Incorporating BDI-Agent Concepts into Object-Oriented Programming

Berndt Müller and Jack Betts

University of South Wales
{bertie.muller,jack.betts}@southwales.ac.uk

Abstract. In 1987 Bratman introduced the model of reasoning on the grounds of beliefs, desires, and intentions (BDI) to mimic important aspects of human reasoning. This view has since been adopted as one of the major paradigms in agent-oriented modelling and agent programming. A number of dedicated agent programming languages have been developed and some of them are successfully used in niche areas. Despite the common BDI foundation, little work has been undertaken in unifying APLs or making them interoperable. There has been much debate as to why agent and multi-agent programming has not had more of an impact on software development in general. To overcome the apparent neglect of agent-oriented programming, we discuss some issues and develop solutions for incorporating agent-concepts in 'traditional' programming-language projects. By traditional we mean the well-established object-oriented programming languages that are mostly evolved as extensions to procedural programming languages. The examples we give use C++ but the techniques are not limited to this programming language.

1 Introduction

The development of an agents' deliberation cycle in a general programming language usually requires starting from a blank page. This 'reinventing of the wheel' not only increases the time taken to complete a system it also leads to the system having to be thoroughly tested before it can be deployed. Once the agent system has been developed as a bespoke system [10, p.234] attempts at implementing in other environments may prove to be unsuccessful. The area of Software Engineering is still learning from Hardware Engineering methodology in the way products are designed and developed. When hardware is being designed and engineered, capacitors and integrated circuit boards are not reinvented for every project. Instead they use readily available components to develop the hardware. These components have been tested and are known to work. These commonly available components are referred to as "off the shelf" components.

"In the hardware world, component reuse is a natural part of the engineering process. In the software world, it is something that has only begun to be achieved on a broad scale." [9, p. 7]

Wooldridge also states that many agent systems projects fail "because basic software engineering good practice was ignored." [10, p.234] Agent systems development has not yet achieved component based design as no 'off the shelf' agent components are available. The LAF (Layered Agent Framework) Project is one attempt at providing a component based approach to developing in agent in the form of layers and modules [2]. The possibility of agent developers being able to find 'Deliberation Modules' or 'Knowledge Storage' code and have it integrate with their LAF based project has the potential to allow rapid development and deployment of agent systems. For this paper we will focus on an agents' deliberation cycle to demonstrate how well established component based design could assist agent development. We have chosen the C++ language in our example due to the object orientated features it provides such as polymorphism, inheritance and abstraction.

2 The BDI Paradigm

In 1987 Bratman [5] introduced the model of reasoning on the grounds of beliefs, desires, and intentions (BDI) to mimic important aspects of human reasoning. This view has since been adopted as one of the major paradigms in agent-oriented modelling and agent programming. A number of dedicated agent programming languages have been developed [4] and some of them are successfully used in niche areas. Despite the common BDI foundation, little work has been undertaken in unifying APLs or making them interoperable [8]. There has been much debate as to why agent and multi-agent programming has not had more of an impact on software development in general (Dagstuhl Seminar *"Engineering Multi-Agent Systems"*, 2012).

To overcome the apparent neglect of agent-oriented programming, we discuss some issues and develop solutions for incorporating agent-concepts in 'traditional' programming-language projects. By traditional we mean the well-established object-oriented programming languages that are mostly evolved as extensions to procedural programming languages. The examples we give use C++ but the techniques are not limited to this programming language.

2.1 Beliefs

An agent's *beliefs* represent the subset of information about the universe of discourse that the agent has acquired and has adopted as their 'knowledge'. Beliefs are inherently subjective to a particular agent and there is no guarantee that the beliefs are all true or accurate. This might stem from information that is out of date but remains a belief since there has been no perceived update since the information gained belief status. To keep beliefs accurate, the agent has to engage in perception of its environment.

Mathematically, beliefs are usually represented by a set of predicates about instances of the universe. A belief set is a finite set of predicate schemas that can denote an infinite set of ground terms. We denote a belief set by

$$B = \{b_1, \dots, b_n\}$$

where n is a non-negative integer and b_i with $i \in \{0, \dots, n\}$ are predicate schemas.

In programming, apart from the the intuitive representation of a belief set as a set data structure in a programming language, there are reasons to consider external databases for the storage and manipulation of beliefs. From a programmer's perspective, the actual implementation should not make any difference to the design and code of the actual agent program. We encapsulate these low-level design decisions in building blocks that act as interfaces.

2.2 Desires

Philosophically, an agent's desires are aspects of the state of the universe that the agent would like to see accomplished. A desire can influence the agent's actions, but is not in itself a set of actions. There is no requirement that all desires can be accomplished at the same time, indeed it might be the case that none of the desires can be brought about. An agent might have to choose which of their desires to pursue when faced with mutually incompatible desires.

Like beliefs, Desires would mathematically be given as a set of predicates describing the 'desired' properties within the universe.

$$D = \{d_1, \dots, b_m\}$$

where n is a non-negative integer and b_i with $i \in \{0, \dots, m\}$ are predicate schemas.

For programmer's, similar low-level options exists but should be oblique at the agent level. Again, using building blocks for the reasoning can allow the programmer to interact with interfaces hiding the actual data structures.

2.3 Intentions

A non-idle agent will work towards some goals. The current goals are determined by the *intentions* the agent has decided to work on. Execution according to intentions can spawn new goals that will usually trigger the creation of further intentions. Intentions exist concurrently and can be created or removed during the deliberation process. Intentions are very closely connected to plans and as such are subject to change. Whereas [5] considers intentions to "resist reconsideration and in that sense have inertia", [3] defines intentions in terms of beliefs and desires, and . This view is taken from *decision theoretic planning* (DTP) and is essential for creating resource-bounded agents. Resource-boundedness is a requirement for tractable programming and verification as shown in [6, 7].[1]

Programmatically, for each intention, there will usually be an intention stack that the agent has to work on sequentially. To guarantee an acceptable run-time behaviour, the costs a plan in terms of all required resources has to be taken into account both for the aim of achieving a goal and for building the plan [3].

2.4 Updating the Sets

Updating the sets, in particular the belief set, involves perception of changes in the environment, execution of actions, and decision-making about intensions. This process is usually performed in a so-called deliberation cycle. Agent programming language interpreters are at the core implementations of a working deliberation cycle, an example of which is shown in Figure 1.

Deliberation does not necessarily have to be carried out in the exact sequence shown in Figure 1. E.g., the lack of an applicable plan would lead to a 'short cut' in the cycle. also, a permutation of the order of actions/events might be desirable. Hence, in the next section we propose re-usable building blocks for the construction of custom deliberation cycles. These blocks can be plugged together by the programmer to accommodate any particular needs.

[1] Theoretically, intentions can be modified during the deliberation, but this is not readily supported by most APLs.

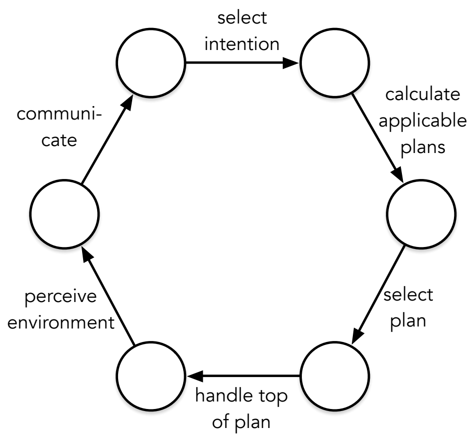

Fig. 1. A typical deliberation cycle of an agent programming language interpreter

3 Agent Software Components

Before we can discuss how having agent software components would benefit the agent development field we must first consider what is required for this vision of agent components. This section lists some requirements that must be met before the field can begin to develop agent software components.

3.1 Software design and development standard(s)

Standardised APIs (Application Programming Interface [1]) are essential for software components to achieve a high level of reusability. Standards are also required, an example of such a standard is a component must clearly state its behaviour. Another example is that all components must fail gracefully without crashing the host system. However defining these APIs and component standards in full are far beyond the scope of this paper and will require the support and collaborative effort of the agent systems community.

3.2 Proof that a components works, and details on how it is to be used

When a component is developed whether it a software or hardware component, it is tested to check if it is 'fit for purpose' [9, p.428]. When the component is tested these results can be shared and thus more developers will trust the component (assuming the component is open source). Once a system has been developed using these components moving towards a standard will become achievable.

3.3 The APIs and any standards must be made free and available for all

Any standards and APIs that are developed to enable agent software components must be made freely available. Components that are developed using these standards and APIs are not required to be made freely available to the community unless the author allows this.

3.4 Security and Integrity of Components

All components that are made available (even through sale) must have a form of digital checksum on the component to ensure data integrity. This is just one measure that could be taken to combat tampering, in which a component may be modified for malicious intent. The potential security risk a component poses may be very real for certain agent systems, especially if the application of this system is military use. Any component must have some functionality to request access to elements of the host system as well as a method in which the component can be tested in isolation in order to monitor its activity.

4 Using OOP to assist agents developers

C++ provides many tools for Object Oriented Programming and Design. We hope that such design practices will become commonplace within the agent development community. Until then we will share how we have used OOP to improve agent development within the LAF. For this paper, we share the concept of Simplicity through hidden expected functionality.

4.1 Simplicity through hidden expected functionality

Hiding functionality from developers at first would appear to be counter-intuitive in making a system easy to develop for. This holds true in most scenarios where a developer may not be fully aware of what the system is doing when parts of that systems functionality is hidden. In some situations hiding expected functionality may make the system easier to work with. What we mean by expected functionality is a certain behaviour or aspect of a system that most, if not all, developers would implement in a similar way and therefore be able to identify this aspect if seen in another system. Functionality like ID number assignment and event handling would be implemented by most developers in a similar fashion. With LAF components these two examples of expected functionality are dealt with in the background so that the developer better focus their time on creating an agent rather than “laying the groundwork” so to speak. Functionality that is required of a given class for it to work within the LAF is implemented and when possible this functionality is automated. A basic demonstration of this can be found within the process of creating a temperature sensor within the LAF. A sensor requires the following information to be valid in order to operate within the framework[2]:

- ID Number
  - Used for monitoring and debugging
- Name
  - Provides a human friendly identifier
- Type of Sensor
  - Allows categorisation of sensors and assumptions about the sensor capabilities
- Time of last update
  - Used by the framework to detect non-responsive components
- Sensor start time
  - Used by the framework to monitor reliability and management component stress

Now for a developer to manage all of this data for each instance of a sensor will likely add to the development time with this system. If 10 sensors needed to be created with all these variables set it would look like:

```
time_t start_time = time(0);
std::vector<sensor> sensorCollection;
sensorClassification type = sensorClassification::↩
    THERMAL_DETECTION;
std::string name = "Sensor";
for (int i = 0; i < 10; i++){
    std::string sensorName = name;
    sensorName+=('0' + i);
    sensorCollection.push_back(sensor(i, name, type, ↩
        start_time)); }
```

[2] This is for the current limited prototype and can easily be extended in future.

One clear issue with this example code implementation of instantiating sensors is that invalid values (such as a duplicate ID) can be passed to the sensor. This could be done and without any safeguards implemented against this could lead to the system behaving incorrectly or not functioning at all. The developer would have to implement these safeguards themselves which would require extra development time. Having to setup and pass four arguments for each sensor can prove cumbersome when compared to how the LAF allows sensor implementation.

```
std::vector<sensor> sensorCollection;
std::string name = "Sensor";
for (int i = 0; i < 10; i++){
   std::string sensorName = name;
   sensorName+=('0' + i);
   sensorCollection.push_back(TempSensor(sensorName););  }
```

Currently instantiating sensors within the LAF only requires a name in the form of a string. This was achieved by having variables such as ID and Type of sensor handled by the parent constructors of the sensor hierarchy. Going from the base interface of `ISensor` to the sensor definition class `TempSensor` to finally the implementation of a `TempSensor`. Each step completes more of the information required by the system. The previous example were for comparison purposes, when looking at how a temperature sensor (see Fig. 2) is added to a LAF based system we gain some insight into how the expected functionality is hidden from the developer while making development easier.

```
_pInternalTempSensor = std::unique_ptr<TempSensor>(
   new TempSensor("Internal Temp Sensor 0001"));
```

The code snippet above shows a `unique_ptr` of type `TempSensor` called `_pInternalTempSensor` which is then assigned a new TempSensor. We are using a smart pointer to assist with memory management. When looking at the code for the constructor for the temperature sensor we begin to see this hidden expected functionality. Here the constructor is not only passing the sensor name to the sensor interface but also passing the sensor classification for the sensor as `THERMAL_DETECTION`.

```
TempSensor(const char* sensorName)
   : ISensor(sensorName, THERMAL_DETECTION){ temp = -6.0; };
```

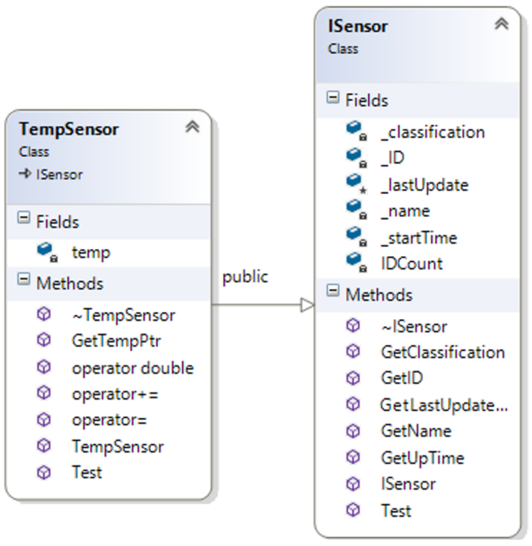

Fig. 2. UML diagramme for a temperature sensor

A developer would be required to create these sensor definition classes using the sensor interface provided by the LAF to access this functionality.

At the sensor interface level we see the constructor fills in the last pieces of information we need. An ID number is assigned to the sensor. This ID number assignment is achieved through use of a private static 64bit integer stored within ISensor disallowing any objects but ISensor from modifying the current ID counter.

```
ISensor(const char* sensorName, sensorClass ↩
    Sensorclassifiction)
: _name(sensorName), _ID(IDCount++), _classification(↩
    sensorClassifiction){};
```

The concept of Hidden Expected Functionality has be utilised within the sensor implementation code to assist the developer. The addition of new temperature sensors using this concept requires the assignment of a name and nothing else, saving the developer time and reducing the chance of an incomplete or invalid object. ID number, classification and other system critical variables are instantiated within the sensor interface and sensor definition class.

5 BDI Blocks for Custom Deliberation Cycles

Figure 3 shows the general architecture of our deliberation cycle framework.

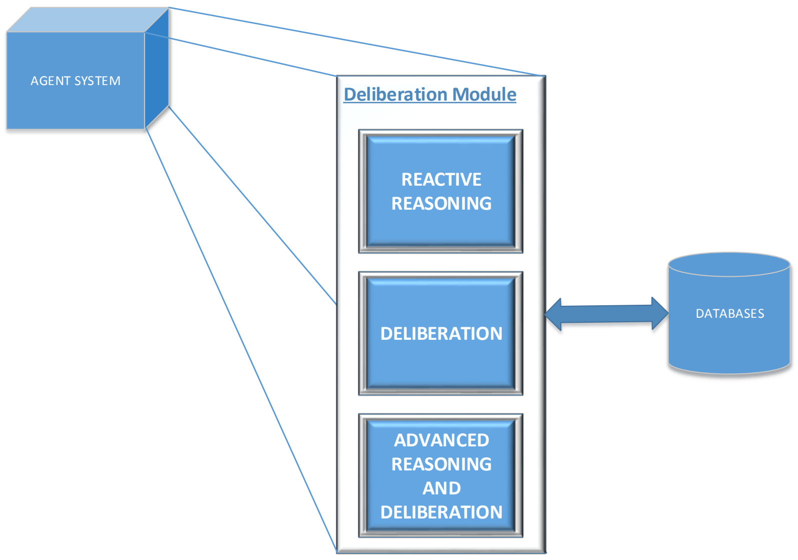

Fig. 3. Schematic of the deliberation modules made available to the programmer

Moving forward with the idea of component based design we propose a BDI module that comprises of components we label 'blocks'. These blocks represent the basic steps within a standard deliberation cycle within a BDI based system. Containing them within blocks we believe will reduce development time as blocks can be reused in other projects or even removed. If an agent does not need to perceive in an advanced way (filter or sort incoming data) this step can be ignored and the raw data passed onto the next step. We now list some of the blocks we are developing for this component based BDI code. These blocks are subject to change in terms of exact functionality and structure but the general descriptions should remain the same.

5.1 Block Descriptions

The framework comprises the following blocks.

Environment Perception Block: Handles how the agent will filter perception data and store it. This block can also be configured to prioritise data or even apply some pre-processing to the data to reduce noise before storage. This block generally passes feedback of the perceived environment to the Intention Block.

Communication and Agent Network Synchronisation Block: Handles how the agents will communicate with other agents and/or devices. This block will also be tasked with what information the agent shares on the network along with what information the agent will take from the network.

Intention Prioritisation and Selection Block: This block defines how the agent will select intentions based on perception feedback, this includes prioritisation and intention conflict. An intention stack is then created by the block in which highest priority intentions are at the top and least important at the bottom. This block does not need to be run every deliberation cycle and can be configure to only run at defined intervals or for certain detected events (such as emergency situations).

Plan Processing and Selection Block: This block will not only be responsible for selecting the plan most appropriate for the selected intention. This block will also monitor the progress of the plan, selecting new plans if required. This block will also need to be aware of when intentions changes and therefore select new plan(s). Intentions are dealt with in a serial manner from highest priority first.

We aim to have it so blocks can be developed and tested in isolation to the BDI module. This will be achieved through the use of C++ interface classes dictating minimum functionality required of a block along with access methods for the module to use. Once developed tests can be run on the BDI module and the blocks attached to it in order to show the potential benefit of component based design within agent systems.

6 Outlook

As we move forward with the design and development of agent software components the benefits of component based design are already apparent. When a component of the LAF is faulty or has been made redundant, it can be removed without the system crashing or requiring a replacement in order to operate (for the most part). With every component automatically assigned an ID number tracking and debugging components becomes easier when compared to every component needing to be manually assigned an identifier. Our next objective is to finalize the BDI block interface, develop some blocks for testing and investigate the possible usefulness to the agent community this BDI module could have.

References

1. API - application program interface. Webopedia, 2014. http://www.webopedia.com/TERM/A/API.html, Last checked: 7 July 2014.
2. J. Betts and B. Müller. Engineering MAS – a device integration framework for smart home environments. In *Proceeding of CS&P 2013*, pages 15–26, 2013. Revised version to appear in Fundamenta Informaticae (2014).
3. Guido Boella. Intentions: choice first, commitment follows. In *AAMAS*, pages 1165–1166. ACM, 2002.
4. Rafael H. Bordini, Lars Braubach, Mehdi Dastani, Amal El Fallah-Seghrouchni, Jorge J. Gómez-Sanz, João Leite, Gregory M. P. O'Hare, Alexander Pokahr, and Alessandro Ricci. A survey of programming languages and platforms for multi-agent systems. *Informatica (Slovenia)*, 30(1):33–44, 2006.
5. M. E. Bratman. *Intention, Plans, and Practical Reason.* Harvard University Press, Cambridge (MA), 1987.
6. Nils Bulling and Berndt Farwer. Expressing properties of resource-bounded systems: The logics RTL^* and RTL. In Jürgen Dix, Michael Fisher, and Peter Novák, editors, *CLIMA*, volume 6214 of *Lecture Notes in Computer Science*, pages 22–45. Springer, 2009.
7. Nils Bulling and Berndt Farwer. On the (un-)decidability of model checking resource-bounded agents. In Helder Coelho, Rudi Studer, and Michael Wooldridge, editors, *ECAI*, volume 215 of *Frontiers in Artificial Intelligence and Applications*, pages 567–572. IOS Press, 2010.
8. Louise Dennis, Berndt Farwer, Rafael Bordini, Michael Fisher, and Michael Wooldridge. A common semantic basis for BDI languages. In Mehdi Dastani, Amal El Fallah Seghrouchni, Alessandro Ricci, and Michael Winikoff, editors, *Proceedings of the International Workshop on Programming Multi-Agent Systems (ProMAS 2007)*, pages 88–103, May 2007.
9. R. S. Pressman. *Software Engineering: A practioner's approach.* McGraw-Hill, 7 edition, 2009.
10. M. Wooldridge. *An Introduction to MultiAgent Systems.* Wiley, 2 edition, 2009.