SOFTWARE ANATOMY OF A KGP AGENT

Alexander Yip [a] Jeremy Forth [b] Kostas Stathis [b] Antonis Kakas [a]

[a] *Department of Computing, University of Cyprus, Cyprus*
[b] *Department of Computing, City University, UK*
{a.yip,jforth}@imperial.ac.uk, kostas@soi.city.ac.uk, antonis@cs.ucy.ac.cy

Abstract

While there is much work in the literature on agent component technologies, there are comparatively few results on the engineering issues arising from the integration of these technologies into a complete and coherent agent architecture. This paper addresses these issues from both implementation and general architectural design point of view. The KGP (Knowledge, Goals and Plan) agent model developed in the EU SOCS project (IST-2001-32530) is used as an example to illustrate these engineering issues.

1 Introduction

Using a multi-agent paradigm to implement an intelligent software system has many advantages over more usual imperative methods that depends on custom design of executable code for each particular application. In this paper we discuss the design process lying behind the engineering of a cognitive agent. The design process requires two main tasks to be undertaken. Firstly, new component technologies are created to fulfill individually identified cognitive requirements. Secondly, those component technologies are integrated together as a coherent system within a suitable architectural framework.

Due to complex interaction of agent components, it has been found that design challenges encountered at the system design level are comparable with the challenges present in developing component technologies themselves. In response to calls for further investigation into architectural challenges and solutions in the pursuit of structured agent design [2], this paper address the topic by presenting an agent architectural framework from an engineering point of view. We have identified several distinct requirements of an agent architectural design, represented those requirements as architectural properties and furnished design solutions that possess those properties.

To illustrate the implementation issues, we use an agent platform called PROSOCS [16] as an example. The PROSOCS system is highly expressive and declarative in nature due to the logical formalization used in both system and agent specifications. This allows direct execution of the logical specification of an agent. PROSOCS was developed in the SOCS project [15] as a prototype platform to verify the KGP (Knowledge, Goals and Plan) agency model [11] and the Cycle Theory control model [3, 10]. (see diagram 1). These technologies will be briefly introduced in section 1.1. But since the main objective of this paper is to discuss the engineering perspective, the reader is referred to the referenced resources for a detailed account of PROSOCS.

The KGP model is presented in an Object-Oriented formalism using UML. The model, described pursuant to object-oriented standards, should demonstrate to an application developer how to design a modular agent system. In addition, by explicating the standard interfaces of the system, a developer is able to enhance functionality and to create future extensions in a modular way.

1.1 Agents with Minds

Existing agent frameworks differ in design principles and realization techniques that significantly alter their behavioural characteristics. The architectures employed among existing agents differ in the degree of anthropomorphism (where agents are designed to model human reasoning) versus goal orientation (where agents are designed to produce actions for a specific task completion).

There is a further difference in architectural choice based upon a division made between high level cognitive function and lower level world interface function. The PROSOCS architecture [17] is an example of an architecture employing a separate 'mind and body' to instantiate these functions. The body senses what is external to the agent using sensors that can access the current state of the external world. Observations are then stored in the body state which is accessed by the mind to produce actions for execution. Actions are presented to the body and enable it to choose what to do next. The advantage of this architecture is that the mind may be designed in a domain independent way, separate from the world interface which could for example consist of motor controllers and actuators or specialist domain-specific software interfaces. As far as we are aware, KGP is unique in this characteristic.

The KGP model [11] considers knowledge about a domain, goals and plans for that domain to represent the total cognitive information description at any point in time. The KGP agent model has a similar approach to the BDI mentalistic reference model [13, 4]. Some examples of BDI agent system are Jadex [8], Retsina [14] and Jason [9], they will be further discussed in section 4.

An agent's mind may be partially defined using a model such as KGP or BDI. However neither of these models describe how the mind should sequence its activity at run time. This is the job of a control model, which defines the transitions between the different cognitive capabilities of the mind. In the PROSOCS system, Cycle Theory is used as a complemental control model for the KGP model.

The role of the Cycle Theory [3, 10] is to control the sequence of the internal transitions that the agent does in its life. It regulates these 'narratives of transitions' according to certain requirements set by the designer of the agent. The aim of the Cycle Theory is therefore twofold. Firstly, it aims to control and characterise the operational behaviour of an agent by giving a form of intelligent control that is based on the declarative knowledge of the agent, responsive to changes in its external environment. Secondly, through the use of different selection functions and in particular the heuristic selection functions using criteria such as urgency, utility and significance, Cycle Theory aims at inducing different preferred behaviour in an agent.

1.2 Overview of the paper

The structure of this paper is as follows. The paper is divided into two main parts, in section 2 we present the KGP model in UML. In section 3 we present a list of architectural features that we believe to be important for practical software agent systems; then the features of the KGP model will be highlighted in answer to those architectural properties. In the related work section we introduce some well known agent systems as subjects for comparison, and give a conclusions with reference to possible future extension.

2 An Implementation Architecture of KGP

2.1 The KGP Model

As part of the SOCS project [15], we demonstrated that it is possible to implement software agents in a logical fashion using the KGP model [11] and the Cycle Theory [3, 10]. Figure 1 shows the conceptual model of a software agent (known as a computee in the project).

As shown in figure 1, there are four main component types in a computee – Control, Transitions, Capabilities and Knowledge. Capabilities provide a means to encapsulate the foundational mechanisms for realizing specific forms of reasoning possible by the agent. These reasoning capabilities, implemented using the agent's underlying logical languages, are distinguishable as distinct operations because they each have a high level semantic specification (e.g. planning). Each capability is defined to work in conjunction with named knowledge bases of the agent. A sequence of capabilities are then combined together to form a coherent agent behaviour. The PROSOCS implementation of KGP provides the following reasoning capabilities:

- Planning – Introduce action to achieve the current goal
- Reactivity – Reaction to external stimulants
- Goal Decision – Reasoning about current goal
- Temporal Reasoning – Reasoning about time

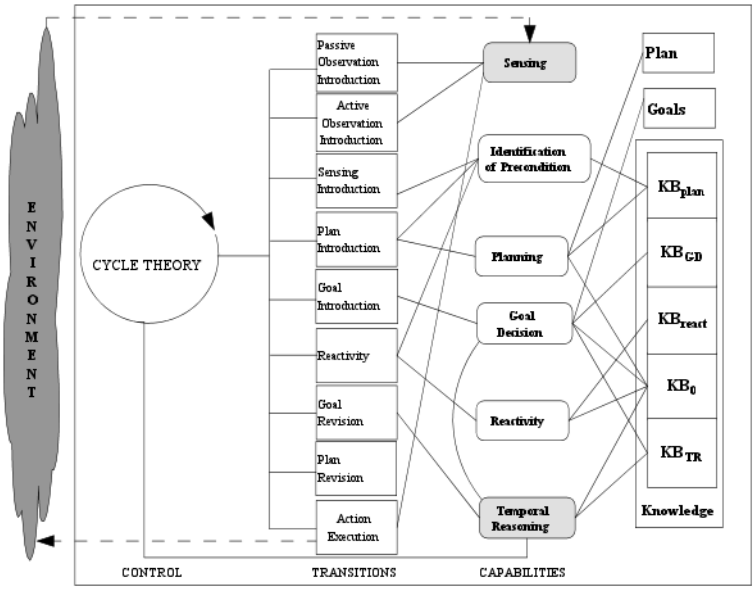

Figure 1: A conceptual model of a Computee

- Sensing – Sensing the environment

For the formal definition of these capabilities, please refer to the project document [6].

Knowledge bases provide input to the capabilities, which in turn contain proof procedures matched to each type of knowledge base. A transition from one agent state to another is dependent on the set of capabilities. A transition defines a possible step (possibly an action) to take while a capability defines the function of that action. The operation trace of a KGP agent is a sequence of transitions. Each transition corresponding to an external action, or internal event. The particular choice of transition types are determined by the Cycle Theory which may vary the choice of transitions depending on the contents and priorities contained in the Cycle Theory knowledge base.

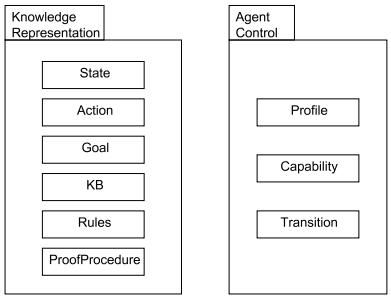

Figure 2: Two packages of computee components

The software package structure recognises a distinction between knowledge representation and reasoning from agent control. Figure 2 shows package partitioning by placing objects concerned with internal state within Agent Control and objects concerned with the domain of situation within Knowledge Representation.

A single KGP agent mind is represented by the top level object KGPAgent (Figure 3). It is a collection of objects representing the cognitive subsystems for agent knowledge and control. Each cognitive function is realized by a Capability which ties together a particular knowledge reasoning operation with a unique name. This capability name is used by the Transition object to link particular reasoning operations with agent control as maintained by the Control object. In this architecture, queries to knowledge bases are generated by Capabilities in response to Transitions.

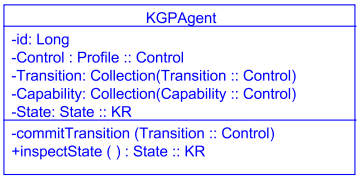

Figure 3: A KGP Agent Mind

2.2 State of an Agent

The cognitive subsystems all have some connection with the agent's top level state (State::KR), encompassing at any particular time instance, the agent's knowledge-base contents (KB::KR), current goal (Goal::KR) and plan (Plan::KR) (see figure 4). The knowledge bases are subdivided into specific knowledge types supporting particular functionality. Goals and plans are derived forms of information representing an agent's target and the anticipated steps needed to be taken to reach that target. The mechanisms and data for triggering updates to the agent's state objects are realized in the form of transitions taken in response to controlled status, and consequent alterations to knowledge base contents.

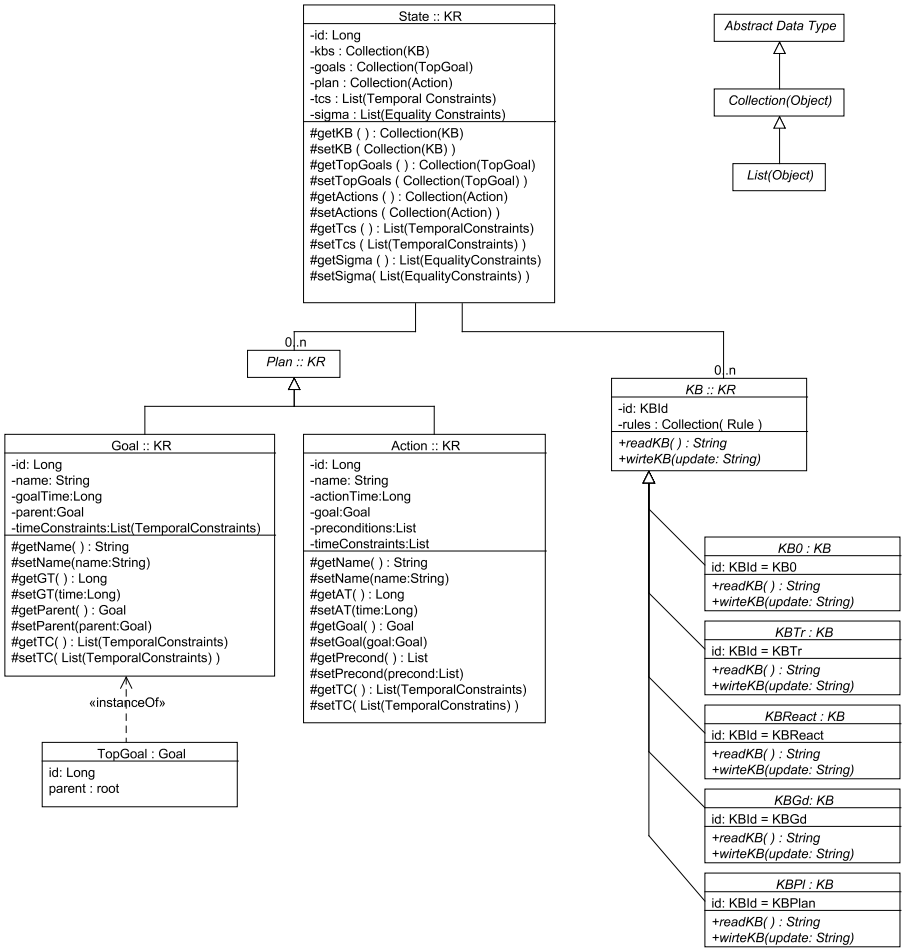

Figure 4: UML diagram of the State components

2.2.1 Knowledge Bases

Knowledge bases are constructed from the abstract container class KB::KR, comprised of objects of class Rule (see section 2.2.2). The definition of knowledge base types is related to the definition of capabilities (see section 2.3). Table 1 shows the knowledge bases defined in PROSOCS.

KB_0	stores the observations of the environment
KB_{GD}	goal decision knowledge base contains information that may be used to derive or choose goals. This knowledge contains preference information, used to arbitrate amongst competing alternative choices
KB_{TR}	contains causal relational information about the domain, and also temporal reasoning formulae that determine how changes happen with time
KB_{Plan}	contains knowledge sufficient for describing causal relations in the domain, and also axioms sufficient for planning
KB_{React}	define the reactive response (to external stimuli) of the agent

Table 1: KBs and their functions

Abstract class KB::KR therefore provides a generic template for a combined knowledge base and reasoning system allowing the encapsulation of specific differences in reasoning according to knowledge representation, and proof procedure employed. The proof procedure is a specialized piece of reasoning software designed to be invoked in response to queries.

Each instance of KB class makes use of an internal representation for its knowledge that is specific to the particular type of KB.

2.2.2 Rules

There are two types of rule in the KGP model, they are distinguished by the reasoner used to reason about them. An abstract class Rule::KR is used as a template for this type of class, each with a set of pre and post conditions. See figure 5.

Figure 6 shows the ProofProcedure class and its instances. A Ciff rule is written in Ciff (constraint if and only iff) logic and is represented by a ciffRule object. All knowledge base rules are of this type and they form the abductive logic programming part of the system. It is also used in the temporal reasoning mechanism which implements an event calculas reasoner.

Each Gorgias rule is a logical forumla written in a logical language with suffucuent expressive power to represent priority information. Each rule is represented by a gorgiasRule object. Its main use is to describe the transitions of an execution cycle and to encode the preferences in a profile.

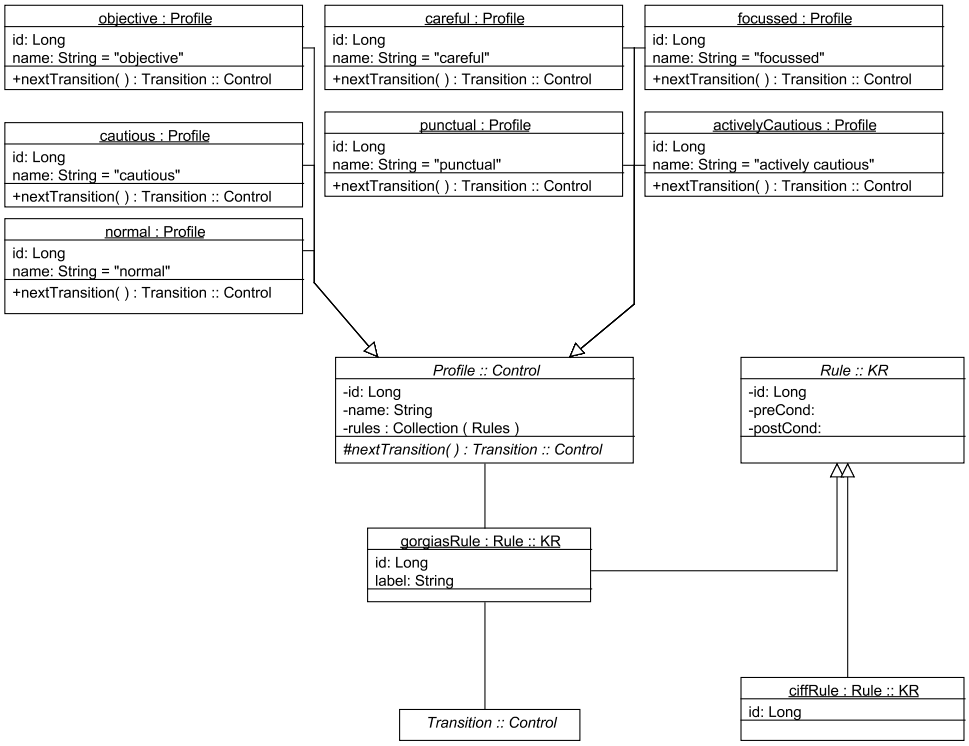

Figure 5: UML diagram showing Profiles

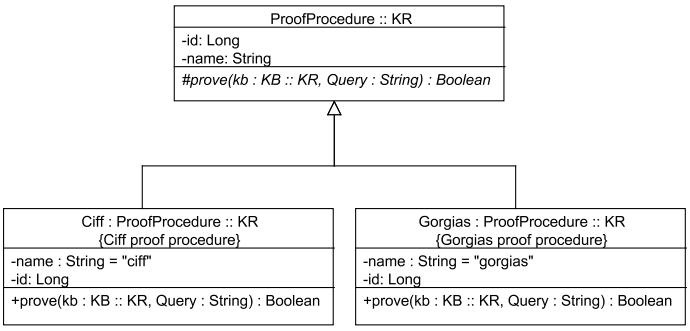

Figure 6: UML diagram of the ProofProcedure components

Transition	Input for Selection Function
GI (Goal Introduction)	*none*
RE (Reactivity)	*none*
GR (Goal Revision)	*none*
PR (Plan Revision)	*none*
POI (Passive Observation Introduction)	*none*
PI (Plan Introduction)	a set of goals selected for planning
SI (Sensory Introduction)	a set of preconditions of actions (associated with the parent of these actions) selected for sensing
AE (Action Execution)	a set of actions selected for execution
AOI (Active Observation Introduction)	a set of fluents to be actively sensed

Table 2: Selection functions of various Transitions

2.2.3 Plan, Actions and Goals

The other subclass of State is Plan::KR. Since plans are comprised of actions, each of which is associated with a particular goal in a goal tree, the abstract class Plan::KR acts as a container for Goal::KR and Action::KR. A Top goal is the highest level goal, and is a notable instance of the goal class.

The selectionFunction in a Capability::Control object takes the current state of the agent as input and generate a new set of Goal::KR and Plan:KR as output.

2.3 Capabilities

Figure 7 shows the definition of an abstract capability class[1] with specialized concrete classes corresponding to each specialized Capability listed above. Reasoning capabilities obtain their invocation and sequence through agent Transitions.

Each capability will have a correlated set of knowledge base which provide input data to the reasoner. A reasoner for a capability is called a Proof Procedure and in PROSOCS there are mainly two: Ciff (constrained if and only if) and Gorgias (a logic with priority).

Each capability also provides a standard selection function which select the next possible plan based on the current state of the agent. [6]

2.4 Transitions

Transitions (see figure 7) are an abstraction of the agent's state modification mechanism within the architecture. Transitions increment the agent's State and make use of calls to particularly selected Capabilities. Transitions determine the next transition of the cycle theory made and contain memory of the previous transition. For a detailed specification of the transitions, please refer to the SOCS project document [10]. In table 2 we document the input parameter for the Selection Functions of each of the transitions.

Each transition defines a function which updates the state of the agent based on the existing state and the result of the action performed by the capability related to the transition. Figure

[1]To aid the clarity of the diagrams, some intuitive access functions were omitted from the diagrams. For example, setId and getId were assumed implicitly.

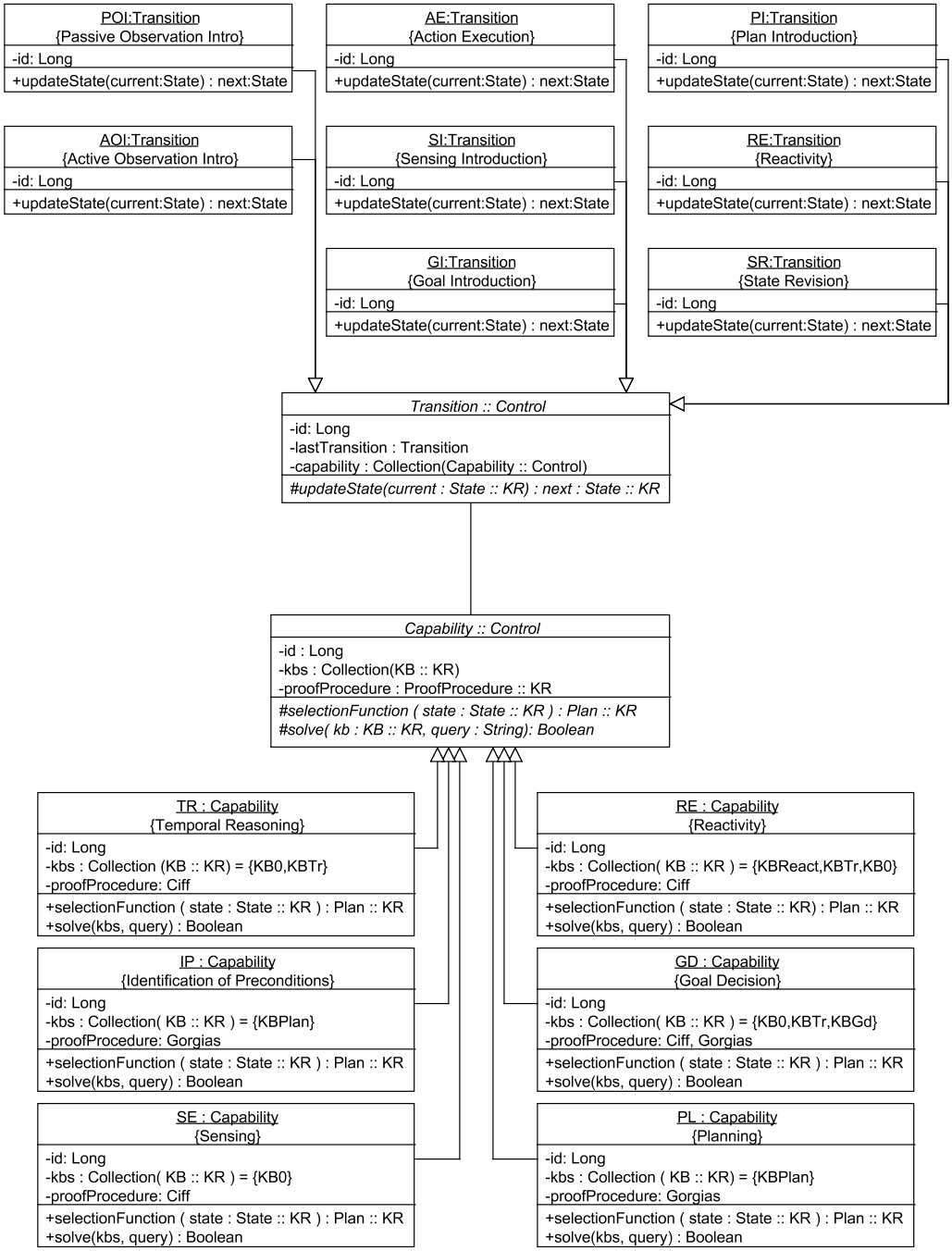

Figure 7: UML diagram showing Transitions and Capabilities

7 also shows the relationship between Transition and Capability. [these two components are realized as enabling conditions in each of the profile module]

2.5 Profile of Behavior

A profile defines the higher level operational behaviour of an agent and is independent from the domain specific information (KBs). A profile defines a standard function which returns the next possible transition (see 2.4) based on the behaviour defined by the profile.

A profile of behaviour (see figure 5) is a collection of Gorgias rules (see 2.2.2) that assign priorities to the transitions based on the current state of the agent. For example, an agent may prefer to execute a sensory introduction over an action execution if it is not certain of the state of the environment.

Because the preference is only defined by a priority, so if the precondition of the most favourable transition does not hold, the next preferable action will be tested. The standard control theory of the KGP model is called the cycle theory because the transitions are purposely defined in a cycle so the agent will not terminate due to a single failed transition. Instead the agent will attempt the next available action, even if the action has been tried before, as it is possible that the precondition may hold in the future.

3 Principles of Agent Design

3.1 Rationale of the Architecture

The design philosophy of KGP model mirrors the goals of computational logic in general. The use of declarative expressions of knowledge about a domain coupled with dynamically generated actions taken to realize an outcome considered desirable. PROSOCS is unique among multi-agent system toolkits as the development paradigm keeps a tight correlation between the specification and implementation, so to provide a bridge between concepts and implementation.

The PROSOCS architecture has been designed to allow for maximum adaptability in terms of customisation to support alternative component technologies, including domain-specific customisation of the basic functions. It will support component interchange, replacement, and addition within its architectural framework.

Agent cognitive functions are normally heavily mutually dependent. In order to separate and reduce interdependency, major cognitive subsystems have been identified to realize the cognitive functions with an associated method of architectural separation. These subsystems are, knowledge representation, knowledge bases, reasoning type, reasoning capabilities, transitions, and agent execution cycle theory.

For any rational agent architecture employing declarative knowledge, a reasoning system and an activity controller, we identify the following postulates of good architectural design:

Mind-Body Separation Separation of domain-independent high-level reasoning from domain-specific coupling and control interfacing.

This represents the recognised and important characteristic of modern software engineering practice – code re-usability. By separating general domain reasoning capability from the specific underlying physical interfaces, it is possible to adapt an agent design to a different application with minimum reconfiguration.

Knowledge Representation Independence Unrestricted knowledge representations across different knowledge bases.

At run time a software agent needs to handle different types of information – temporal information to make sure it meets the deadlines; planning information to plan the course of actions and domain specific knowledge. As these information usually have different representations and require different reasoning mechanisms, it is more flexible and expressive by supporting different knowledge bases rather than trying to integrate all the information in to a single source.

Reasoning System Independence Separated foundational reasoning systems, each associated with a particular type of knowledge representation.

An agent makes use of different types of knowledge representation, and each representation has an associated reasoning system which may be applied to knowledge of the appropriate type.

It is beneficial for an agent architecture to cater for several different reasoning systems in such a way that one is not dependent on the others.

Cognitive Independence Separation of internal cognitive functions.

Knowledge representation and reasoning functionality can be deployed in selected ways to realise a cognitive function. An architecture should separate one cognitive function from another and define a common interface with knowledge representation and reasoning.

Control Independence Separation of the internal agent activity condition control from cognitive function of that activity.

The cognitive function should be independent from the execution of internal functions. In other word an agent's reasoning should determine what function is being executed rather than following a hard wired sequence.

3.2 Benefits of Agent Customisation

In section 2 we presented the components of the KGP architecture using an object based model. What we have presented so far is the basic version of the architecture which was used to implement the PROSOCS system [16]. In this section we are going to discuss how this modular construction of KGP agents allow them to be customisable to suit different applications.

There are two ways to customise a system: Behavioural customisation which changes the general settings of an agent within the existing architecture; and Architectural customisation which modify the implementation of the system components.

3.2.1 Behavioural Customisation

The most commonly customised parts are the components of the *State* of an agent. Utilising an OO metaphor, this is like an object instantiation using the KGP model as the generic class. An agent needs a set of *goals* to give it a purpose and a set of domain specific knowledge and *actions* to achieve the goals. As a physical agent, a KGP agent also needs a set of rules to handle external stimuli (communications and environmental observations) and to reason about temporal events. These are stored in the knowledge bases as described in section 2.2.1.

These domain specific customisations are needed to prepare an agent for a specific application. For more information on how to write the domain specific knowledge base, please refer to the PROSOCS user manual [1].

3.2.2 Architectural Customisation

There are two reasons to customise an architecture at the implementation level – The architecture can be extended to support a new functionality; or sometimes a module of different implementation may be used in place of the standard design.

We identify below the major components within KGP in which we anticipate customization to take place to suit a particular purpose, and for which accommodation of alterations has been made in the architecture. The UML diagrams presented in the last section will serve as a guide to identify the dependence between the components as the change of one component may affect the others. For each component, the dimensions of modification are discussed in conjunction with their motivations and reasons.

Proof Procedures In the PROSOCS implementation, Event Calculus [12] was chosen as the temporal model. However, the developer may choose to have a different temporal reasoning model, an example is Interval logic. To implement this change, it will require a new proof procedure for reasoning about the temporal constraints. As shown in figure 4, the syntax of goals and actions will be affected as they both have temporal constraints as attributes. As a knowledge base is just a collection of actions and goals, in turn the capability objects have to be updated to support the new proof procedure as well (see figure 7).

Another proof procedure is the Gorgias reasoner. It is used to reason about the preferences in a profile (see figure 5). If a different reasoner is used, the new logic model needs to be incorporated into the system. The first major impact is the Cycle Theory as it relies on preferences to maintain a continuous execution cycle. It is necessary to verify whether the new logic model can support the cycle theory, otherwise a new control model is needed to replace Cycle Theory as the control model (figure 1).

Capability Currently there are six capabilities defined (see figure 7). It may be necessary to introduce new capability to the architecture in the future. A new capability can be defined by extending the abstract *Capability* class. For example, a new learning capability may be introduced to extend the KGP model.

However, once a capability is defined, it needs to be referenced in *Transition* so that it can be part of the execution cycle – which may involve defining a new transition.

Transition There are seven predefined transitions (see figure 7) and they define what are the legal states of an agent's mind. If a new transition is defined, it is necessary to also reference it from within a *Profile* so that it will form part of the cycle theory.

Profile of Behaviour A profile is a collection of specific *transitions* (see figure 5). A profile of behaviour defines the operational behaviour of an agent, for example a profile may give higher priority to state revision while another prefers observation introduction. A profile is defined in a logic that supports preferences. By using a Gorgias reasoner, it is possible to determine the preferred transition at any given state of the agent.

While implementing a profile, sometimes it is necessary to impose different enabling conditions on the capabilities. In such case, the selection function (see figure 7) of a capability needs to be modified to suit the intended profile.

3.3 Discussions

In 3.1 we identified five important properties of a good agent architecture. In this section we summarise how these features were satisfied by the KGP model and the properties supported by the PROSOCS implementation.

Mind-Body Separation is realized by the separation of domain independent high level reasoning from domain dependent actuation. A PROSOCS agent has a clear division between the mind and the body. It is possible to reuse the same mind to control different agent bodies, given that the bodies are composed of the same kind of capabilities. Please refer to PROSOCS user manual [1] on how to prepare domain dependent knowledge.

Knowledge Representation Independence is realised by the support of separated KBs. Please refer to 2.2.1 for a list of Knowledge Bases available in KGP.

Reasoning System Independence is supported by the separation of proof procedures within the mind. Cognitive functions are allowed arbitrary use of knowledge bases, which draw on the required proof procedures. In KGP, each Knowledge Base has the use of an allocated reasoner in the form of a suitable proof procedure. This separation allows a specialised reasoning mechanism to be used e.g. Gorgias for priority reasoning. This allows new knowledge or reasoning capabilities to be introduced into the PROSOCS system.

Cognitive Independence is realised by the definition of agent Capabilities.

Control Independence is achieved in two ways: By making the capabilities as individual self-contained states, it allows the execution cycle to be defined at the behavioural level rather than at the architectural level; By defining cycle theory to control the transitions between states, the result is a more flexible and logically sound architecture. PROSOCS supports profiles of behaviour which allows different operational behaviour to be selected at run time.

4 Related work

In this paper we have shown how to build a modular agent architectural framework. In section 3.1 we identified a list of generic architecture properties and have shown how our framework satisfies these properties. To contrast our work with existing agent architectures, this section introduces a selection of well known BDI agent systems – Jadex (with Jade) is a generic agent system; Retsina is a multi-agent system with predefined agent classes, while Jason is an Agentspeak(L) based logical agent platform. The main features of the aforementioned systems are reviewed and compared with the properties we defined in section 3.1. We summarised the comparison in table 3.

Jadex [8] A Jadex agent has its internal memory structure modeled on the notion of mental attitudes. Jadex implements a BDI mental model for Java agents based on the Jade [7] middleware platform, which is FIPA compliant [5]. The FIPA specifications govern the interoperability between different platform implementations.

Jadex implements a reactive agent architecture using a library of plans. The internal states of an agent i.e. beliefs, plans and goals are defined in a XML format called the agent definition file. Inside Jadex all mental objects are first class Java object, which means the manipulation of the mental objects are done in the imperative programming level. It uses three distinct types of goal: 'achieve', 'maintain', and 'perform'.

Retsina [14] Retsina adopted a detailed design paradigm to support specialised agents rather than agents with easy adaptation to arbitrary general purpose applications. Within the Retsina multiagent architecture, each agent is designed and implemented to perform a specific task. Predefined common agent classes are used to assist in the overall system design, these include interface agents (helping users), task agents and information agents (controlling multiple information sources). These agents are then adopted to different situtations with domain knowledge.

Each agent employs four concurrent modules for communicating, planning (using a plan library), scheduling and monitoring. Knowledge is stored in a single collection of rules called the Domain facts and Beliefs database.

Jason [9] stands for A *J*ava-based *a*gentSpeak interpreter used with *s*aci for multi-agent distribution *o*ver the *n*et. Jason uses Java to provide the bodily functions for the agents: an AgentSpeak(L) interpreter to execute the agent's mind and to support communication to other entities. The architecture can be extended by coding new Java functions.

AgentSpeak(L) is an abstract language based on the BDI architecture. Sometimes agents created using AgentSpeak(L) are referred to as reactive planning systems.

Properties	Jadex (with Jade)	Retsina	Jason	KGP (PROSOCS)
Mind-Body Separation	√	×	√	√
Knowledge Representation Independence	×	×	×	√
Reasoning System Independence	×	×	√	√
Cognitive Independence	√	×	√	√
Control Independence	×	×	×	√

Table 3: Comparing the features of different agent systems

5 Conclusions and Future works

To implement a software agent, creating the internal components is only half the story, how to put the components together is as important if not more. In this paper we addressed this concern in two ways.

Firstly, we presented a list of general properties we consider to be important to the architectural design of an agent system, then by using PROSOCS as an example we showed how they can be fulfilled.

Secondly, this paper serves as a road map on how to realise an abstract agent model by implementing it as a modular software agent system. This work has already been verified by the EU SOCS project, but by presenting the KGP model in a object oriented fashion and by providing relevant references, this paper proposes a more systematic approach for engineering a software agent system.

The majority of the PROSOCS development was done within the EU SOCS project. Although the project has ended, the project consortium carried on with the research and development of both the KGP model and PROSOCS. New findings since the end of the project have been documented in this paper. One particular area the project identified deserving of further investigation is the temporal reasoning system. If an agent system is to be useful for real time applications, it needs to function in 'realistic time' and to support synchronised operation among agents in a distributed fashion.

References

[1] Marco Alberti, Andrea Bracciali, Federico Chesani, Ulle Endriss, Marco Gavanelli, Wenjin Lu, Kostas Stathis, and Paolo Torroni. SOCS prototype. Technical report, SOCS Consortium, 2003. Deliverable D9.

[2] Fabio Bellifemine and Giovanni Rimassa. Developing multi-agent systems with a fipa-compliant agent framework. *Softw. Pract. Exper.*, 31(2):103–128, 2001.

[3] A. Bracciali, U. Endriss, A. Kakas, P. Mancarella, F. Sadri, K. Stathis, and F. Toni. Cycle-related properties: Preliminary report. Technical report, SOCS Consortium, 2004.

[4] P. Busetta and K. Ramamohanarao. An architecture for mobile BDI agents. In *Selected Areas in Cryptography*, pages 445–452, 1998.

[5] FIPA: Foundation for Intelligent Physical Agents. Home Page: `http://www.fipa.org/`.

[6] M. Gavanelli, E. Lamma, P. Torroni, P. Mello, K. Stathis, P. Moraïtis, A. C. Kakas, N. Demetriou, G. Terreni, P. Mancarella, A. Bracciali, F. Toni, F. Sadri, and U. Endriss. Computational model for computees and societies of computees. Technical report, SOCS Consortium, 2003. Deliverable D8.

[7] Jade: Java Agent DEvelopment framework. Home Page: `http://sharon.cselt.it/projects/jade/`.

[8] Jadex: BDI agent system for Jade. Home Page: `http://vsis-www.informatik.uni-hamburg.de/projects/jadex/`.

[9] Jason: Java-based Agentspeak interpreter used with Saci for multi-agent distribution Over the Net. Home page: `http://jason.sourceforge.net/`.

[10] A. Kakas, P. Mancarella, F. Sadri, K. Stathis, and F. Toni. A logic-based approach to model computees. Technical report, SOCS Consortium, 2003. Deliverable D4.

[11] A.C. Kakas, P. Mancarella, F. Sadri, K. Stathis, and F. Toni. The KGP model of agency. In R. Lopez de Mantaras and L. Saitta, editors, *Proceedings of the Sixteenth European Conference on Artificial Intelligence, Valencia, Spain (ECAI 2004)*. IOS Press, August 2004.

[12] R. A. Kowalski and M. Sergot. A logic-based calculus of events. *New Generation Computing*, 4(1):67–95, 1986.

[13] A. S. Rao and M. P. Georgeff. Modeling rational agents within a BDI-architecture. In Michael N. Huhns and Munindar P. Singh, editors, *Readings in Agents*, pages 317–328. Morgan Kaufmann Publishers, San Francisco, CA, USA, 1997.

[14] Retsina: Multi agent system. Home Page: `http://www.cs.cmu.edu/~softagents/multi.html`.

[15] Societies Of ComputeeS (SOCS): a computational logic model for the description, analysis and verification of global and open societies of heterogeneous computees. IST-2001-32530. Home Page: `http://lia.deis.unibo.it/Research/SOCS/`.

[16] Kostas Stathis, Antonis C. Kakas, Wenjin Lu, Neophytos Demetriou, Ulle Endriss, and Andrea Bracciali. PROSOCS: a platform for programming software agents in computational logic. pages 523–528, Vienna, Austria, April 13-16 2004. Austrian Society for Cybernetic Studies. Extended version to appear in a special issue of Applied Artificial Intelligence, Taylor & Francis, 2005.

[17] Kostas Stathis, Antonis C. Kakas, Wenjin Lu, Neophytos Demetriou, Ulle Endriss, and Andrea Bracciali. PROSOCS: a platform for programming software agents in computational logic. In Robert Trappl, editor, *Proceedings of the 17th European Meeting on Cybernetics and Systems Research, Vol. II, Symposium "From Agent Theory to Agent Implementation" (AT2AI-4)*, pages 523–528, Vienna, Austria, April 13–16 2004. Austrian Society for Cybernetic Studies.