A Secure Architectural Description Language for Agent Systems

Haralambos Mouratidis
School of Computing and Technology, University of East London, Barking Campus, Longbridge Road, RM8 2AS, England.
H.Mouratidis@uel.ac.uk

Manuel Kolp,
ISYS- Information Systems Research Unit, University of Louvain, Place des Doyens,
B - 1348 Louvain-La-Neuve, Belgium
kolp@isys.ucl.ac.be

Stephane Faulkner,
Management Research Unit, University of Namur
8 rempart de la vierge, 5000 Namur, Belgium
stephane.faulkner@fundp.ac.be

Paolo Giorgini
Department of Information and Communication Technology, University of Trento, Via Sommarive, 14 38050, Rovo, Trento, Italy
paolo.giorgini@dit.unitn.it

ABSTRACT

Multi-agent systems are now being considered a promising architectural approach for building Internet-based applications. One of the most critical and important aspects of software deployed on the web has always been the security of their architectures. However, despite considerable work in software architecture during the last decade, few research efforts have aimed at truly defining languages for designing and formalizing agent architectures and more specifically secure ones. This paper identifies the foundations for an architectural description language (ADL) to specify secure multi-agent systems. We propose a set of system design primitives and conceptualize it with the Z specification language to capture a "core" architectural model to build secure MAS architectures. We apply it on an e-commerce example to illustrate our proposal.

Categories and Subject Descriptors

D.2.11 [**Software Architectures**]: Languages

I.2.11 [**Artificial Intelligence**]: Distributed Artificial Intelligence – Multiagent Systems

General Terms

Design, Security, Languages.

Keywords

Architectural Description Languages, Security, Multiagent Systems

1. INTRODUCTION

The rise of the Internet and World-Wide-Web technologies has resulted in a greater and wider use of information systems not only by major corporations and governments but also from individual users. Due to this wide usage, many of these systems manage and store information that is considered sensitive, such as medical, financial and private data. With the introduction of such information to software systems, and all the advantages that this might introduce (such as easy access and share); the need to secure systems that contain such information becomes a necessity rather than an option. Imagine, for instance, the effects of medical records of individuals becoming widely available.

However, securing such systems is not an easy task. This argument is supported by research [1, 2] as well as by various surveys (see for example www.cert.org) regarding the security of current information systems. This is mainly due to the requirements [2, 3] and challenges [1, 2] imposed when considering security in the development of information systems. Not surprisingly, this has been identified [1,2,3,4] and researchers are looking for new software development paradigms that cope with such requirements and provide answers to the security challenges.

One promising source of ideas for deploying Internet and web-based applications is the area of multiagent system architectures. They appear to be more flexible, modular and robust than traditional; including object-oriented ones. They tend to be open and dynamic in the sense they exist in a changing organizational and operational environment where new components can be added, modified or removed at any time. Moreover, the integration of security issues within an agent system context will require for the agents of the system to consider the security requirements, when specifying their objectives and interactions, and therefore cause the propagation of security requirements to the whole system.

However, such architectures introduce a degree of complexity. To cope with this ever-increasing complexity of the design, it has been recognized the value of making explicit architectural descriptions [5]. To help developers with such descriptions, architectural descriptions languages and architectural styles are employed. An architectural description language (ADL) provides a formal syntax and semantics for specifying architectural abstractions in a descriptive notation. Unfortunately, despite considerable work in defining languages for architectural design (see e.g., [5,6,7]) few research efforts have aimed at truly defining languages for agent architectural design and even these do not adequate include security. This paper deals with this issue in defining a "core" set of structural, behavioural and security concepts, including relationships and constraints that are fundamental to propose an agent architectural description language. The language, called SKwyRL-ADL, includes an agent, a security and an architectural model and aims at describing secure multi-agent systems, more specifically those based on the BDI (belief-desire-intention) model.

The rest of the paper is organized as follows. Section 2 introduces the main concepts of SKwyRL-ADL including the security aspects. Section 3 describes our agent oriented approach on an e-commerce system secure architectural specification. Section 4 presents the implementation of the system and finally Section 5 concludes the paper.

2. SECURE SKwyRL ADL

The SKwyRL (Socio-Intentional ArChitecture for Knowledge Systems & Requirements ELicitation – http://www.isys.ucl.ac.be/skwyrl) project proposes an agent ADL called SKwyRL-ADL [8] that offers a set of concepts, based on the Belief-Desire-Intention (BDI) agent model to formally specify secure agent-oriented architectures. SKwyRL-ADL is compliant with most of the classical ADLs proposed on the software architecture [6] and security literature [9,10,11]. Figure 1 provides a description of these concepts together with their relationships.

SKwyRL-ADL is composed of three sub-models: the agent model, the security model and the architectural model. The Z specification language [12] is used to formally describe SKwyRL-ADL concepts. Z is widely used as a formal specification language in the field of software architecture community and has been shown to be clear, concise and relatively easy to learn. Due to lack of space, we only detail and formalize some aspects of our ADL. We refer the reader to [Fau04] for a more complete formalization.

2.1 The agent Model

The agent model captures the states of an agent and its potential behaviour. The agent needs knowledge about the environment in order to reach decisions. Knowledge is contained in agents in the form of one of many knowledge bases. A Knowledge base consists of a set of beliefs that the agent has about the environment and a set of goals that it pursues.

Beliefs describe the environment of the agent in terms of states of objects with individual identities and properties, and relations on objects as being either true or false. We use *predicate* symbols to specify a particular relation that holds (or fails to hold) between several objects, and *terms* to represent objects. Each term can be build from constant, variable or function symbols. From the above primitives, we can define an *AtomicBelief*. The set of all predicate, function, constant and variable symbols are denoted by [*PredSymb*], [*Function*], [*Constant*], and [*Variable*], respectively.

[*PredSymb*], [*Function*], [*Constant*], [*Variable*]

[*Terms*]:= Function(Term,…) | Constant | Variable

AtomicBelief

head: *PredSymb*

terms: seq Term

head ≠ ∅ ∧ terms ≠ ∅

A *Belief* is specified either as an *AtomicBelief*, a negated *AtomicBelief*, a series of *AtomicBeliefs* connected using logic connectives, or an *AtomicBelief* characterized with a temporal pattern. The following temporal patterns are used in SKwyRL-ADL: ○ (in the next state), ● (in the previous state), ◊ (some time in the future), ♦ (some time in the past), □ (always in the future), ■ (always in the past), $\mathcal{W}$ (always in the future unless), and $\mathcal{U}$ (always in the future until).

[*Belief*]:= *AtomicBelief*
| ¬*AtomicBelief*
| *AtomicBelief* Connective *AtomicBelief*
| Temp_Pattern *AtomicBelief*

With Connective → ∧ | ∨ | ⇒
[*Temporal_Pattern*]:= ○ | ● | ◊ | ♦ | □ | ■ | W | U

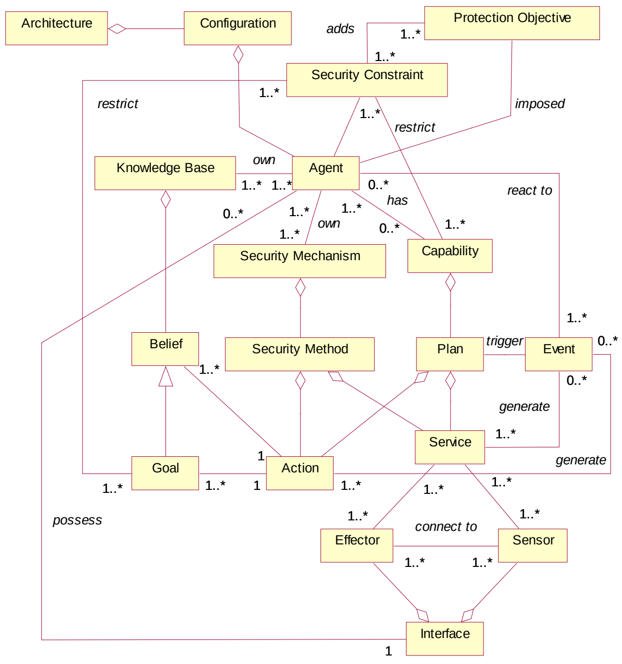


Figure 1: Secure SKwyRL ADL Meta Model

A goal is a set of objects that describe an environment state that an agent wants to bring about. We consider goals according to four patterns:

Achieve: P ⇒ ◊ Q (Q holds in current or some future state)
Cease: P ⇒ ◊ ¬Q
Maintain: P ⇒ □ Q (Q holds in current and all future states)
Avoid: P ⇒ □ ¬Q

With respect to beliefs, goals can be specified as follows:

[*GoalPattern*] := Achieve | Cease | Maintain | Avoid

Goal

head: *GoalPattern*

state: *Belief*

head ≠ ∅ ∧ state ≠ ∅

The goal patterns influence the set of possible agent behaviors: achieve and cease goals generate actions, plans, or events, while maintain and avoid goals restrict them. When a goal is required, the agent identifies a set of plans to achieve or maintain this goal. From then on, the agent chooses according to its current beliefs which of these plans will be executed.

A plan defines the sequence of actions to be chosen by the agent to accomplish a task or achieve a goal. Actions are basic executable commands of agent behaviour. Plans are selected by agents. Selected plans constrain the agent's behaviour and act as intention. Intentions represent the deliberative states of the agent, i.e., which plans the agent has chosen for possible execution. A plan consists of:

- An invocation condition detailing the circumstances, in terms of event, that cause the plan to be triggered;
- An optional context that defines the preconditions of the plan, i.e., what must be believed by the agent for a plan to be selected for execution;
- The plan body, that specifies either the sequence or formulae that the agent needs to perform;
- An end-state that defines the postconditions under which the plan is succeeded;
- And finally a set of internal actions that specify what happens when a plan fails or succeeds.

A plan is specified as follows:

[*PlanName*], [*AtomicPlan*]:= Action | Service

Plan

name: *PlanName* Invocation: *Invocation* context: *Belief* body: seq *AtomicPlan* endState: *Belief* succeed: seq *Atomicplan* failure: seq *AtomicPlan*
name $\neq \emptyset \wedge$ invocation $\neq \emptyset \wedge$ body $\neq \emptyset$

An event is something that happens in the system that can be perceived, and it is either a goal (a new goal or the remove of a goal), a belief (a new belief or the remove of a belief) or a plan (the success or failure of a plan). MAS are event-driven in the sense that agents start interacting by initiating and perceiving events. In the absence of event an agent sits idle. Whenever an event occurs, an agent initiates either a plan or a set of plans to response to that event. In this last case, the agent chooses between the plans it has available to achieve its goal. We defined two types of events: (1) An internal event that an agent posts to itself; and (2) An external event that an agent sends to other agent or to its environment.

According to the definition of event, both types are specified considering the nature of the event, which can be a goal, a belief or a plan. The key difference between belief, plan or goal events is how an agent selects plans for execution. For belief and plan events, the agent selects the first applicable plan for that event and executes an instance of that plan only. The handling of goal event is more complex. An agent can assemble a set of plans for the goal event and apply a sophisticated heuristic to choose the appropriate plans. However, for this matter, at the architectural design level where ADLs are defined, we remain completely independent from such heuristics, considering that they depend directly on the used programming environment.

Finally an event is generated either by an action that modifies beliefs or adds new goals, or by services provided by another agent. Services appear in the architectural model because they involve interactions among agents that compose the MAS. Interactions serve as basic elements to support the construction of configurations.

2.2 The Security Model

With respect to security, an agent has zero or more protection objectives and each security objective imposes one ore more security constraints on the agent. Security constraints might restrict the goals and/or the capabilities of an agent. On the other hand, an agent owns security mechanisms. A security mechanism represents a set of standard security methods that an agent might have and they help towards the satisfaction of the protection objectives of the agent. A security method defines a sequence of actions and/or services to satisfy an agent's security mechanisms.

2.2.1 Protection Objective

A protection objective indicates a desirable security attribute that an agent might have, such as integrity, and availability. An agent might impose a security objective by itself or more likely a protection objective is imposed to an agent through its environment (e.g. from a security policy or through other systems/agents/stakeholders/developers). Moreover, a protection objective alters the agent's motivational state by adding constraint(s) to the agent with respect to security. A protection objective imposes one or more security constraints to an agent, and each agent might have zero or more protection objectives. A protection objective is specified as follows:

[POname], [POimposer]:= self | environment

ProtectionObjective

name*: POname* imposed_by*: POimposer* Imposed_to*: Agent* constraints*: SecurityConstraint*
name $\neq \emptyset \wedge$ *imposed_to* $\neq \emptyset \wedge$ *constraints* $\neq \emptyset$
$(\forall\ po: ProtectionObjective)\ (\forall\ ag: Agent)\ (\forall\ sc: SecurityConstraint)\ [(sc\ \varepsilon\ po) \wedge (po\ \varepsilon\ ag)] \Leftrightarrow constrain(ag,sc)$

2.2.2 Security Constraint

A security constraint defines a set of restrictions to the goals and the capabilities of the agent. These restrictions are security related and are imposed by the agent's environment (either from a security policy, other systems/agents, the developers or the stakeholders).

When a security constraint restricts a goal, the agent must identify a possible way of achieving the goal without endanger the security constraint. On the other hand, when a security constraint restricts a capability (in reality the security constraint will restrict plans and/or events of the capability) the agent must identify alternative ways of satisfying its goals without using the specific capability.

It is possible that some restrictions are communication related. For instance, a restriction that might apply for the communication of one agent with another agent, might not apply for the communication of the same agent with a third agent or vice versa. Also, a security constraint might restrict the goals/capabilities of an agent for a specific time frame. For instance, a restriction that might apply today may not be valid tomorrow. A security constraint can be specified as follows:

[*SCname*], [*SCrestriction*] *:* Goal | Capability

[*SCtimeFrame*]:= All | Function, [*SCcommunication*]:= Agent | All

SecurityConstraint

name*: SCname* restricts*: SCrestriction*

timeFrame*: SCtimeFrame*

constraints: SCcommunication

name ≠ ∅ ∧ restricts ≠ ∅

(∀ ag: Agent) [(g: Goal ε ag) (cap: Capability ε ag) (sc: SecurityConstraint ε ag)] ⇔ restrict(g, sc) ¿ restrict(cap,sc)

2.2.3 *Security Mechanism*

A security mechanism represents a set of standard security methods that an agent might have and they help towards the satisfaction of the protection objectives of the agent.

The security mechanism allows structuring the security behaviour of an agent with respect to its security information. Internally, each security mechanism is structured by a set of different security methods, allowing system architects firstly to build up a library of different security methods, and secondly to build different security mechanisms for different agents of the system, by adding and removing security methods from the library. Because of this, a security mechanism could be either available or unavailable to an agent at a specific point of time.

The security mechanism could be structured by different kind of security methods. Some of them able to detect security breaches, some of them able to prevent security breaches, and some of them able to recover from security breaches. Therefore, the type of a security mechanism could be one of the following: (1) detecting: which involves only detection security methods; (2) preventing: which involves only prevention security methods; (3) recovering: which involves only recovery security methods;(4) combinational: which involves security methods of all types

A security mechanism is specified as follows:

[SMname], [SMavailability]:= Available | Unavailable

[SMtype]:= Detecting | Preventing | Recovering | Combinational

SecurityMechanism

name*: SMname*

composed_of *:* *SecurityMethod*

type*: SMtype*

availability*: SMavailability*

help*: Protection Objective*

name ≠ ∅ ∧ composed_of ≠ ∅ ∧ type ≠ ∅

(∀ SM: SecurityMechanism) (∃ ag : Agent) • use(sm,ag)

2.2.4 *Security Method*

A security method defines a sequence of actions and/or services such as cryptographic algorithms and secure protocols used to realise the protection objectives of the agent. Each security method consists of the following:

1. An entry condition, indicating the factors (such as the invocation of specific security mechanism) that cause the method to be triggered
2. The security action, which specifies the actions/services that the agent needs to perform with respond to the security method invocation
3. An end condition that specifies the desirable conditions of the security action

The results report if the security action has failed or succeeded and what the next steps should be (these steps would be determined by whether the security action succeeded or failed). A security action has succeeded if and only if the output condition corresponds to an end condition.

2.3 The architectural Model

The architectural model describes the interactions among agents that compose the MAS. Configurations are the central concept of in architectural design [5], allowing to define the topology of a MAS. The topology is defined by a set of bindings between provided and required services. An agent interacts with its environment through an interface composed of sensors and effectors. An effector provides a set of services to the environment. A sensor requires a set of services from the environment. A service is an operation performed by an agent that interacts by dialoguing with one or several agents. Finally, the whole MAS is specified with an architecture which is composed of a set of configurations. The concept of architecture allows representing agents by one or more detailed, lower-level configuration descriptions.

Due to lack of space, this section only specifies the configuration concept. A configuration is a set of interconnecting agent instances. Because there may be more than one use of a given agent in a MAS, we distinguish the different instances of each agent type that appear in a configuration. To this end, we define the type *IAgent* representing the name given to an agent instance that has been instantiated within a configuration:

[*IAgent*]

Instantiating an agent also has the secondary effect of instantiating the services that are defined by its interface. We define provided and required service instance type such as follows:

[*IPservice*], [*IRservice*]

Once the instances have been declared, a configuration is specified by describing the collaborations. The collaborations define the topology of the configuration, showing which agent instance participates in which interactions. This is done by defining a one-to-many mapping relation between provided and required services.

[*AgentType*], [*Instance*]:= IAgent | IPservice | IRservice

Configuration

description: *AgentType*
instance: *Instance*

name ≠ ∅ ∧ invocation ≠ ∅ ∧ context ≠ ∅

collaboration: (IAgent X IRservice) ⇸ (IAgent X IPservice)

The configuration separates the descriptions of composite structures from the elements in those compositions. This allows reasoning about the composition as a whole and changing the composition without having to examine each of the individual components in a system.

3. Agent Architecture for e-commerce system

E-Media (http://www.isys.ucl.ac.be/skwyrl/emedia) is a typical business-to-consumer application we have developed using the architectural concepts explained in Section 2. The application offers an e-commerce architecture supporting the creation of information sources that facilitate the on-line transaction of products, services, and payments resulting in an effective and efficient interaction among sellers, buyers and intermediaries.

This section describes how we have applied Secure SKwyRL ADL to formally specify architectural aspects, such as interfaces, knowledge bases, security objectives, security mechanisms, and plans, of the e-Media system.

3.1 E-Media

E-Media provides an on-line interface that allows customers to examine the items on the E-Media catalogue and place orders. Customers can search the on-line store by either browsing the catalogue or querying the item database. An online search engine allows customers to search title, author/artist and description fields through keywords or full-text search. If an item is not available in the catalogue, the customer has the option to order it. Moreover, Internet communications are supported. All web information (e.g., product and customer turnover, and sales average) of strategic importance is recorded for monthly or on-demand statistical analysis. Based of this statistical and strategic information, the system permanently manages and adapts the stock, pricing and promotions policy. For example, for each product, the system can decide to increase or decrease stocks or profit margins. It can also adapt the customer on-line interface with new product promotions.

Apart from the main functional features of the system, security is a very important factor in the development of the E-Media system. Customers need to know that their information remains secure and accessible only to intended participants, and also that the risks, such as receiving wrong product because someone intercepted and changed the order, associated with the online purchase are minimized. Therefore, from the customer's point of view the main security objectives are confidentiality and integrity. Confidentiality guarantees that the information is accessible only to authorized entities and inaccessible to others, whereas integrity guarantees that information remains unmodified from source entity to destination entity.

On the other hand, the stakeholder of the E-Media system need to make sure that the system will always be available for customers to buy, it can confirm the involvement of an entity in certain communications, and it can prove the identity of an entity. In other words, the main security objectives from the e-media's stakeholder point of view are availability, non-repudiation, and authentication. Availability guarantees the accessibility and the usability of information and resources to authorized entities, non repudiation confirms the involvement of an entity in certain communications, and authentication proves the identity of an entity.

For both, the customer and the e-media stakeholder actors to satisfy their security objectives, some security constraints are imposed on their dependencies. Figure 2 models the dependencies between the customer, the E-Media stakeholder and the E-Media system along with the security constraints imposed by the first two actors on the system, using the i* model notation [13] where each node represents an actor (or system component) and each link between two actors indicates that one actor depends on the other for some goal to be attained. A dependency describes an "agreement" (called dependum) between two actors: the depender and the dependee. The depender is the depending actor, and the dependee, the actor who is depended upon. The type of the dependency describes the nature of the agreement. Goal dependencies represent delegation of responsibility for fulfilling a goal; softgoal dependencies are similar to goal dependencies, but their fulfilment cannot be defined precisely; task dependencies are used in situations where the dependee is required.

Actors are represented as circles; dependums – goals, softgoals, tasks and resources – are respectively represented as ovals, clouds, hexagons and rectangles; dependencies have the form depender $\rightarrow$ dependum $\rightarrow$ dependee. Security constraints are represented as clouds.

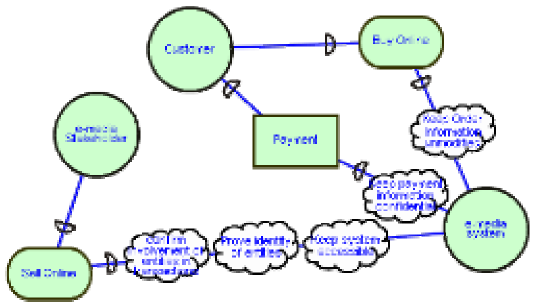

Customer
Buy Online
e-media Stakeholder
Payment
Keep Order Information unmodified
Keep payment information confidential
e-media system
Sell Online
Confirm involvement of entities in transactions
Prove identity of entities
Keep system accessible

Figure 2: E-Media dependencies

For the architecture of the e-media we have followed the structure-in-5 organizational architectural style presented notably in [14]. More information about alternative architectural selections can be found in [15]. According to the structure-in-5 style, the organization of the software architecture can be considered an aggregate of five sub-structures [16]. The *Operational Core*, which carries out the basic tasks and procedures directly linked to the production of products and services; the *Strategic Appex*, which makes executive decisions ensuring that the organization fulfills its mission in an effective way and defines the general strategy of the organization in its environment.

The *Middle Line*, which establishes a hierarchy of authority between the Strategic Appex and the Operational Core; the *Technostructure*, which serves the organization by making the work of others more effective, typically by standardizing work processes, outputs and skills; the *Support*, which provides specialized services, at various levels of the hierarchy, outside the basic operating workflow. These sub-structures are realized in the case of the e-media architecture by the *Store Front*, the *Back Store*, the *Billing Processor*, the *Coordinator* and the *Decision Maker*, as shown in Figure 3.

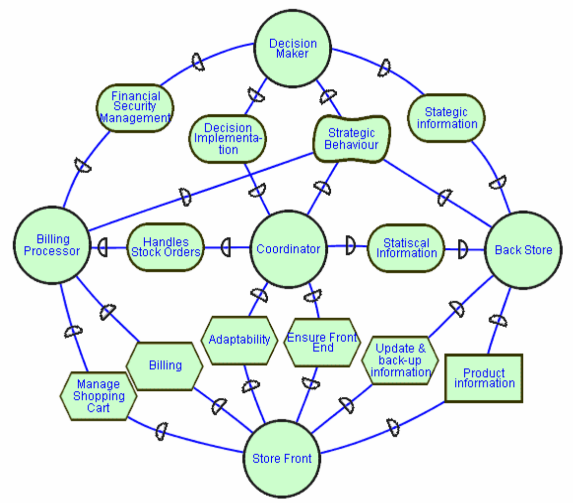


Figure 3: The E-Media Architecture in Structure-in-5

The *Store Front* interacts with customers and provides them with a usable front-end web application for consulting, searching and shopping media items. The *Back Store* constitutes the Support component. It manages the product database and communicates to the Store Front relevant product information. It stores and backs up all web information about customers, products and sales to be able to produce statistical information (e.g., analyses, average charts and turnover reports). Such kind of information is computed either for a predefined product (when the Coordinator asks it) or on a monthly basis for every product. Based on this monthly statistical information, it provides also the Decision Maker with strategic information (e.g., sales increase or decrease, performance charts, best sales, and sales prevision). The *Billing Processor* handles customer orders and bills. To this end, it provides the customer with on-line shopping cart capabilities.

It also handles, under the responsibility of the Coordinator component, stock orders to avoid shortages or congestions. Finally, it ensures the secure management of financial transactions for the Decision Maker. The *Coordinator* assumes the central position of the architecture. It is responsible to implements strategic decisions for the Decision Maker. It supervises and coordinates the activities of the Billing Processor (initiating the stock and pricing policy), the Front Store (adapting the front end interface with new promotions and recommendations) and the Back Store (parameterize statistical computing) ensuring that the system fulfills its mission in an effective way. Finally, the *Decision Maker* assumes strategic roles. It defines the Strategic Behavior (e.g., sales and turnover, product visibility, and hits) of the system ensuring that objectives and responsibilities delegated to the Billing Processor, Coordinator and Back Store are consistent with respect to their capabilities.

3.2 Secure Architectural Description

The architecture described in Figure 3 gives an organizational representation of the system-to-be including relevant actors and their respective goals, tasks and resource inter-dependencies. This model can serve as a basis to understand and discuss the assignment of system functionalities but it is not adequate to provide a precise specification of the system details. As introduced in Section 2, SKwyRL-ADL provides a finite set of formal agent-oriented constructors that allow detailing in a formal and consistent way the software architecture as well as its agent components and their behaviors.

Due to lack of space, we only provide a partial specification in SKwyRL-ADL of the Billing Processor agent. We illustrate some concepts detailed in Section 2 plus other ADL concepts introduced in Figure 1. For a complete SKwyRL-ADL specification of E–Media, we refer the reader to [8]. Five aspects of this agent component are of concern here: the interface representing the interactions in which the agent will participate, the knowledge base defining the agent knowledge capacity, the protection objectives indicating the desired security attributes of the agent, the security mechanisms representing a set of standard security methods that an agent might have and they help towards the satisfaction of the protection objectives of the agent, and the capabilities defining agent behaviors. The partial high-level formal description of the Billing Processor is as follows:

```
Agent:{Billing-Processor
    Interface
        Effector[provide(shopping_cart)]
        Effector[provide(billing)]
        Effector[provide(stock_orders)]
        Effector[provide(finance_security)]
        Sensor[require(strategic_behavior)]
        Sensor[require(statistical_info)]
    KnowledgeBase:
        Stock_KB                Pricing_Kb
        BP_Customer_KB          Providers_KB
        BP_System_KB            Statistical_KB
    Protection Objectives:
        Confidentiality_PO      Integrity_PO
        Availability_PO         Non_Repudiation_PO
        Authentication_PO       AccessControl_PO
    Security mechanisms:
        Encipherment_SM         DIgitalSignature_SM
        AccessControl_SM        DataIntegirty_SM
        AuthenticationExchange_SM
        TrafficPadding_SM       RoutingControl_SM
        Notarization_SM
    Capabilities:
        Shopping_Cart_Management_CP
        Billing_CP              Stock_Management_CP
        Statistic_CP
            }
```

The agent interface consists of a number of effectors and sensors for the agent. Each effector provides a service to other agents, and each sensor requires a service provided by another agent. An interaction is then defined by the correspondence between a required and a provided service. For example, the Billing Processor requires the statistical_info service that the Coordinator provides. The specification of the service description is presented below. Each provided or required service can be detailed by describing the sender agent that initiates the service, a set of receiver agents that interact with the sender, the reply-with that defines the information about which the service expresses an interaction, and optionally a set of parameters that define the information required to execute the service. The parameters as well as the reply-with information can be represented with a belief or a set of terms (e.g., function, constant or variable).

```
Service: {Ask(statistical_info)
            sender: Coordinator
            parameters: (tw: TimeWindows), (id: Id_product)
            reply_with:  to: Turnover ∨ sl: Sales
            receiver: Back-Store
Effect: Add(Statistical_KB, Achieve(statistic("today","on_product")
}
```

The Billing Processor agent has six KBs. Each of them is specified with a name, a KB_body and a KB_type. The specification of the Statistical_Kb is given below.

```
KnowledgeBase: {Statistical_KB
    KB_body:
        statistic_computation(Date,Subject)
        product_turnover(Id_Prod,TimeWindows,Turnover)
        customer_turnover(Id_Card,TimeWindows,Turnover)
        product_sales(Id_Prod,TimeWindows,Sales)
        extrapol_sales(Id_Prod,TimeWindows,setoff Sales)
    KB_type: closed_world }
```

The Billing Processor has six (6) protection objectives as shown in its description. These protection objectives have been identified by the security analysis that took place for the e-media system and partially presented in section 3.1. Each of the protection objectives is specified with a name, information of who imposed it to the agent, the agent to which it is imposed to, and the constraints that it imposes to the agent. For example, the specification of the Non_Repudiation is as follows:

```
Protection Objective: {
            name: Non_Repudiation_PO
            imposed_by: Environment
            imposed_to: Billing_Processor
            constraints: ConfirmInvolvementInTransactions
                    }
```

In addition, the Billing Processor has 8 different security mechanisms that represent a set of standard security methods that help towards the satisfaction of the protection objectives of the Billing Processor. Each security mechanism is specified with a name, the security methods it is composed of, a type, its availability to the agent, and an indication to which protection objective helps. The Notarization security mechanism specification for the Billing Processor agent is as follows:

```
Security Mechanism: {
        name: Notarization_SM
        composed_of: third_party_notary
        type: Combinational
        availability: Available
        help: Non_Repudiation
                }
```

A third-party notary that must be trusted by all participants provides notarization mechanisms. The notary can assure integrity, origin, time or destination of data. For example, a message that has to be submitted by a specific deadline may be required to bear a time stamp from a trusted time service proving the time of submission.

The Billing Processor agent has also some capabilities. A capability is composed of plans and events that together serve to give an agent certain abilities. For example, the Billing Processor Statistic_CP capability is defined as follows. The body contains the plans that the capability can execute and the events it can post to be handled by other plans or it can send to other agents.

```
Capability:{Statistic_CP
    CP_body:
        Plan Prov_Turnover_On_Demand
        Plan Prov_Turnover
        Plan Sales_Average
        Plan Stock_Orders
        SendEvent Grade
        SendEvent Best_Sales
        SendEvent Promotion
}
```

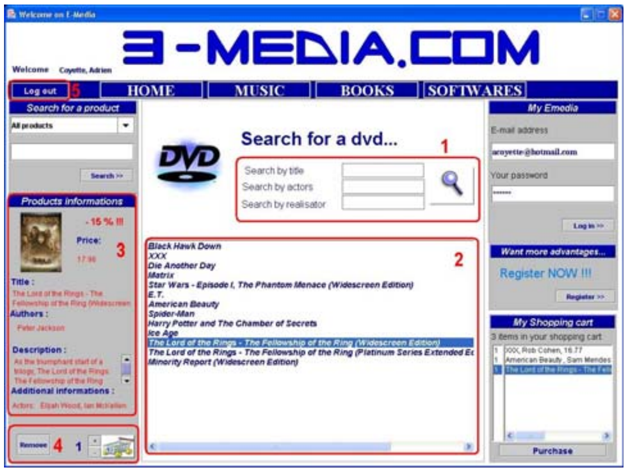
The Stock_Order plan of the Billing-Processor will make sure that the level of stock of each product is permanently higher than the minimal quantity, which is determined by the coordinator on the basis of the strategic orientation provided by the Decision-Maker. In the plan body, the quantity to order is determined and then the order is sent to the publisher. Eventually, the level of stock is updated in the system. In case of plan failure, the 'fail" instructions are carried out. So the billing-Processor searches for the last order sent for this product and reorder the same quantity. Then the stock level is updated with the quantity ordered.

```
Plan:{
Name: Stock_Orders
   invoc:
            Maintain(current_stock(id,Availability > lb)
              // with id: Id_Product
              // From Coordinator.Ask(stock_orders).reply_with
              // with lb: Lower_Bound
              // From Coordinator.Ask(stock_orders).reply_with
   context:
             current_stock(id,Availability < lb)
                ∧ ¬ time (now > "11 am")
                ∧ (day(now ="monday"
                ∨ day(now ="wednesday")
   body:
       action: proceed_order(id, lb)
       effect: Add(Stock_Kb, Sent_Orders(id,qu,date))
   endstate:
             Add(Stock_Kb, Sent_Orders(id,qu,date))
   succeed:
             action: update_stock(id, av)
             //with av: availability
             effect: Add(Stock_Kb, Stock(id, av))
   fail:
             action: search_last(sent_orders(),id)  as  qu: Quantity
                     Add(Stock_Kb, Sent_Orders(id,qu,date))
                     update_stock(id, av)
             effect: Add(Stock_Kb, Stock(id, av))
   }
```

4. E-Media Implementation

The E-Media application has been implemented (~ 10.000 lines of code) with JACK [17], a BDI agent-oriented development environment for JAVA. The implementation was based on the structure-in-5 architecture described in Section 3.1 and the formal SKwyRL-ADL specification overviewed in Section 3.2, We briefly describe the E-Media implementation to illustrate the role of the agents and their interactions as well as presenting some implementation of the secure architectural considerations for the payment information.

When an on-line customer gets connected to E-media, an instance of the Front-Store is created to display an interface that allows the new coming user to register. Then, the Back-Store handles the information provided by the user and checks its validity. If the access is granted, the user can purchase products on E-Media by adding catalogue items to the shopping cart managed by the Billing-Processor. At any time the user can use a navigation-bar to switch from one section of the website to another. Moreover, promotions and best sales are part of the strategic behaviour objective. The promotions policy is initiated by the Decision-Maker based on the strategic information provided by the Back-Store. The Coordinator chooses the best promotions and consequently adapts the Store Front layout. The Coordinator acts similarly for the best sales: the Back-Store computes the five best sellers and the Coordinator accordingly updates the Store-Front. Figure 4 describes the Store-Front interface for the DVD section.



Black Hawk Down
XXX
Die Another Day
Matrix
Star Wars - Episode I, The Phantom Menace (Widescreen Edition)
E.T.
American Beauty
Spider-Man
Harry Potter and The Chamber of Secrets
Ice Age
The Lord of the Rings - The Fellowship of the Ring (Widescreen Edition)
The Lord of the Rings - The Fellowship of the Ring (Platinum Series Extended Ed
Minority Report (Widescreen Edition)

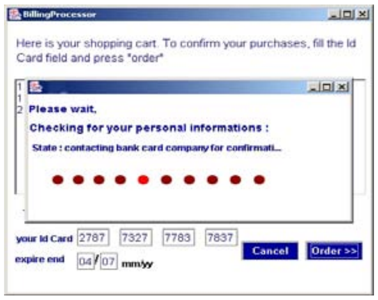
1 XXX, Rob Cohen, 16.77
1 American Beauty, Sam Mendes
1 The Lord of the Rings - The Fell

Figure 4: Interface of e-media DVD section

To search the E-Media DVD catalogue, the user must fill at least one field of the search engine (1). The Store-Front sends the query parameters to the Back Store which provides the results back to the Store-Front (2).

At any moment during the session, the user can click on a product (best seller, query result, and shopping cart); a request is then sent to Back Store to provide more information on this product (3).

When the user starts the billing process, the Billing-Processor displays all the items of the shopping cart and computes the total and sub-total for each product. Next, it checks the validity of the user Id-Card number, either by verifying its database, or by asking confirmation to the Bank Company (Figure 5), or both. Once the payment is accepted, the Billing-Processor informs the Store-Front. A confirmation message is displayed and the shopping cart is cleared.



Here is your shopping cart. To confirm your purchases, fill the Id Card field and press "order"

Please wait,

Checking for your personal informations :

State : contacting bank card company for confirmati...

your Id Card 2787 7327 7783 7837

expire end 04 / 07 mm/yy

Figure 5: Secure Payment Information.

5. Conclusions

Nowadays, software engineering for new enterprise application domains such as e-Business is forced to build up open but secure systems able to cope with distributed, heterogeneous, and dynamic information issues. Most of these software systems exist in a changing organizational and operational environment where new components can be added, modified or removed at any time. For these reasons and more, agent architectures are gaining popularity in that they do allow dynamic and evolving structures which can change at run-time.

Architectural design has received considerable attention for the past decade which has resulted in a collection of formal architectural description languages. Unfortunately, this work has focused on object-oriented rather than agent-oriented systems. This paper has defined a set of system secure architectural concepts to propose such a language for BDI-MAS. This ADL allows formalizing each agent component, behavior and interaction in terms of secure architectural specifications.

The paper has proposed a validation of the framework: it has been applied to develop E-Media, an e-commerce platform implemented on the JACK agent development environment.

The research reported here calls for further work. We are currently working on: (1) The development of a CASE tool to automatically generate code for the future multi-agent system from Secure SKwyRL-ADL specifications; (2) the definition of a set of rules to perform security and consistency analysis to be included in verification tools such as PVS; and (3) the identification of a suitable set of core abstractions, inspired by organizational metaphors, to be used during the design of the secure multi-agent system architecture.

6. REFERENCES

[1] M. Barley, F. Massacci, H. Mouratidis, P. Scerri (eds). Proceedings of the 1st International Workshop on Safety and Security in Multiagent Systems, Third International Joint Conference on Autonomous Agents and Multiagent Systems, N.Y. –USA, 2004

[2] E. Fernandez-Medina, J. Cesar Hernandez Castro, L. Javier Carcia Villalba (eds). The Second International Workshop on Security in Information Systems, 6th International Conference on Enterprise Information Systems, 2004.

[3] H. Mouratidis. A Security Oriented Approach in the Development of Multiagent Systems: Applied to the Management of the Health and Social Care Needs of Older People in England. PhD thesis, University of Sheffield, U.K., 2004

[4] J. Jurjens. Secure Systems Development with UML, Springer Verlag, 2004

[5] M. Shaw, R. DeLine, D. V. Klein, T. L. Ross, D. M. Young and G. Zelesnik, Abstractions for Software Architecture and Tools to Support Them, IEEE Transactions on Software Engineering, 21(4):314-335, 1995.

[6] P. C. Clements. A Survey of Architecture Description Languages. In Proc. of the Eighth International Workshop on Software Specification and Design, Paderborn, Germany, March 1996.

[7] M. Shaw and D. Garlan, Software Architecture: Perspectives on an Emerging Discipline, Prentice Hall, 1996.

[8] S. Faulkner, An Architectural Framework for Describing BDI Multi-Agent Information Systems, Ph.D. thesis, Department of Management Science, University of Louvain, Belgium, May 2004.

[9] R. Anderson. Security Engineering: A Guide to Building Dependable Distributed Systems. John Willey & Sons, New York, 2001.

[10] B. Schneier. Secrets & Lies: Digital Security in a Networked World, John Willey & Sons, 2000

[11] J. Viega, G. McGraw. Building a Secure Software. Addison-Wesley, Reading MA, 2002

[12] J. M. Spivey, The Z Notation: A Reference Manual. Prentice-Hall, second edition, 1992.

[13] E. Yu, "Modeling Strategic Relationships for Process Reengineering," Ph.D. thesis, Department of Computer Science, University of Toronto, Canada, 1995.

[14] M. Kolp, P. Giorgini, and J. Mylopoulos. An Organizational Perspective on Multi-agent Architectures. In Proc. of the 8th Int. Workshop on Agent Theories, architectures, and languages, ATAL'01, Seattle, USA, Aug. 2001.

[15] T. T. Do, S. Faulkner and M. Kolp. Organizational Multi-Agent Architectures for Information Systems. in Proc. of the 5th Int. Conf. on Enterprise Information Systems (ICEIS 2003), Angers, France, April 2003.

[16] H. Mintzberg. Structure in fives: designing effective organizations. Prentice-Hall, 1992.

[17] JACK Intelligent Agents. http://www.agent-software.com/