Title	Agent factory : a framework for prototyping logic-based AOP languages
Authors(s)	Russell, Sean E.; Jordan, Howell; O'Hare, G. M. P. (Greg M. P.); Collier, Rem
Publication date	2011-10-06
Publication information	Klügl, F. and Ossowski, S. (eds.). Multiagent System Technologies, 9th German Conference, MATES 2011, Berlin, Germany, October 6-7, 2011. Proceedings
Conference details	Presented at the 9th German Conference on Multi-Agent System Technologies, Berlin, Germany, 6-7 October 2011
Publisher	Springer
Link to online version	http://dx.doi.org/10.1007/978-3-642-24603-6_13
Item record/more information	http://hdl.handle.net/10197/3480
Publisher's statement	The final publication is available at springerlink.com
Publisher's version (DOI)	10.1007/978-3-642-24603-6_13

Agent Factory: A Framework for Prototyping Logic-Based AOP Languages

Sean Russell[1], Howell Jordan[2], Gregory M.P. O'Hare[1], and Rem W. Collier[1]

[1] CLARITY: Centre for Sensor Web Technologies, University College Dublin, Ireland
[2] Lero - the Irish Software Engineering Research Centre, University of Limerick, Ireland

Abstract. Recent years have seen the emergence of a number of AOP languages. While these can mostly be characterized as logic-oriented languages that map situations to courses of action, they are based on a variety of concepts, resulting in obvious differences in syntax and semantics. Less obviously, the development tools and infrastructure - such as environment integration, reuse mechanisms, debugging, and IDE integration - surrounding these languages also vary widely. Two drawbacks of this diversity are: a perceived lack of transferability of knowledge and expertise between languages; and a potential obscuring of the fundamental conceptual differences between languages. These drawbacks can impact on both the languages' uptake and comparability.

In this paper, we present a Common Language Framework that has emerged out of ongoing work on AOP languages that have been deployed through Agent Factory. This framework consists of a set of pre-written components for building agent interpreters, together with a set of tools that can be easily adapted to different AOP languages. Through this framework we have been able to rapidly prototype a range of different AOP languages, one of which is presented as a case study in this paper.

1 Introduction

The last 10 years has seen the emergence of a number of logic-based Agent-Oriented Programming (AOP) languages, such as 3APL [5], Jason/AgentSpeak [2, 10], GOAL [8], and AFAPL2 [3]. A common criticism of these languages is the associated learning curve, which is often compounded by the lack of supporting tools that facilitate development, deployment and debugging. While some languages do offer reasonable levels of tool support, a secondary criticism is often that there is such cross-language diversity in this tool support that it can be difficult to transfer experience between languages. For example, a developer who learns to program Jason agents may not be able to easily apply their experience to learn how to program in AFAPL2. This issue was demonstrated at a recent Agent-Oriented Software Engineering course held in University College Dublin, in which around 40 students (all professional software engineers with 5+ years industry experience) enrolled in the Advanced Software Engineering Masters programme were asked to develop agent systems using both Jason and AFAPL2. The main criticism raised by the students arose not in understanding the different language concepts, but in the diversity of the supporting machinery. For example, in AFAPL2, students were required to develop perceptors, actuators, modules, and platform services to link the language to their environment, whilst in Jason, the students were required to develop an Environment class that played a similar role. While such diversity reflects differing approaches to building multi-agent systems, it also acts as a barrier to entry for the wider software engineering community as it makes direct comparison and evaluation of AOP languages more difficult. As an aside, informal feedback from the students indicated no clear consensus as to which language was preferred, as some students preferred AFAPL2 whilst others preferred Jason.

With these criticisms in mind, recent work on the Agent Factory framework [4] has focused on supporting heterogeneous logic-based agent architectures with the goal of providing a common toolset that can be adapted to different agent models, ranging from custom Java agents, through to reactive agent architectures and finally to high-level agent programming languages. Whilst primarily these components have been designed to support languages based on first-order logic. Non logic based languages can also be developed using the framework, as long as the language is compatible with the FIPA based Agent Factory Runtime Environment.

Specifically we have redeveloped the AFAPL2 logic framework in order to make it modular and extensible, this was required as it was previously limited to only first order structures. Additionally we have decoupled language syntax and underlying logic structure to allow the component be used in different languages easily. A new planing mechanism has been developed based on the intention stack concept within AgentSpeak(L). Furthermore we have reimagined the environment interface to allow aggregation of relates sensors and actions in the form of modules.

Agent Factory is by no means the only framework or platform that supports heterogeneous multi-agent architectures. Platforms such as JADE [1] provide essential runtime infrastructure such as agent discovery, inter-agent communication, and fault tolerance; through Java APIs, these services are made available to both native Java agents and high-level AOP language interpreters. Though such platforms often provide low-level development tools for deployment and debugging, they typically offer no direct support for high-level AOP languages.

Language-independent frameworks for high-level AOP languages have been the subject of relatively little research. Dennis et al developed an extensible model checker for Beliefs-Desires-Intentions (BDI) agent programs, by distilling the common features of several AOP languages to create an intermediate language called Agent Infrastructure Layer (AIL) [7], and an AIL verifier called Agent Java PathFinder [6]. In this approach, the developer of a new AOP language *X* can obtain a model-checking tool for *X* by simply implementing an *X*-to-AIL compiler; AIL makes no assumptions regarding the source language's interpreter cycle, and has clear semantics, making this task relatively easy. We regard this work as complimentary to our own. Rather than modelling the commonalities of existing languages, Agent Factory aims to provide greater flexibility, in the form of tool and platform components that can be reused to quickly implement and explore novel agent programming language features.

Section 2 presents the various sub-frameworks that make up the CLF. Following this, section 3 describes the process of creating a language using the CLF, and section 4 describes the evaluation of the framework. Finally, section 5 presents some concluding remarks.

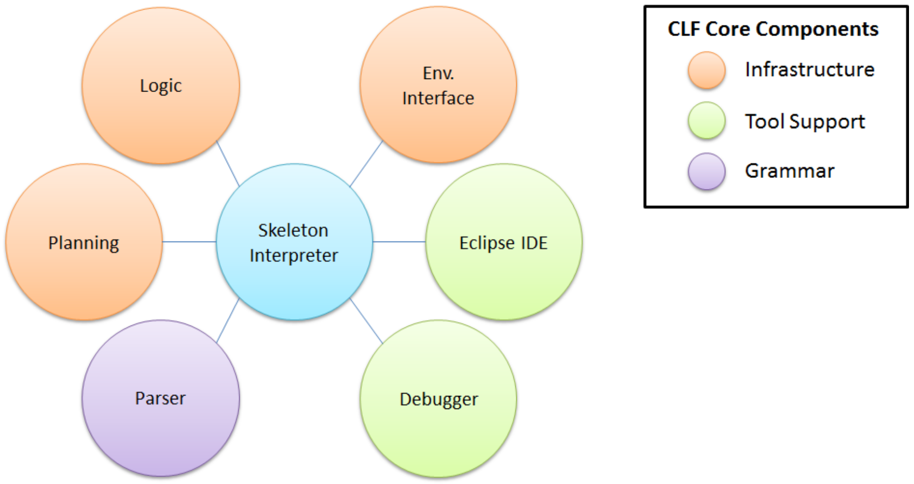


Fig. 1. Schematic of the Common Language Framework.

2 Multi-Language Support in Agent Factory

Agent Factory (AF) is an open-source framework for the development of agent-based systems [4]. Since 2001, AF has been structured over a number of layers, with the lower run-time environment layer providing FIPA-standards based support for agent interoperability much in the same way as is done in other agent platforms, such as JADE [1]. The upper levels of the AF framework then deliver support for the fabrication of agents using the AFAPL2 agent programming language [3].

In order to facilitate ongoing work on Agent Factory and AFAPL2, we have attempted to develop a range of sub-components that can be easily adapted and reused as necessary. We believe that these components have reached a level of maturity that will allow AOP language developers to utilise them to rapidly prototype new agent programming languages. As is depicted in figure 1, these components are known collectively as the Common Language Framework (CLF), and they are outlined below.

2.1 Infrastructure Support

The infrastructure components provided by the CLF are concerned with the internals of the agent. They provide support for: representing and manipulating first-order logic; plan representation and execution; and a standardised interface between the agent and its environment.

Logic Framework: Agent Factory utilises a standard predicate logic language, similar to those used in other systems such as GOAL[8] and Jason[2], which is based on the one originally developed for AFAPL2 [3]. Two key interfaces are used to represent well formed formulae within the language; IFormula and ITerm. Figure 2 shows the constructs supported within the language in which operators implementing ITerm are used to represent the arguments of predicate formulae and comparisons.

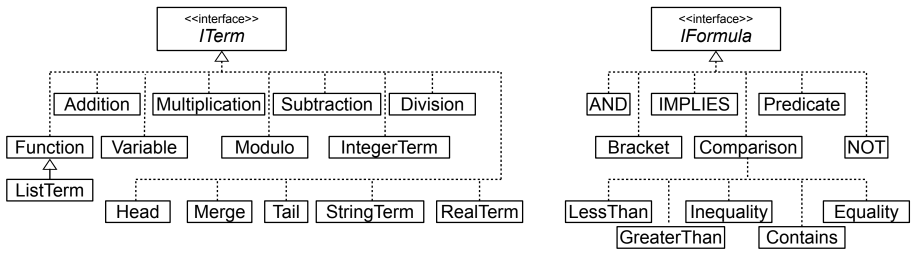


Fig. 2. UML diagrams of IFormula and ITerm.

Agent Factory provides support mechanisms designed to enable the easy creation and conversion of these constructs through two interfaces: *ILogicFactory* and *ILogicPresenter* respectively. Default implementations are provided for both interfaces which can process any of the logical operators shown in Fig. 2, assuming the input consists of well formed formulae only.

Finally, support for reasoning with logic is provided via multiple logic reasoning engines allowing queries over sets of well formed formulae or the generation of beliefs based on these formulae and a set of inference rules. Both of these systems implement the the *IQueryEngine* interface and where neither is appropriate, additional reasoning engines can be built. The IQueryEngine interface is designed to support multiple sources, such as the belief and goal bases of an agent; when a query is run the IQueryEngine determines the applicable sources to check based on the underlying type of logic object. Sources are specified by implementing the *IQueryable* interface.

In summary, the logic framework of Agent Factory implements the standard functionality one would expect to see in a basic logic system and provides clearly defined extension points that allow the logic to be modified for a specific language. However, in many cases, we expect that such modifications will focus on specific formulae that are based on the standard AFAPL2 logic syntax and will not require modification of the reasoning engines or the re-implementation of the default logic engine and logic presenter. In cases where languages do not employ our syntax, we have preferred to adapt the language syntax rather that undertake more time-consuming modifications to the framework.

Environment Interface: The interface between an agent and its environment is based around two core components: *sensors* and *actions*. Generally speaking, sensors are the components that are responsible for generating the agents model of its environment, while actions are the components that cause some change to occur in the environment. As such, the core focus of a sensor is belief generation and the core focus of an action is to facilitate manipulation of the environment. The term action differs from other systems where it is usually referred as actuator, this difference is due to the desire to differentiate actions from perceptors, which are the equivalent concepts in the AFAPL2 logic system.

Experience in developing agent-based systems has shown that actions and sensors alone are insufficient - there are many cases where actions and sensors must interact through some shared data structure or API, for example: a graphical interface, a connection to a remote system (e.g. controlling a robot via bluetooth), or something simpler, such as a queue. While it is possible to model all of this through the FIPA notion of a platform service, the approach is not appealing because platform services are typically shared resources, whereas these resources are more often private (to the agent). As a result, we introduce a third component, known as a *module*. This component is an aggregation of related sensors and actions that share a common data structure (the module). Typically, the sensors and actions associated with the module are implemented as inline classes, making the component self-contained, but this is not required, and the implementation can just as easily be spread over multiple classes. Modules are named so as to maintain a clear distinction between them and services, which are a runtime environment concept.

By introducing the notions of actions, sensors and modules, we offer a simple model for integrating with the environment that engenders reuse of components both across applications and across languages. In particular, modules can be viewed as a form of API that can be used by any CLF-based agent. Currently, module's exist for: creating and manipulating Stack and Queue data structures; and interacting with various pre-existing platform services, including the FIPA Agent Management Service, and our EIS and CARTAGO services. As hinted above, they can also be used to implement clients to connect to remote systems or graphical interfaces for agent-user interaction.

Planning Support for planning comes in the form of an extensible set of plan operators and a plan executor that is based on the intention stack approach adopted in Jason [2]. The default implementation includes: brackets, if-else statements, while loops, state querying, plan expansion (foreach), assignment (assigning a value to a variable), failure handling (try-recover), and durative actions. Each of these operators has an associated class that implements the *IPlanStep* interface. The key method of this class is the *handle(...)* method, which implements the operational semantics of that plan operator. This method is called by the plan executor. Additional operators can be added, for example, in Af-AgentSpeak, goal invocation is implemented as a custom plan step.

2.2 Skeleton Interpreter

The recommended hierarchy of the agent classes is shown in Fig. 3. Whilst it is not required that agents extend the AbstractLanguageAgent class, it is the easiest and quickest way to incorporate Agent Factory's Environment Interface, FIPA standard platform services, and runtime environment features such as scheduling algorithms. The AbstractLanguageAgent class provides basic agent action and sensor functionality, such as printing, ACL communication, migration, state monitoring, and platform service availability.

When prototyping languages we recommend the following naming scheme. The primary functionality of the agent is encapsulated within the *AbstractXXXXAgent* class,

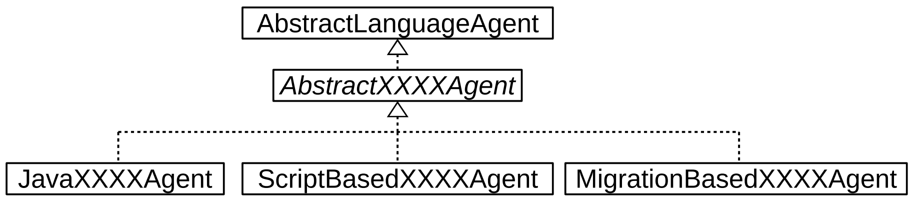


Fig. 3. Recommended agent class hierarchy.

where XXXX is replaced with the name of the language. This scheme is also applied to the three recommended subclasses, *JavaXXXXAgent*, *ScriptBasedXXXXAgent* and *MigrationBasedXXXXAgent*, which will be further explained in section 3.

As is shown in figure 4, the interpreter of the language is represented by the method *execute()* in the AbstractXXXXAgent class. This method performs a single step of the interpreter, during which the agent perceives its environment, deliberates, and performs an action. The 3-step process shown below is not enforced, but is indicative of a typical execution step.

```
public class AbstractXXXXAgent extends AbstractLanguageAgent {
    protected BeliefSet beliefs;
    protected IQueryEngine queryEngine;

    <constructor>(String name) {
        super(name);
        queryEngine = new ResolutionBasedQueryEngine()
        beliefs = new BeliefSet();
        queryEngine.addSource(beliefs);
    }

    public void execute() {
        // 1. Sense environment
        senseEnvironment();

        // 2. Deliberate...

        // 3. Act: Select one or more activities:
        for (Predicate activity : activities)
            performActivity(activity);

        endOfIteration();
    }

    protected void noSuchAction(Predicate activity) {
        ...
    }
}
```

Fig. 4. Recommended structure of an AbstractXXXXAgent.

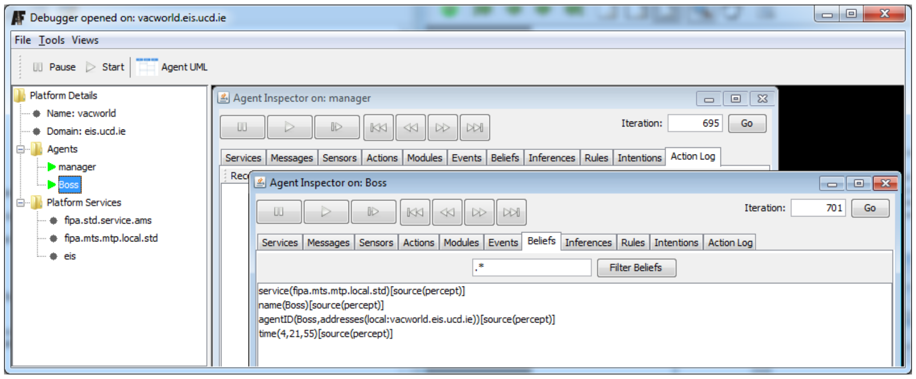
2.3 Parser Support

Parser support is provided using JavaCC and JJTree. While a separate parser must be developed for each language, CLF includes sample JavaCC production rules demonstrating the use of each infrastructure component, which can easily be customised. The CLF provides standard visitor implementations to generate compiler code from these production rules; only the CodeGeneratorVisitor must be modified to handle the new language constructs. At runtime, the ScriptBasedXXXXAgent outlined in section 2.2 invokes this compiler, and uses the resulting object files to initialise the interpreter.

2.4 Tool Support

Debugger Framework The Agent Factory Debugger is a highly extensible tool designed to be easily customisable for different agent architectures. The debugger allows agents to be collectively suspended or resumed as well as individually resumed, suspended and stepped.

The debugger provides a default inspector for all agent factory agents which details the platform services subscribed to and a log of incoming and outgoing messages. This default inspector also manages the state history of the agent and can be easily extended to include much more information, requiring only the extension of two components: the state manager and the inspector.


Debugger opened on: vacworld.eis.ucd.ie
File Tools Views
Pause Start Agent UML
Platform Details
Name: vacworld
Domain: eis.ucd.ie
Agents
manager
Boss
Platform Services
fipa.std.service.ams
fipa.mts.mtp.local.std
eis
Agent Inspector on: manager
Iteration: 695 Go
Services Messages Sensors Actions Modules Events Beliefs Inferences Rules Intentions Action Log
Agent Inspector on: Boss
Iteration: 701 Go
Services Messages Sensors Actions Modules Events Beliefs Inferences Rules Intentions Action Log
.*
Filter Beliefs

```
service(fipa.mts.mtp.local.std)[source(percept)]
name(Boss)[source(percept)]
agentID(Boss,addresses(local:vacworld.eis.ucd.ie))[source(percept)]
time(4,21,55)[source(percept)]
```

Fig. 5. Screeenshot of the debugger.

A screen shot of the Agent Factory debugger is shown in Figure 5, this shows the the Agent Inspectors and controls for two agents of the same type. In this situation there is only one instance of the *StateManagerFactory* and *InspectorFactory*, these factories create a seperate *StateManager* and *Inspector* (Visible in the right pane of the figure) for each agent .

Each agent's StateManager is responsible for updating its Inspector at the end of every execution cycle. To this end the StateManager creates a *Snapshot* of the current internal state of the agent and passes it to the Inspector. Extending the functionality of the inspector requires the addition of new InspectorPanels formatted to display the required information and the extension of the Snapshot class to include the data required to update the Inspector.

Eclipse Integration Eclipse integration with Agent Factory is provided through the use of a number of plugins, one which provides the Agent Factory libraries as well as a number of support classes and an individual plugin for each of the languages developed using the common language framework.

The core functionality provided by each of the individual plugins consists of;

1. An *Editor* to provide syntax highlighting on the various elements of the language.
2. A *Builder* to provide a mechanism for the automatic compilation of the agent code and reporting of errors through eclipse.
3. A new file Wizard to automatically create an agent file from a template.

3 Prototyping an AOP Language

AOP language developers typically employ a wide range of techniques in the design and implementation of an AOP language that includes the design of the underlying reasoning mechanisms, the definition of an interface to the environment, and the integration of the interpreter with an associated run-time framework, and potentially the creation of an associated development toolkit. In this section, we outline an approach to prototyping AOP languages that attempts to remove many of these barriers to allow the developer to focus on the core deliberation algorithm. We do this by advocating a simple structure for the design of AOP languages that is based on the approach adopted in the design of AFAPL2. We do not argue the this approach is better or more suitable, but advocate its use for the purpose of ensuring greater consistency between languages.

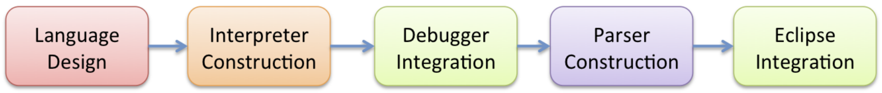


Fig. 6. CLF Language Development Process.

The specific approach advocated combines the default logic implementation outlined in section 2.1 together with the environmental interface described in Section 2.1.

In order to develop a new AOP language using AF, there are a number of key steps to be followed. In this section, we outline these steps through a set of high level instructions, should the reader require more detailed information this is available through the Agent Factory website[3].

[3] `http://www.agentfactory.com`

3.1 Language Design

First the specifications of the language must be defined, this includes both the syntax of the language as well as the operation semantics of the interpreter. The former can be achieved through the construction of a Backus-Naur Form (BNF) grammar describing the unique constructs of the language, this can be done by creation of a grammar in the case of a new language or the modification of an existing grammar when the language already exists. The latter requires the definition of the operation semantics of the interpreter which will be used to structure the actions of the interpreter in the next section. Finally it is recommended that a suitable example agent program is written, as this will make later testing much easier.

3.2 Skeleton Interpreter

Based on the recommended structure provided in Section 2.2, implementation of the skeleton interpreter requires the creation of an abstract class *AbstractXXXXAgent*. This class requires the creation of custom data structure to represent the constructs of the language and the agents belief, goals, etc.. The operational semantics defined in Section 3.1 are then used to structure the implementation of the *execute()* method within the class thus providing the functionality of a single step of the interpreter.

It is also recommended that the class *JavaXXXXAgent* should be created which extends AbstractXXXXAgent, this allows the direct injection of the example agent program into the data structures of the agent without the need for a parser. Through this class the operation of the interpreter and the functionality of the data structures can be tested.

3.3 Debugger Implementation

The next logical step in the development process is the implementation of the debugger as it allows the mental state of the agent be viewed during the operation of the agent. This is instrumental in ensuring the correct operation of the developed interpreter. In short this process requires the creation of a number of classes used for the representation of the agents state and it display within the debugger, namely;

- *XXXXSnapShot*, which holds all the required state information.
- *XXXXStateManager*, which creates the snapshots at the end of an interpreter cycle.
- *XXXXInspector*, which displays the recorded information
- *XXXXStateManagerFactory* and *XXXXInspectorFactory* which manage the automatic creation of the StateManagers and Inspectors respectively.

3.4 Parser Implementation

Having successfully tested the interpreter using the debugger and the JavaXXXXAgent, the next step is the implementation of the parser for the language and its integration with the system. This process involves a number of complex steps;

– Based on the grammar developed in Section 3.1 and the template provided within Agent Factory, complete the JavaCC grammar defining the language. As production rules are supplied for parsing the logic and environmental interface components this is not an overly difficult task.
– Using JJTree, which constructs a parse tree during processing, modify the *Code-GeneratorVisitor* template provided to harvest and store the required data structures from the parse tree.
– Finally the creation of the *ScriptBasedXXXXAgent*, which utilises the parser and CodeGeneratorVisitor to populate all the agent code into the class, thus allowing the creation of agents within the language.
– To fully connect the agent type with Agent Factory we must create the *XXXXArchitectureFactory* which is responsible for automatic creation of agents using the ScriptBasedXXXXAgent class, automatic creation is achieved through the association of the file extension to the agent type.
– A supplementary step in the process is the creation of the *MigrationBasedXXXXAgent* which is similar to the ScriptBasedXXXXAgent in that it allows the creation of an agent of the language on the system. Rather that through the use of the parser it uses the state information from another platform to instantiate the agent.

3.5 Eclipse Integration

At this point the language is fully developed and functional, however the final step of Eclipse integration is recommended as it makes designing agents within the language easier. As discussed in Section 2.4 there are three components to the eclipse integration, the builder, editor and new file wizard. An Eclipse plugin is provided which provides the structure and an example implementation of all the features described.

– The builder is implemented by integrating the parser developed in Section 3.4 and display the errors through eclipse's built in marker system.
– The editor requires only the modification of the example implementation such that the correct keywords, labels and operators are highlighted.
– Finally the new file wizard is created through the modification of the example and the provision of a sample agent file.

4 Evaluation

Scientific evaluation of this system is not an easy task, initial evaluation has been completed and further evaluation is planned.Informally it is the opinion of the authors that the provided components simplify the process of developing AOP languages. The initial evaluation comprised of the use and comparison of two languages created using the framework during participation in the 2010 Multi Agent Contest[4][11].

The contest scenario consisted of developing a multi-agent system to solve a cooperative task in a dynamic environment. The environment was a grid-like world in which

[4] `http://www.multiagentcontest.org/2010`

virtual cows move around collectively in one or more herds exhibiting a swarm-like behavior. There were two corrals, each belongs to one of the two agent teams. The teams of agents competed to control the behavior of animals and lead them to their own corral. The winning agent team being the one which scored highest.

The two languages used were AF-TeleoReactive (AF-TR), which is based on Nils Nilsson's Teleo-Reactive formalism [9], and AF-AgentSpeak (AF-AS), an implementation of AgentSpeak(L). The scenario is very much suited to having two types of agents, a leader agent and a herder agent. The CLF allowed us do develop modules which could be used with both languages and made it possible to rapidly develop a number of prototype agents with those languages. Through these prototypes, we were able to identify that AF-AgentSpeak was suited to the Leader role, this is due to the comprehensive planning support available, and AF-TeleoReactive was suited to the Herder role, as it is designed to react quickly to a changing environment.

This pilot evaluation serves to justify the comparability of the created languages. As this project was completed by members of the team responsible for the creation of the framework, evaluation of the CLFs ease of use is currently future work. This thorough evaluation will be performed when the Agent-Oriented Software Engineering course runs again (Jan 2012). The planned evaluation similarly allows the assessment of the CLF in terms only of the usability and comparability of the produced languages, whilst this evaluation is useful it is not sufficient to validate the goal of providing a common toolset that can be adapted to different agent models.

To properly validate the usefulness of the components for the development of AOP languages would require a large number of people creating languages both with and without the support of the framework. Whilst ideally we would like the feedback this would bring, for logistical reasons alone we will not be attempting it.

5 Conclusions

Our main objective making these changes to Agent Factory is to develop versions of existing AOP languages for Agent Factory, that are adapted to employ a consistent underlying infrastructure which we hope will allow developers to focus on understanding the strengths and weaknesses of the languages rather than the supporting machinery. By providing this common infrastructure and implementing various AOP languages, we also hope to gain additional insight into the weaknesses of the current state-of-the-art in AOP with the goal of identifying and exploring potential features that will underpin the next generation agent programming languages. Finally, we hope that Agent Factory will help to foster the development of new AOP languages by reducing the complexity of AOP language development.

6 Acknowledgements

This work was supported, in part, by Science Foundation Ireland grant 07/CE/I1147 to CLARITY: Centre for Sensor Web Technologies and by Environmental Protection Agency Ireland project number 2008-WRM-MS-1-S1, WAIST - Waste Augmentation and Integrated Shipment Tracking, funded under STRIVE- Waste, Resource Management and Chemicals Call 2007.

References

1. F.L. Bellifemine, G. Caire, and D. Greenwood. *Developing multi-agent systems with JADE*. Wiley, 2007.
2. R.H. Bordini, J.F. Hübner, and M.J. Wooldridge. *Programming multi-agent systems in AgentSpeak using Jason*. Wiley-Interscience, 2007.
3. Rem W. Collier and Gregory M. P. O'Hare. Modeling and programming with commitment rules in agent factory. In Giurca, Gasevic, and Tavater, editors, *Handbook of Research on Emerging Rule-Based Languages and Technologies: Open Solutions and Approaches*. IGI Publishing, 2009.
4. R.W. Collier. The realisation of agent factory: An environment for the rapid prototyping of intelligent agents. Master's thesis, University of Manchester Institute of Science and Technology (UMIST), England, 1996.
5. M. Dastani, M.B. van Riemsdijk, F. Dignum, and J.J.C. Meyer. A programming language for cognitive agents goal directed 3apl. *Programming Multi-Agent Systems*, pages 111–130, 2004.
6. L.A. Dennis, B. Farwer, R.H. Bordini, and M. Fisher. A flexible framework for verifying agent programs. In *Proceedings of the 7th international joint conference on Autonomous agents and multiagent systems*, pages 1303–1306. International Foundation for Autonomous Agents and Multiagent Systems, 2008.
7. L.A. Dennis, B. Farwer, R.H. Bordini, M. Fisher, and M. Wooldridge. A common semantic basis for BDI languages. In *Proceedings of the 5th international conference on Programming multi-agent systems*, pages 124–139. Springer-Verlag, 2007.
8. K.V. Hindriks. Programmingrationalagents in goal. *Multi-Agent Programming:*, pages 119–157, 2009.
9. N.J. Nilsson and Stanford University. Computer Science Dept. *Toward agent programs with circuit semantics*. Citeseer, 1992.
10. A. Rao. Agentspeak (l): Bdi agents speak out in a logical computable language. *Agents Breaking Away*, pages 42–55, 1996.
11. Sean Russell, Dominic Carr, Mauro Dragone, Gregory OHare, and Rem Collier. From bogtrotting to herding: a ucd perspective. *Annals of Mathematics and Artificial Intelligence*, pages 1–20, 2011. 10.1007/s10472-011-9236-z.