Model-Driven Architecture for Agent-Based Systems

Denis Gračanin[1], H. Lally Singh[1], Shawn A. Bohner[1], and Michael G. Hinchey[2]

[1] Department of Computer Science
Virginia Polytechnic Institute and State University, Blacksburg, VA 24061, USA
{gracanin,lally,sbohner}@vt.edu
[2] NASA Goddard Space Flight Center, Greenbelt, MD 20771, USA
Michael.G.Hinchey@nasa.gov

Abstract. The Model Driven Architecture (MDA) approach uses a platform-independent model to define system functionality, or requirements, using some specification language. The requirements are then translated to a platform-specific model for implementation. An agent architecture based on the human cognitive model of planning, the Cognitive Agent Architecture (Cougaar) is selected for the implementation platform. The resulting Cougaar MDA prescribes certain kinds of models to be used, how those models may be prepared and the relationships of the different kinds of models. Using the existing Cougaar architecture, the level of application composition is elevated from individual components to domain level model specifications in order to generate software artifacts. The software artifacts generation is based on a metamodel. Each component maps to a UML structured component which is then converted into multiple artifacts: Cougaar/Java code, documentation, and test cases.

1 Introduction

Agent-based systems provide a foundation for development of large scale applications like logistics management, battlefield management, supply-chain management, to mention some. An example of agent-based systems is Cougaar (Cognitive Agent Architecture). Cougaar provides a software architecture for distributed agent-based applications in domains characterized by hierarchical decomposition, tracking of complex tasks, generation and maintenance of dynamic plans [1, 2].

The ability develop very complex applications comes with a price. It takes a lot of effort and learning in order to have complete understanding and ability to effectively use such agent-based systems. A domain expert must closely collaborate with the developer in order to fully utilize an agent-based system for a particular domain. It is very unlikely that a domain expert will have sufficient understanding of the underlying agent-based system.

A Model Driven Architecture (MDA) based approach can be used to automatically generate software artifacts and to significantly simplify application

development [3, 4, ?]. The domain expert can specify requirements in a familiar, platform-independent format that hides platform-specific details.

The MDA approach can be used for developing applications using the Cougaar agent-based architecture. Cougaar components can be composed into a General Cougaar Application Model (GCAM) and develop a General Domain Application Model (GDAM) for specifying and automatically generating software applications. This approach is discussed in the paper.

The remainder of the paper is organized as follows. Section 2 briefly describes Cougaar and its capabilities. Section 3 describes the use of the MDA approach for Cougaar-based applications. Section 4 discusses the Cougaar-based MDA model while Section 5 describes the implementation. Section 6 concludes the paper.

2 Cougaar Agent-Based System

Cougaar is a "large–scale workflow engine built on a component-based distributed agent architecture" [1]. It is deployed as a *society* of *agents*, which communicate and work together to solve a problem. A Cougaar society is a set of agents running on one or more interconnected computers, all working together to solve a common class of problems. The problem may be partitioned into subproblems, in which case the responsible subset of agents is called a *community*. A society may have one or more communities within.

The relationship between societies, communities, and agents is not a strict one, a society may directly contain both agents and communities. While a society has a real-world representation, a set of computers running a Cougaar system, a community is only notational in nature.

A Cougaar agent is a first-class member of a Cougaar Society [1] and it contains a Blackboard and one or more Plugins. While the specific purpose of any agent is chosen by the system developer; the objective is for a single agent to represent a single organizational entity or a part thereof.

At the most basic level, an agent consists of two parts: a Blackboard and a set of Plugins (Figure 1). The former is a container of objects, with a subscription-based change notification mechanism; the latter is a set of responders to these notifications, with the ability to change the contents of the Blackboard.

The Blackboard serves as the communications backbone connecting the Plugins together. More importantly, it serves as the entry point for any incoming messages to the agent as a whole, which are then picked up by the Plugins for handling. All instance-specific behavior of the agent is implemented within the Plugin. A Plugin listens to add, remove, and change events on the Blackboard. Evaluating the objects involved in the event, the Plugin may respond by performing some computation, changes to the Blackboard, or some external work.

A Cougaar Node conceptually encapsulates a set of agents. Agents can collaborate with other agents in the same Node or with agents in other Nodes. However, it is not a direct collaboration. Instead, Cougaar Tasks are allocated to Cougaar Organizations, which are representations of agents in the local Blackboard. The

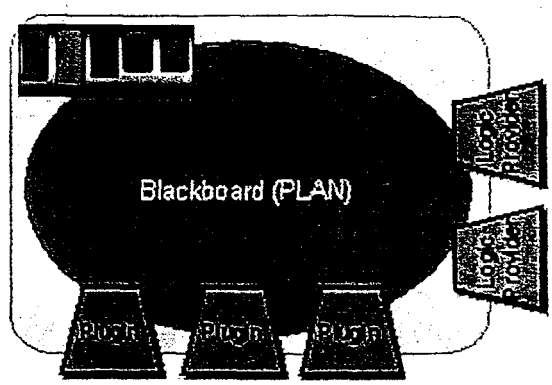

Fig. 1. Cougaar Agent Structure [1]

subscription mechanism allows agents to use Tasks to exchange messages (objects). The Cougaar communication infrastructure then ensures that the Task is sent to the destination Organizations (i.e. agent's) Blackboard.

3 Cougaar Model Driven Architecture (CMDA)

The MDA approach advocates converting a Platform Independent Model (PIM) into a Platform Specific Model (PSM) through a series of transformations, where the PIM is iteratively made more platform specific, ending in the PSM. The PIM is used to represent system's business functionality without including any technical aspects. The PIM allows Subject Matter Experts (SMEs) to work at the domain layer. However the current technologies may not offer the required richness to implement the complex transformation rules. For example, the Unified Modeling Language (UML), the foundation for MDA, lacks in the required precision and formalization.

While the development of PIM and PSM UML models might be easy, a blind adaptation of the MDA approach might create problems during the development of mapping rules and transformations. It should be noted that the MDA approach advocates for a Computational Independent Model (CIM) that needs to be transformed into a PIM. Since UML uses different representations for each of the models, the translation between models is more like translation between natural languages, the mappings are not necessarily exact. Further, while the learning curve associated with UML is fairly low, the SMEs nevertheless need to learn a new technical language and need to "move" out of their work environment.

The productivity of Cougaar system developers can be improved by using the MDA approach. The Cougaar MDA (CMDA) attempts to provide fully automated generation of software artifacts and simplifies Cougaar-based application development by providing two important abstraction layers. The first layer is the

Generic Domain Application Model (GDAM) layer. The GDAM represents the PIM and encompasses the representation of generic agent and domain specific components found in the domain workflow. The second layer, Generic Cougaar Application Model (GCAM) reflects the PSM or Cougaar architecture. The user specifies the intended Cougaar system using workflow paradigm and the system is then refined using GDAM and GCAM models.

The GDAM layer implement the PIM based on a representation that SMEs are comfortable with and result in a proper mapping to the PSM. The goal is to make this mapping as automated as possible, while having human-in-the-loop as a fallback mechanism to correct any mapping imperfections. The initial versions of the tool might force the developers to fine-tune the generated PSM to certain extent, but it is hoped that as the tools and algorithms advance, such fine-tuning would be less and less necessary.

The GDAM layer specifies the structure and semantic information that the tool uses to ensure that the developer has annotated the GDAM model properly. Furthermore, the layer provides all information required by the tool to produce a more specific but still platform-independent PIM that includes details of desired semantics, and guides choices that the approach/tool will have to make (Figure 2).

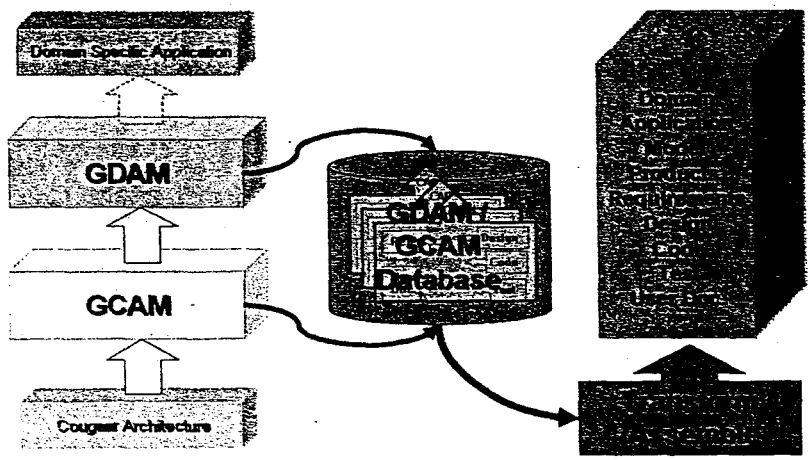

Fig. 2. Basic CMDA Approach

In order to develop a tool based on the proposed approach, the following assumptions and constraints were formulated after detailed research.

- Fully automated software artifacts (requirements, design document, code, and test cases) generation is a desirable goal.
- The generated requirements are partial in nature.

- The validation of generated code and the generation of test cases are of lower priority.
- The development of tools and implementation mechanisms are of lower priority than formulating the "recipes" for transformations.
- The intended users of the system are developers and subject matter experts.
- The developer should be fully aware of the Cougaar system, its capabilities, and constraints.
- The SMEs should have sufficient knowledge about the domain and a basic understanding of the requirements of the intended system.

3.1 GDAM Layer

The General Domain Application Model (GDAM) can be conceptually thought to be similar to various programming language libraries such as MFC or Swing. The libraries abstract and modularize the commonly used functions, thereby helping Subject Matter Experts (SMEs) to focus on encoding business logic. However, the abstractions achieved by class libraries, which are written in implementation language, are limited by the capabilities of the language. Further, SMEs have to work at the implementation language level.

The genesis of the GDAM layer can be traced to the need to allow SMEs to develop systems at the domain layer using current technologies and simple transformation rules. In short, GDAM allows SMEs to represent the specifications of the system in a platform-independent, domain-specific language that can be transformed, without losing information, into specifications of how applications will be implemented in the Cougaar platform. Further, GDAM provides a set of components and patterns representing the different kinds of generic domain elements that can be assembled to specify the application.

There are two, potentially conflicting implications of the GDAM functionality. First, SMEs should be allowed to capture their domain knowledge and application requirements in a manner that is computationally independent. Second, there should be a well defined structure and relationships among requirements to allow for an automatic and mechanic transformation of the requirements/constraints into an internal, platform independent, GDAM representation that can be later transformed into a platform specific GCAM representation. To reduce this potential conflict, the following decisions were made:

- The transformation between the computational independent and platform independent representations should be a lightweight one. In other words, the platform independent transformation should subsume computational independent representation thus requiring only a simple transformation between the two.
- The business logic, i.e. the "semantic" of the application must be embedded within the computational independent representation enforce constraints. The constraint language must be simple and easily transformed into code that can be integrated within the platform.

- The configuration and deployment of the application is treated separately from the application requirements because that is inherently platform specific. While every effort will be made to make it as generic as possible, some platform specific information may be necessary.
- User interactions and user interface represent a separate challenge. Automatic or semi-automatic user interface generation based on the application requirements is not a unique one, i.e. there can be many different user interface designs. Such designs can be customized based on the SMEs preferences. While this effort is outside of the scope of the project, some considerations will be provided for possible future research.

The development of GDAM is an iterative and evolutionary process. In addition to the general system wide assumptions, the following assumptions are specific to the GDAM layer.

- The current scope is restricted to the development of some of the indispensable generic domain components that pertain to logistics domain.
- The GDAM components development is an evolutionary process and it is not expected or possible to develop each and every GDAM component.
- The developer and the SME will work together, sitting side-by-side if required, while developing the GDAM model of the intended system.
- Developers will collaborate with subject matter experts to develop and update the system with GDAM components that are required and not available.

3.2 GCAM Layer

The GCAM is an abstraction layer above the Cougaar code that represents application's design. Therefore, the GCAM hides the Cougaar code implementation while providing a platform specific "environment." One of the important issues is a separation between the GDAM and the GCAM levels. The GDAM level represents requirements and the GCAM level represents design. Each level performs one mapping. The GDAM level maps from requirements to design which then serves as input to the GCAM level. The GCAM level then maps from the provided design to code. Therefore, the GCAM level is taking as an input the design (GCAM representation) that contains constraints, references to the GCAM components, etc. A repository of components contains detailed descriptions of individual GCAM components in a form of "beans." The GCAM engine is assembling the code segments of the GCAM components from the repository and augments them with code generated from constraints and other design information. The resulting code, combined with the configuration information, provides a developed application.

In addition to the general system-wide assumptions, the following assumptions are specific to the GCAM layer. The GCAM is essentially a design level representation of the Cougaar system.

- As the Cougaar system is revised, the revisions will be reflected in the GCAM layer.

- The developers will write Cougaar code to encode details that cannot be represented using GCAM components.
- The code generated by the system is not intended to be modified by developers. The code generator is optimized for runtime performance and simplicity.
- The GCAM engine does not have optimization capabilities and hence the generated code might not be as efficient as manually written code. The GCAM engine does not support model debugging capabilities.

4 CMDA Model

The GDAM requirements necessitated the development of a model representation that is both versatile (to represent domain information) and familiar (to the SMEs and developers). Based on studies conducted, there is enough confidence to choose workflow as the medium to represent the generic domain model. Workflow is familiar to both SMEs and developers and charts out the working mechanism of the intended system. Further, the structure of the workflow (essentially boxes and arrows) is both generic (to represent most domain information) and extensible (to support addition and modifications of GDAM components). However, it should be noted that workflow does not capture all the requisite information. The information that is not captured include:

- Deployment and configuration information,
- Information pertaining to GUI such as screen layout and user interactions,
- Domain and system level constraints, and
- Business rules.

It is necessary to develop and refine the software artifact generation mechanism based on the information that is captured using workflow. Information that cannot be represented using workflow can be captured either by extending the workflow model (to record domain and system wide constraints) or by creating "threads" that will "run" in parallel to the workflow thread.

Figure 3 shows the different threads that exist in the developed tool. The threads are designed to capture information pertaining to (1) workflow, (2) GUI layout, and (3) deployment. While the structure and semantics of the workflow thread are known, the details about the GUI and deployment threads are being worked out. The GDAM model representation consists of the Task model for GUI, Workflow model for Agent code and deployment model for deployment code. The models are transformed into corresponding XML representations by the GDAM engine. The GCAM engine reads in the XML, aggregates and correlates the information to produce the code artifacts. Higher priority is assigned to developing and refining the workflow thread.

4.1 Domain Components presented in Cougaar

The Cougaar system provides mechanisms to encode domain knowledge directly in the code. The domain that Cougaar implement is the planning domain for

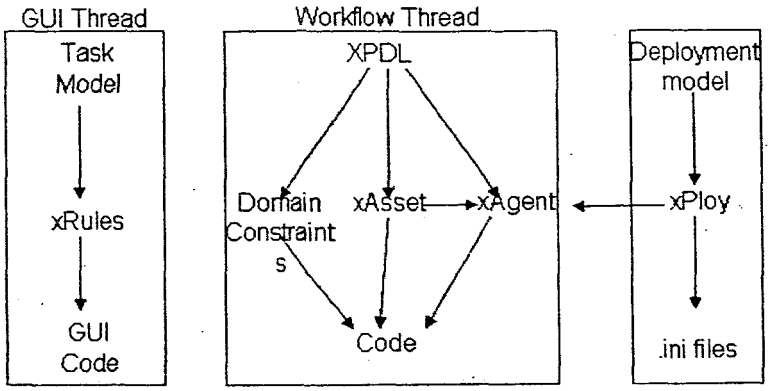

Fig. 3. Threads

which the generic-domain components present in the Cougaar were identified. As the project moves forwards, more detailed study will be performed. The two important domain components found in the Cougaar were the task component and the asset component.

Cougaar defines a task as "A requirement or request from one agent to another to perform or plan a particular operation." The tasks are implemented in the planning domain library and are used by agents to let another agents perform a job or plan the execution of a job.

Cougaar defines an asset as "Resources assigned to the task." Any asset instance will have two key attributes: (1) a reference to its prototype and (2) a reference to the item identification property group. Assets are also implemented in the planning domain library.

4.2 GDAM Representation

Current Cougaar application development practices were analyzed and used to define the GDAM representation. The workflow model is the computational model used by Cougaar developers. Some of its functionality has been already incorporated in the Cougaar based code. As a consequence, the workflow model and its underlying XML Processing Description Language (XPDL) format have been selected as for specifying application requirements [6]. The underlying platform independent GDAM model subsumes the workflow model by using the basic components of the workflow model as templates for the part of GDAM components.

XPDL, defined by Workflow Management Coalition (WfMC), provides a framework for implementing business process management and workflow engines, and for designing, analyzing, and exchanging business processes. Further, XPDL is extensible and versatile to handle information used by a variety of different tools.

While XPDL provides excellent mechanisms to define and record workflow processes, certain customization was needed. The customizations include:

- **Type Declarations**: The type declarations were used to define the assets at the domain level. The SMEs will define and declare the primitive types, PropertyGroups (PGs) and assets using type declarations. The primitive types or elements within a PG were recorded as basic type in XPDL. The basic types were then grouped into a record type, which will represent the PG. The PGs are then grouped into a record to form the asset. The type declarations in XPDL provide all the capabilities required to define an asset.
- **Abstractions**: The generic notations of XPDL were abstracted to represent Cougaar concepts. The agents were represented using participants and the behavior of the agents was described using activities. The transitions represented the tasks generated by agents.
- **Constraint enforcement**: The condition tags present in the XPDL was extended to support constraint representations. While XPDL has many useful features, it lacks some of the required structure and constraint capabilities. As a consequence, the Object Constraint Language (OCL) is selected to capture this information [7]. The OCL constraints are includes in the XPDL as pre- and post- conditions thus eliminating free-text constraints from the original XDPL format.
- **Extended attributes**: The extended attributes section was used to describe Cougaar specific semantics such as tasks, assets, and allocations.

It should be noted that care was taken to extend the XPDL without breaking the XML structure defined by the WfMC. This was done to allow the XPDL file to be loaded in any standard workflow editor that supports XPDL.

4.3 GDAM Components

The current structure and semantics of GDAM components have provision to specify constraints (pre and post conditions), documentation section, need revisions to incorporate fragments of design diagrams, mapping criteria, The workflow component describes the participant, activity and transition elements.

The participant component which is used to represent Agent is defined in XPDL under the participants tag. Each participant has two attributes: ID (unique Id used to reference the participant within the workflow model) and Name (user specified name, which need not be unique). The participant component also contains the tag ParticipantType which is used by XPDL to identify the type of participant.

The activity component is used to describe the behavior. The activity component, described inside Activities tag, consist of two attributes: ID (unique Id used to reference the activity within the workflow model) and Name (user specified name, which need not be unique). The activity component provides details about a particular behavior, which are mapped into Plugins during transformations. The Activity component has performer tag to identify which Agents

behavior is being defined, transition restrictions tag to reference the constraints of a particular Task, and an extended attribute namely asset to identify the asset used by the activity. An activity component can occur more than once in the workflow model. The first occurrence of activity component is mapped into a new Plugin and subsequent occurrences result in appending the Plugins behavior. The Plugins behavior is appended by appending the subscription and action subsets of the Plugin.

The asset component is used to describe the resources attached to tasks. The asset component is described using XPDLs type declarations. The primitive types or elements within a PG were recorded as basic type in XPDL. The basic types were then grouped into a record type, which will represent the PG. The PGs are then grouped into a record to form the asset. The TypeDeclaration tag, consist of two attributes: ID (unique Id used to reference the type within the workflow model) and Name (user specified name, which need not be unique). The TypeDeclaration also lists whether the type is basic type or record type. If the type is a record, the members of the record are listed.

5 CMDA Implementation

The graphical user interface (GUI) for the developed CMDA tool has been implemented as an editor using the Eclipse IDE [8, 9]. The editor allows editing and validation of XPDL data in both text and graphical formats. The XPDL is loaded into the editor, with the workflow displayed. The editor connects to the repository of components. The user drags components from a palette (representing what's available in the repository) assigning the activity to a new instance of the component, which can have all its properties set in a GUI. The editor shows any validation errors detected by the validating compiler. The instantiation data (component name and property values) are stored in the XPDL as extended attributes. The editor also shows a set of available resources, which can be assigned to each activity. As these resources are assigned, they are stored in the XPDL as extended attributes. Completed requirements include a fully defined components with parameters, roles, and deployment data.

Since the entire system is a component itself, with deployment information added, the editor is used to edit any inner component as well. The components are defined in a UML-like XML-based language [10] where an XML schema is defined for specifying components that can be automatically converted to an EMF [11] model. Eclipse's EMF is a modeling system similar to the Meta-Object Facility (MOF) [11]. Those similarities enable the use EMF and the related tools for easy conversion to a UML representation. The UML representation, in addition to documentation generation, provides a better understanding of the application under development.

The characteristics of the metamodel are determined from the parameterization of Cougaar components, related constraints and properties. Components must define properties that can be queried and derived. Interconnected components work together as agents and societies of agents. Composition of components

is specified using graphs describing interconnected and configured components. The graphs can be saved and reused.

Generation of Cougaar/Java code, documentation, requirements, and test cases. depends on components that must provide information for artifact generation Deployment of components requires assignment of hosts and other computational, storage, or other type of resources while maintaining Java compatibility.

Each component maps to a UML structured component with template parameters (Figure 4). The compiler validates components and generates artifacts. The validation insures that the component is valid and is suitable for artifact generation. Artifact generation creates Java code, documentation, test cases, and requirements data.

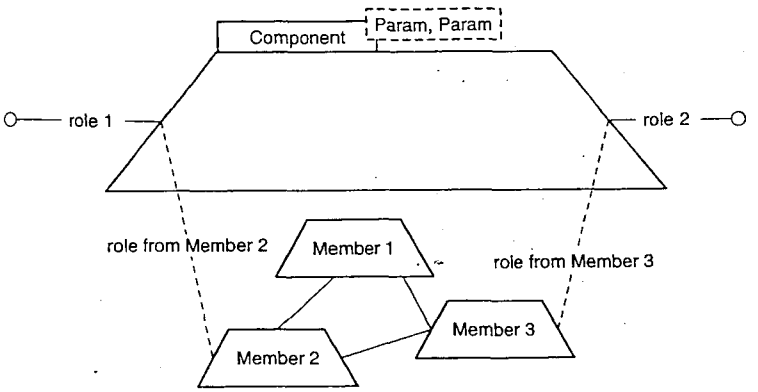

Fig. 4. Component

Components have named parameters that are defined like a very small subset of an XML schema [12]. Parameters specify a name attribute, which is matched when given a value. Parameters may also define a parent parameter, thus allowing sets of parameters and a cardinality. This allows variable numbers of sets of parameters, giving a reasonable configuration language for components.

The metamodel directly provides constraint data through constraints given in the component definition, and implicitly through the typed connections and defined restrictions on the various metamodel elements. The compiler verifies constraints to assure that a valid system can be generated.

The compiler considers the entire system as one top level component. The components are grouped into a tree of instantiations (component names coupled with values for all of their parameters) that is traversed by the compiler. The compiler calls the relevant profile mapping at each node to generate corresponding artifacts. The profile mappings use either an XML tree for the component definition or a set of EMF objects representing them in memory. The former is the serialized form of the latter. Extensible Stylesheet Language (XSL) Trans-

formations (XSLT) [13] or Eclipse's Java Emitter Template (JET) [14, 15] are then used to generate the artifacts.

Components can specify *roles*, named interconnections with other components, that specify data types sent and received over them. Roles are special types of parameters which are fully initialized only with references to other component instances. They also cannot have inner roles or any such hierarchy.

Deployment data is considered a special type of non-hierarchical parameter. Deployment data are not fixed values. They are expression usable for deriving the value when the system is deployed.

Components can specify inner *member* components to define the inner structure. These member components are initialized and connected together. Their parameters, connections, and deployment information have static values or OCL expressions [7] based on the component's parameter data. OCL expressions provide additional information to object-oriented models, including constraints, queries, referencing values, stating conditions and business rules. Each value is expressed using OCL constants or using OCL expressions that allow their derivation. The component can define its properties as the values of properties in its member components, possibly with some modification and renaming.

While the definitions immediately provide useful descriptions of the system, they do not directly provide code, test cases, etc. The compiler, in some cases, needs "help" from the component definitions to create code, test cases, and related artifacts. Each component specifies the name of a *Profile Mapping* that links the component to a set of definitions for how the artifacts are generated. Each profile mapping handles different categories of components, such as Cougaar Plugins, Agents, or Societies.

6 Conclusions

Cougaar is complex requiring considerable mappings and transforms. MDA provides a systematic way of capturing requirements and mapping them from PIM to PSM and ultimately to the code level. The developed CMDA framework is an MDA based approach for the Cougaar agent-based architecture. It enables automatic transformation of the application requirements, expressed in the XPDL format, into a platform-independent, GDAM representation. The artifacts are generated from models assembled using components that contain information related to requirement, design, code, test and documentation details for that component, along with transformation information. Platform-specific GCAM components are derived from the metamodel and then converted into Cougaar/Java code. The CMDA combines assembly approach with transformations to generate the artifacts. While the CMDA-based approach uses the Cougaar architecture, it is applicable to other agent-based architectures.

7 Acknowledgements

This work has been supported, in part, by the DARPA STTR grant "AMIIE Phase II — Cougaar Model Driven Architecture Project," (Cougaar Software, Inc.) subcontract number CSI-2003-01. We would like to acknowledge the efforts, ideas, and support that we received from our research team including Todd Carrico, Sandy Ronston, Tim Tschampel, and Boby George.

References

1. —: Cougaar architecture document. Technical report, BBN Technologies (2004) Version for Cougaar 11.2.
2. —: Cougaar developers guide. Technical report, BBN Technologies (2004) Version for Cougaar 11.2.
3. Arlow, J., Neustadt, I.: Enterprise Patterns and MDA: Building Better Software with Archetype Patterns and UML. Addison-Wesley, Boston (2003)
4. Kleppe, A., Warmer, J., Bast, W.: MDA Explained: The Model Driven Architecture: Practice and Promise. Addison-Wesley, Boston (2003)
5. Weis, T., Ulbrich, A., Geihs, K.: Model metamorphosis. IEEE Software **20** (2003) 46–51
6. Workflow Management Coalition: (Workflow process definition interface – XML process definition language (XPDL)) http://www.wfmc.org/standards/docs/TC-1025_10_xpdl_102502.pdf.
7. Warmer, J., Kleppe, A.: Object Constraint Language, The: Getting Your Models Ready for MDA. Second edn. Addison Wesley Professional (2004)
8. Clayberg, E., Rubel, D.: Eclipse: Building Commercial-Quality Plug-ins. The Eclipse Series. Addison-Wesley, Boston (2004)
9. Gamma, E., Beck, K.: Contributing to Eclipse: Principles, Patterns, and Plug-Ins. The Eclipse Series. Addison-Welsey, Boston (2004)
10. Rumbaugh, J., Jacobson, I., Booch, G.: The Unified Modeling Language Reference Manual. Addison-Wesley Publishing Co. (2004)
11. Budinsky, F., Steinberg, D., Merks, E., Ellersick, R., Grose, T.J.: Eclipse Modeling Framework. Addison-Wesley Publishing Co. (2003)
12. McLaughlin, B.: Java^{TM} & XML Data Binding. O'Reilly, Beijing (2002)
13. Tidwell, D.: XSLT. O'Reilly, Beijing (2001)
14. Azzuri Ltd.: (JET tutorial part 1 (introduction to JET)) http://eclipse.org/articles/Article-JET/jet_tutorial1.html.
15. Azzuri Ltd.: (JET tutorial part 2 (write code that writes code)) http://eclipse.org/articles/Article-JET/jet_tutorial2.html.