Electronic contracting in aircraft aftercare: A case study

Felipe Meneguzzi
King's College London
Department of Computer Science
London, United Kingdom
felipe.meneguzzi@kcl.ac.uk

Simon Miles
Michael Luck
King's College London
Dept of Computer Science
London, United Kingdom
simon.miles@kcl.ac.uk

Camden Holt
Malcolm Smith
Lost Wax
72 Lower Mortlake Road
London, United Kingdom
camden.holt@lostwax.com

ABSTRACT

Distributed systems comprised of autonomous self-interested entities require some sort of control mechanism to ensure the predictability of the interactions that drive them. This is certainly true in the aerospace domain, where manufacturers, suppliers and operators must coordinate their activities to maximise safety and profit, for example. To address this need, the notion of norms has been proposed which, when incorporated into formal electronic documents, allow for the specification and deployment of contract-driven systems. In this context, we describe the CONTRACT framework and architecture for exactly this purpose, and describe a concrete instantiation of this architecture as a prototype system applied to an aerospace aftercare scenario.

Categories and Subject Descriptors

D.2.10 [**Software**]: Software Engineering; I.2.11 [**Artificial Intelligence**]: Distributed Artificial Intelligence—*multi-agent systems*

General Terms

Design, Experimentation

Keywords

Electronic contracting, norms, contracts, BDI, AgentSpeak(L)

1. INTRODUCTION

Interactions in systems composed of heterogeneous and self-interested agents are inherently unreliable, requiring some form of societal control to bind these interactions. The introduction of norms has been proposed to address this need in such systems [13], allowing for open societies of autonomous agents that are, nevertheless, regulated to some degree. Such norms are usually specified using deontic concepts, including the notions of obligations, permissions and prohibitions [4] that govern or direct agent behaviour. More specifically, by incorporating sets of these norms into a formal document representation, it is possible to define electronic contracts [17], to allow commercial agent systems controlling business transactions to be defined in terms of contracts that guarantee certain desirable properties at runtime.

Cite as: Electronic contracting in aircraft aftercare: A case study, F. Meneguzzi, S. Miles, M. Luck, C. Holt, M. Smith *et al.*, *Proc. of 7th Int. Conf. on Autonomous Agents and Multiagent Systems (AAMAS 2008)- Industry and Applications Track*, Berger, Burg, Nishiyama (eds.),May,12-16.,2008,Estoril,Portugal,pp.63-70.

In this context, and as part of the CONTRACT project, created to explore multiple aspects of contract-based systems, we have developed an electronic contracting framework aimed at facilitating the construction of deployable systems, and have instantiated the framework within a system addressing the aerospace aftermarket. Our framework includes components for many aspects considered critical for this type of system, such as contract specification, negotiation and monitoring, as well as the appropriate agent architectures to handle these aspects. The system we have developed using the CONTRACT framework, has two purposes: first to refine our understanding of how real systems may be built by transitioning from an abstract to a concrete agent framework, while generating *design patterns* for the management activities in a contract life cycle; and second to provide a prototype that simulates the dynamics of the clauses included in aircraft aftercare contracts, as described in the use case of Section 2, in a fashion not yet achieved elsewhere.

Our main contribution, therefore, is in providing an implementation of a contract-based BDI-style agent system that uses an interleaved planning and execution architecture. So far, no other work has yielded practical systems that allow contractual obligations to be set and monitored by infrastructure agents in the manner ours does. Importantly, our work on exploring contract-based systems is situated in a *practical* commercial context, by specifying and implementing relevant use cases that require contracting functionality. In Section 2 therefore, we begin this paper by introducing the details of the use case from the aerospace aftercare market. Using the CONTRACT architecture described in Section 3, we then define a contract-based system that implements the aerospace scenario in Section 4. In Section 5 we demonstrate how the completed system operates through an example execution. Finally, we compare our system with existing work in Section 6, and conclude in Section 7.

2. AEROSPACE AFTERCARE

In order to situate and motivate the construction of a contract-based framework and architecture, we adopt the use case of aerospace aftercare contracts [10] provided by Lost Wax's Aerogility software [14]. In our system, we greatly simplify Aerogility's simulated scenario in order to focus on the basic elements of the contract framework and explore the dynamics of contract processing.

Aircraft engine manufacturers are increasingly switching from selling engines in isolation to providing long-term service contracts focused on maintaining an operational engine pool. In these contracts, airlines agree to pay engine manufacturers hourly rates for operational engines, while manufacturers agree to provide certain minimum service levels (*i.e.* operational engine availability) or face predetermined financial penalties when aircraft remain grounded waiting for functioning engines. Aftercare contracts can be very

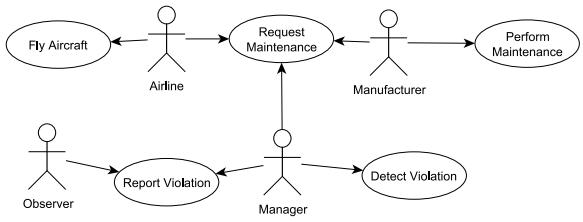

Figure 1: Summary of interactions and use cases.

complex and include provisions for: restricting the provenance of engines, *e.g.* not using engines previously mounted on the aircraft of competitor airlines; specifying a minimum number of available spare engines at specific locations; and allowing maximum idle time for aircraft waiting for repairs.

Engine usage is measured in terms of usage cycles, where the number of cycles clocked up by an engine depends on the length of the flight. Engines have a hard life represented by a predetermined number of cycles, after which they must be refurbished before being used again. In summary, airlines fly aircraft to fulfil a predetermined schedule of flights, which results in cycles being logged for the engines of the aircraft involved. When engines have clocked up enough cycles to end their hard life, an airline sends a request for maintenance to the engine manufacturer, which swaps used engines for new or already serviced ones. Manufacturers need to respond to maintenance requests within a certain time limit, otherwise they are violating the terms of their aftercare contract.

Contract parties can fulfil roles associated with the business goals of an application and administrative roles associated with the maintenance of the contracting environment itself. Business roles are application-specific, and in the case of this aftercare scenario, consist of the *airline operator* and *engine manufacturer*.

Airline operators are responsible for performing flights according to their designated schedule, notifying engine manufacturers of unscheduled events, and scheduling maintenance before an engine's hard life is reached. Engine manufacturers are responsible for performing maintenance operations by the contracted deadlines when these are scheduled, or as soon as possible when these are unscheduled, as well as maintaining a pool of operational engines according to contractual obligations.

We use both *observers* and *managers* as administrative contract parties in our implementation. In particular, an *observer* is responsible for monitoring maintenance requests from the airline and subsequent maintenance operations by the engine manufacturer, notifying the manager when a violation of contract terms is detected. A *manager* is responsible for receiving notifications of violations from the observer and taking action towards remedying them, which in our current implementation consists of notifying a human operator. These interactions and their associated use cases are illustrated in the use case diagram of Figure 1.

3. THE CONTRACT ARCHITECTURE

The CONTRACT framework and architecture allow electronic contracting technologies to be integrated into applications. This provides several benefits.

- Explicit formulations of obligations and prohibitions on contract parties can be reasoned over and acted on by software agents to best meet business objectives.
- Verification of the system, with regard to contracts, enables parties to determine whether it is possible to meet their future obligations.
- Well-specified mechanisms are available to store, maintain the integrity of, and access contract documents.

Since the work presented in this paper aims, in part, to understand and assess the applicability of the architecture, we ensure that each part of it is at least minimally implemented as part of the testbed. In this section, we briefly describe the framework and architecture, in the context of the scenario described above.

3.1 Overall Structure

The CONTRACT *framework* is a conceptual model for specifying applications using electronic contracting. The *architecture* is an instantiation of the contract administration aspects of the framework: a set of service-oriented middleware and multi-agent design patterns to support administration of electronic contracts.

In Figure 2 we show the overall structure of the framework and architecture. As a whole, this can be seen as series of models and specifications, comprising a methodology for adapting application designs to utilise electronic contracts. The primary component is the framework, depicted at the top, which is the conceptual structure used to describe a contract-based system, including the contracts themselves and the agents to which they apply. Each level in the figure provides support for the components below it. Arrows indicate where one model influences or provides input to another.

From the framework specification of a given application, other important information is derived. First, off-line verification mechanisms can check whether the contracts to be established obey particular properties, such as being achievable, given the possible states the world can reach. From this, and the contracts themselves, we can determine which states are *critical* to observe during execution to ensure appropriate behaviour. A critical state of a contract-based system with regard to an obligation essentially indicates whether the obligation is fulfilled or fulfillable, (*e.g.* , achieved, failed, in danger of not being fulfilled, etc.).

The framework specification is used to determine suitable processes for administration of the electronic contracts through their lifetimes, including establishment, updating, termination, renewal, and so on. Such processes may also include observation of the system, so that contractual obligations can be enforced or otherwise effectively managed, and these processes depend on the critical states identified above. Once suitable application processes are identified, we can also specify the roles that agents play within them, the components that should be part of agents to allow them to manage their contracts, and the contract documents themselves.

3.2 Contracts

Agreements between agents are formally described in electronic *contracts*, which document *obligations*, *permissions* and *prohibitions* (collectively *clauses*) on agents. Agents bound by contract clauses are said to be *contract parties*, and a contract specifies *contract roles*, which are fulfilled by contract parties, so that clauses apply to specific contract roles. The life cycle of a contract may be broken down into five stages, as illustrated in Figure 3:

- creation, including the process of finding potential interaction partners and negotiating terms for a contract;
- maintenance and update of the formal representation of a contract document in a controlled repository;
- fulfillment of the contract clauses by the participants;

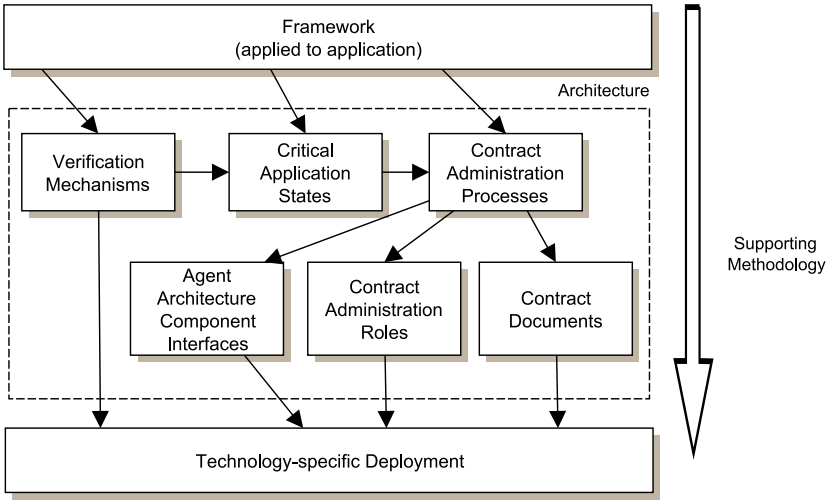

Figure 2: Overall structure of the CONTRACT architecture.

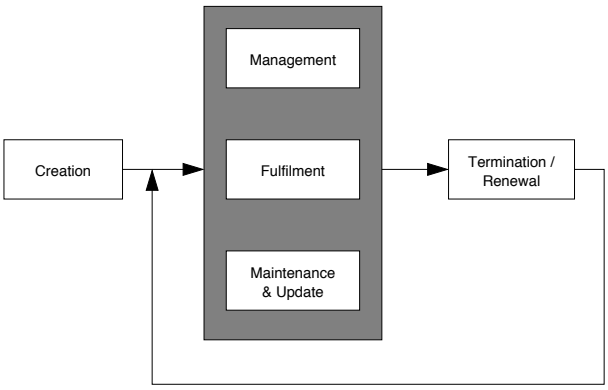

Figure 3: Life cycle of a contract.

- management, consisting of overseeing the fulfillment of obligations by designated agents, and taking action when violations are detected;
- termination or renewal of contracts when contracts are about to expire, or when their validity is violated.

3.3 Contract Parties

Contracts in our system are agreed upon by agents, which are assumed to be autonomous, pro-active, flexible (decision-making) and social. Agents engage in contract-directed interactions to fulfil the clauses specified in a contract. Contract interactions require a minimum of two agents fulfilling the role of participants. Some applications may require contract-related processes to have certain properties, *e.g.* that violations are acted on, or that the integrity of the contract documents is maintained. These requirements lead to obligations on (and the creation and use of) administrative parties, and contracts may document their required behaviour We can roughly classify contract parties into two kinds.

Business Contract Parties Agents for whom the contract is created: the obligations on the business contract parties are largely concerned with the *business* of the application. In our use case example, the aircraft operator and engine manufacturer are business contract parties.

Administrative Contract Parties Agents are required to ensure that the contract is accessible, retains integrity and legitimacy, is monitored and enforced, and other such administrative functions that ensure the contract has force. The obligations on these agents relate to their administrative roles.

3.4 Enforcement

Two particular administrative contract party roles are those of *observer* and *manager*. The former detects whether the system enters a critical state (success, violation, in danger of violation) with regard to a particular clause. A manager reacts on the basis of observation, *e.g.* to inform a user of the problem, penalise a contract party in some way, and so on. There may be several observers and managers for an application, for example checking compliance on behalf of different users, and handling violations in different ways.

As discussed in the next section, we add single instances of both administrative roles to the use case implementation.

4. A CONTRACT-BASED SYSTEM FOR THE AEROSPACE AFTERMARKET

To allow the evaluation of the contracting framework separately from the specific use case, we have divided the system into two main parts: the aftercare simulation and the contract-related functionality. The simulation part of our system implements a small subset of the aftercare scenario, including the scheduling of flights, update of engine usage information and engine maintenance operations. The contracting part of our system focuses on the communication and monitoring of requests by contract partners to fulfil their obligations, and taking action when violations of these obligations are detected.

4.1 AgentSpeak and Jason

As we have seen, the architecture developed for the CONTRACT framework is driven by events in the environment that are associated with certain conditions specified in contract clauses. These

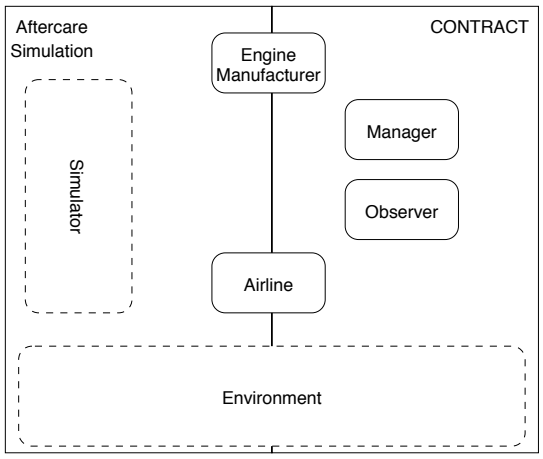

Figure 4: Main parts of the system.

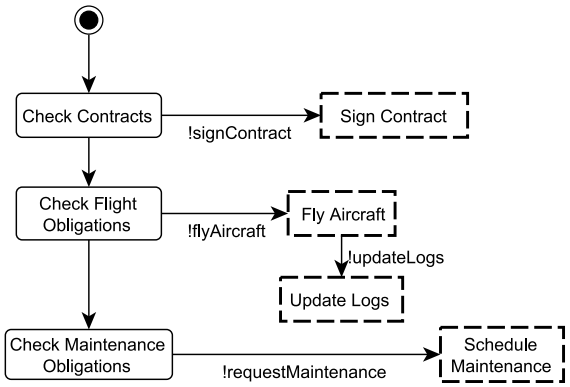

Figure 5: Airline operator reasoning.

events drive complying agents to adopt plans to fulfil their obligations. Such a mechanism lends itself very well to implementation through reactive-planning BDI agents, such as PRS [9], and AgentSpeak(L) [16]. In consequence, our CONTRACT demonstrator was implemented using Jason [2], which is a Java-based AgentSpeak(L) [16] interpreter. More specifically, AgentSpeak(L) [16] is an agent language, as well as an abstract interpreter for the language, and follows the *beliefs, desires and intentions* (BDI) model of practical reasoning [3]. In simple terms, a BDI agent tries to realise the *desires* it *believes* are possible by committing to carrying out certain courses of action through *intentions*, and in AgentSpeak(L), this is simplified in that an agent chooses plans of action that are considered possible by the agent's beliefs, making the notion of desires implicit in the plan representation. The language of AgentSpeak(L) allows the definition of *reactive procedural plans*, so that plans are defined in terms of events to which an agent should react by executing a sequence of steps (*i.e.* a procedure). Plan execution is further constrained by the context in which these plans are relevant.

In this section, we describe the implementation of the aerospace aftermarket scenario in AgentSpeak(L), by detailing the descriptions of the various roles and their corresponding agents in terms of the AgentSpeak(L) plans used by these agents in fulfilling their goals. In what follows, we denote plans by the AgentSpeak(L) triggering event associated with the execution of that plan, as $!plan$.

4.2 Roles and Agents

We have developed agents, and their corresponding behaviours, to fulfil goals in both the simulation and the contract-related parts of the system, as illustrated in Figure 4. Clearly, the administrative roles play no part in the simulation of the aftercare services, whereas the manufacturer and the operator have goals that span both areas, since their goals relate to the simulation as well as the fulfilment of contractual obligations. These agents/roles are summarised in Table 1, and further detailed in the following sections.

4.2.1 Airline Operator

Airline operators manage a fleet of aircraft and have a flight schedule that must be fulfilled throughout the system execution. When the system starts up, operators receive information regarding their aircraft and engines as perceptions: *aircraft(Name, Airline, Location)*, where *Name* is the aircraft identifier, *Airline* is the name of the operating airline, and *Location* is the current location of the aircraft; and *engine(Engine, Location, Cycles, Provenance)*, where *Engine* is the engine identifier, *Location* is the current location of the engine, which may be an airport or an aircraft (in case it is mounted on one), *Cycles* is the number of usage cycles currently logged for the engine, and *Provenance* is the list of aircraft on which the engine has been mounted in the past. Moreover, operators receive the schedule of flights that must be flown through perceptions, *scheduledFlight(Time, Operator, Aircraft, Origin, Destination)*, where *Time* is the scheduled time for the flight, *Operator* is the airline responsible for executing the flight, *Aircraft* is the aircraft that should be used to carry out the flight, *Origin* is the departure point of the flight and *Destination* is the arrival point of the flight. After having the simulation data internalised in the belief base, airline operators seek to sign maintenance contracts with engine manufacturers to provide for the aftercare of the engines used by their fleet. Initially, an operator broadcasts its intention to sign maintenance contracts using the *!requestContracts* plan, and engine manufacturers that are willing to provide this service reply with a message of *acceptContract* (received as a perception by the operator). Upon receipt of an acceptance by an engine manufacturer, an operator signs the contract using the *!signContract* plan.

As flights are carried out, the usage logs of aircraft and their components need to be updated to reflect the number of usage cycles spent in each flight. Operators use *!flyAircraft* plans to fly aircraft, which in turn use *!updateLogs* plans to update the number of usage cycles for an aircraft's engines. An airline operator is responsible for tracking usage data and matching it to the known hard life of engines to allow for maintenance to be scheduled ahead of time with the engine manufacturer, which is achieved through the *!requestMaintenance* plan. Engines are also vulnerable to unscheduled problems, which demand immediate scheduling of maintenance with the engine manufacturer, again through the same *!requestMaintenance* plan. A high-level view of the reasoning cycle for the airline operator is illustrated in Figure 5, including the AgentSpeak(L) plans invoked in this process (as solid boxes) and their results (as dashed boxes).

4.2.2 Engine Manufacturer

Engine manufacturers build and own a pool of aircraft engines that are sold or leased to airline operators which may also contract them to carry out regular maintenance on such engines. Like the airline operator, the engine manufacturer has its belief base

Agent/Role	Percepts	Goals	Plans
Operator	scheduledFlight unscheduledEvent maintenanceDone aircraft engine acceptContract	Perform flights according to schedule Notify engine manufacturer of unscheduled events Schedule maintenance ahead of time	*!flyAircraft* *!updateLogs* *!requestMaintenance* *!requestContracts* *!signContract*
Manufacturer	requestMaintenance requestContract engine	Perform scheduled maintenance according to deadlines perform unscheduled maintenance as soon as possible	*!performMaintenance* *!evaluateContract* *!acceptContract* *!moveEngine*
Manufacturer and Operator		Maintain the observer informed of the communication between the Manufacturer and the Operator	*!contractSend* *!notifyObserver*
Observer	requestMaintenance maintenanceDone	Notify the manager when violations occur Monitor maintenance requests from the airline operator Monitor maintenance actions from the engine manufacturer	*!handleMessage* *!notifyManager* *!checkMaintenanceDone*
Manager	violation	Receive notifications from the Observer and detect contractual violations notify relevant parties of violations	*!handleViolation*

Table 1: Summary of roles.

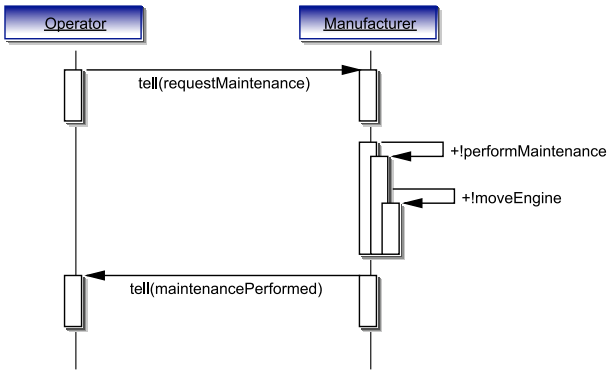

Figure 6: Events associated with a maintenance request.

initialised at the beginning of the simulation with information on the engines it can maintain. As operators send requests for maintenance contracts, manufacturers receive *requestContract* perceptions, and decide whether or not to accept these requests through the *!evaluateContract* plan, possibly accepting the request using the *!acceptContract* plan.

Once a contract is established, an engine manufacturer is obliged to respond to scheduled maintenance requests within a predetermined time frame, and to unscheduled requests as soon as possible. Requests for maintenance are received by the manufacturer as *requestMaintenance* perceptions and, if they are valid, maintenance is performed using the *!performMaintenance* plan. Maintenance involves moving the required spare engine to the location of the aircraft needing maintenance and swapping it with the used engine. Once a manufacturer has finished performing maintenance, it informs the operator that the aircraft is ready to fly by sending it a *maintenanceDone* message. This sequence of events is illustrated in Figure 6.

```
+!contractSend(Target,SpeechAct,Message) : true
   <- .send(Target, SpeechAct, Message);
      !notifyObserver(Target,SpeechAct,Message).

+!notifyObserver(Target,SpeechAct,Message) :
    observer(Observer)
   <- .my_name(From);
      .send(Observer, tell, message(
          From,Target,Message)).
```

Listing 1: Communication plans.

4.2.3 Observer

An observer is responsible for monitoring the activities of contract parties and detecting whether or not any contract violations take place. In the aftercare scenario, an observer only monitors the requests and responses to maintenance operations. In order to monitor these requests, our observer implementation leverages the fact that agents representing contract parties include plans that comprise a communication layer for CONTRACT-related communication. For example, if an operator wishes to send a maintenance request to an engine manufacturer, instead of directly invoking a communication action, it uses plans that include communication actions, as well as replicating sent messages to the observer. This layer consists of a plan that ensures that all messages exchanged between contract parties are also forwarded to the observer, keeping it up to date regarding the status of contractual commitments. The plans to send messages and to notify the observer are shown in Listing 1, and the flow of messages from each role is illustrated in Figure 7. Such an approach is similar in spirit to that of Garcia-Camino *et al.* [6], since agents are not directly aware that their contractual obligations are monitored by an external agent through each agent's communication layer.

Since the observer is not a compulsory part of the system, its existence and identity is not known in advance by airline operators or engine manufacturers. Therefore, at the beginning of the simula-

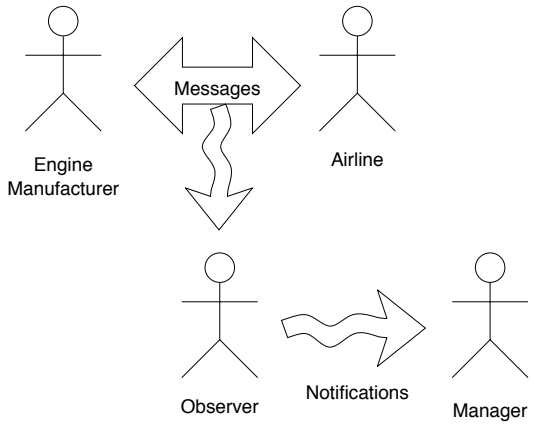

Figure 7: Flow of messages among the agents.

```
+message(From,To,Message) [source(From)] : true
   <- !handleMessage(Message, From, To).
```

Listing 2: Plan to handle messages between contract parties.

tion, all contract parties broadcast a message requesting observers to identify themselves. If an observer is present in the system, it replies with its identity, which allows contract parties to forward their communication to it. This is illustrated in Figure 8.

Observed messages are handled by the observer in a generic way, as illustrated by the plan in Listing 2. In order to detect violations of deadlines for maintenance, observers generate triggers associated with the deadline for maintenance operations. Whenever an observer detects a request for maintenance from an operator, it awaits confirmation from the manufacturer that maintenance has been performed (*i.e. maintenanceDone*), as shown in the plans of Listing 3.

If the deadline for this maintenance request is reached without the observer having perceived a *maintenanceDone* message, it notifies the manager responsible for the actual handling of the violation within the system. This is illustrated by the plans in Listing 4.

4.2.4 Manager

Whenever a violation is detected by the observer, it notifies the manager, which is responsible for taking some sort of remedial action. In our system, managers have a generic plan to handle violation notifications through some pre-defined way, as illustrated in Listing 5. In this example, a plan reacts to the perception of a violation (*violation(Violation, From, To)*), which is forwarded to the *!handleViolation* plan where the violation is actually handled by the Manager. In the aftercare scenario, the only possible violation relates to an engine manufacturer not honouring the deadline for scheduled maintenance (represented as *maintenance(Time)*) and the handling of this violation consists simply of informing a human user of the violation through a console message.

5. EXECUTION EXAMPLE

Figure 9 shows a screenshot of the application depicting the Jason console and the messages sent by the agents as they carry out their obligations. In it, we can see two airlines, *simpleJet* (highlighted in solid boxes), and *millerAir* (highlighted in dashed boxes), performing flights, and updating engine logs as they execute these flights. At one point in the simulation, an engine mounted in a

```
+!handleMessage(
     requestMaintenance(
         Time,Plane,Location,Engine), From, To)
   : true
   <- !print("Handling request for maintenance
       from ",From," to ", To);
      //When I hear a maintenance request,
      //store the request
      +maintenanceRequested(
          Time,Plane,Location,Engine,From,To);
      ?maintenanceDeadline(Deadline);
      TriggerDeadline = Time+Deadline;
      //And create a trigger
      +trigger(maintenance,TriggerDeadline).

+!handleMessage(
      maintenanceDone(Time,Plane,Engine), From,
          To)
    : maintenanceRequested(
        TimeReq,Plane,Location,Engine,To,From)
   <- +maintenancePerformed(
       Time,Plane,Location,Engine,From,To).
```

Listing 3: Observer plans to handle maintenance requests.

```
+!checkMaintenanceDone(maintenance(
    Time,Plane,Engine))
   : maintenanceRequested(
       Time,Plane,Location,Engine,From,To) &
     maintenancePerformed(
         TimeDone,Plane,Location,Engine2,To,From)
          &
     time(Now) &
     (TimeDone < Now)
   <- true. //No violation detected

+!checkMaintenanceDone(maintenance(
    Time,Plane,Engine))
   : maintenanceRequested(
       Time,Plane,Location,Engine,From,To) &
     time(Now)
   <- .send(manager, tell, violation(maintenance(
       Now),To,From)).
```

Listing 4: Observer plan to detect violations.

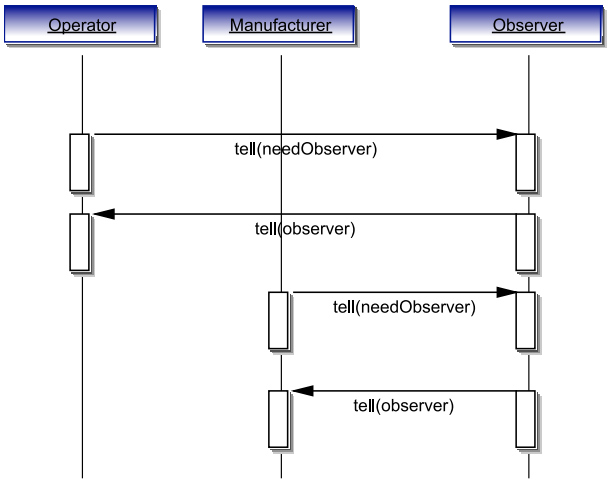

Figure 8: Finding an observer.

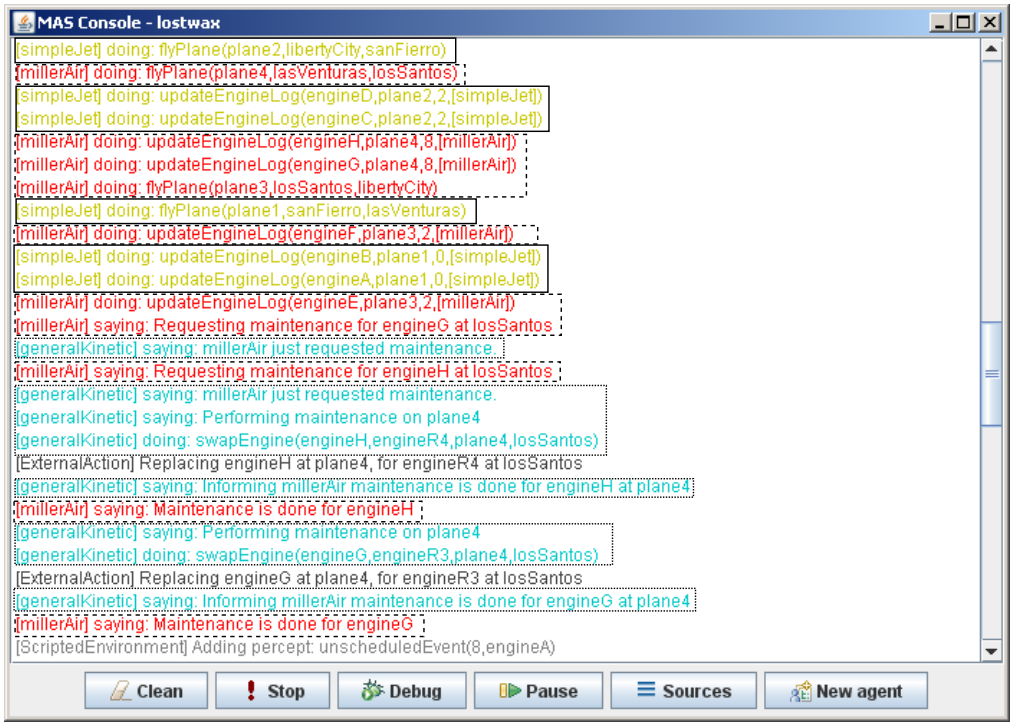

Figure 9: Screenshot of the application.

```
+violation(Violation,From,To) [source(S)]
   : true
   <- !handleViolation(Violation,From,To).

+!handleViolation(maintenance(Time),From,To)
   : true
   <- .print(From, " should have done maintenance
        for ",To," by time ",Time,", but it did
       not.").
```

Listing 5: Manager plan to handle violations.

millerAir aircraft reaches the end of its hard life, and needs to be replaced. This prompts *millerAir* to request its contracted engine manufacturer, *generalKinetic* (highlighted in dotted boxes) to perform maintenance. The engine manufacturer complies by swapping the specified engines and notifying *millerAir* that maintenance has been performed.

6. RELATED WORK

Unlike existing work on automated contracting, which we consider in this section, our system is the first (of which we are aware) to have formal underpinnings, cover the entire contracting ecosystem, and be able to function in complex, realistic domains.

For example, Kollingbaum [12] described the NoA architecture for norm aware agents. In his framework, agents are able to reason about contracts and the obligations imposed on them, before deciding what actions to take. The focus of the NoA framework is on the nature of normative agents, rather than on the exact form a realistic contract should take, and evaluation thus takes place in a simple blocks world.

Other notable work on contracts is also abstract in nature. For example, Dignum et al. [5] describe a language for contracting that is based on modal logics. While elegant, no attention is paid to the rest of the contracting ecosystem. Similarly, most existing work on contracting focuses on the contracting language, or some other very specific aspect, such as verification or conflict detection (c.f. Grosof's SweetDeal [8], Abraham's work using disquotation theory [1], and Giannikis's e-Contracts system [7]), without attempting to situate the contract in a larger environment. Other approaches, such as Neal et al.'s DLP [15] have no formal semantics, tying them to a single specific implementation.

Finally, work on policies (e.g. [11]) seems to attempt to deal with systems as a whole, rather than consulting components in isolation. However, while such work shares similar language to contracts, such as obligations, permissions and prohibitions, such norms form hard constraints in policies. By contrast, within contracts, such norms form soft constraints, providing the requisite flexibility to make contracting a much more practical tool for dealing with many realistic situations where violations are not only often unavoidable, but sometimes even desirable.

7. CONCLUSIONS AND FUTURE WORK

The CONTRACT project seeks to develop frameworks, components and tools that make it possible to model, build, verify and monitor distributed electronic business systems on the basis of dynamically generated, cross-organisational contracts which underpin formal descriptions of the expected behaviours of individual services and the system as a whole. The project covers both theoretical and practical aspects aimed at:

- specifying electronic business-to-business interactions in terms of contracts;
- dynamically establishing and managing contracts at runtime

in a digital business environment;

- applying formal verification techniques to collections of contracts in a digital business environment;

- and, applying monitoring techniques to contract implementation in order to help provide the basis for business confidence in e-Business infrastructures.

In this paper we have shown how the CONTRACT framework and architecture developed in the project provides some of this functionality through instantiation as an implemented system in the context of the aerospace aftermarket. In particular, we have adopted an AgentSpeak(L) approach in our prototype system, and have described its design and operation. We provide concrete versions of the contractual roles envisioned for CONTRACT-based systems, in particular, the administrative roles of *Observer* and *Manager*, and the business roles for the *Airline Operator* and *Engine Manufacturer* from our reference scenario. Moreover, we provide an observation mechanism that includes plan patterns that can be reused in other CONTRACT-based applications.

Our system demonstrates two important issues in relation to practical application of contract-based agent technology. First, it shows how we can provide functionality that moves beyond the current generation of aerospace aftermarket tools, by providing a contract level analysis of the agreements between the parties involved, with the capacity to identify and flag violations or potential violations for individual contracts. This moves significantly beyond aggregated simulation of provision, which is the typical mode of such decision support tools. Second, it shows how traditional (BDI-style) agent architectures such as AgentSpeak(L), most commonly seen in academic environments, can be deployed to provide the reasoning capability that is required to perform this kind of analysis.Thus, not only do we address the business case for contract-based systems, we also demonstrate the use of standard agent technologies in such business contexts.

Further work will continue on refining the architecture in light of experience of using the system, as well as from experience gained with several other use cases in the domains of dynamic insurance settlement, modular certification testing, and service-level agreements in software engineering [10]. In addition, we will incorporate verification techniques into the architecture to provide the complete functionalities described above.

Acknowledgments

The research described in this paper is partly supported by the European Commission Framework 6 funded project CONTRACT (INFSO-IST-034418). The opinions expressed herein are those of the named authors only and should not be taken as necessarily representative of the opinion of the European Commission or CONTRACT project partners.

The first author is supported by Coordenação de Aperfeiçoamento de Pessoal de Nível Superior (CAPES) of the Brazilian Ministry of Education.

8. ADDITIONAL AUTHORS

Additional authors: Nir Oren (King's College London, email: nir.oren@kcl.ac.uk), Nora Faci (King's College London, email: noura.faci@kcl.ac.uk), Sanjay Modgil (King's College London, email: sanjay.modgil@kcl.ac.uk), Martin Kollingbaum (Carnegie Mellon University, email: mkolling@cs.cmu.edu).

9. REFERENCES

[1] A. S. Abrahams and J. M. Bacon. A software implementation of Kimbrough's disquotation theory for representing and enforcing electronic commerce contracts. *Group Decision and Negotiation*, 11(6):487 – 524, 2002.

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

[3] M. E. Bratman. *Intention, Plans and Practical Reason*. Harvard University Press, Cambridge, MA, 1987.

[4] A. Daskalopulu, T. Dimitrakos, and T. Maibaum. Evidence-based electronic contract performance monitoring. *Group Decision and Negotiation*, 11(6):469–485, 2002.

[5] V. Dignum, J. J. Meyer, F. Dignum, and H. Weigand. Formal specification of interaction in agent societies. In *Proceedings of the Second Goddard Workshop on Formal Approaches to Agent Based Systems*, pages 37–52, 2002.

[6] A. García-Camino, J.-A. Rodríguez-Aguilar, and W. Vasconcelos. A distributed architecture for norm management in multi-agent systems. In *Proceedings of the Workshop on Coordination, Organization, Institutions and Norms in agent systems (COIN)*, 2007.

[7] G. K. Giannikis and A. Daskalopulu. Defeasible reasoning with e-contracts. In *Proceedings of the IEEE/WIC/ACM International Conference on Intelligent Agent Technology (IAT'06)*, pages 690–694, 2006.

[8] B. Grosof and T. C. Poon. SweetDeal: Representing agent contracts with exceptions using semantic web rules, ontologies, and process descriptions. *International Journal of Electronic Commerce*, 8(4):61–98, 2004.

[9] F. F. Ingrand, M. P. Georgeff, and A. S. Rao. An architecture for real-time reasoning and system control. *IEEE Expert, Knowledge-Based Diagnosis in Process Engineering*, 7(6):33–44, 1992.

[10] M. Jakob, M. Pěchouček, J. Chábera, S. Miles, M. Luck, N. Oren, M. Kollingbaum, C. Holt, J. Vázquez, P. Storms, and M. Dehn. Case studies for contract-based systems. In *Proceedings of the 7th International Conference on Autonomous Agents and Multiagent Systems*, 2008.

[11] L. Kagal. Rei: A policy language for the mecentric project. Technical Report HPL2002270, HP Labs, 2002.

[12] M. Kollingbaum. *Norm-governed Practical Reasoning Agents*. PhD thesis, University of Aberdeen, 2005.

[13] F. Lopez y Lopez, M. Luck, and M. d'Inverno. A normative framework for agent-based systems. In *Proceedings of the First International Symposium on Normative Multi-Agent Systems*, 2005.

[14] LostWax. Aerogility. http://www.aerogility.com/, 2007.

[15] S. Neal, J. Cole, P. F. Linington, Z. Milosevic, S. Gibson, and S. Kulkarni. Identifying requirements for business contract language: a monitoring perspective. In *Proceedings of the Seventh IEEE International Enterprise Distributed Object Computing Conference*, 2003.

[16] A. S. Rao. AgentSpeak(L): BDI agents speak out in a logical computable language. In W. V. de Velde and J. W. Perram, editors, *Proceedings of the Seventh European Workshop on Modelling Autonomous Agents in a Multi-Agent World*, volume 1038 of *LNCS*, pages 42–55. Springer, 1996.

[17] L. Xu and M. A. Jeusfeld. Pro-active monitoring of electronic contracts. In *Advanced Information Systems Engineering*, volume 2681 of *LNCS*, pages 584–600. Springer, 2003.