Model-Based Evolution of Collaborative Agent-Based Systems

Shawn A. Bohner[1], Denis Gračanin[1], Michael G. Hinchey[2] and Mohamed Eltoweissy[3]

[1]Virginia Tech
Dept. of Computer Science
Blacksburg, VA 24061, USA

[2]Loyola College in Maryland
Dept. of Computer Science
Baltimore, MD 21210, USA

[3]Virginia Tech
Dept. of Electrical and
Computer Engineering
Arlington, VA 22203, USA

Abstract

As demands for behaviorally sophisticated software grow, agent-based systems are increasingly being employed. Software agents are frequently applied to large, complex systems that involve interdisciplinary development teams. These complex systems have proved to be challenging to develop and evolve, even for the most competent software engineers. Taking lessons learned in other engineering disciplines such as computer and architectural engineering we investigated a model-based engineering approach called Model-Driven Architecture (MDA) to automate, whenever possible, the development and evolution of agent-based applications. In our investigation, we use the Cognitive Agent Architecture (Cougaar); one of the most mature and sophisticated collaborative agent-based architectures. MDA and Cougaar served as the primary components and implementation platform for our research. In this paper we present our approach and demonstrate how MDA is effective for producing sophisticated agent-based systems. A key challenge was found in designing a flexible meta-model framework that would accommodate both top-down domain information and bottom-up platform specific constructs, as well as the transformations and mappings between them. We employed a General Domain Application Model (GDAM) as the platform-independent model layer and General Cougaar Application Model (GCAM) layer as the platform specific model respectively. Domain-level requirements are formulated using a XML Process Definition Language (XPDL) based graphical editor and are the refined through a series of model transformations (via the underlying metamodel) to systematically generate the agent-based software system. Through an illustrative case-study, we report on the feasibility, strengths and limitations of the model-based approach as it was investigated with the Cougaar.

1. Introduction

As society increasingly depends on software, the size and complexity of software systems continues to grow making them progressively more difficult to understand and evolve. Moreover, the nature of software change and its concomitant complexity has turned a corner with the advent of web services, collaborative agent-based systems, self-healing systems, reconfigurable computing, and the like. Software complexity has compounded volume (structure) and interaction (social) properties as the Internet has enabled software functionality to be delivered as services. To respond to the sheer volume, and consequent complexity, the software community has increasingly embraced model-based engineering principles. Similarly on the operational side, agent systems have been employed to collaboratively deal with tasking in systems that require very complex behaviors and decisions. Suffice it to say, these agent-based applications are sophisticated, complex, and very hard to develop.

Interdisciplinary development of these systems has emerged as a way to ensure that relevant requirements are rendered properly as the abstract models from the problem domain evolve into increasingly more detailed and complete ones used to generate software. Development and maintenance environments must support this inherent part of producing today's highly integrated and complex computing systems. Software architecture provides a framework to understand dependencies that exist between the various components, connections, and configurations reflected in the requirements. These emergent technologies provide a reasonable basis for addressing complexity issues by separating concerns (integration, interoperability, decision support, and the like) and allowing agents to provide the necessary processing. The task orientation, coupled with intelligent agents, provides a strategic and holistic environment for designing large and complex computer-based systems. These systems may support logistics management, battlefield management, supply-chain management, to mention but a few.

The Cognitive Agent Architecture (Cougaar) can be characterized in the same way. Cougaar is an open source, distributed agent architecture [2], a result of approximately eight years of development for the Defense Advanced Research Projects Agency (DARPA) under the Advanced Logistics Program (ALP) and the Ultra*Log program [3]. The primary focus of development has been on very large-scale, distributed applications that are characterized by hierarchical task decompositions, such as military logistics planning and execution. In addition, during the last four years, particular attention has been given to fault tolerance, scalability, and security.

Many of today's software systems exhibit characteristics that align with agent systems. They are task-oriented and often adaptive, and may involve autonomic behaviors, or engage in collaborative or competitive activities. These and other aspects make it challenging to develop and evolve agent-based systems in a timely fashion. To address this key challenge, we investigate the Object Management Group's (OMG) Model Driven Architecture (MDA) approach [6, 17, 24], which aims at separating application logic from the underlying technologies to improve reusability, portability and development processes. The underlying premise is that business knowledge should be long-lived, whereas technical concerns are generally short-lived and limited to a given technology. MDA provides a means of automating the development process to a significant degree. Additionally, we examine how changes to the software system are characterized and reasoned about in the model-based environment.

In some respects, MDA is an advanced perspective on well-known essential systems development concepts practiced over the years (albeit frequently practiced poorly). OMG promotes MDA advocating Unified Modeling Language (UML) as the modeling technology at the various levels. MDA endeavors to achieve high portability, interoperability, and reusability through architectural separation of concerns; hinging on the long-established concept of separating the operational system specification from the details of how that system implements those capabilities on its respective platform(s). That is, separate the logical operational models (external view) from the physical design for platform implementations.

Development of agent-based systems can be thought of as the evolution of abstract requirements into a concrete software system. Starting with requirements that must be refined and elaborated, the system's evolution is achieved through a successive series of transformations. For non-trivial systems, this can be complex, time consuming, and prone to error as software engineers work together to develop the requisite components, assemble them, and verify that they meet specifications. MDA, also known as Model Driven Development [7], represents an emerging approach for organizing this evolution and its resulting artifacts. Through a successive series of computationally-independent, platform-independent and platform-specific model transformations, MDA facilitates the generation of software systems. A metamodeling foundation [7] allows efficient implementation of the transformation process. The Eclipse tools [13], including the Eclipse Modeling Framework (EMF), are used here to implement an MDA framework based on the Cougaar platform.

Figure 1 illustrates our approach and its basic MDA concepts pertaining to Cougaar applications. Starting with an often-abstract Computation Independent Model (CIM) such as a process workflow or functional description, the Platform Independent Model (PIM) is derived through elaborations and map-pings between the original concepts and the PIM renderings. Once the PIM is sufficiently refined and sta-ble, the Platform Specific Models (PSM) are derived through further elaborations and refinements. The PSMs are transformed into operational systems.

The CIM layer is where vernacular specific to the problem domain is defined, constraints are placed on the solution, and specific requirements illumined. Artifacts in the CIM layer focus largely on the system requirements and their environment to provide appropriate vocabulary and context (e.g., domain models, use case models, conceptual classes). The CIM layer contains no processing or implementation details. Instead, it conveys non-functional requirements such as business constraints, deployment constraints, and performance constraints as well as functional constraints.

The PIM provides the architecture, the logical design

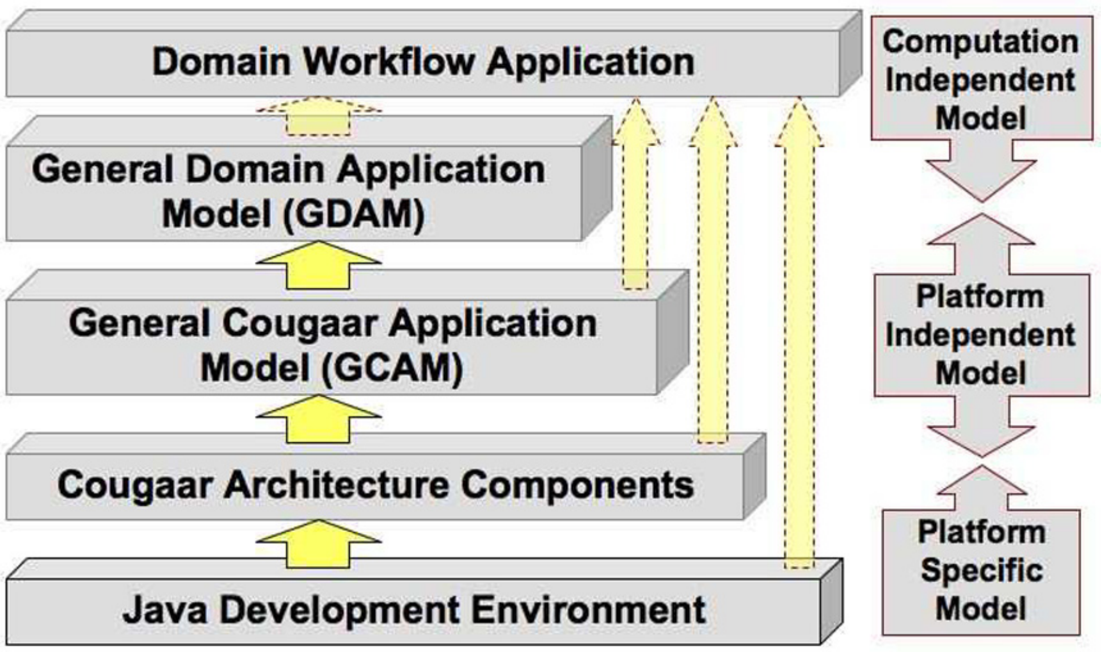
Domain Workflow Application
General Domain Application Model (GDAM)
General Cougaar Application Model (GCAM)
Cougaar Architecture Components
Java Development Environment
Computation Independent Model
Platform Independent Model
Platform Specific Model

Figure 1. Conceptual Cougaar Model Framework

plan, but not the execution of the plan in a tangible form. Beyond high-level services, the problem domain itself must be modeled from a processing per-spective. The PIM is where the logical components of the system, their behaviors, and interactions are modeled. PIM artifacts focus on modeling what the system should do from an external or logical perspec-tive. Structural and semantic information on the types of components and their interactions (e.g., design classes, inter-action and state diagrams) are rendered in UML, the defacto modeling language for MDA.

Mapping from the PIM to the PSM, is a critical element of MDAs approach. Mappings from PIM representations to those that implement the features or functions directly in the platform specific technologies are the delineation point where there is considerable leverage in MDA. This mapping allows an orderly transition from one platform to another. But the utility does not stop there. Like the PIM, there are opportunities to have layers within the PSM to produce intermediate transformations on the way to the executable system. These models range from detailed behavior models to source code used in constructing the system.

This research concentrates on understanding and applying the MDA approach in the Cougaar agent-based architecture. We explore ways of using MDA to facilitate the development of agent-based applications by domain experts and software engineering staff, by abstracting and programming at a higher level — the domain level. We investigate how Cougaar components may be composed into a General Cougaar Application Model (GCAM) and used to develop a General Domain Application Model (GDAM) for specifying and generating software applications. The model-based approach to producing software suggests that software change will be addressed at the appropriate abstraction level. That is, if a change is made at the application domain level, it should be supported through the transformation and mapping process. Hence, we examine the elements necessary to make this possible. While we apply MDA to Cougaar specifically, we believe that the principles are general enough to apply to other agent-based architectures. The main contributions of this paper are: (1) The CMDA metamodel with a novel transformation strategy; (2) Demonstrated feasibility of MDA on a sophisticated technology that must scale (such as Collaborative Agent-Based Systems; and (3) Integrated levels of modeling (we can skip levels).

The remainder of the paper is organized as follows. Section 2 provides an overview of Cougaar and its capabilities. Section 3 describes the new Cougaar MDA framework. Section 4 outlines some of the underlying structures for the transformations. Section 5 provides a brief case study to illustrate the approach. Section 6 provides a discussion of related work and Section 7 concludes the paper.

2. Cougaar Model-Driven Architecture

Cougaar provides a platform for developing complex agent-based applications that can be self-aware, self-healing, self-preserving and fault-tolerant. Like many sophisticated software development technologies, the key challenge is the efficient and timely development of these large-scale applications.

Central to this research effort is an effective technology for developing agent-based systems well-suited for tasking and workflow common in today's business environment. Cougaar is an open-source, Java-based distributed agent architecture for developing large-scale distributed agent-based applications characterized by hierarchical task decompositions [2, 3]. The Cougaar environment enables developers to construct collaborative, agent-based applications that involve high-level tasking, determine suitable processes and activities, and allocate appropriate resources to complete the tasking. From an information systems workflow perspective, Cougaar agents collaboratively accomplish various tasks based on the functional business processes with which they are configured [8].

Cougaar has been used for rapid, large scale, distributed logistics planning and conclusive research has been performed for fault tolerance, scalability, and security for enhancing the survivability of distributed agent-based systems operating in changing environments. A Cougaar agent consists primarily of a blackboard and a set of plugins. The blackboard is a container of objects that follows publish/subscribe semantics. The agent is characterized by one or more plugins that are referentially uncoupled (i.e., they do not know about each other). Plugins implement the core business logic associated with the agent. They publish objects, remove objects, publish changes to existing objects via the blackboard, or create subscriptions to be notified when objects are added, removed or changed in the blackboard.

While agents collaborate with other agents, they do not send messages directly to each other. Instead, a concept of task is used for this purpose. Each task creates an "information channel" used within the society for passing down requirements, and responses going back [2]. Then the agent must be located to allocate the task by creating a subscription that examines the roles or property groups of organizations in the local blackboard. Once the proper organization is found, the task containing the object to be sent to the other agent is allocated to that organization by creating an allocation and publishing it to the blackboard. The Cougaar communication infrastructure then ensures that the task is sent to the specified organization's and the specified agent's blackboard. Details of this are presented later in the illustrative case study.

2.1. CMDA Framework

Cougaar provides a higher level of abstraction than the underlying Java in which it is implemented. Consequently, the conceptual distance between the design abstractions and the source code is somewhat reduced. However, the gap between a domain model (needs and requirements) and a design model is still substantial and the MDA approach is used to bridge that distance and facilitate automatic generation of executable applications.

We use a framework based on the Cougaar Model-Driven Architecture (CMDA),to describe the automatic application generation [14, 15]. CMDA prescribes the kinds of models to be used, how those models may be prepared, and relationships between the various kinds of models. Building on Figure 1, Figure 2 illustrates the CMDA framework and exposing its key elements more concretely.

The Cougaar platform-specific architecture models are expressed in the General Cougaar Application Model (GCAM) [2]. The GCAM provides representation in its model of the basic constructs of Cougaar [3]. The core representation includes: Agents, Communities, Societies, Plugins, Assets, Preferences, Knowledge Rules, Policies, Rules, Constraints, Events, Facts, Services, Service Providers, Tasks, Nodes, Subscriptions, Predicate, Messages, Directives, Logic Providers, Hosts, Domains, and Configuration. Beneath these are are the Java and lower-level constructs relevant to the implementation platform.

The platform-independent General Domain Application Model (GDAM) expresses the domain models and vernacular, and builds upon the foundation of GCAM. That is, the problem domain and models, the requirements and designs collectively define the contents of the GDAM. There are two potentially conflicting implications of the GDAM functionality. First, domain knowledge and application requirements should be captured in a manner that is computationally independent. Second, there should be a well-defined structure and relationships among requirements to allow for an automatic transformation of the requirements/constraints into an internal GDAM representation that can be later transformed into a GCAM representation. To address this conflict, the fol-

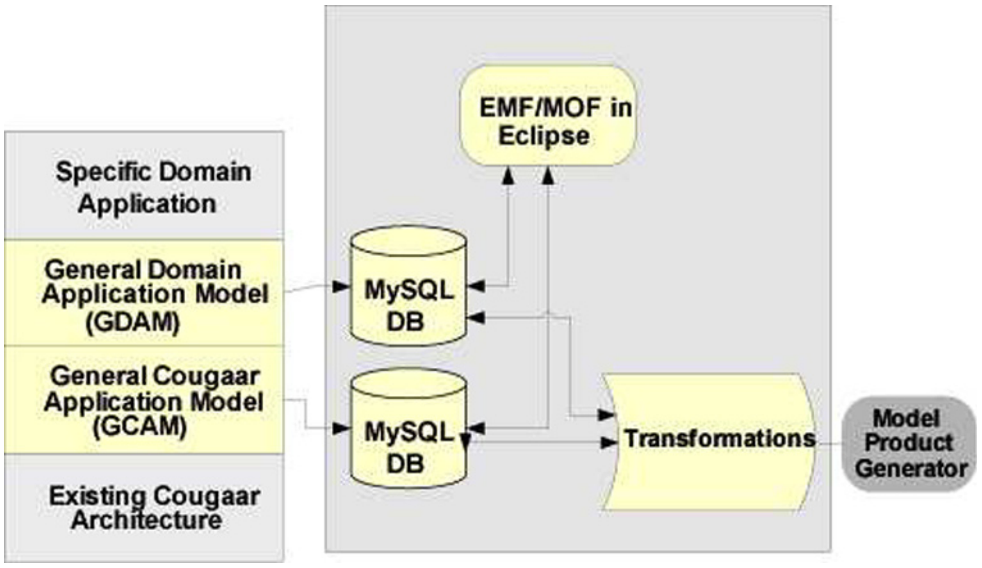
Specific Domain Application
General Domain Application Model (GDAM)
General Cougaar Application Model (GCAM)
Existing Cougaar Architecture
EMF/MOF in Eclipse
MySQL DB
MySQL DB
Transformations
Model Product Generator

Figure 2. Model Driven Architecture Framework

lowing decisions were made:

- Transformations between the CIM and PIM should be lightweight. The platform independent transformations should subsume the computationally-independent ones thus requiring only a simple transformation between the two.
- The business logic must be embedded within the computationally-independent representation to enforce constraints. The constraint language must be simple and easily transformed into code that can be integrated within the platform.
- The platform-specific elements of application configuration and deployment are treated separately from the application requirements.
- User interactions and the user interface represent a separate and important challenge. Automatic or semi-automatic user interface generation based on the application requirements is not unique; i.e., there can be many different user interface designs. Such designs can be customized based on the domain preferences. We chose to leave this area to our research partners as part of their scope and it is not reported in this research.

3. Overall Approach

In this section we outline the overall approach to CMDA and detail some key elements that make it effective. At the core of the CMDA is the metamodel that defines how components are defined and how they are allowed to interconnect. The components are stored within a database repository, which can be queried by a compiler and an editor. Transforms and mappings are the glue in MDA holding together what would otherwise be an software artifact reuse approach. These connections offer a key architecture element for the relating of models and concepts.

3.1. Transformation

For this research effort, capabilities of various formal methods were evaluated by conducting an in-depth survey of some of the key formal methods used for specifying agent-based systems. Formal methods were considered based on their Object-Oriented (OO) modeling support, usability, tool support and concurrency support. Support for representing objects was a key selection criterion, as Cougaar is an object-oriented system and includes the ability to represent objects and their constraints such as pre-conditions and post-conditions. Interoperable tool support was another important criterion for selection since CMDA was to be interfaced with the Eclipse platform [13]. Tool support also includes GUI interfaces to perform consistency checks, type checking and code generation.

The usability criterion gave an indication of the amount of difficulty in learning and using the formal method, with a good rate indicating that the method's syntax were similar to popular programming languages and easy to learn. The scalability criterion are the fourth key criterion that indicates whether the representation was scalable enough to support complex Cougaar systems. The formal basis criterion provided insights into the richness of the formal methods to describe the system completely and correctly.

The transformation challenges entail using multiple representations to represent the CDMA system components [8]. The CMDA project endeavors to build a developer environment that offer developers' components which can be aggregated to represent the system in the workflow, GDAM and GCAM levels. Each of the components, named as Workflow Beans, GDAM Beans and Cougaar Beans, respectively, (similar to the Java beans concept) contain sections of software artifacts and related information pertaining to that bean. Some example sections of the software artifacts that beans contain include:

1. The model from which the transformer gleans the partial set of requirements,
2. The model from which the system's design model is assembled by the transformer,
3. References to the lower-level beans, or links to Java code which can implement the bean (these are traversed by the transformer while assembling the system's components), and
4. Test case fragments that contain information on how to assembly the unit test cases for the beans.

Further, the bean contains documentation information such as a description of the bean, and constraints pertaining to data, operation and connections with other beans. Constraints may be divided into two groups:

1. Port constraints, detailing constraints on input ports of the bean, and
2. Role constraints, detailing the restrictions the bean has on the roles or services the bean provides or supports.

The contents and size of the sections and information in a bean are influenced by the abstract layer to which the bean belongs. The model sections of each bean are represented using the Unified Modeling Language (UML) [21], while the VDM++ or the Object Constraint Language (OCL) representations may be used to delineate connector and other constraint information. The code section contain links to Java code libraries at the GCAM level and pointers to lower levels in the rest of the abstraction layers. The requirements might be a combination of XPDL, text, and UML diagrams, while the constraints also contain mapping (or connection) information that are mostly rule-based with some formalizations applied.

The workflow of the CMDA system proceeds with the developer assembling the system by picking the right workflow bean components and connecting them to represent the workflow. Constraints pertaining to connections are encoded in the beans. When developers attempt to

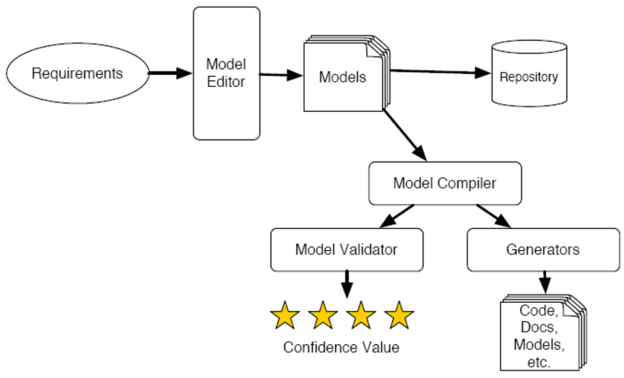

Figure 3. System overview

connect components illegitimately, they are shown a detailed error message. Once the workflow of the system is built, it can be verified for consistency. Figure 3 shows the system overview with its respective major elements.

The developer is then shown a list of GDAM beans that can be chosen to map a particular workflow bean. The system will list only related GDAM beans based on the constraints specified by the developer at the workflow level. The rationale to allow developers to choose the right component is to allow developers to make design decisions with the system assisting them (by showing a list of possible solutions and patterns).

Similarly, GDAM beans are mapped into Cougaar beans. In all layers, as and when required, the developer will input the necessary information to satisfy the completeness and correctness of the bean component. The usability of the system can be improved by developing wrappers that would mask the semantic complexities of the representation language. Once the models are built, the transformation engine will traverse through the beans at each level and generate the software artifacts based on predefined transformation rules.

Figure 4 shows the CMDA system representation. The CMDA allows domain experts to specify the intended application using a high-level descriptive language. The descriptive language comprises of a combination of a custom UML profile, Object Constraint Language (OCL), and templates for code (Java) and documentation. The UML profile is used to delineate the domain and application models of the intended system. The OCL is used to describe the domain and application specific constraints that the generated system must adhere to. The templates for code and documentation are the base structure of the generated artifacts. The templates are populated with parameters that the user inputs into the domain and application models, and the required software artifacts are generated. While the elements of the CMDA system will be

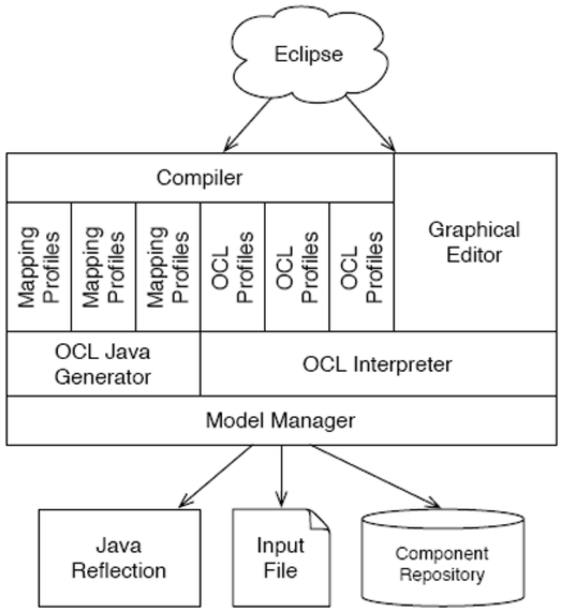

Figure 4. CMDA System Representation

described in detail in the subsequent sections, a quick description is given below to provide an overview of the system. The CMDA system is comprised of the following key parts:

Graphical Editor: allows the creation and editing of the system description. It allows a mouse-based graphical environment for system specification, tied to the OCL sub-system for full specification and interactive validation. The domain and application models of the intended system are created using the editor. The models are assembled from the components available in the component repository. The editor also facilitates users (Cougaar developers) to create a new domain and new Cougaar components.

Component Repository: is comprised of a database that is used to store components and their revisions. The repository has support for version control in order to facilitate smoother distributed collaborative development and publishing of components. The repository can be extended using policies and procedures to enforce effective knowledge management.

Model Manager: provides a unified view of all the components and their contents, either as a Java class visible to the Virtual Machine (VM) via reflection, to the repository, or in the file being processed. The intended system's design documentation can be generated from the information provided by the model manager.

OCL Interpreter: built on top of ANTLR [20]. The interpreter provides validation of constraints that are defined in the component definitions and supports the evaluation of domain-level and application-level constraints that are used to describe the behavior of the intended system.

OCL Java Generator: used to generate Java source code equivalent to the OCL constraints described by the user.

A Compiler: is a translator that converts, with the help of the mapping and OCL profiles, the input high-level description of the intended system into it is equivalent software artifacts such as Java source code, test cases, and documentation (requirements and design).

Mapping Profile: a translator that takes a configured component description and produces an artifact, such as Java source code or documentation.

OCL Profile: a translator taking a configured component and producing OCL expressions to be used by the OCL interpreter.

3.2. METAMODEL

The Cougaar development process was divided into two modeling phases. The completion of these two phases results in creating domain and application models of the intended system.

GDAM can be conceptually thought of as being similar to various programming language libraries such as MFC or Swing. The libraries abstract and modularize the commonly-used functions, thereby helping users to focus on encoding business logic. The GDAM can also be viewed as a layer that roughly corresponds to the Platform Independent Model (PIM) in the MDA. The PIM is used to represent the system's business functionality without including any technical aspects. The MDA approach advocates converting PIM models into Platform-Specific Models (PSMs) through a series of transformations, where the PIM is iteratively made more platform specific, ending in the PSM. Hence, GDAM allows domain experts to represent the specification 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.

GCAM is an abstraction layer above the Cougaar code that represents the application's design. Therefore, the GCAM hides the Cougaar code implementation while providing a platform specific "environment." GCAM represents the PSMs of the MDA concept. The PSMs are converted into software artifacts using transformation

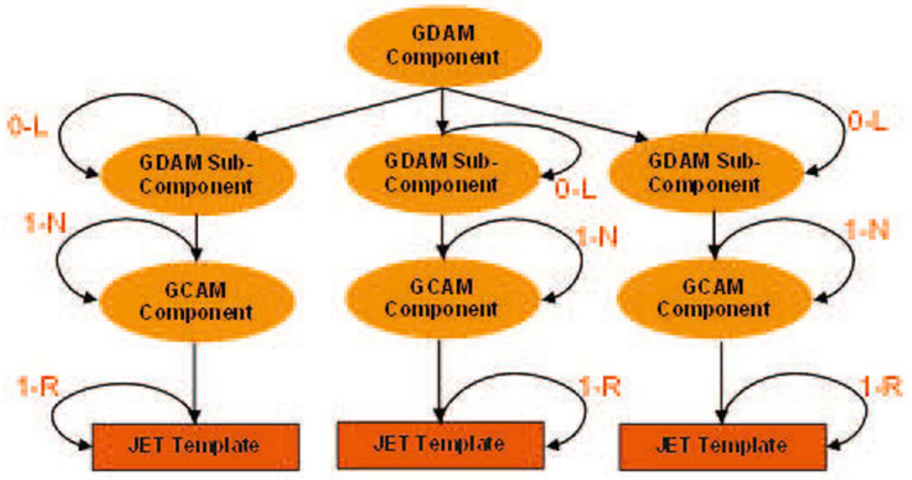

Figure 5. Metamodel

based on templates. That allows a user-developed application model to be converted into software artifacts based on pre-defined templates.

The meta-model of the CMDA system supports the definition of GDAM and GCAM. In order to have a smooth translation between GDAM and GCAM, and to facilitate multiple sub-layers within the two models, the same meta-model was used to define both models. The meta-model is recursive in nature and allows users to specify the intended application as a hierarchy of components as shown in Figure 5.

Each component contains instances of other components, at same or lower layers, and the components at the lowest level (called leaf components) should be specific enough for generation of the artifacts. The leaf components are attached to the templates that contain information on how to process the parameters of the component and generate the required artifacts (Figure 6).

The meta-model simultaneously specifies several parts of the intended system. These include the following.

File Formats: the meta-model provides an XML schema for how input files are given to the system. The schema is directly used for storing reusable components, and is indirectly used as the format for annotating XML Process Description Language (XPDL) files to specify a deployment-ready generated system.

Language: as there is a compiler that translates from the input form to multiple artifacts, an input language specification is needed. Whether from a complete reusable component definition or XPDL, the fundamental structure and relationships of the components stay the same, specified in the meta-model.

Parsing: the meta-model is automatically generated from an XML schema into an EMF model. That model is then used to automatically generate a set of Java classes representing its entities. EMF provides inbuilt mechanisms for serializing and de-serializing this model to and from XML, as specified by the original schema.

Structural Analysis: the generated de-serialization code validates the XML against the input schema, thereby some structural analysis and error checking is performed automatically.

The meta-model continuously evolves as and when functionalities are added to the CMDA system. At present, the meta-model has at its core a small number of entities (Figure 7):

Component: the root element in the hierarchy. It is reusable and specifies the set of parameters that the component takes in when configured as part of the domain or application model. The leaf components (lowest level component) names a Mapping Profile that is used to generate artifacts based on the parameters specified in the leaf component during compilation.

Instantiation: a reference or instance of a component that combined with values for its parameters (ParameterInits) represents a part of the model. Instantiations are contained within components themselves.

Parameter: a declaration of required input data that the component expects from the user. It allows hierarchical or recursive specification and allows optional,

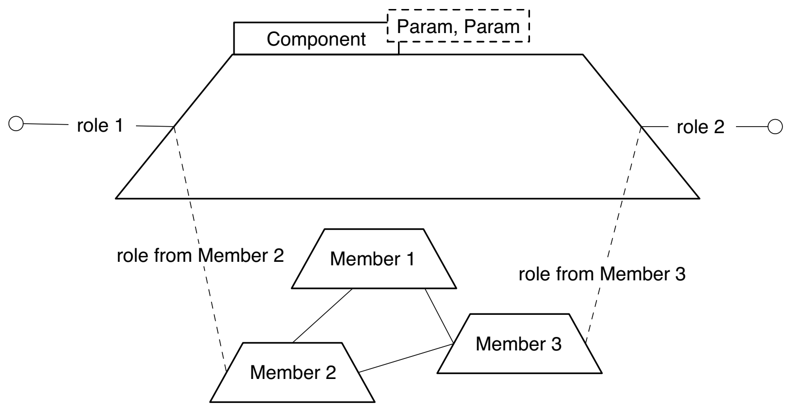

Figure 6. A Component

required, multiple, and singular values. Parameters have constraints expressed over them, specified in OCL, that validate values assigned to them.

Role: a type of parameter that specifies another instantiation. It is used for representing the connections between components.

Resource: a type of parameter that is used to specify deployment-specific values.

ParameterInit: the value for a parameter. The value can include subtypes of parameter, such as a role or resource. These are OCL expressions that result in a relevant value. The results of these OCL expressions are passed to the constraints of the original parameter in order to validate it.

Property: is a named OCL expression indicating some calculated value relevant to the component. This is used for validation, and provides a level of encapsulation around the mechanics of a component's usage.

Components are allowed to have named parameters to specify them. Unlike simple template parameters, the parameter definitions are defined like a very small subset of an XML schema. Parameters do not define their own tag names, but they do specify a `name` attribute, which is matched when given a value. They are also allowed to define a parent parameter, thus allowing sets of parameters and a cardinality. Together these two allow variable numbers of sets of parameters, giving a reasonable configuration language for components.

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

Similar to roles, deployment data is considered a special type of parameter that cannot be made hierarchical. In addition, deployment data cannot be given a true fixed-value in a component definition, only an expression usable for deriving the value when the system is deployed.

Components can specify inner *member* components that define their inner structure. These member components are initialized and connected together. Their parameters, connections, and deployment information are given static values or Object Query Language [23] expressions based on the component's parameter data. This allows the component to fix some parameters, while simply propagating down values for others.

Component properties relevant to the user are exposed through a set of properties, defined as name-value pairs. Each value is expressed using OCL. They can be either OCL constants or expressions that allow their derivation. This allows the component to define its properties as the values of properties in its member components, possibly with some modification (such as unit conversion) and renaming (to make it domain–relevant).

Unlike parameters, properties don't take values from their container; instead, they derive values from their parameters, members, or explicit initialization.

While the definitions immediately provide useful descriptions of the system, they do not directly provide Java code, test cases, requirements documentation, or UML diagrams. That is the job of a compiler. The compiler, in some cases, needs "help" from the component definitions.

Each component specifies the name of a *Profile Mapping*. This links the component to a set of definitions for how the artifacts, Java code, documentation, requirements data, and UML diagrams are generated. Each profile mapping can handle different categories of components, such as Cougaar Plugins, Agents, or Societies.

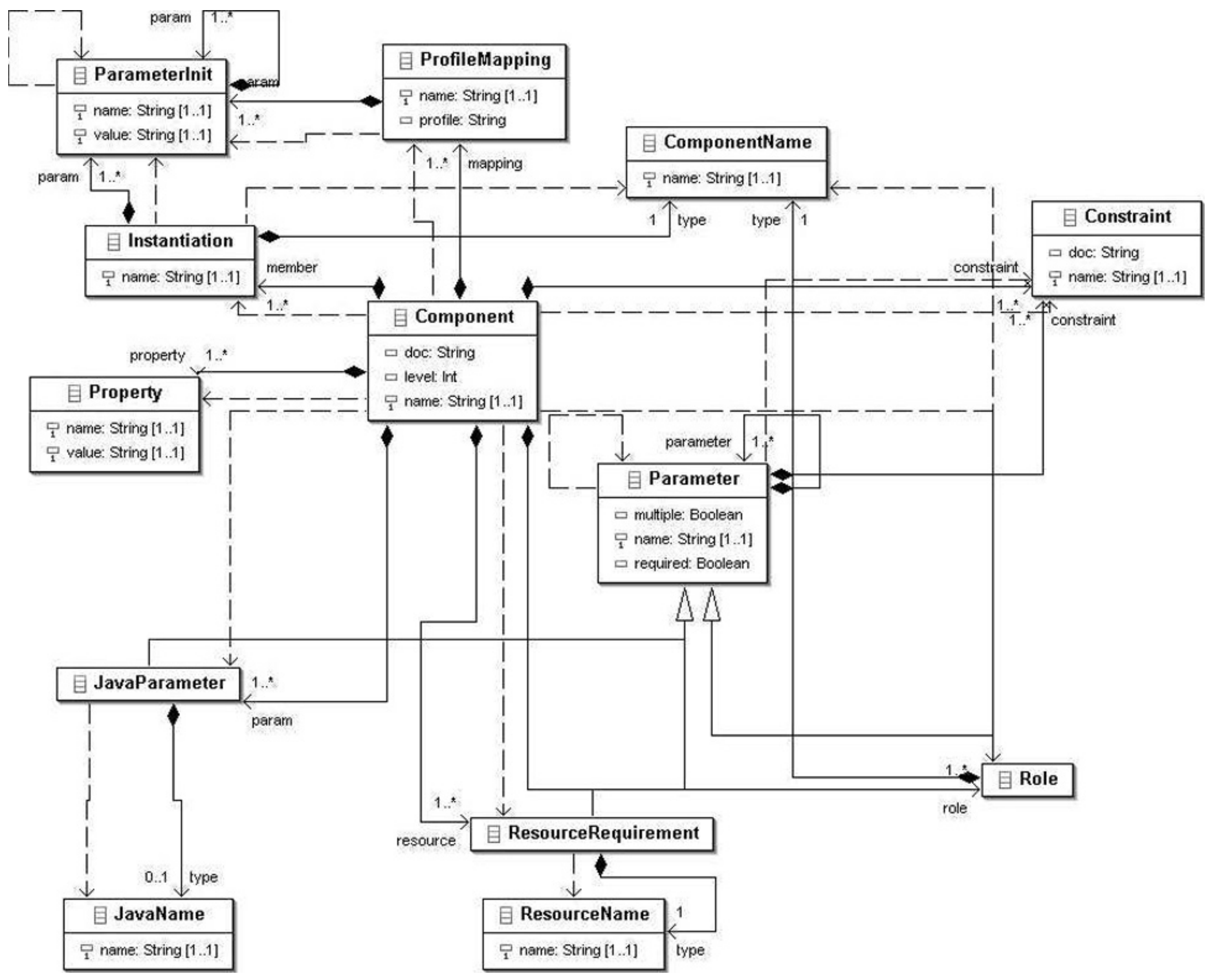

Figure 7. Metamodel UML Diagram

3.3. COMPILER

The compiler simply acts as a driver for the mapping and OCL profiles. Given a top-level component definition, it executes some of its own static validation checks, followed by the component?s constraints, followed by the execution of any mapping profiles. The component's constraints may call OCL profiles as needed. Once the compiler finishes this process for the top-level component, it proceeds through a depth-first traversal of the component's instantiations, repeating the process described above. At the end, the components have all been validated and have had artifacts generated from them. The compiler is an Eclipse builder, watching for changes to files within a project configured to use it. When a file with an extension of `.Xcomp` is modified, the compiler invokes a full rebuild. Through interaction with the Implementation Repository, it minimizes work for instantiations that haven't changed.

Mapping profiles provide the mechanism for generating artifacts from an instantiation. They extend an Eclipse extension point defined by the compiler, and implement a specified interface that generates an artifact when given an instantiation. An additional wrapper is provided that allows Java Emitter Templates (JET) to be used, by extending a second extension point instead. The compiler scans for and detects either type of extension, and maps its fully-qualified id attribute as its name. When an instantiation's component lists this name, the mapping profile is invoked to generate text for an artifact.

Similar to mapping profiles, OCL profiles provide an extension mechanism to the compiler. Instead of generating artifacts, OCL profiles provide an extension mechanism for the OCL interpreter. Components can then call these extensions in their constraints, member initializations, or property values. For example, one such function could analyze the parameters fed into a factory component, and provide an in-OCL declaration of the objects created, allowing other OCL expressions to match them.

As each instantiation can generate artifacts, those artifacts must be created and managed. By properly managing them, we can avoid wasteful regeneration of artifacts on unchanged components. The instantiation repository manages the contents of the dependent project. It creates one directory underneath the project for each component, named after it. Underneath that, it creates a numbered directory to contain each instantiation. The mappings of instantiation parameters to these directory numbers is managed via an XML file that stores the mappings as the instantiations are generated. The dependent project has an Eclipse nature that maps to a class that provides methods for accessing and manipulating the instantiations available in the repository.

The model manager provides a unified name space for the OCL interpreter, combining multiple sources of potential data. The default sources are straightforward: the input file, allowing self-referencing; Java reflection, allowing references to Java types like `java.lang.String` and the component repository, allowing references to components other than the one currently being defined. It should be noted that the details of the model manager's interfaces won't be complete until it is tested against the OCL interpreter and generator's needs. The input file's component is simply wrapped to an instance called self. That variable is required for basic operation. Using standard reflection mechanisms under `java.lang.reflect`, a complete interface is provided to the entire Java type system and all types available to the plugin?s classpath, as determined by Eclipse's management facilities. The component repository offers an interface to the model manager, to allow components to refer to others.

3.4. JET Template

The CMDA system works on two key sub-elements: the meta-model and JET templates. The meta-model is the base for building domain and application models that are used to specify the input parameters of the intended Cougaar application. The data captured in the models (GDAM and GCAM) are serialized into an XML document. The XML document referred to as `.Xcomp` file shows the "design" of the intended system. `.Xcomp` files are a serialized XML version of the EMF model used to represent the CMDA model. They are used as the input to the JET templates to produce the code. The deployment parameters are separate issues and the `.Xcomp` file does not contain any deployment parameters.

The structure of the models can be considered as a typical tree or graph with nodes representing the components, and the edges between nodes representing the linkage between components. The linkage between the components can be of two types: inter-layer and intra-layer. The intra-layer linkage represents the connection between two components residing in the same layer and the inter-layer linkage represents the *composed of* relationship between a component and its lower-layer components. The components in the upper layer have to be composed of components residing in the lower layers. The components in the lowest layer (leaf nodes) are attached to JET templates through means of profile mapping. The profile mapping specifies which JET template needs to be used for converting the parameters into artifacts.

The JET template is the template file written with a JSP-like syntax (actually a subset of the JSP syntax). It can not only express the source code that needs to be generated, but also other software artifacts such as documentation. Like the general JET template, the Cougaar JET template mainly consists of two parts: the static part and the dynamic part. In order to maintain the modularity and reduce the file size of JET templates, the developer can fragment the JET template into multiple files. The breaking of the JET templates will also help in reusing the template fragments. The breaking process is achieved by creating a new JET template, creating an object of the template, and invoking the generate method. The attributes required by the template fragment are passed as parameters of the generate method. A number of ready-made templates were created for the CMDA system.

3.5. Component Development

The CMDA system requires a set of domain and application components to build the model and to generate software artifacts of the intended system. The component development is an evolutionary process and any pre-existing source code can be wrapped with the minimum effort and used in the CMDA system. This is to ensure that least amount of startup time is spend on using a pre-existing code as a new component. While a new component can be created easily from pre-existing source code, the reusability of such component is very limited as these components have very little or no parameters. As and when the new component, created for a specific instance, needs to be reused, the JET template for the component is refactored and parameterized. After a series of such evolutions, the JET template is highly parameterized and that component becomes highly reusable.

The number and reusability of the components increase along with the increase of usage of the CMDA system. In order to start the evolution of the components, a set of seed components was developed. The set of seed components developed includes:

Expander: a highly parameterized component that will generate *ExpanderPlugin* code.

Allocator: a moderately parameterized component that will generate *AllocatorPlugin* code for most instances where no new task is published by the plugin.

AllocatorwithTaskPublish: a component with low parameterization that will generate *AllocatorPlugin* code for some instances in which a new task is published into the blackboard.

Assessor: a component with low parameterization that will generate *AssessorPlugin* code for some specific instances.

Completion: a highly parameterized component that will generate code for a generic *Plugin* that subscribes to a task and masks it as complete.

Execution: a highly parameterized component that will generate code for the *ExecutionPlugin*.

Aggregator: a highly non-parameterized component that will generate the code for *AggregatorPlugin* for very specific instances.

3.6. Development Environment

The Meta-Model is based on a few simple concepts:

- Collaboration Components are designed to work together to finish a task. They interconnect via connection points called "Roles."
- Decomposition Components can be defined in terms of other components. Each use of another component is called an "Instantiation." The inner definition itself can be a network of collaborating components. This and the collaborative concept above make the Meta-Model recursive (Figure 8).
- Reuse Components are designed to be used multiple times. However, each time its used, a component must take parameters that define its specific behavior. An Instantiation of a component contains values to fill those parameters, called "ParameterInits."
- Generation The Component definitions by themselves aren't the desired end-product. Instead, components define (or reference) definitions of how artifacts source code, UML models, documentation, deployment information, etc. are generated from a component definition. These definitions are called "Profile Mappings." Profile Mappings interpret a component instance's parameter values to determine how to generate the appropriate artifacts.
- Validation As the assemblies of models are expected to be complex, some automatic ability to verify the correctness of them is desirable.

CCollaboration omponents were designed to work together. The workflow as imported into the system is a flowgraph representing the stages and decisions of the desired application. At the top level, this represents communities of Cougaar agents collaborating. Due to the recursive nature of the Meta-Model (described below), individual agents or even individual plugins collaborations can be described using the same mechanisms. Collaboration between components is primary described using connections between Roles. A Role is simply an endpoint for a connection, declared and attached to a Component. An Instance of the component is connected via its ports to other instances, forming a graph.

CDecomposition omponents can be defined as its own graph of other component instances . This allows the CMDA system to use the same tools, Meta-Model, and user interface for society, agent, and plugin-level descriptions. Similarly, entire systems can be converted into reusable components for larger systems later.

EParameterization ach component declares what kind of parameters it needs in order validate and generate artifacts. The parameter system in CMDA is fairly complex, so a slow, thorough approach will be taken in its description.

The "Parameter" is an abstract input to a component, like a Java Interface or C++ abstract class. A component actually takes in:

1. Roles Names of other component instances that the generated instance will collaborate with.
2. JavaParameters A Java object value (as a string). Described in detail below.
3. ResourceRequirements A parameter that is only needed for deployment. These should be OCL expressions that are evaluated at deployment time to determine their final values.

TValidation he system provides many avenues for components to provide ways to validate their instances. The system doesnt allow a component to generate artifacts until it has passed all validation steps. Primarily, components specify their validation rules through constraints, top-level and in their parameters. Top level constraints are directly attached to the component, and are validated against the entire set of parameter values. This enables the verification of relationships between parameters (e.g. one parameter value must be twice the value of another, etc.). Parameter-level constraints are validated within their own scope only. They can still validate their specific values as well as those of any subparameter values.

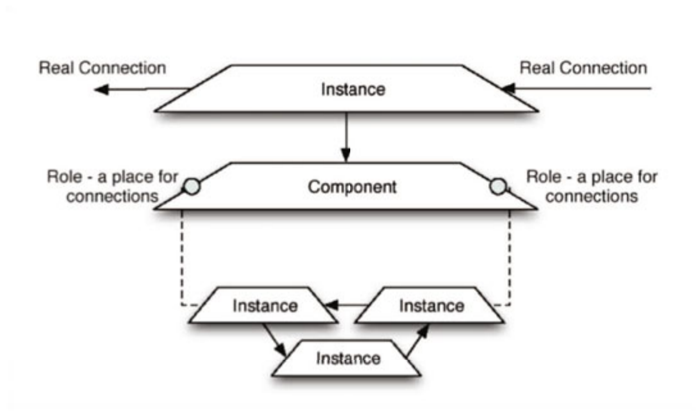

Figure 8. Component structure

OGeneration nce validated, a component instance is passed to its profile mappings (PMs). Profile mappings are compiler extensions that generate artifacts. A component specifies which profile mappings apply to it. Each PM generates output, possibly with its own internal validation system, capable of declaring errors on the input .xcomp file, viewable as normal compile errors. However, those validation failures are internal to the PM and do not cause a full-level validation failure.

4. Transformation Rules

Given the metamodel, we can now describe our transformation approach in some more detail. Our transformations have three primary stages:

1. *Tree Decomposition* — The top–level model is treated as a single component. The component serves as the root of a component tree that contains the entire system being generated. The tree structure prevents cyclic dependencies, stratifies the component graph into layers, and provides a deterministic execution path for the generation of the system.

2. *OCL Evaluation* — Parameters to each component are evaluated as OCL expressions before use (including their connections to other components). OCL supports the UML models, and serves as an expressive means to transform semantics between layers of the system.

3. *Artifact Generation* — When the final artifacts are needed, an expressive Java–based runtime is fully accessible to generators to create artifacts in any way they deem necessary. The system invokes generators with their parameters and provides them with full access to the system being generated *and* the Cougaar CMDA runtime.

4.1. Tree Decomposition

All models are structured as simple trees. Each layer jump between platforms: the platform–independent model, the platform–specific model, etc., reify as levels in the tree's hierarchy.

4.1.1. Component Layering: Transformations occur at the boundaries of the CIM, PIM, and PSM. Even within these layers, there are often many transformations. Trivial transformations are handled in a single level, while non-trivial transforms may take several levels to fully resolve. Abstractly, platform dependent and independent can be represented by two adjacent layers, but complex systems and platforms usually require many layers of transformation before the full system can be specified. If the underlying platform is hierarchal, we can directly map components to the underlying platform, leading to a straightforward transformation.

If the platform is distinctly different, we may define a single child for a component, which converts semantics between the platform independent and dependent layers. After it has done its internal transformations, it can then contain additional layers of children to continue the transformation.

Even further, if this transformation procedure isn't appropriate for a particular part of the platform, the component can define a mapping to do the rest of the work. The mapping can then execute its own procedure for further transformation and artifact generation.

The component developer may choose this route for a few reasons:

1. *Lateral Reuse* — Instead of generating each component, the component developer may want to interact with an *implementation repository* for better reuse. While our own component repository allows the reuse of CMDA components, the generators may have their own ways of reusing artifacts. Seeing one

artifact already generated anywhere else in the system, they may choose to reuse it with differing instantiation parameters.

2. *Refactoring* — Like traditional compilers, some levels of automatic refactoring may be implemented. Common subcomponents could be factored out and reused in several places. In CMDA terms, utility plugins could be dropped in for common combinations of other plugins, resulting in much smaller and faster object code.

3. *Nonhierarchal Processing* — Some platforms may simply not be amenable to hierarchal decomposition. Whatever methods are best for them can be placed in, instead of attempting to coerce it back into a tree.

Even in this case, the complexity is hidden behind a single component. It has its parameters to use as it needs, and through their evaluation, access to the entire containing component's definition — via the `self.parent` parameter provided by the system.

As such an escape from the hierarchal system is still represented as a regular component (with a different mapping), it is compatible with the regular CMDA system above and below it. Regular CMDA components can include it as needed, without any concern for its internal structure. Similarly, this non-traditional component may only choose to do part of its work nonhierarchially. It can still include other CMDA components that do use the regular execution and generation path.

4.1.2. Execution Path: The system's generation path executes in a top–down–top pattern. Top–down, the OCL parameters and constraints are evaluated, and validators run. With a valid system defined, and the parameters' final values evaluated, we find ourselves at the leaves of our system's defining tree. There, we execute our artifact generators for the leaves.

We continue bottom–up. With these leaf artifacts generated, we return an identifier for the artifact back to the parent. The parent's generator is run, with all its child artifacts' identifiers available. We proceed back up the tree this way until we have the topmost artifact generated, resulting in a complete system.

4.2. OCL Evaluation

OCL makes for a very natural parametric language for the CMDA system. First, it comes with UML. Second, it provides a good expression language for describing parameters. Third, it is effectively for building quick validation passes through the declaration of simple constraints.

We use OCL heavily in the system. While we use the tree–level decomposition to break down our layers between the PIM and PSM, OCL wires it all together. Parents specify OCL expressions to configure their children. Beyond normal numeric, string, and boolean expressions, we can supply Sets, OrderedSets, Bags, and Sequences[1].

When the CMDA system executes to generate artifacts, we run OCL expressions in three different parts of the component:

1. *Parameters* — OCL expressions that evaluate to scalar values, strings, and various collections.

2. *Roles* — References to siblings or parent–level entities in the platform. Roles connect components together.

3. *Constraints* — OCL predicates validating the resulting values of the parameters and roles.

Parameters can configure any aspect of a component. Each parameter's definition includes the method it's interpreted with. Some will be OCL expressions, some are simple strings, and others are run–time OCL expressions. In Cougaar, we can configure static attributes resulting from expressions. We can also define OCL expressions that pass on verbatim to generators, which set up the OCL for execution at run time.

Siblings connect to each other through *roles*, which specify collaborators through OCL expressions like `self.parent.other-sibling`. In Cougaar, this includes other agents in a society.

4.3. Artifact Generation

Generators are keyed to different profiles, such as source code, documentation, or formal models. Each generator connects to a component through a *mapping*, which is invoked during the CMDA compiler's execution. Generators have full access to the parameter values of the component, and are allowed to generate a single top–level artifact for that component. The nature of the hierarchy for artifacts depends on the artifact generated (e.g. a Java artifact generator may only return one class name to the parent).

They may, however generate multiple inner artifacts. For example, instead of a single Java class, an entire package may be generated, following a naming convention defined by the coordination of generators between a parent and child component.

In the next section, we discuss an abbreviated case study that outline how the transformations are implemented and used to generate Cougaar applications.

5. Summary Case Study

In order to get a more concrete sense of the CMDA approach and the prototype environment that was devel-

[1]Bags and Sequences allow duplicates, and Sequences are ordered.

oped to explore key aspects of the approach, this section provides an abbreviated case study. The feasibility of the overall approach and representation mechanism was evaluated by developing a workflow model for a sample project, namely Books OnLine 2 [1], a complete example of a Cougaar-centric application. BooksOnLine emulates functionality of an online book store and incorporates many of the infrastructure features.

Figure 9 illustrates the basic elements of an online book store where each circle is a cluster within a Cougaar society of collaborating agents. Each of these organizations will have several different subfunctions implemented using Cougaar PlugIn components.

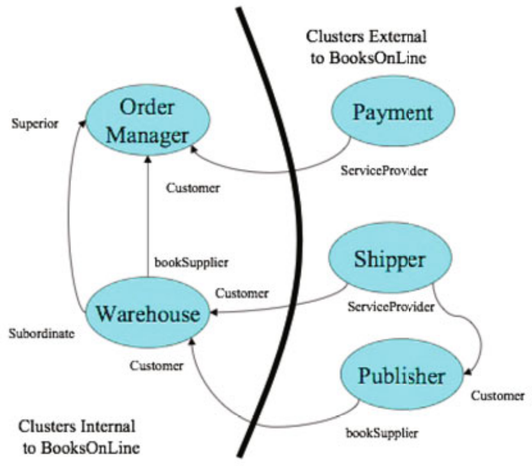

Figure 9. Books Online Example [1]

As an example, Figure 10 shows *Payment* cluster that includes functionality that will be used by a company outside of the corporate boundaries of BooksOnLine.

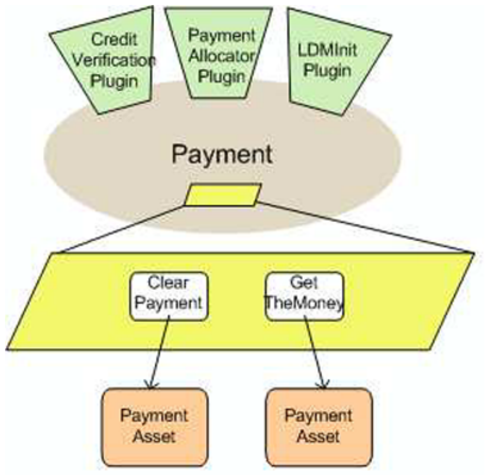

Figure 10. Payment cluster [1]

Figure 9 shows the overall structure of interaction between the agents. At the top level, the interaction is defined through the roles each agent defines for collaborators and the assignments of these roles. Looking at Figure 11, we see that we have a 3–level hierarchy of artifacts to generate.

At each level of the hierarchy, a different set of generators are defined to analyze the model and generate relevant artifacts. At the bottom–most layer we have our Expander, Execution, Allocator, Aggregator, and Completion models. Generators at this level generate directly executable code for the Cougaar API, and correspond 1-1 to each of these models. For example, we have an ExpanderGenerator that analyzes instances of Expander in models and generates Java code that implements the Expander design pattern.

The models have parametric values which the generators read and use for code generation. In Figure 11, there would be several different Java source files generated by the Expander's generator. The generated Java code is in the form of Cougaar plugins, connecting directly into its existing infrastructure as if it was hand–written. Each Java class would be similar in structure to each other, but would have specific differences that cause them to operate in line with their parameters. These Cougaar-API level PSM models are typically implemented using a fill–in–the–blank JET template system, which work similarly to the well–known JSP (Java Servlet Page) system. The majority of the generated code is written verbatim in the template, with generation logic defined as Java code within a block of escaped text.

Parameters transform through two paths. First, components define expressions for their inner components, based their own parameters. As these expressions are OCL, they can be rather sophisticated. The dependency graph for parameters is top–down. Each component defines input parameters that it uses to determine the values for parameters passed to its inner components. The generation process thusly goes top–down to evaluate all the parameters. At the bottom–most level, the values of all parameters have been fully evaluated, and artifacts can be generated. The top–most component cannot define any input parameters.

Second, the generators take the resulting parameters and use them for their own artifact generation. The OCL interpreter is available as a library both at generation time and at runtime for the generated artifacts. In the latter case, the OCL is compiled at program initialization. OCL–aware components have been written to simplify. The loop below uses the `numOfSubscriptions` parameter as a count for a loop, generating several subscriptions as needed.

At the next level up, the agents OrderManager, Warehouse, PaymentManager, and Shipper generate agent.xml files that instantiate the generated Java–based plugins. The agent.xml files use the Cougaar infrastructure for loading our generated Java code. The generator applied to all four of these agents is the same, a JET template that generates the agent.xml file.

Finally, a generator is run against the BOLSociety model, which generates an agent.xml file. Similar to the JET template for agent.xml, it simply generates a society.xml file. Both of these files are defined by two different schema in the Cougaar architecture.

As discussed earlier, the overall generation process occurs top-down. When we run against the top model, it invokes a build against all of its sub–models. The sub–models do the same, recursively. At the bottom level, the generators take their evaluated parameters and generate Java code for the Cougaar plugins. The generators return the name of their generated artifact up to their parent. The parents then generate their artifacts and return the name of their generated artifact up to their parent.

In the Books Online system, the BOLSociety's build will invoke a build of all the agents. Like all the other agents, the Warehouse would invoke generators for it's contents —the Execution, Expander, and Allocator plugins. The plugins' generators would return Java class names to the Warehouse's compiler. The class names would then be used to generate the Warehouse's agent.xml. The Warehouse's agent name would be returned to BOLSociety's build, which would then use it in it's society.xml

At each level, we use Cougaar's standard methodology for parameterization. At some levels, the child ends up being responsible for generating its fully–configured self. At others, the parent will instantiate and configure the child at run–time. When a parent configures a child, the parent stores the child's parameters in it's artifact. When a component is responsible for it's own parameters, the generated artifact uses the parameters directly. For example, some parameters to the Expander will result in specific piece of code being generated. Other parameters will result in text being generated in the agent.xml, which the Cougaar runtime will use to configure our Expander plugin at runtime.

As the generators defined at each level can be JET templates (any eclipse plugin extending our APIs can be generators), we can start each one from a hand–written artifact. The original artifact is simply renamed to indicate it's now a template (`.javajet`), and the model will specify it as a generator. The generator developer then starts defining parameters and its accompanying generation logic to generalize the template.

At the top layer, we have a simple container model that holds the components and wires them together:

```
<?xml version="1.0" encoding="ASCII"?>
<model:component xmlns:model="http://www.cougaar.org/xc/model"
doc="" level="0" name="BOLSociety">
  <member name="TheWarehouse">
    <type name="Warehouse"/>
    <param name="bookSupplier" value="OrderManager"/>
    <param name="superior" value="OrderManager"/>
  </member>
  <member name="TheOrderManager">
    <type name="OrderManager"/>
    ...
  <mapping name="Cougaar11" profile="gcam.jet.Society"/>
</model:component>
```

Using the `mapping` tag, we indicate which generator will create an artifact for this model. Multiple generators may be specified, creating mulitple models. We specify an `OrderManager` for both the `bookSupplier` and `superior` roles here. The Order Manager is instantiated below that. While our simple example only requires one of each, the same agent could have multiple instantiations, with different parameters to specify each.

Below that, we have model definitions for every type of object Here, the Warehouse would define itself as two plugins within an agent.

```
<?xml version="1.0" encoding="ASCII"?>
<model:component xmlns:model="http://www.cougaar.org/xc/model"
      name="Warehouse">
   <member name="WarehouseExecution">
      <type name="Execution" />
      <param name="subscription" value="o.isKindOf(Task) &amp;
            &amp; o.oclAsType(Task).getVerb()='PACKER'">
         <param name="variable" value="PerformJob_performJob">
            <param name="contents"
value="new PerformJob(getBlackboardService(),
                task, logging, getPlanningFactory())" />
         </param>
         <param name="CodeSnipet"
value="threadService.schedule(performJob, 1000);" />
      </param>
   </member>
   <member name="WarehouseExpander">
     <type name="Expander" />
     <param name="subscription" value="o.isKindOf(Task) &amp;
       &amp; o.oclAsType(Task).getVerb()='BOOKSFROMWAREHOUSE'">
...
```

The Warehouse is made of two plugins within the Cougaar agent: an `ExecutionPlugin` and an `ExpanderPlugin`. The first is a simple plugin that accepts Java code in some of its parameters.The `subscription` parameter provides the plugin with an OCL expression to look for on the Cougaar Blackboard. Upon finding an object of this type, it declares a variable `performJob` of type `PerformJob`[2]. The variable is filled with a new `PerformJob` instance, initialized to the current task. The current task is defined by the `ExecutionPlugin`. Once filled, the variable is used in a direct piece of java code fed in through `CodeSnipet`. `threadService` is defined as a standard member variable of the plugin.

`ExecutionPlugin` defines itself quite simply as an XML schema:

[2]Direct Java code in the parameters are a "backdoor" to allow generator implementors to get working quickly. It's usually recommended that Java code be avoided in parameters, and that the individual component type be made more general or broken up into several components to avoid this low–level programming. Here, the `variable` parameter has a convention of using an underscore to separate the variable's type and name. Inner parameters could have done the same thing separately, but this is more convenient for hand–written code that hopefully will get removed as the components get more sophisticated.

```
<?xml version="1.0" encoding="ASCII"?>
<model:component xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:model="http://www.cougaar.org/xc/model" doc="" level="0"
  name="Execution">
   <param name="subscription">
      <parameter xsi:type="model:javaParameter" name="variable">
         <parameter xsi:type="model:javaParameter" name="contents"/>
      </parameter>
      <parameter xsi:type="model:javaParameter" name="CodeSnipet"/>
   </param>
   <mapping name="ObjectName"
      profile="org.cougaar.xc.pm.demo.ExecutionPlugin"/>
</model:component>
```

A simple `subscription` containing `variable` and `CodeSnipets`,

An abbreviated look at the template will be useful:

```
<%@ jet
class="ExecutionPlugin"
...
import java.util.*;

/**
 * ExpanderPlugin with parameters <%= argblock.getInst().getParam() %>
 */

public class <%=className%> extends BOLComponentPlugin {
   private static final String pluginName = "<%=className%>";
   private ThreadService threadService = null;
   // NAME = <%= Interpreter.eval (argblock.getInst(), "self.type.name")%>
<%   ArrayList subs = argblock.paramsNamed ("subscription");
      ArrayList names = new ArrayList (subs.size ());
      ArrayList preds = new ArrayList (subs.size ());
      ArrayList exprs = new ArrayList (subs.size ());
      int i;
      for (i=0; i<subs.size (); i++) {
         String name = "subscriptionNr"+i;
         String pred = "predicateNr"+i;
         names.add(name);
         preds.add(pred); %>
      private IncrementalSubscription <%=name%>;
      <%  ParameterInit pi = (ParameterInit) subs.get (i);
         exprs.add(pi.getValue ()); %>
      private UnaryPredicate <%=pred%> = new
OCLPredicate ("<%= pi.getValue ()%>");
      <% } %>

   protected void setupSubscriptions() {
      <% for (i=0; i<subs.size (); i++) { %>
         <%=names.get(i)%> = (IncrementalSubscription)
getBlackboardService().subscribe(<%=preds.get(i)%>);
         <% } %>
      }
   ...
   /**
   * Expand as indicated.
   */
   protected void execute() {
      System.out.println ("Executing " + pluginName);
      Enumeration e;
         <% for (i=0; i<subs.size (); i++) { %>
            /*
            Check: <%=exprs.get(i)%>
            */
         e = <%=names.get(i)%>.getAddedList ();
      while (e.hasMoreElements()) {
         try {
            Task task = (Task) e.nextElement ();
         <% ParameterInit sub = (ParameterInit) subs.get(i);
         EList subE = sub.getParam();
         Iterator subI = subE.iterator();
         while(subI.hasNext()) {
            ParameterInit var = (ParameterInit) subI.next();
            if(var.getName().equals("variable")) {
               StringTokenizer st = new StringTokenizer(var.getValue(),"_");
               String varType = st.nextToken();
               String varName = st.nextToken();
         ParameterInit contents = DemoUtils.findChild(var,"contents");%>
            <%=varType%> <%=varName%> = <%=contents.getValue()%>;
            <% }else if(var.getName().equals("CodeSnipet")) {%>
            <%=var.getValue()%>
            <% }
                  }%>
         } catch (Throwable t) {
            System.out.println (pluginName+": failed on expansion of
\"<%=exprs.get(i)%>\"");
            }
            }
            <% } %>
      }
}
```

We demarcate elided parts with ellipses. In the class declaration, we define as many `IncrementalSubscriptions` and `OCLPredicates` as we have `subscription` parameters. In `setupSubscriptions`, we initialize the subscriptions with our predicates on the Blackboard. Finally, in `execute`, we go through all of our parameters given in each `subscription`, and do the variable declarations and Java code declared in each.

The workflow model was transformed into PSM and code fragments (code for Assets and Agents) were generated from the PSM. This demonstrated how components are created, assembled (to create application and implementation models) and transformed into design, code and documentation artifacts (Figure 11).

Early on in the effort, there was considerable skepticism around the ability to actually generate Cougaar applications from high level specifications. To a large degree, this has relaxed as we learned more about MDA and about available tools to support the CMDA prototyping effort. While there is still a healthy respect for the effort needed to generate all of the work products from a Cougaar Application development effort, the gap has closed considerably. We have now seen that for general instances of workflow and agents, parameterized component specification is a viable option with reasonably good results. There are still instances where complete specification is difficult, requiring human-in-the-loop effort to supply vital information, but they are not overwhelmingly hard or the norm.

Generating source code from increasingly refined and elaborated models was feasible and in most situations doable. The CMDA Meta-Model based on the recursive GDAM/GCAM structures appear to provide a reasonably good framework from which to implement the MDA approach. While the prototype was not robust enough to develop full Cougaar Applications in a development environment, it did develop them in an experimental environment. We did not examine scale or special cases of developing components; however, for the general cases that we experimented with, there were some promising results.

However, we found that roundtrip engineering requires considerable information and closure on that information to be fully feasible. However, a looser interpretation of roundtrip engineering based on mappings and dependencies coupled with todays reverse engineering tools provided significant leverage towards the objective. Given our experience, we believe that as the CMDA compiler evolves, roundtrip engineering for the larger class of components will be feasible. It will require accounting for component, connections/dependencies, and configurations that are expressed in forms like the .Xcomp file we currently employ. Further, there will need to be a more elaborate artifact management approach to support roundtrip engineering in future implementations of CMDA.

For the CMDA prototype, we took the tact of treating it like a traditional Integrated Development Environ-

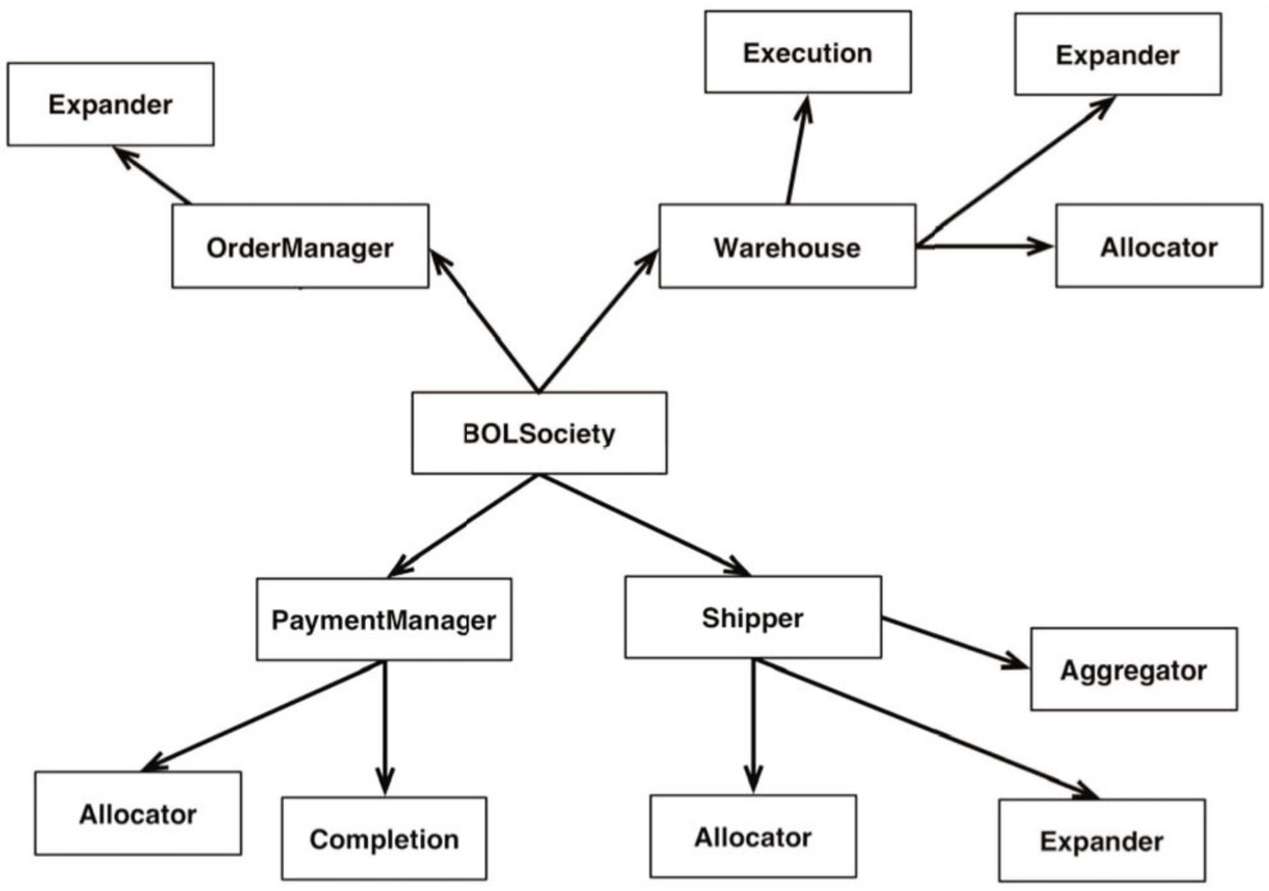

Figure 11. Books Online artifacts

ment (IDE). As it turned out, this worked well for implementation. The UML as a base language provided the input while the Design-Generate-Test cycle fell naturally with the bounds of the IDE. Using this approach, the typical compilation issues such as validation, error recovery, name lookup, and the like, show up quickly.

However, it is important to note that unlike traditional source code, different formats are relevant for different levels of abstraction in CMDA. Different diagram formats are relevant for component, agent, and society levels, such as class, component, and package metaphors. Albeit, we found in Java normal source code is perfectly fine at all levels.

Like other MDA efforts, we found that artifact management quickly becomes complex with all the different kinds of artifacts. Moreover, these artifacts were relevant at different times, much more than any nontrivial system can manage at a low level. We found that deciding which artifacts to generate at any given time are best decided through some delegation mechanism. This is the reason for the tight integration between the compiler and the component templates (the templates plug into the compiler and extend its interfaces) the dependencies could be managed in an automated manner, relieving the developer of the burden.

We found that most artifacts have their own deployment needs (i.e., relative and absolute path locations). For example, java source has to be deployed to match the declared package and class name. Naming the artifacts and preventing conflicts in naming is a complex problem to do well. To ensure uniqueness, while we could use an autogenerator to generate new names from scratch (e.g., autogener 1), we would sacrifice human comprehensibility of the generated code (as well as the other model artifacts).

Some features of an IDE like Eclipse's continuous recompilation model could easily lead to $manycopies$ of the artifacts if special care isn't taken early. Therefore, we designed our instantiation store which is structured to make sure that we only ever generate a given instance of a component once. Even this is not a perfect solution as it does cause some problems when the templates have changed: as the pre-existent instances are already there, no implicit regeneration takes place this can be fixed with a template version tag.

From a project management perspective, there were some reasonable lessons learned as well. The design and underlying architecture needs time and patience to evolve in applied research projects like this one. We should not hesitate to improve the design, even midway through the project (provided the changes do not adversely affect the project schedule). This avoids homeostasis of initial design and provides the necessary learning that is an important part of this type of effort. The willingness to transition to a more aggressive infrastructure was a good move as it resulted in a more flexible solution.

6. Related Work

There have been a number of projects that have addressed multi-agent systems (MAS) and MDA before and since our first research effort using this technology back in 2003 [8]. As we investigated more and more detailed issues, we found leverage using MDA principles [9]. We targeted a model-based engineering approach flexible enough for many model representations (ranging from abstract requirements to concrete code) to be used. While we engaged the MDA structure (i.e., CIM to PIM to PSM) to separate key development and evolution concerns, a distinguishing element of our approach is that we have not stayed strictly with the OMG route of deriving artifacts mostly in UML. Rather, when UML was the expedient route, it was used. Otherwise, we adopted the notion that if we had a close rendering of the capabilities needed in the lower levels and they could be abstracted to the higher layer without creating an interoperability dependency, then we would build the transforms and mappings directly. For example, when we developed the Expander for flexible tasking (described earlier), it was more expedient to opportunistically map the notion of task in the PIM to the agent tasking components in Cougaar (without violating the PIM and PSM separation). This allowed quick and verifiable transformation rules.

Some non-MDA-based MAS methodologies such as Prometheus [19], Tropos [10] and MaSE [12] have proposed the mapping of the design models into implementation code and have provided some tools for supporting both the design and the implementation of MAS. However, it is possible to describe platform specific details during the design of the application — violating the separation of concerns between PIM and PSM. The resulting high-level design models can be platform dependent and, consequently, are not easily portable to any other platform.

As with many MDA-oriented MAS efforts, we chose to use a combination XML and other representations like OCL because they lend themselves to the support of transforms and mappings. As described in [5], the key to MDA lies in the modeling representations and the transforms/mappings. XML lends support for major data transformations while OCL provides the constraints necessary to characterize mappings to relevant components and parameters for configurations. CMDA follows a similar form, but is less UML specific in its representations. Where there exists a line from abstract models to concrete components (source code is a model), CMDA allows the incorporation of the models and relevant transforms/mappings.

Other research has used the MDA approach to define a MAS development process. [22] demonstrated the use of MDA to derive MAS low-level models from MAS high-level models. The authors propose to use the Tropos methodology and the Malaca models in the MDA approach. The high-level models created while using the Tropos methodology are transformed into low-level Malaca models. However, the transformation from the Tropos models into Malaca models is not completely automated. Since human in the loop is not explicitly designed into the approach, there are some discontinuities in the flow when this occurs. Moreover, such an approach does not deal with the transformation from Malaca models into code. In [16], the authors proposed a domain-dependent methodology based on a model-driven approach for the development of distributed mobile agent systems. They define a mobile agent conceptual model for distributed environments and describe a set of components, represented by a collection of intelligent mobile agents.

Koehler et al. outline their transformation method that implements model-driven transformations between PIM business view elements and PSM architectural models [18]. This approach, while more sophisticated for the boundary between PIM and PSM, maps well onto the CMDA approach. In CMDA, we attempted to stay as simple as possible, but complex enough to handle the complex tasking that could arise with collaborative agents in Cougaar. The CMDA metamodel was derived from our experience and provided a reasonable structure for the relevant transformations and mappings.

More recently, Demir compared the Software Factory approach espoused by Microsoft with the Model-Driven approach [11]. While the example with the online bookstore (standard example for MASs), lines up nicely with CMDA, the paper is theoretical and is not supported by an actual prototype or empirical results. CMDA does provide a complete system that demonstrates what Demir discusses in his paper with the exception of the comparison with software factories.

While all of these related works have significant contributions, what distinguishes CMDA is its flexible metamodel that allows for more than UML-based models in a domain-independent manner. The architecture of the

CMDA prototype implementation provides for the natural progression from an unpopulated model repository (with considerable human-in-the-loop) to automated generation of agent-based software. The demonstrated support for scale and sophistication on a couple of MAS standard examples (Books Online and Pizza Delivery workflow) indicates the robustness of CMDA and the transform/assembly approach to generating the Cougaar-based systems.

7. Conclusions

As software systems accommodate more sophisticated tasking and increasingly adapt to an ever-changing environment, agents are a likely choice to respond. While agent systems support many aspects that today's systems demand, in cases where they have been employed, development and evolution have been challenging. To address the challenge, we investigated MDA as a model-based means of moving the abstraction level up so that application domain personnel could be increasingly incorporated into the development teams. This has been shown in other development approaches to increase productivity and decrease errors.

We designed the GDAM to support both computationally independent and platform independent models. They represent the interaction points for the domain specialists with the software developer. We also developed the GCAM to support the platform specific elements of the development. The GDAM and GCAM perspectives were tied together through the metamodel that enables the transformations and mappings necessary to generate the application. We endeavor to automatically generate as much of the application as possible.

While early investigations fostered transitions using UML for both analysis and design models, we found that in many cases, the domain objects could be mapped to Cougaar components, particularly where the abstract concepts were already developed in the Cougaar architecture. This was due to the fact that many Cougaar components were already developed for workflow applications. This was welcomed, since sometimes the full cycle representations in UML were laborious and potentially contrived to accommodate the limitations of UML. In these cases, we generated the UML from the component definitions to support the need for documentation.

To engage the interdisciplinary team, we developed a graphical editor that allows the domain experts and the software developers to work together in producing GDAM models. This enabled the subject matter experts to be involved with the development; thus improving their contributions and expediting what is often seen as a communication bottleneck in the development process. The coupling of the reuse from the MDA and the interdisciplinary interaction are the primary improvements observed in the initial phases of this research.

While the approach currently focuses on the Cougaar architecture specifically, it can be applied readily to other agent-based architectures. It will, with further research and experimentation, form the basis of a methodology appropriate for application to a wider range of agent architectures. The ongoing research is focusing on the evolutionary component development. The current set of seed components is being refined and new components are being developed.

Overall, the research demonstrated that the model-based engineering approach is relevant to sophisticated application development. While a pure MDA approach would be feasible, the extra effort may rob some of the benefits of the overall approach. Changing the system in the MDA approach was effective when it could be made at the most abstract level; however, when the change is made to the code, the implementation was problematic as the roundtrip engineering was a good ideal, but less feasible than expected.

Our CMDA approach exhibts desirable benefits in terms of complexity, scale, and ease of application. CMDA deals effectively with complex situations since it is developed for supporting plugins (behaviors and tasks), and the As to scale, Cougaar itself handles large systems, but CMDA does not inhibit this. With the use of abstractions both in Cougaar and UML models, this is readily handled. Finally, the interface for the user in our system is relatively easy with the Eclipse IDE coupled with the GCME modeling environment for the subject matter experts. To further gain experience with CMDA, we would like to apply it to production systems and observe how the development teams perform. The agent-based system is a good example that can be compared with web services and with service oriented architectures. Future research will examine how model-based approaches could address these situations.

Acknowledgments

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, Tim Tschampel, H. Lally Singh, and Boby George.

References

[1] —. *Book-On-Line: An Advanced Cougaar Tutorial Version 2.0.* Cougaar Software, Inc., July 2003.

[2] —. *Cougaar Architecture Document: Version for Cougaar 11.4.* BBN Technologies, 23 Dec. 2004.

[3] —. *Cougaar Developers' Guide: Version for Cougaar 11.4.* BBN Technologies, 23 Dec. 2004.

[4] —. Workflow Management Coalition workflow standard: Process definition interface — XML process definition language (version 2.00). Technical Report WFMC-TC-1025, Workflow Management Coalition, Lighthouse Point, FL 33064, Oct. 3 2005.

[5] J. Alvez de Maria, V. Torres da Silva, and C. J. Pereira de Lucena. *An MDA-Based Approach for Developing Multi-Agent Systems. Monografias em Cincia da Computao, No. 31/05, Editor: Prof. Carlos Jos Pereira de Lucena*, Sept. 2005.

[6] J. Arlow and I. Neustadt. *Enterprise Patterns and MDA: Building Better Software with Archetype Patterns and UML.* Addison-Wesley, Boston, 2004.

[7] C. Atkinson and T. Kühne. Model-driven development: a metamodeling foundation. *IEEE Software*, 20(5):36–41, Sept.-Oct. 2003.

[8] S. Bohner, B. George, D. Gračanin, and M. G. Hinchey. Formalism challenges of the Cougaar MDA. In M. G. Hinchey, J. Rash, W. Truszkowski, and C. Rouff, editors, *Proceedings of the Third NASA/IEEE Workshop on Formal Approaches to Agent-Based Systems (FAABS III)*, volume 3228 of *Lecture Notes in Computer Science*, pages 57–71. Springer Verlag, 26–28 Apr. 2004.

[9] S. Bohner, R. Ravichandar, and J. D. Arthur. Model-Based Engineering for Change Tolerant Systems. *Journal on Innovations in Systems and Software Engineering, 3(4)*, December 2007.

[10] P. Bresciani. Tropos: An Agent-Oriented Software Development Methodology. *Int. Journal of Autonomous Agents and Multi-Agents Systems*, 8(3):203–236, 2004.

[11] A. Demir. Comparison of Model-Driven Architecture and Software Factories in the Context of Model-Driven Development. *Proceedings of the Fourth Workshop on Model-Based Development of Computer-Based Systems and Third International Workshop on Model-Based Methodologies for Pervasive and Embedded Software (MBD/MOMPES06)*, 2006.

[12] S. DeLoach. Multiagent Systems Engineering: a Methodology and Language for Designing Agent Systems. *Proceedings of Agent Oriented Information Systems*, Washington, 1999.

[13] E. Gamma and K. Beck. *Contributing to Eclipse: Principles, Patterns, and Plug-Ins*. The Eclipse Series. Addison-Wesley, Boston, 2004.

[14] D. Gračanin, S. A. Bohner, and M. Hinchey. Towards a model-driven architecture for autonomic systems. In *Proceedings of the 11th Annual IEEE International Conference and Workshop on the Engineering of Computer Based Systems (ECBS 2004)*, Brno, Czech Republic, May 24–27 2004.

[15] D. Gračanin, L. H. Singh, S. A. Bohner, and M. G. Hinchey. Model-driven architecture for agent based systems. In M. G. Hinchey, J. Rash, W. Truszkowski, and C. Rouff, editors, *Proceedings of the Third NASA Workshop on Formal Approaches to Agent-Based Systems (FAABS III)*, volume 3228 of *Lecture Notes in Computer Science*, Greenbelt, Maryland, 26–28 Apr. 2004. Springer Verlag.

[16] M. Kazakov, H. Abdulrab, and G. Debarbouille. A Model Driven Approach for Design of Mobile Agent Systems for Concurrent Engineering. MAD4CE Project Technical Report 01-002, Université et INSA de Rouen, 2002.

[17] A. Kleppe, J. Warmer, and W. Bast. *MDA Explained: The Model Driven Architecture: Practice and Promise*. Addison-Wesley, Boston, 2003.

[18] J. Koehler, S. Kapoor, F. Wu, and S. Kumaran. A Model Driven Transformation Method. *Proceedings of the Seventh IEEE International Enterprise Distributed Object Computing Conference (EDOC03)*, 2003.

[19] L. Padgham and M. Winikoff. Prometheus: A Methodology for Developing Intelligent Agents. *Proceedings of the First International Joint Conference on Autonomous Agents and Multi-Agent Systems*, Italy, 2002.

[20] T. Parr. *The Definitive ANTLR Reference: Building Domain-Specific Languages*. The Pragmatic Programmers, Raleigh, North Carolina, 2007.

[21] J. Rumbaugh, I. Jacobson, and G. Booch. *The Unified Modeling Language Reference Manual*. The Addison-Wesley Object Technology Series. Addison-Wesley, Boston, second edition, 2005.

[22] A. Vallecillo, M. Amor, and L. Fuentes. Bridging the Gap Between Agent-Oriented Design and Implementation Using MDA. *Proceedings of the Autonomous Agents and Multi-Agent Systems Workshop*, 93–108, 2004.

[23] J. Warmer and A. Kleppe. *Object Constraint Language, The: Getting Your Models Ready for MDA*. The Addison-Wesley Object Technology Series. Addison Wesley Professional, second edition, 2004.

[24] T. Weis, A. Ulbrich, and K. Geihs. Model metamorphosis. *IEEE Software*, 20(5):46–51, Sept./Dec. 2003.