Agent Programming Extensions relying on a component-based platform

Christian Vecchiola  
Department of Communication, Computer and System Sciences  
University of Genova  
Genova, Italy  
cvecchiola@canada.com

Mauro Coccoli  
Department of Communication, Computer and System Sciences  
University of Genova  
Genova, Italy  
coccoli@dist.unige.it

Antonio Boccalatte  
Department of Communication, Computer and System Sciences  
University of Genova  
Genova, Italy  
nino@dist.unige.it

Abstract - This paper describes a suite of extensions to the object oriented model, steering OO programming towards the Agent Programming paradigm; such set of extensions is called Agent Programming eXtensions (APX). APX has been specifically designed to leverage the development of agent-based systems, a widely adopted paradigm to model complex and dynamic software systems. A brief survey of the most commonly used approaches to agent programming is presented, against which APX is evaluated. Then the structure of APX is described and the programming interface offered to the developers illustrated. The developers' point of view, who can greatly benefit from the component-based platform which APX relies upon is also taken into account.

Keywords: Agent programming, program compilers, computer languages, software packages.

1 Introduction

This paper deals with the design and the implementation of some extensions to the well known object oriented programming model, with the aim of promoting the agent oriented software development. The outcome of this work is a suite of programming tools, arranged in a set of extensions called APX: Agent Programming eXtensions. APX encompasses a set of syntactical and semantic elements, added to the object oriented model, that naturally mirror the key-abstractions of the agent programming paradigm.

Agents and multi-agent systems [1] are a widespread research topic. Agent-based technology has been proposed as an effective solution to common requirement of information systems applied to manufacturing, communication, workflow management, and many other subjects [2]. Nowadays many aspects of this subject have been profitably investigated, thus different formal specifications of agents’ semantics and behavior have been proposed. Besides, a wide a range of software libraries and frameworks to develop agent based applications have been delivered [3]; the solution of agent-oriented programming languages has also been evaluated [4].

Applied agent programming is often based on the existence of an underlying agent programming framework, designed as a hierarchy of classes (usually implemented in JAVA, or C++). The object oriented model [5] has proven to be a sound approach, able to cope with complex, dynamic, and large systems. Many programming patterns [6] have been developed to manage, order, and modularize large software projects. By following some agent programming languages [7], [8], the agent oriented extensions have been modeled, by using an object oriented basis. APX fully exploits the expressiveness of the object oriented programming and it offers the programmer a new agent perspective which can significantly improve the code development phase. APX relies on a specific agent programming framework carried on by a local research group, that is AgentService [9]. AgentService is built on the top of the .NET componentware platform [10] and it takes full advantage of the language interoperability features exposed by the Common Language Infrastructure (CLI) [11]; thanks to this, high-level components can be easily integrated inside agent-oriented applications. APX has a twofold design objective: firstly a more agent-friendly interface is offered to the programmer; secondly the programming patterns required by the AgentService framework are automated by the APX compiler. As a result, by using APX, the programmers’ efforts can be focused on defining the peculiar agent characteristics and behavior rather than on solving problems related to technical implementation details.

Sections 2 and 3 will give a brief survey about agent theories and multi-agent systems. The AgentService programming framework will be briefly discussed in Section 4, by pointing out the features used by APX. The same section will include a summary survey of the Common Language Runtime that is the common basis for both APX and AgentService. The main characteristics of APX will be detailed in Section 5 and followed by a comparison with the approaches proposed with previous
agent programming languages in section 6. Final considerations follow in Section 7.

2 Agent and Multi-Agent Systems

2.1 Software Agents

A software agent is a piece of autonomous software provided with some levels of “intelligence”. An agent is a high-level system component with its own goals that may change during its life cycle. An agent is situated in an environment; it senses and acts on it in order to meet its design objectives [12]. Agents should be adaptive and learn from the mutating conditions of the environment: ability of learning and adaptation can make the agent achieve its goals in a better way.

From the perspective of mere programming, agents are commonly implemented as concurrent, autonomous, self-contained objects, yet, this classification can sometimes be reductive. Objects are defined as computational entities, encapsulating a state, which they are able to perform actions on; they also have communication capabilities, through message passing. Objects have full control over their state, but not over their behavior. Agents need to control their behavior since they normally interoperate with entities that can be trusted and un-trusted, and they are plugged into an environment over which they have only partial control. For these reason the object model needs something more to implement the agent model.

Autonomy is an important, distinctive feature of agents: objects are not requested to be autonomous, while agents are autonomous by definition. Even if active objects [13] are generally autonomous – that is they can exhibit some behavior without being operated upon another object – they do not necessarily have the ability to exhibit a flexible autonomous behavior. Some levels of intelligence are usually ascribed to agents, but this does not work for objects. In this case intelligence stands for the exhibition of a flexible behavior exposed by means of reactivity, proactivity and social ability [12]. Agents and objects are clearly different entities, nevertheless, the object oriented model can still be used to implement agent oriented applications: a kind of infrastructure – providing the previously discussed features – has to be shaped on the top of the object oriented layer. This is what some of the agent-oriented programming languages and almost all the agent programming frameworks do.

Different architectures have been proposed to model, to describe, and to define agents. The Believe, Desire Intention (BDI) architecture has been proven useful to implement software agents and good references for this subject can be found by interested readers in [14].

2.2 Multi-agent Systems (MAS)

Social ability is one of the most important features of agent oriented programming: agents do their best when they interoperate. Interaction is obtained by arranging agents in community normally called multi-agent system (MAS). Multi-agent systems are a very important component of the environment in which the agents are situated; the remaining part of such environment is characterized by the entire legacy (non agent oriented) software bounded to the agents’ activity. A more formal definition of a multi-agent system can be found in [15]: “a system where intelligent agents interact in order to satisfy a set of goals and perform some tasks”. Multi-agent systems are normally decentralized and open systems with distributed controls and asynchronous computation: these systems provide an infrastructure that specifies both the interaction and the communication protocols for agents. Multi-agent systems do not provide just a context for agents’ activity, but they are also scalable, fault-tolerant, reliable, and designed for reuse.

An abstract architecture specification of a generic multi-agent system has been proposed by the Foundation of Intelligent Physical Agents (FIPA); this is an international organization that promotes standards for agent technologies. The proposed architecture [16] is implemented by different multi-agent systems and has been taken as a reference model in the comparison of different implementations of MAS.

3 Agent Programming Languages

The idea of providing a support to agent programming by means of a programming language, dates back to 1991 with the Agent-0 programming language [17]. Agent programming languages let the developers define and program agents that fulfill the requirements of a specified agent model. The BDI architecture is the most commonly implemented architecture for agents programmed with such languages. These programming languages normally model agents as components that can be plugged into systems like MAS, or as a set of assertions and clauses processed by some ad-hoc interpreters. Accordingly two programming paradigms have been taken as a basis to develop fresh agent programming languages: the object oriented model and the logic programming paradigm. In particular the Java technology has been adopted for all those languages that adopt the object oriented as reference model. In the following, a selection of the most known agent programming languages will be presented. Such languages will be taken as referencing examples in comparing the results of the present work.
3.1 Agent-0, Placa Extensions, Agent-K

The first step in the field of agent programming languages has been characterized by formal languages and by their translation into programming languages. Most of the work of the AI theorists has been focused on the definition of the semantic of such formal languages; these languages lead to the development of Agent-0 and its further extensions: PLACA [18] and Agent-K [19]. All these languages have similar underlying semantics.

Agent-0 is the first agent oriented programming language and it is built upon three key elements: beliefs, obligation, and capabilities. Beliefs are point based time statements and describe the state of the world at a fixed instant of time. Obligations are commitments to make true a specified statement at a point in time. Capabilities are assertions about what the agent can perform. The execution of an agent is driven by a loop controlled by the Agent-0 language interpreter, who gathers incoming messages, updates the agent’s mental state, and executes the commitments using capabilities.

One of the limitations of Agent-0 is the fact that agents have no mechanism to develop new goals, the PLACA extensions attempt to resolve this lack by introducing new syntactical elements designed to manipulate intentions.

Agent-K is an attempt to standardize the message passing functionality in Agent-0. It combines the syntax of Agent-0 (it does not currently support the planning mechanisms of PLACA) with the format of KQML (Knowledge Query Manipulation Language) [20], to ensure that messages written in other languages can be handled.

3.2 APRIL

APRIL (Agent Process Interaction Language) [21] is a programming language oriented to the implementation of multi-agent systems, built on the top of the C language. APRIL is an object oriented language with objects as processes: it contains libraries for the definition of processes as well as tools that allow processes to inter-communicate in a distributed environment. It also offers a powerful data structuring and expression handling features as it might be found in any high-level symbolic programming language.

3.3 AgentSpeak(L)

AgentSpeak(L) [22] is an agent oriented programming language built on the top of the BDI systems PRS [23] and dMARS [24]. It allows agents programs to be written and interpreted in a similar way to that of horn-clause logic programs. AgentSpeak(L) is a programming language based on a restricted first-order language with events and actions. Beliefs, desires, and intentions of an agent can be defined by means of the AgentSpeak(L) that provides an abstraction on the underlying systems.

3.4 JACK

JACK Intelligent Agents [8] framework is a multi-agent framework that extends the Java language. JACK is composed of a set of class libraries constituting the core of the multi-agent framework, a set of extensions to the Java programming language, and a tailored compiler. JACK is designed to support different agent architectures such as the BDI one and others. It also takes all the benefits of the Java language to program the agent activities.

3.5 APL

APL (Agent Programming Language) [7] is an agent programming language based on the BDI model. It provides templates for each component of the BDI model (Belief, Desire, Intention and Agent) by means of a Java-like syntax. APL is built on the top of the Java type system and every template is translated into a java class by the APL compiler. Each variable in APL has the template type of Belief and each method is considered to be a plan. The language also offers automatic persistence of beliefs and goal-replanning features for agents.

4 AgentService Platform and the CLI

Agent Service is a novel programming framework designed to develop agent applications, based on the CLI and modeled on the architectural specification promoted by FIPA. The platform takes care of the agent deployment and agent activity scheduling system, which takes advantage of both the reflection and the “Application Domain” within the CLI.

The core of the AgentService framework is its active library through which the developer can build a reliable and efficient multi-agent system. The AgentService library provides several features for agent programming: the definition of a real autonomous, independent, and persistent agent; concurrent execution of agents and their multi-behavior activity; persistent shared data structures within a single agent; transactional agent communication based on message exchange; access to the FIPA service components.

The structural agent model proposed by AgentService designs an agent as a software entity whose state is defined by a particular managed set of data (Knowledge) and whose activity is performed by behaviors. Other data exist in the agent but they do not belong to the agent state and they can only be used by a particular behavior in order to perform some local activity. The data composing the agent state must be accessible from every behavior but they are collected into a specific software component, the Knowledge, which
AgentService provides communication services, white and yellow pages services, and some administrative tools. Furthermore, AgentService allows the platform to perform any check on the agent’s state, and it allows the development of advanced features.

5 Agent Programming Extension

5.1 The APX package

The Agent Programming eXtensions package provides the developer with a set of templates for the design and the implementation of software agents, and with a tailored compiler that targets agents to the AgentService platform. APX is built on the top of the Common Language Infrastructure and it strongly relies on one of its most innovative features: language interoperability. APX has been designed to fully exploit the underlying component-aware platform. APX is not a general purpose language, it has no first class types but it offers run-time managed templates that can be programmed by the agent developer using a C#-like syntax.

APX has been specifically designed to take care only of those aspects strictly concerning software agents: by using APX, the programmer can design agents by composing their behaviors and their knowledges, then behaviors can be programmed by defining the computation needed to accomplish their activity. Other elements – or particular components – can be designed by using any other language supported by the CLI, and thus can be used with APX. APX itself comes with a complete access to the object oriented model exposed by the CLI, hence every component designed for such platform can be instantiated and manipulated using APX. APX does not permit the design of object oriented code: neither new classes nor interfaces can be defined, even if objects can be used. This is an important design decision, since the aim of this work is to focus on agent design and implementation and to rely on interoperability to provide other features. Interoperability among languages is the most interesting design decision here. APX offers a set of templates exposed as types in a language: they are not first-class types since they can neither be assigned nor freely manipulated as built-in types. These prototypes can be specialized and programmed by adding methods and fields or by composing them. This approach is not restrictive since such templates are run-time managed by the AgentService platform. Developers’ efforts can be entirely directed to design and to program such agents. Three different kinds of templates are offered to the developer and there is a straightforward mapping between such templates and the classes that constitute the agent model supported by AgentService. Each template has a corresponding class.

Thanks to these programming constructs developers do not need to program agents by using the classes defined within the AgentService library. While code is being generated, the APX compiler can automate many programming tasks required by the agent platform. As a result, agents’ developers are not bothered with technical issues relevant to the object oriented model of the platform. Another programming construct introduced with APX is a sort of lock statement, specifically designed to control the access to knowledge objects. As in the case of the templates, the AgentService platform provides functionalities to access such elements within an agent. Knowledge objects are shared among behaviors and a simple lock statement sometimes could not perform all the necessary operation to guarantee consistency on such data. Without the use of APX, it is up to the programmer managing these tasks. The APX compiler ensures that such data are accessed by applying the right programming pattern. Knowledge objects are visible only within these “lock” statements: such technique does not permit the developers to give raise to race conditions and deadlocks.

All the previously presented features have been designed to leverage the work of the developer and to provide an easier interface for agent programming with AgentService. Many of the difficulties experienced using directly the AgentService class library are not ascribed to architectural design pitfalls, but they are related to the fact that, by using the object oriented language, it is necessary to set up a kind of infrastructure by using explicitly programming patterns. APX automates these programming patterns whenever it is possible. Another important design feature is modularity: developers can program behaviors and knowledges like components and reuse them to define different agents. This is important issues when developing large and complex software systems.

5.2 The Agent Template

APX comes with an extremely modular structure and the defined agents follow this guideline. Agents can be designed by composing the different activities they perform and the data structures they need. These elements can be configured and scheduled inside the agent type, by coding the initialization phase of the agent template. This particular approach adopts the idea of an agent as a container of different components, that can be customized in order to cooperate and to characterize the particular behavior of all the agents instantiated using this template. The agent model exposed with APX is the same as the one of the underlying platform: an agent is described by that set of activities it performs, and the code is not localized in the agent definition but it lies inside classes that model such activities indirectly. This approach encourages code reuse, partitioning, and allows a simple agent definition.
Agents are types managed at runtime by the AgentService platform, because of these reason APX does not provide any means to manipulate agents but only to define them. Agent templates are not assignable within APX, neither references to agent objects can be obtained. APX is intended to design agents, not to manipulate them, only the platform can have access to agents at runtime. Since the platform is a trusted entity, complete isolation and autonomy of agents are guaranteed.

The definition of an agent template is partitioned different sections. These sections hold respectively: the declaration of Knowledge fields shared among the different behavior instances; the declaration of behavior instances; the initialization and the clean-up code for the programmed template. These last two sections mostly contain C# [26] code. The template for agents definition provided with APX, is more restrictive than the general purpose class construct found in every object-oriented programming language. Inside the agent template, arbitrary methods and fields cannot be defined, neither does the notion of inheritance belongs to this template. This is a clear architectural decision since the agent template has been designed to localize code inside behavior templates. Such organization of the code for the agent template stems directly from the underlying agent model used in the AgentService platform, and helps the developers to follow this model more naturally.

5.3 The Knowledge Template

The knowledge template shapes a set of related data that need to be shared between different behaviors inside an agent template. The knowledge template must contain those data that need access control since they are referenced from different threads of execution. This template has been specifically designed to contain all the informations that need to be exchanged among the different activities an agent performs.

The knowledge construct is introduced by the knowledge keyword and is quite similar to a Pascal [25] record. All the types supported by the Common Language Infrastructure are admitted as fields for a knowledge template, but not the templates introduced with APX. The manipulation of knowledge instances is limited: they cannot be instantiated from scratch or assigned. Knowledge instances are declared in the agent template definition or in the behavior template definition. These declarations are sufficient for the automatic initialization of the corresponding elements. Such instances have visibility only inside special blocks that provide a secure context in which the fields contained can be accessed without race conditions. Knowledge instance fields are accessed by means of the dot operator as in the case of an object field. Outside a freeze block these instances loose their visibility and this approach guarantees the proper handling of shared objects by design.

A knowledge is the minimal bunch of data that can be requested for a thread-safe access: hence, it is important to declare knowledge templates that contain data which need to be used together. This is the main criteria that programmers have to follow, when partitioning data among different knowledge templates. Knowledge templates can be thought as components as well that can be reused when defining different behaviors and agents.

5.4 The Behavior Template

The behavior template shapes the concept of an agent activity: this can be a functionality or a goal-driven task. Activities normally implement algorithms or protocols that do something. Behaviors templates can be considered like code-classes in a generic software project and they are the entry point for programmers that develop code by using APX. Behaviors can contain methods and fields but no inheritance is provided for such templates. Behaviors have been designed to be self-contained objects, or components: these templates can communicate with other behaviors by means of the knowledge instances declared in the template, and with objects or legacy software in any way developers may wish to provide (e.g., database access, socket channels, etc.). Reuse is an important feature of behaviours: these templates encapsulate an activity and require only the use of a limited set of knowledge templates, if necessary, to be plugged into an agent definition and let them work.

The behavior template structure mostly resembles the structure of a class but no inheritance is offered to the programmer. Behaviors definitions can contain fields, methods as in a common class, and have a particular section called body that is the entry point for the execution of the behavior. Also declaration of knowledge that need to be used by the behavior are part of the definition of this template

Behavior templates offer to developers all the elements they need to develop powerful and flexible code. Since the methods are almost identical to the methods in a C# class, a rich set of programming constructs and resources are provided: programmers can code algorithms directly inside behaviors or delegate the computation to any other component (probably developed in C#, C++,...), and then reference or instantiate it inside the behavior code.

The possibilities to manipulate behavior instances are limited in APX, but this is not a restriction since the behavior instances that characterize an agent template are automatically managed by the agent instance at runtime. The creation and the assignment of behavior instances happen at runtime and there is no need in APX to manage
such aspects. Using APX, behaviors can only be defined and configured.

5.5 Freeze statements

Within APX the knowledge instances are accessible only inside a thread-safe context: the freeze statement has been designed to set up transparently such a context. Programmers do not have to worry about locking protocols, they just open a freeze block and declare all the knowledge instances they may wish to use.

Freeze statements are really similar to the C# lock statement or the Java synchronized statement, except that the freeze statement has been specifically designed to handle the locking of knowledge instances. The freeze statements automatically update the value of all the fields within a knowledge and give scope to these objects that could not be referenced otherwise. The freeze statement is the only allowed synchronizing construct within an APX based system: at a first look, this imposition can appear reductive but, it is easy to verify that interoperability among languages in the CLI copes with this lack.

5.6 APX Compiler

APX provides the developers a set of extensions to the object oriented model to develop agents. These extensions are mostly new language constructs that represent the agent model adopted by the AgentService platform. Developers who want to build agents have to implement such templates and customize them to their particular needs by using the APX programming interface. The APX compiler then builds the proper classes according to the requirements of the AgentService platform: templates designed are translated into classes, customized via inheritance from the template classes designed inside the AgentService class libraries. The outcome of the compiler is pure object-oriented code: the output of the compilation process is always a class library of agents, behaviors, and knowledges. The APX compiler parses the templates defined and creates the corresponding class definitions. All the syntactic sugar provided with APX is silently substituted by the compiler with the automatized tasks required by the AgentService programming model.

Thanks to the work of the compiler, agent developers deal with an easier agent programming interface and they can concentrate more on the agent oriented aspects, rather than on implementation issues.

5.7 Programmatic Example

In order to illustrate how to develop an agent oriented solution with APX, the problem of a multi-robot planning has been taken as a case study. Robots in with different goals such as moving objects, or reaching given spots, are situated in the same environment. Obstacles may occur on the robots tracks. It is necessary to guarantee that each robot does not crash with obstacles or with other robots, and then to make the robots pursue their goals. By using APX and AgentService, the problem can be studied and simulated by creating a multi-agent system in which a robot is an agent. All the robots can be considered instances of the same agent template, only differing in the configuration parameters. A single agent template needs to be defined and the decomposition of its aggregate behavior in the different activities it needs to perform, leads to the definition of behavior and knowledge templates that it is made up of.

The information every agent-robot needs, can be described as follows: the robot pursues a goal and such a goal can be modeled with a proper knowledge that holds the desired position of the robot, or any other thing that such actions need to be performed; the robot needs to know its internal state, such a state can be modeled with knowledge holding the current position and velocity, the planned trajectory; the robot interacts with the environment by using sensors and actuators and builds a map of the environment during the on-line planning. A knowledge that holds the occupancy grid is therefore necessary.

Three behavior templates can be designed to model all the activities of the robot: the operator, the planner, and the communicator templates. The operator behavior template manages sensors and actuators and uses the knowledges that hold the internal state of the robot, and that shape the environment. The planner behavior template perform the on-line path planning and uses all the three knowledges introduced previously. The communicator behavior template communicates with the other robots to exchange position information, and keeps updating the occupancy grid.

The proposed model supports modularity and reuse: the different behaviors can be reused to compose different kinds of robots, since they are not tightly coupled and interoperate by using knowledge objects. Both the planner and the communicator can also be used to model avatars or other agents that do not necessarily represent robots, but yet need equal or similar functionalities.

6 Comparison

The related works that mostly resemble APX are APL and JACK: APL is strongly based on the BDI architecture, whereas JACK lets developers implement different agent architectures. The APX package does not provide a native support to the BDI architecture but agents can be programmed using a BDI technique by coding the proper behaviors and knowledges.
By using APL, programmers are bound to the BDI model of agents since elements of the language such as variables and methods, are respectively beliefs and intentions; in APX developers are free to organize behaviors as they prefer, even using a BDI approach. APL offers automatic persistence of beliefs: this is a useful feature because programmers do not have to worry about saving such data. The APX package provides persistence with the knowledge templates: these types are automatically persisted by the platform that manages the running agents. In the case of APX the types produced by the compiler are designed for persistence but such aspect is managed by AgentService. Another important difference is the fact that APL is a real programming language with first class types, and has been designed to be used alone, APX is a package containing a set of language extensions, designed to be fully integrated with the Common Language Infrastructure and its components. Because of these reasons APX does not come with a full featured programming language.

The JACK Intelligent Agents System is really similar to the APX package. As the latter, the former provides language extensions to model agent oriented concept, an agent programming framework and other elements, even if JACK is particularly suited to develop agents for simulation systems. Like JACK, APX too has been designed to be fully integrated into an object oriented environment and all the components exposed by means of a certain technology. In JACK the features introduced with the extensions to the Java language let the programmers define plans, agents, and modeling events. Actually APX has a more restricted set of extensions even if they can be strongly customized. This is the first version of the APX package and more features will be added in the future.

In section 3.2 the programming language APRIL was discussed: such language provides features to develop agent oriented application, but it mainly concerns with concurrent processes, and there are few similarities with APX. The other agent programming languages presented in section 3.1, have been designed to build formal specification of agents and are significatively different in their structure from APX.

7 Conclusions

In this paper the APX package has been presented. APX is a set of extensions to the object oriented model that provides templates and facilities to build agent-oriented applications. The main design goal of APX is not to be another new agent-oriented programming language, but rather to provide developers of agent-oriented applications with a friendly interface to the agent oriented programming. Such interface is constituted by a set of templates that models the key aspects of the adopted agent model. This model is the same as that of the AgentService platform that is the target environment for agents designed with APX. Developing agents for a specified framework normally implies object oriented programming, and the application of some architectural patterns required by the agent programming framework chosen. APX by the use of a tailored compiler tries to automate all the patterns required by the AgentService platform. As a result, developers are not bothered with some technical issues related the object oriented implementation, and they can concentrate more on the agent oriented aspects of the software project. APX has been designed to be fully integrated with the underlying technology that is the Common Language Infrastructure. By means of language interoperability APX can benefit from a wide range of reusable components, and deliver to other programming models such as the object oriented one, programming tasks that are not clearly agent oriented but needed.

The agent programming extensions are still a work in progress: new features and a richer set of functionalities will be provided in the next future. Since one of the most important design issues has been the desire to provide an effective tool to develop agents, a deeper integration with the AgentService platform functionality is the next incoming task.

References


[6] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, Design Patterns. Elements Of Reusable Object


