References

[1] Becker J. D. (1969), *The modeling of simple analogic and inductive processes in a semantic memory system*. Proceedings UCA1-69, Washington DC, pp. 655-668.

[2] Boden M. A. (1992), *The Creative Mind*. Abacus.

[3] Bohan A., O'Donoghue D. (2000), *LUDI: A Model for Geometric Analogies using Attribute Matching*. AICS-2000 11th Artificial Intelligence and Cognitive Science Conference.

[4] Cianciolo A. T., Sternberg R. J. (2008), *Intelligence: A brief history*. MA: Blackwell.

[5] Ditchburn R. W. (1991), "Historical Introduction," in *Light*, NY: Dover Publication, Inc., Chapter 1, Sec 1.14,1.16, 1.18, pp. 10-14.

[6] Dreisiger P. (2008), *Artificial Working Memory: A Psychological Approach*. In Proceedings of the 16th School of Computer Science & Software Engineering Research Conference, Yanchep, WA, pp. 14–23.

[7] Evans T. G. (1964), *A heuristic program to solve geometric-analogy problems*. AFIPS 64 (Spring) Proceedings of the April, spring joint computer conference, pp. 21-23.

[8] Falkenhainer B., Forbus K. D., Gentner D. (1989), *The structure-mapping engine*. Artificial Intelligence, 41(1), pp. 1-63.

[9] Falkenhainer, B., Forbus, K., Gentner, D. (1986), *The Structure-Mapping Engine*. Proceedings of AAAI '86, PA: Philadelphia.

[10] Forbus K., Ferguson R., Gentner D. (1994), *Incremental Structure-Mapping*. 16th Cognitive Science Society, pp. 313-318.

[11] Forbus K., Gentner D., Law K. (1995), *MAC/FAC: A Model of Similarity-based Retrieval*. Cognitive Science, 19(2), pp. 141-205.

[12] Forbus K., Oblinger D. (1990), *Making SME Greedy and Pragmatic*. Cognitive Science Society, 12, pp. 61- 68.

[13] French R. M. (1995), "The architecture of Tabeltop" in *The subtlety of sameness: A theory and computer model of analogy-making*. The MIT Press, Chapter 3.

[14] French R. M. (2002), *The computational modeling of analogy-making*. Trends in Cognitive Sciences, 6(5), pp. 200-205.

[15] Gentner D. (1983), *Structure-Mapping: A Theoretical Framework for Analogy*. Cognitive Science, 7(2), pp. 155-170.

[16] Gentner D. (1983), *Structure-mapping: A theoretical framework for analogy*. Cognitive Science, 7(2), pp. 155–170.

[17] Gentner D., Forbus K. D. (1991), MAC/FAC: A model of similarity-based Retrieval. Cognitive Science, 19, pp. 141--205.

[18] Hart, W.D. (1996) "Dualism", in *A Companion to the Philosophy of Mind*, ed. Samuel Guttenplan, Oxford: Blackwell, pp. 265-7.

[19] Hoffman R. R. (1995), *Monster Analogies*. AI Magazine, 16(3), pp. 11-35.

[20] Hofstadter D. R., Mitchell M. (1995), "The copycat project: A model of mental fluidity and analogy-making". in Hofstadter, D. and the Fluid Analogies Research group, *Fluid Concepts and Creative Analogies*. Chapter 5, pp. 205-267.

[21] Holyoak K. J., Thagard P. (1989), *Analogical Mapping by Constraint Satisfaction*, Cognitive Science, 13, pp. 295-355.

[22] Holyoak K., Gentner, D., Kokinov, B. (2001), "The place of analogy in cognition" in *The Analogical Mind: Perspectives from Cognitive Science*, Cambridge, MA: MIT Press, pp. 1–19.

[23] Hummel J. E. Holyoak, K. J. (1996), *LISA: A Computational Model of Analogical Inference and Schema Induction*. Cognitive Science Society, 16, pp. 352-357.

[24] Hummel J. E., Holyoak K. J. (1997), *Distributed Representation of Structure: A Theory of Analogical Access and Mapping. Psychological Review*, 104(3), pp. 427-466.

[25] James W. (1950), “Chapter xiii: Discrimination and Comparison”, in *The Principles of Psychology*, Vol. 1, London: Dover Publications, pp.530.

[26] Jayatilleke K. N. (1978), The Contemporary Relevance of Buddhist Philosophy. BPS Online Edition (2009).

[27] Keane M. T., Brayshaw M. (1988), "Indirect Analogical Mapping: A Computational Model of Analogy”, in *Third European Working Session on Machine Learning*. Ed. D. Sleeman, London Pitman

[28] Keane M. T., Ledgeway T., Duff S. (1994), *Constraints on Analogical Mapping: A comparison of Three Models*. Cognitive Science, 18, pp. 387-438.

[29] Keane M., Brayshaw M. (1988), "The Incremental Analogical Machine: A computational model of analogy", in D. Sleeman (Ed.), *Third European Working Session on Machine Learning*, London: Pitman.

[30] Kokinov B. (1988), "Associative memory-based reasoning: How to represent and retrieve cases", in T. O'Shea & V. Sgurev (eds.) *Artificial Intelligence III: Methodology, systems, and applications*, Amsterdam: Elsevier pp. 51-58.

[31] Kokinov B. N. (1994), "A Hybrid Model of Reasoning by Analogy", in K. Holyoak & J. Barnden (eds.), *Analogical Connections, Advances in Connectionist and Neural Computation Theory*, 2.

[32] Kokinov B., Grinberg M., Petkov G., Kiryazov K. (2008), "Anticipation by Analogy" in *The Challenge of Anticipation*, Springer, pp. 185-213.

[33] Kokinov, B. and French, R. M. (2003) "Computational Models of Analogy-making", in Nadel, L. (Ed.) *Encyclopedia of Cognitive Science. London*, Nature Publishing Group, 1, pp. 113 - 118.

[34] Krawczyka D. C., Holyoakb K. J., Hummelb J. E. (2005), *The One-to-One Constraint in Analogical Mapping and Inference*. Cognitive Science, 29, pp. 797–806.

[35] Legg S., Hutter M. (2007), *A Collection of Definitions of Intelligence*. Frontiers in Artificial Intelligence and Applications, 157, pp. 17-24.

[36] McCarthy J. (1960), *Recursive functions of symbolic expressions*. Communications of the ACM, 3(4), pp 184-195.

[37] McCarthy J. (1962), *LISP 1.5 Programmer's Manual*, MIT.

[38] Minsky M. L. (1961), *Steps toward artificial intelligence*, Proceedings of the IRE, 49(1), pp. 8-30.

[39] Nersessian N.J. (2008), *Creating Scientific Concepts*. Cambridge, MA: MIT Press.

[40] Petrov A., Kokinov, B. (1998), "Mapping and access in analogy-making: Independent or interactive? A Simulation Experiment with AMBR", in K. Holyoak, D. Gentner, & B. Kokinov (Eds.), *Advances in analogy research: Integration of theory and data from the cognitive, computational, and neural sciences*, Sofia: NBU Press, pp. 124-134.

[41] Piyadassi Thera (1972), *The Psychological Aspect of Buddhism*, The Wheel Publication, Kandy, Sri Lanka.

[42] Piyadassi Thera (2008), *Dependent Origination (Paṭicca Samuppāda)*. BPS Online Edition.

[43] Rotha G., Dicke U. (2005), *Evolution of the brain and intelligence*. Trends in Cognitive Sciences, 9(5), pp. 250-257.

[44] Russell S. J. and Norvig P. (2009). *Artificial Intelligence: A Modern Approach* (3rd Edition). Prentice Hall Press, pp. 18-19.

[45] Rzevski G., Skobelev P. (2007), *Emergent Intelligence in Large Scale Multi-Agent Systems*. International Journal of Education and Information Technology, 1(2), pp. 64-71.

[46] Thagard P. (2010), "Cognitive Science", *The Stanford Encyclopedia of Philosophy* (Summer 2010 Edition), Edward N. Zalta (ed.), [online], Available: http://plato.stanford.edu/archives/sum2010/entries/cognitive-science/

[47] Thagard P., Holyoak K. J., Nelson, G., Gochfeld, D. (1990), *Analogue Retrieval by Constraint Satisfaction*. Artificial Intelligence, Elsevier Science Publishers B.V. (North-Holland), 46, pp. 259-310.

[48] Thoreau H. D. (1962), “VIII September 1851” in *The Journal of Henry D. Thoreau*, Vol. 2, NY: Dover publication, pp. 463.

[49] Tomai E., Lovett A., Forbus K. D., Usher J. (2005), *A structure mapping model for solving geometric analogy problems*, In Proceedings of the 27th Annual Conference of the Cognitive Science Society, Stresa, Italy, pp. 2190-2195.

[50] Turing A. (1990), “Computing Machinery and Intelligence” in *The Philosophy of Artificial Intelligence* by Boden M., Oxford University Press, pp. 40-66.

[51] Veale, T. (1998). ’Just in Time’ Analogical Mapping, An Iterative-Deepening Approach to Structure-Mapping, in *the proceedings of ECAI’98*, the Thirteenth European Conference on Artificial Intelligence, Brighton, UK.

Appendix A:

Java Agent DEvelopment (JADE) Framework

A.1 Introduction

This appendix provides sufficient and necessary information about JADE framework. JADE is a software framework to make easy the development of multi-agent applications in compliance with the FIPA specifications. JADE can then be considered a middle-ware that implements an efficient agent platform and supports the development of multi agent systems. JADE agent platform tries to keep high the performance of a distributed agent system implemented with the Java language. In particular, its communication architecture tries to offer flexible and efficient messaging, transparently choosing the best transport available and leveraging state-of-the-art distributed object technology embedded within Java runtime environment. JADE uses an agent model and Java implementation that allow good runtime efficiency, software reuse, agent mobility and the realization of different agent architectures. Following sub sections described the power associated with JADE to fulfil the identified needs in agent development.

A.2 FIPA Specifications

The Foundation for Intelligent Physical Agents (FIPA) is an international non-profit association of companies and organisations sharing the effort to produce specifications for generic agent technologies. FIPA does not just promote a technology for a single application domain but a set of general technologies for different application areas that developers can integrate to make complex systems with a high degree of interoperability.

FIPA has identified the roles of some key agents necessary for managing the platform, and describe the agent management content language and ontology. Three mandatory roles were identified into an agent platform. The Agent Management System (AMS) is the agent that exerts supervisory control over access to and use of the platform; it is responsible for maintaining a directory of resident agents and for handling their life cycle. The Agent Communication Channel (ACC) provides the

path for basic contact between agents inside and outside the platform. The ACC is the default communication method, which offers a reliable, orderly and accurate message routing service. The Directory Facilitator (DF) is the agent that provides yellow page services to the agent platform.

The specifications also define the Agent Communication Language (ACL), used by agents to exchange messages. FIPA ACL is a language describing message encoding and semantics, but it does not mandate specific mechanisms for message transportation. Since different agents might run on different platforms on different networks, messages are encoded in a textual form, assuming that agents are able to transmit 7-bit data. ACL syntax is close to the widely used communication language KQML. However, there are fundamental differences between KQML and ACL, the most evident being the existence of a formal semantics for FIPA ACL, which should eliminate any ambiguity and confusion from the usage of the language. FIPA supports common forms of inter-agent conversations through interaction protocols, which are communication patterns followed by two or more agents. Such protocols range from simple query and request protocols, to more complex ones, as the well-known contract net negotiation protocol and English auctions.

A.3 JADE Runtime System

A running agent platform must provide several services to the applications: when looking at the parts of the FIPA97 specification, is can be seen that these services fall into two main areas, that is, message passing support with FIPA ACL and agent management with life-cycle, white and yellow pages, etc.

A.3.1 Distributed Agent Platform

JADE complies with the FIPA97 specifications and includes all the system agents that manage the platform that is the ACC, the AMS, and the default DF. All agent communication is performed through message passing, where FIPA ACL is the language used to represent messages.

While appearing as a single entity to the outside world, a JADE agent platform is itself a distributed system, since it can be split over several hosts with one among them acting as a front end for inter-platform IIOP communication. A JADE system is made by one or more Agent Container, each one living in a separate Java Virtual Machine and communicating using Java RMI. IIOP is used to forward outgoing messages to foreign agent platforms. A special, Front End container is also an IIOP server, listening at the official agent platform ACC address for incoming messages from other platforms. Figure A.1 shows the architecture of a JADE Agent Platform.

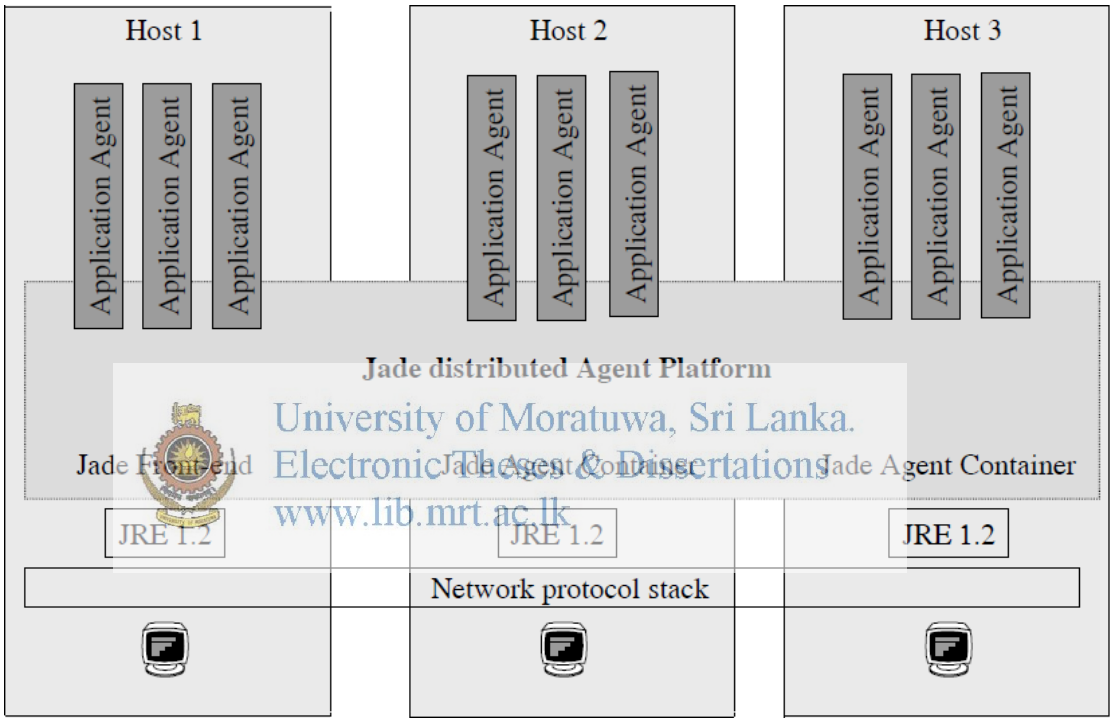

Figure A.1: Software architecture of a JADE Agent Platform

A.3.2 Message Delivery Subsystem

FIPA agent communication model is peer-to-peer though multi-message context is provided by interaction protocols and conversation identifiers. On the other hand, JADE uses transport technologies such as RMI, CORBA and event dispatching which are typically associated with reactive systems. Clearly, there is some gap to bridge to map the explicitly addressed FIPA message-passing model into the request/response communication model of distributed objects. This is why in JADE ordinary agents are not distributed objects, but agent containers are.

A software agent, in compliance to FIPA agent model, has a globally-unique identifier (GUID), that can be used by every other agent to address it with ACL messages; likewise, an agent will put its GUID into the :sender slot of ACL messages it sends around. So, JADE must figure out receiver location by simply looking at: receiver message slot. Since a FIPA97 GUID resembles an email address, it has the form: <agent name> @ <platform address>, it is fairly easy to recover the agent name and the platform address from it. When an ACL message is sent to a software agent, three options are given:

- Receiver on the same container of the same platform: Java events are used, the ACLMessage is simply cloned.
- Receiver on a different container of the same platform: Java RMI is used, the message is serialised at sender side, a remote method is called and the message is un-serialised at receiver side.
- Receiver on a different platform: IIOP is used, the ACLMessage is converted into a String and marshalled at sender side, a remote CORBA call is done and an un-marshalling followed by ACL parsing occurs at receiver side.

A.3.3 Address Management and Caching

JADE tries to select the most convenient of the three transport mechanisms above according to agents location. Basically, each container has a table of its local agents, called the Local-Agent Descriptor Table (LADT), whereas the front-end, besides its own LADT, also maintains a Global-Agent Descriptor Table (GADT), mapping every agent into the RMI object reference of its container. Moreover, JADE uses an address caching technique to avoid querying the front-end continuously for address information.

Besides being efficient, this is also meant to support agent mobility, where agent addresses can change over time (e.g. from local to RMI); transparent caching means that messaging subsystem will not be affected when agent mobility will be introduced into JADE. Moreover, if new remote protocols will be needed in JADE (e.g. a wireless protocol for nomadic applications), they will be seamlessly integrated inside the messaging and address caching mechanisms.

A.3.4 User-Defined Ontologies and Content Languages

According to the FIPA standard, achieving agent level interoperability requires that different agents share much more than a simple on-the-wire protocol. While FIPA mandates a single agent communication language, the FIPA ACL, it explicitly allows application dependent content languages and ontologies. The FIPA specifications themselves now contain a Content Language Library, whereas various mandatory ontologies are defined and used within the different parts of the FIPA standard.

The last version of JADE lets application programmers create their own content languages and their ontologies. Every JADE agent keeps a capability table where the known languages and ontologies are listed; user defined codecs must be able to translate back and forth between the String format (according to the content language syntax) and a frame based representation.

If a user-defined ontology is defined, the application can register a suitable Java class to play an ontological role and JADE is able to convert to and from frames and user defined Java objects. Acting this way, application programmers can represent their domain specific concepts as familiar Java classes, while still being able to process them at the agent level (put them within ACL messages, reasoning about them, etc.).

A.3.5 Tools for Platform Management and Monitoring

Beyond a runtime library, JADE offers some tools to manage the running agent platform and to monitor and debug agent societies; all these tools are implemented as FIPA agents themselves, and they require no special support to perform their tasks, but just rely on JADE AMS.

The general management console for a JADE agent platform is called RMA (Remote Monitoring Agent). The RMA acquires the information about the platform and executes the GUI commands to modify the status of the platform (creating agents, shutting down containers, etc.) through the AMS. The Directory Facilitator agent also has a GUI, with which it can be administered, configuring its advertised agents and services.

JADE users can debug their agents with the Dummy Agent and the Sniffer Agent. The Dummy Agent is a simple tool for inspecting message exchanges among agents, facilitating validation of agent message exchange patterns and interactive testing of an agent. The Sniffer Agent allows tracking messages exchanged in a JADE agent platform: every message directed to or coming from a chosen agent or group is tracked and displayed in the sniffer window, using a notation similar to UML Sequence Diagrams.

A.4 JADE Agent Development Model

FIPA specifications state nothing about agent internals, but when JADE was designed and built they had to be addressed. A major design issue is the execution model for an agent platform, both affecting performance and imposing specific programming styles on agent developers. As will be shown in the following, JADE solution stems from the balancing of forces from ordinary software engineering guidelines and theoretical agent properties.

A.4.1 From Agent Theory to Class Design

A distinguishing property of a software agent is its autonomy; an agent is not limited to react to external stimuli, but it's also able to start new communicative acts of its own. A software agent, besides being autonomous, is said to be social, because it can interact with other agents in order to pursue its goals or can even develop an overall strategy together with its peers.

FIPA standard bases its Agent Communication Language on speech-act theory and uses a mentalistic model to build a formal semantic for the performative agent exchange. This approach is quite different from the one followed by distributed objects and rooted in Design by Contract; a fundamental difference is that invocations can either succeed or fail but a request speech act can be refused if the receiver is unwilling to perform the requested action.

Trying to map the aforementioned agent properties into design decisions, the following list was produced:

- Agents are autonomous, and then they are active objects.
- Agents are social, and then intra-agent concurrency is needed.
- Messages are speech acts, and then asynchronous messaging must be used.
- Agents can say “no”, and then peer-to-peer communication model is needed.

The autonomy property requires each agent to be an active object with at least a Java thread, to proactively start new conversations, make plans and pursue goals. The need for sociality has the outcome of allowing an agent to engage in many conversations simultaneously, dealing with a significant amount of concurrency.

The third requirement suggests asynchronous message passing as a way to exchange information between two independent agents that also has the benefit of producing more reusable interactions. Similarly, the last requirement stresses that in a Multi Agent System the sender and the receiver are equals (as opposed to client/server systems where the receiver is supposed to obey the sender). An autonomous agent should also be allowed to ignore a received message as long as he wishes; this advocates using a pull consumer messaging model, where incoming messages are buffered until their receiver decides to read them.

A.4.2 JADE Agent Concurrency Model

The autonomy requirement forces each agent to have at least a thread, and the sociality requirement pushes towards many threads per agent. Unfortunately, current operating systems limit the maximum number of threads that can be run effectively on a system. JADE execution model tries to limit the number of threads and has its roots in actor languages.

The Behaviour abstraction models agent tasks: a collection of behaviours are scheduled and executed to carry on agent duties (see Figure A.2). Behaviours represent logical threads of a software agent implementation. According to Active Object design pattern, every JADE agent runs in its own Java thread, satisfying autonomy property; instead, to limit the threads required to run an agent platform, all agent behaviours are executed cooperatively within a single Java thread. So, JADE uses a thread-per-agent execution model with cooperative intra-agent scheduling.

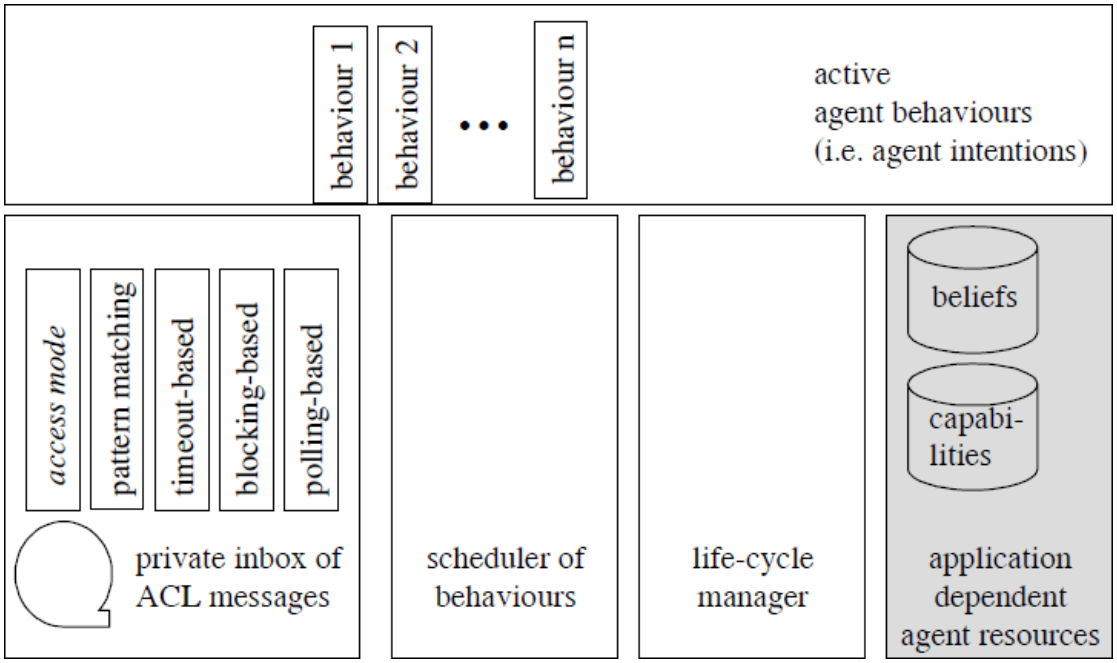

Figure A.2: JADE agent architecture.

JADE agents schedule their behaviour with a "cooperative scheduling on top of the stack", in which all behaviours are run from a single stack frame (on top of the stack) and a behaviour runs until it returns from its main function and cannot be pre-empted by other behaviours (cooperative scheduling).

JADE model is an effort to provide fine-grained parallelism on coarser grained hardware. A likewise, stack based execution model is followed by Illinois Concert runtime system for parallel object oriented languages. Concert executes concurrent method calls optimistically on the stack, reverting to real thread spawning only when the method is about to block, saving the context for the current call only when forced to.

Choosing not to save behaviour execution context means that agent behaviours start from the beginning every time they are scheduled for execution. So, behaviour state that must be retained across multiple executions must be stored into behaviour instance variables. A general rule for transforming an ordinary Java method into a JADE behaviour is:

- Turn the method body into an object whose class inherits from Behaviour.
- Turn method local variables into behaviour instance variables.
- Add the behaviour object to agent behaviour list during agent start-up.

The above guidelines apply the reification technique to agent methods, according to Command design pattern; an agent behaviour object reifies both a method and a separate thread executing it. A new class must be written and instantiated for every agent behaviour and this can lead to programs harder to understand and maintain. JADE application programmers can compensate for this shortcoming using Java Anonymous Inner Classes; this language feature makes the code necessary for defining an agent behaviour only slightly higher than for writing a single Java method.

JADE thread-per-agent model can deal alone with the most common situations involving only agents: this is because every JADE agent owns a single message queue from which ACL messages are retrieved. Having multiple threads but a single mailbox would bring no benefit in message dispatching. On the other hand, when writing agent wrappers for non-agent software, there can be many interesting events from the environment beyond ACL message arrivals. Therefore, application developers are free to choose whatever concurrency model they feel is needed for their particular wrapper agent; ordinary Java threading is still possible from within an agent behaviour.

A.4.3 Using Behaviours to Build Complex Agents

The developer implementing an agent must extend Agent class and implement agent specific tasks by writing one or more Behaviour subclasses. User defined agents inherit from their super class the capability of registering and deregistering with their platform and a basic set of methods (e.g. send and receive ACL messages, use standard interaction protocols, register with several domains). Moreover, user agents inherit from their Agent super class two methods: addBehaviour(Behaviour) and removeBehaviour(Behaviour), to manage the behaviour list of the agent.

JADE contains readymade behaviours for the most common tasks in agent programming, such as sending and receiving messages and structuring complex tasks as aggregations of simpler ones. For example, JADE offers a so-called JessBehaviour that allows full integration with JESS, a scripting environment for rule programming offering an engine using the Rete algorithm to process rules.

Behaviour is an abstract class that provides the skeleton of the elementary task to be performed. It exposes three methods: the action() method, representing the "true" task to be accomplished by the specific behaviour classes; the done() method, used by the agent scheduler, that must return true when the behaviour has finished and false when the behaviour has not and the action() method must be executed again; the reset() method, used to restart a behaviour from the beginning.

JADE follows a compositional approach to allow application developers to build their own behaviours out of the simpler ones directly provided by the framework. Applying the Composite design pattern, ComplexBehaviour class is itself a Behaviour, with some sub-behaviours or children, defining two methods addSubBehaviour(Behaviour) and removeSubBehaviour(Behaviour). This permits agent writers to implement a structured tree with behaviours of different kinds. Besides ComplexBehaviour, JADE framework defines some other subclasses of Behaviour: SimpleBehaviour can be used to implement atomic steps of the agent work. A behaviour implemented by a subclass of SimpleBehaviour is executed by JADE scheduler in a single time frame. Two more subclasses to send and receive messages are SenderBehaviour and ReceiverBehaviour. They can be instantiated passing appropriate parameters to their constructors. SenderBehaviour allows sending a message, while ReceiverBehaviour allows receiving a message, which can be matched against a pattern; the behaviour blocks itself (without stopping all other agent activities) if no suitable messages are present.

JADE recursive aggregation of behaviour objects resembles the technique used for graphical user interfaces, where every interface widget can be a leaf of a tree whose intermediate nodes are special container widgets, with rendering and children management features. An important distinction, however, exists: JADE behaviours reify execution tasks, so task scheduling and suspension are to be considered, too.

Thinking in terms of software patterns, if Composite is the main structural pattern used for JADE behaviours, on the behavioural side we have Chain of Responsibility: agent scheduling directly affects only top-level nodes of the behaviour tree, but every composite behaviour is responsible for its children scheduling within its time frame.

Appendix B:

Geometric ontology

B.1 Introduction

Following XML data provides the information of trivial geometric ontology used for implementation. This can be easily extended for complex analogy problems with improvements that were discussed.

```
<?xml version="1.0"?>
<!DOCTYPE Ontology [
<!ENTITY xsd "http://www.w3.org/2001/XMLSchema#" >
<!ENTITY xml "http://www.w3.org/XML/1998/namespace" >
<!ENTITY rdfs "http://www.w3.org/2000/01/rdf-schema#" >
<!ENTITY rdf "http://www.w3.org/1999/02/22-rdf-syntax-ns#" >
]>
<Ontology xmlns="http://www.w3.org/2002/07/owl#"
    xml:base="http://www.semanticweb.org/ontologies/2011/3/GeometricOntology.owl"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema#"
    xmlns:xml="http://www.w3.org/XML/1998/namespace"
    xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#"
    xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
    ontologyIRI="http://www.semanticweb.org/ontologies/2011/3/GeometricOntology.owl">
<Prefix name="rdf" IRI="http://www.w3.org/1999/02/22-rdf-syntax-ns#"/>
<Prefix name="" IRI="http://www.w3.org/2002/07/owl#"/>
<Prefix name="xsd" IRI="http://www.w3.org/2001/XMLSchema#"/>
<Prefix name="rdfs" IRI="http://www.w3.org/2000/01/rdf-schema#"/>
<Prefix name="owl" IRI="http://www.w3.org/2002/07/owl#"/>
<Declaration>
<Class IRI="#Big"/>
</Declaration>
<Declaration>
<Class IRI="#Circle"/>
</Declaration>
<Declaration>
<Class IRI="#Direction"/>
</Declaration>
<Declaration>
<Class IRI="#Down"/>
```

```
</Declaration>
<Declaration>
<Class IRI="#Ellipse"/>
</Declaration>
<Declaration>
<Class IRI="#Equilateral"/>
</Declaration>
<Declaration>
<Class IRI="#Isosceles"/>
</Declaration>
<Declaration>
<Class IRI="#Large"/>
</Declaration>
<Declaration>
<Class IRI="#Left"/>
</Declaration>
<Declaration>
<Class IRI="#Medium"/>
</Declaration>
<Declaration>
<Class IRI="#Parallelogram"/>
</Declaration>
<Declaration>
<Class IRI="#Rectangle"/>
</Declaration>
<Declaration>
<Class IRI="#Right"/>
</Declaration>
<Declaration>
<Class IRI="#Scalene"/>
</Declaration>
<Declaration>
<Class IRI="#Shape"/>
</Declaration>
<Declaration>
<Class IRI="#Size"/>
</Declaration>
<Declaration>
<Class IRI="#Small"/>
</Declaration>
```

<Declaration>
<Class IRI="#Square"/>
</Declaration>
<Declaration>
<Class IRI="#Triangle"/>
</Declaration>
<Declaration>
<Class IRI="#Up"/>
</Declaration>
<EquivalentClasses>
<Class IRI="#Big"/>
<Class IRI="#Large"/>
</EquivalentClasses>
<SubClassOf>
<Class IRI="#Big"/>
<Class IRI="#Size"/>
</SubClassOf>
<SubClassOf>
<Class IRI="#Circle"/>
<Class IRI="#Ellipse"/>
</SubClassOf>
<SubClassOf>
<Class IRI="#Down"/>
<Class IRI="#Direction"/>
</SubClassOf>
<SubClassOf>
<Class IRI="#Ellipse"/>
<Class IRI="#Shape"/>
</SubClassOf>
<SubClassOf>
<Class IRI="#Equilateral"/>
<Class IRI="#Triangle"/>
</SubClassOf>
<SubClassOf>
<Class IRI="#Isosceles"/>
<Class IRI="#Triangle"/>
</SubClassOf>
<SubClassOf>
<Class IRI="#Large"/>
<Class IRI="#Size"/>

</SubClassOf>
<SubClassOf>
<Class IRI="#Left"/>
<Class IRI="#Direction"/>
</SubClassOf>
<SubClassOf>
<Class IRI="#Medium"/>
<Class IRI="#Size"/>
</SubClassOf>
<SubClassOf>
<Class IRI="#Parallelogram"/>
<Class IRI="#Shape"/>
</SubClassOf>
<SubClassOf>
<Class IRI="#Rectangle"/>
<Class IRI="#Parallelogram"/>
</SubClassOf>
<SubClassOf>
<Class IRI="#Right"/>
<Class IRI="#Direction"/>
</SubClassOf>
<SubClassOf>
<Class IRI="#Scalene"/>
<Class IRI="#Triangle"/>
</SubClassOf>
<SubClassOf>
<Class IRI="#Small"/>
<Class IRI="#Size"/>
</SubClassOf>
<SubClassOf>
<Class IRI="#Square"/>
<Class IRI="#Rectangle"/>
</SubClassOf>
<SubClassOf>
<Class IRI="#Triangle"/>
<Class IRI="#Shape"/>
</SubClassOf>
<SubClassOf>
<Class IRI="#Up"/>
<Class IRI="#Direction"/>

</SubClassOf>

<DisjointClasses>

<Class IRI="#Down"/>

<Class IRI="#Left"/>

<Class IRI="#Right"/>

<Class IRI="#Up"/>

</DisjointClasses>

<DisjointClasses>

<Class IRI="#Equilateral"/>

<Class IRI="#Isosceles"/>

<Class IRI="#Scalene"/>

</DisjointClasses>

<DisjointClasses>

<Class IRI="#Large"/>

<Class IRI="#Medium"/>

<Class IRI="#Small"/>

</DisjointClasses>

</Ontology>

Appendix C:

Analogy problems for verification

C.1 Introduction

Following figures (Figure C.1 to Figure C.10) are the geometric analogies that were used to evaluate the model. Those have been created mainly from the intuition got from Thomas G. Evans testing problems and deliberately made them non ambiguous but merely to capture the analogical reasoning with low order cognition. Furthermore in all the Figures number of solutions were reduced to three to improve the computation speed.

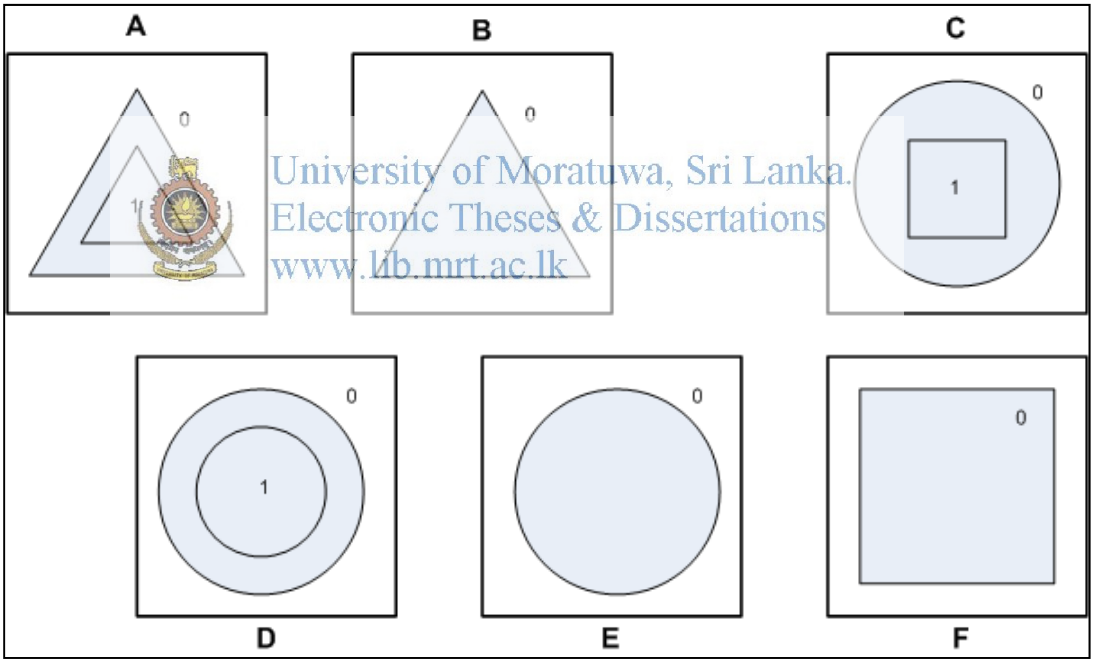

Figure C.2: Analogy problem 1

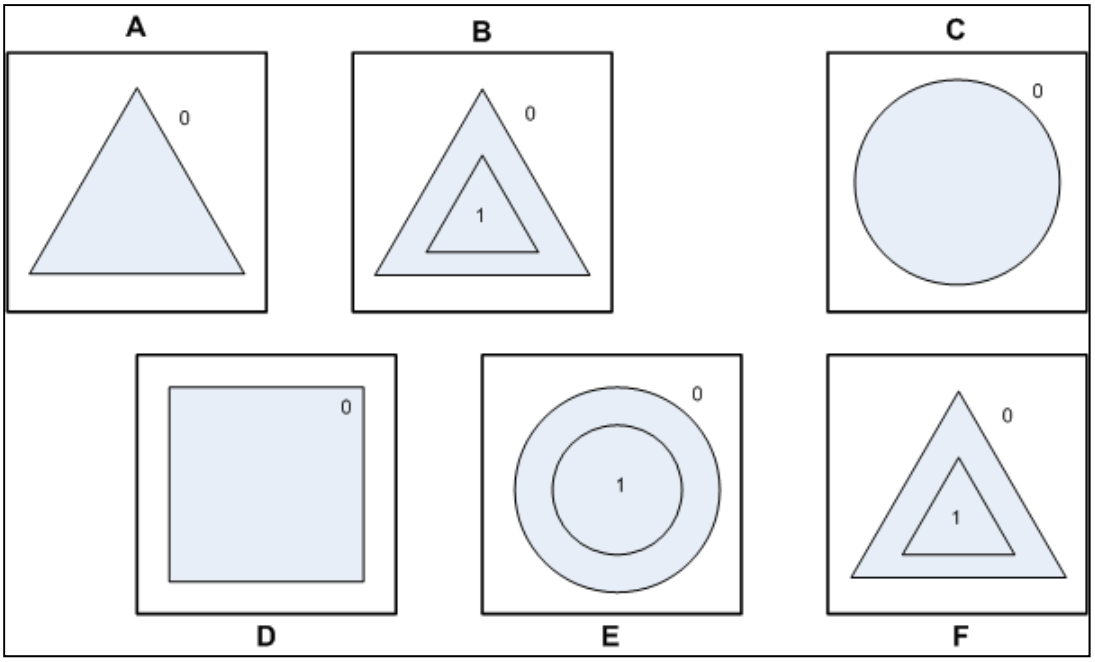

Figure C.3: Analogy problem 2

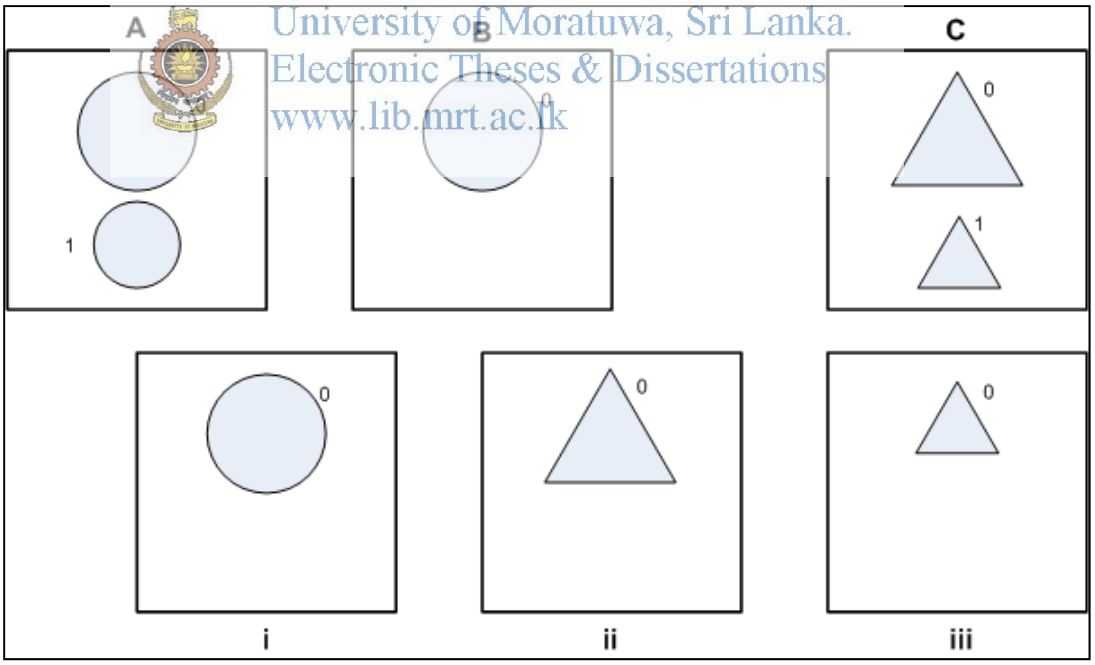

Figure C.4: Analogy problem 3

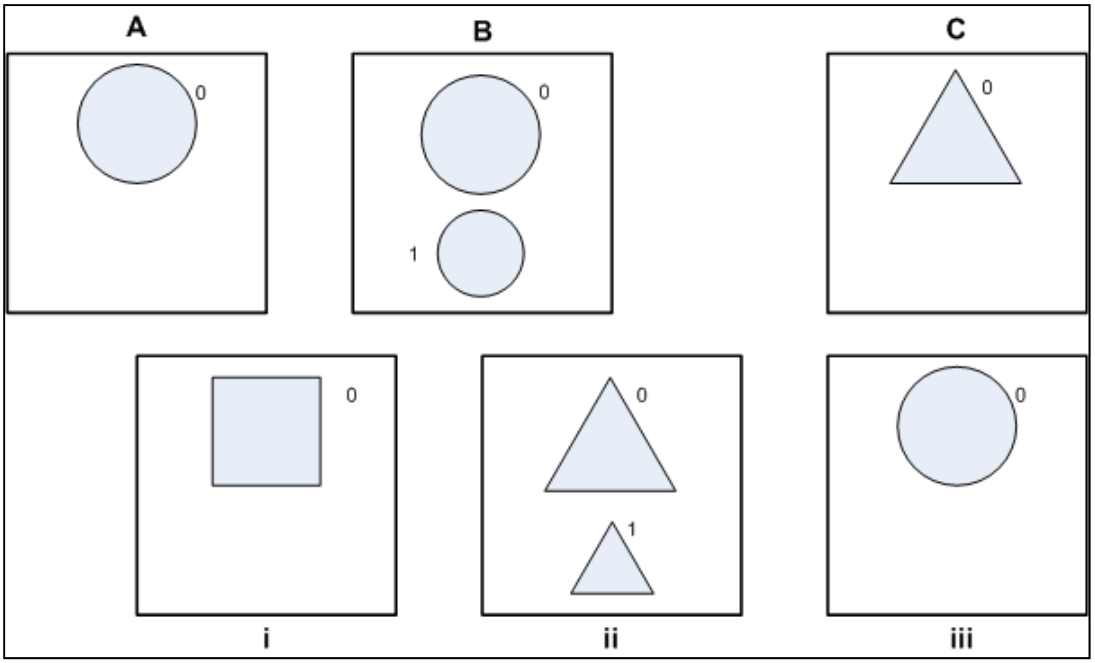

Figure C.5: Analogy problem 4

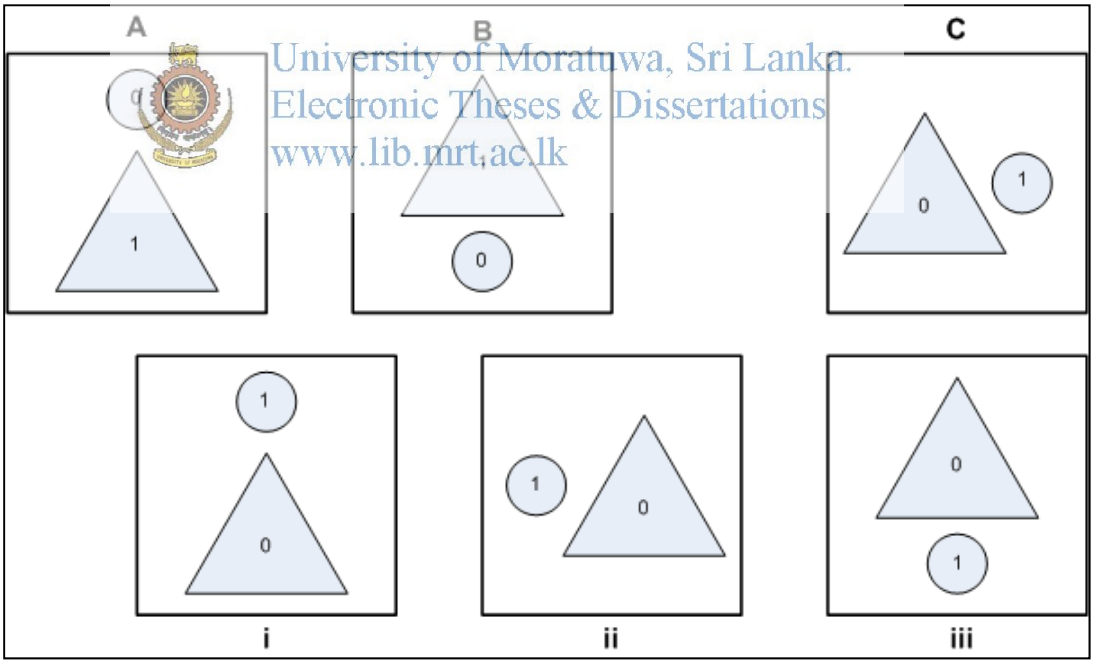

Figure C.6: Analogy problem 5

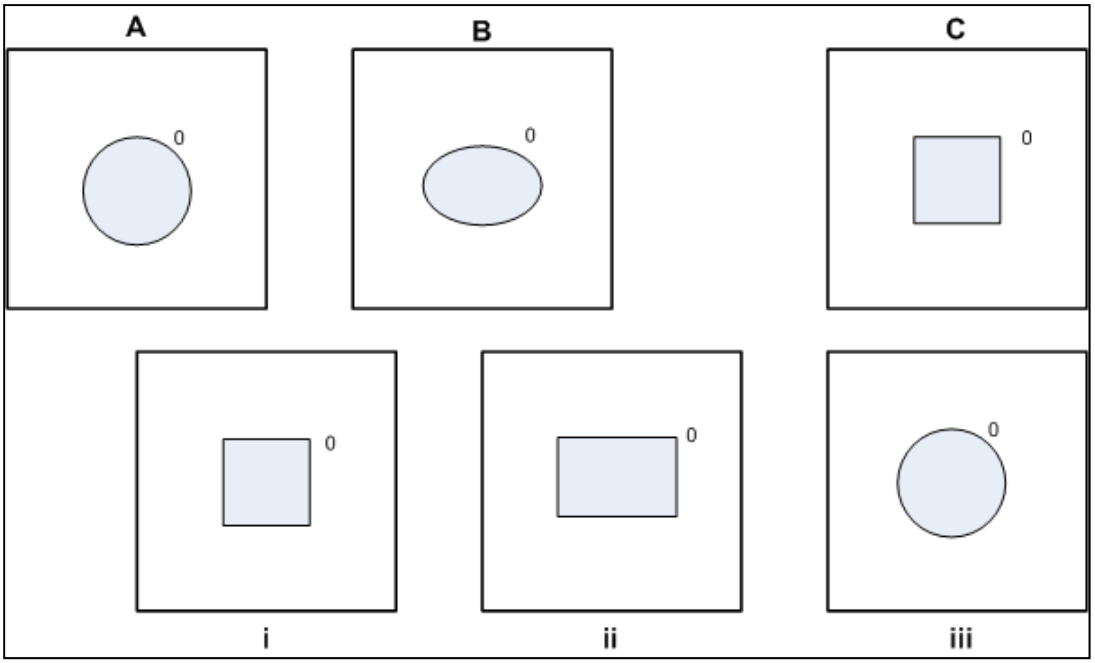

Figure C.7: Analogy problem 6

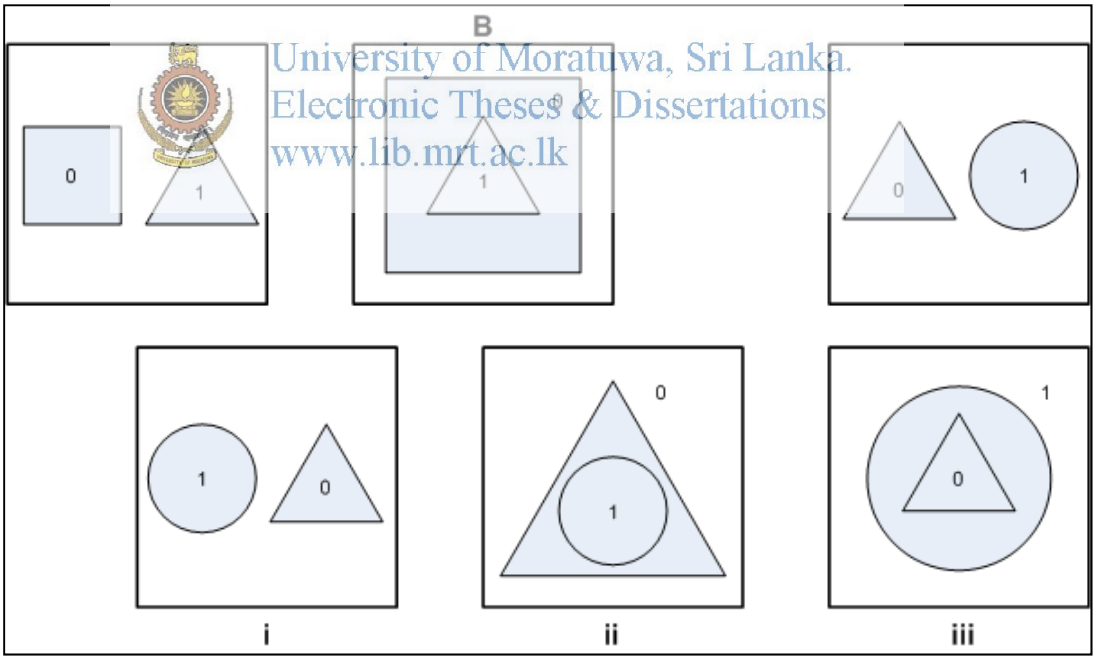

Figure C.8: Analogy problem 7

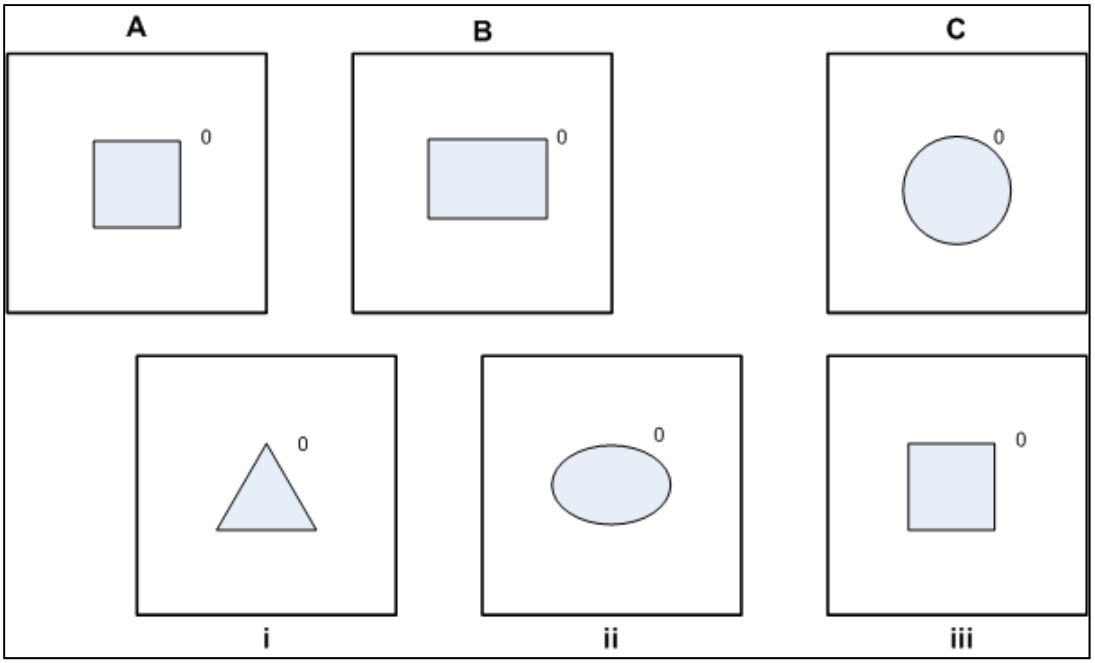

Figure C.9: Analogy problem 8

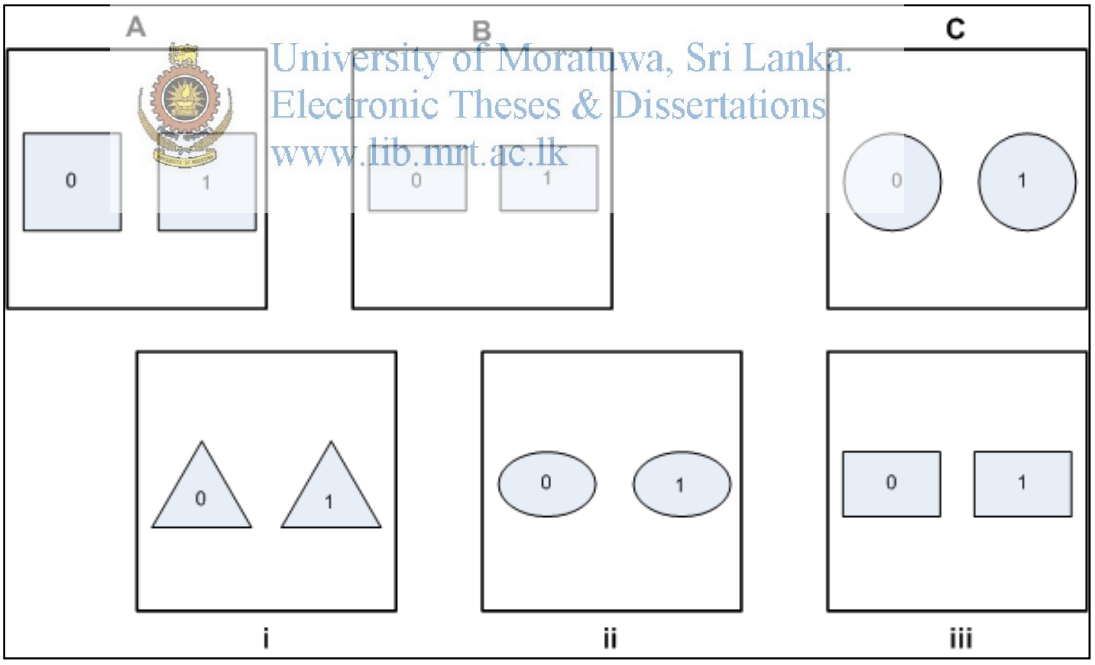

Figure C.10: Analogy problem 9

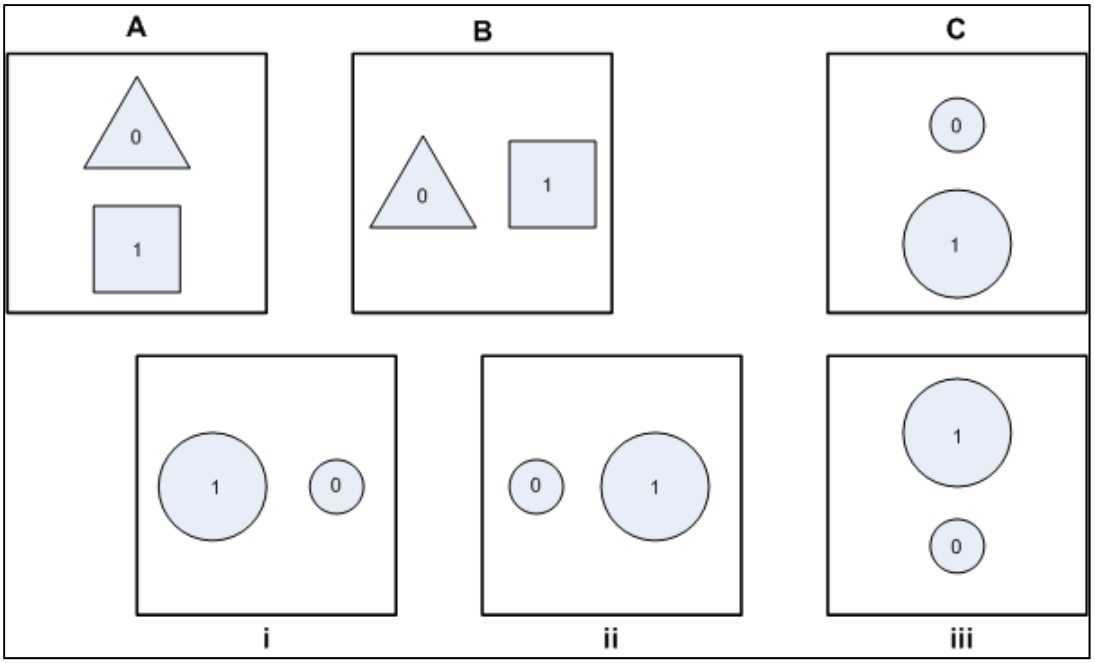

Figure C.11: Analogy problem 10