JOURNAL OF OBJECT TECHNOLOGY

Online at http://www.jot.fm. Published by ETH Zurich, Chair of Software Engineering ©**JOT, 2007**

Vol. 6, No. 4, May-June 2007

Pattern-Oriented Design for Multi-Agent System: A Conceptual Model

Radziah Mohamad and **Safaai Deris**, Faculty of Computer Science and Information Systems, Universiti Teknologi Malaysia, Malaysia
Hany H. Ammar, Computer Science and Electrical Engineering Department, West Virginia University, USA

Abstract

Design patterns represent solutions to specific problem of developing one application that have evolved over time. They leverage the reuse level to the design phase by providing a common vocabulary of design, means of understanding designs and proven building blocks from which more complex applications are built. Much of the research work on design patterns in agent world has primarily focused on discovering and documenting patterns. To reap the benefits of deploying these proven design solutions, patterns should be used as a first-class modelling blocks in designing the agent application. This paper proposes a conceptual model based on patterns and agent abstraction.

1 INTRODUCTION

The explosive growth of application areas such as electronic commerce, knowledge management, peer-to-peer and mobile computing demand software that is robust, can operate within a wide range of environments, and can evolve over time to cope with changing requirements. Autonomous agents and multi-agent systems are being used to cope with ever increasing complexity in such system requirements. As the complexity of software system increases, developers look for approaches to facilitate the development of software applications. Patterns are reusable good-quality design practices that have proven useful in the design of software applications [Gamma et al.95][Buschmann et al.96]. Patterns can help in leveraging reuse to the design level because they provide a common vocabulary of designs and they are proven design units from which more complex applications can be built. Against this background, much work has focused on documenting design patterns for agent world [Deugo et al.01][Aridor&Lange98]. Other work is concerned with applying these reusable designs in constructing agent applications [Lind03][Cossentino et al.03][Kolp et al.01]. To reap the benefits of deploying these proven design solutions, we believe that patterns should be used as a first-class modelling blocks in designing the agent application.

Pattern-Oriented Analysis and Design methodology (POAD) proposed by [Yacoub&Ammar04] offers a systematic process to design object-oriented system using patterns as block of designs. However, the kind of decomposition that POAD offers is at odds with the kind of decomposition that agent oriented design encourages since agents are more coarse-grained than objects. Furthermore, it is hard to capture many aspects in agent system, for example, the agent coordination. Inspired by a systematic process offered by POAD, this paper proposes a conceptual model of pattern-oriented design for multi-agent based system. Conceptual model could provide a solid foundation for the design and implementation phases.

This paper is organised as follows. Section 2 will discuss the motivation of defining the conceptual model. Section 3 will discuss our proposed pattern-oriented design metamodel. Section 4 will show the mapping of the proposed concept to the existing agent design concept. In section 5, we present a translation of the concept to the Object-Z constructs. Section 6 describes how existing design patterns can be described according to the proposed metamodel. Section 7 discusses related work and Section 8 concludes.

2 MOTIVATION

The multi-agent paradigm has shown an increasing potential for modelling and building a wide range of complex applications from various domains. Multi-agent systems used for air traffic control, weather monitoring and air combat simulations are some typical examples. Designing and developing multi-agent system is not easy [Sycara98]. One must manage at least three different levels of abstraction at the same time:

- The level of the individual agents.
- The level of the interactions between the agents
- The level of their organization

With increases in the complexity of software systems, reusing design ideas and models in terms of design patterns is a promising alternative than designing from scratch. Looking at the literature on design patterns for the agent world, we realize that most of the effort is expended in discovering and documenting design patterns. The documentation of design patterns, as it stands, describes details about a pattern, its usage, structure, behaviour of participants, forces and consequences, and guidelines for implementations. Little work is concerned with systematically applying these reusable designs in developing new multi-agent applications. A systematic approach to design with patterns goes beyond just applying a certain pattern. The approach include a development process which defines a pattern composition approach, analysis and design steps, design models and tools to automate the development steps [Yacoub&Ammar04].

This paper discuss the relationship between concepts used in Pattern –Oriented Design for agent world modeling (POD) and the agent metamodels. The POD semantics is defined along two dimensions: abstract syntax and well-formedness rules. According to Kim and Carrington [Kim&Carrington04], a pattern properties must be expressed generically so that the pattern can be realized in various ways depending on the application. The abstract syntax dimension is used to achieve this goal. In this dimension, a pattern-oriented design metamodel is adopted. The abstract syntax of the metamodel is represented using a set of UML class diagrams defining the metamodel construct and their relationships. The well-formedness rules dimension specifies constraints on each of the elements described in the metamodel class diagrams. Object-Z is used to specify the rules. Using these concepts, we define each design pattern for agent world generically in terms of a pattern model.

The overall process is as illustrated in the framework in the Figure 1. The framework is based on Kim and Carrington's metamodeling framework [Kim&Carrington04]. In this framework, the pattern metamodel and pattern models are developed as a single Object-Z specification. The syntactic structures of POD design pattern constructs and the rules for developing a well-formed pattern diagrams are precisely described using the Object-Z notation, extending the approach of [Kim&Carrington04]. Based on this formal description, POD constructs are translated to Object-Z constructs.

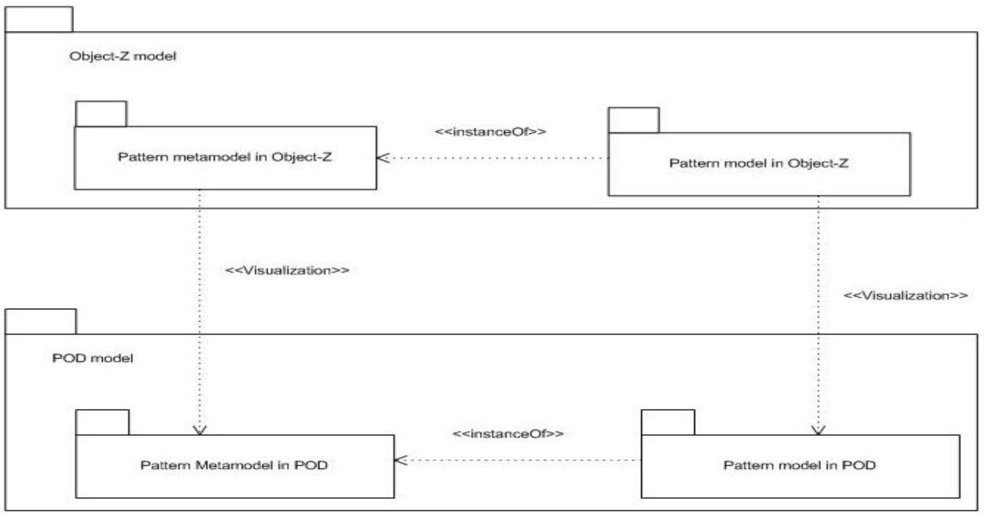


Figure 1 POD metamodel framework

3 PATTERN-ORIENTED DESIGN (POD) METAMODEL

In POD, design pattern is used as a first-class modeling concepts to design agent application. Figure 2 illustrates the proposed POD metamodel.

A metaclass *Pattern* from which different types of patterns are derived is defined. A pattern has a name describing its name and type. *PatternKind* is a data type that defines the design pattern names. Its values denote the type of an instantiated patterns. Examples of the *PatternKind* values are BDI architecture [Sauvage04], Mediator [Deugo et al.01] and Observer [Gamma et al.95]. By inheriting the *Pattern* class, a class of pattern is defined. An agent pattern can be classified into agent structure aspect, agent interaction aspect, agent organization aspect and agent architecture aspect. The following subsections will discuss in detail each of the aspects.

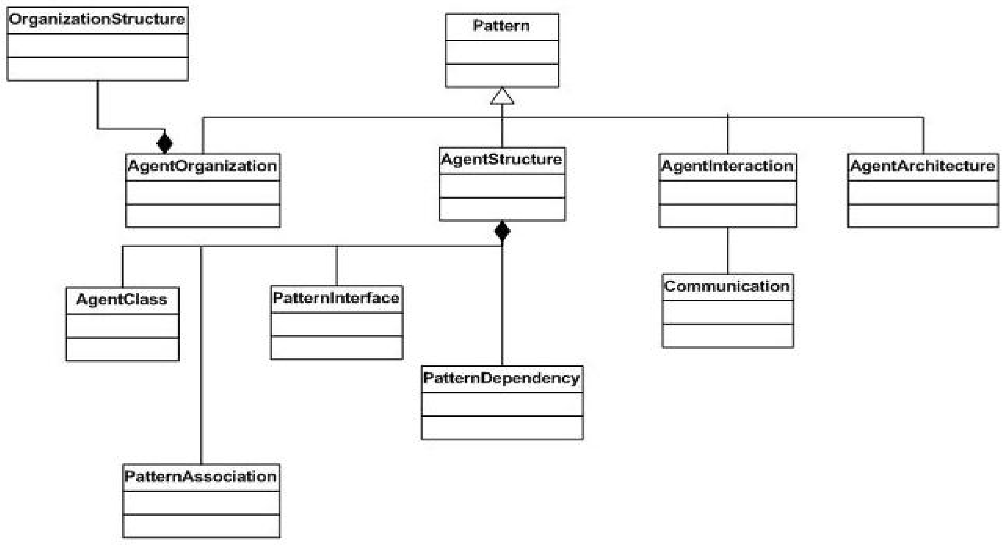


Figure 2 POD metamodel

Agent Structure Class

Agent structure class defines a solution on how the agent is represented an its most common elements. In our approach, we represent the solution as a set of collaborating classes. The pattern is modeled as packages to group the model elements. Inspired by the Pattern-Oriented Analysis and Design methodology defined by Yacoub and Ammar [Yacoub&Ammar04], we treat the agent pattern in this class as a component. Hence, the agent structure is composed of *PatternDependency*, *PatternInterface*, *PatternAssociation* and *AgentClass* metaclasses. A pattern dependency indicates a semantic relationship between two patterns – a situation in which a change to the source pattern may require a change to the target pattern. This type of relationship is used in POD at the high-level design phase where the system is modeled as a composition of patterns and pattern dependency relationship between those patterns. A pattern interfaces define how the pattern interfaces to other patterns. The interface helps in gluing patterns together at the high level of abstraction and hiding internal details and design decisions made within the pattern. A pattern association defines a semantic relationship between the interfaces of two patterns. The instances of an association are a set of tuplets relating instances of the *Pattern Interface*.

Agent Architecture Class

This class describes the internal architecture of an agent. This class shows different ways to arrange and decompose the structure of agent into several modules and their interactions.

Agent Organization Class

This class models a solution of specific organization infrastructure constrained by rules that enforce agents to some collective or individual behaviour. It explains the concepts associated with organisation and their numerous uses. The organization compose of organisation structure which is an abstract structure grouping role descriptions.

Agent Interaction Class

This class models a proven solution on how agents are supposed to interact using communications. The communications are specified by attributes like interaction protocols and content languages.

4 RELATING THE POD METAMODEL TO THE AGENT METAMODEL

Figure 3 illustrates the mapping between the proposed POD concept to the simplified version of the unified agent concept proposed by [Bernon et al.04]. The unified agent concept is resulted from the integration of three agent design methodologies: ADELFE, PASSI and Gaia. It represents the common concepts shared by the three methodologies. In mapping the POD concept to the unified agent concept, we represent it in two level of abstractions: pattern level and agent level. The pattern level deals with the agent patterns classes available in the literature: agent structure, agent interaction, agent organization and agent architecture. The agent level contains the elements of the agent-based solution. In this level, the agent is enriched with all the generic properties an agent may have. The agent is composed of role, service and ontology. The agent can belong to an organization. By mapping these two concepts, we would be able to define a methodology to systematically design multi-agent application with patterns. The motivation is the methodology could be immersed to the existing agent design methodologies to make them more mature.

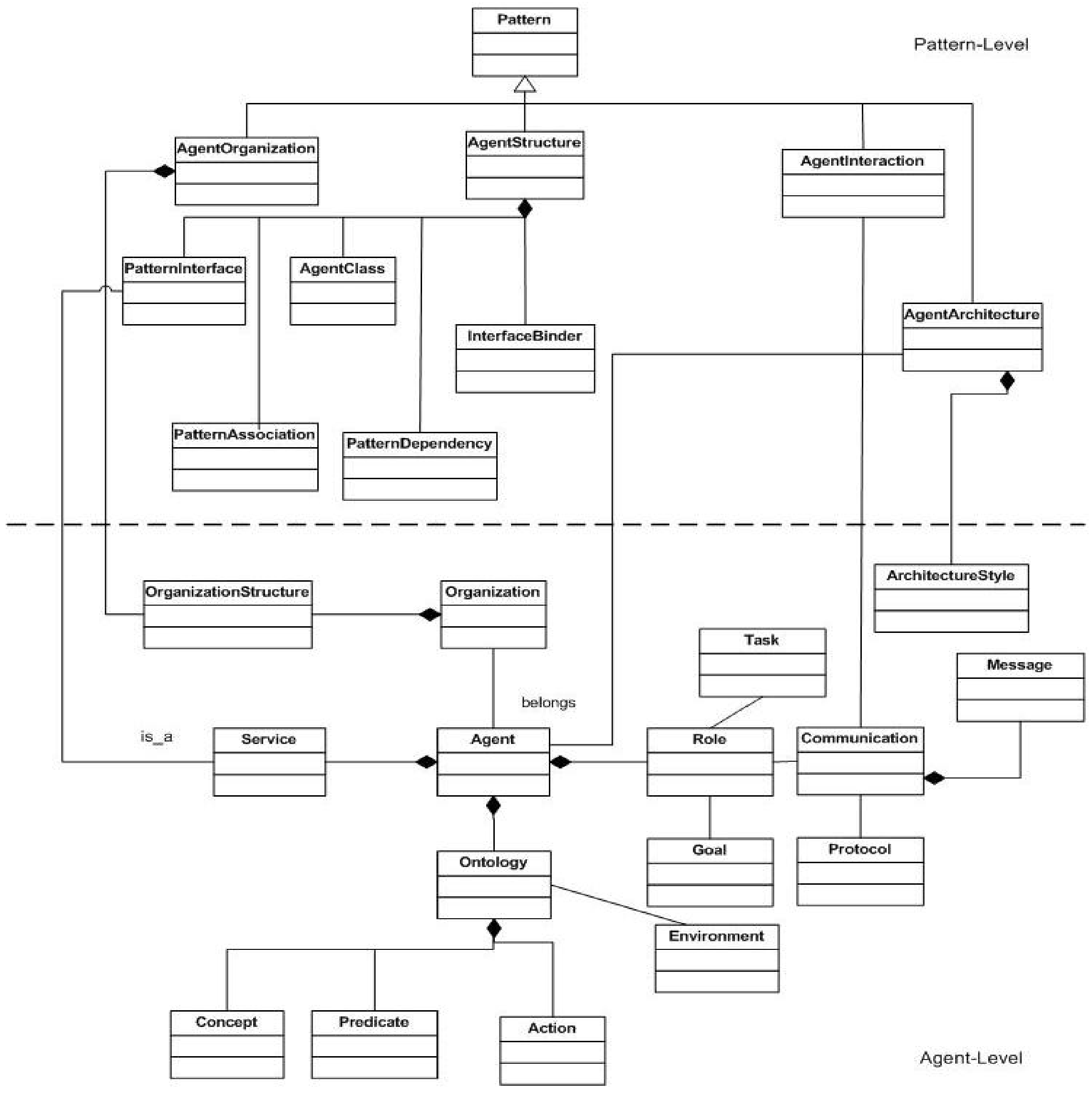


Figure 3 Mapping between the POD concept and agent concept

5 A FORMAL DEFINITION OF THE CONCEPT

In this section, we present a formal definition of POD concept using Object-Z. Object-Z is an object-oriented extension to Z designed specifically to facilitate specification in an object-oriented style and it is used as a meta-language to specify other languages such as UML [Kim&Carrington99]. The model constructs discussed in this section are useful for tool developers and methodologists who are more interested in the formal aspects of the models and their semantics.

Pattern

Prior to formalizing pattern concepts, a given set, *PatternName* and *Name* are defined, from which the names of all patterns and pattern type are drawn.

```
[PatternName, Name]
```

Using this type, we formalize the metaclass *Pattern* in Figure 2 as an Object-Z class *Pattern*. The attribute *patterninstance* is formalized as a variable in the Object-Z class.

```
┌─────────────── PatternKind ─────────────────
│
│   ┌──────────────────────
│   │ name: Name
│   └──────────────────────
└─────────────────────────────────────────────

┌─────────── Pattern ───────────
│
│   ┌──────────────────────
│   │ patterninstance: PatternKind
│   └──────────────────────
└───────────────────────────────
```

Agent Class

In POD, agent class is defined as a descriptor of a set of roles with common properties in terms of tasks and structure. The agent class has a name, roles, attributes and tasks. Attributes have names and types. Following the approach of Kim and Carrington [Kim&Carrington99], two given sets, *AgentName* and *Name* are defined for the agent structure. From these given sets, the name of all agent classes and the names of all attributes, tasks and roles can be drawn respectively. A schema named *AgentClassDecl* denotes the components of an agent class: a finite set of attributes, a finite set of roles and a finite set of tasks played by the roles. A partial function attrstate is declared to map attributes to their types. Agent class names should be unique in the enclosing name space. Thus, the set of agent classes defined as a partial function from *AgentClassName* to *AgentClassDecl*.

```
[AgentName, Name]
```

```
┌─────────────── AgentClassDecl ───────────────
│ attributes : F Name
│ roles: F ↓ AgentRole
│ task: F Name
│ attrstate: Name ⇸ Type
├──────────────────────────────
│ attributes = dom attrstate
└──────────────────────────────────────────────
```

AgentClass

classes: AgentName $\rightarrow\!\!\!\!\!+$ AgentClassDecl

A schema named *AgentRole* denotes the component of an agent role: an association to the agent class, a finite set of role name and a partial function, *plays* to map the agent class to roles it plays.

AgentRole

agent: AgentClass
name: $\mathbb{F}$ Name
plays: AgentClass $\rightarrow\!\!\!\!\!+$ Name

dom plays = agent
$\bigcup$ (ran plays) = name
[1] $\forall i, j$: dom plays | $i=j \bullet$ plays(i) $\cap$ plays(j) $= \varnothing$
[2] $\forall i$: dom plays | plays(i) $= r_1 \Rightarrow \exists r_2 \bullet$ plays(i) $= r_2 \wedge r_1 \neq r_2$
[3] $\forall i$: dom plays | plays(i) $= r_1 \Rightarrow \Diamond$ (plays(i) $= r_2$)
[4] $\forall i$: dom plays, $\forall r_1$: ran plays | plays(i) $= r_1 \Rightarrow \Box$ (plays(i) $= r_1$)

The constraints defined for the *AgentRole* are based on the constraints defined by Zambonelli [Zambonelli et al.01]:

1. No agent can simultaneously play more than one role.
2. Every agent that plays role r_1 must play at least one other role.
3. Every agent that plays role r_1 will eventually play role r_2.
4. Once an agent plays role r_1, it plays r_1 forever.

Pattern Association

A pattern association defines a semantic relationship between the interfaces of two patterns. It has exactly two *AssociationEnd*. Each end is a pattern interface, which refers to services offered by the agent.

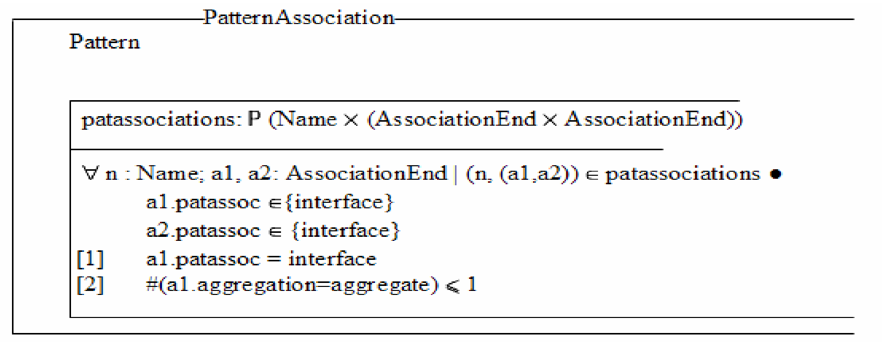
```
assockind ::= none | aggregate | composite
patassockind ::= service
```

AssociationEnd

rolename: Name
multiplicity : $\mathbb{P}\ \mathbb{N}$
attachedclass: AgentName
aggregation: assockind
patassoc: patassockind
attachedpattern: PatternName
name: Name

multiplicity $\neq \{0\}$

A schema named *AssociationEnd* denotes the components of an association end: a role name, a multiplicity constraint, an attached class, an attached pattern. The variable aggregation denotes whether or not the attached class is an aggregate. This variable can take the values none, aggregate or composite. The variable *patassoc* denotes the type of the attached pattern.



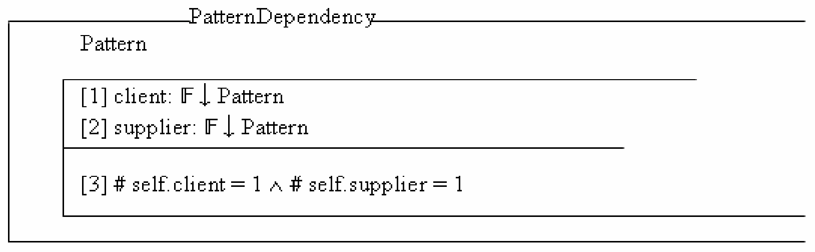
```
PatternAssociation
Pattern
  patassociations: ℙ (Name × (AssociationEnd × AssociationEnd))
  ∀ n : Name; a1, a2: AssociationEnd | (n, (a1,a2)) ∈ patassociations •
          a1.patassoc ∈{interface}
          a2.patassoc ∈ {interface}
  [1]     a1.patassoc = interface
  [2]     #(a1.aggregation=aggregate) ≤ 1
```

The constraints defined for the *PatternAssociations* are:

1. The type of *AssociationEnd* of a *PatternAssociations* is an interface offered by the agent.
2. At most, one *AssociationEnd* is an aggregation.

Pattern Dependency

A pattern dependency indicates a semantic relationship between two patterns – a situation in which a change to the source pattern may require a change to the target pattern. This type of relationship is used in POD at the high-level design phase where the system is modeled as a composition of design patterns and *PatternDependency* relationship between those patterns.



```
PatternDependency
Pattern
  [1] client: 𝔽↓ Pattern
  [2] supplier: 𝔽↓ Pattern
  [3] # self.client = 1 ∧ # self.supplier = 1
```

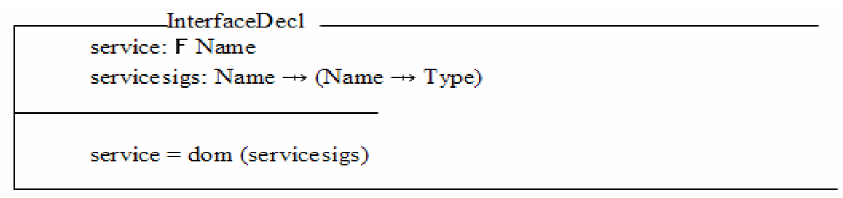
The constraints defined for the *PatternDependency* are:

1. The client of a *PatternDependency* is a pattern.
2. The supplier of a *PatternDependency* is a pattern.
3. A *PatternDependency* is only between two elements.

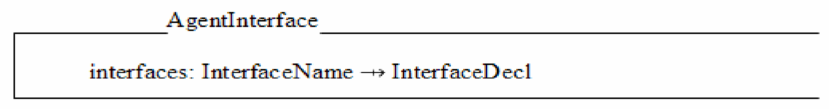
Pattern Interfaces

A pattern interface is a set of classes used to define how the pattern interfaces to other patterns. In our approach, the set of classes refer to the set of services offered by the agent. To qualify as a design component, a pattern has to have interfaces [Szyperski98]. The interface helps in gluing patterns together at the high level of abstraction and hiding internal details and design decisions made within the pattern.

A schema named *InterfaceDecl* denotes the components of an interface: a finite set of services. A partial function *servicesigs* is defined to map services to their parametes and also to map each parameter to its type. Agent interface names should be unique in the enclosing name space. Thus, the set of interfaces is defined as a partial function from *InterfaceName* to *InterfaceDecl*.



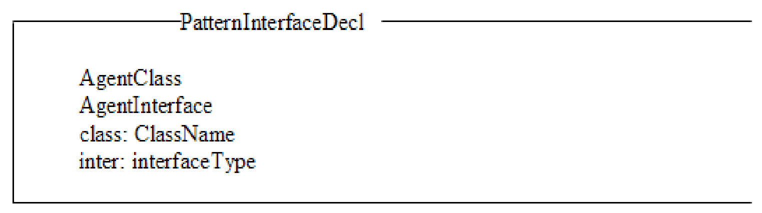
```
InterfaceDecl
service: 𝔽 Name
servicesigs: Name ⇸ (Name ⇸ Type)
────────────
service = dom (servicesigs)
```



```
AgentInterface
interfaces: InterfaceName ⇸ InterfaceDecl
```

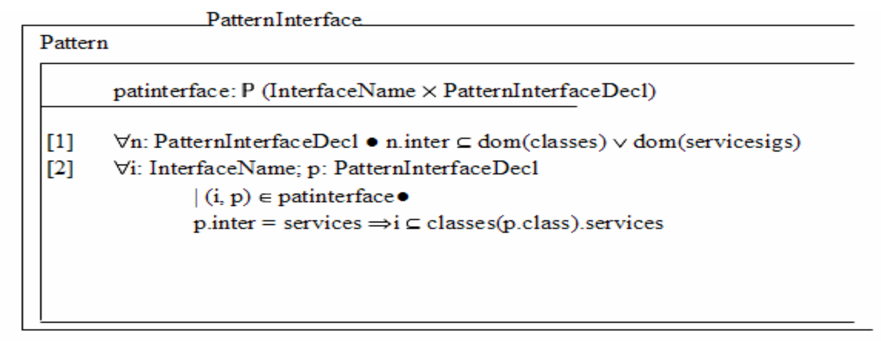
A schema named *PatternInterfaceDecl* denotes the components of a pattern interface. The variable *inter* denotes the type of the interface. This variable can take the values of classes.

```
interfaceType ::= classes
```



```
PatternInterfaceDecl

AgentClass
AgentInterface
class: ClassName
inter: interfaceType
```



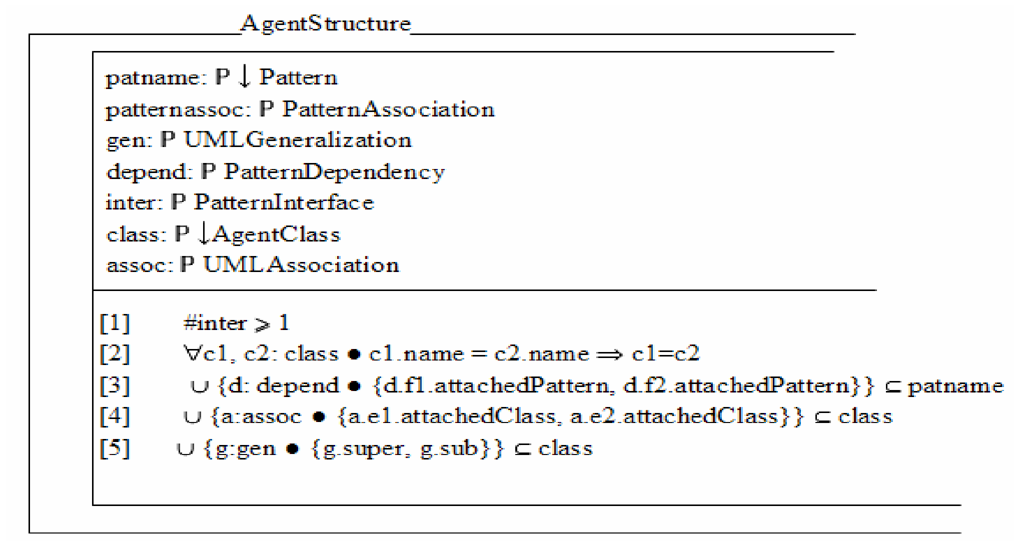
```
PatternInterface
Pattern
────────────
patinterface: ℙ (InterfaceName × PatternInterfaceDecl)
────────────
[1]   ∀n: PatternInterfaceDecl • n.inter ⊂ dom(classes) ∨ dom(servicesigs)
[2]   ∀i: InterfaceName; p: PatternInterfaceDecl
            | (i, p) ∈ patinterface •
            p.inter = services ⇒ i ⊂ classes(p.class).services
```

The constraints described well-formedness rules for the *PatternInterface* are:

1. For each method in a *PatternInterface*, the namespace of the pattern must have a matching service.
2. For each class in a *PatternInterface*, the internal contents of the pattern (pattern classes) must have a matching class.

Agent Structure

A pattern for agent structure used in POD is a unit of building agent software design that encapsulates information; it encapsulates a solution to a frequent recurring design problem, it hides lower level design decisions, and it offers interfaces to other design artifacts. In this sense, the pattern becomes a design component with interfaces. In POD, the pattern to model agent structure can only contains Agent Classes, Associations, Generalizations, Pattern Associations, Pattern Dependency and Pattern Interface.



AgentStructure

patname: $\mathbb{P} \downarrow$ Pattern
patternassoc: $\mathbb{P}$ PatternAssociation
gen: $\mathbb{P}$ UMLGeneralization
depend: $\mathbb{P}$ PatternDependency
inter: $\mathbb{P}$ PatternInterface
class: $\mathbb{P} \downarrow$AgentClass
assoc: $\mathbb{P}$ UMLAssociation

[1] $\#inter \geqslant 1$
[2] $\forall c1, c2: class \bullet c1.name = c2.name \Rightarrow c1=c2$
[3] $\bigcup \{d: depend \bullet \{d.f1.attachedPattern, d.f2.attachedPattern\}\} \subset patname$
[4] $\bigcup \{a{:}assoc \bullet \{a.e1.attachedClass, a.e2.attachedClass\}\} \subset class$
[5] $\bigcup \{g{:}gen \bullet \{g.super, g.sub\}\} \subset class$

The constraints defined for the *AgentStructure* are:

1. A pattern should have at least one *PatternInterface* to qualify as a design component.
2. No two elements have the same name inside a pattern.
3. A dependency relationship between two patterns is only of type *PatternDependency*.
4. Classes that are involved in associations should be classes in the pattern. The constraints for the association relationships is based on the UML association constraints defined by Kim and Carrington [Kim&Carrington99].
5. Classes involved in generalizations should be classes in the pattern. The constraints for the generalization relationships is based on the UML generalization constraints defined by Kim and Carrington [Kim&Carrington99].

Agent Organization

A pattern for agent organization used in POD is reused from the existing catalogues of organisation patterns for object-oriented systems. As pointed by Zambonelli [Zambonelli et al.01], there are similarities between agent organization in multi-agent system with respect to the most widely used organisational structures. In defining rules for the agent organization, we need to express constraints between agent roles, between agent

```
AgentOrganization
  interact: P((AgentRole × AgentRole) × AgentInteraction)
  ─────────────
  ∀r1,r2: AgentRole, i: AgentInteraction • (r1.name ≠ r2.name) ∧
                              ((r1,r2), i) ∈ interact
```

interactions and between agent roles and its interaction. Hence, in one organization, there should exist an interaction that involves one agent role, r1 and another agent role, r2.

Agent Interaction

A multi-agent system consists of an organized group of agents which interact with each other. This interaction is generally regarded as the foundation for cooperative and competitive behaviour in autonomous agents. In POD, an agent interaction should consists at least the following elements:

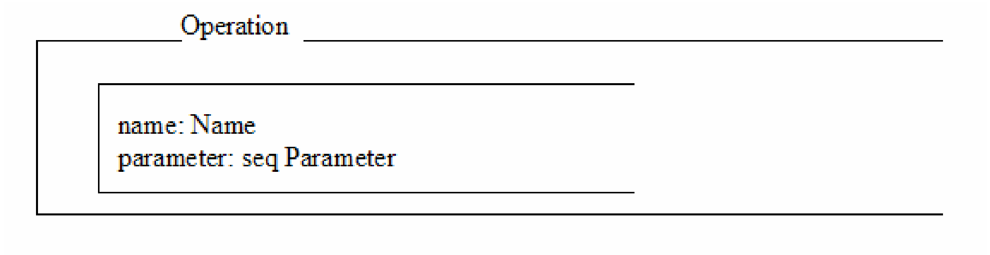
- Type of participants
- Interaction states
- Events which trigger states changes
- Valid actions given the participant and the state.

Prior to formalizing the agent interaction concept, a given set, *State* is defined, from which the names of all states are drawn. A schema named *Transition* denotes the components of a transition in one interaction: a source and target state, a guard expression, events which trigger the transition and actions taken due to the transition.

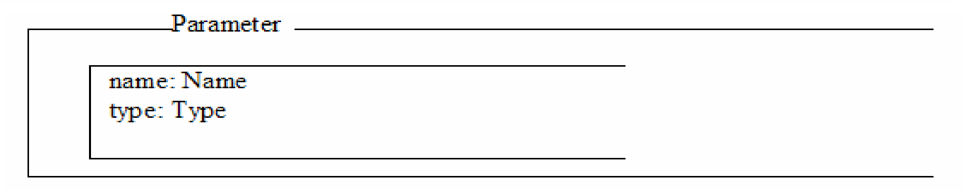
```
[State]
```

```
Transition
  source, target: P State
  guard: P Guard
  trigger: P Event
  effect: P Action
```

Operations used in an event are defined as follows. The attributes name and a sequence of operation parameter are formalized as variables in the *Operation* class.

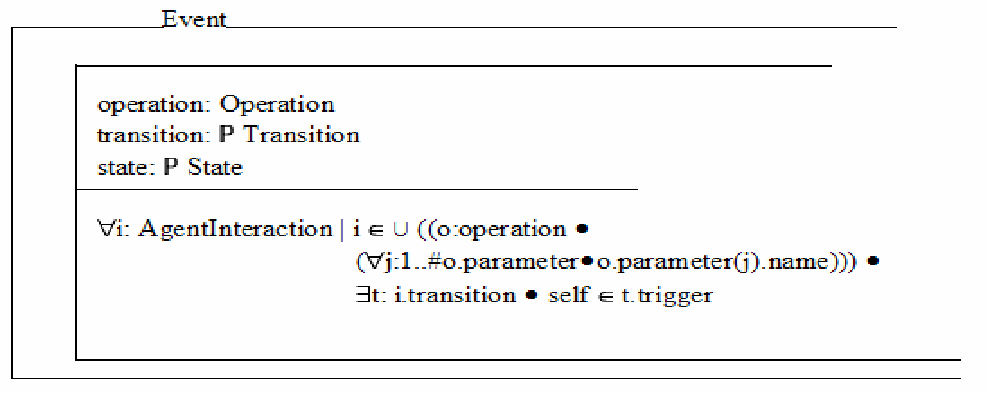


```
Operation
  name: Name
  parameter: seq Parameter
```



```
Parameter
  name: Name
  type: Type
```

As denoted by the schema *Event*, an event in the interaction consists of operation, an association to transition and state.

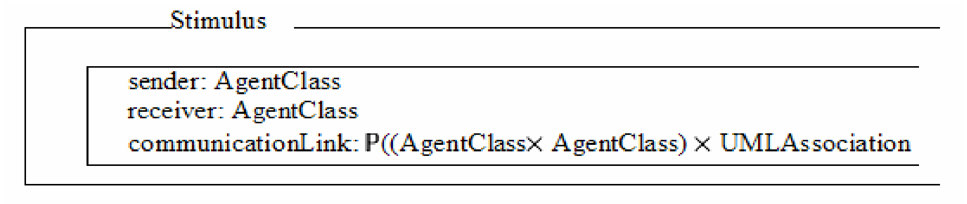


```
Event
  operation: Operation
  transition: ℙ Transition
  state: ℙ State
  ─────────────
  ∀i: AgentInteraction | i ∈ ∪ ((o:operation •
                (∀j:1..#o.parameter• o.parameter(j).name))) •
                ∃t: i.transition • self ∈ t.trigger
```

For consistency, the following constraint is defined for the *Event*:

1. For an event, there should be an agent interaction includes a transition triggered by the event.

An action is a specification of an executable statement that should be performed as a result of the execution of a transition or entering to or exiting from a state. A schema *Stimulus* denotes a component of one communication: a sender class, a receiver class and communication link between the sender and receiver. For an action, there should be an association between the sender agent and the receiver agent in its static model.



```
Stimulus
  sender: AgentClass
  receiver: AgentClass
  communicationLink: ℙ((AgentClass× AgentClass) × UMLAssociation
 ___________Action ____________________________________
|
|    _________________________________________
|   |
|   | stimulus: ℙ Stimulus
|   |_______________________________
|   | ∀a: stimulus• a.sender ≠ a.receiver ⇒
|   |         ((a.sender,a.receiver),a.assoc) ∈ communicationLink
|   |_____________________________________________________
|
|_________________________________________________________
```

```
 ___________AgentInteraction__________________________________
|
|    ___________________________________________
|   |
|   | transition: ℙ Transition
|   |_______________________________
|   | [1] ∀e:Event | (o:e.operation •(∀i:1..#o.parameter •o.parameter(i).name)) ⇔
|   |                          (∃t:transition• e ∈ t.trigger)
|   |
|   | [2] ∀a: ∪ {t:transition •t.effect} • ∀s:a.stimulus •
|   |                 s.sender ≠ s.receiver ⇒
|   |                       ((s.sender, s.receiver),s.assoc) ∈ communicationLink
|   |___________________________________________________________
|
|_______________________________________________________________
```

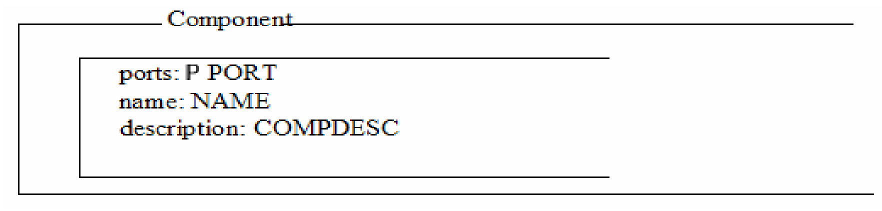
The constraints defined for the *AgentInteraction* are:

1. For all events associated with the interaction, there should be transitions describing the detailed behaviour of the events.
2. For all actions occuring in the interaction, there should be an association between the sender agent and the receiver agent.

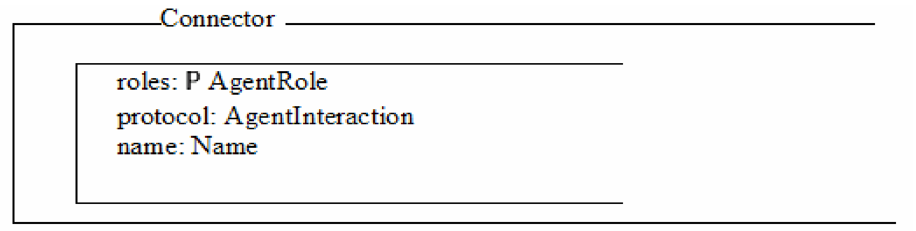
Agent Architecture

Prior to formalizing an agent architecture pattern, two given sets, *PORT* and *COMPDESC* are defined for the agent architecture. As outlined by Abowd [Abowd et al.95], an architecture style consists of components (the locus of computation), connectors (define the interaction between components) and configuration (the collections of interacting components and connectors). In applying this concept to the agent world, an architectural component consists of component name, ports together with a description of its computation. This is represented in an Object-Z schema named *Component*. An architectural connector is then modeled as a collection of agent roles and a description of its interaction protocol. This is represented in a schema named *Connector*.

```
[PORT, COMPDESC]
```

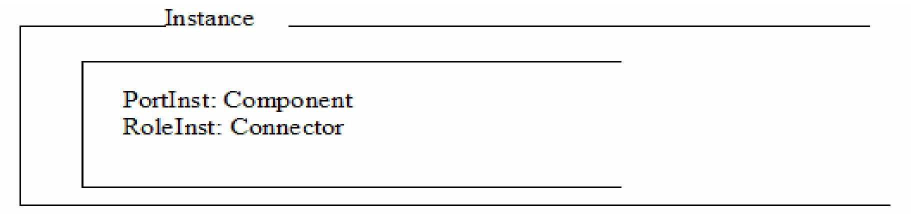


```
Component
  ports: P PORT
  name: NAME
  description: COMPDESC
```



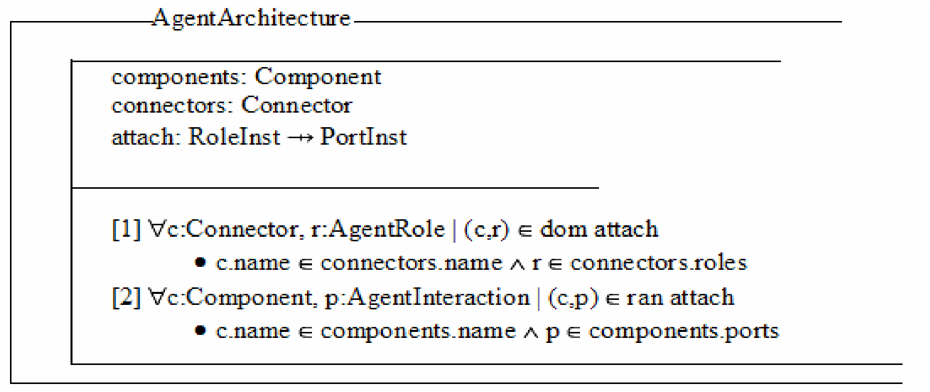
```
Connector
  roles: P AgentRole
  protocol: AgentInteraction
  name: Name
```

A schema named *Instance* is introduced to denote components of instances in the architecture, named PortInst, a variable for the component instance and RoleInst for the connector instance.



```
Instance
  PortInst: Component
  RoleInst: Connector
```

A schema *AgentArchitecture* is defined to represent the configuration. It consists of the component, connectors and association between the component and connectors instances models by the partial function, *attach*.



```
AgentArchitecture
  components: Component
  connectors: Connector
  attach: RoleInst ⇸ PortInst
  ----------
  [1] ∀c:Connector, r:AgentRole | (c,r) ∈ dom attach
          • c.name ∈ connectors.name ∧ r ∈ connectors.roles
  [2] ∀c:Component, p:AgentInteraction | (c,p) ∈ ran attach
          • c.name ∈ components.name ∧ p ∈ components.ports
```

The constraints described well-formedness rules for the *AgentArchitecture* are:

1. Any role instances in the attach is an agent role for some named connector in the architecture.
2. All port instances described by the agent architecture appear on an actual component instance.

6 PATTERN MODELS

This paper develops a precise design patterns model for agent world by directly instantiating the Object-Z specifications defined in the pattern metamodel. An example is given in the following sub-section. Any conformance checks on the pattern model can be achieved using reasoning techniques developed for Object-Z on the Object-Z specification.

Observer pattern model

This section illustrates an example of instantiating the Observer pattern [Gamma et al.95]. Figure 4 shows a visual view to describe the pattern. In representing the patterns for agent structure, the proposed concept follows the representation of the POAD constructional design patterns. By following POAD constructs, agent structure is treated as design components which can be glued together at high design level. This composition defines the agent application overall structure.

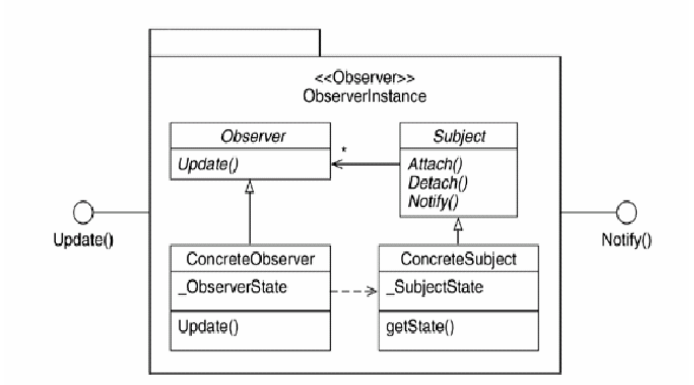


Figure 4 Observer pattern

[State, Interface]

Observer

subjects: $\mathbb{P}$Subject

update $\hat{=}$ [(Δ subjects), $s?$: Subject $\bullet$ $\theta s' = \theta s?$]

ConcreteObserver

Observer

observerState: State

[1] update $\hat{=}$ [(Δ subjects), $s?$: Subject $\bullet$ $\theta s' = \theta s? \wedge$
observerState = subjects.getState]

Subject

$observers: \mathbb{P}\,Observer$

attach

$\Delta(observers)$
$o?: \mathbb{P}\,Observer$

$o? \notin observers$
$observers' = observers \cup \{o?\}$

detach

$\Delta(observers)$
$o?: \mathbb{P}\,Observer$

$o? \in observers$
$observers' = observers \setminus \{o?\}$

$notify \mathrel{\widehat=} attach \parallel observers.update$

ConcreteSubject

Subject

$subjectState: State$

getState

$a!: State$

$a! = subjectState$

$[2]\ notify \mathrel{\widehat=} [\ (\Delta\, subjectState), s!: ConcreteSubject \bullet attach \wedge$
$\theta s!.subjectState = \theta s!.subjectState' \parallel observers.update\ ]$

ObserverPattern

$subj: \mathbb{P}\,Subject$
$obs: \mathbb{P}\,Observer$
$concsubj: \mathbb{P}\,ConcreteSubject$
$concobs: \mathbb{P}\,ConcreteObserver$
$inter: \mathbb{P}\,Interface$

[3] $concsubj \subseteq subjects$
[4] $concobs \subseteq observers$
[5] $\forall sub: subj \bullet sub.observers \subseteq obs$
[6] $\forall ob: obs \bullet ob.subjects \subseteq subj \wedge \# subj \leqslant 1$
$\{subj.notify, obs.update\} \in inter$

The constraints defining the observer pattern are:

1. In an observer's update operation, the notifying subject is asked to reveal its new state (via the getState operation), which then becomes the new state of the observer.

2. The subject object will notify all the observers objects attached to it when its subject's state changes. The subject class will invoke the update operation of each observer to inform those observers of the subject's state change.
3. Concrete Subject class inherits from the Subject class.
4. Concrete Observer class inherits from the Observer class.
5. Observer objects are registered and associated with subjects. Each subject, then, has a registered set of observers.
6. Each observer is associated with only one subject.

7 RELATED WORKS

Several successful experiences have reported on the advantages of using design patterns in designing agent applications. Generally, we classify design approach that utilize design patterns into adhoc and systematic approach. Using an adhoc approach, a design pattern records a solution and forces and consequences of applying this solution. Kolp [Kolp et al.01] presents a set of social patterns as part of the Tropos [Bresciani et al.04] methodology to describe the general architecture of a system under construction. Cossentino [Cossentino et al. 03] presents the design of a particular type of agent pattern immersed in the PASSI methodology. They defined a pattern as consisting of a model and an implementation code. However, this is not usually sufficient to systematically develop applications using patterns. This is simply because there is no process to guide the development and to integrate the pattern with other design artifacts.

A systematic approach to design with patterns goes further beyond just applying a certain pattern. Systematic approaches can be classified as pattern languages and development processes. A pattern language provides a set of patterns that solve problems in a specific domain. Pattern languages not only document the patterns themselves but also the relationships between these patterns. They imply the process to apply the language to completely solve a specific set of design problems. Weiss [Weiss03] proposed a generic agent pattern language. It documents the forces involved in agent-based design and key agent concepts. A domain-specific pattern language for agent-based manufacturing has been proposed by [Weiss03]. The relationships between the patterns are made explicit in such a way that they guide a developer through the process of designing a system. A systematic development process, on the other hand, defines a pattern composition approach, analysis and design steps, design models, and tools to automate the development steps. Such development process produces consistent designs each time the process steps are conducted. This paper is concerned with systematic development processes because they are the way to repeatable software design practice. The concept presented in this paper is an initial step to define a methodology to systematically design agent applications using patterns.

8 CONCLUSION AND FUTURE WORKS

Overall, our work is inspired by the component-based approaches. Specifying the design patterns as design components allows for pattern composition and integration at design level. The syntactic structures are precisely described using the Object-Z notation. This description is then used as a basis for translation rules from POD contructs to Object-Z constructs. The Object-Z specifications are formal models of pattern diagrams. Thus, semantic analysis of these pattern diagrams can take place on these Object-Z specifications using proof techniques provided for Object-Z. Any inconsistency or error discovered during the analysis provide feedback on the pattern diagrams. The Object-Z specifications introduced in this paper are type-checked using the Wizard type-checker.

The formal approach presented in this paper, brings precision to POD descriptions by eliminating ambiguities from the description. The formal pattern metamodel and models facilitate a rigorous reasoning about patterns as design components. After identifying the concept for the systematic agent development using patterns, we are now working on defining a methodology for designing multi-agent systems according to the proposed metamodel. The aim is, the methodology could be applied or embedded to the existing agent methodology. The motivation is to make the existing agent design methodology like Tropos [Bresciani04], MaSE [DeLoach et al.01], Gaia [Zambonelli et al.03] and PASSI [Cossentino03] more mature.

REFERENCES

[Aridor&Lange98] Aridor Y. and Lange D.: "Agent Design Patterns: Elements of Agent Application Design", in Autonomous Agents (Agents '98), ACM Press, 1998.

[Abowd et al.95] Abowd G., Allen R. and Garlan D.: "Formalizing Style to Understand of Software Architecture", ACM Transaction of Software Engineering Methodology, Volume 4, Number 4, 1995, pp. 319-364.

[Bresciani et al. 04] Bresciani, P., Giorgini, P., Giunchiglia, F., Mylopoulos, J and Perini, A.: "TROPOS: An Agent-Oriented Software Development Methodology" in *Journal of Autonomous Agents and Multi-Agent Systems*, Kluwer Academic Publishers Volume 8, Issue 3, Pages 203 - 236, May 2004.

[Cossentino et al. 03] Cossentino, M., Sabatucci, L, Sorace, S. and Chella, A,: "Patterns reuse in the PASSI methodology", *Fourth International Workshop Engineering Societies in the Agents World (ESAW'03)* - 29-31 October 2003, Imperial College London.

[DeLoach et al. 01] DeLoach, S. A., Wood, M. F. and Sparkman, C. H.: "Multiagent Systems Engineering", *The International Journal of Software Engineering and Knowledge Engineering*, Volume 11 no. 3, pp. 231-258, June 2001.

[Deugo et al. 01] Deugo, D., Weiss, M. and Kendall, E.*: Coordination of Autonomous Internet Agents: Models, Technologies and Applications*, chapter Reusable Patterns for Agent Coordination, Springer, 2001.

[Gamma et al. 95] Gamma, E., Helm, R., Johnson, R. and Vlissides, J.: *Design Patterns, Elements of Reusable Object-Oriented Software*. Addison-Wesley, Longman, 1995.

[Kim&Carrington99] Kim, S. K and Carrington, D.: Formalizing the UML Class Diagram Using Object-Z, in the Proceedings of the 2nd. International Conference on the Unified Modeling Language (UML' 99), vol. 1723 of Lecture Notes in Computer Science, Springer, 1999

[Kim&Carrington04] Kim, S. K. and Carrington, D.: Using Integrated Metamodeling to Define OO Design Patterns with Object-Z and UML, in the Proceedings of the 11th. Asia-Pacific Software Engineering Conference (APSEC'04), 2004.

[Kolp et al.01] Kolp, M., Castro, J. and Mylopoulos, J.: "A social organization perspective on software architectures", in *1st. International Workshop from Software Requirements to Architectures*, 2001.

[Lind03] Lind, J.: "Pattens in Agent-Oriented Software Engineering". in Giunchiglia, F., Odell, J. and Weiss, G., editors, Agent-Oriented Software Engineering III, vol. 2585 of *Lecture Notes in Computer Science*, Springer, 2003.

[Sauvage04] Sauvage, S. Design Patterns for Multiagent Systems Design. In *MICAI'04*, in *Lecture Notes in Computer Science*, Volume 2972, pages 352–361, Mexico, April 2004. Springer-Verlag, Heidelberg.

[Sycara98] Sycara K.: "Multiagent Systems", AI Magazine 19(2), 1998, pp. 79-92.

[Szyperski98] Szyperski, C.: *Component Software: Beyond Object-Oriented Programming.* Addison Wesley Longman, Reading, Mass, 1998

[Weiss03] Weiss M.: "A Pattern Language for Motivating the Use of Agents", (P.Giogini et al. (Eds.): AOIS 2003, LNAI 3030, pp. 142-157, 2004).

[Yacoub&Ammar04] Yacoub S. and Ammar H. H.: Pattern-Oriented Analysis and Design: Composing Patterns to Design Software Systems, Addison-Wesley, 2004.

[Zambonelli et al.03] Zambonelli F., Jennings N. and Wooldridge M.: "Developing multiagent system: The Gaia Methodology", ACM Transactions on Software Engineering and Methodology, Volume 12, Issue 3, July 2003, pp. 317-370.

[Zambonelli et al.01] Zambonelli F., Jennings N. and Wooldridge M.: "Organisational Rules as an Abstraction for the Analysis and Design of Multi-Agent Systems", International Journal of Software Engineering and Knowledge Engineering, Volume 11, Number 1, February 2001, pp. 302-328.

About the authors

Radziah Mohamad is a lecturer at the Software Engineering Department, Faculty of Computer Science and Information Systems, Universiti Teknologi Malaysia. She is currently pursuing her PhD at Universiti Teknologi Malaysia. Her research interests include pattern-oriented design, formal methods, agent-oriented software engineering and component-based software engineering. She can be reached at radziahm@utm.my

Safaai Deris is a Professor in the Department of Software Engineering at Universiti Teknologi Malaysia, Malaysia. His research interests include artificial intelligence, object-oriented technology, software engineering, bioinformatics and database technology. He can be reached at safaai@fsksm.utm.my

Hany H. Ammar is a Professor in the Department of Computer Science and Electrical Engineering at West Virginia University. He has been recently a Principal Investigator on a number of research projects on Fingerprint Image Comparison Software funded by Loral Federal Systems, and on Software Verification and Validation funded by NASA Goddard. His research interests include software engineering, reliability engineering, biometrics and computer architecture. He can be reached at Hany.Ammar@mail.wvu.edu