Patterns of Agent Interaction Scenarios as Use Case Maps

Edward A. Billard

***Abstract*—A use case map (UCM) presents, in general, an abstract description of a complex system and, as such, is a good candidate for representing scenarios of autonomous agents interacting with other autonomous agents. The "gang of four" design patterns are intended for object-oriented software development but at least eight of the patterns illustrate structure, or architecture, that is appropriate for interacting agents, independent of software development. This study presents these particular patterns in the form of UCMs to describe abstract scenarios of agent interaction. Seven of the patterns attempt to balance the decentralized nature of interacting agents with an organized structure that makes for better, cleaner interactions. An example performance analysis is provided for one of the patterns, illustrating the benefit of an early abstraction of complex agent behavior. The original contribution here is a UCM presentation of the causal paths in agent behavior as suggested by software design patterns.**

***Index Terms*—Cooperative systems, design patterns, multiagent systems.**

I. Introduction

Complex systems of interacting, autonomous, decentralized agents present challenges on many levels. In particular, it is important to capture a high level of abstraction during the early stages of analysis, in which problem discovery and the consideration of alternative scenarios are important. Use case maps (UCMs) [1], [2] are intended as a visual representation to address these particular challenges. The goal is to capture the behavior of components or entities or agents which spans across large, complex systems. These systems tend to be highly concurrent and populated with diverse elements. Although telephone applications were a prime motivator in the development of UCMs, the representation is wide-open in terms of its applicability and surprisingly simple to use. The basic idea is to capture *causal paths cutting across organizational structures* [2]. It is interesting to note that UCMs have not appeared in this journal, even though the technique addresses relevant problems.

This study is meant as an introduction to UCMs as a useful technique for capturing the interaction of agents and, in particular, their organizational or architectural structure. These structures may represent communication or control paths but the presentation is intended to examine the *causal paths* in a general form.

As an inspiration to possible structures, this study considers the "gang of four" design patterns [3] which have had a large impact on the software engineering of object-oriented applications. However, this study has nothing to do with object-oriented development and, instead, considers the patterns literally as prototypes for the interaction of components or agents.

Admittedly, many of the 23 "gang of four" design patterns are highly tied to object-oriented software development but eight of the patterns are good examples of possible structures for interacting agents. For example, consider the observer pattern in which an observer object (read agent) expects a notification whenever an observable object (read agent) undergoes a modification. This is a good example of interacting agents, independent of any software or programming methodology.

The author is with the Department of Math and Computer Science, California State University, Hayward, CA 94542 USA (e-mail: billard@csuhayward.edu).

The purpose of this paper is to show, with UCMs, a high-level abstraction of interacting agent scenarios in the context of some fundamental organizational structures.

The paper is organized as follows. Section II contains an introduction to related work, and Section III has a short tutorial on a subset of UCM notation. Section IV presents the eight design patterns as UCMs and one larger application which combines all eight patterns. Section V shows the results of an example performance analysis of one of the patterns. Section VI summarizes agent applications in the literature which use, or could use, the eight patterns. The Conclusion is presented in Section VII.

II. Related Work

Autonomous, interacting agents have wide-spread applicability, for example, in distributed artificial intelligence [4], knowledge acquisition [5], and conflict detection [6]. Learning automata have been used to model distributed agents in environments with delays in communication and with decisions regarding group formation [7]–[9]. A variation of design patterns in robotics has been examined [10]; these few examples give a flavor of the large literature on autonomous agents.

The seminal work on UCMs is found in [1], [2] and a user community has developed in this area [11]. There exist studies in the context of UCM agents [12], [13] and UCM patterns [14], [15] but these are different than the current study. The agents tend to be components of telephone applications, which happen to be a prime motivator of the development of the UCM methodology. However, these agents do not have the same flavor as some of the more sophisticated artificial intelligence examples found elsewhere. Also, the patterns presented are directed toward more general scenario development than a typical "gang of four" pattern [3].

UCMs have the added benefit that a map can be automatically translated into a layered queueing network (LQN) [16]–[18] which then allows the performance analysis of the component abstraction at an early time during the analysis phase. This can give insight into possible performance issues or problems.

The "gang of four" design patterns [3], and software development patterns in general [19], have had a large impact on the engineering of object-oriented software. Again, this study is not about this type of development but looks to these patterns for prototypical suggestions of agent interaction architectures. This study examines eight of the patterns with UCMs rather than the typical presentation in the unified modeling language (UML) [20]–[22]. Certainly, UML has had a broad impact on software engineering but UML tends to be most useful in presenting details of software design and is not as simple, nor as powerful, as UCMs in terms of abstracting behaviors across components of large systems. A UCM tends to be a higher level of abstraction and, hence, more useful for free-flowing analysis.

III. Short Use Case Map Tutorial

Fig. 1 shows a subset of UCM notation sufficient to understand the maps presented in this paper. The full notation [2] is still relatively straightforward but is not needed for this exposition. (Note that all of the UCMs in this study were created using UCM Navigator [11].) A start point represents the beginning of a causal path and one should imagine this point as a source of “tokens” (with some distribution, say, uniform) that follow along the path. The path may lead through one or more components (read agents) which have responsibilities (read actions). The path then terminates and the direction of the tokens can

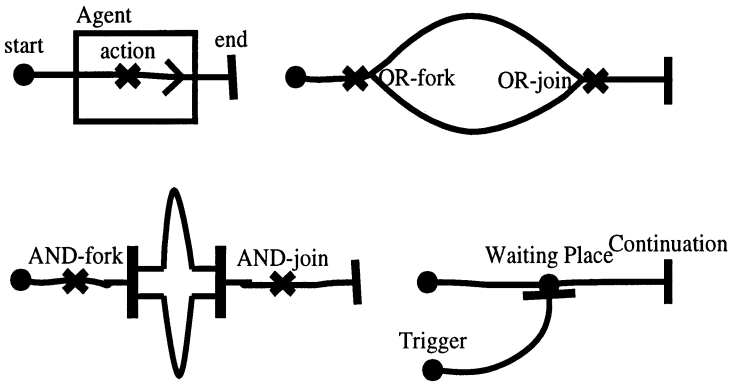

Fig. 1. A subset of UCM notation illustrating a causal path flow through forks and joins, along with possible waiting places.

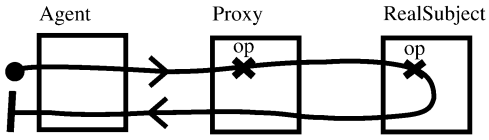

Fig. 2. The proxy pattern has a typical agent interacting with a proxy agent, which shields the real subject agent. Any action on the proxy passes through to the real subject, and a causal action in the return direction is possible.

usually be inferred from the start to end point but an arrow can be used as a guide.

In essence, a path represents a high-level abstraction of the emergent behavior of the agents at runtime that results from the underlying implementations and protocols. The path is intended to reflect this emergent behavior at an early stage of analysis of the problem, even before the underlying implementations are attempted.

The paths become interesting because of AND-forks/joins and OR-forks/joins. An OR-fork represents a probabilistic alternative in which a token may travel one way or another (the figure only shows a two-way split but it can be n-way). Two or more paths may come together in an OR-join. The example shows two paths joining which originated from an OR-fork; this need not be the case. Tokens on different segments can flow into an OR-join, at different times, and then each will travel along the same path.

A token arriving at an AND-fork generates multiple, concurrent tokens, one for each output path. An AND-join requires that a token wait until another token arrives at each of the other joining paths and, again, it is not necessary for the paths, which lead to an AND-join, to have been created by an AND-fork.

Finally, a path may reach a waiting place which halts a token until another token arrives along a trigger path. At this time, the original token moves along the continuation path.

This is sufficient notation for the patterns demonstrated in the next section. The full notation includes, among other things, subcomponents, component pools, stubs, and time-out waiting places [2].

IV. Patterns as Use Case Maps

In this section, eight "gang of four" patterns [3] are presented as UCMs to demonstrate prototypical architectures for interacting agents. The same pattern names, as in [3], will be used here because the names give a good indication of the scenario and architecture. There is some risk to the exposition because the pattern names might suggest that the real issues here concern objects, classes, inheritance, and data; this is not the case. The first example, the proxy pattern in Fig. 2, makes for a good illustration of this point.

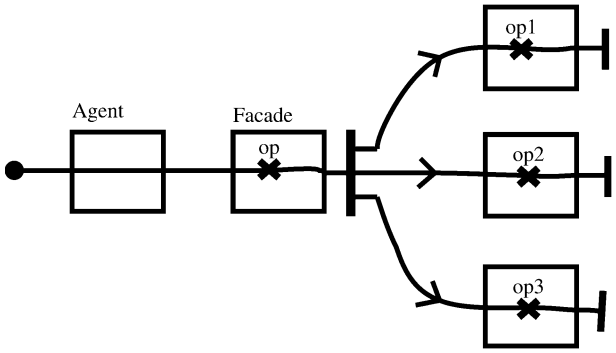

Fig. 3. The facade pattern shields the complexity of interacting with multiple agents, in this case three agents. An agent, outside of the facade, would only need to know how to interact with the facade agent and not the agents hidden behind the facade.

The proxy pattern consists of a typical agent, a proxy agent, and a real subject agent. The typical agent interacts with the proxy as if it were interacting with the real subject, which is hidden by the proxy. The proxy acts as a substitute and has all the same potential for service, support, even nonsupport, as the real subject because it utilizes the real subject to achieve the potential actions. The typical agent does not know, or need to know, that the proxy is not the real subject. This pattern allows the real subject to be distributed somewhere else (an example of a remote proxy pattern) or allows the real subject to maintain isolation from the typical agent for whatever reason.

Now back to the point of object-oriented software development versus interacting agents: the proxy pattern is relevant to both. In the software object world, one object uses a proxy object to access the services of the real subject. In the world of interacting agents, the same is true. The pattern has a natural feel of autonomous, interacting agents. There is something else more subtle in this pattern, and most of those which follow. There are good reasons to distribute and decentralize agents: concurrency, fault-tolerance, redundancy, etc. However, this also increases the difficulty in coordination. The proxy pattern provides for an intermediate to protect direct access to the real subject, hence restructuring the coordination.

The facade pattern, in Fig. 3, is used in software development to hide the complexity of using multiple subsystems. Alternatively, this represents a useful pattern to hide the complexity of interacting with multiple agents. The facade agent knows how to interact with multiple agents hidden behind the facade. An outside agent only needs to know how to interact with the facade agent. This might require only a limited set of interaction protocols, and would certainly limit how much knowledge the outside agent would need to know about the multiple hidden agents. There might be a high degree of difficulty associated

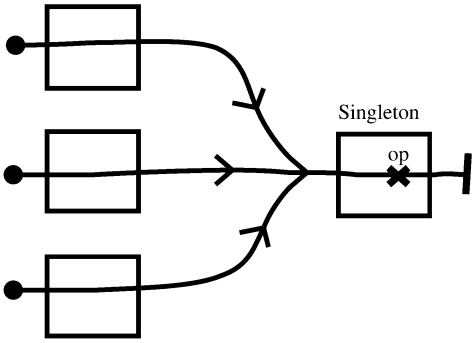

Fig. 4. The singleton pattern focuses attention on a global agent. All other agents know about this particular agent and can interact with this agent.

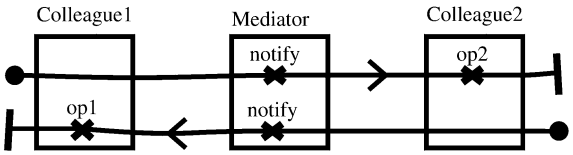

Fig. 5. The mediator pattern uses a centralized mediator to decouple the direct interaction of colleague agents.

with interacting with these multiple agents but only the facade agent needs to know how to perform this interaction.

In the facade pattern, as in the proxy pattern, there is an attempt to alleviate the problems associated with decentralization of agents. In a sense, the facade is a recentralization of the agent interaction. As stated above, decentralization comes with both good and bad aspects; it is important to find the proper balance and the facade is meant to do just that.

The singleton pattern, in Fig. 4, is a very strong attempt at recentralization. In software development, the pattern provides global access to an object, which has only one instantiation. In the agent world, one global agent is known by all other agents and these agents can interact with, and make requests to, this global agent. This would be one technique for resolving multiple conflicting goals. It would also be a good pattern to concentrate knowledge or capabilities.

The mediator pattern, in Fig. 5, is similar to the proxy pattern but the interactions go in two directions, that is, the mediator acts as a go-between for two colleagues or peers. This decouples the direct interaction between these colleagues and centralizes the knowledge of how to interact inside the mediator. In a system with a large number of interacting agents, it is difficult for each agent to know how to interact with all other agents. The pattern provides a mechanism such that an agent just needs to know how to interact with the mediator, usually in a simple and uniform manner. Any changes in existing interaction modes, or any new interaction modes, only need to be reflected in the mediator, with the other agents unconcerned with these changes. This is another example where a good software development design principle—decouple direct object references—is also relevant to the design of agent interactions.

Figs. 6 and 7 illustrate patterns with hierarchical interactions. Note that, in general, hierarchical structures are meant as an organizational mechanism to deal with the decentralization of agents. First, the composite pattern, shows the interactions starting at the root of a hierarchy and spreading downwards, concurrently using AND-forks, toward the leaves. This might be the dissemination of information or knowledge, or the use of some interaction protocol to contribute to a group solution. In this structure, it would be necessary for each agent, or node in the hierarchy, to know a set of "child" agents and to interact with these agents.

Second, the chain of responsibility pattern, in Fig. 7, is also hierarchical but the interaction flows from the leaf agents to the root agent. In this case, an agent only needs to know its "parent" agent. Although this flow of interaction upwards is also concurrent–originating spontaneously at any leaf–there is a difference in the flow with respect to the composite pattern. Each agent is given an opportunity to stop the flow and perform the necessary actions locally. Otherwise, the agent passes along the "problem" to a superior agent, perhaps one that has expanded knowledge, capabilities, and responsibilities. The application, in Fig. 7, shows an alarm system within a building. Sensors within each room might detect fire or motion. A room agent then processes this alarm and may either handle the problem itself—perhaps the fire only requires local treatment with sprinklers—or decides that the supervising agent for the floor needs to be notified. Likewise, the floor agent might handle the problem or pass it to the building supervising agent. Note that this upward flow is represented in the UCM with OR-joins. This pattern is the basis of a performance analysis presented in Section V.

Several of the preceding patterns illustrate how agents might share tasks or jobs. The mediator is a classic pattern which allows agents to work on a shared task but with low coupling between the agents. The composite pattern permits agents at one particular level to work on a shared task, with some control exerted by an agent at the level above. By definition, the chain pattern consists of all agents working on the same task, e.g., processing alarm situations; the only issue is whether an agent can process the alarm locally or needs to notify an agent at the next level.

The observer pattern, in Fig. 8, is a common interaction scenario where one agent, the observer, requests notification by another agent, the observable, in the case the observable agent undergoes some type of modification or interaction. The observer might then initiate some form of interaction–with the observable, or some other, agent-in response to this notification. Note that an observer agent has an AND–fork in the UCM such that the agent can perform other tasks or operations concurrently while the other path is blocked at a waiting place. The path at the waiting place continues only after an observable agent makes a notification. If there are multiple observers–a likely scenario–then this notification is performed concurrently, noted with AND-forks in the UCM. This pattern helps a group of interacting agents maintain contact without having to continually query each other. Only in the event of an important state change does an interaction take place, again a useful scenario to alleviate the complexity of a large number of agents interacting with each other.

Finally, the memento pattern, in Fig. 9, is a typical "snapshot" mechanism to help restore an agent to a previous state, similar to a commit/rollback mechanism in database transaction processing. The originator agent represents the normal operation of the application. The caretaker agent is responsible for initiating the originator, requesting a snapshot of state information at a checkpoint, and monitoring the activities of the originator. In particular, if the caretaker detects a fault, then the state information rolls back to the checkpoint state, and the originator repeats the operations, starting from a consistent state.

Since both caretaker and originator are asynchronous agents, a rendezvous is necessary for the checkpoint, and is achieved with an AND-join: only after a "token" from both the caretaker and the originator arrive at the join, does an outgoing token proceed along the path. The current snapshot of the originator's state is returned to the caretaker. Both the caretaker and the originator proceed concurrently (AND-fork) but later rendezvous again at which time the caretaker informs the originator to either commit (go ahead) or rollback to the previous state and repeat some operations. This pattern illustrates a convenient way to handle faults or other emergency cases which require actions to restart from a consistent state.

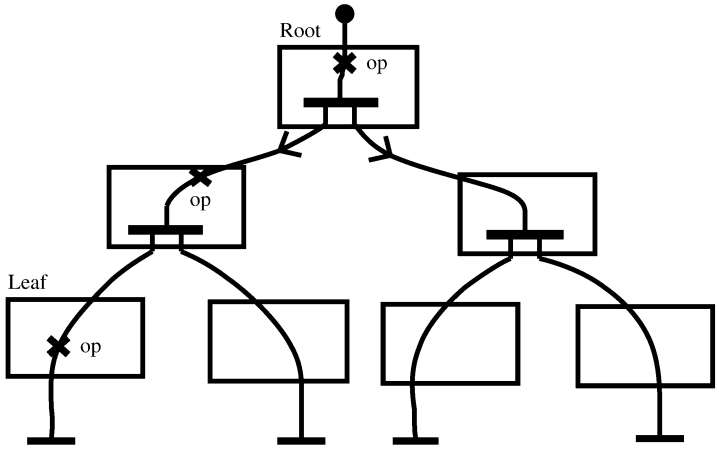

Fig. 6. The composite pattern uses AND-forks to establish a concurrent hierarchical interaction scenario, starting from the root agent.

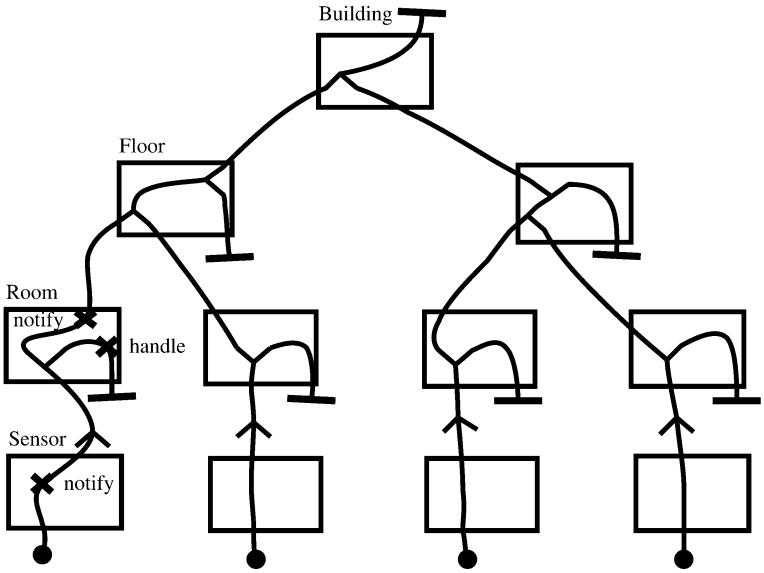

Fig. 7. The chain pattern uses OR-joins to establish a concurrent hierarchical interaction scenario, starting independently from leaf agents. The application is fire or motion detectors within a building. Each agent may handle the alarm itself or pass it along to a supervising agent.

Fig. 10 shows all previous eight patterns connected in a more complex application. Although it is not expected that all eight patterns would arise necessarily in a particular application, the figure does show how these patterns can be used to structure a large set of diverse agents, and the figure shows that a UCM can abstract the complex scenarios of interacting agents.

An initial interaction with a facade causes an interaction with both an agent and a proxy for a real subject. This, in turn, initiates the concurrent interaction of leaves of a chain hierarchy. Some of the interactions, based on the probabilistic OR-forks, will be handled by an agent locally, the others will move along to the supervising root agent. This particular agent has the double role of also being observable. In the case that a notification makes it to the root of the chain, this important state change is broadcast to all observers, which, in this case, is a single observer playing the additional role of a root of a composite set of agents. The causal path of behavior then moves downward to the leaves which then join together to interact with a global singleton agent, which also serves as a caretaker to an originator. The usual commit or rollback of a memento pattern results. At the same time, an AND-fork in the caretaker makes a path through a mediated colleague interaction. Note that the interaction in the opposite direction has its own concurrent path. The point is that the patterns can be used to structure the interactions and that UCMs are able to abstract the causal paths.

V. Example Performance Analysis

In this section, the performance of agents in a chain of responsibility (Fig. 7) is examined analytically. (Note that UCMs can be automatically converted to a Layered Queueing Network (LQN) [16], [23], which is beyond the scope of this paper. The performance of a LQN can be examined with analysis or simulation, yielding insights at the early development of agent architectures [17], [18].)

The chain of responsibility lends itself to an easy queueing analytic solution because of the probabilistic paths. The results, shown in Figs. 11–13, are based on an arrival rate of 100 alarms/s at each of the four sensors. The processing rate at each agent is 401 alarms/s. Each figure shows a performance metric dependent on the probability p that an agent passes the alarm to a parent agent. Conversely, the agent handles the alarm locally with probability 1-p. (The solid lines represent an analytic result assuming M/M/1 queues and the simulation results are the average of five runs using Q-Sim [24].)

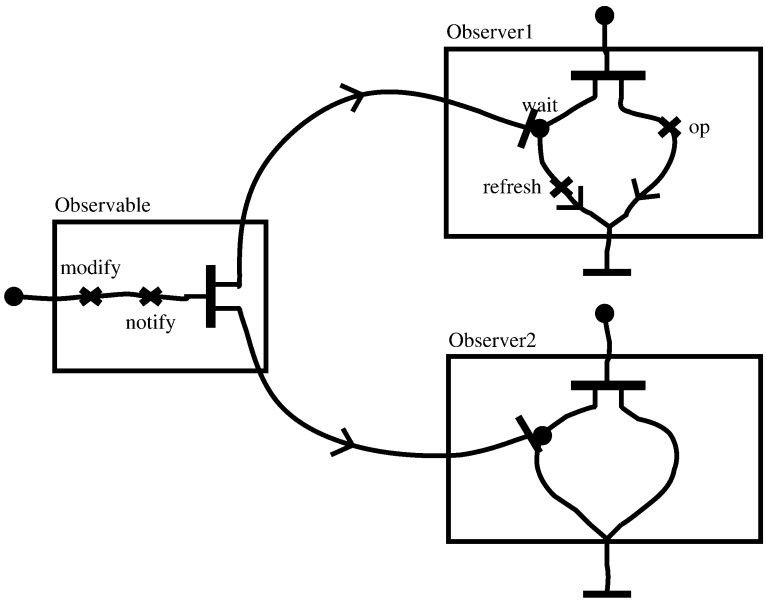

Fig. 8. The observer pattern provides a simple scenario for an observable agent to notify a set of observer agents upon important changes in state. These observer agents have a concurrent path, established with an AND-fork, to wait for notification, rather than continually interact or query for such an event.

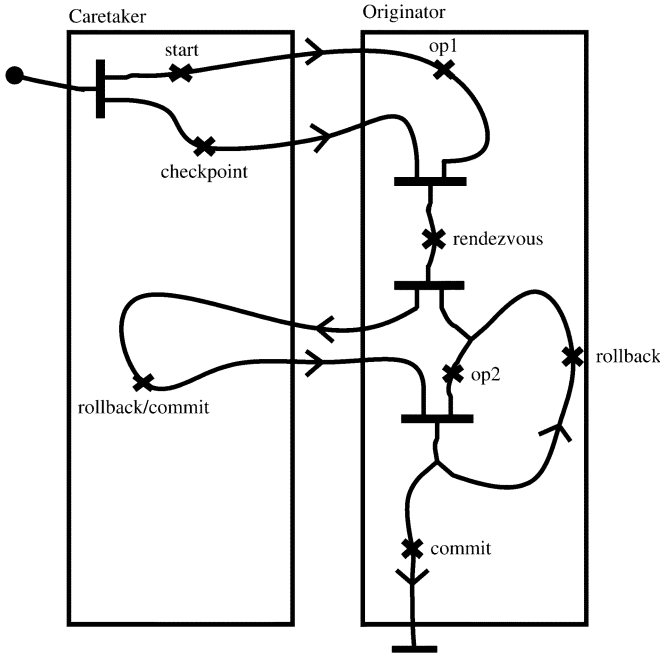

Fig. 9. The memento pattern shows a caretaker which saves a snapshot of the state of an originator. The caretaker is responsible for initiating the originator (normal application) and requesting state information at a checkpoint. If the caretaker decides that a fault exists, the state can be restored (a rollback), and the originator can repeat its actions starting from a consistent state.

The figures demonstrate that a bottleneck might occur within a room agent or a building agent, depending on the probability of passing the alarm along to a parent. The bottleneck also limits the maximum throughput of the entire system. The average response time to process an alarm grows rapidly, as expected, as the probability of locally handling the alarm decreases. Perhaps the number of layers, or the number of agents per level, or the processor type per level, should be modified. This type of analysis is useful during investigation of the agent interaction scenarios.

The main point of a pattern is to provide a reusable design that effectively solves a problem which may arise in many different applications. This design should be the state-of-the-art solution to the problem, following good design principles. There are two implications with respect to performance: 1) selecting an existing pattern might naturally improve the performance and 2) catalogs of well-known patterns might receive extra attention to optimization.

VI. Pattern Applications

In this section, agent applications in the literature are considered for their use of patterns. Some papers which directly reference patterns are summarized, followed by examples in this journal which make use of the eight patterns presented in this paper, but make no explicit reference to patterns per se.

As stated in Section II, UCMs originated in the telecommunication area [2] and some of these studies apply patterns [14], [15]. The proxy pattern is used for remote access in mobile scenarios [25] and network management [26]. This abstraction allows the easy access to a remote component as if the component was present locally. The mediator pattern is used to interconnect agents in distributed manufacturing, in a decoupled, rather than strongly coupled, sense [27]. A pattern not mentioned in the current study, the reflection pattern [19], is used in railway control to separate application logic from meta-level dependability requirements [28]. Besides general-purpose patterns, some domains apply the reuse of domain-specific patterns, such as the oil and gas exploration [29].

In the following, the eight design patterns presented in Section IV appear implicitly in papers which have appeared previously in this journal. Traffic signal control is presented as a hierarchy of interacting, autonomous agents similar to the chain of responsibility pattern in that data flows upwards from lower to higher levels [30]. The system is also similar to the composite pattern in that queries and action control flow downwards. A repository stores recommended policies at the end of each evaluation period as in the memento pattern. Agents in all levels of the hierarchy require global access to this repository which suggests that it be implemented as a singleton pattern.

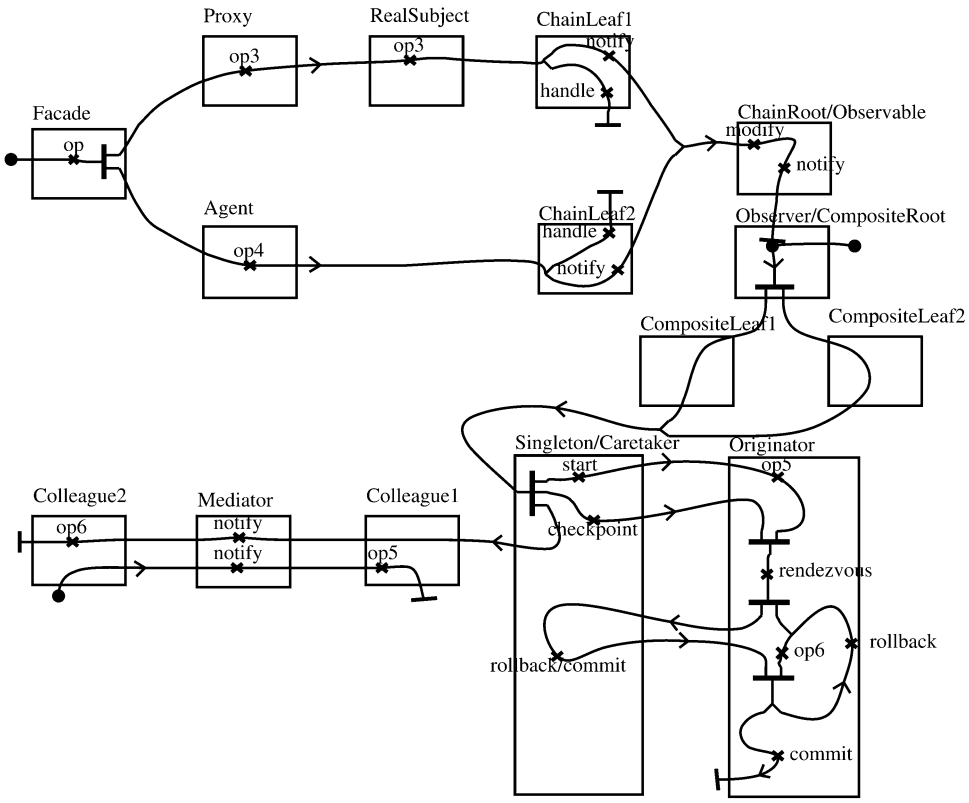

Fig. 10. A pattern-heavy application uses all eight patterns to structure the agent interactions and to illustrate the power of a UCM to abstract the emergent causal behaviors.

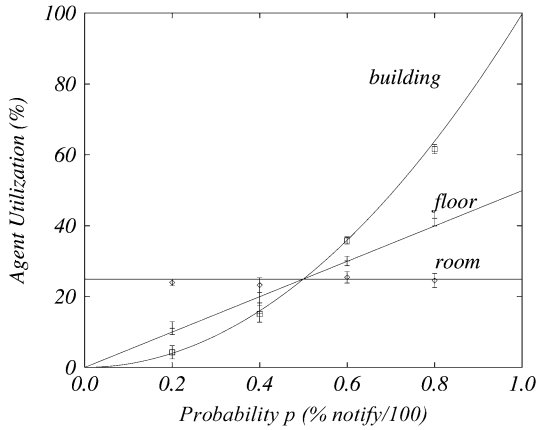

Fig. 11. Agent utilization at each level in the chain of responsibility. The chain is rooted at the building agent, which is the bottleneck in cases where lower level agents tend to pass along alarms rather than handle the alarm locally.

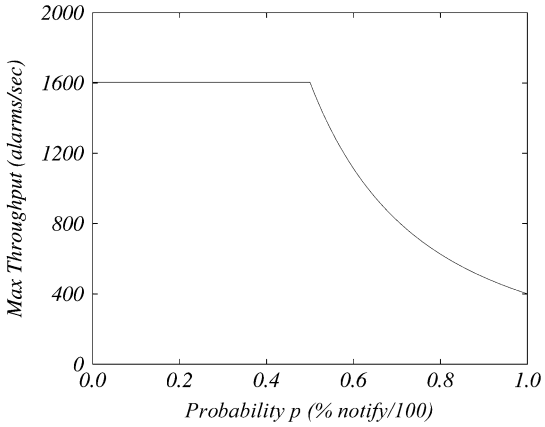

Fig. 12. Maximum throughput in a chain of responsibility. At low probabilities of notifying the parent agent, the throughput is constant because the room agents are the bottleneck and each has a fixed arrival rate of alarms, independent of probability. At high probabilities, the building agent is the bottleneck because the arrival rate increases as the alarms tend not to be handled locally.

Distributed agents providing cellular data services [31] use remote procedure calls, which can be easily implemented in Java using the proxy pattern. The use of common object request broker architecture (CORBA) to connect autonomous agents on a network [32] illustrates the broker pattern [19]. The broker establishes the interaction between an agent and a distributed resource. The broker acts as a mediator but only for the initial stages, whereas, a true mediator maintains the interconnection over the course of interaction.

The UML [20] presentation of container classes for various media in a distributed virtual environment [33] is exactly the composite pattern. This paper also references "area-of-interest managers," similar to a facade pattern that hides underlying complexity.

Another pattern not presented in the current paper is the model-view-controller (MVC) pattern [19], which is useful in user interface (UI) design as it separates the model (data) from the actual view (UI). This pattern makes use of the observable pattern to notify the controller when data changes, hence a refresh of the view(s) is required. A UML presentation of user interfaces very similar to the MVC pattern is presented in [34], along with a hierarchical building application resembling the chain of responsibility example in Section III.

Finally, a study by this author [7] might have benefited with the use of the MVC and observable patterns. The expected-value of game actions is the "model", which is observed by decision-making agents.

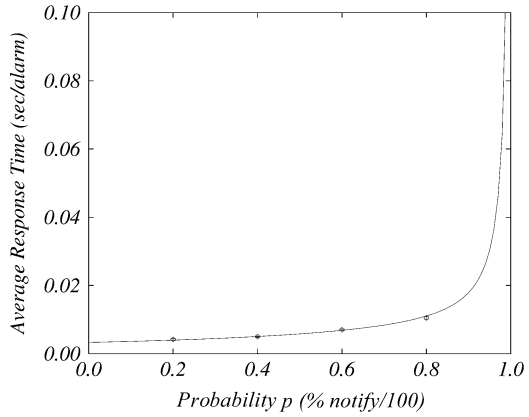

Fig. 13. The agents' average response time to alarms in a chain of responsibility. A high probability of notifying the parent agent means that the floor and building agents become overwhelmed with alarms that are not handled locally.

Each decision by an agent updates the game data and requires the controller to notify the other player/agents in the game. The controller represents the distributed environment, and establishes the delay of view inherent in the environment.

VII. Conclusion

This paper has attempted to show two things: how UCMs can present agent interactions and how some software design patterns are relevant for prototyping scenarios of agent interactions.

Although the "gang of four" patterns are intended for object-oriented software development, at least eight of the patterns suggest structural, or organizational, means of agents interacting with other agents, independent of any software or programming methodology.

Of the eight patterns presented here, seven of the patterns (the exception being the memento pattern) attempt to balance the decentralized nature of interacting agents with some sort of reorganized structure that makes for better, cleaner interactions.

It is hoped that this presentation will lead others to think about prototypical agent interactions and to consider the catalog of these scenarios when designing new systems. Also, UCMs make a convenient visual aid to analyze, at an abstract level, these agent interactions.

References

[1] R. J. A. Buhr and R. S. Casselman, *Use Case Maps for Object-Oriented Systems*. Englewood Cliffs, NJ: Prentice-Hall, 1996.

[2] R. J. A. Buhr, "Use case maps as architectural entities for complex systems," *IEEE Trans. Software Eng.*, vol. 24, pp. 1131–115, Dec. 1998.

[3] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*. Reading, MA: Addison-Wesley, 1995.

[4] E. Durfee and T. Montgomery, "Coordination as distributed search in a hierarchical behavior space," *IEEE Trans. Syst., Man, Cybern.*, vol. 21, pp. 1363–1378, Nov. 1991.

[5] E. Szczerbicki, "Acquisition of knowledge for autonomous cooperating agents," *IEEE Trans. Syst., Man, Cybern.*, vol. 23, pp. 1302–1315, Sept./Oct. 1993.

[6] K. S. Barber, T. H. Liu, and S. Ramaswamy, "Conflict detection during plan integration for multi-agent systems," *IEEE Trans. Syst., Man, Cybern. B*, vol. 31, pp. 616–628, Aug. 2001.

[7] E. Billard and S. Lakshmivarahan, "Learning in multi-level games with incomplete information—Part I," *IEEE Trans. Syst., Man, Cybern. B*, vol. 29, pp. 329–339, June 1999.

[8] E. Billard and J. Pasquale, "Effects of delayed communication in dynamic group formation," *IEEE Trans. Syst., Man, Cybern.*, vol. 23, pp. 1265–1275, 1993.

[9] K. Berbeek and A. Nowe, "Colonies of learning automata," *IEEE Trans. Syst., Man, Cybern. B*, vol. 27, pp. 376–394, June 2002.

[10] A. R. Grades and C. Czarnecki, "Design patterns for behavior-based robotics," *IEEE Trans. Syst., Man, Cybern. A*, vol. 30, pp. 36–41, Jan. 2000.

[11] Use Case Maps Web Site [Online]. Available: http://www.UseCaseMaps.org

[12] R. J. A. Buhr, M. Elammari, T. Gray, and S. Mankovski, "Applying use case maps to multi-agent systems: A feature interaction example," in *Proc. 31st Annu. Hawaii Int. Conf. System Sciences*, 1998, pp. 171–179.

[13] R. J. A. Buhr, D. Amyot, M. Elammari, D. Quesnel, T. Gray, and S. Mankovski, "High level, multi-agent prototypes from a scenario-path notation: A feature-interaction example," in *Proc. 3rd Conf. Practical Application of Intelligent Agents and Multi-Agent Technology*, London, U.K., 1998, pp. 277–295.

[14] G. Mussbacher and D. Amyot, "A collection of patterns for use case maps," in *Proc. 1st Latin American Conf. Pattern Languages of Programming*, Rio de Janeiro, Brazil, 2001, pp. 57–87.

[15] F. Bordeleau, J. Corriveau, and B. Selic, "A scenario-based approach to hierarchical state machine design," in *Proc. 3rd IEEE Int. Symp. Object-Oriented Real-Time Distributed Computing*, Newport Beach, CA, 2000, pp. 78–85.

[16] D. Petriu, C. Shousha, and A. Jalnapurkar, "Architecture-based performance analysis applied to a telecommunication system," *IEEE Trans. Software Eng.*, vol. 26, pp. 1049–1065, Nov. 2000.

[17] K. H. Siddiqui and M. Woodside, "Performance-aware software development (PASD) using resource demand budgets," in *Proc. 3rd Int. Workshop Software and Performance*, 2002, pp. 275–287.

[18] D. Petriu and M. Woodside, "Analysing software requirements specifications for performance," in *Proc. 3rd Int. Workshop Software and Performance*, 2002, pp. 1–9.

[19] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal, *A System of Patterns*. New York: Wiley, 1996.

[20] G. Booch, J. Rumbaugh, and I. Jacobson, *The Unified Modeling Language User Guide*. Reading, MA: Addison-Wesley, 1999.

[21] C. Larman, *Applying UML and Patterns*. Englewood Cliffs, NJ: Prentice-Hall, 2002.

[22] M. Grand, *Patterns in Java: A Catalog of Reusable Design Patterns Illustrated Using UML*. Reading, MA: Wiley, 1998.

[23] Layered Queueing Network Web Site [Online]. Available: http://www.sce.carleton.ca/rads/

[24] E. Billard and A. Riedmiller, "Q-Sim: A GUI for a queueing simulator using Tcl/Tk," *ACM Software Eng. Notes*, vol. 19, no. 4, pp. 82–85, 1994.

[25] J. Roth, "Patterns of mobile interaction," *Personal and Ubiquitous Comput.*, vol. 6, pp. 282–289, Sept. 2002.

[26] R. Keller, J. Tessier, and G. von Bochmann, "A pattern system for network management interfaces," *Commun. ACM*, vol. 41, pp. 86–93, Sept. 1998.

[27] F. Maturana and D. Norrie, "Multi-agent mediator architecture for distributed manufacturing," *J. Intell. Manufact.*, vol. 7, pp. 257–270, 1996.

[28] D. M. Beder, A. Romanovsky, B. Randell, C. R. Snow, and R. J. Stroud, "An application of fault tolerance patterns and coordinated atomic actions to a problem in railway scheduling," *ACM Operating Syst. Rev.*, vol. 34, pp. 21–31, Oct. 2000.

[29] Y. Li and J. Lu, "SEIS++: A pattern language for seismic tools construction and integration," *AMC SIGPLAN Notices*, vol. 34, pp. 57–66, Dec. 1999.

[30] M. C. Choy, D. Srinivasan, and R. L. Cheu, "Cooperative, hybrid agent architecture for real-time traffic signal control," *IEEE Trans. Syst., Man, Cybern. A*, vol. 33, pp. 597–607, Sept. 2003.

[31] J. Chen, "Resource allocation for cellular data services using multiagent schemes," *IEEE Trans. Syst., Man, Cybern. B*, vol. 31, pp. 864–869, Dec. 2001.

[32] A. I. El-Osery, J. Burge, M. Jamshidi, A. Saba, M. Fathi, and M. Akbarzadeh-T, "V-lab—A virtual laboratory for autonomous agents—SLA-based learning controllers," *IEEE Trans. Syst., Man, Cybern. B*, vol. 32, pp. 791–803, Dec. 2002.

[33] M. Matijasevic, D. Gracanin, K. P. Valavanis, and I. Lovrek, "A framework for multiuser distributed virtual environments," *IEEE Trans. Syst., Man, Cybern. B*, vol. 32, pp. 416–429, Aug. 2002.

[34] R. R. Penner and E. S. Steinmetz, "Model-based automation of the design of user interfaces to digital control systems," *IEEE Trans. Syst., Man, Cybern. A*, vol. 32, pp. 41–49, Jan. 2002.