58

Formalism Challenges of the Cougaar Model Driven Architecture

Shawn A. Bohner[1], Boby George[1], Denis Gračanin[1], and Michael G. Hinchey[2]

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

Abstract. The Cognitive Agent Architecture (Cougaar) is one of the most sophisticated distributed agent architectures developed today. As part of its research and evolution, Cougaar is being studied for application to large, logistics-based applications for the Department of Defense (DoD). Anticipating future complex applications of Cougaar, we are investigating the Model Driven Architecture (MDA) approach to understand how effective it would be for increasing productivity in Cougar-based development efforts. Recognizing the sophistication of the Cougaar development environment and the limitations of transformation technologies for agents, we have systematically developed an approach that combines component assembly in the large and transformation in the small. This paper describes some of the key elements that went into the Cougaar Model Driven Architecture approach and the characteristics that drove the approach.

1 Introduction

Software development 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 errors as software engineers work together to develop the requisite components, assemble them, and verify that they meet specifications. Model Driven Architecture (MDA), also known as Model Driven Development, represents an emerging approach for organizing this evolution and its resulting artifacts. Through a successive series of computational independent, platform independent and platform specific model transformations, MDA facilities generation of software systems.

With the relentless advancement of technology, complexity and integration issues often dominate modern computing. To respond to the sheer volume of software and consequential complexity, the software community has increasingly embraced architecture principles. Software architecture provides a framework to understand dependencies that exist between the various components, connections, and configurations reflected in the requirements. Some situations lend themselves to what is called an agent-based architecture.

As software grows in complexity and autonomy, manifold dependencies between critical elements of software increasingly drive many software architectures. Agent-based software systems address this complexity particularly where components may have autonomous properties (i.e., complex information and task-intensive situations) and require mechanisms to control these and other properties in a predictable way. The task orientation coupled with intelligent agents provides a strategic and holistic environment for designing large and complex computer-based systems.

This research concentrates on understanding and applying the MDA approach in an Agent-Based Architecture — specifically, Cougaar. The goal is to explore ways to use MDA to facilitate domain and software engineering staff developing Cougaar Applications, to move up and program at the higher level, the domain level. We investigate how to compose Cougaar components into a General Cougaar Application Model (GCAM) and develop a General Domain Application Model (GDAM) for specifying and generating software applications. While the scope of this research focuses on the establishment of the GCAM and GDAM, it also provides example recipes for transforming the models into relevant software artifacts such as requirements, design, code, and test documents.

1.1 Agent Based Systems

While there are several definitions for software agents [1],we simply define an agent as a software entity that perceives its environment and responds through action(s) or tasks to fulfill a designed purpose. This broad definition covers a wide range of software agents, where agent types are characterized by properties, such as autonomous, interactive, adaptive, sociable, cooperative, competitive, proactive, intelligent, and mobile. By combining these properties in different ways, researchers have defined different agent types and, depending on the criteria, organized these agent types into taxonomies.

An "agent system is a platform on which agents are deployed"[2]. Software agent systems, also known as frameworks, need not be large systems, requiring enterprise-class machines to execute. Some agent systems are characterized by a large footprint and require considerable resources to execute. Others are lightweight and can execute in an embedded architecture.

A general agent platform architecture consists of three major components: a platform manager, an advertisement registry and a set of agents. Key characteristics of this general agent platform are that (a) there is some mechanism by which agents are managed (i.e., created, deleted, suspended, resumed, etc.), registered and also discovered by other agents; and (b) there is a communication mechanism. The platform manager is responsible for managing the agents, handling operations such as the creation, deletion, suspension and resumption of agents. The advertisement registry contains descriptions of the agents in the system and facilitates discovery of those agents. Implied in this architecture is that agents can communicate with each other, with the platform manager and with the advertisement registry.

Some interesting examples software agent systems include Grasshopper [3], JACK [4], Cougaar [5], and JADE [6]. There are also several more agent systems that are compliant with Foundation for Intelligent Physical Agents (FIPA) specifications [7].

1.2 Cougaar

The Cognitive Agent Architecture (Cougaar) is an open source, distributed agent architecture [8] resulting from over eight years of research and development, and over $150 million investment by the Defense Advanced Research Projects Agency (DARPA) under the Advanced Logistics Program (ALP) and the Ultra*Log program [9]. Cougaar is a Java-based architecture for the construction of large-scale distributed agent-based applications characterized by hierarchical task decompositions. ALP demonstrated the feasibility of using advanced agent-based technology to carryout rapid, large scale, distributed logistics planning. Ultra*Log is developing information technologies to enhance the survivability of these distributed agent-based systems operating in extremely chaotic environments. Over the last four years, fault tolerance, scalability and security have become the focus of evolving this platform for more robust applications.

The Cougaar environment enables developers to build intelligent applications that can recognize and accept high level tasking, determine suitable processes and activities, and allocate appropriate resources to complete the tasking. From an information systems workflow perspective, Cougaar agents can accomplish various tasks based on the functional business processes with which they are configured.

Cougaar agents are organized into a society that collectively solve(s) problem(s). A society can encompass one or more communities of agents that share functional purpose or organizational commonality. A Cougaar node refers to a single Java Virtual Machine (JVM) running on a single server that contains one or more agents. A society may be deployed across several nodes. Agents on the same node may compete for resources including CPU, the memory pool, disk space, and network bandwidth.

Figure 1 illustrates the Cougaar agent structure consisting primarily of a blackboard and, a set of plugins and logic providers that are referentially uncoupled. The blackboard is a container of objects that adheres to publish/subscribe semantics. Plug-ins provide business logic. Logic providers translate both incoming and outgoing messages. When an agent receives a message, it publishes it to the blackboard where a logic provider observes its addition and transforms it into an object that plugins work on. Plugins publish/remove objects, or publish changes to existing objects. Plugins create subscriptions to get notified when objects of its interest are added, removed or changed in the blackboard.

Agents collaborate with other agents, however they do not send messages directly to each other. Instead, a task is created. Each task creates an "information channel" flowing through the society, for requirements passing down and

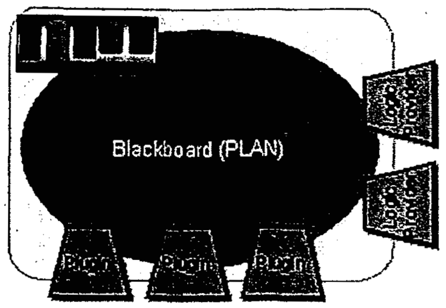

Fig. 1. Cougaar Agent Structure [8]

responses going back. In order to send an object or resource, to another agent, the developer must first associate the object or resource with the task. Cougaar uses the concept of asset to represent objects or resources used by task. Only instances of the Asset class can be associated with the task (i.e., all multi-agent objects must be defined as assets).

Once the task is created, then the task to be allocated must be located. This is typically accomplished 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 and the specified agents blackboard.

A relationship between two agents which can either be a superior/subordinate or customer/provider relationship. The superior/subordinate relationship supports long-standing transactions where a superior gives high-level tasks to the subordinate, which then performs the task and then report aggregate and trend information back to the superior on a periodic basis. Cougaar support dynamic re-planning and execution monitoring, based on these aggregate/trend information. A customer/provider relationship on the other hand is for task-order services between agents on a peer-to-peer basic and may result in large scale discrete data flows between the agents.

1.3 Model Driven Architecture

In some sense, MDA is a natural progression from previous advances in computer science. Using models in the development of a system has been practiced for decades, and even for centuries in other engineering disciplines (e.g., Building Architecture). Perhaps the most telling transition in mindset is how modeling in MDA takes a model (typically an abstraction of a reality) and creates an executable form through a series of predictable transformations. Since the computer uses a conceptual medium developed by a software engineer (i.e., a model or series of models), transforms now make abstractions of the real world accessible and even executable on a computer. In this sense, models are no longer simply an aid in understanding — the model can now become something much more concrete.

Like other engineering disciplines, software architecture helps us deal with the inherent complexities of building today's software systems. Systematically, separating concerns, formalizing the interfaces through standards and the like, provides better leverage for developing and evolving the software we employ. Software architecture — the structure or structures of the system, which encompass software components, their externally visible properties, and the relationships among them[10] — addresses the aforementioned growing complexity by providing structure for thinking about and communicating key relationships between components, whether they are commercial-off-the-shelf software (COTS), middleware, or custom developed.

MDA endeavors to achieve high portability, interoperability, and reusability through architectural separation of concerns. In some respects, MDA is an advanced perspective on well-known essential systems development concepts practiced over the years (albeit frequently practiced poorly). MDA hinges on the long-established concept of separating the operational specification of a system 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 the platform implementation.

Starting with an often abstract computation independent model (CIM) such as a business process workflow or functional description, the platform independent model (PIM) is derived through elaborations and mappings between the original concepts and the PIM renderings. Once the PIM is sufficiently refined and stable, further platform specific models (PSM) are derived through a series of elaborations and refinements into a form that can be transformed into a completed operational system.

The CIM layer is where vernacular specific to the problem domain is defined, where constraints are placed on the solution, and where specific requirements reside. 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 budgetary constraints, deployment constraints, and performance constraints as well as functional constraints.

The PIM provides the architecture, the execution 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 perspective. 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 perspective. Structural and semantic information on the types of components and their interactions (e.g., design classes, interaction 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 the MDA approach. The mappings from platform independent 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 is the opportunity to have layers within the PSM to produce intermediate-transformations on the way to the executable system. These models can range from detailed behavior models to physical source code used in the construction of the system.

Direct PIM to PSM mappings are only possible in relatively simple situations today. Today's modeling languages are not sufficient to express all possible processing mechanisms. While UML 2.0 is attempting to address this limitation, it's too early to measure its impact. Therefore, in this research effort, we have attempted to glean the the benefits of the MDA approach while avoiding, to the extent possible, its inherent limitations.

The MDA approach specifies a system independently of the platform that supports it, specifies the platform(s), chooses platforms for the system, and transforms the system specification into those for particular platforms. While this approach is still evolving, we are encouraged by its progress and skeptical of some claims made by proponents. Therefore, we have adopted an approach that incorporates the more stable concepts supported by tool technology and delayed others that are still in question as far as implementation potential in the next year.

2 Cougaar Model Driven Architecture

The objective of this research project is to improve the productivity of Cougaar system developers by applying Object Management Groups MDA approach. The productivity enhancement is achieved by automatic generation of partial sets of software artifacts such as requirements, design, code and test cases. While technologically, this has not been accomplished before, the Cougaar Model Driven Architecture (CMDA) Project endeavors to inspire solutions toward fully automated generation of software artifacts.

The CMDA system simplifies Cougaar-based application development by providing two important abstraction layers namely Generic Domain Application Model (GDAM) and Generic Cougaar Application Model (GCAM). The GDAM represents the PIM and encompasses the representation of generic agent and domain specific components found in the domain workflow. The GCAM layer, upon which the GDAM layer is built, reflects the PSM or Cougaar architecture, its specifications and environment. The user specifies the workflow of the intended Cougaar system using workflow components and the system is then detailed using GDAM and GCAM models.

CMDA approach uses a combination assembly and transform approaches to assemble components specified in GDAM and GCAM models and then transform them into intended Cougaar-based systems. The GDAM and GCAM engines assemble the respective models and the transform engine parses through the assembled set of models to produce the actual software artifacts such as requirements, design, code and test cases.

Figure 2 depicts how all the pieces fit together conceptually. The CIM is realized through the GDAM/PIM, which is realized through the GCAM/PSM. While this is not a fully implemented MDA approach in every detail, it does conceptually reflect the key principles.

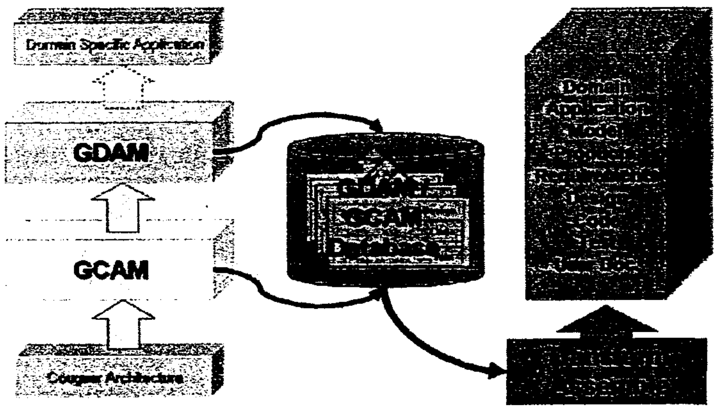

Fig. 2. Basic CMDA Approach

To a large extend, the CMDA systems capabilities are dependent on the effectiveness and efficiency of the transformation process. The transformer generates the system requirements by parsing mostly components present in the workflow layer, as the systems flow of execution and related constraints are described at that layer. While generating the requirements, the transformer also examines the components in the GDAM layer. Such examination is warranted due to the influence or tailoring some GDAM components have on the requirements that are being generated. Further it should be noted that the requirements, which are generated automatically, are partial in nature. The low-level design of the intended Cougaar system is to be elicited from the assembled GCAM components. The low-level design encompasses the GCAM model of the system, which includes (but is not limited to) UML class diagrams, sequence diagrams, state transition diagrams and deployment diagrams. The code and test cases are generated and/or assembled from the GCAM model, whose model representation will be in a suitable representation that provides the required completeness and correctness.

2.1 Formal Method Approach Selection

Cougaar is a highly complex system that implements concepts such as "time phased locality of reference" and "managed inconsistency." Hence, testing and finding errors using traditional testing methods such as testing for all possible states or artificially reducing the states by discerning selection, may be grossly inadequate. In such complex systems, formal methods are the chosen methods to assure correct operation [11]. Formal methods, whose underlying basis is mathematical notations and techniques, offer capabilities to fully specify the system using mathematical models. The completeness and correctness of the system is verified by validating the equivalent mathematical model of the system. However for most applications, due to time constraints, it is not advisable or even economically feasible to apply formal methods to fully specify the entire system. Frequently in real-world projects, formal methods are applied to a small subset of components that have the necessity for formal treatment [11].

The transformation processes for the CMDA system encompass significant challenges. While researchers have conducted transformations before, we are yet to come across any example that has attempted to perform transformations to this scale or depth. While other parts of the system such as mapping between GCAM and GDAM components are significantly difficult, the transformation is beset with some interesting challenges. The transformation challenges include:

1. Difficulties arising due to correctness and completeness errors in the input model,
2. Need for accurate depiction of the complex input model in the generated software artifacts (verifiability), and
3. Need to provide consistent output when repeated transformations (with same input) are performed.

These are particularly important for the portions of the CMDA system where equivalence and rewrite rules are applied. The degree to which these challenges are not met are proportional to the degree to which "human in the loop" will be necessitated. A major decision taken while deciding on the transformation approach was on adopting the assembly approach or synthesis approach. Given the complexities involved, it was decided to follow a combined assembly/transformation approach - thereby leveraging the simplicity of assembly approach and the efficiency of transformation approach. Further, the existence of many-to-many or at the least many-to-one mapping between components in two different levels makes a purely synthesis approach very difficult and highly error prone. In particular, the many-to-many mapping relationships between GDAM and GCAM components could result in a complex and unwieldy system, if synthesis approach or fully automated software artifact generation technique is used.

The following were identified as the key transformation requirements for the CMDA system.

1. Assembling the systems intended external behavior, specified using the workflow and GDAM semantics, into English requirement statements,
2. Assembling the system design represented using GDAM and GCAM components into system design in UML representation,
3. Generating code and test cases from the GCAM model by means of assembly approach,
4. Verification and validation of code generated.

3 Formal Methods

Formal methods, a combination of specification language and formal reasoning, can be classified into three categories: (1) Mainstream formal methods, (2) Theorem provers and (3) Customized formal methods. A brief description of the three categories is given in this section to give a flavor of the decision space available for the CMDA system.

Mainstream formal methods use rigorous mathematical models to specify the system. The foundations for mainstream models are usually based on set theory and first order predicate calculus. Examples of mainstream formal methods capabilities include Z, B, CSP, VDM, RAISE.

Theorem provers use rigorous mathematical proofs to describe software systems. Examples theorem provers include Nqthm, PVS, OBJ, and Isabelle. While theorem provers can be very effective, they may suffer poor usability unintuitive development environments and graphical user interfaces. Further, development of systems using theorem provers can be difficult.

Custom formal methods are essentially extensions and adaptations of mainstream formal methods and theorem provers. Examples of these include VDM++, Temporal PetriNets, and Timed CSP. Formal methods are extended to support specific development paradigm such as object-oriented systems. Hybrid formal methods, a type of custom formal methods, are formed by combining two or more different types of formal methods.

3.1 Formal Methods in Transformation

The capabilities of the formal methods were understood by conducting an in-depth survey on some of the important formal methods that were used for specifying agent-based systems. Table 1 depicts the representative formal methods surveyed based on their Object-Oriented (OO) modeling support, usability, tool support and concurrency support. The rows of the table lists the different formal methods that were surveyed ranked in the increasing order of preference for the CMDA system. The columns of the table indicate the comparison criteria with decreasing order of importance (as far as CMDA system is concerned) as one move from left to right. The criteria were selected keeping in mind the transformation requirements, which necessitate representation notations that have adequate support for representing components and their constraints, scalability to represent large and complex systems and tool support for the assembly approach.

The support for representing objects is the most important selection criterion as Cougaar is an object-oriented system. The OO support criterion includes ability to represent objects and their constraints such as pre-conditions and post-conditions. The tool support is another important criterion for selection since CMDA is to be interfaced with eclipse IDE platform. The tool support should include GUI interfaces to perform consistency checks, type checking and code generation. The usability criterion gives an indication on the amount of difficulty in learning and using the formal method, with a good rate indicating that the methods syntax are similar to popular programming languages and easy to learn. The scalability criterion is the fourth important criterion that indicates whether the representation is scalable enough to support complex Cougaar systems. Formal basis criterion, the least important one, provides insights into the richness of the formal methods to describe the system completely and correctly.

Table 1. Comparison of Candidate Formal Methods [12]

Name	OO Support	Tool Support	Usability	Scalable	Concurrency	Formal Basis
X-machines	Yes	Very Poor	Poor	No	No	Yes (Formal Lang)
WSÇCS	Yes	Poor	Poor	Limited	Yes	Yes (Process Algor.)
B	Yes	Average	Good	Yes	No	Yes (Set theory)
Z variants	Yes	Average to Good	Average	Yes	No	Yes (Set t./Pred. C.)
CSP	Yes	Good	Average	Yes	Yes	Yes (Algebraic)
Petri Nets	Yes	Average	Good	No	Yes	Yes
VDM++	Yes	Good	Good	Yes	Yes	Yes (Set theory)
UML	Yes	Good	Good	Yes	Yes	No

As indicated in the Table 1, among formal methods, VDM++ appears to possess all of the important characteristics required by CMDA system. Some of the other prospective formal methods include CSP and Petri Nets. While CSP does support OO representations and has good tool support, the usability of CSP method is only average. As for Petri Nets, scalability of Petri net models is a major issue. Although VDM++ satisfies the criteria requirements of CMDA system, the time constraints imposed by the project schedule might not permit complete formalization of Cougaar system. Hence, the most apt implementation approach for CMDA system might be to combine the UML and VDM++ methods to exploit the advantages of both methods.

3.2 Vienna Development Method (VDM)

The Vienna Development Method [13] is a notation and set of techniques for formally specifying object-oriented systems (with concurrent and real-time behavior) including modeling the systems, analyzing those models and progressing to detailed design and coding. VDM has its origins in the work of the IBM Vienna Laboratory in the mid-1970s. VDM, one of the most popular and frequently used formal methods, is also one of the few that has ISO Standards for its specification language - VDM-SL, Meta-IV [14]. VDM++ is an extension of the VDM which support object oriented modeling. In this subsection, we outline VDM++ details on performance against the criteria for selection.

Advantages

The advantages of using VDM++ for this project include:

Usability: One key hindrance in using formal methods is the lack of support for programming language like semantics. VDM++ provides a programming language like semantics, thereby enhancing the usability of the method among developers. Further, VDM++ can be used in varying depths from specifying the requirements more correctly and completely, and to develop models for analysis and for implementing the system.

Applicability: Unlike most formal methods that evolved from academic world, VDM method was developed by the industry for solving real world problems. Hence VDM and its extension, VDM++, are used extensively and successfully to solve industrial problems.

OO Modeling Support: VDM++ is designed with OO modeling in mind. Hence the language can be used to model object oriented system, like Cougaar, without any modifications. The language also supports multiple inheritance and provides mechanisms to specify constraints on data and operations. The support for OO modeling is one of the biggest advantages for using VDM++.

Tool Support: VDM has extensive tool support. The class of tools available for VDM includes (1) VDM through Picture (VtP) by IDE: to input/edit formal specifications, to specify requirements using pictures or graphics (2) SpecBox: to print formal specifications captured automatically, to check specifications for grammatical correctness and for specifications completeness (3) Delft VDM SL: to check specifications for grammatical correctness and for specifications completeness (4) mural for proof support, (5) VDM domain Compiler for automated code generation and (6) transformation tools for converting UML models to VDM and vice versa [15, 16]. Further, IFAD VDM++ Toolbox is a set of tools designed to support VDM++. The toolbox provides a number of features that include checker to validate syntax and type, test coverage and statistics tool, and C++, Java code generators. Further, the toolbox provides APIs that allow programs to access and modify the running instance of VDM++ models inside the toolbox. This helps easier interfacing with the Eclipse IDE.

Disadvantages

Mathematical Foundation: VDM++ is based on mathematical notations. Therefore, many domain experts and system developers may not like to encode system specifications using VDM++ language semantics. The disadvantage can be mitigated by developing wrappers that will hide the complexity of VDM++ semantics.

Time Constraints: Even for formal methods experts, large system development with VDM++ would be a lengthy endeavor. The modeling of GCAM components in VDM++ will be time consuming and difficult. Hence, modeling the entire Cougaar system using VDM++ has to be avoided.

3.3 VDM++ Toolbox and CMDA

The VDM++ Toolbox, developed by IFAD, is a set of tools that supports the object-oriented VDM++ extension of VDM-SL. The toolbox, which is part of the VDMTools, differs from most other CASE tools for formal methods in the way the functional aspects of a specification are analyzed. Some of the features of the VDMTools are Specification Manager, Pretty Printer, Syntax Checker, Test

Coverage and Statistics Tool, Type Checker, Dependency Browser, Interpreter and Debugger, Dynamic Link Facility, Couplings to Third-party Tools, and Java Code Generator

The features of the VDM Tools planned to be used for the CMDA system include Rose-VDM++ link to convert UML into VDM++, VDM++ to Java code generator, Syntax and Type checker and Test Coverage and Statistics Tool.

In the next section, we discuss the use of VDM++ and UML in the CMDA approach emphasizing the transformation implications.

4 CMDA Transformation Approach

The transformation challenges detailed above entails using multiple representations to represent the CDMA system components. The representation that we believe, best addresses the challenges is a combination of UML and VDM++. The CMDA project intends to build a developer environment that will offer developers components, which can be aggregated to represent the system in workflow, GDAM and GCAM levels. Each of the components named as Workflow Beans, GDAM Beans and Cougaar Beans respectively (in synonym with Java beans concept) will contain sections of software artifacts and related information pertaining to that bean. Some example sections of the software artifacts that beans contain include:

1. Requirements model from which the transformer gleans the partial set of requirements,
2. Design model from which the systems 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 references are traversed by the transformer while assembling the systems code 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 description about the bean, and constraints pertaining to data, operation and connections with other beans. The 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. For example, a GDAM beans requirement section will be larger than the requirement section of the Cougaar bean, while the code section of a GDAM bean might be pointer to the Cougaar beans or code that can implement the GDAM bean in Cougaar. The models in the design model section of each bean will be represented using UML while the VDM++ representation will be used to delineating connector and other constraint information. The code section will contain links to Java code libraries

70

at GCAM level and pointers to lower levels in rest of the abstraction layers. The requirements might be a combination of XPDL, text and UML diagram while the constraints also contain mapping (or connection) information that are mostly rule based with some formalizations applied.

The workflow of the CMDA system starts with developer assembling the system by picking the right workflow bean components and connecting them to represent the workflow. The constraints pertaining to connection are encoded in the beans and developers are shown a detailed error message when they try to connect two dissimilar components. Once the workflow of the system is build, it could be verified for consistency. The developer is then shown a list of GDAM beans that can be chosen to map a particular workflow bean. The expert system will list only related GDAM beans based on the constraints specified by the developer at the workflow level. The rationale to allow developers chose the right component is to allow developers make design decisions with the system assisting them (by showing a list of possible solutions and patterns).

The GDAM beans are mapped into Cougaar beans in a similar fashion. In all layers, as and when required, the developer will input necessary information to satisfy the completeness and correctness of bean component. The usability of the system can be improved by developing wrappers that would mask the semantics 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.

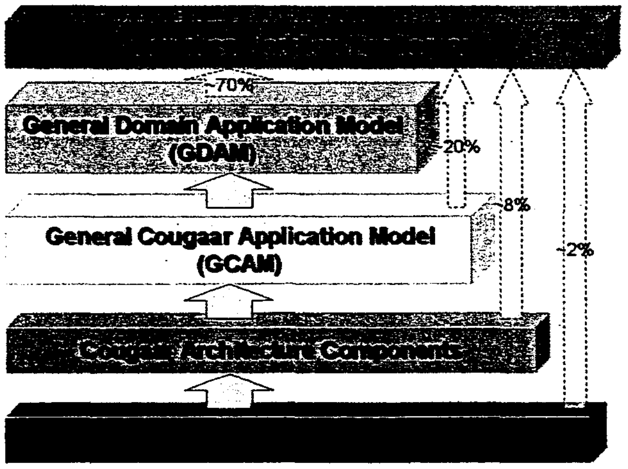
~70%
General Domain Application Model
(GDAM)
~20%
~8%
General Cougaar Application Model
(GCAM)
~2%
Cougaar Architecture Components

Fig. 3. CMDA System Abstraction Layers

Figure 3 delineates all the abstract layers that lie above the Java code. The GCAM layer, which has the largest number of components. The width of the boxes represents the extent to which the application can be represented by the layer. The ability to capture and/or implement the intended application's requirements increases as one progress through the layers, with the Java layer having the capability to implement all the requirements. The workflow is to be described using XPDL standard, defined by the Workflow Management Coalition, which provides a formal model for expressing executable processes that addresses all aspects of enterprise business processes. XPDL was chosen because the language focuses on issues relevant to the distribution of work and workflow processes than defining web services as in other standards such as BPML and BPEL.

The solid arrows moving upwards from the Java layer through the GDAM layer represent the composition of more concrete components to satisfy the domain level abstraction specified by the user. The dashed/transparent arrows pointing up to the domain application layer from the other layers depict the alternative components that can be obtained when a suitable GDAM component is not available. The values on the dashed/transparent arrow indicate the projected amounts of components from the various alternatives in the development environment.

5 Conclusions

Software development can be thought of as the evolution of abstract requirements into a concrete software system through a series of transformations and refinements. Even in moderately complex systems, this transformation is often too involved for fully automated means.

MDA provides a systematic way of capturing details during elaboration and refinement through the mapping from CIM to PIM, PIM to PSM and ultimately rendered as an executable software system. MDA as currently defined appears to have utility if used in moderation. However, for CMDA, it is not a panacea by any stretch. It still requires considerable work and strategic application.

Cougaar is complex requiring considerable mappings and transforms. For this reason, we chose an assembly centric approach with simple formalisms to start. The CMDA approach has substantial transformation challenges in generating software artifacts such as requirements, design, code, and test cases automatically. The artifacts are generated from models assembled using components or beans belonging to two abstract layers namely GDAM (abstracts the domain and generic agent system) and GCAM (abstracts the Cougaar system). A bean will contain nuggets of requirement, design, code, test and documentation details pertaining to that component along with transformation information. The CDMA system combines assembly approach with transformations in small concept to generate the artifacts.

A comparison study of formal methods was conducted to identify the suitable language representation for the GCAM and GDAM components. The selection criteria for the comparative study included criteria such as object-oriented support, usability and tool support. The study concluded that the complexity of the system, coupled with the need for completeness and correctness compels using a hybrid language representation (combination of UML and VDM++) to achieve transformations. The transformation engine will generate the required software artifacts, from the GDAM and GCAM models assembled by the developers, by parsing the various sections and portions in the beans.

6 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 H. Lally Singh.

References

1. Russell, S., Norvig, P.: Artificial Intelligence: A Modern Approach. Second edn. Pearson Education, Inc., Upper Saddle River, New Jersey (2003)
2. Dogac, A., Cingil, I.: Agent technology. In: B2B e-Commerce Technology: Frameworks, Standards and Emerging Issues. Addison-Wesley (2004)
3. —: Grasshopper 2: The agent platform. IKV++ Technologies AG (2004) http://www.grasshopper.de.
4. —: Agent oriented software. The Agent Oriented Software Group (2004) http://www.agent-software.com.
5. —: Cognitive agent architecture (Cougaar). Cognitive Agent Architecture (Cougaar) Open Source Project (2004) http://www.cougaar.com.
6. —: Java agent development framework (JADE). JADE Board (2004) http://jade.tilab.com.
7. —: The foundation for intelligent physical agents (fipa). FIPA Secretariat (2004) http://www.fipa.org/resources/livesystems.html.
8. —: Cougaar architecture document. Technical report, BBN Technologies (2004) Version for Cougaar 11.2.
9. —: Cougaar developers guide. Technical report, BBN Technologies (2004) Version for Cougaar 11.2.
10. Bass, L., Clements, P., Kazman, R.: Software Architecture in Practice. Addison-Wesley Publishing Co. (1998)
11. Dorfman, M., Thayer, R.: A Review of Formal Methods. Computer Society Press (1996)
12. Rouff, C., Vanderbilt, A., Truszkowski, W., Rash, J., Hinchey, M.: Verification of nasa emergent systems. In: Proceedings of the Ninth IEEE International Conference on Engineering Complex Computer Systems, Florence, Italy (2004)
13. —: Information on VDM. The Center for Software Reliability (2004) http://www.csr.ncl.ac.uk/vdm/.
14. Plat, N., Gorm-Larsen, P.: An overview of the ISO/VDM-SL standard. ACM SIGPLAN Notices (1992)
15. McGibbon, T.: An analysis of two formal methods: VDM and Z. Technical report, Data & Analysis Center for Software, Rome, NY 13441-4909 (1997) http://www.dacs.dtic.mil/techs/2fmethods/title.shtml.
16. —: The toolbox newsletter. IFAD (2000) http://shinsahara.com/www/vdm/NewsLetter/issue5.doc.