社団法人 情報処理学会 研究報告
IPSJ SIG Technical Report

2003－ＩＣＳ－133 (27)
2003／9／16

Design and Analysis of Agent Systems by Extended Statecharts *

Heui-Seok Seo†, Tadashi Araragi‡, and Yong Rae Kwon†
†Division of Computer Science, Korea Advanced Institute of Science and Technology, Korea
‡Agent Open Laboratory, NTT Communication Science Laboratories, Nippon Telegraph and Telephone Corporation, Japan
E-mail: †{*hsseo, kwon*}*@salmosa.kaist.ac.kr*, ‡*araragi@cslab.kecl.ntt.co.jp*

Abstract This paper presents a testing method for an agent system against the specification for its behavior described in Statecharts. Here, an agent system means an implementation of an agent application. The testing method controls the execution of the agent system (concurrent system) by using test sequences obtained from the specification and it checks whether the system behaves in accordance with the specification. For specification-based testing, we have extended the design framework of the standard Statecharts so that we can describe the behavior of agents effectively along the concept of Agent UML. Agent systems generally have an enormous number of possible execution sequences because of the agents' autonomy. Our analysis method to produce test sequences makes effective use of the partial order reduction technique. As a result, we can dramatically reduce the number of executions to be tested.
Keywords Agent, Statecharts, Design, Analysis, Concurrency, Testing

1 Introduction

Verification of the agent systems used in the Internet is a crucial subject because independently developed agent systems make use of each other, and bugs in a single agent system may effect a number of other systems. Agent systems are concurrent systems, and applying a testing method for concurrent systems is one of the dominant approaches to verifying them. The main issue of a testing method is how to effectively cover the possible execution sequences of the system.

One of the characteristics of agent systems is the autonomy of agents. That is, an agent can execute its behavior based on its own decision. From the testing point of view, this autonomy is considered to consist of nondeterministic choices in its behavior. As a result, agent systems have an enormous number of possible execution sequences compared with conventional concurrent systems.

In this paper, we introduce a testing method for agent systems that verifies specifications for over all behavior of agents described in Statecharts. To verify agent systems more accurately, we need detailed specifications of the systems, and Statecharts allow us to design the agent behavior more elaborately than message sequence charts, which are mainly used in agent modeling frameworks like Agent UML [1, 9]. The existing Statecharts prove to be insufficient in describing agents' behaviors. Accordingly, we have extended the framework to allow flexible description of agents' behaviors in our testing methods. To solve the problem of an explosion of possible execution sequences in agent systems stated above, we also have introduced a new analysis method based on partial order methods. In this method, reduced reachability graphs are created to efficiently generate substantial test sequences.

The rest of this paper is organized as follows. In Section 2, we briefly describe some related works. Section 3 presents the extended Statecharts and guidelines to describing agent systems with it. In Section 4, we describe the reduced reachability analysis for extended Statecharts. In Section 5, we conclude the paper and present directions for future works.

2 Related Works

In recent years, there has been much research on development methodology for agent systems. For example, Tropos [8], INGENIAS [10] and GAIA [13] have proposed methods for agent-oriented software engineering. As with conventional software development, they use systematic processes to develop agent systems, and the processes cover requirements, designs and implementations. However, these development processes are based not on concrete objects or components but on abstract and logical *roles* of agents, and thus verification of implemented agent systems is out of the scope of such work. In the design phase of the above agent-oriented methodologies, an extended sequence diagram is used as a visual description. In the agent UML [1, 9], the sequence diagrams suitable for describing agent systems are defined. They deal with roles as basic elements and describe interactions between roles. Moreover, they support the extended interactions for concurrent threads: *inclusive or*, *exclusive or* and *and*. However, se-

*This work is supported in part by Nippon Telegraph and Telephone Corporation (NTT)

quence diagrams are scenario-based descriptions, and they present abstract behaviors of systems. Therefore, we propose an elaborate description of agent systems by extending Statecharts, from which we can generate test sequences to verify implemented agent systems.

In previous specification-based testing on Statecharts, test sequences are generated by *flattening methods* [6, 7, 12]. They construct a reachability graph, which presents all possible sequences with all interleavings of concurrent events, from which to generate test sequences. However, the essential limitation of this approach is the *explosion problems* of states, transitions and execution sequences, since a reachability graph presents all possible behaviors of concurrent systems. This limitation becomes more critical in agent systems because they have a huge number of possible execution sequences. Therefore, we propose an efficient analysis technique based on the partial order method, and this technique can construct a reduced reachability graph that omits equivalent sequences.

3 Extended Statecharts for Agent Systems

3.1 Conventional Statecharts

Statecharts [4, 5] provide visual and behavioral specification languages that are suitable for modeling reactive systems. It is also one of the major diagram types in UML for object oriented systems. Although Statecharts in UML focus on the internal behavior of only one object, they are originally defined as an extension of FSM with features such as concurrency, broadcast communications and so on. Therefore, the behaviors of concurrent systems are described by state transitions, and their extended features make it easy to describe them. In this section, we tentatively describe agent systems using conventional Statecharts and then point out the necessity of extending the Statecharts to make them effectively applicable to agent systems.

Agent systems are of course concurrent systems that work through many message exchanges among the autonomously behaving agents. Therefore, we should consider this characteristic in describing agent systems. First, concurrent agents are able to be directly matched with *concurrent* AND components in the traditional Statecharts. Next, a large number of interactions among concurrent agents can be described by using *broadcast communication*, but elaborate considerations are necessary in their description.

The definition of broadcast communication is briefly described as follows: for the transition t with the label (*Event/in_event*), t is executed by *Event* and the internal event *in_event* occurs. Then *in_event* is broadcasted to the entire system, and transitions with the label (*in_event*) are also executed in the current state. However, the interactions

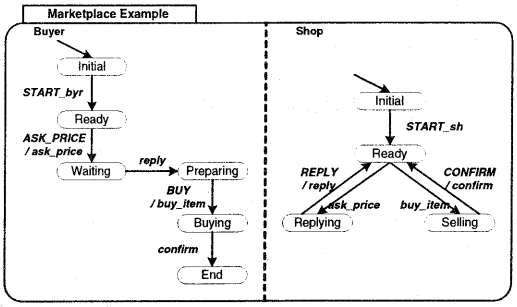

Figure 1. Statecharts specification for a marketplace example: *Buyer* and *Shop* are concurrent components that communicate with each other through internal events.

in agent systems are generally presented by messages, so we propose the following basic form of the labels of transitions. Generally, agent systems have many decision points established not only by their environments but also by their autonomy, and messages are generated as the results of the decisions. Therefore, in Statecharts, the decision point *DP* is managed as an external event and its resultant message *msg* is expressed as an internal event. Consequently, the label of the sending point becomes the basic form (*DP/msg*), and the label of the receiving point becomes the basic form (*msg*).

Figure 1 shows the marketplace example specified with the conventional Statecharts. Two concurrent agents, *Buyer* and *Shop*, are presented as concurrent components. Each component reacts to events such as *START_byr* or *reply*. For understandability, capital words indicate decision points and lower-case words indicate messages. Therefore, *Buyer* and *Shop* can concurrently deal with *START_byr* and *START_sh* because there is no interaction. To show the case of an interaction, let the configuration, which is the overall state of systems, be [*Ready, Ready*]. When *ASK_PRICE* is decided in the decision point, the transition (*ASK_PRICE/ask_price*) in *Buyer* is executed, and then *ask_price* occurs and it is broadcasted to the entire system. In this situation, the transition (*ask_price*) is also executed in *Shop*. Consequently, the configuration changes from [*Ready, Ready*] to [*Waiting, Replying*] as the result of interactions by *ASK_PRICE*.

3.2 Extended Features for Agent Systems

Although Statecharts are able to present both concurrency and communication, it is difficult to use conventional Statecharts for describing agent systems directly because agent systems have different characteristics from reactive systems and object-oriented systems. For example, Figure

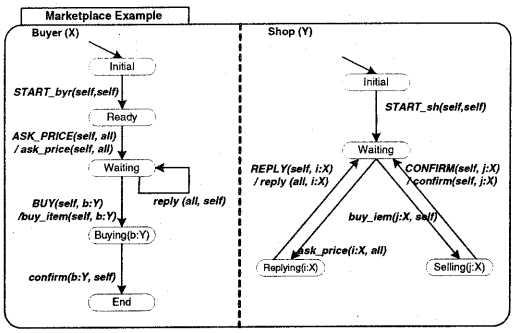

Figure 2. Marketplace system specified with extended Statecharts: they consider the autonomy of agent systems.

1 shows an agent system with only one buyer and only one shop. But in actual agent systems, there can be multiple buyers and shops that have similar behaviors. Therefore, as with other agent-oriented methodologies, we extend Statecharts by focusing on the roles of agents and the various interactions between agents. In this paper, we assume that one agent has only one role and it is implemented with one thread.

Roles are the logical components of agent systems. Agent systems actually consist of diverse concrete agents, but some agents among them can have the same behavior, that is, the same role. In the design phase, it is not required to describe all duplicated agents, and the description is also impossible because we cannot decide the number of concrete agents for one role in advance. Therefore, we define a basic *role-based component* as follows:

- *Role_Name (Agent_ID_Set)*
  - *Role_Name* : name of agent role
  - *Agent_ID_Set* : *id* of the set consists of all concrete agents of the role

From each role, concrete agents belonging to it are instantiated by giving specific identification to the parameters in Role_Name. For example, *Buyer(X)* is one basic component and one role in Figure 2. With this intermediate role, *Buyer(1)* and *Buyer(2)* mean concrete agents with the same behavior. Although the generic charts in the conventional Statecharts also reduce duplication of presentation, they only take care of concrete components and cannot deal with diverse interactions as mentioned below.

Compared with traditional concurrent systems, agent systems have an abundance of interactions. In particular, there are diverse interactions with different relationships between the role of sender and the role of receiver. In agent systems, one role includes several concrete agents. Even if it is the same role, only some agents belonging to it can be related to a given specific interaction. That is, each interaction can be related to *one*, *some* or *all* concrete agents with the same role. For example, when one buyer buys goods in any specific shop, the buyer has no interaction or relation with other shops. Therefore, we define new *event types* and some reserved keywords for communication to be used in this variety of interactions.

- *Event(Senders, Receivers)*
  - *Sender* : specific sending agents
  - *Receiver* : specific receiving agents
- Reserved Keyword
  - **self** : one specific agent (the *id* of itself)
  - **-self** : all agents except **self**
  - **all**: all agents

The *event(a:X, b:Y)* is the fundamental form of events. This means that a specific *agent_a* in *X* sends *event* to a specific *agent_b* in *Y*. In Figure 2, let *X*=1..2 and *Y*=1..2. In this case, the event *ask_price(self, all)* means that *Buyer(1)* and *Buyer(2)*, respectively, send *ask_price(1, all)* and *ask_price(2, all)* to both *Shop(1)* and *Shop(2)*. **-self** is used as follows. Let *Agent_R(Z)* be a role and *Z*=1..4. Then, for the *Agent_R(3)*, *event(i:X, -self)* expresses *event(i:X, 1)*, *event(i:X, 2)* and *event(i:X, 4)*

The final extended feature is the *agent memory* in states. Because agent systems have the dynamic property that agents can be dynamically created and eliminated, the participant agents in communications should often memorize the information about other participants. Moreover, there exist communications between specific agents such as a buyer only buying an item in a specific shop. Therefore, we extend states with the agent *id*: *State(Agent_id)*. For example, *confirm(self, j:X)* in *Shop(Y)* is the reply to *buy_item(j:X, self)*, and *'j:X'* is here set to the same agent as *Buyer(j)* through the state *Selling(j:X)*.

3.3 Description of Basic Communications

We have extended Statecharts to represent autonomy and various types of interactions in agent systems. Now, we describe basic communication types between agents with the extended Statecharts, and the communications are performed by internal events with the extended form. Figure 3 shows the descriptions of 4 communication types according to relationships between senders and receivers.

One-to-one is the fundamental communication in agent systems and means that only one agent with a sender role communicates with only one agent with a receiver role. It is described by assigning a specific sender and a specific

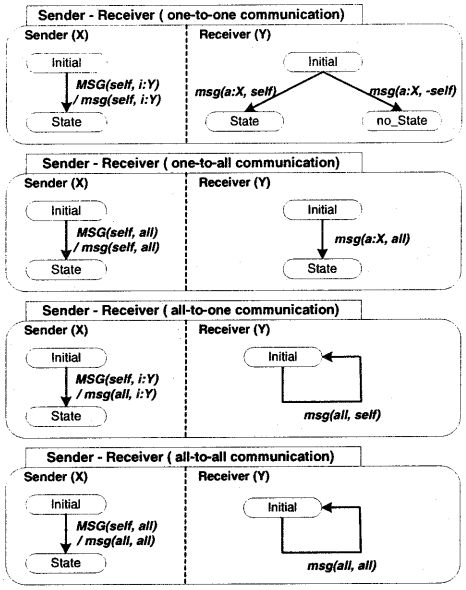

Figure 3. Descriptions for basic communications in agent systems with extended Statecharts

receiver: *msg(i:X, a:Y)*. Moreover, i and a become **self** in a sender agent and a receiver agent, respectively. *msg(i:X, -self)* presents receiving actions of the receiver agents other then self and can be omitted. *One-to-all* means that only one agent with a sender role communicates with all agents with the receiver role. The broadcast communication in Statecharts is suitable for describing it. By definition, *msg(self, all)* is broadcasted from one sender agent, and all receiver agents should react to the event with *msg(i:X, all)*. *All-to-one* means that all sender agents communicate with only one receiver agent. Actually, sender agents behave themselves independently and their concern is a receiver agent. Because a receiver agent should receive all events from all sender agents, its behavior is described with a loop going around itself by *msg(all, self)*. *All-to-all* is described as a combination of *all-to-one* and *one-to-all*.

4 Reduced Reachability Analysis

In the previous section, we propose extended Statecharts that are suitable for describing agent systems. They are based on the conventional Statecharts and keep some essential features of the conventional ones: *state transition mechanism*, *concurrency* and *broadcast communication*. Moreover, we introduce some additional features: *role-based components*, *various event types* and *agent memory* in states. In our reduced reachability analysis for generating test sequences, all of these features are considered, and explosion problems are resolved by adapting the partial order method.

4.1 Overview

The *partial order method* [3] is one of the verification approaches used to avoid explosion problems. Its basic idea is that exploring all interleavings is not necessary and that *independent* events can be left unordered, since, intuitively, independent events have an irrelevant order of occurrence and have the same result by their occurrence. In concurrent programs, event dependency is formally defined as follows:

- Let E be a set of events and $D \subseteq E \times E$ be a binary, reflexive and symmetric relation. Then, D is a *dependency relation* iff all pairs of independent events ($(e_1, e_2) \notin D$) satisfy the following properties for all global states $s \in S$.
  - if e_1 is executable in s and ($s \xrightarrow{e1} s'$), then e_2 is executable in s if and only if e_2 is executable in s'.
  - if e_1 and e_2 are executable in s, there is a unique state s'' such that $s \stackrel{w1}{\Rightarrow} s''$ and $s \stackrel{w2}{\Rightarrow} s''$: $w1 = \langle e_1, e_2 \rangle$ and $w2 = \langle e_2, e_1 \rangle$.

In this definition, ($s \xrightarrow{e} s'$) denotes that the global state changes from s to s' by executing an event e, and ($s \stackrel{w}{\Rightarrow} s''$) denotes that the global state changes from s to s'' by executing an event sequence $w = \langle e_1, e_2, .., e_i \rangle$. By the definition of dependency, the order of executing independent events does not affect overall changes in global states, and so two sequences obtained from each other by permuting adjacent independent events are *equivalent*. While the flattening method causes the explosion problem by generating all possible sequences for each equivalent class, the partial order method can avoid the problem by generating only one sequence for each equivalent class.

As we have seen, the partial order method can avoid the explosion problem by ignoring the other equivalent execution sequences in its equivalent class. Our main task is to introduce a testing method based on the partial order method, that is applicable to extended Statecharts. On the other hand, this approach sometimes loses correctness. In other words, for some detailed requirements, an essential bug can be included in the ignored execution sequences. To deal with this problem, we have introduced into our method a facility for efficiently creating the other executions in the class. We employed our previous result [11] to realize this facility.

Figure 4 shows the overall process for testing agent systems with extended Statecharts. Its basic idea is to classify

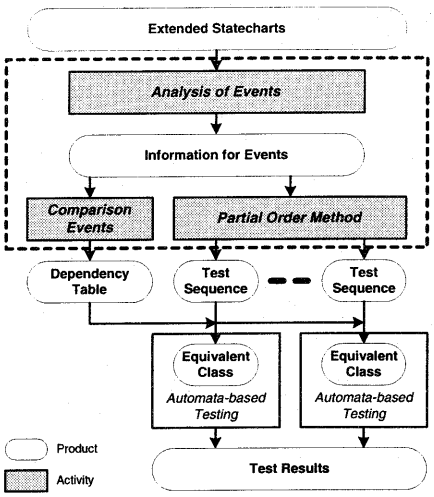

Figure 4. Overall process for testing agent systems with the extended Statecharts: the bounded area presents reduced reachability analysis as described in this paper

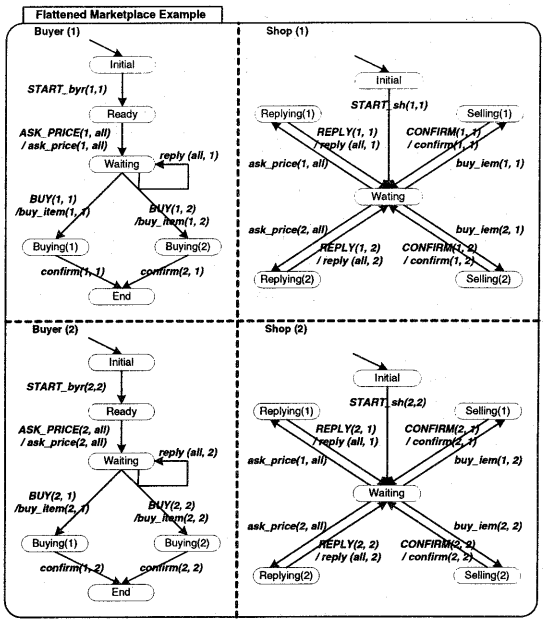

Figure 5. Specification for marketplace example with actual parameters

all possible sequences into equivalent classes. In the front-end part, we construct a reduced reachability graph from the extended Statecharts and generate test sequences. Because the graph omits equivalent sequences, test sequences are representative sequences for equivalent classes. In the back-end part, we generate individual execution sequences of each equivalent class from its representative sequence with the dependency table. In this part, we deal with more detailed requirements that cannot be expressed in terms of the equivalent classes. In this case, the entire testing is partitioned into equivalent classes and performed independently. The essential procedure of the back-end part, called *automata-based testing*, was already discussed in our previous paper [11]. In this paper, we focus on the front-end part, the bounded area in Figure 4.

4.2 Analysis Process

In Figure 4, the front-end part consists of three major procedures: 1) analyzing a given specification in extended Statecharts and getting information on events, 2) constructing the dependency table for all events, which is used for generating equivalent classes, and 3) constructing a reduced reachability graph and generating representative sequences from it. The second and third steps use only information on events, which is the result of the first step.

The first step is the *analysis of events* emerging in specifications, and its objective is to indicate how each external event affects systems. The affections are presented in *PMSs (possible macro steps)* for each external event. A PMS is defined as overall changes in configurations, the global states of systems, by not only a given external event but also internal events caused by it. Therefore, PMSs are generated from specifications: we first find the configurations for an external event and then refine the configurations with the resulting internal events. Now, we show how to generate PMSs in the marketplace example of Figure 2. Before analyzing it, we should set the number of agents because we statically analyze specifications. Here, we consider two buyers (X=1, 2) and two shops (Y=1, 2). We get the following external events by setting **self** to its *id*.

- Buyer (1):
  - START_byr(1,1), ASK_PRICE(1, all), BUY(1, b:Y)
- Buyer (2):
  - START_byr(2,2), ASK_PRICE(2, all), BUY(2, b:Y)
- Shop (1):
  - START_sh(1,1), REPLY(1, i:X), CONFIRM(1, j:X)
- Shop (2):
  - START_sh(2,2), REPLY(2, i:X), CONFIRM(2, j:X)

Source Configuration	Events	Target Configuration
[I , * , * , *]	START_byr(1,1)	[R , * , * , *]
[R , * , W , W]	ASK_PRICE(1,all)	[W , * , R(1) , R(1)]
[W , * , W , *]	BUY(1,1)	[B(1) , * , S(1) , *]
[W , * , * , W]	BUY(1,2)	[B(2) , * , * , S(1)]
[* , I , * , *]	START_byr(2,2)	[* , R , * , *]
[* , R , W , W]	ASK_PRICE(2,all)	[* , W , R(2) , R(2)]
[* , W , W , *]	BUY(2,1)	[* , B(1) , S(2) , *]
[* , W , * , W]	BUY(2,2)	[* , B(2) , * , S(2)]
[* , * , I , *]	START_sh(1,1)	[* , * , W , *]
[W , * , R(1) , *]	REPLY(1,1)	[W , * , W , *]
[* , W , R(2) , *]	REPLY(1,2)	[* , W , W , *]
[B(1) , * , S(1) , *]	CONFIRM(1,1)	[E , * , W , *]
[* , B(1) , S(2) , *]	CONFIRM(1,2)	[E , * , W , *]
[* , * , * , I]	START_sh(2,2)	[* , * , * , W]
[W , * , * , R(1)]	REPLY(2,1)	[W , * , * , W]
[* , W , * , R(2)]	REPLY(2,2)	[* , W , * , W]
[B(2) , * , * , S(1)]	CONFIRM(2,1)	[E , * , * , W]
[* , B(2) , * , S(2)]	CONFIRM(2,2)	[* , E , * , W]

Table 1. All possible macro steps (PMSs) by analyzing the specification of the marketplace example

A1a	1	1	1	1	1	1	1	1	1	1	1	1	1	1
B11	1	1	1	1	1	0	1	1	1	1	1	0	1	0
B12	1	1	1	1	0	1	1	0	1	0	1	1	1	1
A2a	1	1	1	1	1	1	1	1	1	1	1	1	1	1
B21	1	1	0	1	1	1	1	1	1	1	0	1	0	1
B22	1	0	1	1	1	1	0	1	0	1	1	1	1	1
R11	1	1	1	1	1	0	1	1	1	1	1	0	1	0
R12	1	1	0	1	1	1	1	1	1	1	0	1	0	1
C11	1	1	1	1	1	0	1	1	1	1	1	0	1	0
C12	1	1	0	1	1	1	1	1	1	1	0	1	0	1
R21	1	1	1	1	0	1	1	0	1	0	1	1	1	1
R22	1	0	1	1	1	1	0	1	0	1	1	1	1	1
C21	1	1	1	1	0	1	1	0	1	0	1	1	1	1
C22	1	0	1	1	1	1	0	1	0	1	1	1	1	1

Table 2. Dependency table in marketplace example: *START* events are omitted.

In external events, the variables for agent *id* are instantiated exhaustively. Therefore, all possible cases are considered in the analysis phase, and Figure 5 shows the specification with actual values in their parameters. In this example, the configuration consists of the states of four agents: [State of *Buyer (1)*, State of *Buyer (2)*, State of *Shop (1)*, State of *Shop (2)*]. Table 1 shows all PMSs generated from the actual marketplace system: in each source and target configuration,'*' indicates the *'don't care'* state, which can be an arbitrary state of that agent, and each state name is abbreviated to its head character in a capital letter. To appreciate the process of generating PMSs, let's consider the event *ASK_PRICE(1,all)*. By the effect of the external event, the source and target configurations of its PMS become [R,*,*,*] and [W,*,*,*], respectively. But the event also invokes the internal event *ask_price(1,all)* and both *Shop(1)* and *Shop(2)* reacts *ask_price (1,all)*. The internal event changes the configurations from [*,*,W,W] to [*,*,R(1),R(1)], and thus the overall source and target configurations of the PMS are refined as [R,*,W,W] and [W,*,R(1),R(1)].

After obtaining PMSs, the second step is to analyze *event dependency*. Although we introduced the general definition of event dependency, it is impractical to directly check dependency between events using the definition. Therefore, we introduce another definition of dependency under Statecharts which is equal to the general definition.

- Let E be a set of events and $D \subseteq E \times E$ be a binary, reflexive, and symmetric relation. D is *dependency relation* iff all pairs of independent events $((e_1, e_2) \notin D)$ satisfy one of following conditions.
  - e_1 and e_2 don't affect the same component (agent), that is, either e_1 or e_2 has *don't care* state for each component.
  - if both e_1 and e_2 affect any same component, the state in the component is the same state

For example, two PMSs $\langle$ [S1,*], e_1, [S2,*] $\rangle$ and $\langle$ [*,S3], e_2, [*,S4] $\rangle$ satisfy the first condition. In the configuration [S1,S3], both e_1 and e_2 are executable. Also, e_1 is executable in [S1,S4] after executing e_2 and vice-versa. After executing both events, the configurations become [S2,S4]. Therefore, e_1 and e_2 are independent. Two PMSs $\langle$ [S1,S3], e_3, [S2,S3] $\rangle$ and $\langle$ [*,S3], e_4, [*,S3] $\rangle$ satisfy the second condition. In the configuration [S1,S3], they are also independent. The other cases of event pairs cannot satisfy all properties in the general definition. Therefore, we can determine dependency among events by this definition, that is, we can directly find the event dependency by comparing their configurations. Table 2 shows the dependency table for the marketplace example: 'Aia', 'Bij', 'Rij' and 'Cij' are short representations for *ASK_PRICE(i,all)*, *BUY(i,j)*, *REPLY(i,j)* and *CONFIRM(i,j)*. '0' indicates the independent events and '1' indicates the dependent events. To simplify the problem, we remove the events *START*, which are always concurrent and independent.

The final step is to construct a *reduced reachability graph* based on the partial order method and to generate test sequences from the graph, which are representative sequences for equivalent classes. A reduced reachability graph is a subgraph of a reachability graph that presents all possible changes of configurations by events. The reduced graph removes the changes in configurations by equivalent sub-sequences and presents only one interleaving for independent events. For example, let e_1 and e_2 be independent events, that is, $(s_0 \xrightarrow{e1} s_1 \xrightarrow{e2} s_3)$ and $(s_0 \xrightarrow{e2} s_2 \xrightarrow{e1} s_3)$. While a reachability graph presents both changes, a reduced reachability graph presents either sequence, and so either s_1 or s_2 is removed. Therefore, the basic idea of the graph construc-

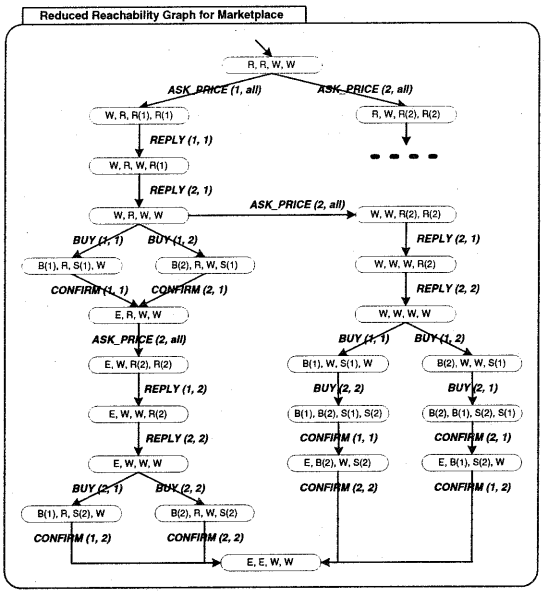

Figure 6. Reduced reachability graph for marketplace example: *START* events and some unusual behaviors are omitted.

tion is to present all partial orders among dependent events. In the above example, e_1 and e_2 are independent, and so e_2 has the same effect on systems before and after execution of e_1. Therefore, a reduced reachability graph presents only e_1 from the configuration s_0 to s_1 and then presents e_2 from s_1 to s_3. If e_1 and e_2 are dependent events, their orders are partial orders and so the graph presents both events from s_0. The algorithm of graph construction starts at an initial configuration and is organized as follows:

- For each configuration,
  - Find all executable events where the source configuration of the PMS includes a current configuration.
  - Repeat the following process with the set of executable events,
    - add any event to the graph and generate a new target configuration.
    - remove its independent events from the set.
- Repeat the above process for new configurations.

By applying the above algorithm to the PMSs in Table 1, the reduced reachability graph of the marketplace system

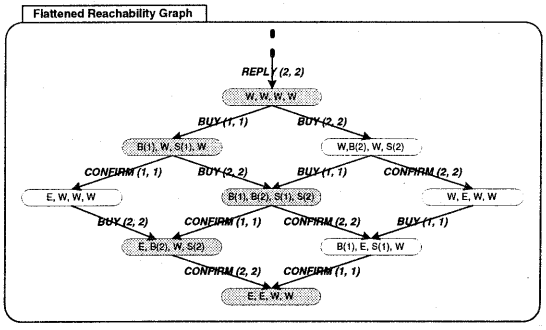

Figure 7. All equivalent reachabilities for the sequence (BUY(1,1), BUY(2,2), CONFIRM(1,1), CONFIRM(2,2)): white states are removed in the reduced reachability graph

is constructed as shown in Figure 6: it omits the behaviors to start with *ASK_PRICE(2, all)* and some unusual behaviors like the execution of *BUY(1, 1)* in [W,R,W,R(1)]. In the graph, the configuration [W,W,W,W] shows good examples for the algorithm. In this configuration, four events, *BUY(1,1)*, *BUY(1,2)*, *BUY(2,1)* and *BUY(2,2)*, are executable, and *BUY(2,2)* and *BUY(2,1)* are independent of *BUY(1,1)* and *BUY(1,2)*, respectively. In this situation, the algorithm only adds the edges for *BUY(1,1)* and *BUY(1,2)* by dependency, and each independent event is added in each following configuration. Therefore, the sub-sequence *<BUY(1,1), BUY(2,2), CONFIRM(1,1), CONFIRM(2,2)>* is representative of six equivalent sequences with the partial orders ($BUY(1,1) \succ CONFIRM(1,1)$) and ($BUY(2,2) \succ CONFIRM(2,2)$). Figure 7 shows all equivalent reachabilities to the sub-sequence *<BUY(1,1), BUY(2,2), CONFIRM(1,1), CONFIRM(2,2)>*. In comparison with a reachability graph, we know that there are four states, eight transitions, and five sub-sequences in the reduced reachability graph. Consequently, while a reachability graph presents 33 states, 49 transitions and 64 sequences, a reduced reachability graph presents 22 states, 25 transitions and 6 sequences. As with these results, our testing approach reduce states, transitions and test sequences in a reduced reachability graph and thus avoids explosion problems.

Now, we compare our testing approach with previous approaches from the viewpoint of test sequences. Table 3 presents the number of equivalent sequences for each test sequence: test sequences are numbered from the leftmost sequence in Figure 6. For example, let's consider *TS5*: *<ASK_PRICE(1,all), REPLY(1,1), REPLY(2,1), ASK_PRICE(2,all), REPLY(1,2), REPLY(2,2), BUY(1,1), BUY(2,2), CONFIRM(1,1), CONFIRM(2,2)>*. This test sequence has two more pairs of independent events than the above sub-sequence with *BUY* and *CONFIRM*: (*RE-*

Test Sequences	Number of Sequences in Equivalent Class
TS1	4 Equivalent Sequences
TS2	4 Equivalent Sequences
TS3	4 Equivalent Sequences
TS4	4 Equivalent Sequences
TS5	24 Equivalent Sequences
TS6	24 Equivalent Sequences

Table 3. Number of equivalent sequences in equivalent classes to test sequences: test sequences are numbered from the leftmost sequence in Figure 6

PLY(1,1), REPLY(2,1)) and (*REPLY(1,2), REPLY(2,2)*). By these independent events, the test sequence has 24 equivalent sequences, including itself, in its equivalent class. The number of test sequences affects costs and efforts in test execution. To verify the behaviors of equivalent classes, previous approaches require 64 test executions in the worst case because they don't consider equivalence relations among test sequences. However, our approach only requires six test executions with representative sequences of equivalent classes. In elaborate testing with more sequences in equivalent classes, the reduced sequences, which are equivalent sequences to a test sequence, are managed in the back-end part of our approach [11]

5 Conclusions and Future Works

In this paper, we proposed a design method for agent systems. In order to acquire an elaborate description, we extended not sequence diagrams but Statecharts. The extended Statecharts retain some features of conventional Statecharts, such as *state transition mechanism*, *concurrency* and *broadcast communication*, and introduce extended features for agent systems, such as *role-based components*, *various event types* and *agent memory* in states. Because it focuses on autonomy as one of the major characteristics of agent systems, our approach is suitable for describing and analyzing agent systems with elaborate behaviors.

Moreover, we proposed reduced reachability analysis for generating test sequences from the extended Statecharts. Because of the autonomy and concurrency in agent systems, conventional analysis methods have limitations such as explosion problems. However, our analysis technique is based on the notion of partial order methods and is adapted to extended Statecharts. Therefore, our approach generates only representative sequences for equivalent classes as test sequences and thus resolves explosion problems.

Our future work is to extend our analysis method to a dynamic version. In the current approach, we generate a given number of agents and analyze them statically. However, in actual agent systems, agents can be created and eliminated dynamically. Therefore, to test such systems we have to deal with this dynamic behavior effectively. One possible approach is to extend the partial order method so that it is applicable to an unbounded number of homogeneous agents. By developing such an extended method, we believe that we can successfully obtain finitely represented control information for dynamic systems.

References

[1] Fipa interaction protocol library specification. *http://www.fipa.org/specs/fipa00025/XC00025E.html*, 2001.

[2] J. O. ed. Agent technology: Green paper, version 1.0. *Agent Working Group OMG Document ec/2000-08-01.*

[3] P. Godefroid. Partial-order methods for the verification of concurrent systems - an approach to the state-explosion problem. *volume 1032 of Lecture Notes in Computer Science(LNCS-1032), Springer*, 1996.

[4] D. Harel. Statecharts: A visual formalism for complex system. *Science of Computer Programming*, 8:231–274, 1987.

[5] D. Harel and A. Naamad. The statemate semantics of statecharts. *ACM Transaction on Software Engineering and Methodology*, 5(4):293–333, Oct. 1996.

[6] Y. G. Kim, H. S. Hong, S. M. Cho, D. H. Bae, and S. D. Cha. Test cases generation from uml state diagrams. *IEE Proceedings Software*, 146(4):187–192, Aug. 1999.

[7] D. Lee and M. Yannakakis. Principles and methods of testing finite state machine - a survey. In *Proceedings of the IEEE*, volume 84, pages 1089–1123, Aug. 1996.

[8] J. Mylopoulos, M. Kolp, and J. Castro. Uml for agent-oriented software development: The tropos proposal. In *Proceedings of the 4th International Conference on the Unfied Modeling Language (UML 2001)*, Oct 2001.

[9] J. Odell, H. V. D. Parunak, and B. Bauer. Extending uml for agents. In *Proceedings of the Agent-Oriented Information Systems Workshop at 17th National Conference in Artificial Intelligence*, pages 3–17, 2000.

[10] J. G. Sanz and R. Fuentes. Agent oriented software engineering with ingenias. In *Proceedings of the 4th Iberoamerican Workshop on Multi-Agent Systems (Iberagent 2002)*, Nov 2002.

[11] H.-S. Seo, I. S. Chung, B. M. Kim, and Y. R. Kwon. An design implementation. In *Proceedings of APSEC01*, pages 3–17, 2001.

[12] B. Y. Tsa, S. Stobart, N. Parrington, and I. Mitchell. An automatic test case generator derived from state-based testing. In *Proceedings of Asia-Pacific Software Engineering Conference(APSEC98)*, pages 270–277, Dec. 1998.

[13] M. Wooldridge, N. R. Jennings, and D. Kinny. The gaia methodology for agent-oriented analysis and design. In *Journal of Autonomous Agents and Multi-Agent Systems*, volume 3(3), pages 285–312, 2000.