Available online at www.sciencedirect.com

ScienceDirect

AASRI Procedia 5 (2013) 126 – 132

AASRI Procedia

www.elsevier.com/locate/procedia

2013 AASRI Conference on Parallel and Distributed Computing Systems

Formal Modeling and Verification of Multi-Agent System Architecture

Ling Yuan[a], Ping Fan[b*]

[a]School of Computer Science, Huazhong University of Science and Technology,Wuhan, China
[b]School of Computer Science, Hubei University of Science and Technology, Hubei, China

Abstract

In a multi-agent system, the multiple distributed intelligent agents interact with each other to solve problems. To guide the development of multi-agent system, the multi-agent system architecture would provide a framework. The specific multi-agent system can be customized from the multi-agent system architecture, which does not need to rewrite the construction. In order to satisfy the failure recovery property of multi-agent system, we propose dependable multi-agent system architecture with fault tolerant mechanisms. The PVS formal language is used to build a system architecture, which can provide common patterns and idioms to the system developers. In order to satisfy the reliability requirements, the powerful PVS theorem prover can be used to analyze the high reliability property of the proposed architecture.

Keywords: Formal Modeling; Formal Verification; Multi-agent; System Architecture; Theorem prover; Fault Tolerance

1. Introduction

A multi-agent system is a system composed of multiple interacting intelligent agents [1]. When it is difficult to solve the problems with an individual agent, we use the multi-agent systems. These problems could

* Corresponding author: Mobile Telephone: 86-18986636933.
E-mail address: fanping1028@126.com.

doi:10.1016/j.aasri.2013.10.068

be online trading, disaster response, and modeling social structures, etc. Since the software architecture is very good choice to ease the development complexity of distributed system [2], [3], we apply the architecture style in the development of multi-agent systems. Since these kinds of multi-agent system applications have common in the construction level, a Multi-Agent System Architecture (MASA) can be very helpful in developing the multi-agent system.

The multi-agent system also tends to be rapidly self-recovering to satisfy reliability requirements. In the proposed MASA, fault tolerant mechanisms [4], [5] are incorporated to provide exception handling capability to the multi-agent systems.

With the well-defined semantics, the formal methods [6], [7] can be used to write the precise specification and process the rigorous verification for the architecture design. Prototype Verification System (PVS) [8], [9] is very powerful in writing formal language and process the verification for complex high reliability systems. The formal language of PVS is easy to learn how to specify the model. The theorem prover of PVS can be controlled by the users by just inputting proof lemmas. These strengths of PVS are very helpful in verifying whether Multi-Agent System Architecture can satisfy the high reliability requirements.

The remainder of this paper is organized as follows. The basic concepts of PVS are described in Section 2. The structure of MASA is illustrated in Section 3. The PVS specification model of MASA is described in Section 4. Section 5 demonstrates whether the model of MASA can satisfy high reliability requirements. In Section 6, we conclude the paper.

2. Background-Prototype Verification System

In the Prototype Verification System (PVS), the formal specification is normally composed of theories. As shown in Fig.1, a list example is used to explain how to write a theory. The *LIST THEORY* has a parameter *Entry*. There are two important functions here. One Function *Leave* specifies how an old *entry* leaves the list. And the function *Join* specifies how the list receives a new *entry*.

```
LIST [Entry. Type+]:THEORY
  BEGIN
  entries: TYPE=[#size: nat, elements: ARRAY[{i|i<size}->Entry]#]
  ents: VAR entries
  nonemptylist?(ents): bool=(size(entries)>0)
  nents: VAR(nonemptylist?)
  join(entry, ents): entries=(#size:=size(ents)+1, elements:= elements(ents)WITH[(size(ents)):=entry]#)
  leave(entry, nents): entries=(#size(nents)-1,)
    entries:=(LAMBDA(j:{ i|i<size(nents)-1}:elements(nents)(j+1))#)
END Queue
```

Fig.1 Example of LIST Theory

In the process of verification, the PVS theorem prover can construct a proof tree, where all nodes should be true. The node of a proof tree can be considered as a *sequent.* And the sequent is composed of antecedents and *consequents*. For example, the *A1* and *A2* can be found in the *antecedent*, and the *B1* and *B2* can be found in the *consequent*, as shown in Fig.2. By entering PVS proof commands, the PVS theorem prover can process the verification.

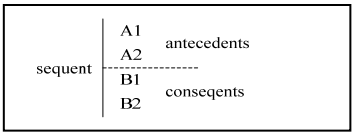

Fig.2 Proof Tree of PVS Theorem Prover

3. Multi-Agent System Architecture

In this section, we describe the architecture style of Multi-Agent System Architecture (MASA). As shown in Fig.3, distributed agents in MASA: Brokering Agent (BA), Query Agent (QA), Query Tool Agent (QTA), Communication Agent (CA), and Monitoring Agent (MA); shared resource: Information Node (INode); fault tolerant agent: FTA; and connectors: BtoQ, QtoB, CtoB, BtoC, QtoQT, QTtoQ, BtoQ, and QtoB.

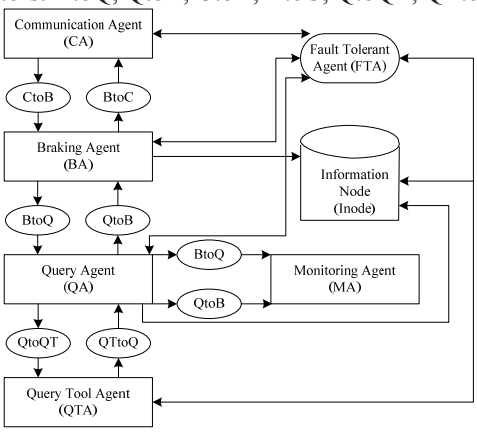

Fig. 3 Multi-Agent System Architecture

The distributed agent can interact with each other to accomplish a job. The information resources are all stored in the Information Node. Two agents communicate with each other by using the connector. When raising exceptions in the system, the fault tolerant agent is responsible for dealing with them.

QA and BAs interact with each other to transfer the interface query. QA is responsible for deciding whether to find other BAs. BA is responsible for deciding which INode has the relevant information. And the QTA is responsible for communication among the agents. MA is in the control position, and its job involves build QA to set up a query, receiving the query results, and analyze the results. CA is responsible for searching potential projects, analyzing them, and choosing the appropriate project.

In the architectural level, the fault tolerant techniques have been integrated to deal with exceptions. When raising an exception, we should use an appropriate handling strategy to deal with it. However, the handling result could be false, and then this exception should be passed to the fault tolerant agent, signalled as global exception. The global exception may affect other interacted agents. Thus, the global exception should inform to the related agents, and these agents have to deal with the informed global exception.

4. PVS Formal Model of MASA

In this section, the fault tolerant agent FTA and distributed agent BA are illustrated as the examples to show the PVS formal model of MASA.I

4.1. Specification Model of BA

The BA is responsible for interacting with agent QA. When exceptions are raised, BA can handle these exceptions. The BA theory is shown in Fig.4.

In the *BA* theory, there are four important functions. One function is *SendData*, which is used to describe how the BA sends the information to QA by using the connectors. How to pass the raised exception to the exception receiving agent is illustrated in the function *ExceptPropagate*. And the function *UniExceptReceive* illustrates how the exception can be received from the exception receiving agent. The last Function *UniExceptHandle* explains whether the BA can handle the received exception successfully. If successfully, the inter state could be normal. Otherwise, the state should indicate that this agent cannot deal with this exception successfully.

```
BA[BASTATE: TYPE+]: THEORY
BEGIN
   IMPORTING FTC
n_states, excepts: set of[BASTATE]
tsin_ports, tsout_ports:set of[PORT]
dc_msg:[PORT ->MSG]
send data[[BASTATE,[PORT->AMSG]] ->[BASTATE]]
except_context:[EXCEPT->EH]
except_handle:[EH->SRSTATE]
Send: TYPE=[#inter_state: BASTATE, checkpoint:SRSTAET,ue_re:SIG#]
sd:VAR Send
SendData(sd):Send=IF member(inter_state(ts), n_states) THEN(#inter_state:=PROJ)_1(senddata
(inter_state(sd),(LAMBDA p:dc_msg(p)))), checkpoint:=inter_state(sd), ue_rec:=0 #)
ExceptProp agate(sr):BASTATE=Ifmember(inter_state(sd),
excepts)AND ue_rec(sd)=0 THEN inter_state(sd)
ftap:VAR FTA
UniExceptReceive(sd, ftap):Send=IF uni_exception=
Except_graph(exceptions(ExceptRec(ftap))) THEN (#inter_state:=uni_exception,
checkpoint:= checkpoint(sd),ue_rec:=1#)
UniExceptHandle(sd):Send=
IF member(inter_state(sd), excepts)AND ue_rec(sd) =1 THEN (IF member(except_handle
(except_context(inter_state(sd))),n_states)THEN
(#inter_state:=except_handle(except_context(inter_state(sd))),checkpoint:= inter_state(sd),
ue_rec:=0#)ELS IF
except_handle(except_context(inter_state(sd)))=Fail
THEN (#inter_state:=Fail, checkpoint:= inter_state(sd), ue_rec:=0#)
END BA
```

Fig.4 BA Theory

4.2. Specification Model of FTA

The specification model of FTA is responsible for illustrate how to handle the raised global exception.

The exception handling strategy could be coordinated error recovery mechanism. The FTA theory is shown in Fig.5.

```
FTA:THEORY
BEGIN
IMPORTING G Generic Type, List[EXCEPTION]
except_graph:[items[EXCEPTION]-> EXCEPTION]
exception: EXCEPTION
FTA:TYPE=[# exceptions: items[EXCEPTION], uni_exception: EXCEPTION#]
fta: VAR FTA
emptyfta: FTA=(#exceptions:=empty,uni_exception:=e#)
ExceptRec(fta):FTA=(# exceptions:=join(exception, exceptions(oc)), uni_exception: exception#)
ExceptGraph(fta):FTA=
 IF exceptions(ExceptRec(fta))/= empty
  THEN (#exceptions:=empty,uni_exception= except_graph(exceptions(ExceptRec(fta)))#)
  ELSE emptyfta
END IF
```

Fig.5 FTA Theory

In the FTA theory, there are two important functions. One function *ExceptRec* illustrates when a global exception has been raised, the exception handling agent should receive this exception. The function *ExceptGraph* specifies that when receiving a set of global exceptions, these exceptions could be resolved into a global exception. And this resolved exception should be handled by the exception handling agent.

5. Verification of MASA using PVS

With the PVS formal model of MASA, the PVS theorem prover can be very helpful in verifying the fault tolerant properties of MASA.

5.1. Raising an Exception

In this section, we illustrate how the MASA deals with a raised exception. This fault tolerant property *inode_pred1* indicates that when the shared resource component *INode* raises an exception *Inodettacked*, the distributed agent (e.g. BA) should receive this raised exception, and handle it, as shown in Fig.6.

Fig.6 shows how the theorem prover of PVS works. When the theorem prover prompts a hint Rule? The users can send a proof command. For example, the proof commands could be *flatten.* This command can be used to resolve the disjunctive connectives, and convert the consequent *inode_pred1* into a sequent.

```
inode_pred1:
|------
{1}(EXISIS(ba:BA):member(inter_state(sd), excepts)AND ue_rec(sd)=0) IMPLIES
    (FORALL(qa:QA),(ftap:FTA):
    Inter_state(UniExceptReceive(dc,ftap))=except_graph(exceptions(ExceptRec(ftap))))
Rule?:(flatten)
tsft_pred1:
{-1}(EXISIS(ba:BA):member(inter_state(sd), excepts)AND ue_rec(sd)=0)
|------
{-1}(FORALL(qa:QA),(ftap:FTA):
Inter_state(UniExceptReceive(dc,ftap))=except_graph(exceptions(ExceptRec(ftap))))
Rule?:
```

Fig.6 Rule Inputting

5.2. Fault Tolerant Property Verification in PVS

As shown below of Fig.7, the proof script of property *inode_pred1* is composed of all the proof commands that the theorem prover of PVS receives from the users when verifying the property. These proof commands are *flatten*, *skolem!*, *asset,* and so on. The explanation of the proof commands can be check up in the PVS tool manual. For example, the *flatten* is used to use a constant to replace the quantified variable in the antecedent.

```
(flatten)(skolem!)
(lemma"Propagate")(assert)(skolem!)(ins tantiate -1("ba!1"))
(assert)(prop)(hide -2)( hide -2)
(lemma"Propagate")(assert)(skolem!)(ins tantiate -1("coc!1"))
(assert)(prop)(hide -3)( hide -3)
(lemma"ExceptPropagate")(instatiate -1("pc!1"))
(replace -1(-1-3)rl)(hide -1)
(lemma"NonEmpty")(instatiate -1("ccp!1" "pc!1"))
(lemma"CCReceive")(instatiate -1("ccp!1"))( assert)
(lemma"ExceptGraph1")(instatiate -1("ccp!1" "qa!1"))
(replace -1(-1-2)rl)(hide -1)
(lemma"UniExcept")(instatiate -1("ccp!1" "dc!1"))(prop)
(lemma" ExceptGraph1")(instatiate -1("ccp!1" "crr!1"))
(replace -1(-1-2)rl)(hide -1)
(lemma"UniExcept")(instatiate -1("ccp!1" "ccr!1"))(prop)
```

Fig.7 Proof Script

In the proof script, besides those primitive lemmas, there are other important induced lemmas. These lemmas can guide the verification process when the verification result is not true. For example, the exception list is not empty is indicated in the *NonEmpty* Lemma. When raising an exception from the *UniExceptReceive* function, how the exception handling component receives the exception is indicated in the *ExceptGraph1*. When verifying a fault tolerant property, the proof script can be entering to the PVS theorem prover directly. The, the fault tolerant property can be verified automatically. Besides this fault tolerant property, we also can verify other properties successfully.

6. Conclusion

In this paper, a Multi-Agent System Architecture (MASA) is proposed to help developing the complex multi-agent systems. In order to make this proposed MASA more precise, we use the formal language of PVS to specify MASA. With the PVS formal model of MASA, the powerful PVS theorem prover can be helpful in verifying the high reliability requirements.

Acknowledgements

The authors would like to thank for the sponsor supported by National Natural Science Fund (No. 61100059). And strong support from National Natural Science Fund of Hubei Province under grant 2012FFB00901, the Science and Technology Research Project of Department of Education of Hubei Province under grant D20132803, Doctoral start Fund of Hubei University of Science and Technology under grant BK1204, the Teaching Research Project of Hubei University of Science and Technology under grant 2012X016B, and the Science and Technology Research Project of Xianning City under grant XNKJ-1203.

References

[1] Michael Wooldridge, “An introduction to MultiAgent systems”, John Wiley & Sons Ltd, 2002.
[2] G.T.Leavens and M.Sitaraman. Foundations of Component-based Systems. Cambridge University Press, 2000.
[3] M.Shaw and D.Garlan, “Software architecture: perspectives on an emerging discipline”, Prentice Hall, 1996.
[4] G.D.Abowd, R.Allen, and D.Garlan. Formalizing Style to Understand Descriptions of Software Architecture. ACM Transactions on Software Engineering and Methodology, 1995, vol4(4): 319-364.
[5] P. Clements, D. Garlan, L. Bass, and J. Stafford. Documenting Software Architectures: Views and Beyond. Pearson Education, 2002.
[6] L.Yuan, J.S.Dong, J.Sun, and H.A.Basit. Generic Fault Tolerant Software Architecture reasoning and customization. IEEE Transactions on Reliability, 2006，55(3):421-435.
[7] J.C.Laprie. Dependability.Basic Concepts and Terminology. Dependable Computing and Fault-Tolerant Systems. Springer-Verlag, 1992.
[8] L.Yuan, J.S.Dong, and J.Sun. Modeling and Customization of Fault Tolerant Architecture using Object-Z/XVCL. In Proceedings of the 13th Asia Pacific Software Engineering Conference(APSEC’06). IEEE Computer Society Press, 2006:209-216.
[9] J. Xu, A. Romanovsky, and R. Campbell. Exception Handling and Resolution in Distributed Object Systems. IEEE Transactions on Parallel and Distributed Systems, 2000, 11(10): 1019-1032.