Exception Handling in Goal-Oriented Multi-Agent Systems

Ibrahim Cakirlar, Erdem Eser Ekinci,
Oğuz Dikenelli

Ege University, Department of Computer Engineering,
35100 Bornova, Izmir, Turkey
icakirlar,erdemeserekinci@gmail.com
oguz.dikenelli@ege.edu.tr

Abstract. Cooperative, autonomous and distributed properties of multi-agent systems deduce the dynamic capabilities of multi-agent system applications. On the other hand, these suitable features increase the error proneness of these applications. In this paper, we propose an exception handling approach to make multi-agent system applications more reliable and robust. And also we classify the multi-agent exceptions and implement these levels with our approach on SEAGENT goal-oriented multi-agent development framework[1].

1 Introduction

One of the most important properties that makes a software technology applicable in the industrial settings is robustness. Robust technologies can detect, diagnose and recover from failures and uncertain situations[11,6]. Traditional technologies, such as object oriented paradigm, provide an infrastructure to develop robust software applications that have great degree of robustness maturity. Nevertheless, this problem still stands in front of the multi-agent systems (MASs) as an important obstacle that frustrates MASs to meet the industry.

Nowadays, developed software applications satisfy the requirement of robustness by exception handling mechanism, which are provided within used programming languages[4]. Exception handling is based on the exception concept that is defined as an error occurred during execution flow of a program [4,9,12,20]. Programming languages give chance of recovering exceptions by their exception handling mechanism. When an exception occurs, programming language manages the deviation from the normal execution flow of the program to the handler, which is implemented by developer. To make MASs robust, the exception handling approach must be redesigned according to the general characteristics of the agents.

[1] These research is supported by The Scientific and Technological Research Council of Turkey, Electric Electronic and Information Research Group with 106E008 project.

As known, multi-agent systems consist of cooperative, distributed and autonomous software entities, named agent [19]. These primitive properties prevent agents from direct usage of classical exception handling approach[13]. Distributed but cooperative work of agents mean spreading the program over agent organization and execution flow of the program is managed by separate agents at different times. Handling an exception of this distributed execution flow requires to reconstruct the classical handling approach. Moreover, it must be considered that how the autonomy of agents is effected by implementation of such an distributed handling approach.

In this research we aim to develop an exception handling approach for goal-oriented MASs. In this type of MAS, cooperation and internal intents of agents are modeled with goals. In other words, goals are used to define distributed execution flow, spread over agents, and standalone execution flow of an agent. In our handling approach, we advocate modeling exception handlers with specific goals to recover exceptions. In addition, exception of an agent may not be recovered by that agent and this exception may hamper whole agent organization to achieve organizational goals. For handling such an exception, cooperation of agents may be required. In [18], Lamsweerde and Letier also claim using goals to model exception handlers at the requirement analysis phase of goal-oriented MAS development. But at design time and runtime only goals are not enough for handling exceptions of goal-oriented MASs. Plans are another important artifact of goal-oriented MASs that are used to define detailed internal execution flow of agents to achieve agent goals. So, exceptions of plans should also be considered. In this research, an infrastructure for handling plan exceptions are also defined.

This paragraph defines how the rest of this paper is organized. In section 2, we criticize other exception handling techniques for MASs. In section 3, we determined our approach. In sub-sections of section 3, we classify exceptions of goal-oriented MAS and conveniently, an abstract architecture for handling these types of exceptions is designed. The next section contains development details of this architecture which is implemented within SEAGENT Goal-Oriented Multi-Agent System. Section 5 touches on a case study that shows how efficient the implemented architecture works. Finally we conclude the paper in Section 6.

2 Exception Handling Approaches in Multi-Agent Systems

Up to now, agent researchers have worked on exception handling mechanisms for MASs and contribute to literature with different notions of handling approaches. These approaches can be group in two category. First group has an organizational view, which situated the handling mechanism outside the agent[5,7,17]. So, organizational view approaches bring some organizational entities to MAS and these entities monitor agents by listening their internal events and messages to detect exceptional situations. Approaches subsumed by organizational view can be grouped in two sub-category; centralized and decentralized views. Centralized organizational view, define an exception handling entity in the archi-

tecture of MAS for handling exceptions. Conversely, decentralized sub-category of organizational view spread responsibility of handling exceptions through more than one organizational entity. The other main category of handling approaches have an agent view. Researches grouped in this category similarly listen messages and events of the agent, but they propose a mechanism to handle the exceptions inside the agent[10,15]. In the following paragraphs of this section, we give brief definition about these views according to aforementioned exception handling techniques.

One of the organizational approach is Tripathi and Miller's guardian agent[17]. They purpose an agent, named *guardian*, in a multi agent system that manages exception handling centrally for global exceptions. Guardian agent, dedicated to exception handling, encapsulates defined rules for general exceptions and presents exception handling service to the agent system. When a global exception occurs in MAS, guardian agent perform user defined exception handling rules. For handling the exception, the guardian agents create appropriate exception handler. At the end of handler execution, agent task can continue or terminate its execution as defined.

Another approach for organizational view is Klein and Dellarocas's Exception Handling Service(EHS)[7]. They propose a shared service to provide exception handling to whole multi-agent system. This service is aware of weak points of multi-agent system. EHS follows all the events in the system for exception detection with specialized agents and performs defined rules for correction. At first glance, this approach seems to be in decentralized organizational view but sentinel agents just responsible for exception detection. These agents intend to prepare a knowledge base for calling exception handling service. When an exception occurs, the exception is compared with the predefined candidate exceptions and the general handling policy is determined. To provide this, exception handling service communicates with other agents to define and handle perceived exceptions.

Klein and Dellarocas improve their approach in [8]. In this version, their approach become more decentralized by assigning the responsibility of exception handling to sentinel agents. Every agent in the multi-agent system has a sentinel agent that control agent communication and have ability of using a centralized reliability database shared by all sentinels.

Last of the organizational approaches is Haegg's research[5]. We position Haegg's exception handling approach in decentralized organizational view. Because Haegg proposes special agents, named sentinel, in multi-agent system for exception handling. This approach proposes sentinel agents for each agent in the system. A sentinel controls agent's communication for error detection and recovery. When an exception occurs during the interaction between agents or at the execution cycle, sentinel agent performs error recovery task.

One of the agent view approach is SaGE framework[15]. This framework handles exceptions in three vertical category; service, agent and role. Souchon et al. proposes a layered handling approach depending on Java call stack structure. And also they propose a concerted exception handling mechanism to resolve

errors depending on several agents. When an exception occurs during a shared operation, the operation is terminated to handle exception and participant agents are notified. They have extended the standard Java exception mechanism in order to differentiate higher level exceptions from language levels.

An other approach for agent view is Mallya and Singh's commitment protocols [10]. They propose modeling and handling exceptions via commitment protocols. Exception occurs during the interaction when a protocol is not respected along agent interactions. These exceptions are handled via the definitions of the recovery plan for the exceptional situation. For example alternate protocol or the execution flow of the protocol in an exceptional situation is predefined. They propose an exception handler repository to support handling exceptions dynamically.

Besides all of the suitable features of aforementioned exception handling approaches, these approaches have certain shortcomings. Firstly, we criticize weakness of organizational view approaches. When participant agents count reached huge numbers, its clear that the centralized organizational exception handling techniques requires great resources for specialized agents to handle exceptions of whole MAS. Although decentralizing of organizational exception handling techniques remove the resource problems of centralization agent, but also don't solve problem of performance. Moreover, organizational exception handling techniques are not efficient since handlers don't have agent's internal knowledge. Without this knowledge, an external handler can not find the proper solution for recovering from the exceptional situations because of lacks of the agent's knowledge.

On the other hand, agent view approaches_use resources efficiently by assigning exception handling responsibility to agents. Most of them propose the usage of agent knowledge base to determine exceptional situations. But these handling techniques don't define how autonomy property of agents effects the exception handling process.

In this paper we purpose an approach, which can be classified in agent view approaches, by taking care of shortcomings of aforementioned researches. Our approach is based on goal-oriented MAS and define an architecture how the exception can be handled with goal oriented MAS artifacts**.** Hence, agents are enable to reason for recovering exceptional situations using defined artifacts**.** Although our approach is classified in agent view, thanks to the using goal concept, different exception handling patterns can be implemented by assigning specific exception handling goals to different agents. The variety of exception handing patterns offer different handling perspectives. Following section explains our exception mechanism and handling approach.

3 Exception Handling Mechanism for Goal-oriented Multi-agent Systems

To explicate proposed exception handling approach for MAS, first we have to classify exceptions then we must define an approach to handle the instances of these classifications. Exceptions are generally specified as deviations from the

regular execution flow of a program. In MAS, execution flow with the terms of cooperation and standalone. So, to classify MAS exceptions that cause deviations, entities directly used in different types of execution flows have to be specified.

In a goal-oriented multi-agent system, execution flow of agents (standalone and cooperation) are designed by system goals, agent goals and plans as directly used executable entities. System goals are used to define execution flow of MAS from the cooperation perspective. System goals model the cooperation of participant agents to achieve organizational aims by defining roles, goals, and communication protocols between these goals. System goals also composed of agent goals. Agent goals are used to define internal execution flow of an agent. Although goals are used to specify agent and MAS execution flow, they are not enough for detail execution on their own inside the agent. At this point, plans are required to specify execution details of an agent to achieve agent goals.

Our MAS exception handling approach is inspired with the idea of modeling exception handling mechanism on the simplicity principle[3]. Conveniently to this principle, we claim that each exception should be handled with the executable entities that are same level with the exception occurred in. If it can not be handled in the same level, it hampers the execution flow of one upper level. So, it should be handled in the upper level. In other words, if standalone execution flow of an agent crashes, initially it must be tried to fix internally by agent. If it can not be fixed and causes crash of cooperative execution flow, it has to be compensated cooperatively by participant agents to provide robustness of MAS.

As adverted previously, an agent's internal execution flow is designed with agent goals and plans. In particular, plans define the execution flow more specifically than agent goals and agent goals defines what to do for agent in more generic perspective. According to the our approach, if a plan crashes, it must be handled by another plan. If a plan level exception is not handled, the goal that the agent want to achieve by executing this plan also can not be achieved. Similarly, exceptions of agent goals may be handled via other agent goal(s).

Through agent organization perspective, execution flow of cooperation that is designed by system goals, can be broken down for the reason of irregular behaviors of participant agents. In this circumstance agent organization try to make new decisions to achieve its organizational goals. These exceptions can only be handled via cooperative handling.

3.1 Exception Levels

We categorized exceptions of multi-agent systems due to hierarchy of executable entities defined above. In our approach, according to the executable entities of MAS, exceptions are classified in three levels; plan level, agent level and MAS level exceptions. In [13], Platon also defines a classification for MASs. In his research, there are two classification levels; agent level and code level. This classification is extended by adding one more upper level of MAS level exceptions. These entities and our exception levels are shown in Table 1 and following paragraphs define these levels in detail.

Exception Level	Entity	Standalone	Cooperative
MAS Level	System Goal	-	X
Agent Level	Agent Goal	X	X
Plan Level	Plan	X	-

Table 1. Classification of MAS Exceptions

1. **Plan Level Exceptions:** Agent performs plans to achieve designed agent goals in its life cycle. Along life cycle of an agent, plans can crash for the reason of the logical, implementation, system, communication or knowledge base errors. In detail, a plan may be implemented inadequately according to domain requirements or to programming requirements such as unexpected parameter types or values. System that executes the plans, planner of the agent in our case, may not work properly and cause exceptions. Additionally un-reached or improper messages according to the communication protocol also make a plan crashes. Finally, the knowledge that is used during execution of a plan can cause an exception for the reason of inconsistent operations with the knowledge base. All these types of exceptions are classified in Plan Level Exception for the reason of causing a deviation in the execution flow during plan execution.
2. **Agent Level Exceptions:** Agents try to achieve agent goals corresponding to objectives. An agent may not be able to achieve its goals for several reasons. For example, an agent may decide to achieve an agent goal. Along life cycle of agent knowledge base errors, unhandled plan exceptions and system errors may frustrate the agent to achieve the specified goal. Occurrence of several exceptional situations hamper execution of agent goal. For example occurrence of an uncertain situation in agent beliefs, unreached agent messages, agent system failures generate exceptional situations. All these type of execution errors are classified in Agent Level Exceptions for the reason of causing a deviation in the execution flow during agent goal execution.
3. **Multi-Agent Level Exceptions:** In MAS, agents cooperate to achieve system goals and these system goals are modeled in a specific order. During the cooperation for achieving a system goal, an agent goal can be broken down for the reason of exceptions occurred lower levels. Erroneous execution of one of the participant agent can prevent the achieving of organizational goal. All these type of execution errors are classified in Multi-agent Level Exceptions and represent upper level exceptions that are occurred during this cooperation between agents.

3.2 Abstract Exception Handling Architecture of MAS

To determine exception handling mechanism for agents, the autonomy characteristic should be considered. Agents may have choices to recover exceptions according to their preferences in their knowledge base. So, the exception handling architecture has to be supported with the decision making mechanism of an

agent. To define an exception handling architecture, how a goal-oriented MAS works to make decisions should be determined. In goal-oriented MAS, agents cooperate to achieve a system goal conveniently with their agent goals. So, execution of a system goal is started soon after one of the participant agents, called as initiator agent, desires to fulfill its agent goal that is also designed as sub-goal of the system goal. After deciding which agent goal to take in execution, plans that can achieve the agent goal should be matched. Parallely, other participant agent of the system goal starts to work when the initiator agent's request reach to cooperate. Participant agent also do the same decision making actions with initiator one.

In the life cycle of the agent, agent's planner should take place in exception handling process as well as decision making process. The handling process should spread over three phase of agent cyclic execution model; perception, reasoning and action[14]. In perception phase, planner of the agent should listen the execution of plans and agent goals, which are in execution. After detection of an exception for an agent goal or plan, it should decide what to do for handling this exception. It may find proper plans instead of crashed ones by reasoning. If there are no implemented plans to compensate the execution flow, the agent planner should select a new agent goal instead of one with crashed plan. All these actions are related to adjust the internal execution flow of the agent. If an agent can not succeed to correct the internal execution flow, the cooperation for achieving organizational (system) goal can be frustrated for the reason of one of the participant agent's crash. In this circumstance, the agent with the crashed plan should take in execution of the exceptional system goal.

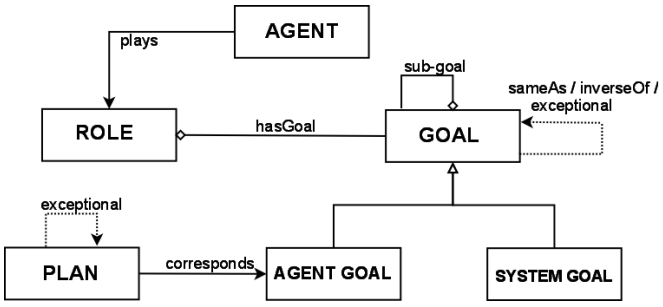

Fig. 1. Executable Entities Meta model

To provide robustness of MAS application within above mentioned exception handling process, developers of the MAS application should define the alternative goals and plans as well as ordinary goals and plans. Figure 1 illustrates a partial view of goal-oriented MAS meta-model that focuses on exception handling. As seen in figure, we extend the general meta-model of MAS by adding special relationships to the semantics of goal and plan concepts. These added special relationships can be listed as *sameAs*, *inverseOf* and *exceptional*. The agent

planner should make decision by watching out these semantic relations when an unexpected state occurs. At such state, the planner firstly should follow existing *exceptional* relations that are defined for recovering agent from exceptional state. If there is no designed exceptional goal or plan for exceptional state, it should try to find alternative goals by querying *sameAs* relations instead of crashed one. Defined *sameAs* goals aims to find a goal with same objective with the crashed one. So the exceptional state of the agent can be disappeared by trying the same objective with another alternative goal. Lastly inverse goals should be queried to roll back the unexpected state. These inverse goals correspond opposite objective with the crashed one for rolling back the effects.

4 Implementation of Proposed MAS Exception Handling Approach

Previously defined abstract exception handling architecture is implemented in SEAGENT Multi-Agent Development Framework[1]. To clarify how the implemented exception handling architecture works, execution life-cycle of SEAGENT planner and used artifacts at run-time must be defined in detail. The primitive properties of SEAGENT are semantic web enability, goal orientation and HTN based plan execution. To support these features, SEAGENT planner[2] uses artifacts, which are defined semantically (with OWL ontologies) and stored in the knowledge base of agents.

In detail, SEAGENT planner uses semantically described goal and HTN ontologies. The HTN ontology resembles to the HTN structure presented in Sycara et. al [16]. In the HTN formalism, plans, which are composed to achieve a predefined agent goal, consist of two types of tasks: complex tasks called as behaviors and primitive tasks called as actions. Each plan has a root task, which is a behavior itself consisting of sub-tasks. On the other hand, actions are primitive tasks, that are directly executable. Nextly, main concepts of goal ontology are system goal and agent goal. System goal specify the organizational aims and agent goals define the intents of agents.

During the decision making process, all SEAGENT planner internal modules use aforementioned concepts defined in goal and HTN ontology. Life-cycle of decision making process is shown in Figure 2. The decision making process starts with an objective that may be sent from outside of the agent or generated by listening internal events. After an objective is reached, then the planner's goal resolution module queries the ontologies in the knowledge base to decide what to do. Soon after deciding on a goal, the plan resolution takes place in the planner and queries the knowledge-base to find proper plan for achieving the found goal. Nextly, the planner generates a graph that represents the execution flow of goals and plans in the reduction phase. After the graph generation, it binds the graph to the current execution process and executes the nodes (represent goals and plans) of graph.

Exceptions that are occurred during the execution of the graph are directed to the planner as recover objective. Initially, plan resolution module queries the

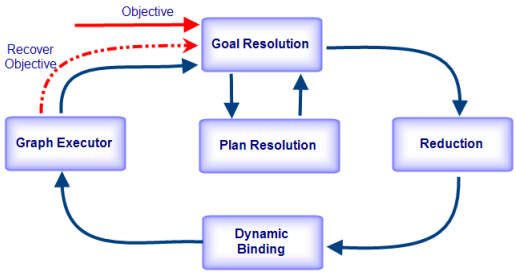

Fig. 2. SEAGENT Planner Decision Making Life-cycle

knowledge base to find the proper plan to recover. Found plan that is defined exceptional in plan ontology is selected for the handling process. The reduction module takes the recovering plan and generates graph and this graph is bound to the execution dynamically. Otherwise, if there is no defined plan for recovering exceptional plan (that means the intended goal by the plan is also crashed), then the goal resolution module seeks an alternative goal for handling the crashed goal, which is aimed to fulfill with the crashed plan. At this time, if the goal resolution module finds an alternative goal for handling, reduction module converts found goal to graph at reduction cycle and deviated inner model replaced with the new one to fulfill exception handling by dynamic binding.

The goal and HTN ontologies are extended to support reasoning ability of SEAGENT planner for exception handling process. To define handling exceptions that are occurred in plans, an exception property is added HTN ontology. Figure 3 shows partial view of HTN ontology that focuses on exceptions. Agent plans consist of tasks, and also tasks consist of sub-tasks. *Exceptional* property makes the definition of handling task between possible agent plan's tasks.

```
<owl:class rdf:about="behaviour">
 <rdfs:subclassof rdf:id="task"/>
</owl:class>
<owl:class rdf:about="action">
   <rdfs:subclassof rdf:id="task"/>
</owl:class>
<owl:objectproperty rdf:about="subtask">
   <rdfs:range rdf:id="task"/>
   <rdfs:domain rdf:id="behaviour"/>
</owl:objectproperty>
<owl:objectproperty rdf:about="exceptional">
   <rdfs:range rdf:id="task"/>
   <rdfs:domain rdf:id="task"/>
</owl:objectproperty>
```

Fig. 3. HTN Ontology

On the other hand, into the SEAGENT goal ontology, *exceptional*, *sameAs* and *inverseOf* properties are added for defining the exceptional handling goals. The ontology is illustrated in Figure 4. *Exceptional* property is used for defining exception handling goals. The *SameAs* property is sub-property of *owl:sameAs*, and used for defining agent goals that achieves same objective with the corrupted one. And *inverseOf* property is sub-property of *owl:inverseOf* and expresses agent goals achieving opposite objective to roll back effects of the crashed one.

```
<owl:Class rdf:about="Goal">
  <rdfs:subClassOf rdf:ID="MasEntity"/>
</owl:Class>
<owl:Class rdf:about="AgentGoal">
  <rdfs:subClassOf rdf:ID="Goal"/>
</owl:Class>
<owl:Class rdf:about="SystemGoal">
  <rdfs:subClassOf rdf:ID="Goal"/>
  <rdf:type rdf:resource="owl-Class"/>
</owl:Class>
<owl:ObjectProperty rdf:about="Exceptional">
  <rdfs:domain rdf:resource="Goal"/>
  <rdfs:range rdf:resource="Goal"/>
  <rdf:type rdf:resource="owl-ObjectProperty"/>
</owl:ObjectProperty>
<owl:ObjectProperty rdf:ID="inverseOf">
  <rdfs:subPropertyOf rdf:resource="inverseOf"/>
</owl:ObjectProperty>
<owl:ObjectProperty rdf:ID="sameAs">
  <rdfs:subPropertyOf rdf:resource="sameAs"/>
</owl:ObjectProperty>
```

Fig. 4. Goal Ontology

Details of exception handling in the graph structure is specified on an example in Figure 5. The example based on a system goal SG1 corresponds an organizational objective of Role1 and Role2. SG1 consist of agent goals AG1 and AG2 related with aforementioned roles. The B1 is the root behaviour of the plan corresponds the agent goal AG1. The right side of the figure illustrates graph model of the B1. The model includes the normal and exceptional execution flow of the B1. The dotted arc between A1 and A3 sub-tasks of B1 emphasize the exceptional execution flow. This link has specific semantic that shows reduction of the behaviour on exceptional cases. Although the link relates two HTN actions in this case, it can be defined on any type of tasks.

When an exception occurs, during the execution of the A1, the execution of whole model, for B1, is aborted to start the recovery process. Plan resolution module selects A3 plan for handling the exception. Reduction module converts

A3 to graph and dynamic linking module replace A2s' graph model with A3s' one. The execution of the recovered graph model for B1 has to be continued from the deviation point to fulfill the handling task.

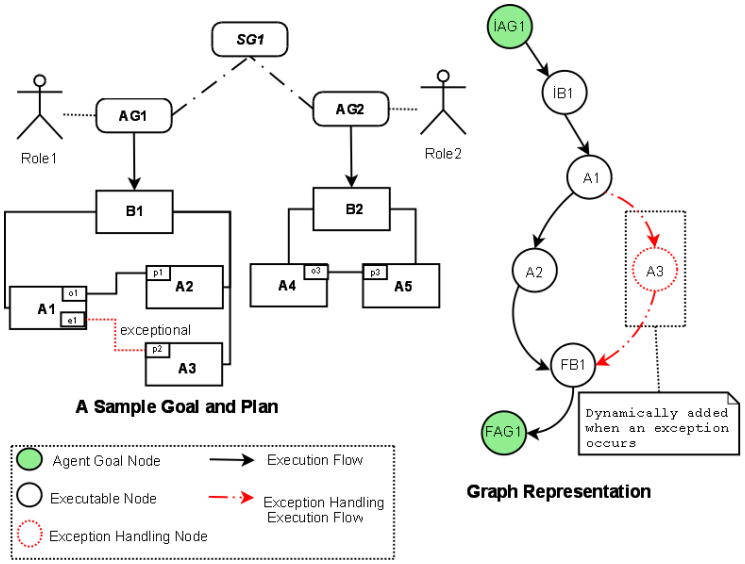

Fig. 5. Planner Executable Entities Sample

The extended planner determines the handling level from lower to higher. If the exception perception is about for an erroneous plan, the plan resolution module queries agent knowledge base to find a proper plan for the recovery of the aborted one. But, if there is no way to fix the crashed plan, that means intended goal crashed, then the goal resolution module queries agent knowledge base to find a proper agent goal to recover the aborted one. This process is executed by reasoning *exceptional*, *sameAs* and *inverseOf* in sequence. The goal resolution module firstly queries agent goals that handles the exception, defined in goal ontologies with *exceptional* property. Afterwards the agent goal for the same objective with crashed one ,defined with *sameAs* property tries to be found. And lastly the agent goal achieves rolling back the effects of crashed one, defined with *inverseOf*, tries to be reasoned. Thereafter, the discovery of finding the proper agent plan for selected one is started. As a result of goal resolution, an agent plan for the found exception handling agent goal, is converted to graph model. The deviated inner model of the aborted agent goal is replaced with the new one.

For example agent goals AG1 and AG2 are sub-goals of the SG1 system goal as shown in Figure 5. When an exception occurs during the execution of the graph model for AG1, execution of the whole graph is aborted to start recovery process. If the goal resolution module can not find a proper agent plan, for

recovering B1, tries to find a proper agent goal instead of AG1. As a result of reasoning process an agent goal that is defined as *exceptional*, *sameAs* or *inverseOf* with AG1 recovers the AG1 from the corruption. Proper agent plan for handling goal is converted to the graph and dynamically replaced with the AG1s' graph model.

At higher level, if the exception perception is about for an exceptional agent goal, goal resolution module queries agent knowledge base to find a proper system goal for the recovery of the aborted one. System goals related with the crashed one is queried with the *exceptional*, *sameAs* or *inverseOf* order previously defined. As a result of reasoning, selected system goal's sub-goals is started to handle the occurred exception during cooperation. Figure 5 depicts the composition of the system goal SG1. If an exception occurs during the execution of the SG1 sub-goal AG1, execution of the whole graph for AG1 is aborted to start recovery process. If goal resolution module can not find a proper agent goal for AG1, tries to find a proper system goal for SG1. As a result of reasoning process a system goal that is defined as exceptional, sameAs or inverseOf with SG1, recovers both the AG1 and AG2 from the corruption. The exception handling sub-goals related with each role of SG1, is about to be started to start the recovery process of SG1.

5 Case Study

An electronic barter application is implemented as a case study with SEAGENT Multi-Agent Framework. In this application, base scenario is achieved by the *Customer*, *Barter* and *Cargo* roles assigned to the agents. Customer agents are responsible for adding, evaluating barter proposals. The Barter agent manages all trades in the system. This agent is responsible for collecting barter proposals, matching proper barter proposals and tracking the bargaining process between Customer agents. After finalization of bargaining, Customer agents send engagement message to the Barter agent. Then, the Barter agent notifies the Cargo agent for transporting barter products between Customer agents. This scenario is completed by the acceptance of all participant agents. The goal model of this scenario is shown in Figure 6.

As seen in the goal model, all main goals of the *ExchangeBarterProducts* system goal is illustrated. ExchangeBarterProducts system goal consists of eight main agent goals; *prepareBarterProposals*, *evaluateBarterProposal*, *evaluateTrnasport, startExchangeOfProducts, tradeBarterProposal, matchBarterProposals, organizeBarterProductsTransportRequest* and *organizeBarterProductsTransport* goals, which are assigned to the *Customer*, *Barter* and *Cargo* roles. To achieve *ExchangeBarterProducts* goal, initially Customer role achieve its *prepareBarterProposal* goal to request trade from Barter role. Barter role collects these proposal with *tradeBarterProposal* and tries to find proper trades with *matchBarterProposals* goals. After finding convenient trades, the Barter agent notifies the related agents that plays the Customer role to start bartering. Then agents playing Customer roles communicates with each other to achieve *evaluateBarterPro-*

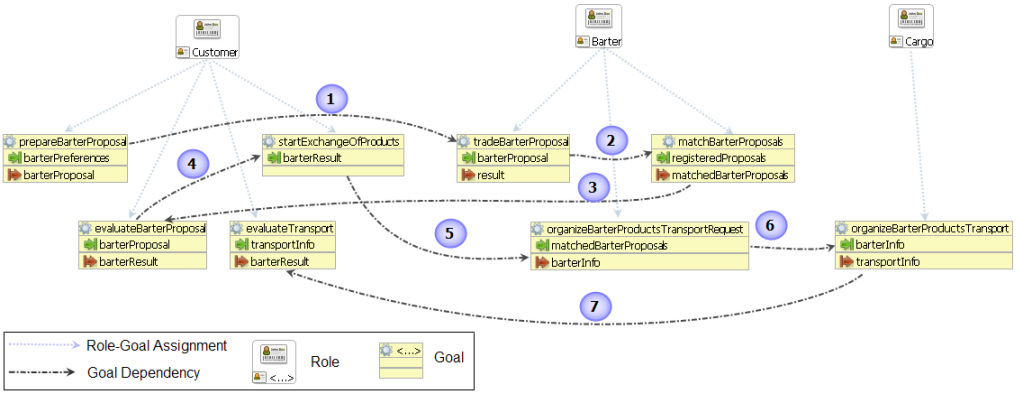

Fig. 6. Case Study Goal Model

posal goals. After they engaged on trading, players of Customer role send engagement messages to the Barter role with *startExchangeOfProducts* goal. Then Barter role starts the execution of *organizeBarterProductsTransportRequest* goal that triggers *organizeBarterProductsTransport* goal of Cargo role. The transport information is notified by all participants and the execution of *ExchangeBarterProducts* system goal is completed by engagement messages of Customer roles.

During trading process, Barter agent matches proper barter proposals and start bargaining between Customer agents. Figure 7 illustrates agent plan that corresponds *evaluateBarterProposal* goal. During evaluation process, existing product database can be unavailable to satisfy the requests. Here we implement an user-defined java exception, named *UnavaibleDatabaseException,* that corrupts the execution of the *BHEvaluateBarterProposal* if database is unavailable. To cause *UnavaibleDatabaseException*, our planner modules do not commit the changes in the knowledgebase that is performed by the crashed task. After this operation over knowledge, the recovery task that handles the occurred exception, *ACHandleDatabaseException*, is dynamically added to the model to provide the robustness of the plan. As a result of execution, database become available by opening the connection to the database. At the end of exception handling process, the plan is recovered from the exceptional situation then agent normally continues executing the plan and accepts or refuses the barter proposal.

Let us assume that the exception occurred during the execution of *evaluateBarterProposal* agent goal is not handled at plan level. During the execution, if the same exception, unavailable database exception, is forced to be thrown, Customer agent tries to handle occurred exception within plan level initially. But it can not find proper plan for recovery then it seeks *exceptional*, *sameAs* or *inverseOf* goal definitions related with *evaluateBarterProposal.* As a result of reasoning *withDrawBarterProposal* agent goal, that is defined as *exceptional* in the goal ontology, is matched to handle occurred exception and this goal is dynamically added to the Customer agent to fulfill exception handling. This agent

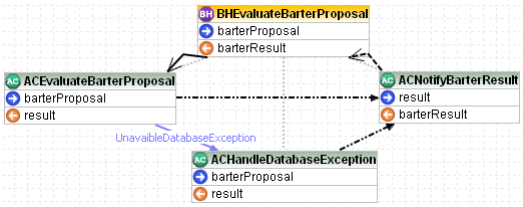

Fig. 7. Plan Level Exception Definition

goal recovers bargaining between customer agents and starts a new objective corresponding withdraw of barter proposal. Consequently bargaining between agents are recovered by withdraw result. Customer role's withdraw request triggers *cancelProposal* goal of Barter role. This agent goal terminates the bargaining process between customer agents and withdraws the proposal from the barter system.

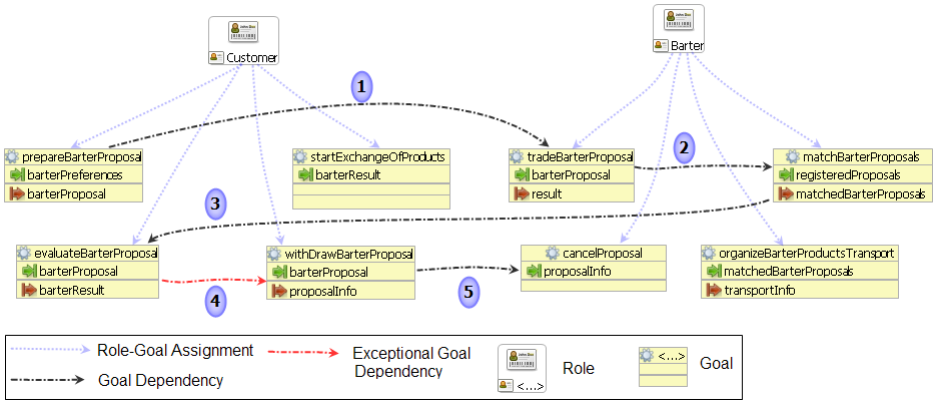

Fig. 8. Agent Goal Level Exception Definition

Lower level multi-agent exceptions can be handled within the internal life cycle. But all of the multi-agent exceptions can not be handled via locally. In our case after the bargaining process between Customer agents, Barter agent determines the transport of the barter products. The Barter agent requests the transportation from the Cargo agent. If an exception occurs during the execution of *organizeBarterProductsTransport*, the barter process between Customer agents can not be completed successfully. Transportation of barter products can not be accomplished due to various reasons such as inconvenient weather condition at the date of the transportation. This exception can only be handled via cooperatively because the crash of the *organizeBarterProductsTransport*, make other agents goals crashed and all sub-goals of *ExchangeBarterProducts* system

goal crash. The reasoning process produces a decision that can be handled via a system goal named *CancelProductExchange*. When Cargo agent percepts this recover objective, it cancels the execution of current goal and starts the execution of proper agent plan , ,related with itself in the definition of *cancelTransport* goal. The execution of *cancelTransport* goal within the Cargo agent, triggers cooperation with participant agents. Afterwards, the Barter agent starts the execution of *cancelBarterMatching* goal to cooperatively cancel the barter between Customer agents' via *cancelBarterRequestEvaluation* goal.

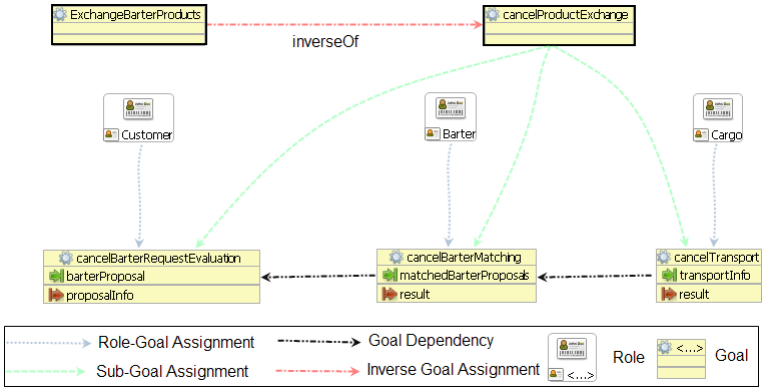

Fig. 9. System Goal Level Exception Handling

Conclusion

Within this paper, we propose an exception handling approach for goal-oriented MASs. Initially, we position our work in the literature. In section two, we classify other researches in two categories; approaches called as organizational view and agent view. Handling techniques of organizational view propose architectural elements. On the other hand, agent view proposes some mechanism to handle exceptions of agents internally.

Our approach provides goal and plan level exception mechanism inside the agent. Hence, it can be classified within the agent view. But, MAS level exception support, via system goal exception goals, makes our approach applicable in organizational level like other organizational view approaches. This makes our approach unique in terms of supporting both views using the goal concept. Additionally, proposed handling mechanism respects to autonomy of agents. The autonomy of the agent is provided within its planner's decision making life-cycle, which consists of three main phases: perception, reasoning and action. We specify our exception handling mechanism with the decision making phases and we extend the agency meta-model of MAS by adding exception semantics.

The approach also implemented in SEAGENT. To show its ability, we implement a case study on barter domain. It is observed with the case study that our approach and implementation support robustness with respect of agents autonomy.

References

1. Oguz Dikenelli. Seagent mas platform development environment. In *AAMAS (Demos)*, pages 1671–1672, 2008.
2. Erdem Eser Ekinci, Ali Murat Tiryaki, Onder Gurcan, and Oguz Dikenelli. A planner infrastructure for semantic web enabled agents. In *OTM Workshops*, volume 4805 of *Lecture Notes in Computer Science*, pages 95–104, Vilamoura, Algarve, Portugal, 2007. Springer.
3. John B. Goodenough. Exception handling design issues. *SIGPLAN Not.*, 10(7):41–45, 1975.
4. John B. Goodenough. Exception handling: issues and a proposed notation. *Commun. ACM*, 18(12):683–696, 1975.
5. Staffan Haegg. A sentinel approach to fault handling in multi-agent systems. In *Revised Papers from the Second Australian Workshop on Distributed Artificial Intelligence*, pages 181–195, London, UK, 1997. Springer-Verlag.
6. Gal A. Kaminka, Milind Tambe, and C. M. Hopper. The role of agent modeling in agent robustness. In *AI meets the real world: Lessons learned (AIMTRW-98)*, 1998.
7. Mark Klein and Chrysanthos Dellarocas. Exception handling in agent systems. In *AGENTS '99: Proceedings of the third annual conference on Autonomous Agents*, pages 62–68, New York, NY, USA, 1999. ACM.
8. Mark Klein, Juan-Antonio Rodriguez-Aguilar, and Chrysanthos Dellarocas. Using domain-independent exception handling services to enable robust open multi-agent systems: The case of agent death. *Autonomous Agents and Multi-Agent Systems*, 7(1-2):179–189, 2003.
9. J. L. Knudsen. Better exception-handling in block-structured systems. *IEEE Softw.*, 4(3):40–49, 1987.
10. Ashok U. Mallya and Munindar P. Singh. Modeling exceptions via commitment protocols. In *AAMAS '05: Proceedings of the fourth international joint conference on Autonomous agents and multiagent systems*, pages 122–129, New York, NY, USA, 2005. ACM.
11. R. A. Maxion and R. T. Olszewski. Improving software robustness with dependability cases. In *FTCS '98: Proceedings of the The Twenty-Eighth Annual International Symposium on Fault-Tolerant Computing*, page 346, Washington, DC, USA, 1998. IEEE Computer Society.
12. Robert Miller and Anand Tripathi. Issues with exception handling in object-oriented systems. *Lecture Notes in Computer Science*, 1241:85–103, 1997.
13. Eric Platon, Nicolas Sabouret, and Shinichi Honiden. An architecture for exception management in multi-agent systems. *International Journal on Agent-Oriented Software Engineering*, 2008.
14. Stuart Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach*, chapter Intelliget Agents, pages 42–45. Prentice-Hall, Englewood Cliffs, NJ, 2nd edition edition, 2003.

15. F. Souchon, C. Dony, C. Urtado, and S. Vauttier. Improving exception handling in multi-agent systems. In *Advances in Software Engineering for Multi-Agent Systems*. Springer-Verlag, 2003.
16. Katia Sycara, M. Williamson, and K. Decker. Unified information and control flow in hierarchical task networks. In *Working Notes of the AAAI-96 workshop 'Theories of Action, Planning, and Control'*, August 1996.
17. Anand Tripathi and Robert Miller. Exception handling in agent-oriented systems. pages 128–146, 2001.
18. Axel van Lamsweerde and Emmanuel Letier. Handling obstacles in goal-oriented requirements engineering. *Software Engineering*, 26(10):978–1005, 2000.
19. Gerhard Weiss, editor. *Multiagent Systems: A Modern Approach to Distributed Artificial Intelligence*. MIT Press, 1999.
20. Shaula Yemini and Daniel M. Berry. A modular verifiable exception handling mechanism. *ACM Trans. Program. Lang. Syst.*, 7(2):214–243, 1985.