212/$ - see front matter © 2004 Elsevier Inc. All rights reserved.
doi:10.1016/j.jss.2004.06.024

last stock to someone else. Agreement is not reached because of inconsistent views of negotiation, caused by inconsistent information. The ability to tolerate this information inconsistency, and being able to minimise its negative effects by taking corrective or compensating action, may reward the negotiator with greater success.

To compound these issues, a negotiator cannot be certain whether their experience of such problems with another negotiator is because of natural occurrence, or faulty or even malicious behaviour. It is also possible that the two negotiators are simply unable to reach agreement because they exhibit mutually incompatible negotiation strategies. To succeed, automated negotiators must be able to survive and progress despite such eventualities, without knowing the intent of other negotiators.

It is not uncommon for communities of automated negotiators to establish stable norms of behaviour. Over time, despite negotiators' different behavioural characteristics, initially erratic patterns of negotiation can eventually settle into predictable patterns of apparent co-operation (Fogel, 2000; Young, 1998). However, making successful predictions about how and in what form such stability will emerge can prove difficult (Axelrod, 1984, 1997). Even more difficult are attempts to predict how the community will react when potentially disruptive elements are introduced into the environment.

1.2. Previous work

Previously we have examined architectures for e-commerce systems (Henderson, 1998, 2002; Henderson and Walters, 2001), to investigate how federations of applications co-operate. We have also investigated the use of a fixed-length tournament-based approach to judge the fitness of negotiation algorithms against each other (Henderson et al., 2003). Certain patterns of negotiation were observed during the tournament between various algorithms, where certain pairs of algorithms did consistently better with each other than others. Examining their negotiation traces, we observed that stability would often emerge in their negotiations; their behaviour following a predictable path until negotiation was positively or negatively concluded. However, due to the nature of the experiment, negotiation between two participants did not affect other participants during a simulation. This meant that we were unable to investigate how stability would emerge at a *communal* level. Essentially, the environment was incapable of answering some interesting questions. Given a community of algorithms, would stability emerge? If so, in what form? Then, if a stability were to emerge in a community of negotiators, how would this stability be affected if we: introduce or extract an algorithm mid-experiment? Introduce an unreliable environment? Adapt algorithms to cope with this environment?

In this paper, we attempt to explore, and to some extent answer, the above questions. We extend the fixed-length tournament approach to encompass the concept of a continuously operational environment where negotiators may join and leave this community at any time. In our implementation of such an environment we are able to develop new algorithms using the framework described in this paper, then introduce, observe and evaluate them as they participate in negotiations with others. We are also able to introduce uncertainty into the message-passing infrastructure, and observe how this affects the participants. Of particular interest is how these changes affect the stability of negotiation communities.

2. Reactive and proactive negotiation

In essence, the negotiation process consists of a number of offers being exchanged between two participants until agreement is reached. This process consists of the following steps: wait until an offer is received, evaluate the new offer, then either reply with a counter-offer (going back to the first step), or send an acceptance or quit negotiations.

Notwithstanding the initial offer from either of the participants, this process is a reactive cycle. This process is depicted in Fig. 1. The dashed box at the top represents an initial action conducted by only one of the participants. This is the only proactive task in the process.

It is natural to assume that the structure of algorithms should follow this same rigid process. This idea is also easily extended to allow multiple negotiations with multiple participants.

In practice, adopting a purely reactive approach to the negotiation process is simply not sufficient. Developing negotiators in such a way does provide clarity of process, and simplicity of implementation. However, the success of the negotiator becomes ultimately dependent on the success of the negotiation process, which is itself dependent on the reliability of the operating environment. Notwithstanding 'bad' behaviour exhibited by negotiators, when this environment becomes unstable, the negotiation process is liable to collapse.

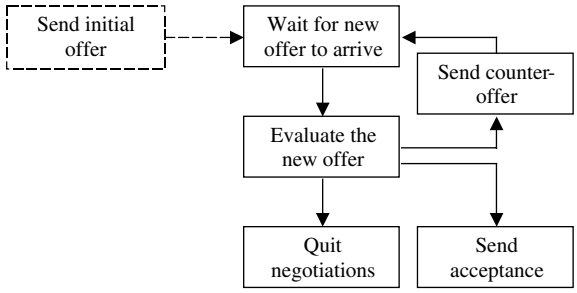

Fig. 1. Reactive negotiation process.

What is required is a more abstract, proactive approach to conducting negotiation (Murugesan, 2000). An approach that allows negotiators to reason about their circumstances at a higher level than the negotiation process alone, and adopt a more proactive view. Such a proactive approach should view negotiation as a fully manageable process: a means to achieve its objectives. In this way, a negotiator accepts more responsibility for its survival and success, and reduces dependence on a potentially imperfect operating environment. Ideally, we would like the clarity offered by a reactive approach, coupled with the managerial power offered by the proactive approach.

In the context of this paper, we define success as a measurement of how well a negotiator *performs*, and we define survival as a measurement of whether a negotiator is able to *progress* in its environment. As will be observed later, recognising this separation of task success and environmental survival can enable more robust negotiators to be built.

3. The simulator

3.1. The car hire scenario

The scenario adopted for the negotiation simulation was car hire. If we consider a single participant in this environment, their objective is to secure a set number of 'hires' per day with respect to a given set of car specifications. For Buyers, we use a 'hire' unit to represent the number of cars they are in possession of for a given day. For Sellers, this unit represents the number of cars that are hired out for a given day.

The participant has their own set of example deals they would ideally like to achieve. Each entry in this set consists of two attribute name and value pairs for the following attributes:

- **Days** the length of time we wish to hire the car.
- **Price** the price we would like to pay.

A set of examples consists of a number of these pairings, each representing an acceptable outcome of negotiation. In practice, an instance of a Buyer participant within this framework uses their examples as deal targets for acquiring a number of hire cars, whilst an instance of a Seller participant views their examples as deal targets for hiring out available stock. Since negotiation is based on two attributes, negotiators are potentially able to compromise and thus avoid a myopic focus on "best price" (Youll, 2001). A more realistic model of car hire would incorporate more attributes, (e.g. car size, car features, etc.), introducing more opportunities for compromise, however the main objective of the experimentation is to observe overall communal behaviours, and so we have kept the model simple.

Specifying negotiation criteria as examples provides an abstract yet flexible method of stating a negotiator's desires, although the potential exists for ambiguity between these example criteria. There is not always a clear correlation between these examples, and the process of interpreting these examples in the context of the negotiation process is a task for the negotiator (Sesseler, 2001).

Consider the example set in Table 1. If we assume they are a set of Buyer examples, we can easily determine that they would be willing to pay 250 for 9 days of car hire, but also would pay significantly less on a cost per day basis for 6 days. In reality, such a discrepancy is often reasonable. It may be that the creator of this example set unavoidably requires a car for 9 days, so therefore ideally wants 9 days of car hire. If this were unavailable, the Buyer would be willing to accept 6 days of hire, but for a lot less per day to compensate for the extra effort of having to acquire 3 days of car hire after 6 days. If the Buyer receives an offer close to one of these examples, they would be inclined to accept it. If they have to make a counter-offer, the method they use takes into account their examples and offers received. If, for example, a Buyer negotiator who requires 2 cars per day (its quota) were able to reach agreement for this quantity with a Seller for 9 days at 250, as in the set of examples, they would not need to negotiate again for another 9 days. If they only succeeded in obtaining 1 car for the same deal, they would need to attempt to find another deal somewhere else for the remaining car. The possibility exists that they will only be able to achieve their hires per day objective in part, or perhaps not at all.

3.2. The negotiation environment

An environment was developed which enables automated Buyers and Sellers to participate in the described scenario. This environment is similar in concept to a market run over an indefinite number of days; where Buyers and Sellers enter and leave at will on a daily basis. There are no restrictions on how many days they are able to participate, or how and with whom they conduct negotiations, although Buyers only negotiate with Sellers, and vice versa. Since there is no fixed duration to the simulation, negotiators cannot take advantage of other negotiators by exploiting the length of the simulation (Binmore and Vulkan, 1999).

Table 1
A typical example set

Days	Price
9	250
6	100

Participants are able to negotiate with anyone at any time. Their algorithm determines the manner in which they conduct negotiations with others to achieve their objectives. This allows us to construct and observe behaviour-rich simulations.

The environment consists of two components:

- **Supervisor** initiates, maintains and controls the environment, including the negotiators. Also maintains measures of negotiator performance.
- **Negotiator** given a set of negotiation parameters (including a set of examples and a target for the number of cars to possess/hire out each day), and is responsible for conducting negotiation.

To initiate a new environment, the Supervisor is launched, which then enables negotiators to be configured and instantiated, so they may participate in the simulation. Performance is measured by two factors: average hires per day, and average money spent/accrued per day. Each average calculation is based over the over last 6 days. This effectively gives us a running indicator of success (money per day) and survival (hires per day) as the simulation progresses. In short, if a negotiator is managing a high number of hires per day, it is surviving. If it manages a high amount of money for a seller—or a low amount for a buyer—per day, it is succeeding. Success and survivability are not only dependent on the reliability of communications, but on the structure of the community itself. If there is a shortage of car hire for sale, Buyers will do badly. If there is a shortage of demand, Sellers will do badly. In addition, in most cases the further apart the Buyer and Seller example sets are, the lower the likelihood of many deals being reached.

Fig. 2 details the operation and message flow within the negotiation environment.

A message (e.g. offer) sent from one negotiator to another is stored in the receiving negotiators first-in-first-out 'message inbox', and it is each negotiator's responsibility to service their inbox and process its contents. Within this negotiation model, either negotiator (Buyer or Seller) may send an initial offer to the other. An arbitrary number of counter-offers are then subsequently made until one sends a request to accept their partner's last sent offer (*accept*). With each message type, a quantity is attached. For a Buyer, this represents the quantity of cars they want for that deal. For a Seller, this represents the quantity of cars they wish to hire out.

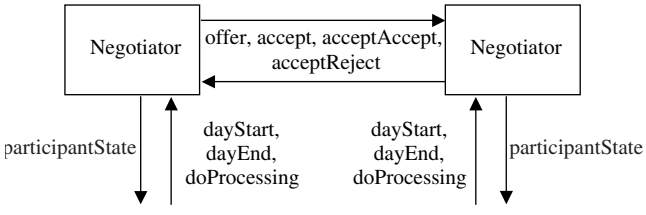

Fig. 2. Operation and message flow within the negotiation environment.

In such an asynchronous system, offers may become out of date; the negotiator may no longer be able to supply the quantity requested. Therefore, on receipt of an *accept*, an acknowledgement is required. This acknowledgement is either an *acceptAccept*, which is confirmation that the deal is accepted, or *acceptReject*, which is rejection. During this handshake, the quantity of resources stated in an *accept* are locked until either confirmation is received. This ensures that only one party accepts these resources. The handshake is also designed to protect against *accept*-ing an offer for a quantity greater than that which is in stock by the supplying party (Seller), or required by the requesting party (Buyer). Essentially, if either is true, the *accept* still goes ahead, but for a maximum quantity that satisfies both Buyer and Seller requirements. For example, if a Buyer requested 6 hires for a day but whose quota was 4, and the Seller was able to supply 4, the quantity would be 4. If the Seller could only manage 2, the quantity would be 2. If the Buyer's quota was 2, but requested 6, and the Seller could supply 4, the quantity would be 2, and so on and so forth. The reason for this relaxed attitude when dealing with offer quantities is explained in Section 4.3.

How the Supervisor and Negotiator fit into this framework is examined in more detail in the following section.

4. Specification of the framework

Each specification follows an event-driven paradigm, described using an abstract pseudocode. The design of the pseudocode was important; it needed to be powerful and descriptive yet not too strict or laborious, and various languages with different descriptive styles and levels of abstraction were devised before one was selected. It is from the pseudocode specifications of each algorithm that implementations of each behaviour can be derived mechanistically and inserted into the simulator for experimentation. In the next three sections we will discuss the specification of each part of the above framework, and how they integrate together.

4.1. Supervisor

The Supervisor coordinates the environment according to the behaviour shown in Fig. 3.

Here we can observe how the Supervisor manages the negotiators. Essentially, the simulation runs forever, and for each hour of each day, all the negotiators are requested to do a single 'chunk' of processing (*doProcessing()*). *dayStart()* and *dayEnd()* are called on each

```
on startSimulation {
   do forever {
      for each participant in Participants {
         participant.dayStart()
      }
      for hours = 1 to 24 {
         randomize order of Participants
         for each participant in Participants {
            participant.doProcessing()
         }
      }
      for each participant in Participants {
         participant.dayEnd()
} } }
```

Fig. 3. Specification of the Supervisor.

negotiator at the beginning and end of each day for performance measurement and maintenance purposes.

The *randomize order of Participants* statement introduces an element of fairness into the simulation. Without this statement, each *participant* in the list of *Participants* would always be called in the same order; the simulation would perhaps favour the first negotiator, since they have a greater chance of securing the first deal.

4.2. Negotiator

A negotiator developed in the framework requires two distinct areas of its behaviour to be specified: reactive and proactive. The reactive aspect of its behaviour handles the negotiation process. The proactive aspect of the negotiator's behaviour handles survival issues, assessing its situation and instigating corrective actions based on negative aspects of this assessment. Both these aspects are shown in Fig. 4. Algorithmic behaviour, covered in the next section, is inserted into this skeleton framework.

The conditionally proactive behaviour in *algorithmProcess( )* is specified as rules, in the form of *condition → action* pairs. As will be demonstrated later with an example, we are able to specify survival behaviours within this function.

When the Supervisor invokes a negotiator's *doProcessing( )* instruction, the framework performs two actions transparent to the algorithm's creator:

- **Reactive—Service message box** new messages are read and the negotiator's appropriate *on receive* functions are invoked depending on the message type of each.
- **Proactive—Invoke algorithm's proactive function** (***algorithmProcess***) this allows the negotiator the opportunity to evaluate their situation, and possibly take proactive action.

```
on receive offer from participant {
   // What to do when an offer is received
}

on receive accept from participant {
   // Agree to an acceptance proposal?
   // (i.e. the other participant wishes to
   // accept your last offer)
   // Returns either True or False
}

on receive acceptAccept from participant {
   // What to do when an acceptance to a
   // previously sent accept is received. Here,
   // negotiation is positively concluded with
   // 'participant'
}

on receive acceptReject from participant {
   // What to do when a reject to a previously
   // sent accept is received. Here, negotiation
   // is negatively concluded with 'participant'
}

on algorithmProcess {
   // proactive behaviour is specified here
}
```

Fig. 4. Outline specification of reactive and proactive behaviour.

Thus, in this framework, there is a distinct separation between the reactive and proactive parts of the negotiation process. Suppose a negotiator initiates negotiation with another negotiator by sending them an initial offer. From this point, the negotiation process is dealt with by the reactive functions; multiple negotiations with multiple partners are handled automatically. However, the algorithm is still able to take proactive action if required, enabling the negotiator to monitor and *manage* these negotiation processes at a more abstract level.

Each negotiator is responsible for a number of negotiation processes; a maximum of one per partner at any given time. As a result, the possibility exists that negotiations with one party will be abandoned in favour of accepting another deal from another party. However, this does not prevent the original two negotiators from resuming negotiations (from where they were abandoned) at a later date, if both are prepared to do so. The algorithms present in this experiment do exhibit this forgiving behaviour, not forcing negotiations with the one who abandoned the negotiation to begin at the start

of their behavioural process. As will be observed in the next section, this leads to some interesting behaviour.

4.3. Algorithm

For this experiment, we initially wished to examine the behaviour of negotiation algorithms based on the reactive negotiation process given in Section 2. Therefore, the behavioural structure of the algorithms used in the experiment was based around the same core negotiation process specification. Before the core process is presented, we need to state a few points about the pseudocode:

- The pseudocode provides some basic means of determining various parameters and state for a negotiator. *Examples()* is an ordered list of (*day, price*) tuples of negotiator examples given to the negotiator upon creation. *opponentList* is a list of negotiators of the opposing type (Buyer or Seller instances). In addition, access to a history is provided which logs the history of actions in a set of negotiations with each participant. This essentially forms a negotiation 'memory' for the negotiator.
- Other than the negotiation histories, the pseudocode does not dictate how local state is to be stored or retrieved. Negotiators may wish to store and retrieve certain data as negotiations progress, to aid later in decisions in the negotiation process. However, providing a strict means of specifying this tended to distract the reader from clearly understanding the core behaviour of the negotiator, which the pseudocode is primarily designed to convey.
- Negotiators may send offers to their own message boxes. This enables negotiators to postpone dealing with an offer they are not interested in at the time of first reading until the next round. This allows negotiators to resume negotiations at a later date, essentially 'pausing' negotiation with the other party.

The core negotiation process is specified in Fig. 5.

This constitutes the common negotiation process of each of the two algorithms. Appended to this are the functions *makeInitialOffer()* and *makeOfferDecision()*, which together form the different behaviour for each of the two algorithms and is defined separately. Within these functions, algorithms conditionally elect to send offers to participants using the *send* directive, which is an intrinsic part of the framework (see Section 5.2 for examples of how this is employed). The format for *send* is simple:

```
send <m> to <r> with quantity <q>
```

Where *m* represents the type of message, *r* represents the unique identity of the intended recipient, and *q* the quantity desired. If we wished to define/alter properties of a message, we can directly access *m*. For example, if we wished to send a reply to an *offer* we had received, which was based on that offer but with a 10% reduction on price, we could say:

```
on receive offer from somebody {
  offer.price = offer.price * 0.9
  send offer to somebody with quantity 1
}
```

It should be clear that the pseudocode is very loosely typed. However, although we are using *offer* to represent a type and an object, the intended behaviour should be obvious. It should be emphasised that this is not an implementation language, nor is it a formal specification language. Utilising a well-defined type system would add further complexity. It is the essence of behaviour we wish to convey, not implementation. With judicious use of appropriate programming constructs, and maintaining a consistent style, ambiguity is avoided.

```
on receive offer from participant {
   if (haven't filled our quota for today) {
      makeOfferDecision(offer, participant)
   } else {
      put offer back in MessageBox for next round
} }

on receive accept from participant {
   clear sent offer history with participant
   return True
}

on receive acceptAccept from participant {
   clear sent offer history with participant
}

on receive acceptReject from participant {
   clear sent offer history with participant
}

on algorithmProcess {
   if (we haven't filled our quota for today) {
      for each participant in opponentList {
         if (offers made to participant == 0)
            makeInitialOffer(participant)
} } }
```

Fig. 5. Specification of the core negotiation process in abstract pseudocode.

To react to receiving message type *m*, an entry for the event is defined in the receiver thus:

```
on receive <m> from <p> { ... }
```

Of course, in this framework the negotiation message 'policy' is fixed to a set series of message types as depicted in Section 4.2, so effectively only the *offer* and *accept* types can be 'sent' from a negotiator. The handshake process (the sending of *acceptAccept* or *acceptReject* in response to *accept*) is handled by the framework, but the algorithm is free to respond to these events. In the case of the *accept* receive event, the algorithm can decide whether to accept an *accept* (see Section 4.2), although in this core negotiation process, assuming no rules are broken, *accept*s are accepted automatically.

Various safeguards were built into the framework to ensure that algorithms could not 'cheat' e.g. renege on deals, change offer details when accepting, etc. However, as discussed in Section 3.2, a relaxed attitude was employed when dealing with quantities. This is not necessarily unreasonable. For example, if a Buyer required 4 cars for one day, he may send many initial offers to many Sellers. If, however, following negotiations the Seller offering the best deal could only offer 3 cars, (perhaps a number of cars were hired out to others during the negotiation process), renegotiation on quantity would be required. As discussed, the framework deals with this discrepancy; the quantity agreed being the highest common denominator of supply/demand between the Seller and Buyer. This means specifications do not have to handle this detail which detracts from the clarity of the specified behaviour. If this were not the case, the framework and specifications would be far more complex.

Upon completion of negotiation with a participant, our specification of the core negotiation process clears the offer history with that participant. Therefore, the offer history is used as a means of holding state for current negotiations only. This reflects the simplistic nature of the reactive negotiation process we wish to model. Both algorithms defined within this framework work on a per-negotiation basis; they do not attempt to take into account previous dealings with a participant when negotiating. In addition, more complex means of dealing with and analysing histories is possible, but the algorithms defined in the next section were not designed to be realistic; rather, they were designed to be straightforward and simple so that we could readily reason about their behaviour in a complex interacting environment. Therefore, from the behavioural characteristics of *autonomy*, *cooperation* and *learning* (Murugesan, 2000), the algorithms presented here possess autonomy and cooperation. They are able to operate independently from human intervention during negotiation, and are able to cooperate with other negotiators to achieve their goals. However, they do not actively learn from negotiation to negotiation and subsequently adapt their strategy.

Notice that *algorithmProcess()* is used to perform the only proactive task in the specification: *makeInitialOffer()*, used to initiate negotiations with another. It is performed when a negotiator has not filled their quota for the day. For every participant with whom we are not negotiating, we send an initial offer to initiate negotiations with that party.

5. The experiment

5.1. Overview

This paper will detail one experiment, which consists of four simulations. Other experiments were conducted which used different algorithms and different example sets. The results presented here are representative of these other experiments.

The first simulation establishes how a community of negotiators develops in a reliable message-passing environment. The negotiators are nave in that they adopt a purely reactive approach to negotiation, as discussed.

In the second simulation, it is established how the same community of negotiators develops with an element of uncertainty introduced into the message-passing environment. In this simulation, there is a 10% chance that a sent offer will not reach its destination. This random unreliability only has an effect at the message communication level. Connections between negotiators are unaffected.

For the third simulation, it will be shown how each algorithm can be adapted to incorporate an example proactive survival behaviour, and the simulation executed again with reliable communications. This provides us with an opportunity to observe how these adapted algorithms behave in a reliable environment.

The fourth simulation illustrates how these adapted algorithms behave in the unreliable communications environment.

After each day, the performance of each negotiator is evaluated, with respect to average hires per day and average money accrued (Seller) or spent (Buyer) per day. Thus, we have measures of survival and success respectively. This paper focuses on the results for average hires per day, since we wish to examine the survivability of the community. By examining how the number of hires per day for each participant changes over time, we are able to reason about the survivability of the negotiation processes conducted by the negotiators, and ultimately, the survivability of the negotiation community.

5.2. The algorithms

In each simulation, instances of two algorithms form the negotiation community. These algorithms were not designed to be realistic negotiators. Rather, their strategies are designed to be sufficiently simple that we are able to reason about their behaviour in a complex, evolving community. Each of the specifications of the following algorithms integrates with the negotiation process specification given in Section 4.3.

5.2.1. Stubborn

This algorithm exhibits 'stubborn', anti-concessionary behaviour by sending only its examples as offers to a negotiation partner. It does not attempt to reason about the offers received. Initially, it sends its first example, then its second, etc. When it has sent all its examples, it starts again. After 30 rounds of negotiation, it simply accepts the last received offer from its partner.

Fig. 6 shows the behaviour of this algorithm. Whilst the first two functions integrate with the negotiator framework, the third function provides an algorithm-specific means of evaluating an offer for acceptance.

From the specification above, the algorithm's simple behaviour is obvious. In the experiment, the Stubborn algorithms are only given a target of 1 hire per day, so for clarity the pseudocode above dictates this constant instead of determining it.

5.2.2. Experimental

This is a far more reactive and concessionary algorithm than stubborn. Its first example forms its initial

```
function makeInitialOffer(participant) {
   offer = first example
   send offer to participant with quantity 1
}

function makeOfferDecision(offer, participant) {
   if (evaluateOffer(participant)) {
      send accept to participant with quantity 1
   } else {
      num = offers made to participant + 1
      if (num > #Examples()) then num = 1
      offer = Examples(num)
      send offer to participant with quantity 1
} }

function evaluateOffer(participant) {
   if (offers made to participant > 30)
      return True else return False
}
```

Fig. 6. Specification of the Stubborn algorithm.

```
function makeInitialOffer(participant) {
   offer = first example
   r = daily quota requirement - hires made today
   send offer to participant with quantity r
}
function makeOfferDecision(offer, participant) {
   if (offers made to participant != 0) {
      lso = last sent offer to participant
      compareSet() = ( lso )
      concession = concession made in lso
      exNum = example we used as basis for lso
   } else {
      compareSet() = ( Examples() )
      concession = 0
      exNum = 1
   }
   if (evaluateOffer(offer, compareSet())) {
      r = daily quota requirement -
         hires made today
      send accept to participant with quantity r
   } else {
      concession = concession + 2
      if (concession > 20) {
         concession = 0
         exNum = exNum + 1
         if (exNum > #Examples()) exNum = 1
      } elseif (offer.days !=
         Examples(exNum).days) {
         concession = 0
         if (exists x where offer.days ==
            Examples(x).days)
            exNum = x else exNum = 1
      }
      offer = Examples(exNum)
      offer.price = concede concession
         on offer.price
      r = daily quota requirement - hires made
         today
      send offer to participant with quantity r
} }
function evaluateOffer(offer, compareSet()) {
   for each cOffer in compareSet {
      if (cOffer.days = offer.days &&
         cOffer.price is close to offer.price) {
         return True
   } }
   // Otherwise…
   return False
}
```

Fig. 7. Specification of the Experimental algorithm.

offer. When a new offer is received, it attempts to find an example that matches the number of days in the received offer. If found, it sends a sequence of offers for this example, each a little more concessionary on price. If not found, it states the first example. Concessions are made progressively only as long as negotiations continue on the same number of days. Otherwise, the concession process begins again. When it reaches a $20 concession on price for the selected example, it attempts to move negotiation cyclically to its next example by specifying it as the next offer. The specification of this algorithm is given in Fig. 7.

This algorithm requires a more sophisticated offer evaluation function. It compares a received offer with either the last offer that was sent to the other participant, or the contents of the example set. Therefore, this function takes an offer and compares it against a set of offers. If the received offer is deemed close enough to any offer in the set, the function returns *True*, and the algorithm accepts the received offer.

5.3. The negotiation community

Initially, a simulation begins with 2 'Stubborn' buyers and 2 'Stubborn' sellers conducting negotiations. After 20 days of negotiation, an 'Experimental' buyer and 'Experimental' seller are introduced to observe how this affects the community.

5.4. Negotiator example sets and objectives

Each algorithm has different Buyer and Seller example sets. However, for each algorithm, every Buyer is given the same set of Buyer examples, and every Seller is given the same set of Seller examples. The Stubborn Buyer and Seller example sets are given in Tables 2 and 3 respectively.

The Buyer and Seller example sets for the Experimental algorithms are given in Tables 4 and 5.

The Stubborn and Experimental example sets are deliberately designed to be 'close' together, to encourage negotiations between the two behaviours. In the case of the Experimentals, if the Seller and Buyer examples were sufficiently far apart, they would never reach agreement, since their mutual concessions would not go far enough to appease either one of them.

The objective of each Stubborn Buyer and Seller is to make one deal per day, whilst the objective of the Experimental Buyer and Sellers is to make four hires per day. However, this is very difficult for the Stubborn negotiators to achieve, due to the length of their negotiation process. It is not impossible, however, as will be observed in the results in the next section.

Table 2
Stubborn Buyer example set

Days	Price
2	160
4	240

Table 3
Stubborn Seller example set

Days	Price
1	100
2	180
3	200

Table 4
Experimental Buyer example set

Days	Price
2	140
4	320

Table 5
Experimental Seller example set

Days	Price
1	100
2	180
3	200
4	300
5	340
6	360
7	400

6. Results

Fig. 8 shows the results of simulation 1, representing average hires per day for each negotiator (y-axis) over a 40 day period (x-axis). The four lines clustered at the bottom represent the four Stubborn negotiators, whilst the two at the top represent the two Experimental negotiators introduced after 20 days.

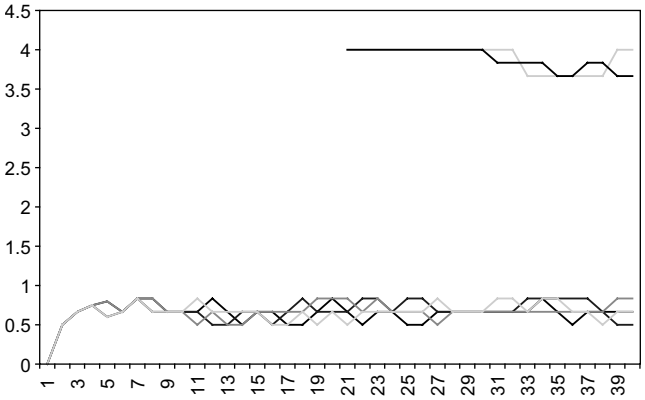

Fig. 8. Simulation 1: average hires per day.

Since Stubborn always takes 30 rounds (hours) to reach a deal, we observe that there are no deals struck on the first day. After about 4 days, in which all negotiators perform equally well, these negotiators begin to exhibit a certain communal behaviour. Note that no Stubborn reaches their maximum hires per day. This is due to the inflexible, laborious nature of the Stubborn algorithm.

One might expect a strict pattern to be observed. However, there are two reasons why this is not the case. Firstly, the environment has been designed with fairness in mind. As mentioned in Section 4.1 the order in which negotiators perform processing is random. Secondly, participants are able to abandon negotiations with a particular party, and resume them at a later date (see Section 4.2). Together, these factors decrease the likelihood that such behavioural harmonics will occur.

Also interesting are the results of the two Experimental algorithms. Following their introduction after 20 days, they remain very stable in their behaviour, in fact achieving their maximum required hires per day for nearly 10 days. After this however, a significant slump occurs. This is because initially, they deal only with each other; their negotiation process is far more efficient than Stubborn's, and they reach agreement quickly. Following this, however, one of them chances to strike a deal with one of the Stubborn negotiators, and this causes the Experimentals to be affected by their erratic negotiation strategy. The reason that one of the Experimental negotiators makes this choice, despite it resulting in decreased hires per day, is that they are not able to anticipate that this will have a negative impact later. They make a choice that appears optimal at the time, but have no way of knowing that it represents a poor global choice (Schelling, 1978).

Fig. 9 illustrates the effects of introducing a 10% chance of an offer being lost. The results are as expected. The Stubborn negotiators are unable to reach agreement at all and do not even appear in the figure. After the two

```
on algorithmProcess {
   ...
   for each participant in Participants {
      if (have sent offer to Participant &
        have not had reply in last 3 hours)     {
         send last sent offer to participant
      }
   }
   ...
}
```

Fig. 10. Specification of the timeout proactive survival behaviour.

Experimentals are introduced, they do better due to their more efficient negotiation process. They reach agreement quickly (with each other), but following this, they are unable to maintain their initial success. Their negotiation processes also become affected by their inability to reason about their failure and take corrective action.

In order to conduct the third simulation, a survival behaviour had to be integrated into both algorithms. The example survival behaviour chosen was a timeout. The behaviour was specified as shown in Fig. 10.

Essentially, if we have not received a response to an offer sent in the last 3 h, just resend the offer. The results of simulation 3 are given in Fig. 11.

The results were not as expected. Introducing this survival behaviour into this system has maximised the efficiency of the community; they are achieving greater hires per day on average. Even more surprising is the stability that has clearly emerged among the Stubborn negotiators. These effects are being observed because the survival behaviour has affected the negotiation process of the algorithms. This behaviour essentially takes action in response to unresponsive negotiators. So, even under normal circumstances, it is still likely this behaviour will be instigated. Effectively, a negotiator now

Fig. 9. Simulation 2: average hires per day.

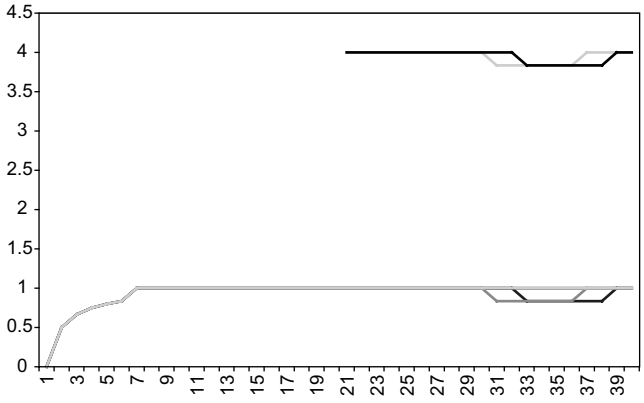

Fig. 11. Simulation 3: average hires per day.

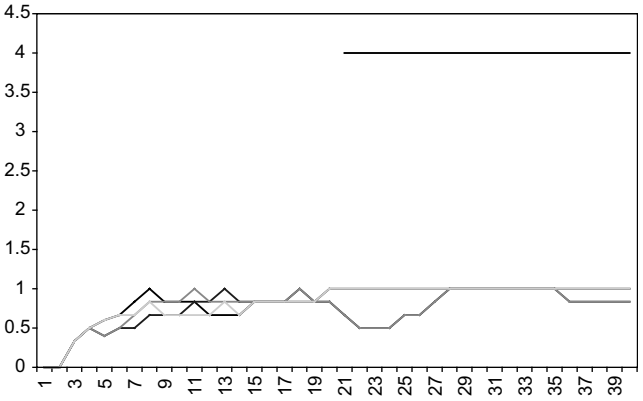

Fig. 12. Simulation 4: average hires per day.

responds to abandonment from another negotiator by resending their last offer.

As with the first simulation, both algorithm groups initially reach agreement only with those of their own type, but are eventually tempted by negotiations with the other group. Here we clearly observe that this behaviour affects the stability of the community as a whole.

Fig. 12 displays the results of simulation 4. Surprisingly, introducing a 10% message loss to the more robust community of negotiators has not dramatically affected the survival of their community. In fact, despite an initial inability to attain the stability noticed in simulation 3, the community eventually achieves a similar degree of survival. More significantly, this community achieves a greater degree of survival than observed in the first simulation. The result that the experimental algorithms always achieve their objectives is anomalous in this case, and not representative.

Why is it that the negotiators in *both* simulations 3 and 4 survive so much better? The reason is because we have introduced *chaos* and *opportunity* into the community. By introducing the survival behaviours into these algorithms, we have improved their ability to survive in an unreliable environment. But, as a side-effect we have also made their individual negotiation processes more agile; they no longer follow a rigid reactive path to a negotiation's conclusion. But, as a result, the negotiation process that occurs between two negotiators is less predictable. If we extend this perspective and observe the actions of the community as a whole, it has become more chaotic. However, this chaos has introduced a very welcome factor: opportunity. Negotiators are making more offers to their partners. This increases the chances that their quota is achieved.

The unreliable environment depicted in simulations 3 and 4 represents an unfortunate reality in which negotiators are often expected to survive and progress. Or at the very least, an unreliable environment potentially exists. When this happens, it would be advantageous for our negotiators to be able to survive and progress despite this. If we examine the results of the second simulation, the lack of progress in a negotiation can be characterised by the views of the negotiation process of each party becoming, and then continually remaining, inconsistent. Of course, since we have adopted an asynchronous model of communication, each party's view of negotiation will often be inconsistent because of offers in transit. But, with reliable communication, this inconsistency is eventually resolved and negotiation continues. However, in an unreliable environment, where an offer is made by one party which is lost, this results in both parties *always* believing they are waiting for the other to make an offer, which cannot be true. This inconsistency is an unfortunate state caused by the unfortunate reality of the unreliable environment. These negotiators have no control over this environment, just as software working over the internet has no control over the internet. Inconsistency and unreliability are simply inherent properties of such an environment. Endowing software with additional survival behaviours to help resolve inconsistencies is important, but realising the full consequences of such changes is equally as important.

7. Conclusion

Developing robust automated negotiators able to survive and succeed in the complex, evolving environment of the internet will become increasingly difficult. The ability to evaluate progress and proactively take corrective or compensatory action outside of the rigid negotiation process confers a greater degree of survival.

This paper has described a three-layer negotiation framework and specified this using a behavioural pseudocode developed for the purpose. Within this environmental framework an algorithm's behaviour may be specified, and then its operation in the context of other algorithmic behaviours observed. This algorithmic side of the framework enables the developer to specify reactive and proactive behaviour separately. Developers are able to specify conditional behaviour at a higher level than the negotiation process that is able to manage and optimise this process. By integrating an example survival behaviour into the algorithms we have demonstrated that not only has this improved their ability to take proactive action in cases of suspected message loss, but as a side-effect have made their negotiation processes more agile. By responding to ineffectual negotiations at this abstract level with our example survival behaviour, we have created more opportunities for success. Negotiators are no longer strictly adhering to the rigidity of the reactive negotiation process; they proactively increase their potential to make more deals by maximising efficiency within this process.

With the need to increase the robustness of negotiators in an ever more complex environment, we need to

be able to predict how this robustness will affect their behaviour with others. When implementing self-protection measures to ensure survival, we need to know that we are not hindering the negotiator's ability to succeed.

References

Axelrod, R., 1984. The Evolution of Co-operation. Basic Books Inc., New York.

Axelrod, R., 1997. The Complexity of Cooperation. Basic Books Inc., New York.

Burg, B., 2002. In: Agents in the World of Active Web ServicesLecture Notes in Computer Science, vol. 2362. Springer-Verlag, pp. 343–356.

Bichler, M. et al., 1998. Component-based e-commerce: assessment of current practices and future directions. ACM Sigmod Record: Special Section on Electronics Commerce 27 (4), 7–14.

Binmore, K., Vulkan, N., 1999. Applying game theory to automated negotiation, Netonomics, January. Available from <http://www.worcester.ox.ac.uk/fellows/vulkan>.

Cranor, L.F., Resnick, P., 1997. Protocols for automated negotiations with buyer anonymity and seller reputations. In: Proceedings of the Telecommunications Policy Research Conference (TPRC 97). Available from <http://www.si.umich.edu/~presnick>.

Farhoodi, F., Fingar, P., 1997. Developing enterprise systems with intelligent agent technology, distributed object computing. Object Management Group.

Fingar, P. et al., 2000. Enterprise E-Commerce, first ed. Meghan-Kiffer Press, Tampa, FL.

Fogel, D.B., 2000. Applying Fogel and Burgin's 'Competitive Goal-Seeking through Evolutionary Programming' to Coordination, Trust and Bargaining Games. In: Proceedings of the 2000 Congress on Evolutionary Computation (CEC 2000). IEEE Press, Piscataway, NJ, pp. 1210–1216.

Henderson, P. et al., 2003. In: A Comparison of Some Negotiation Algorithms Using a Tournament-Based ApproachLecture Notes for Computer Science, vol. 2592. Springer-Verlag, Berlin, pp. 137–150.

Henderson, P., 1998. Laws for dynamic systems. In: Proceedings of the Fifth International Conference on Software Reuse (ICSR 98). IEEE Computer Society Press, pp. 330–336. Available from <http://www.ecs.soton.ac.uk/~ph/papers>.

Henderson, P., Walters, R.J., 2001. Behavioural analysis of component-based systems. Information and Software Technology 43 (3), 161–169.

Henderson, P., 2002. Asset mapping—developing inter-enterprise solutions from legacy components. In: Systems Engineering for Business Process Change—New Directions. Springer-Verlag, UK, pp. 1–12.

Murugesan, S., 2000. Negotiation by software agents in electronic marketplace. In: Proceedings of the IEEE Region 10 Conference (TENCON 2000), vol. 2, pp. 286–290.

Schelling, T.C., 1978. Micromotives and Macrobehaviour. W.W. Norton and Company, Inc., New York.

Sesseler, R., 2001. Building agents for service provisioning out of components. In: Proceedings of the Fifth International Conference on Autonomous Agents, Montreal, Quebec, Canada, pp. 218–219.

Young, H.P., 1998. Individual Strategy and Social Structure: An Evolutionary Theory of Institutions. Princeton University Press, Princeton, NJ.

Youll, J., 2001. Agent-based electronic commerce: opportunities and challenges. In: Proceedings of the 5th International Symposium on Autonomous Decentralized Systems (ISADS 2001), pp. 146–148.

Peter Henderson is Professor of Computer Science in the School of Electronics and Computer Science at the University of Southampton in the UK. Prior to his move to Southampton in 1987 he was Professor at the University of Stirling. Henderson is also an ICL Fellow. He is head of the Declarative Systems and Software Engineering group (see http://www.dsse.ecs.soton.ac.uk/) which combines research interests in Software Engineering, Formal Methods and Programming Languages. His own research includes executable specifications, component-based systems and process modelling.

Stephen Crouch is a graduate in Computer Science of the University of Southampton and received his Ph.D. in computer science in 2001. He is a research fellow at the School of Electronics and Computer Science at the University of Southampton in the UK. His research interests include information propagation, enterprise systems, middleware architectures, negotiation, process and information system modelling, and component based software engineering.

Robert John Walters is a graduate in Mathematics with Computer Science of the University of Southampton and received his Ph.D. in computer science in 2003. He is a research fellow at the School of Electronics and Computer Science at the University of Southampton in the UK. His research interests include optimistic inconsistency solving techniques, enterprise systems, middleware architectures, graphical representation of formal languages, process modelling, model-checking, and component based software engineering.

Qinglai Ni is a graduate in Computer Science from Nanjing University, P.R. China, and passed his Ph.D. in civil engineering in 2003 at the University of Southampton. He is currently working as a middleware/database developer at the University of Reading in the UK. His research interests include middleware architectures, object-oriented data representation and serialization, robustness of web applications, and numerical simulation of soil shear behaviours.