Original Research Paper

An Exception Management Model in Multi-Agents Systems

[1]Belkacem Athamena and [2]Zina Houhamdi

[1]*Department of Management and MIS, Al Ain University of Science and Technology, Al Ain, UAE*
[2]*Department of Software Engineering, Al Ain University of Science and Technology, Al Ain, UAE*

Article history
Received: 27-12-2016
Revised: 07-02-2017
Accepted: 29-05-2017

Corresponding Author:
Belkacem Athamena
Department of Management and MIS, Al Ain University of Science and Technology, Al Ain, UAE
Email: athamena@gmail.com

Abstract: Multi-Agents Systems (MAS) are modern approaches that need an additional investigation to improve their reliability and adaptability levels. Exception management is one way to reach this goal and this paper is dedicated to this specific subject. The purpose of this document is to examine the exception concept in MAS domain and to suggest a model adjusted to MAS challenges such as heterogeneity, openness and particularly agents' autonomy. Previous attempts in the agent's society have concluded set of findings that demonstrated the necessity of exception handling in MAS at the system level. The handling includes management and the needed processes related to management. The attainment up to now can be applied only to special MAS type. Usually, agents are non-autonomous and the system-level strategies need an impeccable cooperation between agents in the exception handling process. In our proposed model, the agent's ability to approach exceptions by itself is considered as a prerequisite to assure agent autonomy. Then, exception handling depends on agent-level processes to deal with the limitations of contemporary attainments and thus, they are complementary. Agent preserves the ability to independently decide when to activate exception handling and when to receive system-level help or believe in its skills.

Keywords: Multi-Agent System, Agent Autonomy, Exception Handling

Introduction

MAS belong to the most recent versions of intelligent systems. They are made of software programs called agents that work concurrently and collaborate to complete the system functionalities in a certain context (Weiss, 2013). They are used in case of a complex task that can be decomposed into a set of sub-tasks: Agents work out the assigned sub-tasks and they collaborate to produce a global output. The notable characteristic of MAS consists of the feature that agent is assumed *autonomous decision-making entity* (Houhamdi and Athamena, 2012). That is, agents, collaborate to fulfill their jobs. However, they haven't direct authority through others and they can decline to collaborate (Athamena and Houhamdi, 2012).

In order to guarantee the autonomy, agent state should be hidden and can't be read or updated by others agents. The agent has then local techniques to collaborate with others and to divulge or conceal details of its state. Consequently, the autonomy concept is appropriate to today software requirements. The tasks globalization and the Internet emergence require having entities collaborate across workstations connected by a network. The entities are organizations, societies, or people who are autonomous and desire to protect their private data. The software developers need to carry the tasks of the participants, either online using Internet or within a small environment using a local network. These participants may be modeled as autonomous agents that operate in a social system. Thus MAS is remarkably appropriate to the actual requirements. MAS are suitable software frameworks to cope with these concerns and produce relevant solutions to the software developers.

Besides the MAS fitness to actual requirements in the software engineering, autonomous agents are moreover encouraging techniques to the eternal increase needs of tasks' computerization. The previous entities cooperate to perform their tasks and these activities are repetitious and sometimes needless, but their completion needs a special autonomy ratio. Bringing software agents to help or act for users in the completion of their activities has been an objective from the arrival of Artificial Intelligence system with a unique agent.

Diverse methods are developed to improve the software reliability and Exception Management is one of them which is reputably known for its power and simplicity. Decentralized execution has demonstrated that exception management techniques need particular improvements for distributed systems and achievements in software design and component-based software engineering has set out the necessity for new methods also. MAS present challenging characteristics that require reexamining the exception topic (Goodenough, 1975; Platon *et al*., 2006).

The purpose of this paper is to supply the agent with exception handling skills. An execution model is at the top of these skills to identify exceptions and get ready the agent for their handling. The model proposed in this document guarantees the agents' autonomy by developing a new execution model that ensures the agent keeps itself control during the task processing even with the detection of exceptions. Thus, the agent settles without help (stand-alone) if a situation is an exception or not, consequently increases more its autonomy.

Background

MAS seem like an efficient solution to actual problems in the different application domain. During the literature survey, the current research achievements can't establish some properties required by the software developer and the end-users from modern applications and that were incipiently pledged by agent community as advantageous characteristics. Among these characteristics that need more investigation and considered as hard to realize are *reliability* and *adaptability*. The two properties are linked to the MAS reaction to unusual circumstance, namely exception.

Reliability concerns the software qualities, about dependability, availability, security and safety (Weyns *et al*., 2005). Accordingly, software is reliable if it can provide continual services, it doesn't provoke harms and it ensures the participant's privacy. In the MAS context, such investigation is in fact related to the traditional software engineering subjects, particularly, in the distributed systems field. Limited works discuss explicitly problems related to MAS (Athamena and Houhamdi, 2012; Guessoum *et al*., 2006; 2004; Houhamdi and Athamena, 2011a; Sichman *et al*., 1994). Fault tolerance methods including replication and monitoring are applied to autonomous agents to certify certain degree of reliability. The main problem with the actual techniques is the difficulty to find an agent-oriented approach that is accepted as a Software Engineering approach and respects all MAS properties, more precisely the autonomy property.

Adaptability is the ability of the software to reach its goal regardless local and/or external problems. The external problems are related to the environment which is dynamic and usually undependable. So, Adaptability describes how well the MAS accommodates to local or external stress. MAS without adaptability feature operates inappropriately when the agents behave in a unpredicted way, or the environment doesn't meet the desired requirements. On the other hand, MAS with high adaptability level can accommodate to modification in the behavior of the agents or the context and continue to operate correctly. Accordingly, Adaptability is unavoidable to ensure reliability. On the other hand, it is linked to the notion of self-recovery software and usually autonomic computing. An autonomous agent is supposed to be adaptable: To fulfill their tasks in spite of the exceptional situation.

Consequently, the MAS reliability can depend on the adaptability of its agents. The majority of the adaptability techniques are macro-approaches since they consider the entire system, conversely the micro-approaches focus on the agent. Interaction protocols, distributed algorithms and other system-level methods are an example of macro approaches where the agents act with the certain level of adaptability (Klein *et al*., 2003). But, the achievements of micro approaches are very fewer, in spite of the advantage of having agents extremely adaptable about their autonomy. Few investigations have been done, for example, commitment protocol and self-controlled agent (Mallya and Singh, 2005) and many problems need to be solved, encompassing the hybrid approach that combines the macro and micro approaches.

There are a lot of techniques to enhance the MAS reliability and adaptability. They belong to Artificial Intelligence, Distributed Computing and Software Engineering fields and they can be applied to MAS under some constraints (Platon *et al*., 2006). Among these techniques, we can find *Exception Handling* which is established for many years in programming languages and recognized as a useful and robust mechanism, but simple in its concepts. If a program encounters an unusual situation such as missed parameters or unexpected type, an Exception Handling Mechanism (EHM) deviates the execution flow to a handler (a program prepared to control a particular circumstance for the benefit of the main program). The EHM redirects the execution flow, on handler termination, back to the main program. The fundamental EHM is represented in Fig. 1.

An EHM includes supplementary tools to consider the exceptions including the handler identification from the program call stack in case non-availability of the handler. The call stack contains a list of procedure calls that are performed during the program execution. If no handler is found at exception occurrence, a handler is searched and asked the prior caller in the call stack. The exploration carries on until identification of a handler or the call stack is empty. In the last case, the program isn't able to manage the exception and must abort.

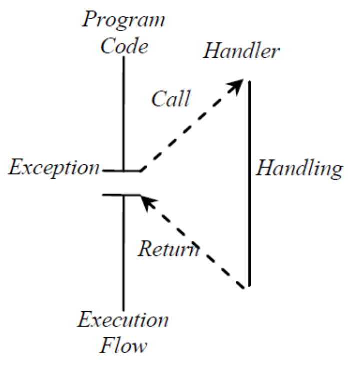


Fig. 1. Exception handling mechanism

In the MAS context, the EHM concept is interesting; however, the distribution and autonomy are two challenges making hard the application of this concept. Works in distributed computing indicated that the exception handling semantics is insufficient to solve problems like *simultaneous exceptions* (Issarny, 2001). In decentralized systems, simultaneous exceptions happen when collaborating agents detect exceptions which are parallel and must be handled. The difficulty is in the determination of the handling order and how to resynchronize the processes to collaborate effectively. Also, agent autonomy increases the difficulty in the collaboration when the agent refuses to cooperate in the management of exceptions detected by others. Thus, EHM in distributed systems must be powerful to deal with the possible collaboration rejection from some agents. MAS are software, so the previous mechanism of exception is still helpful. Autonomy and distribution require new techniques to solve the problems they produce. Particularly, the exception extend isn't limited to the agent level, but also the entire system.

Autonomy is an agent property. This concept is intangible and hard to describe in a formal and explicit manner. Different definitions are suggested related to the application domain but usually perceived as the exclusion of global control (Houhamdi and Athamena, 2011b). In the dictionary, the Formal definition of autonomy is the quality or state of being self-governing; more accurately: The right of self-control. For an artificial agent, autonomy is a more realistic concept.

Definition

Autonomy is the ability to make a decision without support from other agents and to possess a self-control and private information.

The autonomy means that the agent is qualified to assess its inputs and to deliver outputs without support from other agents. Particularly, the agent can determine the situations of collaborations (the environment whereby the agent decides to collaborate with other agents). The properties of owning the control flow and local hidden data are important to guarantee the autonomy: In the absence of these properties, the agent can't ensure that control isn't captured by another agent, even momentarily. The local hidden data includes the agent knowledge and its state; accordingly, the exclusion of this kind of data avoids autonomy, because the agent will be inconsistent. Autonomy and the agent encapsulation (similar to the object encapsulation) are related. Nevertheless, the autonomy certifies a powerful encapsulation concept to the agent, because it can decide dynamically when to allow access to the encapsulated data.

On the other side, MAS society has suggested approaches to defining dependencies between agents. These dependencies are directly connected to autonomy because they essentially permit agents to evaluate their contextual and resource relationships with other agents and therefore to change their behaviors autonomously (Weyns *et al*., 2005). Contextual autonomy represents the degree of agent autonomy toward other agents in a system.

Agent Autonomy has an additional impact on MAS that is important and related to exception handling. It accentuates the agents decoupling and the system modularity. Both characteristics rise from the autonomy definition that guarantees the agents encapsulation. They are essential because they are often required in exception management and fault-tolerance methods. They make the software architecture more robust since the propagation of undesired situations (like errors) doesn't diffuse to the whole system, but only to a small set of modules.

Similar Works

Exception handling studies are conducted under Artificial Intelligence and Software Engineering research. Since MAS belong to these two domains, various explicit results were achieved, both at theory and practical levels. However, the majority of the achievements don't fall in line with the essential requirements to manage exceptions in MAS: Contemporary approaches consider agents as software objects and then apply the programming exception mechanism which is a well-known theory. They don't take into consideration the specific properties of MAS such as *openness, heterogeneity and autonomy*.

The current achievements in distributed systems, software design and previous research in MAS identify additional important concerns to develop a mature exception handling mechanism, reputably the concurrency and dynamic issues in handling. The existing techniques manage the MAS openness and heterogeneity at a certain level, but unfortunately, they can't deal with the autonomy characteristic. The most notable works that approach exception handling in MAS are the *Sentinel Architecture* (Haegg, 1996), the Sentinel-Like Agents (Klein *et al*., 2003), *Commitment Protocols* (Mallya and Singh, 2005) and *SaGE in the Mad-Kit Platform* (Souchon *et al*., 2004).

An unexpected output of this literature review is that there is approximately no tentative to establish an explicit definition of the exception concept in MAS, particularly in the agent research society. Crucial terms are defined in depth, for example, the agent death; however, the exception term resides intuitive. Accordingly, this work suggests an explicit definition of *Agent Exception* and extends the traditional agent execution model to approach the exception concept in MAS in a better way. Regardless the model doesn't address the complete issues related to the agent exception; it determines the basis for eventual studies concerning autonomy propriety.

Exception in MAS requires special mechanisms to assist developers in handling the exception. The proposed model contributes to the current works by defining the *Agent Exception* in the MAS context, preserving the *agent autonomy* and preparing the agent execution model. The suggested approach handles agent exceptions at the agent level, while current research handles the exceptions at the system level. The two methods are complementary in their advantages to MAS. The system level addresses the global exceptions effectively, because of the central or distributed support that coordinates the handling. The agent level addresses the local and global exceptions in a decentralized manner, which is more complex, thus inefficient, however more flexible and powerful in the case of a subset of agents facing exceptional situations. Consequently, the system level improves the system efficiency and the agent level improves the system robustness, mainly because of the agent *autonomy*.

Our approach equips the agent with pertinent capabilities related to exceptional situations and preserves agent characteristics. Existing systems satisfy part of the agent features, but our model addresses the autonomy issue appropriately. The principal model advantage compared to other systems is its robustness and reduction of the developer task, in this manner, the developer will focus on important processing matters.

Methods and Techniques

The Subsumption and BDI model are popular agent frameworks. Still, these models have two weaknesses related to agent exceptions. They don't integrate exception management mechanisms explicitly in the agent execution model and also they don't identify the occurrence of agent exceptions. Exceptions are often treated as programming exceptions and count on the mechanisms of the used language. However, the agent exceptions handling needs to consider the hypothesis of MAS and good practice of Software Engineering asks to isolate clearly the methods for exception management from the methods for the application logic. The purpose of this paper is to develop a new execution model of an agent that incorporates exception handling facilities and the previous separation of methods is materialized.

Agents often perform an iterative execution model, traditionally the *percept-reason-act* cycle. Our proposed model follows the same cycle and expands the *percept* and *act* processes to adequately support the reasoning process when exceptions occur, respecting the agent autonomy concept.

We start by defining the structure of the message, protocol, handler and knowledge of the agents and then we describe the proposed execution model.

Protocol and Handler Models

Message Structure

We denote ACL message as follow (Equation 1):

$$m = (id, source, destimation, action, content, time) \qquad (1)$$

In Equation 1, *m* is a message, *id* manes its protocol, *source* and *destination* identify the sender and recipient, *action* is the performative, *content* describes the message text and *time* is the acquisition time. The FIPA ACL representation can be used, if necessary. If one of the *m* parameters is '-', it implies the parameter is not defined and any value is acceptable.

Handlers and Protocols Structure

Handler and protocol can be expressed by sequence diagrams or by graphs (Houhamdi and Athamena, 2015). We prefer to represent them formally by graphs. They are described as directed trees, where the root represents the initial message and the rest of the tree is formed by applying the relation *R*, specified as follows: If *T* is a directed tree, *L* represents the leaves Kit ($L \subset T$) and *M* the edges kit. The edges represent operations such as send a message in handlers and protocols.

R is non-symmetric, non-reflexive and transitive binary relationship. *T* verifies the following structural properties:

- $\forall m_1 \in M \setminus L, \exists m_2 \in M, m_1 R m_2$
- $\forall m_1 \in M \setminus L, suc_T(m_1) = \{m_2, m_1 R m_2\}$
- $\forall m_1 \in M \setminus \{root\}, \exists m_2 \in M, m_2 R m_1$

The first definition declares that all sent messages have a successor except leafs. $suc_T(m_1)$ represents the successors set for a given edge of *T* in definition two. Definition three states that all sent messages have a predecessor, except the root. In the case where protocol comprehends a loop in its description, the tree specification utilizes the cycles unrolling over the tree branches. Such unrolling action is usual, e.g. Petri nets.

Protocol Representation

The protocol is described by the following algebra on message sending:

$$P :: m \vee end \vee p^* \vee p \mid p \vee p, p \tag{2}$$

In Equation 2, *m* defines the operation of sending the message, the special operation *end* defines the last message that marks the termination of a protocol *p*, *p** means an iterative (0 to many times) sending a message in the protocol, (*p*|*p*) indicates the protocol selection (or) by the agent and (*p*,*p*) signifies the sequence of two protocols execution.

Handler Representation

Handler differs from the protocol in that a handler contains as operation a message sending or another kind of operation private to the agent, for example, modify the private data or actions on protocol (such as an interrupt, resume, terminate). Local operations are treated as silent transitions as τ in the π- calculus, consequently similar notations are used. The set of these operations is M-{τ} and noted as M for short. The formal representation of handler H is defined in Equation 3:

$$H = end_h \vee end_p \vee \tau(?) \vee mg \vee H, H \vee H^* \vee H \mid H \tag{3}$$

The *Handler representation* uses the same semantic of the protocol representation concerning the operators. But, the handler representation deals with operations that are the end_h message to abort handler, the end_p message to abort a protocol, an internal operation $\tau(?)$, the message sending *m*, a sequence, or the selection on handlers. The formula $\tau(?)$ is an adequate notation where the symbol "?" will be replaced by a local operation related to the application, or insert/delete/update data from the agent knowledge base.

Handler paths don't necessarily terminate with the end_p message, indicating the abort of the suspended protocol to execute the handler. However, the message will be transmitted when the handler needs this operation. All tree leaves terminate with the end_h message to abort the handler.

Protocol and Handler Semantics

The analogous syntax of protocol and handler permits to develop a general execution model. We start by describing two sets: *M* is messages Set, *H* is histories Set where $\phi \in H$ (Empty execution). The execution continues based on the acquired message kind and the handler (*h*) and protocol (*p*) state which the agent executes. '*perform*' defines the progress of the agent running the protocol and the handler.

Perform: $M \times H \times H \times H \times H$:

$$(m, H_p, \phi) \rightarrow \begin{cases} (\phi, \phi) & \text{if } m = end \\ (H_p \cup \{m\}, \phi) & \text{if } m \neq end \end{cases} \tag{4}$$

$$(m, H_p, \phi) \rightarrow \begin{cases} (H_p, H_h \cup \{m\}) & \text{if } m \notin \{end_p, end_h\} \\ (H_p, \phi) & \text{if } m = end_h \\ (\phi, H_h \cup \{m\}) & \text{if } m = end_p \end{cases} \tag{5}$$

(m, H_p, ϕ) in Equation 4 describes the protocol *p* execution. The execution history evolves during messages processing (sent and received) and the processing terminates when *end* is obtained, in this case, the protocol history is cleaned out. But (m, H_p, H_h) in Equation 5 represents a handler execution. Consequently, the handler treatment succeeds the protocol execution. When *m* is end_p, the handler starts after the protocol interruption. Finally, when *m* is end_h, the handler processing is completed with success and the protocol execution is restarted.

Knowledge Structure

Agent maintains some data structures to treat its inputs and identify unusual situations from usual ones. These data structures are described in first order predicate logic and all identifiers are unique.

The essential knowledge used by the agents in detection and management of exceptions is *beliefs* (Sun, 2005). *Expectation* is described according to protocol and handler orders: At the end of each step of a sequence execution, the agent expectations are the following probable step in the sequence. This expectation model and pertinent comparison techniques grant the detection of uncommon events and execute the corresponding handler.

For clarity reasons, we present the agent knowledge structures like tables. Every agent possesses four tables. The Pertinence Table, shown in Table 1, assembles filters for the input. Filters are templates of consistent messages. Messages that aren't conforming to the filters are rejected. Thus, filters decline the messages aren't pertinent to the agent. Discarding these messages before any processing is required in open and dynamic context because of the computational cost (Wooldridge, 2009).

Table 1. Pertinence Table

ID	Source	Destination	Action	Content	Time
C_1	-	-	-	-	-
C_2	-	-	-	-	-
-	Ag_1	-	-	-	-

The Beliefs Table, shown in Table 2, contains the agent expectations. Expectations are templates of messages that the agent is awaiting according to the messages order in the protocol. The agent uses its expectations to separate normal states from unusual situations, i.e., unpredicted state.

Table 2 illustrates an agent that expects an offer from Ag_1 about the protocol C_1 before the time T_{offer} (first row in Table 2). The non-reception of this message on time is considered as an exception. For the last row in Table 2, the agent expects to receive any message about C_2 before T_{offer}.

The State Table, shown in Table 3, contains all running and interrupted protocols and handlers implicating the agent and they are represented as a 3-tuple.

Table 3 shows an example of a status table. The protocol C_1 is interrupted in the fifth step of its execution and waiting for the termination of the handler H_1 and the protocol C_2 and the handler H_1 are in running state.

Finally, agent maintains a Handler (Table 4) to associate exception with the corresponding handler. The handler is associated with one or more messages that specify the type of applicability condition of the handler. Also, a message leads to multiple handlers and the agent will select the appropriate one at runtime.

For example, according to the Table 4, the agent identifies a *Delay Notification* at any time the received message matches the message template, i.e., an *Inform* with a predicate that declares a delay. If this message is detected and considered as an exception, the relative *Delay Notification* handler is executed.

Execution Model

The Fig. 2 presents the general execution model of an agent, containing three layers which we will describe in depth.

First Layer

This layer encompasses three processes which are receiving the message, filtering messages and comparing with beliefs: These processes are the elementary steps of the execution model. The received messages are gathered by the agent from its inbox. They are sent to filter out message process which discards the messages that aren't important for the agent by Pertinence Table, relying on its autonomy. The pertinent messages are then matched with the agent *beliefs* in its beliefs table. Figure 3 shows the flowchart of this process.

The message is searched in the Beliefs Table until a match is located, or the table is completely scanned. If an equal entry is located then the output is *expected message expm* ← and *unexpm* ← *null*. Else, when no equal entry is located, then the output is an *unexpected message* and the opposite assignment is executed. In the first case, we activate the *Take- Decision* Process, but we activate the *Select Handler* Process in the second case.

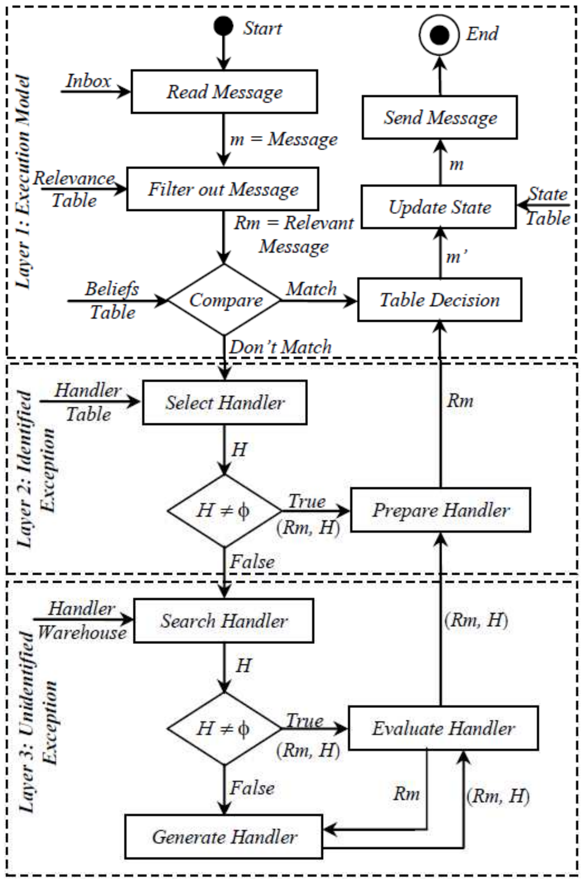


Fig. 2. Agent execution model

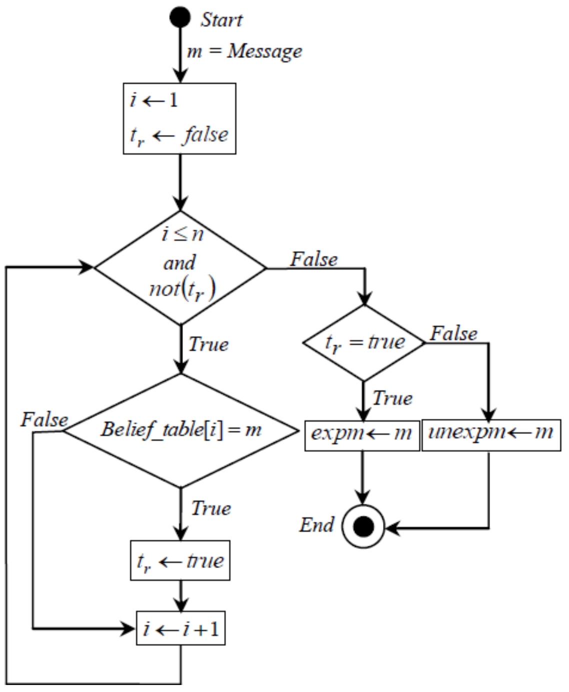


Fig. 3. Beliefs matching flowchart

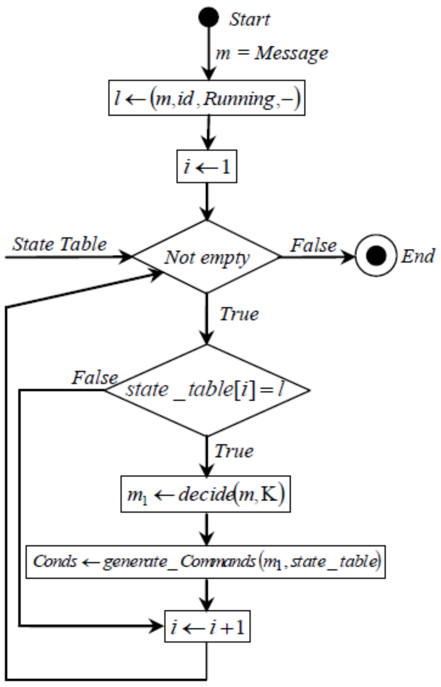


Fig. 4. Take-decision process flowchart

Table 2. Beliefs Table

ID	Source	Destination	Action	Content	Time
C_1	Ag_1	*Self*	*Inform*	*offer*(S_1,-)	$T<T_{offer}$
C_2	-	-	-	-	$T<T_{offer}$

Table 3. State Table

ID	State	Dependency
C_1	*Interrupted*$_5$	H_1
C_2	*Running*	*Null*
H_1	*Running*	*Null*

Table 4. Handler Table

Message	Handler
(–,–,–,Inform, Delay(–,–),–)	*Delay notification*

Take-Decision Process

This process is the rational component of the agent. The message is treated to deduce the following operation of the agent, as described in Fig. 4. Besides this treatment, the *Take-Decision* Process performs continually and it doesn't need an input message to generate an output. This task isn't illustrated in the process because it doesn't contribute to the exception management mechanism. Nevertheless, it is essential since it represents the dynamic component of the agent, indispensable for the agent to start operations.

Based on the actual progress of the protocol related to the received message, the agent *decides* and produces pertinence and beliefs commands in *Generate-Command*, they are dependent on the application domain of the agent that utilize the agent knowledge. However, the *Generate-Command* is supplied with the mechanism, which is independent of the application domain, to generate beliefs and pertinence filters related to the messages expected by the agent in accordance with the operating protocols and handlers. Figure 5 shows the flowchart of *Generate-Command*. It isn't dealing with initiating a protocol or with further modifications that can be done afterward in a domain-dependent manner. Figure 5 processes only data that is independent of the application domain, which are the knowledge tables for controlling the agent execution and the type of the message.

The output message *m* of *Take-Decision* Process is forwarded to the *Generate-Command* to create commands used in the following process *Update-State*. The external loop analyzes every tree of the Agent Execution Table. If *m* is the root of the tree, it implies that the agent has dynamically established protocol (*m* is empty). Two commands are created to modify the Pertinence and Beliefs tables with data related to the new tree. If *m* closes up a tree with either *end* or end_h, the algorithm removes the pertinence and beliefs tuples for the related tree from the corresponding tables. We consider the case *m* closes up a tree with end_p as special because it happens when the handler end h end p end execution terminates the assigned protocol. The *Execution Table* includes a *dependency* attribute that uses to find the entry of the protocol to abort, consequently that two commands are generated to delete the corresponding data in the tables. All remaining cases need the replacement of old beliefs rules by next beliefs expectations. The Pertinence Table doesn't require to be modified because the related protocol is in running state and important to the agent.

The commands are used in the following step *Update-State* to update the pertinence and beliefs filters for the forthcoming cycles and to perform *Send-Message* which is an optional action in the context.

Update-State

Figure 6 illustrates the different steps to update the agent tables. The update sequence is not important in the procedure. This procedure is independent of the application domain since it simply commits the commands on the tabular knowledge, as described in Fig. 6.

Second Layer

This level is related to exception handling mechanism and deals with *Identified Exceptions*, i.e., the agent possesses a handler in *Handler Table* that is appropriate to the revealed exception. The unforeseen message is sent to the *Select Handler* process to find out a handler. The agent identifies unforeseen message whenever an expectation isn't satisfied at the beliefs matching step. The execution flows routed to intermediate level of the agent execution model.

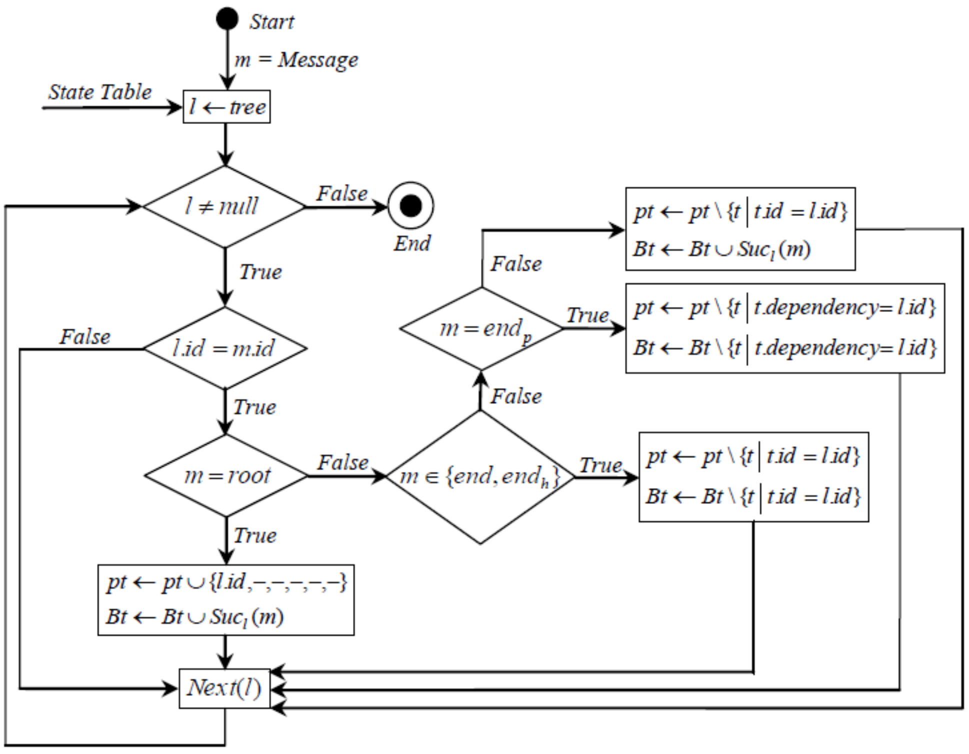

Start
m = Message
State Table
$l \leftarrow tree$
$l \neq null$
False
End
True
False
$l.id = m.id$
True
$m = root$
False
True
$pt \leftarrow pt \cup \{l.id,-,-,-,-,-\}$
$Bt \leftarrow Bt \cup Suc_l(m)$
$m \in \{end, end_h\}$
True
$pt \leftarrow pt \setminus \{t \mid t.id = l.id\}$
$Bt \leftarrow Bt \setminus \{t \mid t.id = l.id\}$
False
$m = end_p$
True
$pt \leftarrow pt \setminus \{t \mid t.dependency = l.id\}$
$Bt \leftarrow Bt \setminus \{t \mid t.dependency = l.id\}$
False
$pt \leftarrow pt \setminus \{t \mid t.id = l.id\}$
$Bt \leftarrow Bt \cup Suc_l(m)$
Next(l)

Fig. 5. Generate-Command flowchart

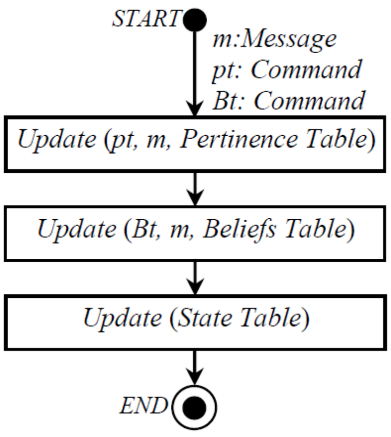

START
m:Message
pt: Command
Bt: Command
Update (pt, m, Pertinence Table)
Update (Bt, m, Beliefs Table)
Update (State Table)
END

Fig. 6. Update-State flowchart

Select Handler, described in Fig. 7, explores the *Handler Table* to find out a convenient *Handler* by comparing the message of each entry of the table with the received message. If they match then a *Handler* is located and returned by the function. If multiple handlers are located, then the *favored* function determines which handler is preferable to the agent, based on its structure and environment. Accordingly, the *favored* function is dependent on the application domain. The favorite functions use metrics to appraise handlers such as the *Handler* complexity.

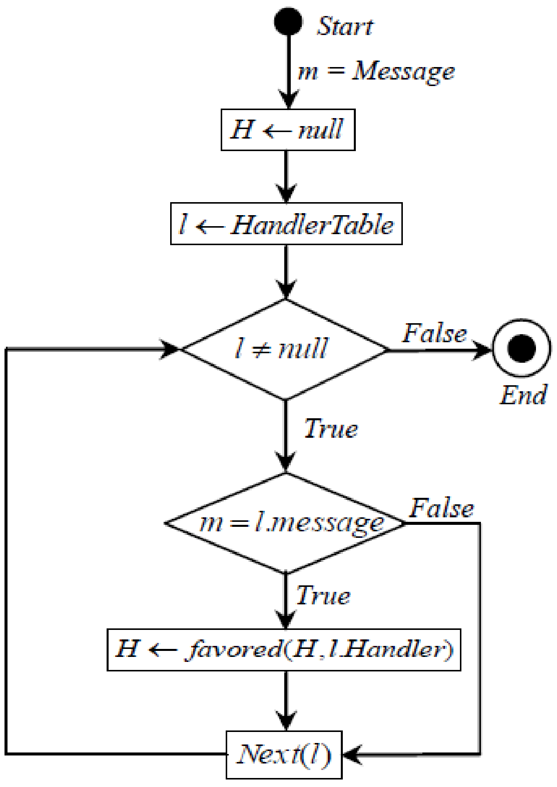


Fig. 7. Select handler flowchart

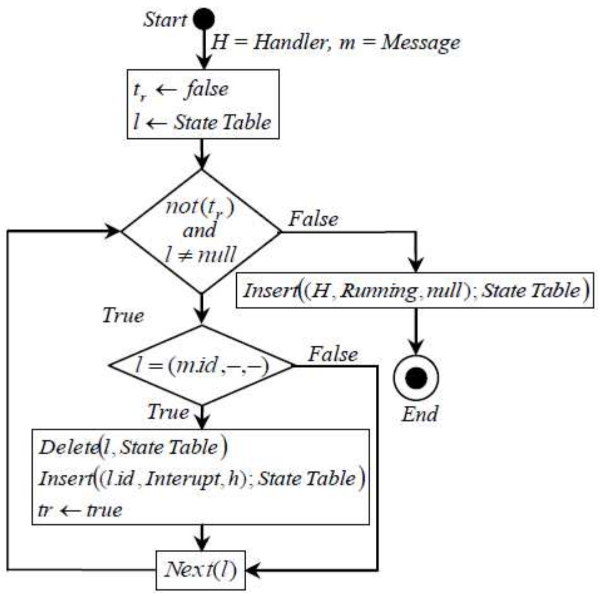


Fig. 8. Prepare Handler Flowchart

Prepare Handling

In case a *Handler* is located, prepare handling procedure is executed (Fig. 8) which suspends the protocol affected by the unexpected message, starts the execution of *Handler* and specifies that the interrupted protocol will be evaluated at the end of the *Handler* execution by inserting a dependency between the protocol and the *Handler* in the status table. Thus the agent decides to continue the interrupted protocol or to abort it. In the end, the procedure sends the message to the *Take-Decision* process, able to deal with the exception, due to the ready handler.

Third Layer

If *Select Handler* procedure fails to find a *Handler*, the agent confronts an *Undefined Exception*, i.e., the agent doesn't possess a *Handler* for this kind of situation. In this case, the agent will collaborate with other agents in the system or a handler warehouse to find a *Handler*. A request is broadcasted to a cooperative agent or such warehouse to try detecting a handler. A successful search returns a *Handler* that will be routed to *Evaluate Handler* function to review the handler efficiency to the current situation, to maintain the autonomy of the agent regarding this outer handler and to update the handling table with the exception type and the *Handler*. Usually, the evaluation function is complex and we consider a simple method: We consider a *Handler* as adequate if it allows the suspended protocol to resume its execution.

Formally, H is adequate if and only if $H_n = P_{it}$

with H is a Handler: $H = (H_i)$, $i \leq n$
and P is a protocol: $P = (P_i)$, $i \leq n$ *Interrupted at statement* P_{it}
and $end_p = end$

Explicitly, the agent relies on outer handler if it directs the execution flow to the earlier state before detection of the exception. However, this easy test doesn't ensure that the handler is adequate for the agent at any stage. Such global technique is application domain depend.

Generate Handler

If the handler search fails or the evaluation is inadequate then the agent tries de create a handler. In the proposed model, this creation unavoidably generates a default handler in case of non-availability of possible. This step is important for the continuation of the execution, to guarantee the nonstop of the model in such situation. The default handler consists in ignoring the received message for certain times after that it admits the failure of the corresponding protocol. The default handler and the expected message are placed in the handler table in the prepare Handler step.

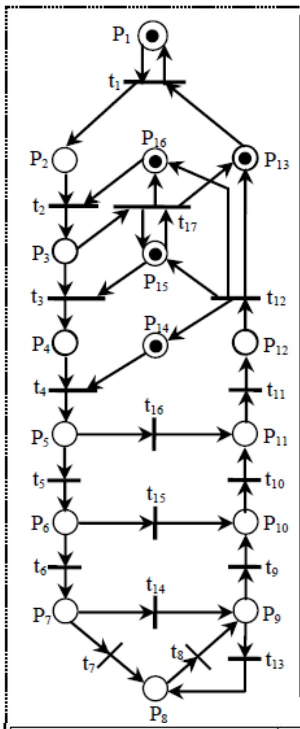
Results and Discussion

The Agent Execution Model (AEM) was described as an architecture that involves particular data structures and procedures. This section aims to examine the model characteristics at the high level of abstraction: Analyze the execution flow of the procedures to test systematic characteristics of the model such as the liveness of processes. Specifically, the model is an iteration of message treatment and creation.

The AEM is modeled as a Colored Petri Net (CPN) and analyzed using automated CPN Tool (Jensen *et al*., 2007). This tool allows simulating and revising the AEM and utilizing a CPN analyzer to test abstract properties of the AEM, Especially fairness, liveness and deadlocks problems during the execution. Figure 9 illustrates the entire AEM modeled as CPN.

Analysis of the Model

The model analysis is conducted by simulations and model analysis (Fig. 10). The simulation generates logfiles as records; also the tool provides animation of the CPN to perceive the marking evolution. Divers executions of the CPN will never terminate even with deadlock or liveness problems. Nevertheless, the executions can't decide if the model is starvation-free and safe. Model analysis permits an exhaustive investigation of the state space.



p_1 : *Start*	
p_2 : *Input*	
p_3 : *Pertinence Test*	
p_4 : *Belief Test*	
p_5 : *Evaluation*	
p_6 : *Selection*	
p_7 : *Search*	
p_8 : *Hand-generation*	
p_9 : *Evaluation*	
p_{10} : *Preparation*	
p_{11} : *Decision Process*	
p_{12} : *Commands*	
p_{13} : *Output*	
p_{14} : *Beliefs*	
p_{15} : *Pertinence*	
p_{16} : *Ignore*	

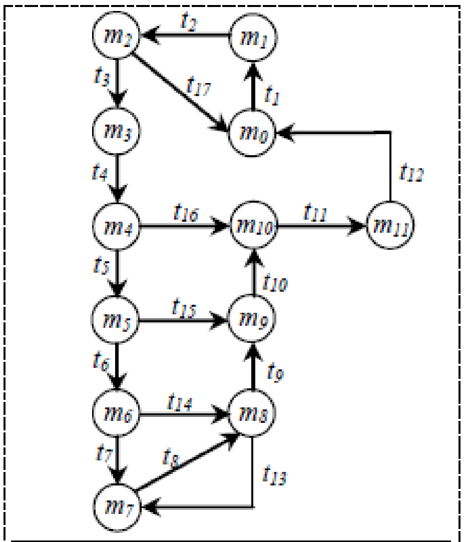
t_1 : *Observation*	t_{10} : *Handler Preparation*
t_2 : *Acquisition*	t_{11} : *Determine*
t_3 : *Pertinence*	t_{12} : *Execution*
t_4 : *Beliefs*	t_{13} : *Generate Other*
t_5 : *Identified Case*	t_{14} : *Handler Search*
t_6 : *Unidentified Case*	t_{15} : *Handler Selection*
t_7 : *Generation Case*	t_{16} : *Expected Case*
t_8 : *Generation Evaluation*	t_{17} : *Ignore*
t_9 : *Handler Evaluation*	

Fig. 9. CPN Formalization of AEM

A deadlock in the AEM signifies that the processing will abort in a not final state, i.e., there is no executable transition. Since the AEM is conceived to run eternally, it doesn't contain a deadlock. However, we have to avoid the deadlocks to prove that the processing always evolves and remains in states determined by the AEM.

On the other hand, liveness problem occurs when a subset of transitions can't be fired at all or from certain execution point. Liveness signifies that portions of the CPN can't be executed anymore. We have to avoid the liveness problems to ensure that the agent preserves its complete services.

Fairness corresponds to a reasonable selection of the agent services, which signifies that any service is ultimately performed if the agent executes eternally. Fairness problem occurs when some transitions are executed remarkably more usually than others.



$$m_0 = (1,0,0,0,0,0,0,0,0,0,0,0,1,1,1,1)$$
$$m_1 = (1,1,0,0,0,0,0,0,0,0,0,0,0,1,1,1)$$
$$m_2 = (1,0,1,0,0,0,0,0,0,0,0,0,0,1,1,0)$$
$$m_3 = (1,0,0,1,0,0,0,0,0,0,0,0,0,1,0,0)$$
$$m_4 = (1,0,0,0,1,0,0,0,0,0,0,0,0,0,0,0)$$
$$m_5 = (1,0,0,0,0,1,0,0,0,0,0,0,0,0,0,0)$$
$$m_6 = (1,0,0,0,0,0,1,0,0,0,0,0,0,0,0,0)$$
$$m_7 = (1,0,0,0,0,0,0,1,0,0,0,0,0,0,0,0)$$
$$m_8 = (1,0,0,0,0,0,0,0,1,0,0,0,0,0,0,0)$$
$$m_9 = (1,0,0,0,0,0,0,0,0,1,0,0,0,0,0,0)$$
$$m_{10} = (1,0,0,0,0,0,0,0,0,0,1,0,0,0,0,0)$$
$$m_{11} = (1,0,0,0,0,0,0,0,0,0,0,1,0,0,0,0)$$

Fig. 10. CPN analysis model

The outcomes of the model checker, illustrated in Table 5, show that the AEM is free from deadlock and liveness problems. This outcome affirms that agent will work eternally without facing difficulties caused by the execution model and it can use all its services during any execution. The second outcome shows that almost transitions are fair. The observation and acquisition transitions are partial. The simulations of the CPN show that the two transitions are executed more usually than the remainder. The message is defined as Token by the Start and Output places, so unavoidably executing the two transitions. The set observation-Acquisition is accordingly executed significantly. On average, they are trigged twice as frequently as others.

Performance Analysis

The experiments were conducted on three versions of systems: No-exception, basic and AEM, to evaluate the distinct approaches quantitatively. Multiple executions are performed as agent-based simulation, one for each a particular approach. Exactly, the executions offer similar functionalities and they vary in the exception handling mechanisms. Set of two different types of experiments were performed to assess the implementation characteristics and the quantitative cost of the AEM. The first experiment type is applied to No-exception (without exception handling) and basic (exception handled using ad hoc mechanisms) versions of the system. The second experiment type is applied to systems with different exception handling mechanisms. Table 6 presents a qualitative comparison of the three systems used in the simulation.

Quantitative Analysis

The experimental results are shown in Table 7 as numerical values.

The maximum indicates the difference between the two experiments. The performance rate of 56% for AEM against the No-exception system means the AEM divides the performance by a factor 1.68. The minimum value is almost unchanged in both experiments. The overhead of the EMS is therefore bounded since the corresponding agents are run at least once in each period.

Table 5. State Space Report for AEM.cpn

Fairness Properties		Liveness Properties	
Commands	*Fair*	Live Transition Instances	*All*
Generation Evaluation	*Fair*	Dead Transition Instances	*None*
Beliefs	*Fair*	Dead Markings	*None*
Generation Case	*Fair*		
Generate Other	*Fair*		
Handler Search	*Fair*		
Handler Evaluation	*Fair*		
Handler Selection	*Fair*		
Handler Preparation	*Fair*		
Identified Case	*Fair*		
Expected Case	*Fair*		
Unidentified Case	*Fair*		
Determine	*Fair*		
Ignore	*Fair*		
Pertinence	*Fair*		
Observation	*Impartial*		
Acquisition	*Impartial*		

Table 6. Systems Comparison

	No-Exception	Plain	AEM
Concerns separation	unavailable	unavailable	available
Autonomy behavior robustness	unavailable	low	Medium to high
Exception handling activities	unavailable	Ad-hoc	Handlers
Exception handling maintenance	unavailable	Low	High

Table 7. Comparison of the Performance Characteristics

			Stable Period		
	Max	Min	Max. Δ	Max	Min
No-exception	3.56	1.00	0.08	1.08	1.00
AEM	2.11	1.00	0.04	1.04	1.00
Ratio	0.56	1.00	2.00	1.05	1.00
Factor	1.68	1.00	0.50	0.96	1.00

Table 8. Complexity Evaluation

	Theoretical complexity	Order (ms)
No-Exception free	N_{DP}	$10^3(2345)$
AEM	$N_{base} = Max(O(n_{pro})N_{DP}$	$Max(O(1),10^3)$
Identified exception	$N = Max(O(n_{pro}), \mathrm{O}(n_k))$	$O(1)$
Total estimation	$N_{base} + N$	10^3
Measured value		$10^3(5081)$

Similarly, for the stable period, the mentioned values are collected after half-time when the system attains a stationary state. The two systems have closer maximal values (4% difference). However, the results show a difference between the minimal and maximal values in the plateau (Δ). Regardless, the clear decrease in the gap between the two systems after a long execution (agents perform a same number of times on average); the AEM possesses a high cost because its Δ value differs by 55%. In the No-exception system, the average execution time is approximately 2345 ms and its deviation is about 530 ms. In the EMS system, the average execution time is around to 5081 ms and the average deviation is closer to 1535 ms which means that the AEM cost is 2.17 more expensive. Since, the standard deviations are similar in the two systems ([22%, 30%] of the mean values), 2.17 is treated as significant. However, it seems that the reduction of this rate is possible by improving the data structures utilized for the agent's knowledge. In our experiments, the data structures used for the agent knowledge are tables and the majority of tasks in the AEM needs costly search through the table. Finally, the results are used for comparing the theoretical complexity and quantitative analysis. Table 8 shows the complexity analysis, where N_{DP} represents the complexity of the No-exception system, $O(n_{pro})$ for *Handler Preparation* and $O(n_k)$ the complexity for *Handler Selection*. The complexity is related to execution-time/cycle; consequently, the evaluation depends on the order of *execution time*.

The measured and theoretical values have equal order. The initial analysis expected that the AEM integration increases the complexity by one order, which isn't that expensive in practice because the experiments depend on the software structure rather than the execution model and also the agent activity was limited to perform a subset of protocols concurrently. The agent's performance analysis shows the AEM impact on the agent execution cycle.

Conclusion

This paper aims to examine the exception management in MAS and to propose an appropriate framework fitting with the heterogeneity and openness proprieties and particularly the autonomy. Our model guarantees the agent autonomy by proposing a new execution model that ensures the agent keeps its control during its execution even in the case of exceptions. The agent decides alone if a situation is an exception or not, so reinforces more its autonomy. The new approach is explicitly defined and the corresponding algorithms are implemented.

Since the agent exception depends on the concept of the unexpected situation; the proposed model defines this concept as a violation of the agent beliefs by interaction protocol. Agent executes the interaction protocols in their actions and it expects the results as stated in the protocols specifications by producing a list of beliefs. Then messages that don't satisfy these beliefs are assumed as an exceptional situation, hence call the exception handling mechanisms.

The model analysis demonstrates it is alive and free of deadlocks for each transition; consequently, the agent reacts to every well-formed input and maintains the availability of its services all the time. The fairness matter proves that the input function cleans up most of the events and may stop the agent execution. This situation isn't a problem in our model and it is considered as the model feature because the introduction of the filtering function allows to the agent to treat only significant events. The filtering function is really important when MAS is used in foreign contexts in which pertinent messages must be determined at the beginning to avoid losing processing time on inutile information. Thus, the agents focus on essential messages and execution iterations are protected from the partial feature of the Observation and Acquisition transitions.

As a perspective of this work, the *nested exceptions* need more investigation and explicit definition. Nested exceptions arise during the management of another exception, consequently necessitating the interruption of the current handler and the execution of a new handler. The proposed execution model supports this function informally. The execution of the handler generates some outputs that must be validated otherwise producing other exception. Thus, the handler will be interrupted and resumed similarly to protocols in the nested exception management. However, the proposed model doesn't investigate the nested exception concepts in depth, because of the resemblance of their handling.

Acknowledgment

We thank the anonymous reviewers for their valuable comments and useful suggestions.

Author's Contributions

Belkacem Athamena: Contributed in all stages of the paper, in the compilation of the suitable scientific materials, in the writing of the manuscript, editing and reviewing.

Zina Houhamdi: Contributed in all stages of the paper, in the compilation of the suitable scientific materials, in the writing of the manuscript, editing and reviewing.

Ethics

The authors confirm that this manuscript has not been published elsewhere and that no ethical issues are involved.

References

Athamena, B. and Z. Houhamdi, 2012. A Petri net based multi-agent system behavioral testing. Modern Applied Sci., 6: 46-46. DOI: 10.5539/mas.v6n3p46

Goodenough, J.B., 1975. Exception handling design issues. SIGPLAN Not., 10: 41-45. DOI: 10.1145/987305.987313

Guessoum, Z., N. Faci and J.P. Briot, 2006. Adaptive Replication of Large-Scale Multi-Agent Systems - Towards a fault-Tolerant Multi-Agent Platform. In: Software Engineering for Multi-Agent Systems IV, Garcia, A., R. Choren, C. Lucena, P. Giorgini and T. Holvoet (Eds.), pp: 238-253.

Guessoum, Z., M. Ziane and N. Faci, 2004. Monitoring and organizational-level adaptation of multi-agent systems. Proceedings of the 3rd International Joint Conference on Autonomous Agents and Multiagent Systems, Jul. 23-23, IEEE Xplore Press, New York, pp: 514-521.

Haegg, S., 1996. A sentinel approach to fault handling in multi-agent systems. Proceedings of the 2nd Australian Workshop on Distributed Artificial Intelligence: Multi-Agent Systems: Methodologies and Applications, Cairns, Australia: Springer-Verlag, pp: 181-195. DOI: 10.1007/BFb0030090

Houhamdi, Z. and B. Athamena, 2011a. Structured integration test suite generation process for multi-agent system. J. Comput. Sci., 7: 690-697. DOI: 10.3844/jcssp.2011.690.697

Houhamdi, Z. and B. Athamena, 2011b. Structured system test suite generation process for multi-agent system. Int. J. Comput. Sci. Eng., 3: 1681-1688.

Houhamdi, Z. and B. Athamena, 2012. A Petri net based agent behavioral testing. Am. J. Applied Sci., 9: 1876-1883. DOI: 10.3844/ajassp.2012.1876.1883

Houhamdi, Z. and B. Athamena, 2015. Ontology-based knowledge management. Int. J. Eng. Technol., 7: 51-62.

Issarny, V., 2001. Concurrent Exception Handling. In: Advances in Exception Handling Techniques, Romanovsky, A., C. Dony, J.L. Knudsen and A. Tripathi (Eds.), Springer, pp: 111-127.

Jensen, K., L.M. Kristensen and L. Wells, 2007. Coloured Petri Nets and CPN Tools for modelling and validation of concurrent systems. Int. J. Software Tools Technol. Transfer, 9: 213-254. DOI: 10.1007/s10009-007-0038-x

Klein, M., J.A. Rodriguez-Aguilar and C. Dellarocas, 2003. Using domain-independent exception handling services to enable robust open multi-agent systems: The case of agent death. Autonomous Agents Multi-Agent Syst., 7: 179-189. DOI: 10.1023/A:1024145408578

Mallya, A.U. and M.P. Singh, 2005. Modeling exceptions via commitment protocols. Proceedings of the 4th International Joint Conference on Autonomous Agents and Multi-agent Systems, Jul. 25-29, ACM, Netherlands, pp: 122-129. DOI: 10.1145/1082473.1082492

Platon, E., S. Honiden and N. Sabouret, 2006. Challenges in exception handling in multi-agent systems. Proceedings of the International Workshop on Software Engineering for Large-Scale Multi-Agent Systems, May 22-23, ACM, Shanghai, China, pp: 45-50. DOI: 10.1145/1138063.1138072

Sichman, J.S., R. Conte, C. Castelfranchi and Y. Demazeau, 1994. A social reasoning mechanism based on dependence networks. Proceedings of the 11th European Conference on Artificial Intelligence, (CAI' 94), pp: 188-192.

Souchon, F., C. Dony, C. Urtado and S. Vauttier, 2004. Improving Exception Handling in Multi-agent Systems. Proceedings of the International Workshop on Software Engineering for Large-Scale Multi-Agent Systems, (MAS' 04), Springer Berlin Heidelberg, pp: 167-188. DOI: 10.1007/978-3-540-24625-1_10

Sun, R., 2005. Cognition and Multi-Agent Interaction: From Cognitive Modeling to Social Simulation. 1st Edn., Cambridge University Press, ISBN-10: 0521839645, pp: 450.

Weiss, G., 2013. Multiagent Systems. 2nd Edn., MIT Press, Cambridge, MA, ISBN-10: 0262313561, pp: 920.

Weyns, D., K. Schelfthout, T. Holvoet and T. Lefever, 2005. Decentralized control of E'GV transportation systems. Proceedings of the 4th International Joint Conference on Autonomous Agents and Multiagent Systems, Jul. 25-29, ACM Press, New York, pp: 67-74. DOI: 10.1145/1082473.1082806

Wooldridge, M., 2009. An Introduction to MultiAgent Systems. 2nd Edn., John Wiley and Sons, Chichester, ISBN-10: 0470519460, pp: 461.