A Multi-Level Technique for Modeling Agent-Based Systems

Francisco Hernandez, Jeff Gray, Kevin Reilly

Department of Computer Science
University of Alabama at Birmingham
{hernandf, gray, reilly}@cis.uab.edu

Abstract. A layered technique is introduced in this paper to examine agent-based systems from three different levels of abstraction. The technique provides analysis capabilities of an agent system from the point of view of roles, agents, and corresponding implementation objects. The core of the technique involves the specification of meta-models for each level. Three different translators are implemented to interpret each model and to create a corresponding specification of the next subsequent level. The final translator generates code suited for a specific agent environment.

1 Introduction

Research into multi-agent systems software development has taken two different directions: 1) the creation of new methodologies specifically tailored to agents, and 2) the adaptation of current object-oriented techniques to the design of multi-agent systems [1], [2]. Current studies have considered one of these two approaches, and an extensive list of studies according to this categorization is given in [2] and [3]. Both approaches have their own benefits and drawbacks because each considers agent systems from two different perspectives. A major advantage could be achieved by integrating both approaches into a coherent technique [4].

During the analysis and design of a multi-agent system it is desirable to look at agents as atomic objects. However, the implementation of agents as objects could lead to an implementation that is complex, and difficult to understand, maintain and reuse [5]. Often, objects are too fined-grained to be able to represent the complex behavior that agents carry out [6].

The problem is that agents and objects typically operate at a different level of abstraction [6]. Agents are not objects [7], per se, and therefore they require special techniques to analyze them. A technique that unifies agents and objects is required [4]. Such a technique should be able to consider the whole life-cycle to achieve benefits from both approaches. In order to do this, the technique should be able to enable iteration along the software development cycle at different levels of abstraction.

There are several studies that consider different levels of abstraction through the development cycle. Odell et al. presents a layered approach to protocols and uses an extension of UML to model different properties of agents [8]. Bergenti and Poggi also present an agent toolkit that can operate at two different levels of abstraction and

helps to bridge the gap between the specification and the implementation of agent-based systems [9].

This paper introduces a three level technique that looks at multi-agent systems from three independent levels of abstraction. This technique considers the system from the perspective of roles, agents, and objects. The core of the technique creates meta-models for each level. Three translators are necessary to interpret each model and create a corresponding specification that is fed into the next subsequent level. The final translator generates code suited for a specific agent environment.

In this paper, the internal behavior of an agent is specified. The interactions between agents are specified with different models. It is beyond the scope of this paper to define those models. Nevertheless, they need to be mentioned within this technique. Section 2 provides a general background on meta-modeling concepts. Section 3 presents the three-level technique. Section 4 presents an example of the technique; Section 5 introduces some future work relevant to the present study. Finally, section 6 offers conclusions of the present work.

2 Meta-Modeling

In domain-specific modeling, a design engineer creates models for a specific domain using concepts and terminology from that domain [10]. The meta-model defines the kind of models that can be built – it specifies the ontology of the domain. In order to create the meta-models, a modeling paradigm must be used. The modeling paradigm is the modeling language of the application domain. The modeling paradigm provides a set of requirements that the meta-model needs to be able to generate models of the given domain. These requirements involve the concepts that will be used to construct the models, the relationships and organization of those concepts, how the concepts are viewed by the modeler and rules and constraints governing the construction of the model [11]. Models constructed with this style capture information relevant to the system under design. An interpreter can translate these models into executable specifications used to automatically synthesize software [12].

In the technique described in this paper, a meta-model is constructed for each level. Each meta-model is specific to the technique or methodology implemented in that level. Three translators or interpreters are also constructed. These translators interpret a model from a specific level and generate a model from the subsequent level. The final translator generates implementation code suited for the specific agent environment.

3 Three Level Technique

The technique introduces three independent levels of abstraction. Each level generates a model for the next level leading to the generation of code at the object level. This technique uses a top-down approach in which the first model created is the one at the highest level of abstraction. A multilevel approach is used because it combines the advantages of the agent-oriented and object-oriented approaches. It is more intuitive

to analyze an agent-based system using common abstractions of the domain [9] such as roles, responsibilities, and interactions properties; and at the same time proven object-oriented modeling techniques provide a more natural approach for implementing the system. Three levels of abstraction provide for a more intuitive mapping between agent-oriented analysis models and traditional object-oriented design techniques that may be applied to implement the agents. The code that is generated is suited for a specific agent environment, and should be completed with application specific code as in [9].

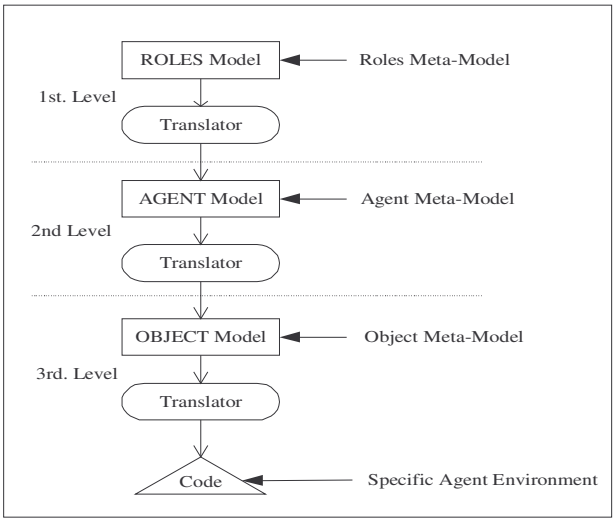

Fig. 1. Three level technique

A specification of a level can evolve by changing the corresponding meta-model. If the meta-model of a level is changed, the translator of the previous layer should also be changed since the previous level generates the initial specification for that specific level. A visual modeling language assists the modeler in manipulating the model restricted by the constraints of the meta-model. Figure 1 presents a diagram of the whole technique.

The process for using this technique is summarized by the following tasks:

- Specify the meta-model for each of the levels.
- Specify the translator for each level to the next one in the hierarchy (starting from the top level). The translator from the object level will generate code.
- Create a role model and translate it into its corresponding agent model.
- Modify the agent model and translate it into its corresponding object model.
- Modify the object model and generate code from that model.
- Add the application specific code for the agent.

Each level requires further explanation.

3.1 The Roles Level

In this level, agents are viewed as having a role to follow, with an agent-based system considered as a collection of roles. This point of view is not new and can be found in the literature (e.g., [6]). Kendall [13] presents a comprehensive list of reasons why role modeling is appropriate for intelligent agent systems. We utilize a role modeling level as an approach for analyzing an agent-based system, as in [6]. A previous analysis is conducted to be able to determine which roles would be part of the system.

The way in which a role is defined varies among different studies [6], [13], [14], but as a standard for the remainder of this paper, roles have: beliefs, goals or responsibilities, capabilities or activities, and interaction protocols. The beliefs are the knowledge that a role should have in order to perform its actions. The goals specify the function of the role or the responsibility that the role has. The capabilities are the actions or activities that the role can perform. Finally, the interaction protocols specify how a role will interact with other roles.

The meta-modeler should take into consideration the properties defining a role and create a meta-model that relates those properties in a coherent form. Because the interaction protocols may involve different roles, and maybe different agents, a separate model is needed to specify such protocols. This follows the approach taken in [6]. The specification of this kind of model is out of the scope of this paper, nevertheless they need to be mentioned in this level.

While creating the meta-model, we need to specify the constraints of the model (e.g., we may enforce that each agent will implement just one role). This is typically done within the meta-model by specifying constraints in the Object Constraint Language (OCL) [15].

After the meta-model is defined, role properties need to be specified as instantiations of the meta-model concepts. After all the roles in the system have been modeled, the roles that are implemented by each agent need to be specified. Roles are grouped by the agents they represent and are modeled accordingly. The role model is passed to a translator that interprets the model and creates a specification that becomes the starting point of the next level - in this case the agent level. Before defining the translators, the meta-models for the three levels need to be defined, because the translator should know what it is supposed to generate for the next subsequent level.

3.2 The Agent Level

Each agent is formed from a collection of roles. The roles that are part of each agent are defined in the specification of the previous level, which defined the properties forming each role. That specification had a lack of detail about how each particular property or concept should behave. At this point of the process, it is important to define each property more completely in order to give a more concrete specification of all the properties of the agent.

The meta-modeler should create a meta-model of the particular way in which the agent level needs to be modeled. At a minimum, this new meta-model should provide artifacts to specify how to implement each one of the properties and concepts generated on the roles level. The meta-modeler could either create the meta-model from

scratch, or leverage off of an existing methodology that is already developed (e.g. UML activity diagrams [8]). Some of the important concepts that need to be modeled in this level are:

- The beliefs of the agent.
- The capabilities or activities of the agent.
- The responsibilities or goals of the agent.

Another important example of information that would be useful is a specification of the inter-agent properties that the particular agent supports. Inter-agent properties define the type of agent that is going to be created [4]. These properties (e.g., autonomy, reactivity, mobility, interactivity, social ability) vary among different studies [3]. Specifying which properties are going to be implemented by the agent can yield significant code reuse [2] when transitioning to the object level.

A major goal of software engineering is reuse. By providing enough information at the agent level, the design of the object level can be reused [16]. The agent level has as its primary goal the specification of each individual agent. It also has a side goal of being able to deliver information that will ease the modeling of the object level.

The design of the agent meta-model should also consider constraints that must be taken into account while modeling a specific agent. Once the agent meta-model is created, we could start modeling each individual agent by using the specifications generated by the translator coming from the roles level (recall that the roles level generated empty buckets that need to be filled using the corresponding agent model).

After modeling the agent, the model can be sent to the agent translator. This translator generates the initial specification for the object level. As with the roles translator, the object level meta-model should be completely defined before the creation of the translator.

3.3 The Object Level

In this level the agent is analyzed as a collection of objects or a group of objects working together to provide the functionality that the specific agent requires. The purpose of this level is to model agents at a lower layer of abstraction. Models from this level should have a direct relation to the run-time system in which the agent is executing. Code generated by this level should be completed with application specific code as in [9].

The meta-modeler should take into consideration the following concerns while designing the meta-model for this level:

- The facilities that are shared between agents (i.e., inter-agent properties), because they can provide code reuse.
- The environment in which the agent is going to perform [17].

The environment must be taken into consideration because it should provide some services to the multi-agent system (e.g. communication, and persistence). These services are shared among all the agents running in a specific environment [5] so it is of

utmost importance to have them preloaded in the system. Further information necessary to create an optimized version of the generated code should also be included.

Because the purpose of the system is to model agents at the object level, the meta-modeler could consider existing object oriented techniques to create the meta-model (e.g., separation of concerns and aspect oriented programming [5], [18]). Starting from the specification generated from the translator of the agent level, it is possible to model the corresponding object representation of the agent. The final stage of this level is the translation of the model to generate its corresponding code.

5 Example of the Technique

In this section an example is presented to illustrate the technique. An example taken from Wooldridge et. al. [6] is expanded to show how this technique will provide a path from a roles model (analysis phase) to an implementation phase. Current methodologies will be used to create the meta-models of each level. The roles model of Gaia [6] is used as the methodology for the roles level. UML [8] is used to further specify the activities of the agent in the agent level. Finally, an aspect oriented approached based on [2], and [3] is used for the meta-model of the object level.

Wooldridge et. al. [6] introduces an example of a CoffeeFiller role. The responsibilities of this role are to keep a coffee pot full and to inform workers when fresh coffee has been brewed.

Role Schema: CoffeeFiller

Description:
This role involves ensuring that the coffee pot is kept filled, and informing the workers when fresh coffee has been brewed.

Protocols and activities:
Fill, InformWorkers, CheckStock, AwaitEmpty

Permissions:

reads	supplied coffeeMaker	// name of coffee maker
	coffeeStatus	// full or empty
changes	coffeeStock	// stock level of coffee

Responsibilities:
Liveness:
CoffeeFiller = (Fill. InformWorkers. CheckStock. AwaitEmpty)^{w}
Safety:
- coffeeStock > 0

Fig. 2. Schema for role CoffeeFiller. Taken from [6]

In Gaia [6] a role is defined by four attributes: responsibilities, permissions, activities, and protocols. Responsibilities define the functionality of the role, and they are further classified by liveness and safety properties. Liveness properties indicate the principal function of the agent, and safety properties identify invariants that should be maintained along the execution of the agent. Activities are actions that the agent performs without interacting with other agents. Protocols are actions in which other agents need to be involved. Finally permissions are rights associated with a role.

Having decided on which properties are needed for the roles model, the corresponding meta-model is needed. The next step on this level is to model the schema given by figure 2. At this point there is a shift from the Gaia methodology by translating the model into the initial specification for the agent level. The translator takes every protocol, activity and responsibility specified in the roles level and generates empty diagrams that need to be completed using the agent level meta-model.

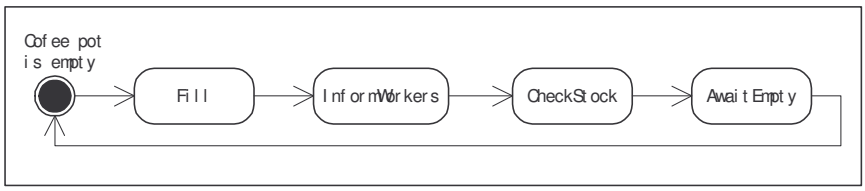

Fig. 3. Activity Diagram for the CoffeeFiller responsibility

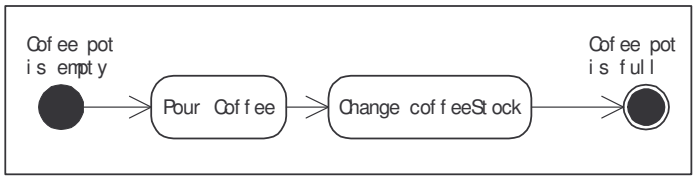

Fig. 4. Activity Diagram for the Fill protocol

For the agent level UML techniques are used to specify the internal agent processing more completely. Activity diagrams and StateCharts (as in [8]) are used to complete the diagrams generated by the roles translator. At this point, additional activities could be introduced to help define the ones specified in the role.

Figure 3 presents the activity diagram for the CoffeeFiller responsibility. The internal activities in the diagram can be further specified. Figure 4 presents the activity diagram for the protocol "Fill".

The specification of the particular inter-agent properties this agent needs to implement is also required. For example, the CoffeFiller agent is autonomous and interactive. The interactive property allows the agent to send and receive messages [3].

After the modeling of the internal processing of the agent is finished, the translator is used to generate the initial specification for the object level. The process for this translator is to generate a UML static structure diagram with the activities from the previous level as methods. The services that are going to be provided by the system in which the agent is going to execute should also be known by the translator. In this

case, the system is going to provide a communication mechanism using the interactivity property on the agent. The translator should generate the corresponding relationships between the interactivity property and the agent.

An aspect oriented approach that is based on [2], [5] is appropriate for the object level meta-model. A generic agent class is provided to which aspects are adapted for accomplishing interactivity and autonomy. The CoffeFiller agent class is then inherited from that generic class. Figure 5 presents the diagram of the object level model. This is a normal UML diagram with the addition of the aspects that are related to the agent class. The model can be modified in order to optimize the generated code. Finally from this point the corresponding code from the object model can be generated. The last step is to add the application specific code.

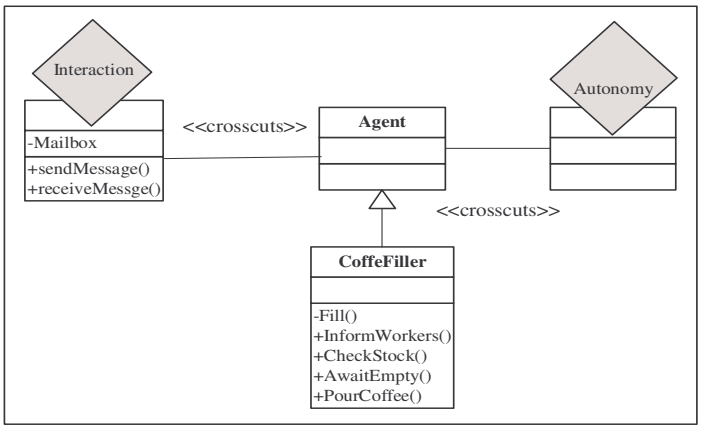

Fig. 5. UML Representation of the Object level model for the CoffeFiller agent

5 Future Work

The presented technique considers a top-down approach that starts at the highest level of abstraction and moves down each level by generating a model for the subsequent level. This ultimately leads to the generation of code at the object level. This work is under development and further work is required to create a working model.

This model could also potentially be designed to function in the other direction (i.e. modifying the object model and reflecting those changes into the agent and roles models). Adapting the model in this way could further improve the maintainability of the system as a form of round-trip agent modeling.

The use of current methodologies on each level could be of great importance to test the approach. In the example presented in this paper, we examine using the Gaia methodology [6] as the roles level meta-model, UML [8] for the agent level, and a combination between UML, separation of concerns and Aspect Oriented Development (AOD) implementation for the object level [2], [5]. The use of other methodologies is something that still needs to be explored.

6 Conclusion

The proposed technique aims to design agents at three independent levels of abstraction: the roles, the agent, and the object level. Three levels of abstraction are used because they provide for a more intuitive mapping between agent-oriented analysis models and traditional object-oriented design techniques that may be applied to implement the agents. By doing this, it is possible to optimize the benefits and minimize the disadvantages of each methodology considered, thus improving the overall quality of the system. This technique uses a meta-modeling approach that automatically translates models from one level to the subsequent level. This helps to manage the maintainability and reusability of the code. This technique could also use current methodologies for a particular level, allowing the selection of the best methodologies for solving a particular problem within a level of abstraction. The proposed technique aims to unify the abstract agent oriented analysis and design with the more concrete object oriented implementation.

References

1. C. Lucena, A. Garcia, J. Castro, A. Omicini, and F. Zambonelli, "Software Engineering for Large-Scale Multi-Agent Systems – SELMAS 2002," *Proceedings of the IEEE/ACM International Conference on Software Engineering (ICSE 2002)*, Orlando, USA, May 2002, pp. 653-654.
2. A. Garcia, C. Lucena, D. Cowan, "Agents in Object-Oriented Software Engineering," Technical Report CS-2001-07, Computer Science Department, University of Waterloo, Waterloo, Canada, February 2001.
3. A. Garcia, V. Torres, C. Lucena, and R. Milidiú, "An Aspect-Oriented Design Approach for Multi-Agent Systems," Technical Report, Computer Science Department, PUC-Rio, Brazil, June 2001.
4. R. Depke, R. Heckel and J.M. Kuster, "Formal agent-oriented modeling with UML and graph transformation," *Science of Computer Programming*, 44 (2) (August 2002), pp. 229-252.
5. A. Garcia, C. Chavez, O. Silva, V. Silva, and C. Lucena, "Promoting Advanced Separation of Concerns in Intra-Agent and Inter-Agent Software Engineering," *Workshop on Advanced Separation of Concerns in object-oriented Systems (ASoC) at OOPSLA'2001*, Tampa Bay, Florida, USA, October 14, 2001.
6. M.J. Wooldridge, N.R. Jennings, D. Kinny, "The Gaia Methodology for Agent-Oriented Analysis and Design," I*nternational Journal of Autonomous Agents and Multi-Agent Systems*, 3 (3) (2000), pp. 285-312.
7. C. A. Iglesias, M. Garijo, and J.C. Gonzalez, "A survey of agent-oriented methodologies," in *Intelligent Agents V-Proceedings of the Fifth International Workshop on Agent Theories*, Architectures, and Languages (ATAL-98), Lecture Notes in Artificial Intelligence, J.P. Müller, M.P. Singh, and A. S. Rao, (Eds.), Springer-Verlag: Heidelberg, 1999.
8. J. Odell, H. Van Dyke Parunak, and B. Bauer, "Extending UML for Agents," AOIS Workshop at AAAI 2000.
9. F. Bergenti, and A. Poggi, "A Development Toolkit to Realize Autonomous and Interoperable Agents," *Proceedings of Fifth International Conference of Autonomous Agents (Agents 2001)*, Montreal, Canada, pp. 632-639.

10. J. Gray, T. Bapty, S. Neema, and J. Tuck, “Handling Crosscutting Constraints in Domain-Specific Modeling,” *Communications of the ACM*, October 2001, pp. 87-93.
11. A. Ledeczy et al., “The Generic Modeling Environment,” *Workshop on Intelligent Signal Processing*, Budapest, Hungary, May 17, 2001.
12. J. Sztipanovits and G. Karsai, “Model-Integrated Computing,” *Computer*, Apr. 1997, pp. 110-112.
13. E. A. Kendall, “Agent Roles and Role Models: New Abstractions for Intelligent Agent System Analysis and Design,” *ECOOP’99*, AOP Workshop, Lisbon.
14. B. Bauer, “UML Class Diagrams: Revisited in the Context of Agent-Based Systems,” *In Proceedings of Agent-Oriented Software Engineering (AOSE) 2001*, Agents 2001, Montreal, Canada, pp. 1-8.
15. Jos B. Warmer, Anneke G. Kleppe, *The Object Constraint Language: Precise Modeling With UML*, Addison-Wesley, 1999.
16. Ivar Jacobson, Martin Griss, Patrik Jonsson, *Software Reuse: Architecture, Process, and Organization for Business Success*, Addison-Wesley, 1997.
17. James Odell, “Modeling Agents and their Environment: The Physical Environment,” *in Journal of Object Technology*, vol. 2, no. 2, March-April 2003, pp. 43-51. http://www.jot.fm/issues/issue_2003_03/column5
18. G. Kiczales et al., “Aspect-Oriented Programming,” *Proceedings of the ECOOP’97*, Finland. Springer Verlag LNCS 1241, June 1997.