Towards Practical Reasoning Agents for the Semantic Web

Ian Dickinson
Hewlett-Packard Laboratories
Filton Road, Stoke Gifford
Bristol BS34 8QZ, U.K.
Ian.Dickinson@hp.com

Michael Wooldridge
Department of Computer Science
University of Liverpool
Liverpool L69 7ZF, U.K.
M.J.Wooldridge@csc.liv.ac.uk

ABSTRACT
We describe *Nuin:* a flexible agent architecture designed for practical development of agents in Semantic Web applications, based around belief-desire-intention (BDI) principles. We outline the central design features of the platform, and show how the implementation is designed to give maximum flexibility to agent designers, while retaining the overall coherence of the design.

Categories and Subject Descriptors
I.2.11 [**Artificial Intelligence**] Distributed Artificial Intelligence – *Intelligent Agents*.

General Terms
Design, Languages.

Keywords
Agent architectures, BDI, Semantic Web, agent programming.

1 INTRODUCTION
For some years, *software agents* have been proposed as a new metaphor for the design and construction of computational systems. Agent-based software promises many innovative capabilities, including those that many researchers regard as defining characteristics for agency: proactivity, reactivity, social ability and autonomy [29]. The more recent promotion of the Semantic Web [3] as a vision of the evolution of the World Wide Web from a publishing medium to a general services fabric, shares many ideals with the vision of agent researchers. Indeed, many descriptions of the semantic web include the use of agents as an enabling technology for delivering services to the users [15].

To date, agent systems research has delivered many results in the theories and architectures of agent software. Practical agent-based toolkits and applications have been developed, but there is an observable separation between theory and practice. While conducting research into user experiences of autonomous agents in a Semantic Web context, we discovered a lack of agent platforms that are both robustly engineered and that strongly embody the results of research into agent theories. We have therefore developed our own agent tool, named *Nuin*, to address this need. This paper describes the architecture and design principles of Nuin.

The rest of the paper is structured as follows. §2 describes our overall motivation and goals in more detail, while §3 and §4 presents the design solutions to these goals. §5 evaluates Nuin's contributions compared to other platform projects. §6 presents some conclusions.

2 MOTIVATIONS
One goal of our research is to evaluate user reactions when using autonomous agents to assist with information-centric tasks on the Semantic Web. Whether these evaluations are done in a usability lab or by deploying applications "in the field", we need to be able to construct reliable and capable agent systems – otherwise we are only examining user reactions to unreliable software. As an enabling step towards these research goals, we have developed a platform for agent programming designed to allow us to concentrate on the key capabilities of the agents that support the applications we want to test (in contrast to, for example, distributed agent infrastructure issues). This paper reports progress towards a *practical architecture for deliberative agents for the Semantic Web*. There are several components to this overall objective. We have attempted to design a practical tool that is both flexible and robustly engineered, so that it can support a variety of interesting applications. Our software will be freely available to other research groups to use, so flexibility is an essential feature if the platform is to be generally useful. We want to embed agent characteristics as part of the overall user experience, rather than force application designers to use the agent platform as an application container.

A key interest in exploring the user experience of autonomous agents is the ability to delegate goals and preferences from the user to the agent. While recognising the utility of reactive and hybrid agent architectures, we chose to make as a first priority the development of *deliberative agents* [28]: agents who deliberate over symbolic knowledge representations. This will allow us to explore the central issues in communicating declarative goals from a human to an agent, and using goals as a basis for collaboration.

The most commonly studied architecture for deliberative agents is the *belief-desire-intention* (BDI) model [19]. Since this is, at its core, a model founded on a view of practical reasoning, it is well suited to be the basis of a robust implementation. There are known problems in providing a complete implementation of a logical BDI model; we mention the some of the consequences of these problems below. Nevertheless, we took as an objective that the implementation should strongly embody a BDI theoretical foundation.

We specifically want to build agents for the *Semantic Web*. There are several reasons for this. Firstly, a number of key tools and

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
AAMAS '03, July 14–18, 2003, Melbourne, Australia
Copyright 2003 ACM 1-58113-683-8/03/0007…$5.00.

techniques likely to be of use to a deliberative agent are being developed under the aegis of Semantic Web research. In particular, we note the increasing prevalence of ontology design tools, ontology reasoners and persistent RDF [30] stores among others. Secondly, if it delivers on its promise, the Semantic Web will make available a variety of sources of knowledge for use by agents. The use of abstract or toy domains as a vehicle for demonstrating agent capabilities reduces the impact of the demonstration, but more importantly does not provide a suitable context for conducting realistic experiments with users. Finally, it can be hoped that the drive for the Semantic Web will provide a 'gravity well' which will pull a variety of agent tools and applications into closer collaboration.

We distil these top-level goals into the following design principles:

1. Base the agent design on accepted BDI theory
2. Use good software engineering practices to engineer a solid platform enabling robust agent applications;
3. Embed the agent capabilities in the end-user application, not vice-versa;
4. Re-use existing tools and standards where applicable;
5. Design the architecture to interoperate with the emerging standards and tools of the Semantic Web.

In order to make use of existing tools, and to minimise dependencies on the underlying operating system, our software is written in Java™.

3 OUTLINE OF APPROACH

3.1 BDI Foundations

We wish to design agents that can perform practical reasoning in a dynamic, unpredictable world. The use of explicit *mental attitudes*, it has been argued, allows the agent to manage its internal structures and external environment, and so achieve a balance between optimal behaviour and resource limitations. A common choice of foundational mental attitudes is the set *belief, desire,* and *intention* [19]. BDI agents have been extensively studied in the agent literature, and have attracted both formal logical characterisations [28] and (rather fewer) practical implementations and re-usable tools (e.g. [14]).

Our objective with the Nuin architecture is to create practical agents: that is, agents that not only perform practical reasoning, but realistically deployable embodiments in software. We must, therefore, balance the desire to build upon a formal, but non-computable logical model, against *ad hoc* implementation approaches that do not have well-characterised properties. We have based our design on Rao's *AgentSpeak(L)* [18]. Below, we briefly summarise AgentSpeak(L), and motivate our extensions to Rao's framework. AgentSpeak(L) was selected as it distils experiences with the design of PRS [19] into a simple formalism that has both a tractable implementation and a formal characterisation.

3.1.1 Overview of AgentSpeak(L)

Rao proposed AgentSpeak(L) as an abstraction of the reactive planning model that underpins the Procedural Reasoning System (PRS) [19]. An AgentSpeak(L) agent consists of a set of beliefs, constructed from an alphabet of first-order terms, and a set of plans.

Plans are constructed from a set of action symbols, together with connectives for serial action sequences, tests, achieving a goal and raising an event. A plan has the structure:

$$e : b_1 \wedge \ldots \wedge b_m \leftarrow h_1 ; \ldots ; h_n$$

where the h_i are actions forming the plan body, b_j are belief literals denoting the plan context, and e is a triggering expression. The triggering expression matches events, and thus may be used to respond reactively to the environment, or to chain between plans. AgentSpeak(L) distinguishes *external* (from the environment) and *internal* (raised from a plan body) events. Handling a new external event creates a new intention frame; internal events are handled in the context of the intention frame of the parent plan. For further details of AgentSpeak(L), including the proof theory, the reader is referred to [18].

3.1.2 Limitations of AgentSpeak(L)

As a preliminary design exercise, we coded an implementation of AgentSpeak(L) in Prolog. This helped us identify the following issues that need to be addressed to make a practical programming tool based on AgentSpeak(L):

- Intentions in AgentSpeak(L) are defined as a commitment to a particular plan to handle an initiating exogenous event. There is no sense of committing to achieve a goal held by, or shared with, another agent, nor of committing to maintain a state of the world.
- All choices in the AgentSpeak(L) interpreter are non-reversible and have no contingencies. If a plan body fails, the whole plan (and in a naïve implementation the whole interpreter) simply fails.
- The subtlety in programming an agent comes down to which event to focus on, and which plan to intend to handle that event. These choices are encapsulated in *choice functions* (for example functions S_O and S_I in [18]) but these are otherwise not discussed in Rao's paper.

These limitations notwithstanding, AgentSpeak(L) provides an elegant foundation for an extended architecture. In particular, we note the use of events and triggers to provide both goal-directed (back chaining) and data-directed (forward chaining) reasoning, and to allow the agent to multi-task multiple simultaneous plans.

3.2 Architectural Foundations

We now briefly outline the software engineering principles that provide the architectural foundations for our implementation. Firstly, we aim to re-use existing components wherever possible, without sacrificing the coherency of the design. For example, a key component of agent systems as a whole is the inter-agent communications framework. Many solutions exist for this capability, whether custom [22] or standards-based [2;6]. Rather than select and bind strongly to any one such platform, our approach is to abstract the key capabilities these platforms provide into *interfaces*. Using a particular agent middleware platform then becomes a matter of defining bindings from the platform's services to the abstract communications interfaces with Nuin.

The design pattern of using interfaces for all key abstractions, sometimes termed *interface-driven design*, provides a more extensible and adaptable starting point than other design approaches, such as the use of class hierarchies [5]. We therefore use interfaces, rather than classes, to define all of the key

abstractions in the Nuin toolkit. The *factory* design pattern [11] is then used to insulate programmers from the detailed implementations of these interfaces. To create a new object conforming to a given interface, the programmer invokes a method on the appropriate factory object rather than directly invoke the class constructor. To extend the capabilities of one of these abstractions, for example adding probabilistic weights to a logical term, the programmer defines a new set of implementations of the logical value interfaces and registers a new factory object for creating them. This addresses one of our central goals of making the platform flexible and extensible for programmers.

3.3 Semantic Web Foundations

There is as yet no crisp definition of the Semantic Web, so it is difficult to be precise about what it means for an agent to be designed to operate on the Semantic Web. There are some principles, however, that are emerging:

- symbols are *uniform resource identifiers* (URI's);
- XML namespaces [23] are used to keep vocabularies of symbols from clashing accidentally;
- RDF [30] triples[1] are the basic foundation of knowledge representation;
- ontological information is encoded in DAML+OIL [1] or OWL [25], which extends the representational capability of the underlying RDF;
- knowledge sources are openly available and decentralised, typically using HTTP as an access mechanism.

Doubtless additional principles will continue to evolve as more Semantic Web applications are investigated and deployed.

Beyond these technology foundations, a further theme in Semantic Web processing is the "web-ness" of information resources. This is primarily a social point, though it has technical implications. Roughly, it says that information, and authority, is decentralised, sometimes put as "anyone can say anything about anything". It implies that any set of definitions or statements can be extended, and none can be considered authoritative except by consensus. In this worldview, provenance of information becomes paramount, and agents necessarily will have to deal with incomplete and contradictory information, tangled ontologies and potentially deliberate falsehoods. We do not claim that Nuin solves these problems; rather it is our intent to create a platform for exploring these issues in further research.

4 NUIN AGENT ARCHITECTURE

Figure 1 shows the principal components of the Nuin architecture. The configuration of the agent itself is defined by an RDF model. Thus the only start-up parameter that an agent requires is a URL from which it can retrieve its configuration. The components from figure 1 are described in further detail in the following sections.

[1] A triple is a ground tuple from a binary relation, and is the only knowledge structure that basic RDF contains.

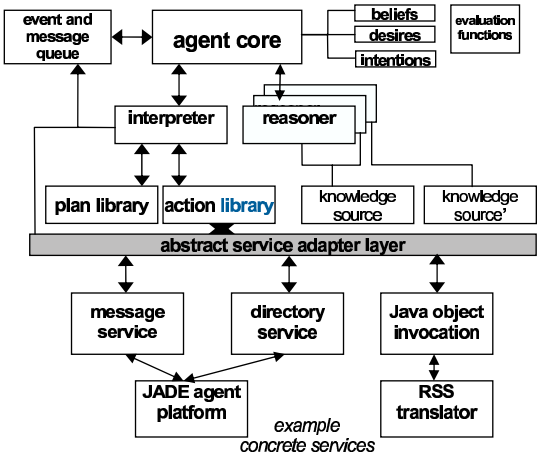

Figure 1 Nuin architecture overview

4.1 Knowledge-representation model

Nuin agents are deliberative reasoners in the tradition of first-order logic-based inference. In this section we describe our knowledge representation formalism.

4.1.1 Knowledge representation vocabulary

Knowledge structures in Nuin are composed from a vocabulary of literals: integers, reals, Booleans, strings, and symbols, together with named single-assignment variables. Ordered lists may be composed of any value, and functional terms may be composed from a symbol as functor, and zero or more arbitrary values as arguments. Two values may be unified (or fail to unify) in the standard way. Logical sentences are formed from the usual first order connectives.

An abstract syntax, using Java interfaces, provides a flexible representation, into which many surface syntaxes may be parsed. Currently implemented parsers include s-expressions (i.e. KIF-like, though not currently supporting all of KIF's definition machinery), and a Prolog-like syntax with infix operators. In addition, binary predicates are translated directly from RDF sources into the abstract syntax.

4.1.2 Knowledge-sources and reasoners

Logical sentences are stored within the agent in *knowledge-sources* (KS's). There is no commitment to any particular storage strategy: a KS may be stored entirely in-memory within the Java virtual machine, or may be stored in a persistent database. However, every KS is associated with at least one *reasoner*, which provides a set of abstract services for manipulating the contents of the KS. In particular, a reasoner may support the following services:

- **core services** – serialisation, matching, identification, query dispatching, get meta-data;
- **backward chaining** – query, query all, retry;
- **forward chaining** – add listener;
- **updateable** – assert, deny, retract.

Every KS must support the core services, noting that 'matching' denotes a simple unification pattern-match against the facts in the KS. Other services are optional. Each of these sets of capabilities

is encapsulated in a Java interface, allowing a KS to be tested at run time for the services it supports. It is also convenient to have an explicit description of the capabilities of the KS. The meta-data on a KS is an RDF model, containing statements from a pre-defined Nuin configuration ontology. Such explicit access to its own capabilities provides a convenient means for an agent to be able to make assertions about its own capabilities, perhaps during negotiations or in yellow-pages advertising. This self-modelling via RDF is a recurring theme in the Nuin design.

We want agents be able to compose multiple KS's when responding to queries or performing forward-chaining inferences. A similar requirement is addressed by Frank [10]. In Frank's approach, queries are routed to different reasoners by a special-purpose reasoner named the *dispatcher*. This implies that all complex queries must be directed to the dispatcher, which then delegates the query to a specialist reasoner, perhaps based on the predicate name of the query. Our scheme is slightly different from Frank's, since we want to partition the knowledge bases (perhaps to distinguish the agent's own knowledge, from that of its acquaintances). Thus, the two KS's may contain instances of the same predicate, which are modelling entirely different things. Rather than rely on a dispatcher to direct queries to KS's, we add a *context* parameter to the standard query interface, which encapsulates a strategy for delegating queries to other KS's .

4.1.3 Resource bounded reasoning

Predicate logic is a powerful knowledge representation language, capable of a wide range of representation tasks [21]. However, first-order reasoning is computationally undecidable. Some queries will never terminate, or consume a great deal of computing resources before terminating. This has lead many researchers to define representation systems that are weaker than full first-order logic, but which are computationally more tractable. An example is *description logics* [12], which are widely used in the ontology research community.

Our view is that practical agents will need rich representations to cope with the noisy and contradictory information available on the semantic web. We will not be able to rely on computational tractability alone to provide predictable response times and reasonable performance from our agents. We have therefore decided not to restrict our agents to reasoning over description logics, but to support resource bounds on reasoning tasks. All reasoners must be able to terminate cleanly if time or other bounds are exceeded. The agent's plans must allow for contingencies arising from exceeding resource bounds.

4.2 Agent mental states

Using the representations described above, we now elaborate the representation of agents' mental states. Each agent has three key state variables: *beliefs*, *desires*, and *intentions*.

Beliefs are modelled as sets of first-order sentences. In the current architecture, sentences do not include modal operators, so the belief modality is not represented directly. Instead, we partition the agent's knowledge sources, and label each with an implicit modality. Thus,

$\mathbf{B}_i\, p(x)$ is modelled as: $KS_{\mathbf{B}i} \models p(x)$

A similar scheme is used in the user-modelling system BGP-MS [16] for representing the user's beliefs about the system, the system's beliefs about the user, etc. If sentences containing complex compositions of modal operators are commonplace in a given domain, then this representation scheme will not be sufficient. We may, in future, extend the knowledge representation to include modal operators. For the time being, however, our hypothesis is that this folding of modalities into the KS label will be sufficient for many practical applications.

Desires are modelled as collections of first-order sentences that represent characteristics of the world that an agent wishes to bring about, or the agent's general preferences. The agent's desires are available to the choice functions in the interpreter (below). Thus an agent's desire to be helpful, or loyal to a particular user, might influence choices that it makes. In this way, we hope in future to be able to model social attitudes of agents [4], particularly with respect to human-agent interaction.

Intentions model the agent's current commitment to a course of action. In AgentSpeak(L), intentions are formed when a plan is adopted in response to an exogenous event. The intention then provides a scope in which variables may be bound as plans and sub-plans are executed. We adopt a similar view. Our intentions are explicitly triggered by various conditions, and provide a computational environment for the execution of actions and sub-plans .

Events In addition to the above mental states, each agent maintains two ordered lists of events. The first models the agent's sensing of the world: all perceptions are delivered as events. The second models the agent's memory of recent percepts. This is a fixed-length chronological queue of the *N* most recent events (*N* is a positive integer from the agent's configuration). Maintaining a recent-event history allows agents to trigger behaviours on patterns of chronologically correlated events.

4.3 Interpreter and processing model

We now outline the processing model for the agent. Recall that all components of our architecture are pluggable using the interface-based programming pattern described above. Therefore, the behaviours described here may be considered the default or built-in behaviours, any of which may be extended by the programmer.

4.3.1 Events

All of the agent's perceptions of the environment are delivered as events. These may be genuinely exogenous occurrences, such as a user instruction, a message from another agent, or a sensor value. Following AgentSpeak(L), endogenous events are also used as a uniform abstraction for managing control flow within the agent.

An event is represented as a logical term, denoting the event type, and optional arguments. An *event pattern* is a Boolean expression formed from event terms and the predicates *on* and *after*:

- `on` *E*
  is true if E is a term that unifies with the term representing the most recently observed event;
- `after` *E*
  is true if E is a term that unifies with any event term in the agent's event history, or the most recent event.

Thus, moderately complex but computationally tractable triggering conditions may be straightforwardly defined.

4.3.2 Plans and actions

Plans are the key abstraction defining the agent's behaviour. A plan minimally is an action expression, together with either a predicate representing its post-condition, or a triggering pattern representing the conditions under which the agent will perform the

action expression. A plan may have both a trigger and a post-condition, but it may not have neither. In addition, a plan may be named (with a URI), take arguments, have a comment or have a priority.

Action expressions are composed from atomic *actions* and *tests*, together with operators for sequencing actions (α | α' means perform action α followed by action α'), and non-deterministic choice (α ; α' means perform either action α or action α'). Non-deterministic choice records a *choice point* in the evaluation of the action expression. Choice points may be backtracked through, providing no side-effecting action has been performed. Once a side-effecting action is performed, all of the open choice points in the plan are collapsed. This is because we assume that such actions are not, in general, reversible, and therefore it is not valid for evaluation to continue down a different branch from the choice point once the external environment has changed. Each action definition determines whether that action is side-effecting or not. By default, atomic actions are assumed to be side-effecting, tests are not.

A number of standard actions are built-in to the standard script parser and interpreter. However, since each action is represented as an instance of a Java class implementing the `Action` interface, it is easy for the programmer to define new types of action and invoke them from the agent's script.

Due to lack of space, we can only list, but not define, some of the built-in actions. They include `achieve`, `add intention`, `add desire`, `drop intention`, `assert`, `retract`, `suspend`, `resume` and `send`. Tests include `holds`, `on` and `after`.

4.3.3 Interpreter

Each agent has one interpreter that will process the events from the agent's environment, and, in conjunction with the agent's plans and other mental states, determine the agent's behaviour. Once again, the interpreter is a configurable object, defined as a fixed interface and a default implementation providing the standard behaviours described here.

In essence, the interpreter acts similarly to the AgentSpeak(L) interpreter ([18], figure 1), and to Wooldridge's abstract agent interpreter ([28], figure 2.7). The key steps for the Nuin interpreter are shown in Figure 21 below.

At each interpreter cycle, the agent must select its focus for that time step. The first choice is whether to respond to an incoming (queued) event, or whether to continue pursuing a current intention. There is no general solution to the right choice to make at this point. Some agents will benefit from being highly *responsive*, and choosing as their focus any percept as soon as it is detected. Other agents will be better to be less *distractible* and ignore events for the sake of completing the current plan. Clearly some events will be more ignorable than others. We could fix this choice in the architecture, as some agent interpreters do. Our preference is to make this strategic decision one that agent designer should control. Therefore, the interpreter contains a scriptable *choice function* for selecting the current focus at each step. The default choice is to be responsive, and always process events in preference to current intentions.

```
while true do
  f = select-focus(B, D, I, Ei)

  if f is new-percept
    add percept to history
    let p* = plans-triggered(f)
    if not empty p*
      for each p ∈ p* do
          add new intention-to p
    endif
  elseif f is active-intention
    let α = next action of f
    record choice point if backtrackable(α)
    if side-effecting(α)
      commit
    endif
    perform α
    case
      failed(α) → backtrack
      completed(α) → remove α from f
    endcase
  endif
endwhile
```

Figure 2: Interpreter cycle

If there is no current event in the queue, or the agent chooses not to process an available event, there is then the choice as to which intention to pursue on this iteration. There are several methods here also. The agent could choose to process the intention that it regards as most advantageous (i.e. has the highest utility), given its current beliefs. Alternatively, it could choose to employ a scheduling algorithm, such as round robin or priority ordering, to ensure that intentions are processed in a suitable order. Again, there is no general solution, so we make the intention selection function configurable.

In general, an agent is permitted to have a number of simultaneous intentions that it is pursuing. Any domain constraints on such commitments, such as being unable simultaneously to move in different directions, must be imposed by the agent designer. In effect, this changes the control structure of [28], fig 2.7, so that instead of having to decide when to *reconsider*, the agent must decide how to schedule multiple intentions.

Our percept selection function corresponds to the function S_E in AgentSpeak(L). We do not need Rao's function S_O since the interpreter can backtrack. One choice that must be accounted for is the agent's decision about which course of action to take to achieve a given postcondition when backward chaining. The PRS interpreter can recurse to meta-level planning (i.e. using the agent's current mental state to reason about plans to invoke). While this has a certain mathematical elegance, we feel that it may make for overly complex agent programs. Currently, where alternative plans are applicable in Nuin, plan selection effectively occurs in a Prolog-like backtracking search. We intend to add a scriptable evaluation function for selecting between alternative, valid, courses of action in a future version.

Note that in Wooldridge's abstract interpreter, there is a function *brf()* that updates the agent's beliefs given a percept. In Nuin, we delegate all revisions of the agent's mental states to actions executed by the interpreter. Thus there is no requirement to have a separate belief revision function in the interpreter loop. Plan actions include making assertions into any of the agent's KS's, adding and dropping intentions explicitly or implicitly (by

attempting to achieve a given post-condition), and adding and dropping desires.

4.4 Operational details

In the preceding section, we outlined the operation of the abstract interpreter. In this section, we briefly discuss some of the operational details of our platform.

4.4.1 Agent configuration

Given that we have an objective to make Nuin agents highly flexible and adaptable, we must provide some means of configuring a given agent prior to its operation. Consistent with the use of semantic web technology, Nuin agent configuration is specified using an RDF model. This is typically expressed as an RDF document with a resolvable URL. The RDF document is fetched when the agent starts, and used to configure the agent's services, scripts and initial knowledge.

We have defined an ontology to represent the various configuration options an agent may take. Options from the configuration model are passed through to any objects created, allowing fine-grain control over objects' behaviour. Java reflection may also be used to directly instantiate custom elements of the agent implementation corresponding to the public interfaces (see §3.2).

Since the agent needs only the URL of the configuration model to start up, it is easy to programmatically create agents in the context of other applications. This, we believe, is more consistent with application designers' needs, rather than being forced to fit their application logic into an agent framework.

4.4.2 Services

While a single agent can be a useful abstraction in an application design, perhaps as locus for advanced user-facing affordances, the agent metaphor is perhaps most closely associated with *multi-agent systems (MAS)* [26]. To participate in an ecosystem of autonomous, distributed agents, agents require access to a variety of key services. These include name resolution, messaging, migration, etc.

There are many tools available to assist with fulfilling these requirements. This is, in part, due to an increasing emphasis on distributed systems in general, encouraged by the success of the WWW. To allow the Nuin agent platform to operate in a MAS context, we want to provide our agents with access to these multi-agent services. However, the plethora of competing solutions presents the problem of which to choose.

One solution would be to commit to the standards defined by FIPA [8]. While the FIPA abstract architecture (FAA) [9] does capture some of our requirements, it is not *a priori* clear that all application designers will take the FAA as the starting point for their system designs. It is clear, however, that there is extensive innovation in distributed-systems in general, some of which our agents may wish to take advantage of. For example, peer-to-peer message passing is an increasingly well-studied technology, for which a number of high-quality implementations exist.

Our solution is to identify the key underlying services that our agents rely on, and package these as abstractions that may be instantiated in different ways. We build upon the research embodied in the FAA by re-using the names and abstractions from that standard where appropriate (though we map the names to URI's *per* RDF).

We define an abstract *service* that an agent has access to. The agent also has access, via the KS, to first-order assertions about the available platform services. A general service-invocation procedure is available as one of the built-in actions in the interpreter. Furthermore, some well-known services are more closely integrated, and supported with special-purpose actions. An example is *message sending* (see below).

4.4.3 Interoperation with agent middleware

The existence of the FIPA standards has encouraged the development of a number of freely available implementations of FIPA platform services (e.g. Jade [2]). Since such platforms exist precisely to provide the agent middleware services we discuss above, and given that we do not want to re-implement functionality that is already available, we aim to rely on FIPA platforms to provide the necessary platform services. Specifically, we provide *service adapters* that map between the abstract services in our architecture, and the capabilities of the host FIPA platform. In principle, this should be possible for a variety of Java-based FIPA platforms. To date, we have only investigated hosting our agents on the Jade platform.

Alternatively, it may be that some application designers will want to base their system-level architecture on interoperation standards other than the FIPA agent middleware. In a web-services deployment, interactions are typically based on HTTP message transport, with XML payloads (e.g. SOAP [24] or XML-RPC [27]). Assuming that an appropriate binding to the abstract services can be defined, there is no reason why a web-services architecture should not provide a suitable basis for multi-agent operations with Nuin. This is not something we have yet investigated, however.

4.5 Message passing

The dominant metaphor for inter-agent communications is *message passing* (in contrast to, say, remote procedure call). Message passing is well suited to the view that agents have autonomous control over their own behaviour. Agent communications languages such as KQML [17] and FIPA-ACL [7] provide standard encodings for messages.

Architecturally, message passing (and its ancillary services) are just another of the abstract services that an agent designer may wish to use. However, due to the ubiquity of message-passing in agent systems, we include built-in actions that directly invoke the messaging and directory services.

4.5.1 Abstract messaging model

`Message` is a sub-class of `Event`. A message has zero or more named *attributes*, corresponding to the fields of the encoded message structure. These include the `to` and `from` agent ID's, `ontology`, `content`, `reply-with`, etc. Note that addresses are only ever agent identifiers. Resolution of names to transport bindings (including the transport mechanism and message encoding) are handled transparently by the messaging service.

The *messaging service* provides the operations to create a new or reply message, encode and send a message, suspend until an expected message arrives and extract content sentences from incoming messages. The *directory service* provides both yellow-pages and white-pages registration and name resolution operations. We currently make no assumption about the ability of directory services to federate, or perform distributed queries.

The implementation of these services is largely delegated to the underlying platform (e.g. Jade), with a thin layer of adapter code to map between the two conceptual models.

Parenthetically, we note that we had to define our own ontology in DAML+OIL of the various terms in the FIPA ACL standard. We hope that FIPA will provide its own, definitive versions of these ontologies in due course.

4.6 Examples

To illustrate the syntax of the default scripting language, we show in Figure 2 a translation of the example program from Rao's AgentSpeak(L) paper. Note that plans may be named or anonymous, and that all names are URI's in the default namespace, unless given explicit prefixes.

As a second, again very simple, example, Figure 3 shows a basic plan for automatically booking a restaurant table near to the theatre if the user requests the agent to book theatre tickets. This plan depends on an ontology of dining concepts, referenced by the URI prefix `dining`. Because these queries have a distinguished namespace, they can easily be routed to the appropriate knowledge source. As background to the plan fragment shown, a suitable DAML+OIL ontology might define an instance `SpiralGateCafe` to be an instance of `VegetarianBistro` – a sub-class of both `dining:Restaurant` (a class of eating establishments) and `dining:Vegetarian`. (a class of meat-free food providers). Thus, taxonomic reasoning is used to entail additional beliefs.

5 EVALUATION

The central objective of this phase of our work is to design and build a practical toolkit for developing deliberative agents for Semantic Web applications. Our emphasis is on the higher-level behaviour of the agents, rather than infrastructure issues, and on the engineering aspects of constructing a stable, flexible and extensible platform. While this is an ongoing project, we can make a preliminary assessment of these factors by comparing the Nuin platform to related projects in this field. There are other tools that address processing information on the Semantic Web. The Jena toolkit [13] provides programmatic access to RDF sources, ontology documents, ontology reasoning and RDF query. However, Jena provides only low-level access to RDF data. The BDI metaphor and associated tools provides much higher level abstractions for defining useful end-user services that make use of RDF data. To build upon the useful tools provided by Jena, Nuin can use a Jena RDF or ontology model as a knowledge source. This provides Nuin agents access to a wide range of Semantic Web data, including RDF stored in persistent databases or remotely accessed via HTTP.

Huber's JAM [14] is a Java re-implementation of the U.Michigan C++ implementation of PRS. JAM is therefore BDI-based. It does not, however, address Semantic Web reasoning, nor does it integrate with agent middleware platforms (JAM's design pre-dates both FIPA and the emergence of the Semantic Web). We claim that we also have a coherent BDI implementation, grounded in BDI theory, but that our implementation significantly more flexible and extensible than JAM. To illustrate this, consider adding defining an agent that handles RSS [20] data streams. RSS is an asynchronously updated series of meta-data descriptions, each of which describes a publication event. RSS descriptions are

```
use default <http://hpl.hpl.com/nuin-demo#>.

plan
    trigger
        on location( waste, ?x )
    do
        holds location( robot, ?x ) &&
              location( bin, ?y );
        perform pick( waste );
        achieve location( robot, ?y );
        perform drop( waste )
end.

plan move1
    postcondition
        location( robot, ?x )
    do
        holds location( robot, ?x )
   end.

plan move2
    postcondition
        location( robot, ?x )
    do
        holds location( robot, ?y ) &&
              not ?x == ?y &&
              adjacent( ?y, ?z ) &&
              not location( car, ?z );
        perform move( ?y, ?z );
        achieve location( robot, ?x )
end.
```

Figure 3: simple robot navigation example (from [18])

```
plan bookRestaurant
  trigger on bookTheatre( ?t, ?d, ?n )
  do
    holds for some business(?r)
      holds
        rdf:type( ?r, dining:Restaurant ) &&
        rdf:type( ?r, dining:Vegetarian ) &&
        nearTo( ?t, ?r ) in mapDistances;
      invoke bookTable( ?r, ?d, ?n )
    end;
    println "Booked ?r on ?d for ?n people"
end.
```

Figure 4: Plan for booking a restaurant

encoded in RDF. It is not clear how to integrate RSS handling into JAM's architecture without significant programming. In Nuin, we would define an RSS service, that would place an event into the interpreter's input queue when an RSS item is updated. The event could either contain the RSS item data, or just a URI for retrieving the data from an RDF KS. Forward-chaining plans would trigger on these events allowing the agent to respond appropriately.

Currently available FIPA platforms generally do not provide sophisticated reasoning capabilities. JADE, for example, allows agent designers to code agent behaviours using forward production rules, finite state machines, or as custom Java classes. While useful, these tools do not correspond to a developed agent theory such as BDI. If the agent designer wishes to define an agent that can be delegated goals by a user, there is no built-in knowledge representation in Jade that would assist directly to do so. The FIPA platforms do have significant strengths in providing and managing infrastructure services. Nuin makes use of these services by supporting service adapters modelled on the FIPA Abstract Architecture.

6 CONCLUSIONS AND OPEN ISSUES

We have outlined the Nuin agent platform: an open, extensible platform for developing intelligent agents for Semantic Web applications. The essence of our approach is to take the principles of multi-agent and autonomous agent theory, combined with strong adherence to the principles of good software engineering practice, to create a practical tool for agent designers. Our primary motivation in creating this platform has been to provide a basis for our own research into user interaction with Semantic Web agents. However, the software will also be available under an open license for other research groups to use.

There are many difficult issues that require further research to meet our ambitions of a practical toolkit. Two central themes for our ongoing work on this platform are: how to extend the reasoning capabilities of the agents, and how to facilitate effective co-operation between agents with strong mental states and human users. A particular interest with respect to Semantic Web agents is to explore the relationships between the strong, but theoretically intractable, reasoning of BDI agents and the weaker, computationally tractable, reasoning embodied in OWL DAML+OIL – particularly in a open, changeable environment such as the Semantic Web.

7 Acknowledgements

The authors would like to thank the anonymous AAMAS'03 reviewers for their constructive and detailed comments. Thanks also to Dave Reynolds and Steve Cayzer of HP Labs for their comments on earlier drafts.

8 REFERENCES

1. The DARPA Agent Markup Language (DAML+OIL). 2001. http://www.daml.org
2. Bellifemine F., Poggi A. & Rimassa G. Developing Multi Agent Systems With a FIPA-Compliant Agent Framework. Software Practice and Experience. Vol. 31:2. 2001. pp. 103–128.
3. Berners-Lee, Tim, Hendler, James, and Lassila, Ora The Semantic Web. Scientific American. 2001.
4. Castelfranchi, C. Modeling Social Action for AI Agents. In: Proc. 15th International Joint Conference on AI (IJCAI 97). 1997. pp. 1567 – 1576.
5. Coad P, Mayfield M. Java Design : Building Better Apps & Applets Prentice-Hall, 1998.
6. emorphia. FIPA-OS agent platform. 2002. http://fipa-os.sourceforge.net/
7. FIPA. FIPA ACL Message Structure Specification. (XC00061) 2000. http://www.fipa.org/specs/fipa00061/
8. Foundation for Intelligent Physical Agents (FIPA). 2001. http://www.fipa.org
9. FIPA. Abstract Architecture Specifiation. 2002. http://www.fipa.org/specs/fipa00001/
10. Frank, G. A General Interface for Interaction of Special-Purpose Reasoners Within a Modular Reasoning System. In: Proc. Question Answering Systems. AAAI Press, 1999. pp. 57–62.
11. Gamma E, Helm R, Johnson R, Vlissides J. Design Patterns Addison Wesley Longman, 1994.
12. Horrocks I. Reasoning With Expressive Description Logics: Theory and Practice. In: Andrei Voronkov, (ed) Proc. 18th Int. Conf. on Automated Deduction (CADE-18). Springer Verlag, pp. 1–15, 2002.
13. HP Labs. The Jena Semantic Web Toolkit. 2002. http://www.hpl.hp.com/semweb/jena-top.html
14. Huber, M. JAM: a BDI-Theoretic Mobile Agent Architecture . In: Proc. 3rd Int. Conf. on Autonomous Agents. ACM, 1999. pp. 236–243.
15. Kagal, L., Perich, F., Chen, H., Tolia, S., Zou, Y., Finin, T., Joshi, A., Peng, Y., Cost, R. S., & Nicholas, C. Agents Making Sense of the Semantic Web. In: Proc. First GSFC/JPL Workshop on Radical Agent Concepts (WRAC).
16. Kobsa A. & Pohl W. The User Modeling Shell System BGP-MS . User Modeling and User Adapted Interaction. Vol. 4:2. 1995. pp. 59–106.
17. Labrou, Yannis and Finin, Tim. A Proposal for a new KQML specification. Computer Science and Electrical Engineering Dept, University of Maryland. 1997. http://www.cs.umbc.edu/~jklabrou/publications/tr9703.ps
18. Rao, A. AgentSpeak(L): BDI Agents Speak Out in a Logical Computable Language. In: Proc. 7th European Workshop on Modelling Autonomous Agents in a Multi-Agent World (MAAMAW '96). Springer-Verlag, 1996. pp. 42–55.
19. Rao, A. & Georgeff, M. BDI Agents: From Theory to Practice. In: Proc. First Int. Conf on Multi-Agent Systems *(ICMAS-95)*. 1995.
20. RSS-Dev Working Group. Rich Site Summary (RSS) 1.0 Specification. 2001. http://purl.org/rss/1.0/spec
21. Sowa J. Knowledge Representation: Logical, Philosophical, and Computational Foundations Brooks Cole, 1999.
22. SRI. The Open Agent Architecture. 2002. http://www.ai.sri.com/~oaa/main.html
23. W3C. Namespaces in XML. 1999. http://www.w3.org/TR/REC-xml-names/
24. W3C. Simple Object Access Protocol (SOAP) 1.1. 2000. http://www.w3.org/TR/SOAP/
25. W3C. Web Ontology Working Group. 2002. http://www.w3c.org/2001/sw/WebOnt/
26. Weiß G. (ed.). Multiagent Systems. A Modern Approach to Distributed Artificial Intelligence MIT Press, 1999.
27. Winer, D. XML RPC. 1999. http://www.xmlrpc.com/
28. Wooldridge M . Reasoning about rational agents MIT Press, 2000.
29. Wooldridge M. & Jennings N. Intelligent Agents: Theory and Practice. Knowledge Engineering Review. Vol. 10:2. 1995. pp. 115-152.
30. World Wide Web Consortium (W3C). The Resource Description Framework (RDF). 1999. http://www.w3.org/TR/REC-rdf-syntax/