Author(s): Mouratidis, Haralambos; Kolp, Manuel; Giorgini, Paolo; Faulkner, Stephane
Title: An Architectural Description Language for Secure Multi-Agent Systems
Year of publication: 2010
Citation: Mouratidis, H., Kolp, M., Giorgini, P., Faulkner, S. (2010) 'An Architectural Description Language for Secure Multi-Agent Systems' *Web Intelligence and Agent Systems* 8 (1) pp.99-122
Link to published version: http://dx.doi.org/10.3233/WIA-2010-0182
DOI: 10.3233/WIA-2010-0182

An Architectural Description Language for Secure Multi-Agent Systems

Haralambos Mouratidis[1,a], Manuel Kolp[b], Paolo Giorgini[c], Stephane Faulkner[d]

[a] *Innovative Informatics Group, School of Computing, IT and Engineering, University of East London, England*
[b] *Information Systems Research Unit, Catholic University of Louvain (UCL), Belgium*
[c] *Department. of Information and Communication Technology, University of Trento, Italy*
[d] *Information Management Research Unit, University of Namur, Belgium*

Abstract. Multi-Agent Systems (MAS) architectures are gaining popularity for building open, distributed, and evolving information systems. Unfortunately, despite considerable work in the fields of software architecture and MAS during the last decade, few research efforts have aimed at defining languages for designing and formalising secure agent architectures. This paper proposes a novel Architectural Description Language (ADL) for describing Belief-Desire-Intention (BDI) secure MAS. We specify each element of our ADL using the Z specification language and we employ two example case studies: one to assist us in the description of the proposed language and help readers of the article to better understand the fundamentals of the language; and one to demonstrate its applicability.

Keyword: Architectural Description Language, Multi-Agent Systems, Security, BDI Agent Model, Software Architecture

[1] Corresponding Author: H.Mouratidis@uel.ac.uk

1. Introduction

The characteristics and expectations of new application areas for the enterprise, such as e-business, knowledge management, peer-to-peer computing, and web services, are deeply modifying information systems engineering. Most of the systems designed, for this kind of application areas, are now *de facto* concurrent and distributed. They tend to be open and dynamic, in that they exist in a changing organizational and operational environment where new components can be added, modified, or removed at any time. It is also important that such systems are secure, since they are often used for the management and storage of sensitive information, such as medical, financial and private data.

Given these needs, many researchers are looking for paradigms that will enable them to conceptualize, design and implement information systems that can operate effectively in such circumstances. In this context, we advocate the use of Multi-Agent Systems (MAS) to build today's enterprise information systems. MAS architectures appear to be more flexible, modular and robust than traditional; including object-oriented ones. MAS do represent dynamic and evolving structures and components, which can change at run-time to benefit from the capabilities of new system entities or replace obsolete ones. Moreover, MAS represent a suitable paradigm for the consideration of security challenges introduced by the current information systems. 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 [34].

However, as the expectations of business stakeholders are changing day after day; and as the complexity of systems, information and communication technologies and organisations is continually increasing in today's dynamic environments; developers are expected to produce architectures that must handle more difficult and intricate requirements than ever before.

A critical issue in the design and construction of any complex information system is its architecture. That is, its gross organization as a collection of interacting components. A rigorous architectural design can ensure that a system will satisfy key requirements in such areas as performance, reliability, portability, scalability, and interoperability [40]. To assist developers in specifying information system architectures, architectural description languages are used. An Architectural Description Language (ADL) provides a concrete syntax for specifying architectural abstractions in a descriptive notation. Architectural abstractions concern the structure of the system's components, their behaviour, and their interrelationships.

Unfortunately, despite the progress on the field of software architectures (see for instance [2], [17], [28], [30], [39]) and the simultaneous progress on MAS research; few research efforts have aimed at truly defining description languages for MAS architectures, and even these do not adequately include important issues of MAS such as security. This paper deals with this problem and it proposes a novel ADL for MAS, which defines a "core" set of structural, behavioural and security concepts, including relationships and constraints, which are fundamental for a complete MAS ADL. The language, called

SKwyRL-ADL[2], aims to describe secure Belief-Desire-Intention (BDI) MAS.

The paper is structured as follows. Section 2 overviews the notions of agent and MASs and identifies the main concepts of the BDI model, which is used in our ADL. Section 3 discusses security in MAS and Section 4 models SKwyRL-ADL using the Z specification language. An example case study is used to assist reader understanding of the language's elements. Section 5 demonstrates the applicability of the proposed ADL with the aid of a real case study from the e-media domain. Section 6 describes related work; it indicates how that work has influenced the presented work, and it discusses how the presented work differs from related work. The last section summarizes the contributions of the paper and discusses future work.

2. Agents and Multi-Agent Systems

An *agent* defines a system entity, situated in some environment that is capable of flexible autonomous action in order to meet its design objectives [46]. Three key concepts support this definition:

- *Situatedness*: an agent receives input from the environment it operates and can perform actions, which change the environment in some way;
- *Autonomy*: an agent is able to operate without direct, continuous supervision. In other words it has full control over its own actions;
- *Flexibility*: an agent is not only reactive but also pro-active. *Reactivity* means that an agent has perceptions of the world, which force the agent to act and react to change in quasi real-time fashion. *Pro-activeness* means that an agent's behaviour is not exclusively reactive but it is also driven by internal goals, i.e. an agent may take initiative.

With these concepts in mind, a MAS can be defined as a set of autonomous and proactive agents that interact with each other to achieve common or private goals. This definition leads us to two different types of MAS: *Cooperative* or *Competitive*.

A *Cooperative* MAS has a unique high-level global goal (or set of goals) decomposed recursively into parallel activities to be performed by the set of agents that compose that MAS. This kind of system is typically adapted to perform distributed problem solving. In a *Competitive* MAS, each of the component agents has its own set of goals that may or may not meet those of other agents. In this case the MAS is an architecture that allows agents to interact, while each one pursues personal goals and defends own interests. This kind of system meets typical engineering requirements of e-commerce, information retrieval applications, web services or peer-to-peer networks. In such environments, every agent generally represents either a *client*, aiming at obtaining some resources or have some service accomplished; or a *provider*, aiming at selling resources or services at a certain (not necessarily financial) cost. Each agent pursues the goals of the (human or system) actor it represents, and these goals can usually be in conflict.

In order to reason about these goals and act in an autonomous way, agents are usually built on rationale models and reasoning strategies that have roots in various disciplines including Artificial Intelligence, Cognitive Science, Psychology

[2] Socio-Intentional ArChitecture for Knowledge Systems & Requirements ELicitation (www.isys.ucl.ac.be/skwyrl)

or Philosophy. Agent models are proliferating; some include learning capabilities, others intelligent agendas based on statistics, others yet are based on genetic algorithms and so on. An exhaustive evaluation of these models is out of the scope of this paper or even this research work. However, a simple yet powerful and mature model coming from Cognitive Science and Philosophy that has received a great deal of attention, notably in Artificial Intelligence, is the Belief-Desire-Intention (BDI) model [6]. This approach has been intensively used to study the design rationale of agents and is proposed as a keystone model in numerous agent-oriented development environments such as JACK[3] or JADE[4]. The main concepts of the BDI agent model are (except the notion of agent itself that we have just explained):

- *Beliefs* that represent the informational state of a BDI agent, that is, what the agent knows about itself and the world;
- *Desires* (or goals) that are its motivational state, that is, what the agent is trying to achieve;
- *Intentions* that represent the deliberative state of the agent, that is, what plans the agent has chosen for possible execution.

In particular, a BDI agent has a set of *plans*, which defines sequences of *actions* and steps available to achieve a certain *goal* or react to a specific situation. The agent reacts to *events*, which are generated by modifications to its beliefs, additions of new goals, or messages arriving from the environment or from another agent. An event may trigger one or more plans; the agent commits to execute one of them, that is, it becomes its intention. Plans are executed one step at a time. A step can query or change the beliefs; it can perform actions on the external world; and it can submit new goals. The operations performed by a step may generate new events that, in turn, may start new plans. A plan succeeds when all its steps have been completed; it fails when certain conditions are not met.

[3] http://www.agent-software.com.au/jack.html

[4] http://jade.tilab.com

3. MAS and Security

Security of software systems, agent-oriented, object-oriented or otherwise, is concerned with methods providing cost effective and operationally effective protection from undesirable events[30]. In principle security is usually defined in terms of the existence of any of the following properties:

- *Confidentiality*: The property of guaranteeing information is only accessible to authorized entities and inaccessible to others;
- *Authentication*: the property of proving the identity of an entity;
- *Integrity*: the property of assuring that the information remains unmodified from source entity to destination entity;
- *Access Control*: the property of identifying the access rights an entity has over system resources;
- *Non-repudation*: the property of confirming the involvement of an entity in certain communication;
- *Availability*: the property of guaranteeing the accessibility and usability of information and resources to authorized entities.

In MAS, each of these properties is associated with the characteristics of agents and need to be considered during the development. For instance, regarding *situatedness* [46], the authentication,

confidentiality and availability of an agent needs to be considered according to the environment in which the agent operates. On the other hand, the social behaviour property of an agent involves communication with other agents and other entities and as such the properties of non-repudiation, integrity and access control are important. Therefore, it is crucial for the agents of a MAS to consider, during run-time, the systems' and their individual security requirements when specifying their objectives and interactions. For this to happen, agent developers must integrate security considerations when they define the architecture of their MAS [4]. However, research efforts so far have been mainly focused on the solution of individual security problems of MAS, such as attacks from an agent to another agent, attacks from a platform to an agent, and attacks from an agent to a platform. Developers of MAS ADLs have mainly neglected security and have failed to provide evidence of successfully integrating security concepts as part of their ADLs. As a result, MAS developers find no help when considering security during the architectural design of a MAS.

4. SKwyRL: An Architectural Language for Secure MAS

4.1. ADL Concepts

Architectural description languages are formal languages that are used to specify the architecture of a system [40]. By *architecture*, we mean the components that compose a system, the behavioural specification for those components, and the mechanisms for interactions among them. Based on our analysis of existing literature (see Section 6 for a discussion of related work), we have identified the following concepts as the common foundation of concepts and concerns for system architecture descriptions:

Component. *Components are units of computation and data store. Therefore, components are loci of computation and state*. A component, in architecture, may be as small as a single procedure (e.g., Wright [1] procedures) or as large as an entire application (e.g., hierarchical components in Rapide [28]). It may require its own data and/or execution space, or it may share them with other components.

Interface. *Interfaces are set of interaction points among components and the external world.* All ADLs support specification of component interfaces. They differ in terminology and the kinds of information they specify. For example, each interface point in ACME [17]and Wright is called a *port*. In UniCon [39], an interface point is a *player*, and in Rapide a *constituent*. In Darwin [29], the interface consists of a collection of services that are either *provided* or *required*.

Type. *A type describes how architectural element representation is built up*. ADLs can support reuse by modelling abstract components as types and instantiating them in an architectural configuration. All the surveyed ADLs distinguish component types from instances. However, UniCon and Darwin lack explicit means of introducing new component types, as they support only a predefined set. Other ADLs such as Rapide and Wright make explicit use of parameterisation.

Connector. *Connectors are used to model interactions among components and the rules that govern those interactions*. Some ADLs that model connectors as first-class entities are called explicit configuration

languages, as opposed to in-line configuration languages. The first-class includes languages such as Wright. On the other hand, Rapide and Darwin are examples of in-line configuration languages. In these languages, connectors cannot be named. They are described solely in terms of bindings between the provided service of a component and the required service of another component.

Configuration. *Configurations (or topologies) are connected graphs of components and connectors that describe architectural structures*. Configurations are needed to determine whether appropriate components are connected, their interfaces match and their combined semantics result in desired behaviour. Descriptions of configurations enable the assessment of concurrent and distributed aspects of architecture, e.g., potential for deadlocks and starvation, performance, reliability, security, etc. Configurations also enable analysis for adherence to design heuristics and style constraints. In this sense, a major role of configuration is to facilitate the understanding of systems at a high level of abstraction. Therefore, configurations must model structural information with simple and understandable syntax.

Hierarchical Composition. *A* h*ierarchical composition allows the representation of an entire architecture as a single component in another, larger architecture*. Several ADLs provide explicit features to support hierarchical composition. For example ACME provides templates, Darwin composition elements, and UniCon and Wright maps.

4.2. The GOSIS example

To assist understanding the specification of the language components, we represent extracts of the a**G**ent-**O**riented **S**ource **I**ntegration **S**ystem (**GOSIS**) architecture to compliment our theoretical description of the language. GOSIS provides a MAS architecture that supports the integration of data coming from dynamic, distributed and heterogeneous sources. GOSIS is a hybrid approach that combines the advantages of in-advance and on-demand processes[15]. In such an approach, the user information needs can be extracted in-advance or on-demand by the *mediator*. The information extracted in-advance is stored in a central database managed by the mediator. The information contents of this central database can be seen as a materialized view where the database resides at the information sources. In this way, in comparison with a basic mediator, a hybrid mediator adds functionalities essentially in order to perform materialized view maintenance.

In particular, when a user wishes to send a request, it contacts the broker agent, which serves as an intermediary to select one or more mediator(s) that can satisfy the user information needs. Then, the selected mediator(s) firstly decomposes the user's query into one or more sub-queries regarding the appropriate information sources and then it eventually compiles and synthesizes results from the source and returns the final result to the broker. When the mediator identifies repetitively the same user information needs, this information of interest is extracted from each source, merged with relevant information from the other sources, and stored as knowledge by the mediator. Each of the stored knowledge bases constitutes a materialized view that the mediator has to keep up-to-date. Moreover, two types of agents, a *wrapper* and a *monitor*, are connected to each information source. The wrapper is used to translate the sub-query issued by the

mediator in the native format of the source and also to translate the source response in the data model used by the mediator. The monitor is responsible for detecting changes of interest (e.g., a change which affects a materialized view) in the information source and for reporting them to the mediator. Changes are then translated by the wrapper and sent to the mediator.

It may also be necessary for the mediator to obtain information concerning the localisation of a source, and for the associated wrapper to provide current or future relevant information. This kind of information is provided by the matchmaker agent, which allows a direct interaction between the mediator and the correspondent wrapper. The matchmaker plays the role of a "yellow-page" agent. Each wrapper advertises its capabilities by subscribing to the yellow page agent. Finally, the multi-criteria analyzer reformulates a sub-query (sent by a mediator to a wrapper) through a set of criteria in order to express the user preferences in a more detailed way, and refines the possible domain of results.

4.3. The ADL concepts

Following the identification of the common foundation of components necessary for an ADL; an architecture based on the proposed SKwyRL-Architectural Description Language (ADL) includes the following concepts:

- ***Component.*** In SKwyRL-ADL, we consider an agent as a system component with a set of plans determining its computation dimension and a set of beliefs defining its data space.
- ***Interface.*** SKwyRL-ADL specifies an interface point in the same way as ACME and Wright, with the difference that a port is either a *sensor* requiring a service or an *effector* providing a service for the agent environment. In this sense, like Darwin, an interface can be seen as a collection of provided or required services.
- ***Type.*** We introduce parameterisation in order to distinguish the different instances of each agent type that can appear in a configuration or to expand an agent description from a single system to families of systems. SKwyRL-ADL permits any part of an agent description to be replaced with a "placeholder", which is then filled with a parameter when the type is instantiated.
- ***Connector.*** SKwyRL-ADL follows the Rapide and Darwin notion of connectors according to the definition of agent interaction that we will discuss in the following section.
- ***Configuration.*** Because Wright offers the most complete and formal definition of configuration specification, we follow it with the aim of describing a complete system architecture. Like a Wright configuration description, the components (i.e., agent) and connectors (i.e., bindings between provided and required services) must be combined into a configuration. In this sense, in SKwyRL-ADL, a configuration is a collection of agent instances combined via bindings between provided and required services.
- ***Hierarchical Composition.*** SKwyRL-ADL allows for hierarchical composition, but provides no specific constructs to support it. In particular, SKwyRL-ADL permits replacing basic components in an architecture by a (sub) configuration so as to form a new

configuration. This is done by supporting the specification of an agent by one or more detailed lower-level descriptions.

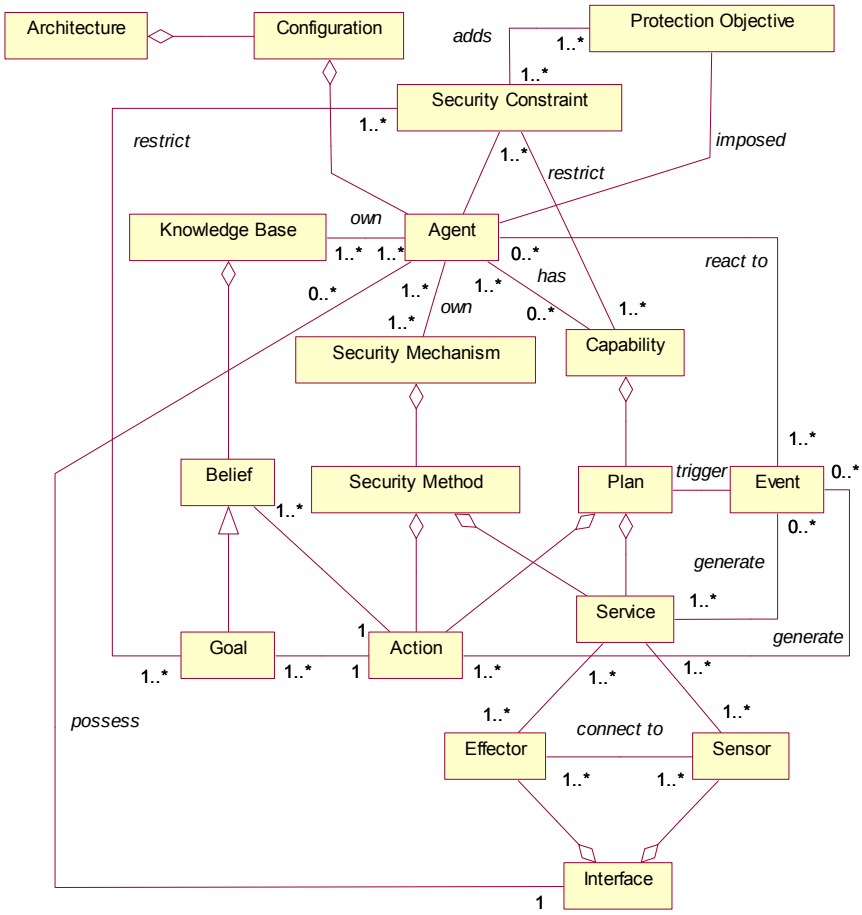

Fig. 1: Conceptualization of SKwyRL-ADL.

4.4. Metamodel

Figure 1 introduces the main entities and relationships of the elements of SKwyRL-ADL. Each entity has been identified from the generic features of current ADLs, from architectural security considerations, and from the concepts defined through the theoretical BDI architecture model.

For clarity we have further subdivided the model into three sub-models: the agent model; the security model; and the architectural model. The three following sub-sections describe these models in greater detail. Each entity is formalized using the Z specification language [41]. We have adopted Z in order to formalize the concepts and relationships of our ADL model for a number of reasons. Firstly, Z provides modularity and abstraction and is sufficiently expressive to allow for a consistent, unified and structured account of

a software system and its associated operations. Such structured specifications enable the description of MAS architectures at different abstraction levels, with system complexity being added at successive lower levels. Secondly, Z is particularly suitable in squaring the demands of formal modelling with the need for implementation by allowing for transitions between specification and software. Our approach to formalize the specification of SKwyRL is thus pragmatic: we need to be formal to be precise about the concepts we discuss, yet we want to remain directly connected to implementation issues.

Furthermore, Z is widely used as formal specification language within the software engineering and the artificial intelligence communities. Z has been shown to be clear, concise and relatively easy to learn compared with other languages [27].

4.4.1. Agent Model

The agent model, illustrated in Figure 1, is composed of eight main design entities. An agent needs knowledge about its environment in order to make good decisions. Knowledge is contained in an agent in the form of one or many *knowledge bases* structuring its informational state. A knowledge base consists of a set of *beliefs* that the agent has about its environment. A belief represents a view of the current environment of an agent.

However, beliefs about the current state of the environment are not always enough to decide what to do. In other words, in addition to a current state description, the agent needs *goal* information. A goal describes an environment state that is (or is not) desirable. An agent pursues one or more goals that represent its motivational state.

The intentional behaviour of an agent is represented by its *capabilities* to react to *events*. A capability is a set of events that an agent can handle, post or send to its environment and a set of plans. An event is generated either by an *action* that modifies beliefs or adds new goals, or by *services* provided by another agent. Note that services also appear in the structural model because they involve interactions among agents that compose the MAS.

Interactions serve as basic elements to support the construction of configurations. An event may invoke (trigger) one or more *plans*; the agent is committed to executing one of them, that is, it becomes its intention. A plan defines the sequence of actions or services to be chosen by the agent to accomplish a task or fulfil a goal. An action can query, add or remove beliefs, generate new events or submit new goals.

As an example, consider the model shown in Figure 2 representing a partial specification of the mediator agent of the GOSIS case study. The Mediator agent has a set of interfaces, Knowledge Bases (KB), Protection Objectives (PO), Security Mechanisms (SM) and a set of Capabilities (CP).

This formalization is intended as an intuitive aid to introduce the fundamental design entities and relationships, and assist in the comprehension of the ADL. However, the description level chosen here does not specify the details of the beliefs composing the KB; the plans and events composing each CP; or the security methods composing each SM. For this reason, the proposed ADL supports refinement specification, using the Z language, for each of the main aspects of the agent: interface, knowledge base, security mechanisms and capabilities.

Agent: { Mediator
Interface:
Sensor[require(*query_translation*)]
Sensor[require(query r*eformulation*)]
Sensor[require(*results*)]
Sensor[require(*locate_wrapper*)]
Sensor[require(*change_advertizings*)]
Effector[provide(*found_items*)]
KnowledgeBase:
Results_KB
MatchMaker_Info_KB
DataManagement_KB
Request_KB
Notification_KB
Protection Objectives:
Confidentiality_PO
Availability_PO
Security mechanisms:
DataIntegirty_SM
AuthenticationExchange_SM
Capabilities:
Handle_Request_CP
Handle_Results_CP
Materialized_Views_CP
Wrapper_Localization_CP
Handle_Change_CP
}

Fig. 2: Partial specification of the GOSIS mediator agent.

Knowledge base. *A knowledge base is a set of beliefs that the agent has about the environment and itself.* A knowledge base (KB) is a means of structuring the informational state of agents and it encapsulates a set of states describing a specific part of the current environment of an agent. Each individual state is called a belief. A knowledge base specification is also described by the type of KB (*KBtype*) and a name that assists to identify the KB. Whenever an agent wants to query or modify a KB, it does so by using this name. The set of all names is denoted by [*KBname*] and a KB can be specified as in Figure 3.

The *KBtype* describes the kinds of formal knowledge used by agents that compose the MAS. *closedWorld* states that an agent knows only the beliefs included in its knowledge base [25], and anything not in its knowledge base simply doesn't exist. Inversely, *openWorld* states that an agent accepts every belief that it "considers" possible [21]. Although closed-world states do not often occur in the real world, they are useful in many simulation and programming environments [12].

[*KBname*]
[*KBtype*]:= closedWorld | openWorld

KnowledgeBase

name: *KBname*
composed_of: $\mathbb{P}$ *Belief*
type: *KBtype*

name $\neq\emptyset\wedge$composed_of $\neq\emptyset\wedge$type $\neq\emptyset$
$(\forall$kb:*KnowledgeBase*$)$ $(\exists$ ag : *Agent* $)$ $\bullet$*use*(kb,ag)

Fig. 3: KnowledgeBase specification.

An example of a KB specification for the Mediator agent of the GOSIS system is shown in Figure 4. In particular, three KBs are specified in the agent specification presented above: the Matchmaker_Info, WrapperSubscription and Translation_Management. The contents of these KBs concern, respectively, wrapper localisations and translation abilities; accepted and refused wrapper subscriptions; mediator queries and the specific information needed to translate it. This specific information is represented by the following beliefs:

- source_resource that defines the kind of data available from the connected source;
- source_modeling that describes how the information is structured;
- dictionary that provides the term correspondence between the mediator and the source.

KnowledgeBase: {
name: *MatchMaker_Info_KB*
composed-of:
wrapper(WrapperLocalization,TranslationService(+))
type: *closed_world* }

KnowledgeBase: {
name:*WrapperSubscription_KB*
composed-of:
refusal_subscription(Id,TranslationType, WrapperLocalization,Reason)
accepted_subscribe(Id,TranslationType, WrapperLocalization,Date)
type: *closed_world* }

KnowledgeBase: {
name: *Translation_Management_KB*
composed-of:
search(RequestType,ProductType,FilteredKeyword(+))
source_resource(InfoType(+))
source_modeling(SourceType,Relation(+),Attributes(+))
dictionary(*MediatorTerm,SourceType,Correspondence*)
type: closed_world}

Fig. 4: KB specification for the GOSIS mediator agent.

Belief. *A belief is a predicate describing a set of states about the current agent environment being either true or false.* Beliefs describe the environment of an 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. Constant symbols are therefore terms. But sometimes it is more convenient to use an expression to refer to an object. This is what function symbols are for. Thus a complex term can be formed by a function symbol followed by a parenthesized list of terms as arguments to the function symbol.

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. An *AtomicBelief* is formed from a predicate symbol followed by a sequence of terms (Figure 5).

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

AtomicBelief

head: *PredSymb* terms: seq *Term*
head ≠∅∧terms ≠∅

Fig. 5: AtomicBelief Specification.

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. We use the following temporal patterns: ○ (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*
| *Temp_Pattern AtomicBelief*
| AtomicBelief Connective AtomicBelief

[*Connective*] →∧ | ∨ | ⇒

[*Temporal_Pattern*]:=○ | • | ◊ | ♦ | □ | ■ |$\mathcal{W}$|$\mathcal{U}$

Goal. *A goal describes an environment state that an agent wants to bring about.*Beliefs about the current state of the environment are not always enough to decide what to do. In other words, in addition to a current state description, the agent needs goal information, which

describes situations that are (not) desirable. Goal information is an operational objective to be achieved by an agent. Operational means that the objective can be formulated in terms of appropriate state transitions under the control of one agent. We consider goals according to four patterns [10]:

Achieve: P $\Rightarrow$ $\Diamond$Q
P means "*state P holds in the current state*"
$\Diamond$Q means "*state Q holds in the current or in some future state*"

Cease: P $\Rightarrow$ $\Diamond\neg$Q

Maintain: P $\Rightarrow$ $\Box$Q
$\Box$Q means "state Q holds in the current and in all future states"

Avoid: P $\Rightarrow\Box\neg$Q

With respect to beliefs, goals can be specified as in Figure 6.

[*GoalPattern*] := Achieve | Cease | Maintain | Avoid
[*GoalStatus*]:= Fulfilled | Unfulfilled

Goal

head: *GoalPattern*
state: *Belief*
Status: *GoalStatus*

---

head $\neq\emptyset\wedge$state $\neq\emptyset$

$(\forall$ g: *Goal*$) \wedge$ g.status = Fulfilled
$\Rightarrow (\exists$ blset = {$bl_1,\ldots,bl_n$: *Belief*$\} \wedge$ g.state $\subseteq$ blset)

Fig. 6: Goal specification.

The state explicitly describes (in terms of beliefs) the environment in which the goal is fulfilled. The status indicates whether the goal has been fulfilled or not. The goal patterns influence the set of possible agent behaviours: achieve and cease goals generate actions, plans, or events; 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.

Capability. *A capability is a set of plans that an agent can execute and a set of events that it can post to itself or send to its environment.* The capability is a means of structuring the intentional behaviour of agents. In a perspective of modularity, it allows to encapsulate a set of agent functionalities that can be plugged in as required. This component approach allows a system architect to build up a library. These components can then be (re)used to add selected functionality to different agents of a system. Also, some of these components can be temporally not available for the agents in the system. Capabilities are structured from a set of events, plans or sub-capabilities that can be combined to provide complex functionality. A Capability specification takes the form in Figure 7.

[*CapName*]
[*AtomicCap*] := Plan | Capability | Event
[*CapAvaibility*]:= Available | Unavailable

Capability

name: *CapName*
composed_of: $\mathbb{P}$ *AtomicCap*
availability: *CapAvailability*

---

name $\neq\emptyset\wedge$composed_of $\neq\emptyset$

$(\forall$ cap: *Capability*$) \exists$ ag: *Agent* $\wedge$ cap $\in$ ag.has $\Rightarrow$ cap.availability = "available"

Fig. 7: Capability specification

Referring back to the GOSIS case study, the mediator specification holds five capabilities:

- Handle_Request decomposes a user query into one or more sub-queries and sends them to adequate wrappers;
- Handle_Results synthesises the source answers and returns the answers to the broker;
- Materialized_Views manages the storage, the updates, the queries and the results related to a set of materialized views;
- Wrapper_Localization manages the information (provided by the matchmaker) concerning the localisation of a source and its connected wrapper;
- Handle_Change executes the materialised view updates when a monitor detects changes from its source.

The proposed ADL allows the specification of each capability with a name and a body (composed-of) containing the plans that the capability can execute and the events that it can post to itself (handled by one of these plans) or send to other agents. For example, the Handle_Request is specified as in Figure 8.

Capability: {
name: *Handle_Request_CP*
composed-of:
Plan: *DecompNmlRq*
Plan: *DecompMCRq*
SendEvent: *FaillUserRq*
SendEvent: *FailDecompMCRq*
PostEvent: *ReadyToHandleRst*
availability: available}

Fig. 8: Handle_Request capability specification

The concept of *Event* and *Plan* are specified later in this section.

Action. *An action is an internal operation executed in order to achieve goals or accomplish tasks*. An operation is a basic executable command of agent behaviour. The type of operation that agents can perform may be classified as either external (the domain of the operation is the environment outside the agent) or internal (the domain of the operation is the agent itself). Actions concern only internal operations. We will explain later in the paper how external operations are specified in SKwyRL-ADL and why they play an important role in the definition of the system topology. From the set of all internal operations [*Operation*], we can specify an *AtomicAction* as in Figure 9.

[*Operation*]

AtomicAction

head: *Operation* input:*Belief*
head $\neq\emptyset\wedge$input $\neq\emptyset$

Fig. 9: AtomicAction specification

In order to design agents that present efficient behaviour in specific environment states, preconditions can be defined allowing the agent to choose a better action than it would otherwise have chosen. Once the action is selected, the agent can execute it. This affects (affect) the agent's informational or motivational states (Figure 10).

Output:= Belief | Goal
function:= Add_KB | Rem_KB
[*Affect*]:= function X output

Action

precondition: $\mathbb{P}$ *Belief* body: *AtomicAction* affect: *Affect*
body $\neq\emptyset\wedge$affect $\neq\emptyset$

Fig. 10: Action specification

Event. *An event is a belief or goal occurrence generated by an action or a service, which triggers the execution of a plan*. Events are the origin of all activity within an agent-oriented system. In the absence of events, an agent sits idle. Whenever an event occurs, an agent selects between the available plans and executes the selected plan (or plan set depending on the event processing model chosen), until it succeeds or fails. An event is either the effect of an action or a service, or it is exogenous to the system, resulting from an action or a service not accomplished by an agent in the system. We define an event as in Figure 11.

[*ExogEffect*]
[*TriggerEvent*]:= Effect | ExogEffect
[*Evtype*] = postEvent | sendEvent

Event

Trigger: *TriggerEvent*
destination: $\mathbb{P}$ *Agent*
type: *EVtype*

Trigger $\neq\emptyset$

$(\forall$ ev: *Event*) ev.type = sendEvent $\Rightarrow$ ev.dest $\neq\emptyset$

Fig. 11: Event specification

SKwyRL-ADL allows the specification of two types of events: postEvent and sendEvent. A postEvent describes an event that the agent can post. Posting an event means that an agent creates an instance of the event and posts it internally (i.e., sends the event to itself). Such as event needs to be handled by the agent's own plan. Inversely, a sendEvent identifies events that the agent sends externally (i.e., to another agents) or considered exogenous to the system.

Plan. *A plan defines a sequence of actions or/and services to accomplish a task or achieve a goal.* Plans are selected by agents, as described below. Selected plans constrain the agent behaviour and act as intentions. A plan can be specified as in Figure 12 and consists of:

- *invocation condition,* detailing the circumstances, in terms of beliefs or goals, that cause the plan to be triggered;
- *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*, which specifies either the sequence of formulae that the agent needs to perform. A formulae being either an action or a service (i.e., action that involves interaction with other agents) to be executed;
- *end state,* which defines the post-conditions under which the plan is succeeded;
- a set of services or actions that specify what happens when a plan *fails* or *succeeds*.

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

Plan

name: *PlanName*
invocation: $\mathbb{P}$ *Event*
Context: $\mathbb{P}$ *Belief*
Body: seq *AtomicPlan*
endState: $\mathbb{P}$ *Affect*
succeed: seq *Atomicplan*
Failure: seq *AtomicPlan*

name $\neq\ \emptyset\wedge$invocation $\neq\ \emptyset\wedge$body $\neq\ \emptyset$

Fig. 12: Plan specification

A *Plan* is said to have succeeded when it reaches its end state, and it is said to have failed if it is not in the end state and there

are no available actions or services. For instance, the DecompNmlRq and the DecompMCRq plans of the mediator agent deal with the decomposition of normal and multi-criteria (expressing the user preferences) requests.

```
Plan: {
      name: DecompNmlRq
      invocation: Add(Request_KB,
                  user_keyword(pt(+),kw(+))
  /* with pt:ProductType From Mediator.Ask(user_info-
needs).reply_with and  with kw:Keyword
FromMediator.Ask(user_info-needs).reply_with
      context: ¬ materialized_view(ProductType
                  = pt(+),Keyword = kw(+))
      body: ∀ pt : ProducType ∈
                  user_keyword(pt(+),kw(+))
                  Do
                  Action:
                    select_wrapper(wrapper(WrapL
                    ocalization,TranslationService(+))
                  as  wp(+): Wrapper
                  Service:
                    performative:
                     Ask(query_translation)
                    sender: Mediator
                    parameters: rt:RequestType∧
                                pt:ProductType∧
                                kw(+):Keyword
                    receiver: wp(+): Wrapper
                    Affect:
                  Add(Translation_Management
                    _KB, search(rt,pt,kw(+))
                  End-Do

      endstate: ∀ pt : ProducType ∈
                    user_keyword(pt(+),kw(+))
                  Do
                   Add(Translation_Management_KB,
                    search(rt,pt,fk(+))
                  End-Do

      suceed:
         Action:  count(search(rt,pt,kw(+))
         Affect:  Add(Request_Kb,
                     old_user_keyword(pt,kw(+))
      Failure:    Plan: DecompMCRq
}
```

Fig. 13: DecompNmlRq plan specification

The DecompNmlRq plan, shown in Figure 13, is triggered each time a new user_keyword belief is added to the Request KB. The argument values of the user_keyword belief are required by the Ask(*user_info-needs*) service that the mediator initiates. However, the plan is only executed if a materialized_view belief which has the same argument values as the invocation user_keyword belief does not exist. A materialized_view belief represents a repetitive user information need whose content is extracted from each source, merged with relevant information from other sources, and stored as a belief by the mediator. The complementary condition on the existence of a materialized_view belief is specified by the context. The context helps for the selection of the most appropriate plan in a given situation.

As soon as the invocation condition and the context are true, the sequence of actions or services specified in the plan body can be executed. The plan body of the DecompNmlRq plan is composed by the sequence of an action and a service. The mediator selects from their wrapper beliefs one or more wrappers (wp(+)) capable of translating the decomposed subqueries. Then, a translation service Ask(query_translation) is asked from the selected wrappers.

The plan will only succeed if the statement described by the end state is successful. Moreover, SKwyRL-ADL also allows specifying what happens when a plan reaches its endstate or fails, by considering further courses of action or service. For example, the succeed specification of the DecompNmlRq plan counts the number of occurrences of the current subquery in order to identify a possible new materialized view, while the fail specification returns to the execution of the DecompMCRq plan.

4.4.2. Security Model

The security model is composed of four main design entities. An agent has zero or more *protection objectives* and each security

objective imposes one or 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.

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 agents and/or 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 in Figure 14.

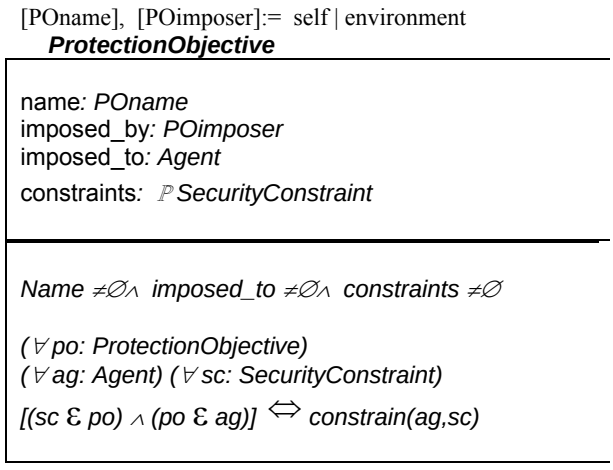

Fig. 14: Protection objective specification

In particular, a Protection Objective specification is described by a name (*Name*), which assists to identify the Protection Objective, and an imposer (*Imposed_by*), which describes who imposes the Protection Objective to the agent. *Imposed_to* indicates the agent the Protection Objective is imposed to, and *Constraints* provides the set of security constraints imposed as a result of the specific protection objective.

Referring to the Mediator agent specification, there are two protection objectives (Confidentiality_PO and Availability_PO). Following the Protection Objective specification of the proposed ADL, the Availability_PO is specified as shown in Figure 15. In particular, the *Mediator* agent is imposed an Availability Protection Objective from its Environment. As a result of this, a security constraint (*ConfirmServiceAvailability*) is imposed to the Mediator.

Protection Objective: {
name: *Availability_PO*
imposed_by: *Environment*
imposed_to: *Mediator*
constraints: *ConfirmServiceAvailability*}

Fig. 15: Availability_PO specification

Security Constraint. *A security constraint defines a set of restrictions to the goals and the capabilities of an 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 in Figure 16.

[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)

Fig. 16: Security constraint specification

Going back to the Mediator specification, the *ConfirmServiceAvailability* security constraint restricts the Mediator's *Keep Materialized View Up-to-date* goal at all times and for every communication. This is specified in Figure 17.

Security Constraint: {
name: *ConfirmServiceAvailability_SC*
restricts: Keep_MaterialisedView_Uptodate
timeFrame: *All*
constraints: *All*
}

Fig. 17: ConfirmServiceAvailability_SCspecification

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 types of security methods. Some of them related to the detection of security breaches, some of them related to the prevention of security breaches, and some of them related to the recovery from security breaches. Therefore, the type of a security mechanism could be one of the following: (1) detecting: which involves only security methods that aim to detect anomalies; (2) preventing: which involves only security methods used to prevent security intrusions; (3) recovering: which involves security methods used only to recover after a security incident; (4) combinational: which involves security methods of all types. A security mechanism is specified in Figure 18.

Going back to the GOSIS example, the Mediator agent has two security mechanisms (see Figure 2): *Data Integrity* and *Authentication Exchange*. The Data Integrity security mechanism (DataIntegrity_SM) is composed of a security method (Error Detection), which is

available for the Mediator agent; it is a combinational type of Security Mechanism and it helps towards the Confidentiality Protection Objective of the agent.

[SMname], [SMavailability]:= Available| Unavailable
[SMtype]:= Detecting | Preventing | Recovering | Combinational

SecurityMechanism

name*: SMname*
composed_of *: ℙ SecurityMethod*
type*: SMtype*
availability*: SMavailability*
help*: ℙ Protection Objective*

name ≠∅∧ composed_of ≠∅∧ type ≠∅

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

Fig. 18: Security mechanismspecification

Using the Security mechanisms specification of the proposed language, the above example is specified as shown in Figure 19.

Security Mechanism: {
name: *DataIntegrity_SM*
composed_of: *Error_detection*
type: *Combinational*
availability: *Available*
help: *Confidentiality* }

Fig. 19: DataIntegrity security mechanismspecification

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:

- an entry condition, indicating the factor(s) that cause the method to be triggered;
- the security action, which specifies the actions/services that the agent needs to perform with respond to the security method invocation;
- 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.

A security method is specified as in Figure 20.

[SMETname], [SMEToutput]:= Success| Fail
*[SMETtype]:=*Detect|Prevent|Recovery

SecurityMethod

name:*SMETname*
type: *SMETtype*
entry_condition: *SMETentrycondition*
action:*SMETaction*
end_condition:*SMETdesirable*
output:*SMETouput*
next_step: *SMETnextstep*

name ≠∅∧ type ≠∅∧ output ≠∅

(∀ SMET: SecurityMethod) (∃ ag : Agent) • use(smet,ag)

Fig. 20: Security method specification

Referring to the GOSIS example, previous analysis has identified that the *Data Integrity* security mechanism of the Mediator agent is composed of the *Error Detection* security method. Error detection is a method that allows some communication errors to be detected (for instance on communication between the

various agents of the system). The data is encoded so that the encoded data contains additional redundant information about the data. The data is decoded so that the additional redundant information must match the original information. This allows some errors to be detected. A number of error detection methods are currently used. For the sake of this paper we assume that the Mediator agent employs a Cyclic Redundancy Check (CRC) method. CRC is calculated by dividing the bit string of the block by a generator polynomial. The value of the cyclic redundancy check is the reminder of the calculation which is one bit shorter than the generator polynomial. Figure 21 shows an example of the CRC security method specification.

Security Method: {
name:*CRC_SM*
type: *Detect*
entry_condition:*EncodedData*
action:R*unCrcAlgortihm*
end_condition:*NoError*
output:*SMEToutput*
}

Fig. 21: Data Integrity security mechanism specification

4.4.3. Architectural Model

The main entities and relationships of the architectural model are illustrated in Figure 1. It is composed of seven main design entities. It describes interactions among the agents that compose the MAS. *Configurations* are the central concept of architectural design, consisting of interconnected set of agents. The topology of a configuration 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 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 an agent by one or more detailed, lower-level configuration descriptions. In the rest of this section, we define and specify, using Z, each entity of this structural model.

Interface. *An interface defines a collection of connection points through which an agent interacts with its environment.* An interface is a specification of how an agent appears to the rest of the system. Its primary constituents are a set of effectors and a set of sensors, which model the points through which an agent interacts. An effector provides a service for other agents and/or human users. A sensor requires a service from another agent and/or human user. For each interaction, there is always a correspondence between a service provided by an effector and a service required by a sensor, e.g. the send and receive request services of a client-server application.

We specify an interface as a non-empty, finite set of effectors or sensors that represents the complete set of interaction points through which an agent communicates. These two basic interfaces are distinct, yet they share many characteristics. It can be useful (when appropriate) to consider them as specialisations of the same type. In Z, this can be accomplished by defining them as disjoint subsets of the Interface type. We introduce the *AgentInterface* type as the infinite set of all possible effectors and sensor definitions (Figure 22).

[*AgentInterface*]

Interface

effector: $\mathbb{P}$ *AgentInterface*
sensor: $\mathbb{P}$ *AgentInterface*

(*effector*, *sensor*) partition *AgentInterface*

Fig. 22: Interface specification

The predicate partition indicates that the sets of effectors and sensors are disjoint. We define an *Effector* (Figure 23) or a *Sensor* (Figure 24) as an individual connection point that defines a set of required or provided services.

[*EffectorName*]

Effector

name: *EffectorName*
provide:*Service*

name ≠ ∅∧provide ≠ ∅

Fig. 23: Effector specification

[*SensorName*]

Sensor

name: *SensorName*
require:*Service*

name ≠ ∅∧require ≠ ∅

Fig. 24: Sensor specification

Referring to the GOSIS example, the Mediator needs the query_translation service that the Wrapper provides. Such interface definition points two aspects of an agent. Firstly, it indicates the expectations the agent has about the agents with which it interacts. Secondly, it reveals that the interaction relationships are a central issue of the architectural description. Such relationships are not only part of the specification of the agent behavior but reflect the potential patterns of communication that characterize the ways the system reason about itself. However, the required query translation service needs to be specified in greater detail. This is possible with the aid of a service.

Service. *A service is an operation performed by a sender agent that interacts by dialoguing with one or more receiver agents*. We represent a service as a kind of action called speech act in the literature [3]. A speech act is an action available to communicate what an agent knows about the environment. It has the effect of changing the state of the environment just as any action.

A service is specified using the model of action defined in Section 4.4.1, i.e., with preconditions and affects, as described in Figure 25.

Service

precondition: $\mathbb{P}$ *Belief*
body: *AtomicService*
affect: *Affect*

body ≠∅∧affect ≠∅

Fig. 25: Service specification

We define an *AtomicService* (Figure 26) in the same way as a KQML [10] inter-agent communication. It consists of:
a *performative* that names the services; a *sender* that identifies the agent initiating the services in the architecture; the *reply-with* that defines the information about which the

service expresses an interaction; a set of *receivers* that identify the agent's interaction with the sender; the *parameters* that define the information required to execute the service; and an optional *ontology* that defines the agreed-upon terms that will be used in the exchange.

[*Performative*]
[*Ontology*]

AtomicService

name: *Performative*
sender: *Agent*
parameter: $\mathbb{P}$ *Term*
reply-with: *Belief*
receiver: $\mathbb{P}$ *Agent*
ontology: *Ontology*

name≠ ∅∧sender ≠ ∅∧receiver ≠ ∅

(∀ s: *Service*) s.sender ≠ s.receiver

(∀ s: *Service*) ∃ ag_1: s. sender∧∃ ag_2: s. receiver⇒ s.parameter ∈ ag_2.belief

Fig. 26: AtomicService specification

Referring back to the GOSIS case study, we can see (Figure 27) that the mediator (sender) initiates the service by asking the wrapper (receiver) to translate a query. To this end, the mediator provides to the wrapper a set of parameters allowing the definition of the contents of this query. Such mediator query is specified as belief with the predicate search and the following terms:

search(*RequestType*,*ProductType*(+),*FilteredKeyword*(+))

Each term represents, respectively, the type of the request (normal or advanced in the case of multi-criteria refinement); the type of product; and one or more keywords that must be included in or excluded from the results.

The Affect indicates that a new search belief is added to the Translation_Management knowledge base of the wrapper.

Service*:*
performative: Ask(*query_translation*)
sender: Mediator
parameters: rt: RequestType ∧ pt:ProductType ∧ fk(+):FilteredKeyword
receiver: Wrapper
Affect: Add(Translation_Management_KB, search(*rt*,*pt*,*fk*(+))

Fig. 27: Example of service specification

Configuration. *A configuration is an interconnected set of agent instances*. A MAS modelled at the architectural level of design is represented as a configuration of instantiated agent components. The topology of the system is defined by a set of bindings between services provided by effector instances and services required by sensor instances. 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 of the composition without having to examine each of the individual components in a system. Because there may be more than one uses of a given agent in a MAS, we distinguish the different instances of each agent type that appear in a configuration. To this end, in our ADL we define the type *Instance* 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 types: [*IRService*] and [*IPService*].

Once the instances have been declared, a configuration is completed by describing the collaborations. The collaborations define the

topology of the configuration, by showing which agent instance participates in which interactions. This is done by defining a one-to-many mapping relation between provided and required services. A configuration can be then specified as in Figure 28.

[*AgentDescription*]
[*IAgent*]
[*Instance*] := IAgent | IPService | IRService

Configuration

description: $\mathbb{P}$ *AgentDescription*

inst:ance: $\mathbb{P}$ *Instance*

collaboration: (*IAgent* X *IRService*) → (*IAgent* X *IPService*)

description ≠ ∅∧*instance* ≠ ∅∧*collaboration* ≠ ∅

Fig. 28: Configurationspecification

Part of the GOSIS configuration with instance declarations and collaborations is given in Figure 29.

"(*min*)...(*max*)" indicates the smallest acceptable integer, and the largest. An omitted cardinality (as is the case with (*max)* in the broker, mediator and wrapper agents), means no limitation. Dynamic and evolving structures can change at runtime. Such a configuration allows for dynamic reconfiguration and architecture resolvability at run-time. Configurations separate the description of composite structures from the description of the elements that form those compositions. This permits reasoning about the composition as a whole and allows reconfiguration without having to examine each component of the system.

Architecture. *Architecture models the full set of design information defined within an architectural specification.* An important property for an ADL is to allow basic components in architecture to be replaced by (sub) configurations, in order to form new configurations.

Configuration GOSIS
Description
Agent Broker[*nb: 1…*]
Agent Mediator[*nm: 1…*]
Agent Wrapper[*nw: 1…nS*]
/*with *nS* = number of information sources
Agent Monitor[*nmo: 1…nS*]
Agent Matchmaker
Agent Multi-Critria-analyzer
Service Tell(*query_translation*)
Service Ask(*query_translation*)
Service Achieve(*result*)
Service Do(*result*)
Service Tell(*subscription_info*)
Service Ask(*subscription_info)*
…
Instance
BR_{nb}: Broker
ME_{nm}: Mediator
WR_{nw}: Wrapper
MO_{nmo}: Monitor
MA: Matchmaker
MCA: Multi-Criteria-Analyzer
Tellquerytrans: Tell(*query_translation*)
Askquerytrans: Ask(*query_translation*)
Achres: Achieve(*result*)
Dores: Do(*result*)
Tellsubs: Tell(*subscription_info*)
Asksubs: Ask(*subscription_info*)
…
Collaborations
ME_{nm}.Askquerytrans - Tellquerytrans.WR_{nw};
ME_{nm}.Achres --- Tellres.WR_{nw};
ME_{nm}.Asksubs --- Tellsubs.MA;
…
End GOSIS

Fig. 29: Part of the GOSIS configuration

This is usually referred to as hierarchical refinement in software architecture. To support hierarchical descriptions, SKwyRL-ADL permits the representation of an agent by one or more detailed, lower-level descriptions. This is specified with the concept of architecture. The architecture models a complete specification. Each design has at least one configuration corresponding to the top-level system model. However, our model also supports

hierarchical system descriptions; that is, an agent may be further specified as being implemented by a configuration. An architecture maintains the set of all of the configurations that have been defined, and it can be specified as a non-empty set of configurations that has been defined in the specification (Figure 30).

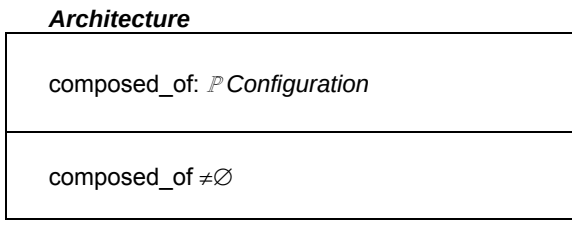

Fig. 30: Architecture configuration

5. The e-commerce system Case Study

E-Media[5] is a typical business-to-consumer application we have developed using the ADL described in the previous sections. 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 SKwyRL ADL to formally specify architectural aspects of the system, such as interfaces, knowledge bases, and security mechanisms. In particular, in section 5.1, we introduce the case study and we discuss with the aid of secure Tropos [5] and *i**[47] diagrams the analysis of the system, its requirements and the selected architectural style for the e-media system. In Section 5.2 we illustrate how the proposed ADL was used to support the architectural description of the system by focusing on one of the system's main components; the Billing Processor Agent. Then in section 5.3, we explain how the developed architectural solution was implemented and we provide illustrations of the system's implementation.

5.1. 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 on this statistical and strategic information, the system continuously 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 online purchases are minimized. Therefore, from the customer's point of view the main security objectives are confidentiality and integrity.

[5] (http://www.isys.ucl.ac.be/skwyrl/emedia)

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 needs 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 31 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 secure Tropos language [35] where each node represents an actor (or system component) and each link between two actors indicates a dependency. 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; soft-goal dependencies are similar to goal dependencies, but their fulfilment cannot be defined precisely; task dependencies are used in situations where the dependee is required. A *Secure Dependency* introduces security constraint(s) that must be respected by actors for the dependency to be satisfied [35]. This means that the depender expects from the dependee to satisfy the security constraint(s) and also that the dependee will make effort to deliver the dependum by satisfying the security constraint(s).

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 → dependum → dependee. Security constraints are represented as hexagons.

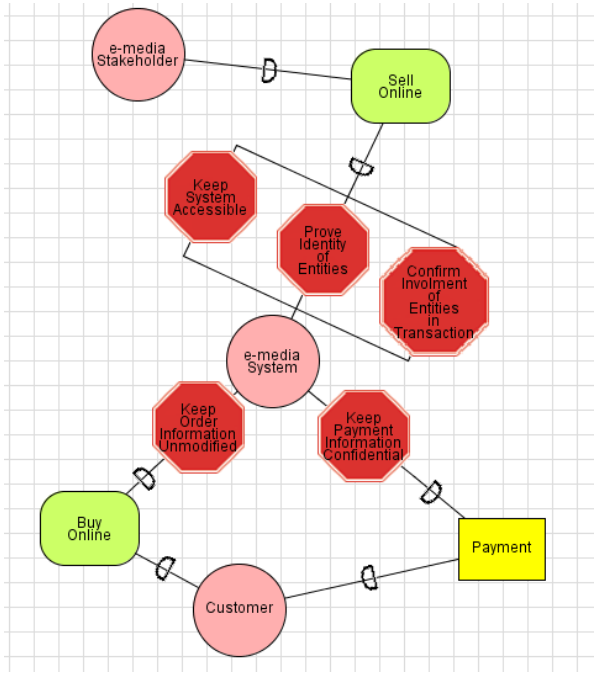

Figure 31: E-Media dependencies

Following the secure Tropos analysis process, the structure-in-5 organizational architectural style, presented in [23], was

identified as suitable for the e-media system. More information about alternative architectural selections can be found in [13]. According to the structure-in-5 style, the organisation of the software architecture can be considered an aggregate of five sub-structures [33]. 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 Apex and the Operational Core; the *Technostructure*, which serves the organization by making the work of others more effective, typically by standardising 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 32.

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. To be able to produce statistical information (e.g., analyses, average charts and turnover reports), the *Back Store* stores and backs up all the appropriate web information about customers, products and sales. Such kind of information is analysed either for a predefined product (when the *Coordinator* asks it) or on a monthly basis for every product. Based on this monthly statistical analysis, strategic information (e.g., sales increase or decrease, performance charts, best sales, and sales prevision) is also provided to the *Decision Maker*.

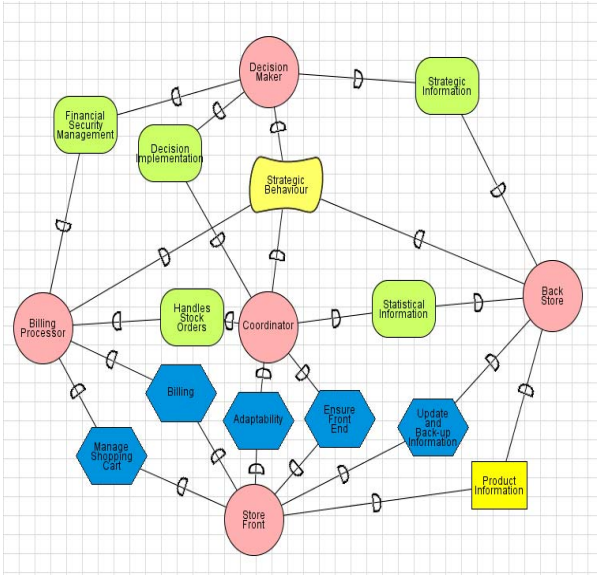

Figure 32: The E-Media Architecture in Structure-in-5

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*, 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 implement 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 fulfils its mission in an effective way. Finally, the *Decision Maker* assumes strategic roles. It defines the strategic behaviour (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.

5.2. *Architectural Description*

The initial analysis of the case study, as partially presented in the previous section, provides an organizational representation of the system-to-be including relevant actors, security constraints and their respective goals, tasks and resource inter-dependencies. Such analysis can serve as a basis to understand and discuss the assignment of system functionalities and security issues but it is not adequate to provide a precise specification of the system details. As introduced in the previous sections, 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, their behaviours and the corresponding security issues. The rest of the section describes the specification, in SKwyRL-ADL, of one of the main components of the e-media system as introduced in 5.1 and 5.2; the Billing Processor agent. Focusing on this component of the system allows us to demonstrate the applicability of the proposed ADL and also in the same time to keep the description to a reasonable and manageable length. For a complete specification of the E–Media case study, we refer the reader to [14].

There are five main aspects of the Billing Processor (BP) agent that need to be considered: 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 help towards the satisfaction of the protection objectives of the agent, and the *Capabilities* defining agent behaviours.

In particular, the Billing Processor agent Interface consists of a number of effectors and sensors for the agent. Effectors provide services to other agents, and sensors require services provided by other agents. An interaction is then defined by the correspondence between a required and a provided service. As shown in Figure 33 the BP agent's interface consists of four effectors and two sensors.

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_POAccessControl_PO
Security mechanisms:
Encipherment_SMDIgitalSignature_SM
AccessControl_SM DataIntegirty_SM
AuthenticationExchange_SM
TrafficPadding_SMRoutingControl_SM
Notarization_SM
Capabilities:
Shopping_Cart_Management_CP
Billing_CP Stock_Management_CP
Statistic_CP
}

Figure 33: Billing Processor agent specification

Moreover, the BP agent has a number of knowledge bases related to customers, providers, pricing and stock as well as a number of protection objectives, such as (amongst others) confidentiality, integrity, and availability. The BP agent also has a number of security mechanisms, such as access control and notarization, as well as a number of capabilities, such as stock management and billing.

Once all the five aspects have been defined, more detail specifications are constructed for each one of these. For instance, the BP agent Interface components are further specified. In particular, 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 with 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) as shown in Figure 34.

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")}

Figure 34: Ask statistical specification

As discussed above, the BP agent has six KBs. Following the ADL specification for Knowledge Bases, introduced in the previous sections, each KB is specified with a name, a KB_body and a KB_type. For example, the specification of the Statistical_KB for the BP agent is shown in Figure 35.

That specification describes the formal knowledge that the BP agent has with respect to the statistical analysis required. As shown in Figure 35, this includes, amongst other things, product turnover, customer turnover and product sales. Since the BP agent only knows the beliefs included in its KB, the type of the Statistical KB is *closed world*.

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

Figure 35: Statistical Knowledge Base for BP specification

The high-level specification of the BP agent indicates that the agent has six (6) protection objectives. These protection objectives have been identified by the security analysis that took place for the e-media system and partially presented in section 5.1. In particular, the initial security constraints (illustrated in Figure 31) were further analysed into secure goals and plans which in turn allowed us to identify a number of protection objectives to satisfy the secure goals and plans of the BP agent. Following the ADL structure and in particular the Protection Objective specification (as presented in previous sections), each of protection objective of the BP agent is specified with a name, information of who imposed it to the agent, the agent to which it is imposed to (in this case the Billing Processor), and the constraints that it imposes to the agent. For example, the specification of the Non_Repudiation Protection Objective is illustrated in Figure 36.

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

Figure 36: Non Repudiation Protection Objective specification

Our security analysis, during the analysis stage of the development process, indicated that the BP agent should have 8 different security mechanisms in order to satisfy its security requirements. Following the specification for Security Mechanisms presented in the previous section, 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. For instance, the Notarization security mechanism specification for the Billing Processor agent is shown in Figure 37. It is important to emphasise that the notarisation mechanism is provided by a third-party notary, which must be trusted by all participants. 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.

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

Figure 37: Notarization security mechanism specification

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 specified as shown in Figure 38. 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
}
```

Figure 38: Billing Processor Statistic_CP capability specification

The Stock_Order plan of the Billing-Processor (Figure 39) ensures that the level of stock of each product is constantly higher than the minimal quantity, which is determined by the coordinator on the basis of the strategic orientation provided by the Decision-Maker.

```
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,d
                  ate))
                  update_stock(id, av)
          effect: Add(Stock_Kb, Stock(id, av))
}
```

Figure 39: Stock_Order plan specification

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 it reorders the same quantity. Then the stock level is updated with the quantity ordered.

5.3. *E-Media Implementation*

Following the analysis of the system (partially presented in 5.1) and the specification of its architecture (partially presented in 5.2), the next step included the implementation of the system based on the developed architecture. In the rest of this section, we briefly describe the E-Media implementation to illustrate the roles of the agents (Front Store, Decision Maker, Back Store, Coordinator and Billing Processor) identified in 5.1 and their interaction. We also focus the discussion on implementation aspects of the Billing Processor agent described in section 5.2. The E-Media application was implemented (~ 10.000 lines of code) using JACK [26]; a BDI agent-oriented development environment for JAVA.

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 user to sign in. 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 40 illustrates the Store-Front interface for the DVD section.

To search the E-Media DVD catalogue, the user must fill in at least one field of the search engine (see section 1 in Figure 40). The Store-Front sends the query parameters to the Back Store which provides the results back to the Store-Front (see section 2 in Figure 40).

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 (see section 3 in Figure 40).

Figure 40: Interface of e-media DVD section

The implementation for the Billing Processor has followed the architectural description presented in the previous section (5.2). In particular, when a user starts the

billing process, the Billing Processor agent invokes the shopping cart and billing effectors and displays all the items of the shopping cart and it computes the total and sub-total for each product. It then employs a number of its KBs, POs and SMs to validate the user id and process the payment card. As discussed in the previous section (5.2) a third party is used to support the notarization security mechanism of the BP agent. During all this time, the BP agent communicates with the user, through user messages on the screen, as illustrated in Figure 41, by employing its sensors and effectors. Once the payment is accepted, the Billing Processor uses again its interface to communicate with the Store-Front. Furthermore, a confirmation message is displayed and the shopping cart is cleared.

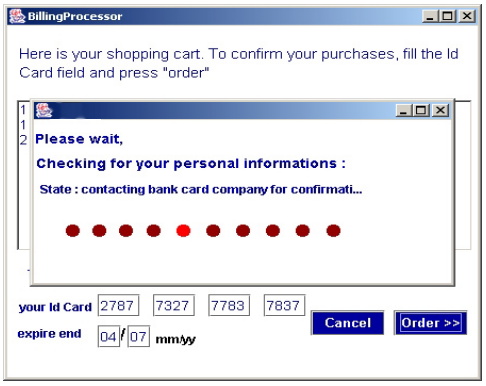

Figure 41: Secure Payment Information.

6. Related Work

Over the past decade, the field of software architecture has received increasing attention as an important subfield of software engineering. Practitioners have come to realise that getting an architecture "right" is a critical success factor for system design and development. They have begun to recognise the value of making explicit architectural descriptions and choices in the development of new systems. In this context, a number of researches have proposed architectural description languages ([2], [17], [28], [30] and [39]) for representing and analysing architectural designs. In particular, a number of Architectural Description Languages (ADLs) have been proposed such as Rapide[29], Darwin[32], Aseop[18], Unicon[39],Wright [2] and ACME[17]. However, these efforts have not been undertaken with agent orientation in mind, and therefore the resulted languages are not applicable for specifying MAS architectures. Nevertheless, the study and careful examination of the above efforts has enable us to identify the essential aspects that any ADL should be able to specify, and develop the common foundation of concepts and concerns for the language proposed in this paper (see Section 4 for details). On the other hand, few efforts have been made to define an architectural description language (ADL) for MAS. MAS-adl[10]is a simple customized architectural description language for MAS used to describe the various classes of agents involved in the MAS and the interconnections between their instances. In particular, for any given class the architectural description provides information about the kind of the architecture; the interpreters and the services of agents in that class. ADLMAS [48] is an architectural description language for MAS, which adopts Object-Oriented Petri nets as a formal theory basis. ADLMAS is suitable for representing concurrent, distributed and synchronous MAS. ADLMAS can visually and intuitively depict a formal framework for MAS, from the agent level and social level, which describes the static and dynamic semantics, and analyse, simulate

and validate MAS and interactions among agents with formal methods. Similarly Yu et al. [49], have developed a MAS ADL based on π-net. The proposed ADL supports the Belief-Desire-Intention (BDI) model and it makes use of two formalisms, namely Agent-Oriented Petri nets (AOPN) and π-calculus. AOPNs are used to visualize the static architecture and model the behaviors of the MAS under development, while π-calculus is used to represent the dynamic architecture of MAS. These works are important, but they fail to adequately consider the issue of security in MAS.

There is, of course, a large effort of works in the area of MAS, with respect to security. These include policy specification languages (for example [23],[4]), Trust and Reputation mechanisms(for example [20], [22], [30]), agent-oriented software engineering methodologies for the analysis and design of secure MAS [35], security patterns[36], and security mechanisms and protocols [4]. Although such works do not directly input into our proposed language; their study has enable us to consider a variety of security considerations, issues and challenges faced for the development of secure MAS and therefore develop a security model for the proposed ADL that supports appropriate architectural concepts.

7. Conclusion

Today's information systems must be based on open architectures to accommodate new components and meet fast evolving requirements. MAS architectures provide an effective way to design such systems since they do support open and evolving configurations that can change at run-time to exploit the services of new agents, or replace existing ones.

Based on the analysis of existing classical ADLs, security consideration for multi-agent systems and the BDI agent model, this article has defined a set of system architectural concepts to propose an ADL for secure BDI-MAS. This ADL allows specification of agent components (such as knowledge base, interface and capabilities), agent behaviour (such as belief, goal and plan), agent security (such as security constraints, protection objectives and security mechanisms) and agent interactions (such as service and configuration).

The research reported here calls for further work. We are currently working on:

- the development of a CASE tool to automatically generate the code skeleton of the future multi-agent information system from their specification with the ADL;
- the definition of a set of rules to perform consistency analysis that could be included in commercial verification tools such as PVS (http://pvs.csl.sri.com/);
- the identification of a suitable set of core abstractions, inspired by an organizational metaphor, to be used during the design of multi-agent systems;
- the development of a clear methodology, centred around these organizational abstractions, for the design of multi-agent systems architectures.

References

[1] R. Allen, “A Formal Approach to Software Architecture,” Ph.D. Thesis, Carnegie Mellon University, Technical Report Number: CMU-CS-97-144, 1997.

[2] R. Allen and G. Garlan, “Formal Connectors,” Software Architecture Lab., Carnegie Mellon University, Pittsburgh, USA, Technical Report CMU-CS-94-115, 1994.

[3] J. L. Austin, How to do things with worlds, Oxford University Press, 1962.

[4] M. Barley, H. Mouratidis, A. Unruh, D. Spears, P. Scerri, and F. Massacci,(eds.) (2008) "Safety and Security in Multiagent Systems: The Early Years",Springer-Verlag, forthcoming.

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

[6] M. Bratman, Intentions, Plans and Practical Reasoning, Harvard Univ. Press, 1988.

[7] P. C. Clements, “A Survey of Architecture Description Languages,” in Proc. of the 8th Int. Workshop on Software Specification and Design, Paderborn, Germany, 1996, pp. 123-131.

[8] D. Calvanese, S. Castano, F. Guerra, D. Lembo, M. Melchiori, G. Terracina, D. Ursino, and M. Vincini, “Towards a Comprehensive Methodological Framework for Semantic Integration of Heterogeneous Data Sources,” in Proc. of the 8th Int. Workshop on Knowledge Representation meets Databases, Rome, Italy, 2001.

[9] D. Calvanese, G. De Giacomo, M. Lenzerini, D. Nardi, and R. Rosati, “Schema and Data Integration Methodology for DWQ,” Foundations of Data Warehouse Quality Project, Dipartimento di Informatica e Sistemistica, Università di Roma "La Sapienza", Report DWQ-UNIROMA-004, 1998.

[10] A. Cuppari, P. L. Guida, M. Martelli, V. Mascardi, and F. Zini. “An Agent Based Prototype for Freight Trains Traffic Management”. In Proc. of FMERail'99 Workshop, Toulouse, France, September 1999.

[11] A. Dardenne, A. van Lamsweerde and S. Fickas, “Goal-Directed Requirements Acquisition,” Science of Computer Programming, 20(1): 3-50, 1993.

[12] A. Dal Formo and U. Mendenlo, “A Multi-Agent Simulation Platform for Modeling Perfectly Rational and Bounded-Rational Agents in Organizations,” Artificial Societies and Social Simulation, 5(2):166-177, 2001.

[13] 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.

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

[15] S. Faulkner, M. Kolp, T. Nguyen and A. Coyette, “A Multi Agent Perspective on Data Integration Architectural Design”. In Knowledge-Based Intelligent Information & Engineering Systems (KES), 8(1):1150-1156, LNCS 3213, Springer, 2004

[16] T. Finin, R. Fritzson, D. McKay and R. McEntire, “KQML as an Agent Communication Language,” in Proc. of the 3rd Int. Conf. on Information and Knowledge Management,Gaithersburg, USA, 1994, pp. 456-463.

[17] D. Garlan and R. Monroe, “Acme: an architecture description interchange language,” in Proc. of the 7th Annual IBM Centre for Advanced Studies Conference, Toronto, Ontario, 1997, pp. 78-86.

[18] D. Garlan, R. Allen, and J. Ockerbloom.Exploiting Style in Architectural Design Environments. InProceedings of SIGSOFT’94: Foundations of SoftwareEngineering, pages 175–188, New Orleans, Louisiana,USA, December 1994.

[19] H. Garcia-Molina, Y. Papakonstantinou, D. Quass, A. Rajaraman, Y. Sagiv, J.D. Ullman, and J. Widom, “The TSIMMIS Approach to Mediation : Data Models or Languages,” Journal of Intelligent Information Systems, 8(2):117-132, 1997.

[20] A. Herzberg, Y. Mass, J.Mihaeli,D.Naor, and Y. Ravid, Access Control meets Public Key Infrastructure : OrAssigning Roles to Strangers. In Proceedings of 2000 IEEE Symposium onSecurity and Privacy, Oakland, May 2000, 2000.

[21] J. Hintikka, Knowledge and belief, Cornell University Press, 1962.

[22] Y. Hu. Some thoughts on Agent Trust and Delegation. In Proceedings ofAutonomous Agents 2001, 2001.

[23] L. Kagal and T. Finin. Modeling conversation policies using permissions and obligations. Autonomous Agents and Multi-Agent Systems 14, 2 (Apr. 2007), 187-206.

[24] 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.

[25] K. Konolige, “A first order formalization of knowledge and action for multi-agent planning system,” Machine Intelligence, 10: 41-72, 1982.

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

[27] M. Luck and M. d’Inverno, “A formal framework for agency and autonomy,” In Proc. of the 1st Int. Conf. on Multi-Agent Systems,San Francisco, USA, 1995. pp. 254-260.

[28] D. C. Luckham, J. J. Kenney, L. M. Augustin, J. Vera, D. Bryan and W. Mann, “Specification and Analysis of System Architecture Using Rapide,” IEEE Transactions on Software Engineering, 21(4):336–355, 1995.

[29] D. C. Luckham, and V. Vera, “An Event-Based Architecture Definition Language,” IEEE Transactions on Software Engineering, 21(4):717-734. 1995.

[30] B. B. Madan and K. Goseva-Popstojanova and K. Vaidyanathan and K. S. Trivedi, “Modeling and Quantification of Security Attributes of Software Systems”, in Proc. of the 2002 International Conference on Dependable Systems and Networks (DSN’02), Washington, DC, USA, pp. 505-514, 2002.

[31] Y. Mass and O. Shehory. Distributed Trust in Open Multi Agent Systems. InWorkshop on Deception, Fraud and Trust in Agent Societies, Autonomous Agents2000, 2000.

[32] J. Magee and J. Kramer, “Dynamic Structure in Software Architectures,” in Proc. of the 4th Int. Conf. on the Foundations of Software Engineering, San Francisco, CA, USA, 1996. pp. 3-14.

[33] H. Mintzberg. Structure in fives: designing effectiveorganizations. Prentice-Hall, 1992.

[34] H. Mouratidis, S. Faulkner, M. Kolp, and P. Giorgini, “A Secure Architectural Description Language for Agent Systems”. In Proc. of the 4th International Joint Conference on Autonomous Agents and Multi-Agents Systems (AAMAS'05), Utrecht, The Netherlands, pp. 578-585, 2005.

[35] H. Mouratidis and P. Giorgini. Enhancing secure Tropos to effectively deal with security requirements in the development

of multiagent systems, in the proceedings of the 1st International Workshop on Safety and Security in Multiagent Systems, AAMAS 2004, N.Y. USA, 2004

[36] H. Mouratidis, P. Giorgini, M. Schumacher . Security Patterns for Agent Systems. In Proceedings of theEight European Conference on Pattern Languages of Programs (EuroPLoP), Irsee, 2003.

[37] Y. Papakonstantinou, H. Garcia-Molina and J. Ullman. Medmaker, "A mediation system based on declarative specification," in Proc. of the 12th Int. Conf. On Data Engineering, New Orleans, 1996, pp. 132-141.

[38] V. S Subrahmanian, S. Adali, A. Brink, R. Emery, J. J. Lu, A. Rajput, T. J. Ross, C. Ward, "Hermes: A Heterogeneous Reasoning and Mediator System," in Proc. of the Int. Conf. On Database Theory, Delphi, Greece, 1997, pp 19-40.

[39] 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.

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

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

[42] S. Vestal, "A Cursory Overview and Comparison of Four Architecture Description Languages," Honeywell Technology Center, Technical Report, 1993

[43] J. Widom, "Research Problems in Data Warehousing," in Proc. of the 4th Int. Conf. on Information and Knowledge Management, Baltimore, Maryland, USA, 1995, pp. 25-30.

[44] G. Wiederhold, "Mediators in the architecture of future information system," IEEE Computer, 25(3): 38-49, 1992.

[45] G. Wiederhold, "Intelligent Integration of Information," in Proc. of the ACM SIGMOD Conference on Management of Data, Washington, USA, 1993, pp. 434-437.

[46] M. Wooldridge and N.R Jennings, "Special Issue on Intelligent Agents and Multi-Agent Systems," Applied Artificial Intelligence Journal, 9(4):74–86, 1996.

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

[48] Z. Yu, Z. Li. "Architecture description language based on object-oriented Petri nets for multi-agent systems" In Proc. of the 2005 Int. Conf on Networking, Sensing and Control, Tucson, USA, pp. 256-260, 2005.

[49] Z. Yu, Y. Cai, R. Wang, J. Han, "π-net ADL : An architecture description language for multi-agent systems", In Proc. Of International Conference on Intelligent Computing, Lecture Notes in Computer Science vol. 3645, Springer-Verlag, 2005.