Contract Driven Agents

John Knottenbelt and **Keith Clark**
Dept of Computing, Imperial College London
180 Queens Gate, London, SW7 2AZ, UK
{jak97,klc}@imperial.ac.uk

Abstract

We propose, as part of an agent architecture, a system where contracts are represented as first-class entities to allow the relationships between agents to be specified and developed separately from their built-in capabilities. The contracts allow the agent's problem-solving ability to be augmented by the potential of outsourcing tasks and obligations to other agents. The contracts also allow the agent's response to requests from other agents to be formally defined. The contracts are represented as event calculus rules which dictate the evolution of obligations and permissions on the agents concerned. The event calculus is used *deductively* to infer current and past obligations and permissions, and *abductively* to make plans to fulfil obligations subject to any restrictions imposed by the contracts.

Introduction

Multi-agent systems is a growing research area and has already started to find application in industry in web services and the semantic web. There is also increased interest in agent coordination and choreography. Our approach sees contracts as a means of formally describing the relationships between agents in terms of obligations and permissions, as well as providing a coordination function. By allowing contracts to be reasoned about in the agent cycle, the agent gains a precise understanding of what obligations it bears and which agents are obliged to it. Furthermore enough information is encoded in the contracts to allow the agent to outsource its own goals and obligations to other agents.

The core of the contract representation language is the event calculus (Kowalski & Sergot 1986), with which we assume the reader is familar, where communications are events and the contract rules specify how the events initiate and terminate obligation and permission fluents. In this paper we are using the Prolog variable syntax convention where variables begin with an uppercase letter.

Contract Representation

The text of a contract is represented by a binary relation, `contractClause`, between the label of the contract and the clauses belonging to the contract. Variables can be shared between the contract label and the clauses, those appearing in the label are conceptually parameters to the contract. We adopt a compact syntax where the label is written once at the beginning of the contract and the rules are written inside the following brace delimited block. Macro definitions for frequently used terms are written in small caps and marked with $\equiv$ and should be textually substituted by the reader as they are occur.

Figure 1 shows the full text for a purchase contract, parameterised by the item being purchased, the price, the vendor, the customer, the vendor's- and customer's bank, and a delivery tracking service.

```
customerVendorContract_purchase(
     customer:C, vendor:V |
     item:I, price:P, vendorBank:VB,
     customerBank:CB, deliveryTracker:DT) {
  PAID(R) ≡ paid(payer:C, payee:V, price:P, ref:R).
  PAYOBLIG(R, D) ≡ oblig(C, achieve(PAID(R)), D).
  DELIVERED(R) ≡ delivered(item:I, dest:C, invoice:R).
  DELIVEROBLIG(R, D) ≡ oblig(V,achieve(DELIVERED(R)),D).
  INVOICEOBLIG(D) ≡ oblig(V,achieve(value(invoice,_),D).

  initiates(start, INVOICEOBLIG(T+100)).
  initiates(E, PAYOBLIG(R, T+100), T) ←
    initiates(E, value(invoice, R), T).
  initiates(E, DELIVEROBLIG(R, T+300), T) ←
    initiates(E, value(invoice, R), T).
  initiates(E, owns(owner:C, item:I), T) ←
    holdsAt(value(invoice, R), T) ∧
    initiates(E, fulfilled(PAYOBLIG(R, _))).

  terminates(E, owns(owner:V, item:I), T) ←
    holdsAt(value(invoice, R), T) ∧
    initiates(E, fulfilled(PAYOBLIG(R, _))).

  authoritative(V, value(invoice, _)),
  authoritative(VB, PAID(_)).
  authoritative(CB, PAID(_)).
  authoritative(DT, DELIVERED(_)).
}
```

Figure 1: Simple Purchase Contract

The first `initiates` rule, as described above, obliges the vendor to determine an invoice number. Upon determination of the invoice number, the second and third `initiates` rules oblige the customer to pay within 100 time units and the vendor to deliver within 300 time units. The last `initiates` and `terminates` rules indicate that the customer will become the new owner of the item when the pay-

ment obligation has been fulfilled.

The `authoritative` clauses indicate which agents are authoritative for which fluents. In the example, only the vendor is able to determine the invoice number by sending an inform message to the customer. This mechanism also identifies trusted-third parties: the banks are authoritative for the paid fluents (that is any payment from customer to vendor) and the delivery tracker agent is authoritative for the proof of delivery fluent.

The agent maintains a record of all the sent and received events, which are accessible in the `happens` predicate (see below). We require that the events relevant to a contract are observed by all contract principals because the state of the contract depends on the history of events relevant to it. In a two party contract, this requirement is trivially satisfied when one principal sends a message to another.

We abstract away from the details of message format and transport through a set of predicates on the messages. These predicates are used in the contract language and agent code either as tests on received messages, or as constraints on messages about to be sent (this occurs when an agent is obliged to `do` a communication subject to some specified constraints). (Knottenbelt & Clark 2004)

Contract Execution and Planning

Since an agent may have many contracts active at the same time, it is important to be able to consider the contracts both independently of each other (for example to determine a contract's state), and also in combination (for example when outsourcing goals). For this reason we define a meta-interpreter predicate, `selon`, which evaluates queries relative to a specified contract. The first parameter is the contract label, and the second is the formula to be evaluated. The agent can now query what obligations are current with respect to a contract by asking `selon(C, holdsAt(oblig(A, G, DL), Now))` where `Now` is a time point representing the current time and `C` is the label of an active contract.

Both the agent rules and contract rules are represented in the event calculus, so it is possible to create plans to fulfil obligations using an abductive event calculus planner. Planning is about future events and it is necessary to make some assumptions about what kinds of events will (or can) occur in the future. For example, we can plan to perform any actions of our own choosing, but we may only assume that other agents will do those actions that they are obliged to do. We encode these assumptions as integrity constraints in a similar style to (Endriss *et al.* 2004).

Related Work

In Agent0 (Shoham 1991), agents are programmed by specifying a set of capabilities (commitment rules). Instead of building the commitment rules directly into the agent, our architecture allows these rules to be specified in the contract in the form of event calculus initiates and terminates rules.

(Verharen & Dignum 1997)'s cooperative information agents are based on the language action perspective. The architecture specifies three main categories of activities: tasks, transactions and contracts, which are represented as deontic state machines of transaction transitions. Our work differs in conceptual break down, as well as in the underlying formalism (being the event calculus). We expect it to be more reactive and dynamic through the use of both an abductive planner as well as a plan library.

(Kollingbaum & Norman 2002) describes a system of supervised interaction where agents are supervised by a third party called an authority. We do not require this infrastructure, although it does admit a logging agent should the particular situation demand it. Furthermore the agents themselves are responsible for enforcing the contractual norms.

The event calculus has been successfully applied in the representation and monitoring of service level agreements in the utility computing domain (Farrell *et al.* 2004).

Conclusion

Contracts are a useful way to program agent behaviour. When the agent contracts are analogous to real world contracts, they can greatly speed business transactions and lend flexibility to the manner in which they are executed. Furthermore, specifying the relationships between agents separately to the agents' capabilities is not only good software engineering because concerns are separated, but also facilitates analysis and verification since the contracts are represented in a formal language, the event calculus.

Event calculus is especially suitable for contract language representation because the semantics are unambiguous and, given a reliable log of events, the conclusions derived cannot be disputed. Additionally, abduction over event calculus logic programs produces partially ordered plans. Integrity constraints can be used to control the generated plans, and can be used effectively to encode assumptions about obligation fulfilment and future agent action. By executing the abductive proof procedure over all the agreed contracts, an agent may use one contract to solve obligations imposed on it by another, effectively outsourcing the obligation.

References

Endriss, U.; Mancarella, P.; Sadri, F.; Terreni, G.; and Toni, F. 2004. Abductive logic programming with ciff: Implementation and applications. In *Proceedings of the Convegno Italiano di Logica Computazionale (CILC-2004)*. University of Parma.

Farrell, A.; Sergot, M.; Trastour, D.; and Christodoulou, A. 2004. Performance monitoring of service-level agreements for utility computing using the event calculus. In *Proceedings of First IEEE International Workshop on Electronic Contracting, 2004*, 17–25. IEEE.

Knottenbelt, J., and Clark, K. 2004. Contract-based communicating agents. In *Proceedings of the Second European Workshop on Multi-Agent Systems*.

Kollingbaum, M. J., and Norman, T. J. 2002. Supervised interaction: creating a web of trust for contracting agents in electronic environments. In *Proceedings of the First International Joint Conference on Autonomous Agents and Multiagent Systems*, 272–279. ACM Press.

Kowalski, R. A., and Sergot, M. 1986. A logic-based calculus of events. *New Generation Computing* 4(4):319–340.

Shoham, Y. 1991. Agent0: A simple agent language an its interpreter. In *Proc. of AAAI-91*, 704–709.

Verharen, E., and Dignum, F. 1997. Cooperative Information Agents and communication. In Kandzia, P., and Klusch, M., eds., *Cooperative Information Agents, First International Workshop*, number 1202 in LNAI, 195–209.