NAVAL
POSTGRADUATE
SCHOOL

MONTEREY, CALIFORNIA

DISSERTATION

RAPID PROTOTYPING OF ROBOTIC SYSTEMS

by

William James Smuda

June 2007

Dissertation Supervisor: Mikhail Auguston

REPORT DOCUMENTATION PAGE	*Form Approved OMB No. 0704–0188*

1. AGENCY USE ONLY	**2. REPORT DATE** June 2007	**3. REPORT TYPE AND DATES COVERED** Dissertation

4. TITLE AND SUBTITLE: Rapid Prototyping of Robotic Systems **6. AUTHOR** Smuda, William J.	**5. FUNDING NUMBERS**
7. PERFORMING ORGANIZATION NAME AND ADDRESS Naval Postgraduate School Monterey, CA 93943–5000	**8. PERFORMING ORGANIZATION REPORT NUMBER**
9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES) N/A	**10. SPONSORING / MONITORING AGENCY REPORT NUMBER**

11. SUPPLEMENTARY NOTES

The views expressed in this dissertation are those of the author and do not reflect the official policy or position of the Department of Defense or the U.S. Government.

12a. DISTRIBUTION / AVAILABILITY STATEMENT Approved for public release; distribution is unlimited.	**12b. DISTRIBUTION CODE A**

13. ABSTRACT

This effort describes a systems engineering approach to the design and implementation of software for prototyping robotic systems. Developing networked robotic systems of diverse physical assets is a continuing challenge to developers. Problems often multiply when adding new hardware/software artifacts or when reconfiguring existing systems. This work describes a method to create model-based, graphical domain-specific languages. Domain-specific languages use terms understandable to domain engineers as well as abstract software engineering decisions. This methodology enables domain engineers to create quality executable prototypes without being versed in the intricacies of software engineering.

14. SUBJECT TERMS Robotics, Prototyping, Component Based Software Engineering, Model-driven Architecture, Domain-specific Languages	**15. NUMBER OF PAGES** 251
	16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT Unclassified	**18. SECURITY CLASSIFICATION OF THIS PAGE** Unclassified	**19. SECURITY CLASSIFICATION OF ABSTRACT** Unclassified	**20. LIMITATION OF ABSTRACT** UL

NSN 7540–01–280–5500 **Standard Form 298 (Rev. 2–89)**
Prescribed by ANSI Std. 239–1

THIS PAGE INTENTIONALLY LEFT BLANK

RAPID PROTOTYPING OF ROBOTIC SYSTEMS

William James Smuda
United States Army, Tank Automotive Research Development and Engineering Center (TARDEC)
B.S., Engineering, University of Illinois, 1977
M.S., Computer Science and Engineering, Oakland University, 1988

Submitted in partial fulfillment of the
requirements for the degree of

DOCTOR OF PHILOSOPHY IN SOFTWARE ENGINEERING

from the

NAVAL POSTGRADUATE SCHOOL
June 2007

Author: ______________________________
William James Smuda

Approved by:

Mikhail Auguston Professor of Computer Science Dissertation Supervisor and Chair	Luqi Professor of Information Sciences
Kevin Squire Professor of Computer Science	Don Brutzman Associate Professor of Applied Science
James Overholt US Army, Research Engineer	

Approved by: ______________________________
Peter Denning, Chair, Department of Computer Science

Approved by: ______________________________
Julie Filizetti, Associate Provost for Academic Affairs

THIS PAGE INTENTIONALLY LEFT BLANK

ABSTRACT

This effort describes a systems engineering approach to the design and implementation of software for prototyping robotic systems. Developing networked robotic systems of diverse physical assets is a continuing challenge to developers. Problems often multiply when adding new hardware/software artifacts or when reconfiguring existing systems. This work describes a method to create model-based, graphical domain-specific languages. Domain-specific languages use terms understandable to domain engineers as well as abstract software engineering decisions. This methodology enables domain engineers to create quality executable prototypes without being versed in the intricacies of software engineering.

Software systems, like physical systems, require explicit architectural descriptions to increase system level comprehension. Since non-software specialists do most experimental work, this effort suggests a convenient graphical, domain-specific notation to specify the prototype architecture framework. The framework specifies components using domain-specific icons. The Meta-model defines constraints, connections and available operations with components transparently to domain expert.

In this domain, the reuse of hardware/software artifacts (platforms, sensors, controls) is common. The challenge is to configure them into a prototype to examine a particular requirement. This architecture description supports multiple communication strategies between components and the tool and automatically configures the necessary wrappers for the artifacts.

This dissertation suggests a uniform framework for a component and documentation repository. A set of rules operate on the domain model to compose software components needed to create an aggregate system. The same set of rules composes documentation for aggregate system operation. As a result, users of the prototyping environment are able to stay at a high level of abstraction and need not concern themselves with the details of the composed and generated code.

Simultaneously, the prototyping environment generates appropriate information for installation and operation of all parts of the system.

TABLE OF CONTENTS

THIS PAGE INTENTIONALLY LEFT BLANK

LIST OF FIGURES

THIS PAGE INTENTIONALLY LEFT BLANK

LIST OF TABLES

ACKNOWLEDGMENTS

It would be impossible to list everyone who encouraged and supported me over the nine years that this effort spanned. From co-workers near and far to my extended family and friends I was always able to count on a few words of encouragement.

At US Army Tank Automotive Research Development and Engineering Center (TARDEC), I would like to first thank TARDEC's Ms. Pame Watts, without whom this would never have happened. I would like to thank the late Dr. Mike Sabo, a friend and mentor, who convinced me this was something I should try. I would like to acknowledge my Supervisors Tom Washburn at the start, Dr. Grant Gerhart in the middle and Dave Thomas at the end who provided unwavering support and encouragement. Finally, I would like to thank my teammates in the TARDEC Robotics Mobility Lab for their support and enthusiasm. Finally, I would like to thank TARDEC and TARDEC management over the years for their support of continued learning for engineers and scientists.

I would like to thank those who served on my committee, Dr. Valdis Berzins, Dr. Luqi, Dr. Kevin Squire, Dr. Don Brutzman, Dr. Barrett Bryant at the Naval Postgraduate School, and Dr. Jim Overholt from TARDEC. I would especially like to thank Dr. Mikhail Auguston, my Advisor and Committee Chair for his insights and encouragement, especially over the last two years when this effort really began to take shape.

Finally, I would like to acknowledge my family, my exceptional wife Pat and my two exceptional children, Craig and Gwendolyn. Whenever I felt low or ready to stop, all I had to do was look to my family for encouragement. During the time I was working on this program, Pat finished a Masters of Business Administration. Craig finished high school, earned a BS and MS in Bio-medical Engineering, and in now studying for a dual MD, PhD at Northwestern Medical School. Gwen finished middle school, high school and is now a senior studying Mathematics at the University of Chicago.

THIS PAGE INTENTIONALLY LEFT BLANK

EXECUTIVE SUMMARY

This effort describes a systems engineering approach to the design and implementation of software for prototyping robotic systems. Developing networked robotic systems of diverse physical assets is a continuing challenge to developers. Problems often multiply when adding new hardware/software artifacts or when reconfiguring existing systems.

DoD needs tools and techniques that can accentuate current acquisition guidelines. Acquisition is requirements-driven. When requirements are not representative of the need, DoD wastes money and time. Initial requirements are often hasty, unclear or contradictory, particularly when generated based on current operational needs. Therefore, requirements analysis plays an important part in the DoD acquisition strategy. A recognized tool to assist DoD acquisition professionals in their analysis is prototyping.

This work describes a method to create model-based, graphical domain-specific languages for robotic system prototyping. Domain-specific languages use terms understandable to domain engineers as well as abstract software engineering decisions. This methodology enables domain engineers to create quality executable prototypes without being versed in the intricacies of software engineering.

Software systems, like physical systems, require explicit architectural descriptions to increase system level comprehension. Since non-software, specialists do most experimental and prototyping work, this effort suggests a convenient graphical, domain-specific notation to specify the prototype architecture framework. The framework specifies components using domain-specific icons.

In the DoD robotics domain, the reuse of hardware/software artifacts (platforms, sensors, controls) is common. DoD labs are often called on to examine proposed new uses of existing equipment or modifications to existing equipment. The challenge is to configure artifacts into a prototype to examine a particular requirement.

A Meta-model is an architecture description of the system under consideration. The Meta-model defines constraints, connections and available operations with components transparently to domain experts. Software engineers use the Meta-model to define elements, connections and constraints. The Meta-model has explicit definitions of the artifacts and their connections as well as constraints on the elements and the connections.

The Meta-model translates into a domain-specific model. The domain-specific model supports multiple communications strategies between components and provides an output for a set of rules that automatically configure the necessary wrappers for the artifacts.

This dissertation suggests a uniform framework for a component and documentation repository. A set of rules operate on the domain model to compose software components needed to create an aggregate system. The same set of rules composes documentation for aggregate system operation. As a result, users of the prototyping environment are able to stay at a high level of abstraction and need not concern themselves with the details of the composed and generated code. Simultaneously, the prototyping environment generates appropriate information for installation and operation of all parts of the system.

This effort includes an example of applying the tools and techniques to a basic robotic system. A Meta-model for a constrained robotic development strategy (multiple input, single output artifacts) is created and described in a Defense Advanced Research Planning Agency (DARPA) developed tool. The tool generates a domain model and an example domain application is developed. A set of rules associated with the Meta-model paradigm operate on the completed domain model. The rules assemble and configure preprogrammed components and create a main program. The result is a set of programs to create an operational aggregate prototype system.

I. INTRODUCTION

A. PROBLEM STATEMENT

The Army has made a heavy investment in robotics and robotic sensor technology, both for current operations and especially for Future Combat System (FCS). Both current robots and FCS rely on distributed assets communicating over wired and wireless networks, albeit on different scales. Material developers need high-quality hardware and software prototype systems to conduct large tradeoff analysis in a reasonable time.

Integrating artifacts to create a complete system is an arduous, time-consuming task, particularly when the requirements are poorly stated or understood. Developers undertake prototyping efforts to refine and clarify requirements. Developers require tools to create rapid prototypes and the assumptions used in creating these prototypes must be captured.

B. OBJECTIVE

The objective of this work is to apply a systems engineering approach to developing new methodologies and tools for creating software to rapidly link disparate hardware and software items, such as Robot Platforms, Operator Control Units and stand-alone Software Controls. Currently, engineers create most prototype software by hand in an ad hoc fashion. Ad hoc processes lead to much rework and many non-reusable software modules. The engineers creating the modules are usually not software experts. They may be talented, but they are not versed in the intricacies of current software engineering practices. The codes produced are subject to “rot,” particularly when the engineer that produced them moves on to another project.

The suggested approach leads to definitions of tools and techniques to transfer software engineering expertise to domain engineers, who are not necessarily versed in software engineering. The suggested tools allow software engineers to build models in a language understandable to software engineers, Unified Modeling Language (UML) for example, then translate and abstract the software engineering models into domain-specific models understandable to domain engineers. The design of models and associated software components considers preservation, along with documentation, up front to facilitate reuse.

The methodology includes automatic generation of software to simplify development of distributed, embedded and real-time robotic systems. This work extends into the more general case of communicating heterogeneous distributed systems. Automatically generated code allows insertion of legacy and newly developed artifacts into a prototyping environment. The code intercepts artifact functions and binds them with functions needed to exercise the artifacts outside of the native environment. Wrappers and glue code are tailored the current state of the prototyping environment. Automatic generation of the code will enhance the development environment by reducing rote work and producing consistently behaving module interfaces.

The major contributions of this effort are tools and techniques to transfer software engineering expertise to domain engineers.

- This effort discusses in depth a method to specify model-based architecture. An abstract software engineering Meta-models defines a system paradigm. This Meta-model will constrain the operations available to the domain engineer, which will enable creation of a family of domain models using a common message set and reusable, pre-defined software components.
- The domain-model is a graphical model with domain-specific notation. Icons of the model will be familiar to the domain engineer as hardware and software artifacts of interest. Each artifact will be wrapped wit a set of configurable components to translate between legacy protocols and a common message protocol; add instrumentation; and enable artifacts to communicate via an arbitrary network protocol
- A set of rules operate on the completed domain model to generate and compose software needed to create an aggregate system for prototype operation. The rule set composes documentation along with the software. The documentation will guide the prototype user in setup and operation of the prototype.

– Self-documentation. Domain engineers create prototypes using application specific models from a domain paradigm predefined by a Meta-model. Predefined and coded software objects are available in the prototype system Assembly takes place within a domain-specific model that is an offspring of a Meta-model. The approach saves both for future reference.

– Simplify the software development process by using standard message sets defined in XML. Proprietary commands create a "Tower of Babel" effect. Attempting to extended systems with third-party hardware or software is difficult and time-consuming. Wrapping legacy items and emerging items with adaptors that interface a common message set reduces rework and increases the potential for interoperability in the prototype environment. The prototype systems create a set of concise requirements to facilitate production systems.

C. MOTIVATION

In virtually every case when the military seeks to develop a system, initial requirements do not fully represent the final solution. Members of the Army and other Research and Development communities often receive requirements for a new system or an upgrade of a system that depend on capabilities that do not yet exist. The requirements need to be refined, tuned and better understood. One of the most powerful tools to accomplish this is to create prototypes and use them in a limited test. Therefore, prototypes should be amenable to instrumentation. Developers specifically configure prototypes to measure some aspect of the system, either internal or external. In either case, straightforward interfaces are necessary for the engineers and technicians creating and operating the prototype. The prototype, in its environment, must capture information, both about the configuration of the prototype and the results of running it.

Considering the steps in creating a system, if development proceeds in an ad-hoc fashion, all too often a considerable amount of information is lost; mainly due to the lack of feedback and lack of documentation requirements in ad hoc development. In general, a system creation has four steps:

- Initiation – Someone has an idea, or a need. It may be refined and matured informally or semi-formally. The need may come directly from the user, or someone may perceive a user need and move toward inventing a solution to fulfill that need.
- Requirements – A review of the need translates it into a requirements document. In the military, the need may be expressed in a short (as little as one page) Operational Need Statement (ONS) or Urgent Need Statement (UNS). A chain of command for review validates the need statement and system developers receive the document. An inventor may perform this step informally, or via research proposals. A subsequent team attempts to formalize the requirements and create a structured requirements document.
- Design – A design team acts on the requirements document. The design team does their job and creates a scope-of-work.
- Implementation - The implementation team creates an artifact that, if all went well meets the need of the original idea.

Of course, this is a highly idealized scenario. Unfortunately, often there is some disconnect in the four steps, ending in an unacceptable result to the customer. Many things can go wrong at each level. The original idea may be too complex; the original idea may request physically impossible attributes, unreasonably expensive attributes or it may have conflicting attributes. In addition, access to the originator of the idea is usually not available, or difficult to obtain. In some cases, the requirements may not be fully obtainable due to technical obstacles. The design team may misinterpret requirements, or the requirements may not fully capture the idea. Finally, implementation decisions may render the product unusable in the target environment.

This is of course why researchers and developers create prototypes. Early creation of prototypes prevents misunderstandings from propagating through the development cycle. If created properly, the prototype facilitates information capture and produces a satisfactory product.

Yet another aspect of prototyping is spiral development. Organizations are increasingly implementing spiral development to produce partial solutions. Spiral development prototyping is useful when requirements are unclear, technology gaps are known and to address cost issues.

D. SUMMARY - SYSTEMS ENGINEERING APPROACH

A systems engineering approach is defined by the DoD [1] as a problem-solving process used to translate operational needs and/or requirements into a well-engineered system solution. It is an interdisciplinary approach. It includes engineers, technical specialists, and customers. Systems engineering creates and verifies an integrated and life-cycle-balanced set of system product and process solutions that satisfy stated customer needs.

This effort addresses prototyping efforts at the early part of a system life cycle but treats the prototyping process as a mini-life cycle within the larger procurement effort. An interdisciplinary team interacts with the various phases of the prototype and creates a set of interrelated products to produce rapid prototypes. The results of the prototype evaluations are included as products in the larger effort. The individuals within this effort may play a part in the larger effort as time goes on or not, but the process preserves prototype effort intermediate results for use, as needed, in future segments of the system life cycle.

THIS PAGE INTENTIONALLY LEFT BLANK

II. GROUND ROBOTICS BACKGROUND

A. INTRODUCTION - THE COMING OF AGE OF GROUND ROBOTICS

News articles since the beginning of the Wars in Afghanistan and Iraq abound with stories of how unmanned aircraft have made significant contributions to the war effort. Flown from many miles away, even as far as halfway around the world, they have brought a new element to our fighting capability. Initially flown for reconnaissance, they have since been fitted with armament and take aviators out of harms way.

News articles immediately began to chronicle the impact of ground robots on our war efforts. Unlike their airborne brethren, soldiers more typically control ground robots in the areas where they operate. Smaller in size, and operating among the obstacles on the ground, they have difficulty communicating with satellites; they also cannot survive the time delays that a long-range communication link imposes. (Typically launched and landed locally long-range Unmanned Air Vehicle control transfers to a distant remote pilot at cruising altitude). Nonetheless, particularly in Explosive Ordnance Disposal (EOD), inspection and scouting missions, small robots are coming of age.

An article in the December 2004-*Wired* Magazine [2] describes a Foster Miller Talon Robot equipped with a weapon. A human operator controls the robot from out of the line of fire. Soldiers and robot developers say it only makes sense; robots do not have a family at home, robots do not get tired and robots are replaceable. On the other hand, armed robots are a tremendous paradigm shift for the Army. In 2003, even small tele-operated robots generated suspicion and sometimes even outright disdain. Even though Army Transformation via Future Combat System calls for armed robots, the Army did not expect them for at least a decade. The Army still needs to work out important issues of training, tactics and safety completely, but commanders on the ground and the course of the war are accelerating efforts.

One of the reasons for this change of heart is the phenomenal success of the ground robots deployed in Iraq and Afghanistan beginning in 2004. The deployment of EOD robots to counter Improvised Explosive Devices (IEDs) has made a significant contribution to the safety of EOD technicians. The Army Rapid Equipping Force (REF) deployed Omni-Directional Inspection System (ODIS) robots for Traffic Control Point vehicle inspections. These robots provide the operators with all-important standoff from potential explosive devices, and get them out of the line of fire during ambushes often associated with a contrived situation.

The Army can use Unmanned Ground Vehicle (UGV) technology in a number of ways to assist in counter-terrorism activities now. In addition to the conventional uses of tele-operated robots for unexploded ordnance handling and disposal, water cannons and other crowd-control devices, robots can also be employed for a host of terrorism deterrence and detection applications. As recently as 2004, users were not ready for fully autonomous vehicles [3]. By 2006, units began to request autonomy for repetitive behavior. There are many dangerous missions in Military and Homeland Defense operations. Research engineers need to respond quickly to emerging threats and enemy tactics. The current threats are often booby traps, Improvised Explosive Devices (IED), car bombs and suicide bombers. These are often low-signature devices, with a large danger zone. The task is to create tools to detect and neutralize threats from within the danger zone, while keeping our soldiers safe (see **Figure 1**).

These tools include robotics and sensor networks. They need to be effective within the danger zone, as well as operate in a timely fashion. To increase operational efficiency, military requirements are now including autonomous mobility attributes; however, many in the robotics community hold to the tenet that there is a complex intertwining between autonomous vehicle behavior and autonomous mission understanding. Researchers believe that they should develop robots with autonomous mobility in parallel with mission sensor understanding. The robotics community needs tools and techniques to make the process efficient and effective.

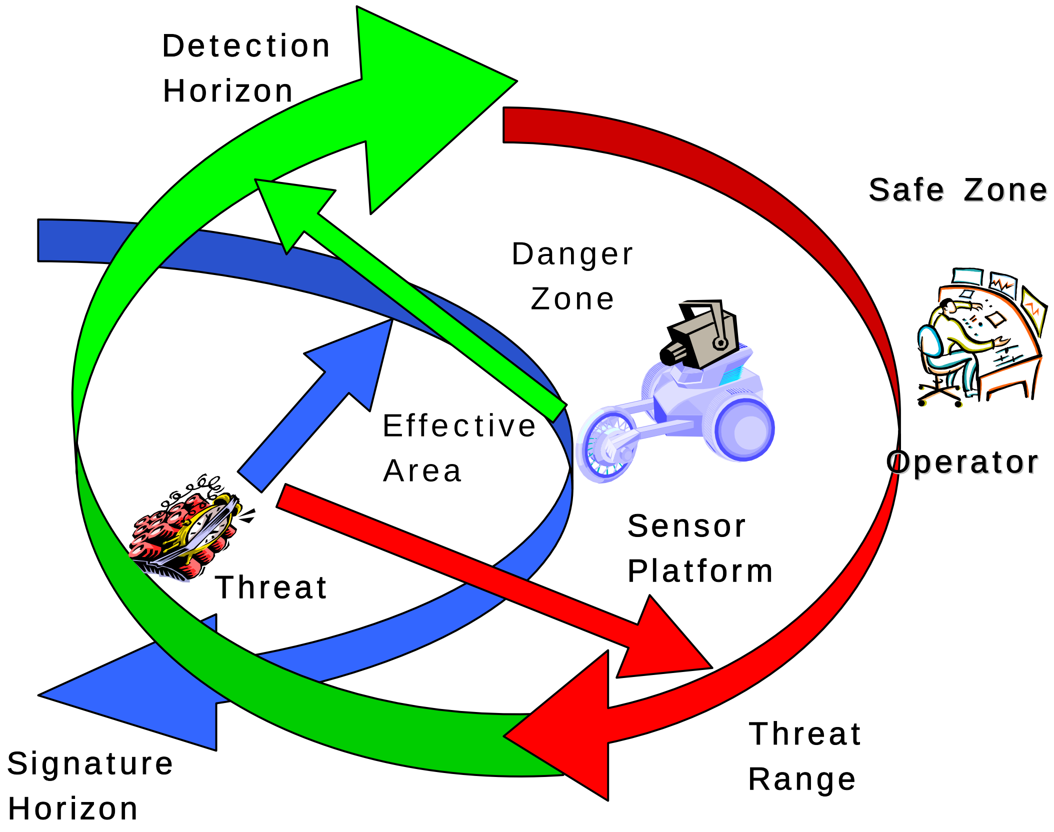


Figure 1. A Robot's Effective Area is the Overlap between the Signature Horizon and the Detection Horizon

The US Army Research Institute conducted Human performance studies to explore new approaches for battle command as may be experienced by soldiers using the Future Combat System (FCS). FCS concepts call for unprecedented integration of automation, sensors and robotics. One of the FCS goals is to reduce the size of the command group. An FCS challenge is to find the optimum workload for command group soldiers. As expected, as workload increases, at the "too-high" levels of complexity, the information and battle space managers' performance drops sharply [4]. The robotic system developer's challenge is to invent fused sensor information and mission awareness tools to reduce the amount of information that the humans in the loop need to process and communicate to their associates.

In order to reduce the levels of complexity introduced to the soldier on the battlefield, or the first responder in a disaster situation, researchers in the robotics community plan to increase levels of autonomous mobility for robotic systems. Autonomous mobility is made possible by creating hardware-software systems to fuse sensor data, creating mission-planning algorithms and creating mission-execution algorithms.

Sensor Fusion is a complex interaction of proprioceptive sensors and algorithms to provide a composite indicator of a parameter of interest, such as position or obstacles. A human analogy is what humans call flavor, a fusion of taste, smell and texture. Fused sensor data may be layered; continuing the human analogy, taste is the fusion of salty, sour, sweet, bitter and umami taste sensations. An example of robotic sensor fusion is a navigation package composed of several different types of sensors. For example, Ojeda and Borenstein fuse three different types of proprioceptive sensors in their FLEXnav system [5].

FLEXnav collects gyroscopic information, wheel encoder outputs and accelerometer outputs into a fuzzy rule based system (**Figure 2**). This expert system outputs attitude estimates. The attitude estimates are collected and joined with the same wheel encoder outputs in a second expert system to estimate position. The simple interface to this complicated software package is a simple {x,y,z} triplet.

Mission Planning is another complex aggregate of available fused sensor information, algorithms and à priori real world data, combined with current operational requirements. Mission planning for a robotic convoy includes physical or temporal separation of vehicles in the convoy and their positions within the convoy, as well as route preferences. An real-time inspection mission plan would include using fused sensor information to determine the geometry of the vehicle (tire positions, bumper positions, physical location, etc.), and then creating a driving path for the robot to obtain maximum coverage of the vehicle undercarriage.

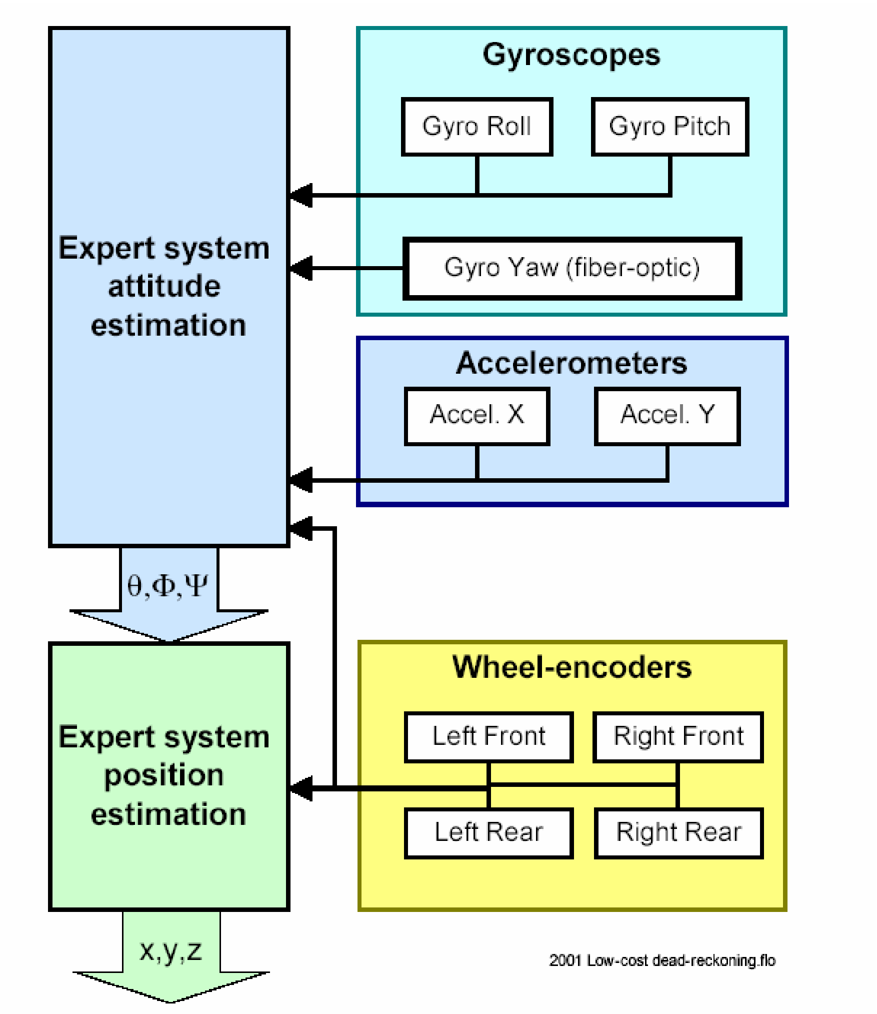


Figure 2. Sensor Fusion Example, a Block Diagram of the FLEXnav System [From Ref. [5]]

Mission execution uses real-time information provided by fused proprioceptive sensors and the mission plan to accomplish the mission. Mission execution may also involve operation and fusion of mission sensors, as well as other mission packages including manipulators. In complex scenarios, mission execution may involve communicating with other assets, such as other robots or fixed sensors.

Developers overlay mission awareness onto the autonomous mobility to fuse data from mission sensors in order to provide a composite indicator of threat to the operator. This is the key to successful autonomous operation. The goal is to provide some hardware/software modules to reduce the data load on the operator and/or enable automation of robotic operation [6]. The first goal is to remove personnel from the danger zone. In automotive applications, manufacturers can sometimes create sensitive sensors that alert the operator to a hazard in time for human reaction. For military and police activities, this is often technically unfeasible or cost-prohibitive; a solution is to move sensors into the danger zone on a robotic mobility platform. In either case, the key is creating modules to interpret sensor data and alert the human operator that a hazard is near.

Integrating these modules is a software intensive task in most cases, since the software modules involve non-deterministic algorithms. Engineers and researchers implement the algorithms within artificial intelligence tools, such as expert systems and neural networks. These types of tools involve intensive prototyping and training for particular situations. As the situation changes, either due to new enemy tactics or due to new missions, the algorithms or training sets often need updating. New sensors, both proprioceptive and mission, are becoming available due to intense research and rapid commercialization. To be responsive to user need, researchers must have tools and architectures in place to rapidly integrate sensors, mission planning, mission execution and mission awareness modules as they mature.

B. POTENTIAL NEAR TERM ROBOTIC MISSIONS

This section describes selected robotic missions that are current topics for both sensor and autonomy research. Many have common characteristics, such as communications, mission package interfaces and proprioceptive sensor requirements. Others have diverging requirements, such as safety, that depend on their application as well as the vehicle size and operation scenarios.

1. Under-vehicle Inspection

Vehicle inspections at critical checkpoints have always been an important part of area security scenarios. At the most secure locations, soldiers routinely conduct real-time inspections to both detect and deter transportation of contraband, and bombs. Soldiers typically conduct these inspections manually by physically climbing under-vehicles or with a so-called “mirror on a stick.” In either case, the inspector exposes himself to the vehicle under inspection and must attempt to inspect relatively dark and inaccessible cavities.

Some relatively simple autonomous tasks are under consideration, such as automatic staging and parking. Tasks that are somewhat more difficult include autonomous path planning in the presence of additional axles and/or trailers, as well as lines of vehicles.

A more difficult computing problem presents itself when considering mission understanding for under-vehicle inspection. Experiences at vehicle checkpoints tell us that human inspectors are good at noticing anomalies. That is, they do not memorize under-vehicle configurations. The inspectors notice shiny things, unusual lumps or panels, disturbed areas or extra dirty areas. These indicators may not only alert the operator, but also feed back into the inspection vehicles motions as the operator maneuvers to get a better look. This scenario is certainly not amenable to deterministic computing, but it has the potential to become progressively more doable as adaptive computing techniques mature.

2. Convoy

Military Definition: A group of motor vehicles organized for the purpose of control and orderly movement with or without escort protection. The Army also defines a convoy as any group of six or more vehicles temporarily organized to operate as a column with or without escort proceeding together under a single commander, or the dispatching of 10 or more vehicles per hour to the same destination over the same route.

Future Army robotics applications talk about convoys as leader-follower operations. Leader-follower operations can be familiar columns of vehicles or vehicles separated by up to a day. Obviously, in the latter case, each follower vehicle will be an autonomous vehicle following a pre-defined route. The route may have experienced changes during the time since the leader passed that requires a reaction from the follower.

3. Explosive Ordinance Disposal (EOD)

EOD Robots are traditionally tele-operated. Increasingly, though, features that make the task easier for the operator, such as preprogrammed arm positions, are being included. One should note, however, that there are a number of tasks that may seem simple, but are often hard problems for robots; collecting played out fiber optic cable is one.

As technology capability increases, mission assessment packages will become more important. It is routine to blow up a suspect package today, because technological aids for risk assessment are not available. In the future, miniature x-ray and chemical detection equipment will be on board the EOD robot, often thus reducing or eliminating this necessity.

In many cases, it is desirable not to explode a found bomb, but to disassemble and analyze it. In some cases, exploding a found bomb will cause damage to surrounding infrastructure (consider a large bomb near glass-walled high-rise buildings). In cases such as this, it might be desirable to have two or more EOD robots working together to either move or contain and disable the explosive.

4. Mine Clearing

Mine clearing is now typically accomplished using specialized equipment. Often it is a heavy combat vehicle with mine plows or rollers attached. Increasingly, however, military units are disabling anti-personnel mines by beating them with chains attached to a flail mechanism on a lightweight construction vehicle, similar to a bobcat.

These are brute force methods, and not 100-percent effective. Future operations, especially in heavily mined areas like Afghanistan will require a new paradigm in mine detection, marking and defusing. Robots performing this work will often have their mobility functions working in coordination with their mission packages; again, a highly non-deterministic problem.

5. Scout

Scout operations imply that a robot will autonomously take a journey into uncharted territory, inside a building, a cave, a tree line or several kilometers into an urban area. The mobility and navigation package must work closely with the mission package. Suspicious activities detected and recorded by the mission package must be transmitted back to the base and prompt the modification of operations to increase stealth, trigger retreat procedures or both. In scout operations, certain maneuvers may deny sensor data (e.g., Global Positioning System (GPS) data is not available under dense foliage) requiring the triggering of alternate sensors or behaviors.

6. Perimeter Patrol

Perimeter patrol is somewhat like scout, but with a more defined world map. Perimeter patrol robots must obviously work with their detection and suppression mission packages to control the mobility packages.

An interesting solution to perimeter patrol is a fully robotic network of ad-hoc sensors. A host or mother robot has a payload consisting of a number of small robotic sensor platforms. The mother robot learns a route, perhaps from a soldier walking or driving the perimeter. The mother robot dispenses the smaller sensor platforms to provide full coverage of the perimeter. Depending on the situation, the sensors can be either active or passive. In the active state, they may recall the mother robot and/or human guards when a threat is detected. In the passive state, they may just record information and wait for the mother robot to make a pass and relay the data over a low power link. At the end of the mission, the small mobile sensors return to the mother robot as it passes and are available for reuse.

C. SENSORS

Two classifications of robotic sensors are proprioceptive sensors and mission sensors. In some cases, there is overlap or a single sensor has more than one purpose. For instance, a camera used for navigation is also as a visual mission package. The output from the sensor package routes to two different analysis packages simultaneously. A dazzling array of sensors is available today in both groups.

1. Proprioceptive Sensors

Proprioceptive sensors are for navigation and health monitoring of a robot. These sensors may be simple, such as an Infrared (IR) bump sensor, or complex, as in a stereo night vision system. Often, on high-end systems, they are an array of complex sensors with fused data output. In any case, there are rules or implied consequences when a sensor triggers or when a complex sensor output analysis indicates a distinct outcome.

In some cases, a singular sensor output will cause an action, i.e., a bump sensor triggers a stop and reconsiders action. In other cases, there will be conflicts. As long as the conflicts are few and simple, the robot control uses distinct low-level rules, possibly governed by a low-level arbitrator. However, as the quantity, quality and information output of sensors increases to levels required to operate in an arbitrary real world environment, a combinatorial explosion occurs and it is no longer feasible to anticipate every low level action.

In military robotic systems, the desire for autonomous operation requires a plethora of sensors for even relatively simple navigation tasks. Military combat vehicles, by definition, operate in an unstructured environment. Multiple Visual, Infrared, LIDAR, RADAR, and others are needed to understand the near road, far road and road edges. A complex world map needs to be created and updated on the fly. When the vehicle goes off road, the processing required becomes even more difficult. Is a stand of grass or a leafy branch hiding an impenetrable obstacle? Is the vehicle about to drive into a ditch or off a cliff? These are decisions made by human operators regularly.

In civilian or tactical military systems, the situation is even more complicated. Yes, there may be a structured environment; yes, there may be active sensors in or near the road to aid operation. Nevertheless, real world situations do include anomalies: unauthorized pedestrians, breakdowns and repair crews to name a few.

At the June 2005 North American Fuzzy Information Processing Society (NAFIPS) conference, during a Fuzzy Logic panel discussion, one of the panelists, Dr. Lotfi Zadeh, made a comment about the infancy of fuzzy systems. He stated that humans could drive an automobile without consulting any sensors at all. By this, he meant that they were using their senses and contact with the vehicle, as well as their experience. It is not necessary to monitor the speedometer closely; there is no acceleration gauge and GPS coordinates are meaningless to the average driver. During the International Society for Optical Engineering, (SPIE), Defense & Security 2006, Intelligent and Unmanned Systems conference an audience comment was that many people drive on Urban Interstate Highways every day and use behaviors, such as following the car in front of them at a distance at which it is impossible to stop if the vehicle in front panic-stops. Yet they are able to find visual cues to prevent many of these collisions. The stated implication, in both cases was, “Why can’t we automate driving?”

The answer is that human drivers use heuristics to judge when our driving behavior is safe, and this works perfectly well for all but about 40,000 people a year who die in about 3,000,000 traffic incidences each year [7]. Autonomous systems are held to a higher standard. The populace would never accept an automated system that drives anywhere nearly as bad as a person. Sensor systems added to vehicles to increase autonomy also have to have a safety aspect, as well as a capital aspect. Civilian efforts in ground vehicle autonomy focus on highway safety, but may also have a convenience factor as a consumer selling point. Military efforts in ground vehicle autonomy may have additional aspects, but the military trains and transits on civilian highways, and must at least meet the civilian goals.

This discussion leads to a major focus for prototyping robotic systems, that is, safety. Cost is always a driver; prototype sensors need to be integrated into systems for test purposes, fused with other sensors and tested again. Often the prototype sensor is expensive and in short supply. Tools that help to accommodate the availability of hardware in a test environment are also in short supply.

2. Mission Sensors

Mission Sensors may or may not be part of the proprioceptive network on the robot. They may stand-alone or they may trigger higher-level mobility responses in the robot. For instance, if the robot has a vision package with image understanding, it may trigger on an anomaly and request that the robot perform a maneuver to rescan and take a closer look with the camera zoomed.

Other mission scenarios depend, by design, on mission sensors. A mine detector mission package may trigger a marking response on the robot, initiate a defeat device and/or initiate a mobility maneuver to avoid prematurely detonating the mine.

D. SPIRAL DEVELOPMENT STRATEGY

The TARDEC Robotics Mobility Lab (TRML) is the target organization for the first working versions of this effort. TRML has subscribed to a spiral development strategy for robotics since the late 1990’s. This strategy has been particularly effective for development of the Omni-Directional Inspection System (ODIS) robots. Experiences in Software Engineering lead to this approach. It requires early user involvement to mitigate risk.

Spiral development [8] is an evolutionary, risk-driven approach to system development. The spiral development process for software development (**Figure 3**) has been successfully used in a number of DoD software acquisitions and fits well with the DoD Instruction 5000.2 preferred evolutionary acquisition strategy The spiral development process requires user involvement and frequent testing; it is particularly useful when the system requirements are not known up front.

A complete spiral cycle includes 1) client communication, 2) planning, 3) risk analysis, 4) engineering, 5) construction and 6) client evaluation. For each cycle (spiral) the client and developer closely work together to ensure a functional prototype. They reassess risks and assumptions to meet current contractual requirements while leaving room for future growth.

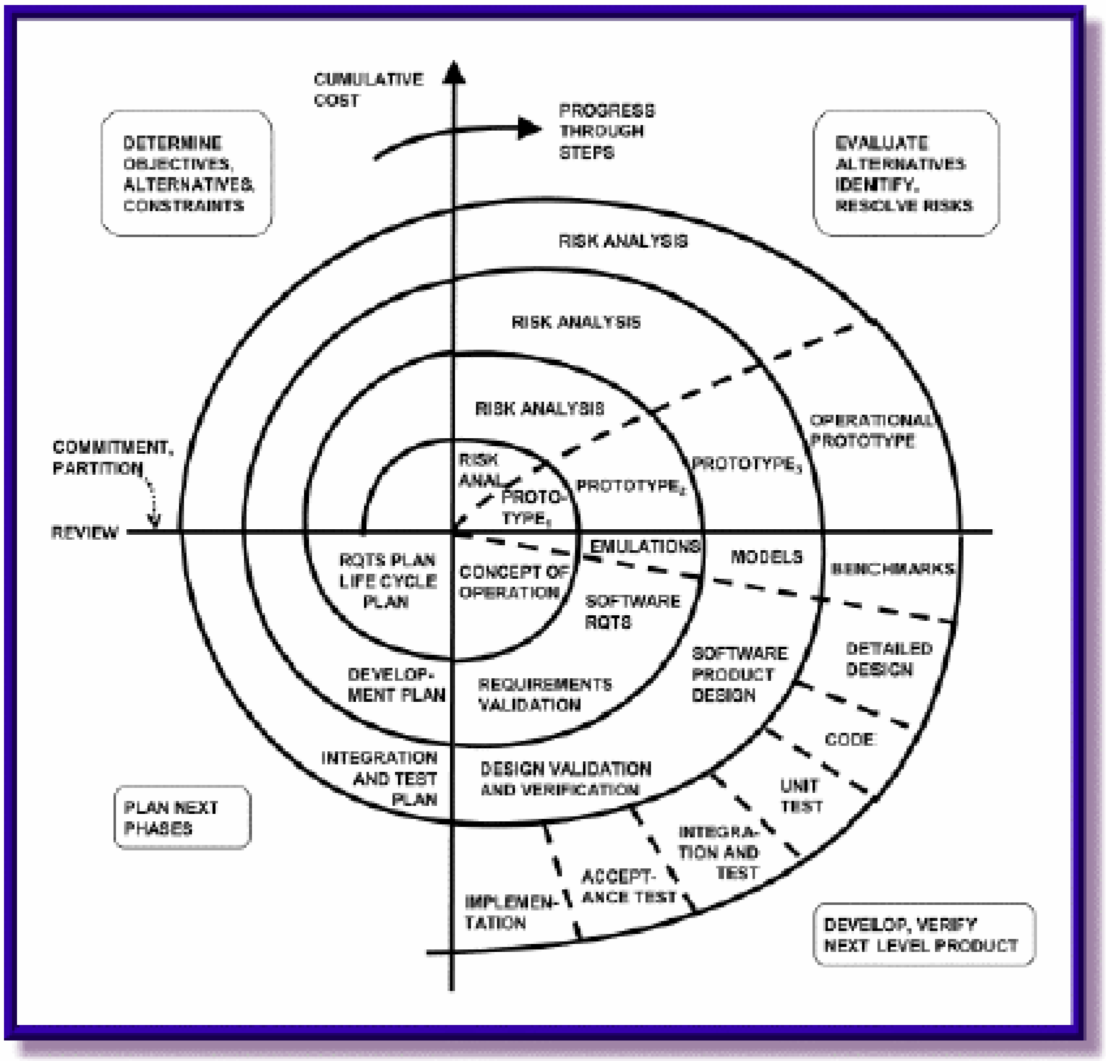


Figure 3. Boehm's Spiral Model for Software Development and Enhancement Illustrates How Progressive Phases Add More Detail [From Ref. [8]]

E. ODIS SPIRALS

1. ODIS-A

ODIS-A was the original vehicle in the ODIS series. It was the outcome of ongoing Army-University basic research conducted cooperatively between TARDEC and Utah State University.

ODIS-A was a convergence of several lines of research:

- Autonomous path planning;
- Autonomous path execution;
- Autonomous control of multiple intelligent wheels; and
- ODIS-A was also a practical follow on to the T-series robots.

Late in 2001, TARDEC conducted an ODIS–A robot demonstration for guards at the Tank Automotive Command (TACOM) front gate. The guards showed no inclination to use the autonomous features, so the next turn of the spiral resulted in a simplified, tele-operated version of ODIS.

As mentioned, the ODIS project was originally a basic research project. After the attack on the World Trade Center on Sept 11, 2001, need for frequent under-vehicle inspection became apparent. The TRML examined current activities and revised the Omni-Directional Vehicle project goals. A new goal was to provide tools to soldiers, guards and first responders as soon as possible. This included releasing partial solutions as they matured to the point where they fulfilled emerging requirements. This project was important to local and Army organizations, as well the TRML customer, the DoD Joint Robotics Program, because it was the only project with realizable budget and timelines that was completely in control of government labs. This meant the TRML was in a unique position not only to develop tools that were essential to the safety of operational personnel, but could apply the spiral development process and demonstrate how to apply the process to other emerging missions.

The ODIS project progressed from several hand-built tele-operated robots to a semi-production unit. The original robots were used for initial interaction with users, including members of the Army Force Protection community, police departments and border security agents. The second version was deployed in Iraq and Afghanistan with operational units for a long-term experiment that continues to this day. The spiral development paradigm developed in the ODIS project is widely heralded as a model of rapid transition of Army-developed technology to the soldier in the field.

2. ODIS-T1

Utah State University built three ODIS-T1 robots. The three ODIS-T1 robots participated in several user experiments, including a Limited Objective Experiment at Fort Leonard Wood in 2002 [9] and a Demonstration Project at the Ports of Los Angeles and Long Beach (POLA/POLB) in 2003 [10].

3. ODIS-T2

Work with the three ODIS-T1 robots led to a hardened, semi-manufacturable prototype, ODIS-T2. The ODIS-T2 featured a ruggedized body and a wearable Operator Control Unit. Delivery of ODIS-T2 units to Iraq and Afghanistan started in February/March of 2004. Additional deployments followed as well as a deployment to a camp in Qatar, where it operated without fault for the duration of a six-month experiment. Lessons learned with ODIS-T2 are leading to the final mobility prototype, ODIS-T3.

4. ODIS-T3

ODIS-T3 features modular wheels for maintainability, as well as an option for off-road wheels. Other features include support for mission packages, such as arms and sensors as well as lower cost internal components.

With the introduction of the ODIS-T3, additional development spirals are already under way. One spiral is investigating multi-spectral sensor integration for under-vehicle change detection [11]. This mission understanding spiral has several facets and internal Army researchers and university researchers are working on it semi-independently.

Another spiral will take advantage of improved sensor support and communications in the ODIS-T3 robot. Neural-Fuzzy Controllers for Autonomous Reactive Navigation [12] are under investigation for integration into the ODIS robot. Combined with the above-mentioned mission understanding spiral, autonomous inspection behaviors will soon be possible.

F. SUMMARY

Since 2004, robots have become an integral part of many military operations. Robots perform the “dirty and dangerous” jobs today. EOD and scout missions predominate today. Convoy, patrol, mine clearing and inspection missions are desirable in the near-term.

Military organizations, encouraged by the successes are looking to robotic researchers to develop new capabilities. In parallel, University, Industry and Government researchers are developing new capabilities, both in platforms and in sensors.

Spiral development strategies deliver the best of the current technology to soldiers now. The ODIS project was an early example of a spiral development applied to a small research robot fielded in a short time. Additional spirals improve the capability or add new functions for additional missions.

III. RELATED WORK

A. INTRODUCTION

The effort touches several research areas. There are some efforts directly related to robotics, but there is more related work at a higher level of abstraction investigating design and construction of embedded systems in general.

1. Inspiration

a. Track Vehicle Workstation

The Track Vehicle Workstation [13] (TVWS) was a research project to allow assembly of dynamic models of tracked vehicles from a collection of hardware component models. In the TVWS, highly parameterized components were stored in a database. The components were accessible via the parts browser (a visualization of the database tree). Components selected from a parts browser were copied into a model tree. Once copied, drop down forms became available to parameterize the components. A completed TVWS model was then "compiled" and researchers conducted an input model to a dynamic analysis program. The TVWS had a collection of tools to monitor execution of the dynamic analysis and to post-process the results.

TWVS had a few insurmountable shortcomings:

- The TVWS project depended on closed tools. The TVWS software could not keep up with changes to the underlying commercial software products.
- It was specialized to interface to a Cray2 supercomputer. When the Cray2 was decommissioned, the cost benefit to port the TVWS could not be justified.
- The projected users were ambivalent toward the TVWS. The domain experts that were to provide components saw TVWS as a threat.

– The development team was composed primarily of mechanical engineers that were new to software development and documentation suffered.

b. LEGO Mindstorms

LEGO Mindstorms™ is a very sophisticated toy [14]. It educational uses range from in middle school science projects, to college-level introductory AI courses. The hardware is restricted, but robust enough for recreational and educational purposes. The computer RCX "brick" is based on a commercial microcontroller and can be programmed with Lego's supplied GUI or with freely downloadable programming tools, NQC or RCX-ADA to name a few. The GUI environment is the inspiration. The GUI provides restricted software architecture to allow programming of the RCX by young and inexperienced users. Children as young as eight years old have been able to successfully program LEGO robots after as little as an evening of tutorials. The GUI is severely restricted, and often described as grade-schoolish, but it represents an intentional architecture and exhibits all the elements of such an architecture: components, connections and constraints. It is robust in its intended environment, and if its marketplace success is any indication, it has achieved its goal.

The LEGO Mindstorms™ GUI features drag-and-drop components that correspond to LEGO components. It has parameters to control how the hardware components interact with the program. It has timers and conditional controls built in. But it is not extensible; the GUI cannot accept new components.

2. Robotic Development Environments

Two significant proprietary efforts related to robotic software development have emerged during the span of this research.

a. Microsoft Robotics Studio

Microsoft Robotics Studio builds on the Visual Studio product and with it hopes to fuel the "Home Robotics Revolution" [15]. Robotics Studio has components to interface with Lego™ and iRobot Roomba™ robots. The Robotics Studio predicates a set of libraries to create controls for Robotic systems. At this point, it seems immature and seems to cater to the robotics hobbyist.

On the other hand, home robotics is in its infancy, while military robotics is rapidly maturing. The Robotics Studio has possible synergy with this effort. Microsoft has expressed interest in the Joint Architecture for Unmanned Systems, as well as in pursuing cooperative efforts with Robotics Labs at the Army's Tank Automotive Research and Development Center (TARDEC) and with the DoD Joint Ground Robotics Enterprise. Hung Pham, co-chair of the Object Management Group's Robotics Domain Task Force, has expressed his excitement in [16].

Microsoft Robotic Studio is currently available as a Microsoft Download [17]. It has a limited run-time library, a simulation environment and a visual programming language, suitable for programming robotic behaviors that is similar to the Lego Mindstorms™ visual language.

b. iRobot Aware

iRobot Aware 2.0 is currently in Beta Test. Microsoft does not yet sell robots: the focus of Robotics Studio is outward. iRobot sell robots; Aware 2.0 focuses on iRobot platforms and a collection of proprietary, licensable and open libraries. Aware 2.0 is a tool for third-party developers to integrate payloads with iRobot platforms. Third party developers can license to iRobot or others.

iRobot describes Aware 2.0 as an extensible networked collection of services along with a component-based architecture. Other architectural descriptions include layered arbiters and publish/subscribe. Aware 2.0 relies on the Python language to configure and customize applications. C++ is the component development language for Aware 2.0. There appears to be a graphical component browser, but the bulk of application development is in Python. Aware 2.0 is amenable to prototype development but targets software developers.

Aware 2.0 treats protocols as components; its applications may be Joint Architecture for Unmanned Systems (JAUS) compatible or compatible with any number of other proprietary or open protocols. The work in this effort treats Aware 2.0 applications as legacy artifacts. Programmers create an appropriate adaptor component to use Aware 2.0 applications in the rapid prototyping environment. The Aware 2.0 architecture does not have explicit constraints. This means domain instance developers are not sharing a mutual set of constraints. This can lead to ambiguity in production efforts and subsequent support.

B. FEATURE MODELING

Prototyping and spiral development is all about features of a system. A spiral can refine capabilities or add new ones, such as an autonomous mission mode (i.e., path following), or a physical attribute (i.e., a manipulator arm). A spiral may also be used to improve a less tangible feature, such as reliability, maintainability or cost.

In order to have smoothly progressing spirals, robotic and other system designers need to be cognizant of how features are related and of where new features should appear. At domain analysis time, analysts create a feature model [18]. It is a tool for domain analysis to communicate information between developers and users and, if preserved, a temporal tool to determine what the original developers were thinking during a previous spiral. Feature models present software developers with a tool, much like an assembly diagram for a mechanical developer. As Czarnecki and Eisenecker state in their chapter on feature modeling in their book, *Generative Programming* [19], "Feature Models provide an abstract, concise and explicit representation of the variability present in the software." One should note that the feature model, like the assembly drawing, is not a full representation of the system; it combines with other models for full system representation. For mechanical systems, assembly drawings are associated with other types of drawings and models. A part may have a detail drawing and an engineering analysis of the strength of the materials required. A software feature model may have other diagrams and analysis, such as timing constraints, state transition diagrams and object diagrams.

The standard example of a feature model is a representation of a car:

- Mandatory features: engine and transmission;
- Optional features: sunroof;
- Alternate features: manual or automatic transmission; and
- Or-Features: Electric motor, an internal combustion engine or both (hybrid).

The car model above is a simple representation of the physical characteristics of a vehicle. In this effort, a engineers construct similar feature models for robotic systems. A feature model may also consider a higher level of abstraction, the prototyping process, for instance. In a robotic prototyping process, the robot feature model is a sub feature of the prototyping process.

The feature model is the foundation of this research. Formalized feature models can automate segments of the prototyping process. Annotating features and translating models preserves information, defines data and component storage, and automates many segments of the process. In particular, by automating the integration of concrete realizations of features using components, a prototyping environment can present domain engineers with constrained choices that will greatly simplify the task of assembling software to construct prototype robotic systems.

Examining a high-level feature model (**Figure 4**) based on the above four recurring phases of a prototyping, one finds a separation of concerns that can be used to advantage in defining a hardware-in-the-loop prototyping environment.

Features are the building blocks used to describe concepts. Features are configurable reusable requirements of a concept. Each feature is associated with a stakeholder or client program. Semantic content of a feature is not directly associated with a feature. Adding semantic information requires associating the feature with an additional model.

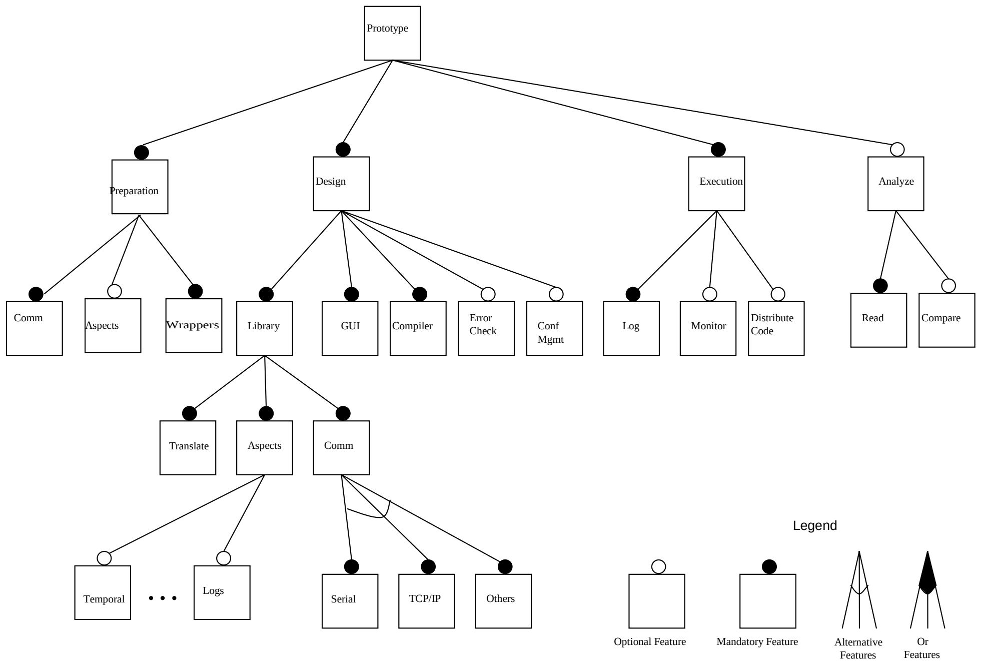


Figure 4. Feature Diagram for a Prototyping Environment

A feature model represents the variable and common features of a concept. An important note is that feature diagrams of concepts can be graphs. This indicates that sub-features may be associated with more than one parent feature.

Feature diagrams begin with a root node known as the concept. The parent of a feature is either the concept or another feature. Features are either mandatory, optional, alternative or or-features. Features can only be included in a concept instance if their parent features are included. Thus, mandatory features are included if their parent feature is included. Optional features may be included in a concept instance or not. Only one feature of a set of alternative features is included in a concept instance. One or more features of a set of or-features can be included in a concept instance.

Features diagrams are decorated with symbols to show their status in the model. Filled circles on the top of a feature are mandatory features; open circles are optional features. Arcs drawn across the lines connecting features indicate the requirements for child features. An open arc indicates a set of alternatives; a filled arc indicates a set of or-features.

Tree-shaped feature diagrams can be used to discover what features may be common to all instances of a concept. A feature is common to all instances of a concept if it is a mandatory feature, and there is a chain of mandatory parent features to the concept root. Similarly, tree-shaped feature diagrams provide a tool to analyze and categorize variability between instances of a concept to understand where the variability occurs.

As noted earlier, not all information explicitly appears in a feature diagram. Additional information, such as semantic description, rationales for inclusion, stakeholders, priorities, etc., is included as annotations and associated diagrams. Implementation details may be expressed using UML class diagrams.

The variability of the concept may suggest implementation strategies. Dimensions suggest compile time variability mechanisms, while extension points may suggest run-time variability.

There are no adequate production tools to support feature modeling available today. Desirable traits of a feature-modeling tool include support for model notation, tools to manage additional information and hyperlinks to supporting CASE or other modeling tools.

The feature modeling process is a study of variability in domain concepts. The process is continuous and iterative, involving identifying as many Use cases, existing feature models, system requirements and additional UML models as possible to identify potential variability points. It also involves recording all supporting information as features become available to the concept.

Feature modeling is used in conjunction with decomposition techniques to create clean and adaptable code. The principle of “separation of concerns” indicates that localizing issues in our models will help us verify that our programs modules map to requirements. In order to separate issues, analysts need to decompose systems:

- Modular decomposition separates systems into units with internal cohesion and minimal coupling between units.
- Aspectual decomposition separates systems into a set of perspectives that cross program module boundaries.

Addressing variability is a key issue in creating reusable software. Decomposition decisions that address the variability discovered in feature modeling result in software with a high level of reusability.

1. Feature Modeling, Discussion

An example use of feature models might be to specify Quality of Service (QOS) requirements for distributed real-time systems. A feature analysis might be a tool to elicit QOS requirements for a particular sub-feature of a concept. If there is a hard QOS requirement, model it as an exclusive mandatory feature. If there is room to operate in a degraded mode, then model QOS as two or three alternate features, i.e., QOS met, QOS failure or network failure.

There exists a Backus–Naur Form (BNF) Grammar [20] for feature diagrams. Information frames supply supplemental information associated with the feature model but not explicitly represented. This frame could then be processed with prolog or another logic engine to automate analysis of large feature models. Processing and analysis of feature models determine where requirements conflict, agree or are redundant, among other considerations. The information frame could be added to the BNF Grammar and an automated feature-diagramming tool could be created.

a. Advantages

Feature models contain a great deal of information. The top-level, visible feature diagram, presents an uncluttered view of the concept in all its variants. From an engineering standpoint, this allows analysis of tradeoffs at variability points. Depending on the type of variability, feature models highlight where the system instances are compile time, or run-time-dependent. Additionally, feature models show where concept instances diverge; this may lead to areas where parallel development teams may be able to work without a great deal of coupling. Conversely, feature models also illustrate areas where instances of a concept are common. The prototype can reuse these areas for other instances of a concept or for future extensions to a concept.

b. Disadvantages

The main disadvantage is also the large amount of information. A feature model represents a multidimensional model; however, not all dimensions are explicitly visible. The features themselves are one dimension, with the edges being a partial dimension. That is, the edges have no explicit information assigned to them. All other dimensions, such as semantics, priorities, examples and rational are below the surface and are not completely defined. Another disadvantage is that the size of the feature diagram can grow quickly. This can result in either a diagram with a large number of features and unless there is a large-format printer available, detail will disappear. Alternately, the feature diagram might be separated into a collection of sub-diagrams. This might work if color or some other discriminator can be used to connect the diagrams.

C. MODEL-DRIVEN DESIGN

There is considerable research conducted in Model-driven Design and Model-driven Architecture. The Object management Group's (OMG) Unified Modeling Language (UML) 2.0 provides increased support. The Generic Modeling Environment from the ISIS center at Vanderbilt University provides a platform for developing Model-driven designs and the embedded systems community has recognized the power of Model-driven design for developing software product lines for automotive, signal and aerospace applications. The Eclipse Foundation has several projects focusing on Model-driven paradigms.

1. UML2.0

The goal of Model-driven Design is to alleviate difficulties created by the low level of abstraction used in creating today's software systems. The OMG Architecture

Group has responded to this by embracing a vision to expand UML and provide support for all phases of the software lifecycle [21]. UML2.0 is an outcome of this vision.

UML2.0 supports modeling from different viewpoints. Structural, interaction, activity, and state viewpoints have some interdependencies, but allow modelers to concentrate on specific concerns.

UML is still in the development phase as a standard. It is a large and complex, making it difficult to grasp in whole. It is clear that experience "from the field" is required to refine and mature the standard.

2. Chrysler AG

Czarnecki, Bednasch, Unger and Eisenecker report on their experience at Chrysler AG for automotive and satellite applications [22]. They describe their experience with Model-driven Design and Feature Modeling tool support with the Generic Modeling Environment (GME) tool.

Domain-specific concepts and features from the problem space map to a set of combinable elementary components in the solution space using configuration knowledge, such as combination restrictions, default settings, dependencies, and construction rules. They use a feature model to define the common and variable features of the products along with supplemental information (binding, priorities etc.) unique to the product under development.

The feature model has a root or concept node and child nodes. The child nodes or sets of child nodes are mandatory, optional, alternative or or-features. The nodes combine in various ways to produce an instance of a concept, (i.e., a car) which can have a manual, automatic or CV transmission, but only one transmission. A car may also have a fossil fuel motor, and electric motor or both.

In the referenced work, they present a UML Meta-model for feature modeling notation-using GME. They also show a derived domain-specific model, also using GME.

3. Embedded System Control Language

Additional work at Vanderbilt University uses the Generic Modeling Environment (GME) tool, along with Mathworks Simulink and Stateflow tools to create the Embedded Control Systems Language (ESCL) to support development of distributed embedded automotive application. ESCL imports the Simulink/Stateflow models into the GME environment. ESCL is a graphical modeling language suitable for use with a suite of sub-languages. Sub-languages are provided to support functional modeling, component modeling, hardware topology modeling and deployment mapping.

The ESCL also has a code generation component. The generated artifacts can synthesize the entire application behavior code, or link external application behavior code.

4. Architecture Analysis and Design Language (AADL)

The AADL is a Society of Automotive Engineers (SAE) aerospace standard for analysis and design of architectures for performance-critical systems [23]. The AADL is a textual and graphical language. It is used to analyze software and hardware architecture.

AADL addresses non-functional aspects of performance-critical systems, such as timing for real-time systems, partitioning safety and security. Using AADL, system designers develop components and analyze the impact of the composed system. Multiple alternatives are created to study trade-offs and the impact of change.

AADL is primarily a software engineering tool. It has International support via affiliation with the SAE. The main effort described in this dissertation is complementary to AADL. Where AADL addresses the big picture, the work described in this dissertation addresses a subset of the bigger effort. Both efforts, AADL and this work, are Model-driven. Two-way metadata exchange has great potential via the Object Management Group (OMG) XML Metadata Interchange (XMI) specifications. The potential is there to generate Prototyping Meta-models from AADL models when they exist, feed back into those models, or form segments of emerging AADL-modeled systems by using XMI.

D. SUMMARY OF RELATED WORK

There are a number of related areas to this work from children's toys to SAE Standards. Microsoft and iRobot are investigating tools for software developers. There is considerable interest in feature modeling, particularly in the automotive and aerospace industries as a component of model driven design strategies. The UML is maturing with version 2.0 and is getting wider acceptance across a broader user base. Additional efforts with the GME, such as, ESCL show the utility of GME. The Society of Automotive Engineers has adopted the AADL as an Aerospace Standard.

This effort builds on and extends from many of the same roots as the above-mentioned work. This work, however, is unique due to the application and user base. The above-mentioned tools have a single specific user; this effort collects and propagates information from an expert in software engineering to an expert in a domain to a technical user. This effort will break the cycle of creating a complete set of adaptor and glue code needed to bring hardware/software prototypes to the test site.

IV. ROBOTIC SYSTEM PROTOTYPING – EXAMINING EMERGING HARDWARE / SOFTWARE SYSTEMS

A. WHY PROTOTYPE?

The section above alludes to the problems examined by robot developers. Many of the operations described in the previous chapter are relevant today in the current conflicts. Many organizations are working to develop the artifacts, robots, communication systems, Operator Control Units, Mission Packages and Controls. In some cases, artifacts work together because they originated at a single organization. However, in most cases, it is difficult to add or change functionality to a particular robotic system. These instances cause great expense and long lead times to respond to changing requirements; a reality forced on us by an adaptive enemy. We are in dire need of tools to assist us in moving solutions forward faster. One tool that is under used is Rapid Prototyping.

1. Rapid Prototyping

Iterative development is one of the identified six best practices of software development [24]. The speed at which prototypes are developed affects the number of iterations available and the time to market. Rapidly producing prototypes allows examining more concepts, finding the useful aspects and excluding suspect concepts. Prototypes also promote collaboration and interaction with customers prior to final decisions.

Rapid prototyping processes have a wide breath, from paper and whiteboard sketching [25] to formal executable languages, such as the Computer Aided Prototyping System (CAPS) [26]. A rapid prototyping environment should be modular for easy modification. It should be simple and easy to use. It should support reuse from a collection of modules. The prototype must be adaptable; small changes should not require revisiting the implementation. The environment should contain a set of abstractions to describe common items encountered while prototyping, such as, data, timing and functions. A rapid prototyping environment must also produce a trace of choices and decisions encountered in the prototyping process.

There are many reasons to create a prototype:

- One might want to investigate and gain a better understanding of the requirements.
- One may need to understand a physical attribute that is too complex for calculation.
- There may be compelling needs; situations require an aspect to be addressed quickly, while normal development progresses in parallel.
- One may subscribe to a spiral development model, where each successive prototype adds additional functionality.
- Marketing.

In any case, a common attribute of prototypes is that they are not the complete and final solution, but they are an important part of system development. Properly used, prototyping reduces risk, shortens time to market, and reduces life-cycle cost. On the other hand, improperly used, prototyping can use up project funds and increase development time. To avoid improper use, prototyping tools are needed to develop concepts and perform test events in parallel. Prototype engineers capture test analysis and feed information back into the process to avoid rework and to highlight the successes and shortcomings of earlier iterations.

2. Prototyping in DoD Acquisition

In today's military acquisition environment, we are moving away from so-called Stovepipe development. There is more interchange between the Combat Developer, who articulates requirements for the soldier, and the Science and Technology Community whose concern is Research and Development. In some cases, scientists develop and mature technology due to new requirements. In other cases, new technologies become available and for demonstration to Combat Developers; programs adapt as needs change and as technologies mature (**Figure 5**).

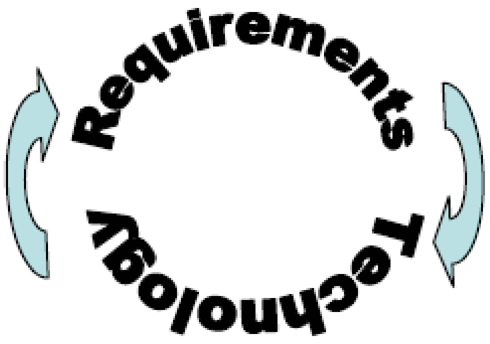


Figure 5. A Simplified DoD View of Technology / Requirements Interactive Push & Pull [From Ref. [27]]

The high-level acquisition process is a progression of activities moving from concept to delivered capability. **Figure 6** illustrates the acquisition process for the DoD, Joint Ground Robotics Enterprise [27]. Within the phases of this program are a myriad of sub-programs, as well as a complex mix of stakeholders. Requirements, the far left side of **Figure 6**, initiate in a variety of ways. Users may generate them, and they often do, but they may also come from new capabilities demonstrated by the science and technology community. Requirements may also be political; Congress has the power to declare that DoD pursue certain technologies. For instance, Congress has mandated that unmanned vehicles will compromise one third of Ground Combat Vehicle fleet by 2015 [28] .

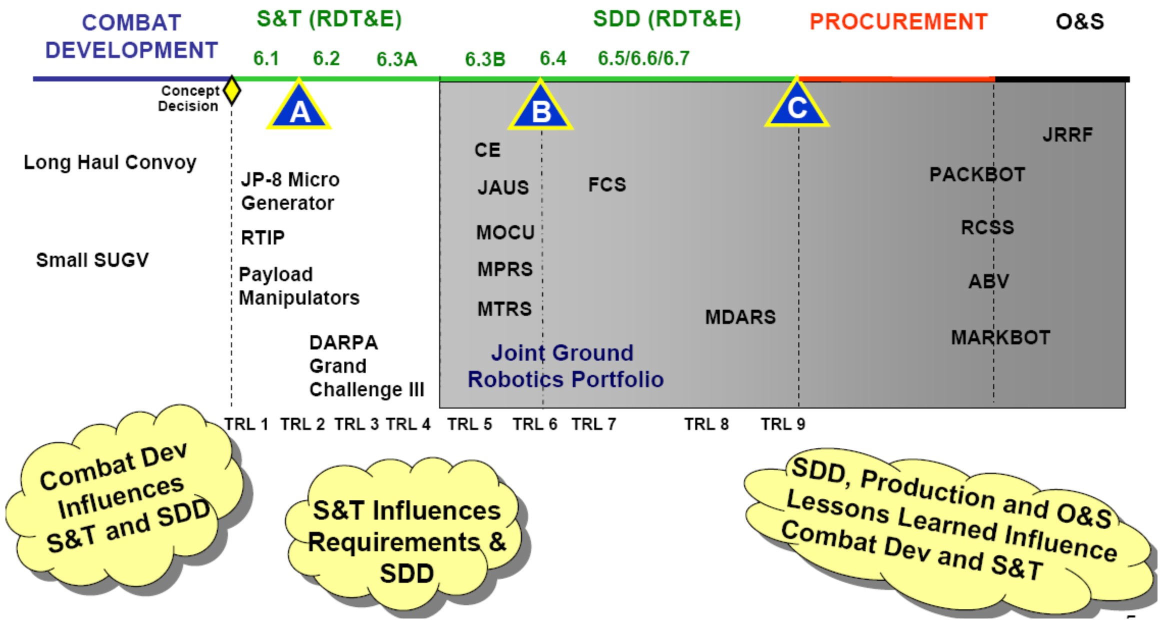


Figure 6. The DoD Joint Ground Robotics Enterprise operates within the DoD Acquisition Process [From Ref. [27]]

Successful robotic programs are mutable and responsive to rapidly changing technology. Prototypes are extremely useful to help bridge the gaps between requirements and technological limitations. All stakeholders need to understand the limitations and/or the possibly unexploited reach of technology. All stakeholders also need to understand the proposed solution in terms of requirements. Since there are many things researchers and end users need to understand, creating prototypes for development and operational tests that help them to understand the relationship between solutions and requirements is crucial.

Once a requirement is validated and accepted, DoD researchers and engineers begin to mature and develop designs. As they move up in Technology Readiness Levels (TRL), they begin to have prototypes that look more and more like useful objects. Particularly when working with complex systems, such as robotics, researchers may span several technology requirements, as well as several technology generations. A designer may use a mature technological mobility platform with a very immature, but promising, sensor technology. The prototype must be developed, presented and documented such that all stakeholders are aware of the aspect that developers are trying to understand; in the example case, the immature sensor. If not all stakeholders are aware of the purpose of the prototype, misunderstandings may hamper progress, (i.e., the sensor is on an operational robot, the sensor must be ready for the field, or, if the prototype system performs less than stellar, there must be something wrong with the mobility platform).

At various stages in the process, designers and researchers form an implementation team to create a prototype. This team may be part of the design team, or it may be an outside entity contracted for creating a prototype. Depending on the maturity of the technology and the phase of development, the prototype can range from simulation to full working models.

Remember, the purpose of creating a prototype is to gain an understanding through tests. If an implementation team creates an artifact that meets the need of the original idea, researchers can test it. Researchers conduct simple tests within the confines of the project lab. A dedicated test group often runs tests that are more complex. In this idealized set of scenarios, engineers and researchers capture ideas, make them real and then compare them back to the need.

B. PHASES OF A PROTOTYPE

An ideal prototype development proceeds as shown in **Figure 7**.

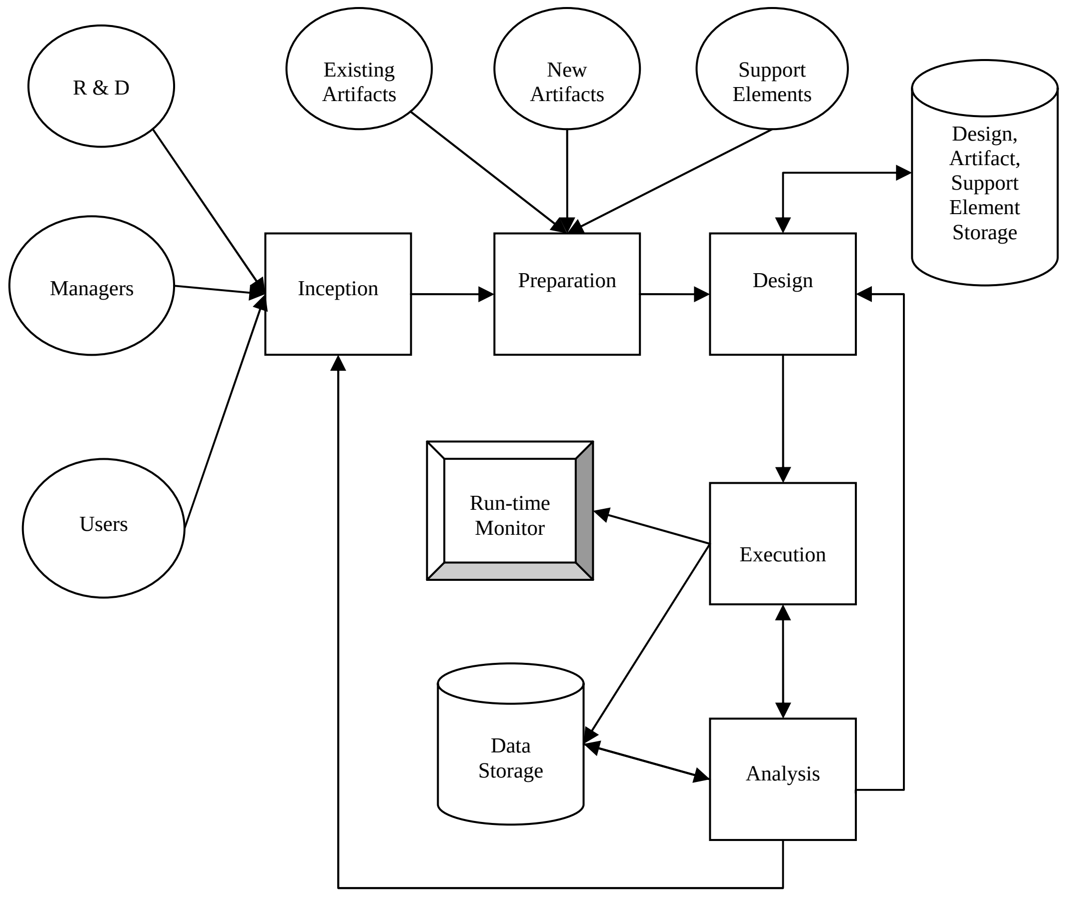


Figure 7. An Idealized Prototyping Process Contains Inputs, Processes and Storage Elements as Well as Feedback

1. Inception

This is the initiation of the project. End users and developers formulate needs and researchers conceive ideas of how to fulfill these needs, although the needs and ideas may come from a variety of sources. Technology developers may discover an application for an emerging technology or a new application for an existing technology. Users may articulate a need for their current operations. Managers may conceive a new or improved way of accomplishing their mission.

2. **Preparation**

The preparation phase initiates the process of creating a prototype. Prototype engineers collect items of interest heretofore known as artifacts. Rather than create the entire prototype by hand, engineers create the interfaces to the artifacts of interest and additional aspects needed to execute a prototype, most commonly instrumentation and communications. These small packages, possibly components in their own right, are stored for future use. The minimum set includes the code to translate the interface of the component of interest into a standard representation, and an interface to the world, usually a network interface. Along with the code, developers specify a set of documents specifying compilers, installation instructions and instructions on how to start the component if necessary.

3. Design

In the setup or assembly phase, the prototype is built from the elements created or installed in the preparation phase. Ideally, this is accomplished using a graphical user interface (GUI) to connect the components. The components cannot communicate directly, so each component is associated with a wrapper, customized during the assembly phase. After a graphical model is created, parameterized and error-checked, designers create a set of wrapper programs. The wrapper programs should have attached documentation needed for their deployment. The assembled prototype should be stored for future reference and configuration management.

4. Test

When the technicians operate the prototype, researchers capture and store various aspects of the system for future analysis. There is also the possibility at this point for them to monitor the run-time instrumentation in real time. By doing so, researchers can display various conditions to the test team as the prototype execution continues. The test team can monitor real-time constraints described in the instrumentation and gain an indication of how closely the system is operating in regards to timing bounds. They can also monitor network traffic and display the capacity of a particular link as a color change on the graph's edge. If test conditions exceed predefined bounds, or indicate a safety problem, the test team has the option to abort the test.

5. Analysis

Data collected during run-time is retrieved with the assembled prototype for analysis. In a spiral development effort, the results of the analysis feed back to the "Initiation Phase" as part of the next cycle.

C. CASE STUDY – AD HOC SPIRAL DEVELOPMENT OF THE OMNI-DIRECTIONAL INSPECTION SYSTEM (ODIS) ROBOT PROTOTYPE

1. Introduction

The following case study describes efforts to apply spiral development to a prototype robotic system. It details our efforts to respond quickly to rapidly changing requirements by preparing a series of functional electro-mechanical systems intended to leverage current R&D efforts and present a tool to management and user communities. It also documents where the shortfalls in current methodology are. In particular, US Army Tank Automotive Research, Development and Engineering Center (TARDEC) engineers needed to create an under-vehicle inspection system, which would be more effective than the existing manual inspection. In general, they needed to begin to develop methodology to improve their development process.

The case study will discuss TARDEC efforts with the ODIS robot in terms of the idealized prototyping process. As you will see, the high points of this case study are user interaction and feedback. The areas that need improvement are information transfer, particularly when the team is in flux.

2. Inception

As mentioned above, the ODIS Robot project began as a University Research Project. In the late 90's, The US Army Tank Automotive Research, Development and Engineering Center contracted with Utah State University to conduct research into small Omni-Directional Vehicles drives as part of its continuing research into novel mobility concepts. Three six-wheel electric vehicles were produced (ODV-T1 to ODV-T3) of varying sizes from 40 to 1500 pounds to demonstrate scalability. A four-wheel hybrid hydraulic version was also produced (ODV-T4). **Figure 8** is a picture of a collection of Omni-Directional vehicles at Utah State University.

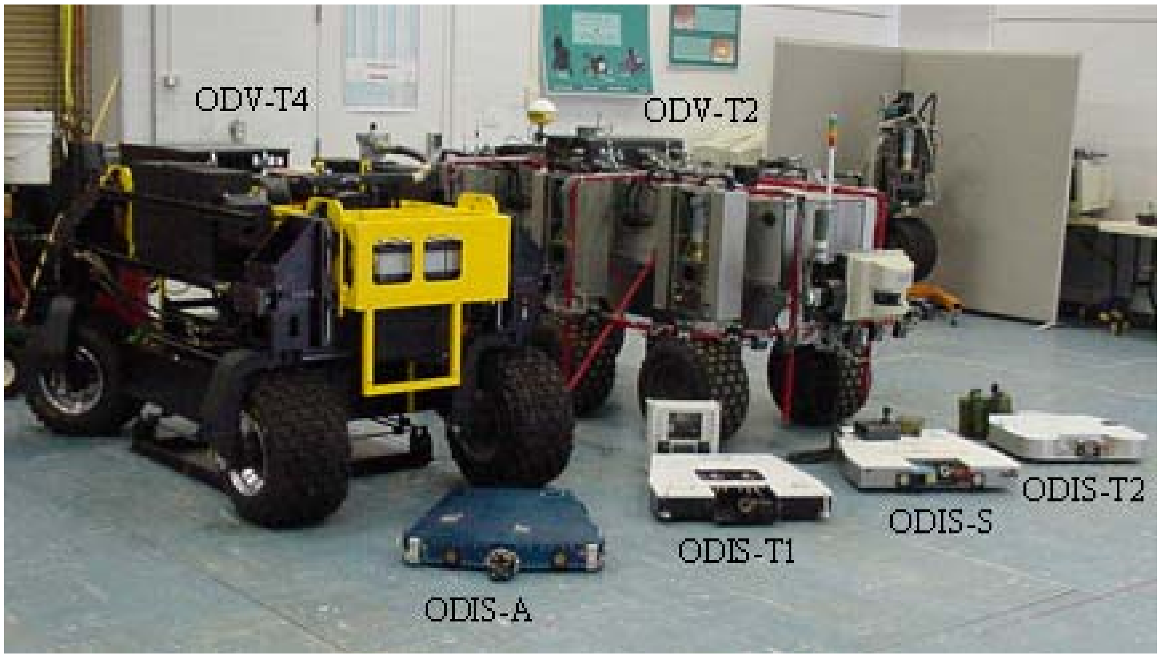


Figure 8. Shown are selected Utah State University Omni-Directional Vehicles Developed under TARDEC's Intelligent Mobility Program

The three-sub tasks of this research included control of multiple intelligent wheels for Omni-Directional Drive, autonomous path planning for Omni-Directional Vehicles, and autonomous path execution. The fourth vehicle produced was the original ODIS robot, ODIS-A (A for autonomous). The ODIS-A experiment converged the three lines of research into a practical application, under-vehicle inspection. Investigators chose under-vehicle inspection, not because there was a great need for this type of robot at the time, but because it was an interesting, constrained problem. Under-vehicle inspection was a task that addressable and capable of field demonstration without expensive field-testing. ODIS-A was completed in the spring of 2001.

As mentioned above, TARDEC's main purpose in this research was to explore mobility concepts for small ground robotic vehicles. Due to their small footprint, many known mobility metrics for heavier vehicles do not apply. The desire for autonomy in small robots, as well as large robots was also a given. This robot was demonstrated at a

Workshop on Future Unmanned Vehicles at Ft. Leonard Wood, MO on September 5, 2001. Representatives from the Army's Military Police, Engineering and Chemical Schools and Combat Developer's offices attended.

An interesting aside: while the robot impressed those in attendance, many missed the point. Many commented that the technology was cool, but how often do we inspect under-vehicles? Although this appeared to be a prototype "under-vehicle inspection" robot, it was in reality, a prototype for a unique ability, autonomous operation, on a unique mobility platform.

Seven days later, immediately after the events of 9/11, under-vehicle inspection became a priority. Somewhat serendipitously, researchers immediately realized they had a prototype that would fill a need.

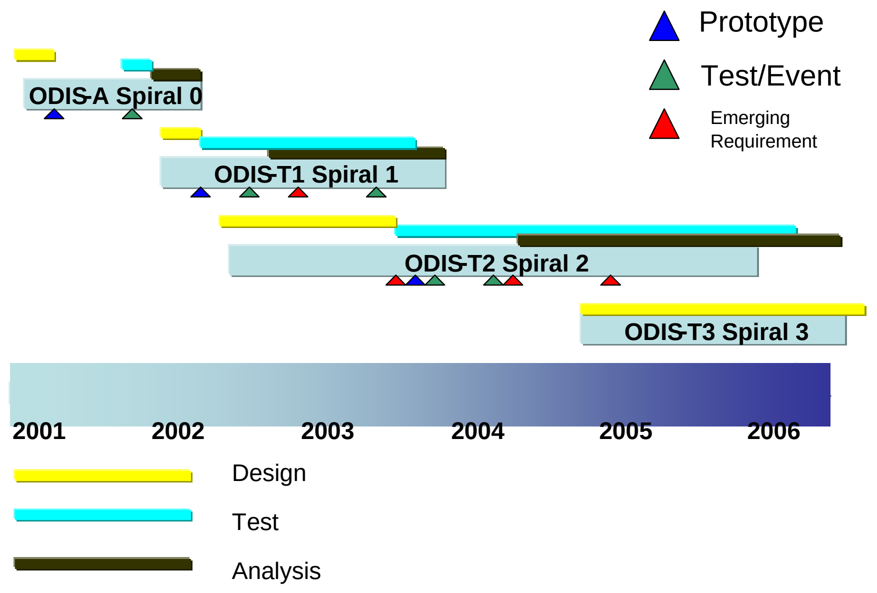


Figure 9. Project History from 2000 to 2006, ODIS Spiral Development Timeline

Figure 9 shows the timeline for ODIS prototype progression, and makes clear that the time span between prototypes is growing at an unacceptable rate, as is the funding expenditure. The ODIS development is not an isolated case. Additional case studies, similar to ODIS can be described for other ground robots and mission packages.

Funding issues contribute, but mostly due to rightful skepticism that this is not a simple task or that some proposed solutions are not fully fleshed-out. Researchers need tools to help them quickly develop lab prototypes of existing systems with new mission packages that mitigate risk. Once risk is understood and documented, decision-makers will provide funding for promising solutions.

3. Spiral 0

a. Preparation / Design - Spiral 0

Preparation was simple. The ODIS-A existed and could be evaluated by users almost immediately. The preparation consisted mainly of documenting the existing robot and submitting proposals to various emerging disaster agencies. Design was also almost trivial at this point. TARDEC researchers and designers knew, though, that this robot was a laboratory device, and further design would be necessary. From the onset of this effort, they decided that they would use a spiral development model. That would give them the best opportunity to leverage existing research funds against this need.

b. Test - Spiral 0

For the initial test, the ODIS-A was brought to the Detroit Arsenal and demonstrated to the Civilian Guards and Force Protection Team. Operations were filmed and documented for future use in briefings to DoD and Homeland Defense Managers. The goals of this brief test were to:

– Better understand requirements and decide on a path forward.
– Marketing (even Military R&D organizations need to compete for funding).

c. Analysis – Spiral 0

The purpose of this limited test was to help make a decision to go forward with the ODIS project. There were a number of attributes of ODIS-A that the analysis team expected the users and managers to criticize and they did. Both the robot and Operator Control Unit were LINUX-based, with cumbersome and time-consuming boot procedures. ODIS-A was a laboratory robot, so it was not water resistant, and not very rugged. Battery life was also an issue. The analysis team had to be careful to explain these limitations to prevent early dismissal of the concept.

The surprising part was that the guards did not want to operate it, except for short trial runs. They articulated they were afraid to break it. The other surprise was they wanted nothing to do with autonomous operation. They preferred when someone was tele-operating the robot; they said it gave a better inspection and they liked having the control.

4. Spiral 1

a. Design – Spiral 1

The positive feedback the team received from their original investigation led to an almost immediate effort to begin an iteration of the ODIS robot, designated ODIS-T1, that addressed new requirements and needs, as well as a better understanding of the original requirements. In particular, the next spiral was to reduce complexity, improve reliability, drive cost down and increase user acceptance.

TARDEC was fortunate to have a modifiable open contract with Utah State University. Thus, they retained the majority of the key developers from the ODIS-A project. This greatly simplified their ability to create three ODIS-T1 robots rapidly.

Because of the original tests, researchers defined some key elements. Microcontroller operating systems replaced the LINUX operating systems, reducing boot time from minutes to seconds. Eliminating autonomous functions removing costly sensors. This also improved reliability, since the autonomous functions were still not fully defined and the question lingered as to how well the autonomous functions would work in the real world. The OCU was redesigned into a self-contained unit with an integrated video monitor. The camera was redesigned to make it water resistant.

The team reused much of the original low-level control software for the wheel nodes from the ODIS-A spiral. The team maintained the approximate profile of the physical design, including the wheel positions, and designed a new battery that would allow for almost instantaneous battery replacement. An engineer from the Army Lab was assigned to work in Logan, Utah with the Utah State Design team for six months.

b. Test – Spiral 1

Three ODIS-T1 robots were hand-built at the Center for Self-Organizing Intelligent Systems at Utah State University. The first was delivered on March 1, 2002 and unveiled at the Society of Automotive Engineers World Congress in Detroit, MI that week. The reason for unveiling at this Exposition was primarily marketing. Many high-level Army officers attend this conference, as well as high-level Army civilian managers.

Following the March event, the team continued marketing and producing an awareness of this robotic capability by showing it at a number of Military and Security technical conferences. ODIS-T1 was also used intermittently at the TACOM main gate.

In August of 2002, researchers conducted a Limited Objective Experiment at Fort Leonard Wood, MO. This test was designed to exercise the ODIS-T1 robot in a controlled environment with real-world users. The test coordinators selected ODIS-T1 robot operators from National Guard Military Police activated for Force Protection at Fort Leonard Wood and by civilian gate guards from Fort Leonard Wood. The test scenarios evaluated the robot inspection against the proverbial “Mirror-on-a-stick.” Tests were conducted during the day and night, in August sun and in Thunderstorms. Test scenarios for ODIS included Ad-Hoc Checkpoints (operators outside using the portable OCU) and fixed checkpoints (operators in a climate-controlled environment with the video piped to a 19” television). Many of the vehicles inspected were rigged with simulated small explosive devices.

In July of 2003, the team also conducted a demonstration/experiment at the Ports of Los Angeles and Long Beach (POLA/POLB) in cooperation with the California Highway Patrol, Coast Guard, Port Security, TSA and California State University. The goals of this experiment were to use the prototypes to discover new requirements and to assess training requirements as well as assess inspection time. The primary operators were California State University students with no previous robotics experience and no previous under-vehicle inspection experience. This experiment placed no simulated hazards. The students were asked to point out suspicious items and were supervised by TARDEC Engineers and California Highway Patrol Officers.

c. Analysis – Spiral 1

The purposes of the evaluations and experiments the team conducted were to improve the ODIS robot by soliciting feedback from the users. Most users were very excited about ODIS. It was a tool to keep them out of harms way while allowing them to do their job even better. During the tests at Ft. Leonard Wood, ODIS performed better than the mirror on a stick in all categories. At night in the rain, it was markedly better. ODIS used from a climate controlled guard shack performed better than ODIS used outside with the OCU hung from the inspector by straps. Sun glare on the video screen was an issue outside.

ODIS-T1 originally only supported a visual sensor. With minor modification, engineers added a thermal imager in parallel to the visual camera. The thermal imager selection was a switch on the OCU. Testers did not use the thermal imager at Ft. Leonard Wood; the soldiers' mission did not rank it as a very desirable item. At the POLA/POLB event, the California Highway Patrol (CHP) officers were very impressed by the thermal imaging. The CHP primary mission is truck safety; they saw the thermal imager as a great tool to allow them to inspect truck brakes for proper operation. Homeland Defense officials were also interested in the thermal imager as a tool to quickly detect false tail pipes and to assess the time a vehicle had been parked by looking at heated elements under the vehicle.

Potential end users also requested other sensors. A radiation detector was evaluated by placing a standard Total Level Detector (TLD) on an ODIS robot. The TLD was set to beep at three times the background radiation level. Soldiers were able to find a Cesium source hidden in a vehicle 100% of the time with this simple sensor. Users also asked for an explosives detector; however, at the time of these experiments, there were no compact non-contact sensors available.

Mid-level management brought up the issue of power. The team learned that there was a strong desire not to put a new battery and a new charging system in the field. During the POLA/POLB experiment, researchers also had the CHP evaluate a new portable OCU built into a vest. The officers were very adamant that this was the way to go, since the majority of their operations are at Ad-Hoc Checkpoints. They also made some suggestions that greatly improved vest OCU ergonomics.

5. Spiral 2

Spiral 2 began in June of 2003 at a Joint Robotics Program meeting. The research team briefed the status of ODIS-T1 to the group. Also during this time, the Iraq war was under way and the most dangerous job soldiers had in Iraq (according to the news reports) was vehicle inspection at Traffic Control Points (TCP); two soldiers lost their lives to snipers at the manned TCPs that week. The Rapid Equipping Force asked TARDEC researchers if they would like to participate in a field experiment and send several ODIS robots to Iraq and Afghanistan.

a. Design – Spiral 2

Design for Spiral 2 consisted primarily of incorporating lessons learned from Spiral 1 into the new prototype, as well as making it more robust and survivable. At this time, Utah State University sub-contracted with Kuchera Defense Systems (KDS) in Winbur, Pennsylvania, for mechanical fabrication and assembly. In particular, the clunky ODIS-T1 OCU box was replaced with a vest-mounted OCU, the hand-fabricated sheet metal body was replaced with a hull machined from billet aluminum, and standard military batteries replaced the hand-built robot batteries.

Even with these clear requirements, the first article was delivered in August with a camcorder battery for the OCU and a head mounted display. The team knew from experience with other programs that head mounted displays may have important uses, but not for this application. The head mounted display makes it difficult, if not impossible to share information that is projected on the display. It is also unwieldy for individuals wearing glasses and helmets.   An OCU rework task was initiated with very specific guidance.

There was very little software rework, other than porting the master node software from an Onset TT8 microcontroller to a Phytec MPC555 Microcontroller. The MPC555 was substituted to allow additional functions as time went on. Although the ODIS-T2 was to be assembled at the KDS facility, many of the original developers were still available at Utah State University, and much of the original work was done by the developers at Utah State University The contract with USU was winding down, and Kuchera Defense licensed the ODIS technology rights from Utah State University. With several USU employees temporarily working now for Kuchera Defense, Kuchera Defense delivered the first 10 ODIS robots in January 2004.

b. Test - Spiral 2

A first limited test for the spiral 2, ODIS-T2, robots was the so-called “Beltway Sniper Trial.” Lee Malvo’s trial was held in a courthouse with an underground parking structure. The Chesapeake, VA police wanted no surprises. They asked to use an ODIS robot to conduct their under-vehicle inspections. The ODIS-T2 robots performed as expected and the test generated no new requirements.

The next ODIS-T2 robots were completed for a limited fielding to soldiers on the ground in Iraq, Afghanistan and other CENTCOM elements. Ten robots were taken to Iraq and Afghanistan in February 2004. There are now additional ODIS-T2 robots in theater.

The CENTCOM testing generated several new requirements. Maintenance of the robots became an issue. The team had not made user maintenance or configuration available. ODIS-T2 has very low ground clearance; it could not be used to inspect trucks parked off road. A requirement for a mast to raise the camera so that it could look inside the vehicle cab and trunk will allow all soldiers to stay clear of the vehicle until they have a first look from standoff. Researchers had developed a small TNT detector to the point where it could be used on a robot.

The experiment continues to this day, with additional requests for ODIS robots, particularly with the camera mast, regularly coming from soldiers in the field.

c. Analysis – Spiral 2

None of the new requirements could be completely satisfied with the ODIS-T2 robot. The maintenance issue and higher ground clearance were interrelated. Researchers recognized that redesign of the physical robot was necessary, to produce modular wheel nodes to allow ease of maintenance and mission-specific wheel sets. They initiated work on a “camera mast” mission package that would raise the camera from 4” to as much as 12’; but the team could not access any control lines to maneuver the mast. The FIDO TNT detector also required remote control and data return; however, the software was again not up to the task without major work.

The team ported the software from the ODIS-T1 prototype to the new ODIS-T2 prototype using a seat-of-the-pants effort. Very little documentation remains. The development environment was an expensive package, but since the majority of the work was done at Utah State University, they used a student version. In addition, custom libraries and in-line assembler codes were issues. The contract with Utah State University expired and their expertise was no longer available. TARDEC researchers did not have an open contract with Kuchera Defense Systems that they could draw on. With an extensive in-house effort, the team reverse-engineered the ODIS-T2 serial command packets, but neither the robot nor the OCU had available hardware lines to utilize. Both the ODIS-T2 robot and OCU became black boxes.

6. Spiral 3

Spiral 3 was initiated by establishing a contract with Kuchera Defense Systems to create yet another more robust system that could address requirements discovered in Spiral 2, as well as to add flexibility and standard messaging to address yet unknown requirements.

a. Design - Spiral 3

The ODIS-T3 was to be another prototype that would be used in contingency operations. ODIS-T3 was not necessarily a full-production version, but one that would shake out final issues and lead to a production version.

Requirements for the spiral 3, ODIS-T3 design included:

- Design for manufacturability.
- Design for maintainability.
- Compliance with the Joint Architecture for Unmanned Systems (JAUS) standard.
- Robust interfaces.
- Modular wheel nodes with low and high clearance versions.
- Support for arms and masts.

The design team had some members that worked with the former Utah State University Team, but the USU team was completely gone from the contractor’s facility by the time ODIS-T3 design officially started. Two of the requirements, design for manufacturability and design for maintainability, entailed a complete re-design of the wheel nodes Redesign of the wheel nodes forced researchers to do a complete rewrite of the low-level wheel node control software. In addition, JAUS-compatibility requirements forced them to redesign the top-level control software.

Due to the lack of acceptable prototyping software, and the lack of intermediate information documents from the early part of the program, the contractor design team attempted to develop the software in an ad-hoc method; this lead to a failure to produce an acceptable design and schedule slip.

D. SUMMARY AND DISCUSSION

The above case study should make it clear how important prototype systems are in quickly developing tools that improve as time goes on and have the ability to save lives now. When the ODIS-T2 robots were first considered for experimental deployment in current conflicts, there were a handful of Unmanned Ground Vehicles in theater. Most were tele-operated mine-clearing vehicles and some scout robots for cave exploration. By the time the first ODIS-T2 robots were delivered six months later, there were about 200 small robots in theater, most being delivered to Explosive Ordinance Disposal (EOD) teams to defeat Improvised Explosive Devices in Iraq. Users were still skeptical. At this writing, there are about 2000 small robots in theater, again most delivered to EOD teams, who, today, consider them a necessary tool. Every day, acceptance levels increase, but with that acceptance comes a price. As Unmanned Ground Vehicles penetrate operations, soldiers in the field come to depend on them. If they fail, the soldier on the ground cannot perform his or her job effectively, at least if they are not willing to face the risk from reverting to non-robotic Tactics, Techniques and Procedures (TTP).

This puts the onus back on developers. Army research and development organizations are looked to develop new tools and capabilities for existing and emerging platforms quickly. The developers need to assemble concept vehicles or existing vehicles with emerging concept mission packages, rapidly test and demonstrate them in a benign environment and then harden the vehicles and mission packages for actual use. All this must occur in a period measured in months not years.

As noted above, spiral development has great potential to bring partial solutions to the field quickly. The initial ODIS spiral development was heavily driven by hardware. Hardware designers have at their disposal a wealth of tools and techniques to move them forward quickly. CAD systems let them see high-level views of the configuration well before they cut metal. CAM and rapid prototyping systems, such as new 3-D printers, allow them to refine fit and rapidly change some of the parts built.

Software developers, on the other hand, do not yet have the wealth of tools that hardware designers enjoy. Software prototyping tools remain in the domain of high-end software houses, mainly due to their cost, complexity and lack of acceptance. Software prototyping tools are needed to address modeling and transfer of the models to executable code that can be used in conjunction with hardware prototyping tools. Today’s electro-mechanical systems, such as robotic systems and unattended sensors are highly dependent on software. Designers, developers and testers need tools to allow them to rapidly configure new systems, or modify current systems.

A goal of this effort is to produce a working model for such efforts. TARDEC researchers want to investigate the potential for a set of tools that will allow rapid development of robotic systems, much like CAD tools allow mechanical designers to pass their models to machinists that produce the hardware without having the machinists develop a full understanding of the engineering involved in the design.

V. PROTOTYPING ENVIRONMENT REQUIREMENTS

A. INTRODUCTION

To make the prototyping process efficient, tools and guidelines are necessary. In most cases, the artifacts researchers begin with are black boxes, a combination of Commercial-off-the-Shelf (COTS) hardware and software. Hardware items may include mobility platforms, Operator Control Units, sensor packages and controls. Time and money are of the essence, as researchers and engineers cannot always afford to go back to the original developer, and usually do not have the time to contract to do so when they can afford it. Thus, they need an approach that will allow them to use these black boxes, as well as simulations of the black boxes, in a hardware-in-the-loop environment. Researchers and engineers need to be able to create these prototypes in government labs, or using third-party support contracts. Auxiliary hardware can be placed at the interfaces to the black boxes to translate known inputs and outputs into messages understandable to the prototyping environment.

This research propose a model-driven approach, applied very early in the process, to gain an understanding of requirements, sort out the promising solutions from the snake oil and demonstrate outcomes to senior managers. A model-driven approach is desirable, since models facilitate generated software and automate additional aspects of development such as testing, and debugging.

1. Preparation

The preparation phase is where researchers work with their software experts to create well-formed, well-documented components necessary for assembly of software to integrate artifacts. Each artifact, which might be a mobility platform, Operator control Unit, sensor package, or control, is considered a black box, and researchers expect to have no knowledge of the internal software or structure of the artifact. All that is available is some interface. It may have documentation, or it may have to be reverse-engineered.

Researchers and engineers also know that they want the artifacts to communicate. The original communications may be unsuitable for the current task, however, or the engineers might be experimenting with new communications strategies.

Lastly, researchers want to understand or control aspects of the messages in a data flow. They might want to count messages of a certain type, modify messages to insert faults or throttle the data flow.

Software experts create the interfaces and components necessary for wrapping the black box as components. This includes parsing messages originating from the network or destined for the network and translating the data structures to the legacy format understandable to the artifact. They complete the instrumentation and other optional aspects.

They create communication component sets, be they for simple TCP/IP or complex mesh networks. The completed set of components from the wrapped black box to the output of the communication component forms a node, or functional element of the prototype, capable of sending messages, receiving messages or both (**Figure 10**).

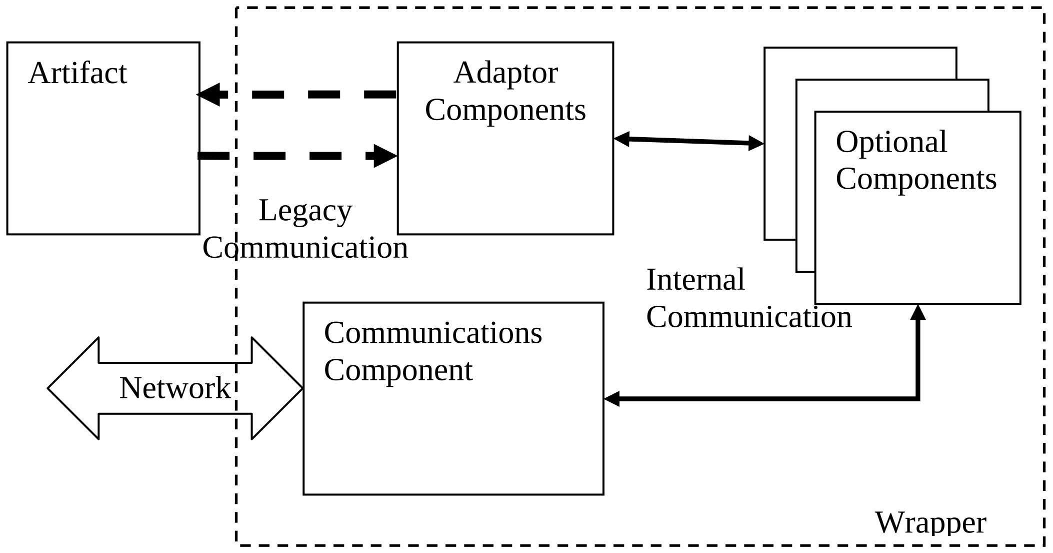


Figure 10. Prototyping Environment Node Components Block Diagram Showing the Relationship between the Network, Wrapper Components and a Legacy Artifact

a. Artifact Wrappers

Creating a wrapper is usually not a trivial task. In many cases, TARDEC robotics designers and researchers are faced with a robot or sensor that can only be accessed via a serial radio link. The serial link may communicate with proprietary software applications and they want to access it to create an input into a new control algorithm or integrate it into an existing display. If they are lucky, they have source code available. Often they do not; logic analyzers must be set up to capture and decode the bit stream. Creating an interface to a module with only a serial output may take several weeks of effort, and may include creating wiring harnesses as well as code.

The engineer charged with this task must first sort through the code and documentation (if available) to discover the format of the serial packets. An example is the ODIS robot. It communicates with its Operator Control Unit (OCU) via a serial port connected to a radio modem. Source code was available, but not completely up to date. The engineers tasked with communication with the robot via a laptop computer spent considerable time and effort to decode and document the command packets. They documented the packet structure using an Excel spreadsheet. (**Figure 11**). The spreadsheet has no external documentation; it sits as a stand-alone file in a project directory on a server, but only because a researcher put it there; it formerly resided on one of the engineers' workstation. It is understandable, only because that researcher can walk up to the engineer that created it and ask him about it. Is this poor software engineering practice? Absolutely. Nevertheless, it is common practice in prototyping environments. Neither of the engineers working on this project were software engineers; one was an electronics engineer and the other a computer hardware engineer. The team has no dedicated software engineer on staff.

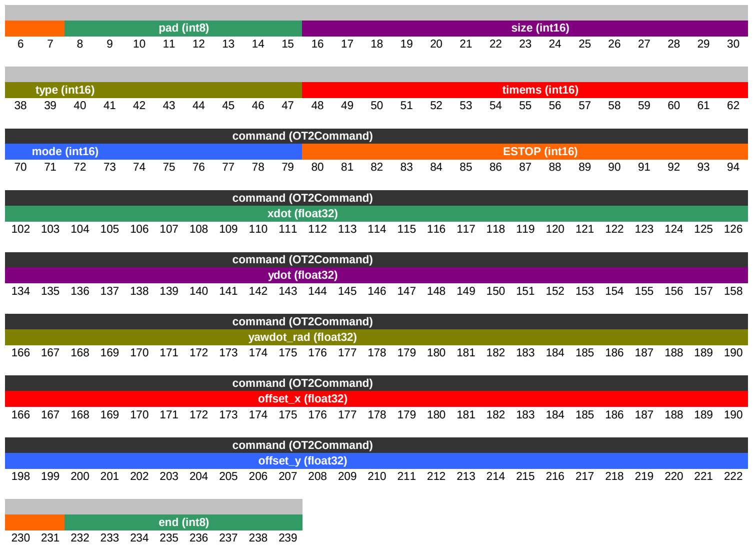


Figure 11. ODIS Serial Command Packet, Binary Field Description

Field #	Name	Type	Units	Interpretation
1	Presence Vector	Unsigned Short	N/A	See mapping table that follows.
2	Propulsive Linear Effort X	Short Integer	Percent	Scaled Integer Lower Limit = -100 Upper Limit = 100
3	Propulsive Linear Effort Y			
4	Propulsive Linear Effort Z			
5	Propulsive Rotational Effort X			
6	Propulsive Rotational Effort Y			
7	Propulsive Rotational Effort Z			
8	Resistive Linear Effort X	Byte	Percent	Scaled Integer Lower Limit = 0 Upper Limit = 100
9	Resistive Linear Effort Y			
10	Resistive Linear Effort Z			
11	Resistive Rotational Effort X			
12	Resistive Rotational Effort Y			
13	Resistive Rotational Effort Z			

Figure 12. JAUS Message 405h Set Wrench Effort (specification)

Once the serial packet formats are understood, then a message that matches the packet data must be chosen. For the case of this effort, an ODIS command packet example, a JAUS wrench command was chosen (**Figure 12**). The second of the two engineers led work on this side of the effort. The resulting work resided on the second engineer's workstation. The result was the same as above. Although the results were acceptable within the parameters of the assignment, the work left much to be desired from a software reuse standpoint. The ultimate result of this effort should have been a reusable artifact wrapper; instead, the researchers had isolated code that was used as a temporary measure to experiment with a single aspect of the prototype development.

Obviously, no one wants to loose this work, but it often happens. This research provides support in the preparation phase for wrapper creation, as well as documentation. Guidelines, probably in the form of design patterns are necessary to reduce wrapper construction effort as well as increase understandability for those using the wrapped artifact in design. Additional support is necessary to collect and make available descriptions of the artifact capabilities, documentation about the team that created the wrapper, any physical wiring diagrams necessary to connect the artifact to any necessary auxiliary hardware, revision history, etc.

To provide universal reuse, another part of the puzzle is to associate a set of XML data structures to describe messages between artifacts, where the artifacts are recognized as black boxes. That is, researchers do not know, or pretend that they do not know anything about the internal workings of the artifact. In the previous example, the data structure translation was ad hoc. To add instrumentation or change communications strategy requires significant rework; rework that must be done every time they want to use any part of the wrapper for a new prototype.

XML data structures are very important to allow automation of component configuration in the design phase of the prototype environment, as well as forming machine-readable messages for execution and analysis phases. Ideally under these conditions, a set of XML data structures will be associated with each translator component, one for each of the message types supported, and data generated during the translation will be passed through the wrapper in XML format. The latest version of JAUS will be defined in XML format; proper use of XML will allow the prototyping environment to keep pace with the changeable nature of standards. As the standards mature, different versions of the message set can be applied. Design time configuration of components can be automated with dependency on XML message sets. This allows for component software reuse as the standard evolves. The environment could even be used with other message sets, as long as they are defined in XML.

b. Communications Components

In addition to the wrapper, a communication component must also be created to interface the node to the prototyping environment. Of course, multiple components are necessary, even in a homogeneous communication since the data flow is directional; one each (translation and communication) component for read and write. The prototyping environment must support insertion of alternative communications strategies. In some cases, the prototype nodes will communicate using standard TCP/IP, but there also could be communications via wired busses, mesh networks or even combinations.

c. Optional Components

Other components are necessary to make the prototype useful as a test subject. These components include logging and/or and monitor instrumentation, temporal logic, data throttle, fault insertion, security and encryption or other aspects are just a few that are necessary and useful for prototype evaluation. Programming elements, such as counters, loops and conditional gates are also necessary. Once again, design patterns are necessary to reduce component construction effort as well as increase understandability for those using the aspect in their design. Again, additional support is necessary to collect and make available descriptions of the aspect capabilities, documentation about the team that created the wrapper, and application examples.

2. Design Phase

a. Top Level Design

In the design phase, the prototyping environment is turned over to a domain expert to construct a prototype system of interest. The engineer runs a Graphical User Interface (GUI) to create a dataflow model of a distributed system (**Figure 13**).

Each node represents a separate computing element. When a node is opened, the screen changes and a node-programming environment is displayed.

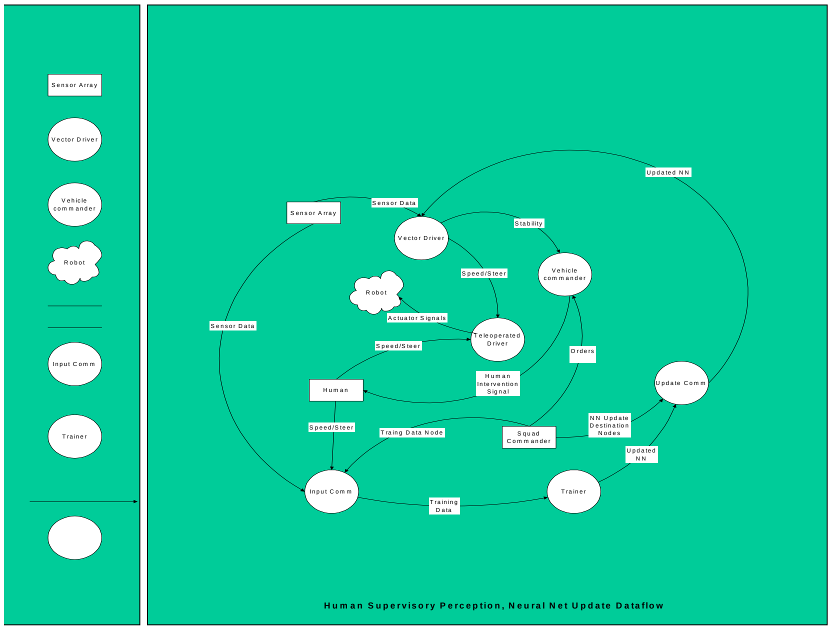


Figure 13. Prototyping Environment, Conceptual Model for the Graphical User Interface

b. Node Level Design

Like the main programming environment, the node programming screen will have components in one window and a work area to the another In addition a list of allowed messages are displayed. If a message is checked, the environment will read the XML file for the message and display the data items in the checked messages. These will be used with context-sensitive menus to configure components that are capable of making decisions based on message parameters.

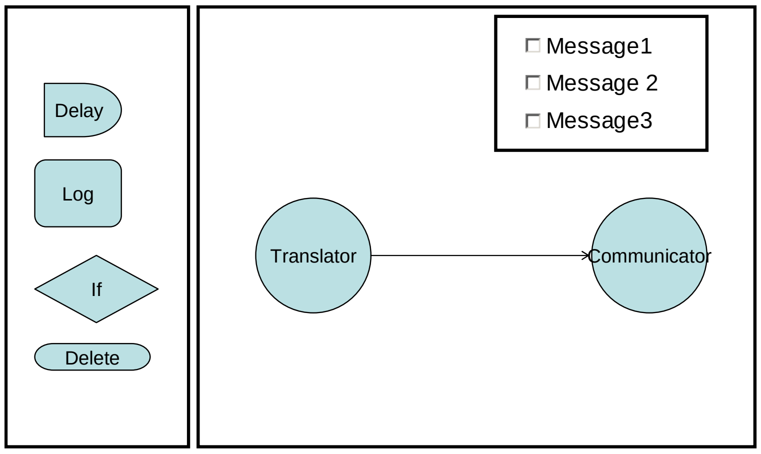


Figure 14. Node Programmer, Initial Condition

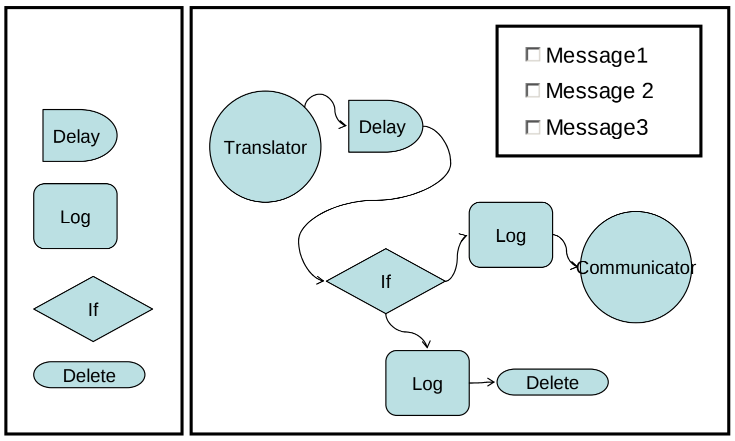


Figure 15. Node Programmer, Final Condition

The node-programming screen should start up with artifact wrapper and communicator component (**Figure 14**). The user can accept this as it is, and complete any communicator setup, or the user can add additional aspects to the node (**Figure 15**). Since researchers may be dealing with real-time systems or limited speed processors, timing is a concern. As they add additional aspect components, they slow down the system. Researchers have to live with some overhead, and some of the overhead may be made up for by using faster communications than they expect in the completed system. In other cases, they may have to limit the aspects they install in the nodes and possibly create two instances of a prototype with different aspects in each.

Additional standard utilities are necessary such as storing the prototype at any time, retrieving a previous prototype, notes and cut/paste.

At this point, everything needed is in place to generate wrappers and glue code for each node.

c. Code Generation

Given a model-based approach, there is, in the background, a Meta-model that is amenable to code generation and composition. At a minimum, code for each node, along with compilation and deployment instructions should be generated. A desirable element would be to pipe the generated code into appropriate cross-compilers and create executable codes. A final (and not far-fetched) item would be support to distribute the finished product to the appropriate execution elements.

3. Test

At the conclusion of the design phase, all the elements of the prototype are created and a final executable distributed system is available, along with instructions on how to operate it (turn on individual nodes, dependencies, etc.). Now researchers can put the prototype through its paces. A simple example is operating a chemical detector on a small robot. The design team may be interested in how well the detector responds under various environmental conditions. They might want to determine if the power assumptions are valid, or to watch the message traffic under design time installed bandwidth constraints; the team will definitely want to store any data collected for future analysis.

Researchers' observations may be on a macro level; e.g., can robot X satisfy the requirement with sensor Y? If so, does it exceed expectations, or narrowly meet requirements? Alternately, observations may be on a more microscopic level. A research team may want to examine temporal relationships. It is one thing to calculate bandwidth, quite another to be operating concurrently with other unknown elements in the field. For micro-observations, researchers and engineers may want the design data flow to project visual information to them, communications lines that change color depending on available bandwidth or color-coded nodes that are communicating or not.

4. Analysis

Successful work depends on a number of factors. Data collected at test time should be linked to the design time diagram. Standard analysis components should be available, as well as capability and support for creating new analysis components. Analysis results should be comparable to other configurations' analyses and stored in a form compatible with the prototyping environment.

5. Summary – Prototyping Environment Requirements

This prototyping environment relies on software wrappers that encapsulate legacy artifacts in a consistent manner. Separation by time and physical location of the Software engineers and programmers mandate wrapper creation support and guidelines. Ultimately, a searchable data repository for, components, models, rule sets, and test results is needed as in any modern engineering effort. A set of focused Graphical User Environments enhances model and instrumentation understanding. Engineers create Prototypes to interact with an aspect of the system under test; instrumentation and monitor components are necessary.

VI. PROTOTYPING SYSTEM AND RESULTING PROTOTYPE ARCHITECTURES

A. INTRODUCTION - ARCHITECTURE DISCUSSION

The Software Engineering Institute Website at Carnegie Mellon [29] has many definitions of Software Architecture, one of which is prominently displayed at the top of their Software Architecture Definition Web Page and is attributed to software architect Eoin Woods [30]:

"Software architecture is the set of design decisions which, if made incorrectly, may cause your project to be cancelled."

In TARDEC researchers' experience, projects are rarely cancelled at the appropriate time. Many drag on for years, consuming resources and involving unsuspecting scientists and engineers as the original team dissolves or flees. Eventually, either someone declares success, or a re-organization occurs, and the project quietly ceases to exist.

D'Souzaand and Wills, in their book, Objects Components and Frameworks with UML [31], provide another definition of software architecture particularly appropriate for this work:

"The set of design decisions about any system that keeps its implementers and maintainers from exercising needless creativity."

This is appropriate, because they state exactly what researchers need to promote software reuse and remove the software creation burden from non-software engineers tasked with creating operational prototypes. That is, this somewhat tongue-in-cheek definition helps guide us to a separation of concerns, a set of guidelines for composing the reusable components, and a disciplined approach to developing sets of potential prototype solutions. In essence, engineers would eventually like to remove all creativity from the creation of the final software to create the prototypes. The end-users creativity belongs at a higher level of abstraction. This effort will show the utility of transferring the end-users creativity into a constrained, graphical environment that leads to composed and/or generated code.

A discussion of software architecture is important to this dissertation because it sets the stage for what is to follow. In the "Handbook of Software and Knowledge Engineering," Rick Kazman discusses three reasons why software architecture is important [32]. These are Communication, Early Design Decisions and Transferable Abstraction (a characteristic of an architectural model that make it useful in similar systems). A common element of the definition of architecture, from the physical domain to software is that architectural documents are high-level abstractions that describe a high-level design of the system. Note that the word "abstractions" is plural. There can be many views of the system. In the physical world of cities and buildings, people often talk of the "style" of a particular architect that refers to the external view. Observers intuitively understand that, at lower levels, the architect or more often, the team of architects selects from a variety of architectures for interior design and building services, such as elevators, plumbing and electrical. The same holds true for software; even a radically new top-level architecture may reuse a common database architectural view (**Figure 16**) if it needs database services, or contains an internal database.

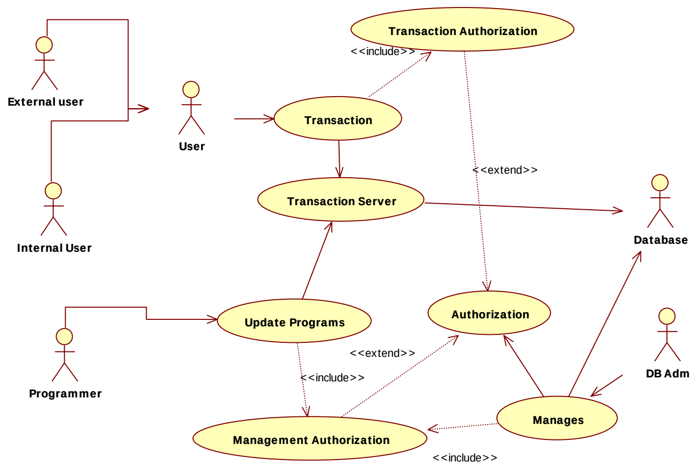


Figure 16. UML Use Case Diagram for Top Level Database Architecture [From Ref. [32]]

B. ARCHITECTURAL ANALYSIS IS NECESSARY AT MULTIPLE LEVELS

As mentioned above, System Architecture is a high-level abstraction. It is a reflection of system requirements and provides a high-level view of the implementation. Since the system may consist of multiple features, the top-level architecture may explicitly force a particular lower level architecture, or it may allow flexibility.

Figure 17 below presents the Use Case diagram for the top-level architecture of this effort. Recall that primary motivations for architecture representation are Communication, Early Design Decisions and Transferable Abstraction.

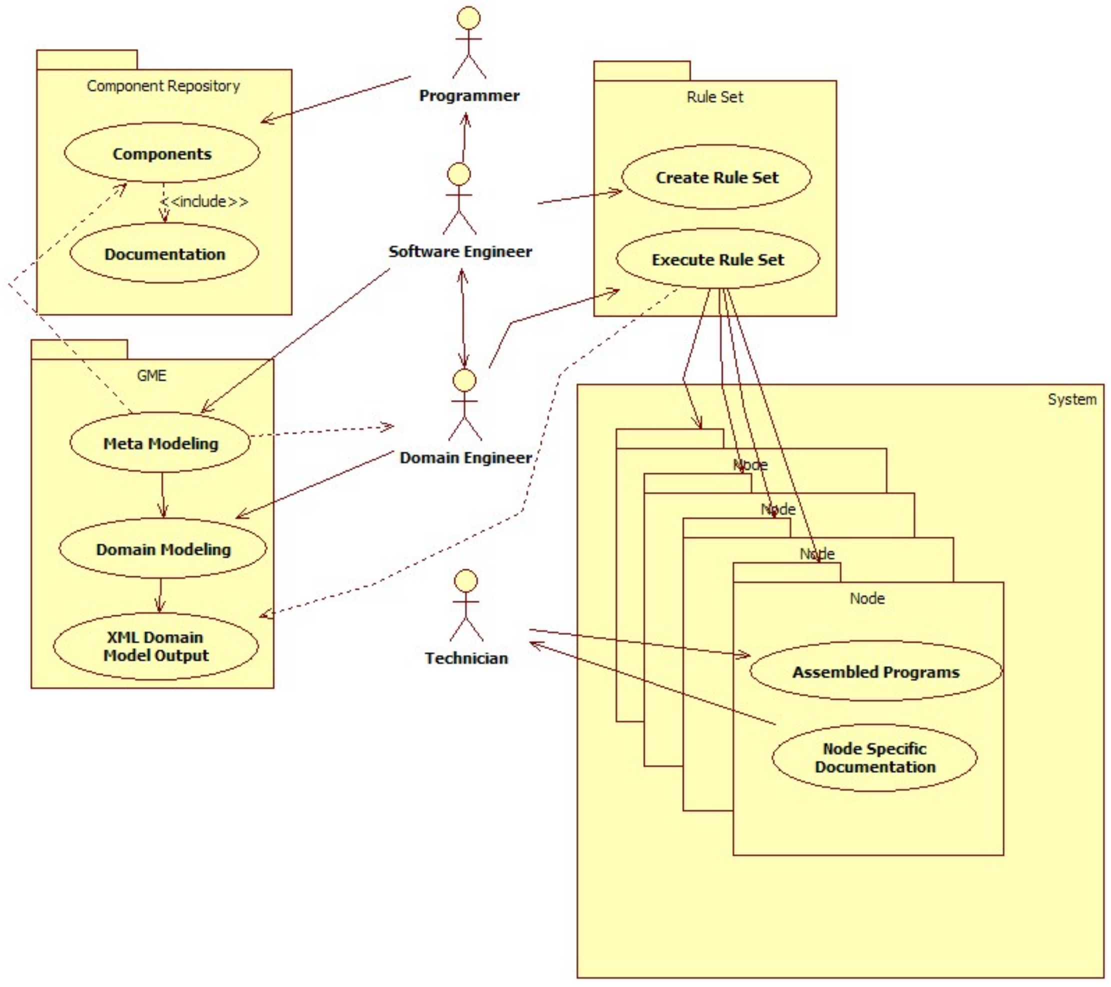


Figure 17. Use Case Diagram for Top Level Prototyping Environment Architecture

1. Communications

The first motivation for creating architectural representations is communication. For this very reason, the top-level drawing is simple and abstract. UML representations are growing in favor, but the architect needs to create a representation understandable to a general audience to achieve buy in; if the managers and other non-software types have difficulty understanding the notation, they potentially loose interest. Therefore, simple and abstract is better.

A single diagram is not usually able to communicate all information. The UML has a collection of diagram types. A Use Case diagram shows high-level relationships between activities and actors, the functionality the system. Other views are necessary to convey additional information. Sequence Charts provide a logical view. They show the functionality inside the system and dynamic behavior.

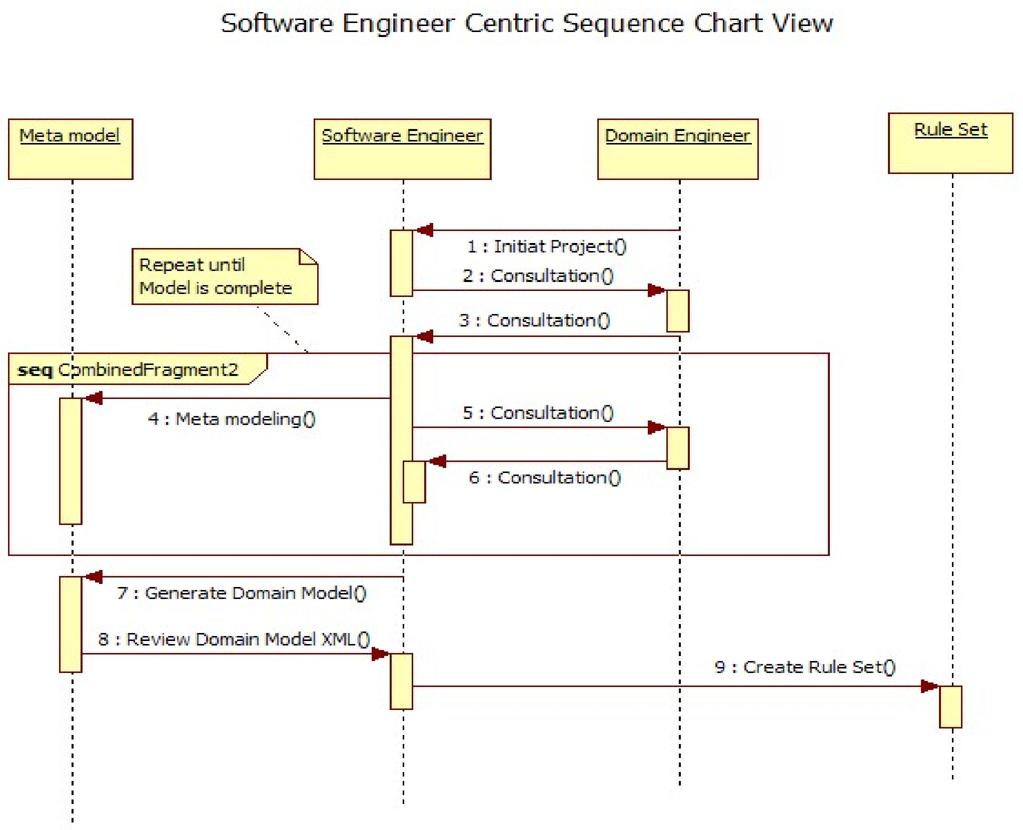


Figure 18. UML Top Level Sequence Chart View for Software Engineer

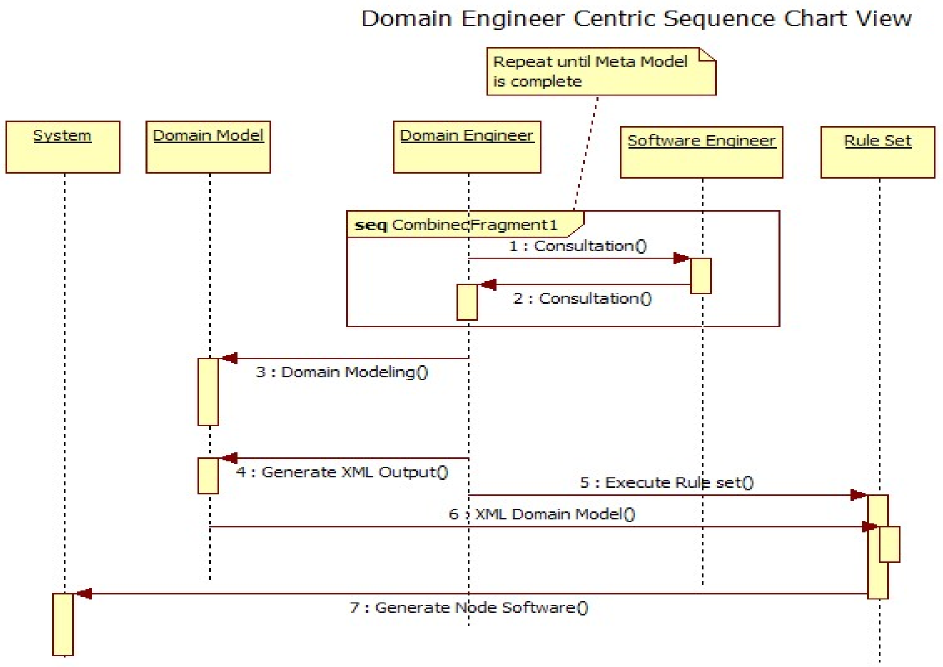


Figure 19. UML Top Level Sequence Chart View for Domain Engineer

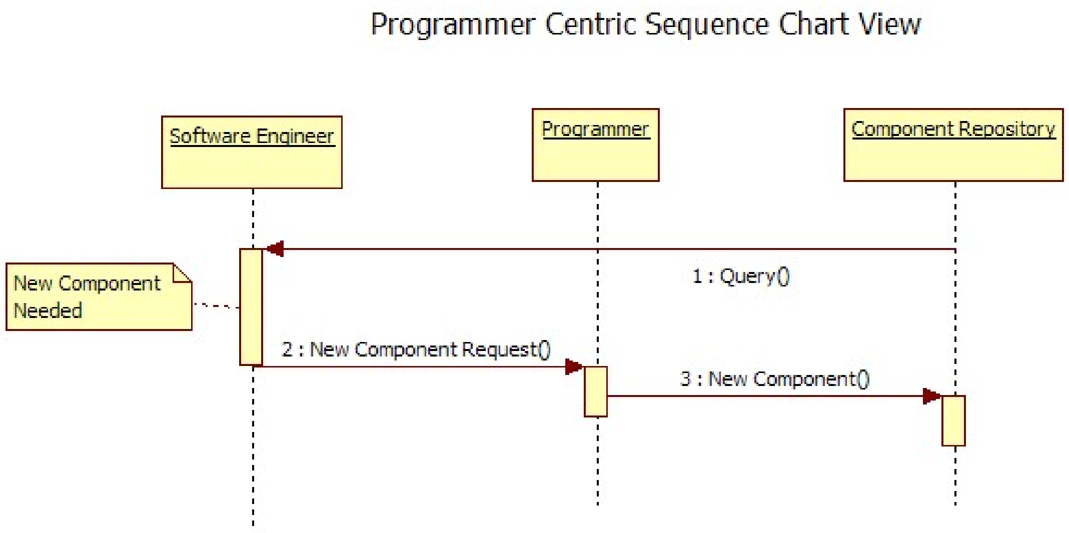


Figure 20. UML Top Level Sequence Chart View for Programmer

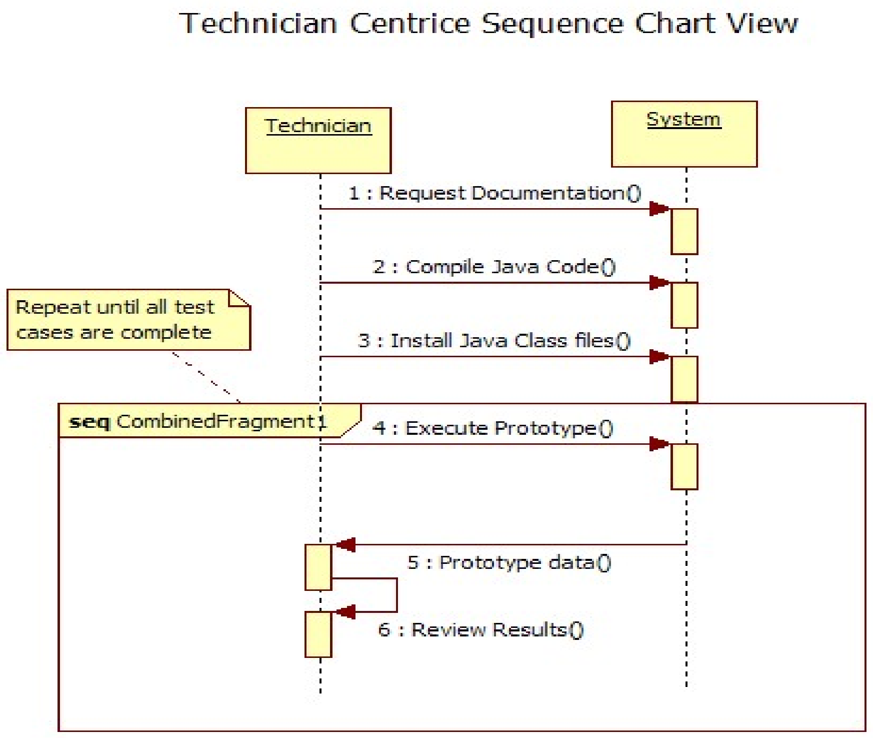

Technician Centrice Sequence Chart View
Technician
System
1 : Request Documentation()
2 : Compile Java Code()
Repeat until all test cases are complete
3 : Install Java Class files()
seq CombinedFragment1
4 : Execute Prototype()
5 : Prototype data()
6 : Review Results()

Figure 21. UML Top Level Sequence Chart View for Technician

Figure 18-Figure 21 add dynamic information to the Use Case diagram for each of the four principle actors. Early communication of the required interaction may help managers understand scheduling and staffing concerns.

2. Early Design Decisions

The second motivation for creating architectural representations involves early design decisions. According to M. Simos, et al. [33], the list of architectural styles for software is very short:

- Generic Architecture – Fixed frame with sockets that allow alternative and extension components. Fixed Topology and fixed interfaces.
- Highly Flexible Architecture – Supports variations on its topology can yield a particular generic architecture.

At the domain level, designers want a generic architecture. This type of architecture is amenable to a variety of positive attributes for a domain-specific language. Generic Architecture:

- Is a candidate for automation: well understood;
- Has defined constraints;
- Is repeatable; and
- Is easily versioned.

At the design level, designers want flexibility. Well-trained, highly skilled software engineers (possibly with a robotics background) must be able to express their creativity and produce the domain models for use in a more structured environment. Lacking Software engineers with a robotics background, management must make commitments for robotics engineers' time for collaboration with the software engineers.

3. Transferable Abstraction

The third motivation of architectural description is transferable abstraction. A transferable abstraction describes an architectural model that is useful for current purposes and reusable in whole or in part. This is a common in the world of physical structures. For example, there are many different realizations of colonial architecture for homes and offices. All have the same abstract design, and may share common sub abstractions, such as interior stairs placement, attic space and foundation properties, but the size, interiors and building services are different. The same goes for software architecture. Being able to transfer architectural abstractions sets the stage for potential design and implementation reuse. In the future, researchers might see this architecture used for other distributed systems, for future implementations of production robotic software or for an implementation of a "Service Oriented Architecture" for robotic truck fleets.

Returning to **Figure 17** we find a Use case diagram representation of the proposed system. As you can see, it is a simple diagram of high-level abstractions. There is little concrete information about implementation. The abstractions indicate a collection of flexible sub-architectures, the realm of the software engineer.

These include :

- A Meta-modeling environment, Meta-modeling is the realm of the software engineer in consultation with the domain engineer;
- A domain modeling environment, domain-specific modeling is the realm of the robotics engineer;
- A rule set for encapsulating software engineering knowledge for use by the domain engineer; and
- The product, the realm of the user, and a tool for the technical evaluator of the robotic system in this case.

a. Component Repository

Staring in the upper left corner, there is a block labeled "Component Repository." This block represents a storage abstraction. It can be as formal as the database depicted in **Figure 16**, or it can be a simple set of files in a folder[*]. As the project matures, we expect the former, but there may be some new storage paradigm implemented in the future.

Accessibility is a function of the availability of the components. Components are subject to the owning entity's access policies. If that entity is a public web site authorized users of the website can access the component. On the other extreme, access may be via a private file system, as in the case of the experimental implementation. This limits sharing to "chunks" of release components. The ultimate solution is a database with variable access.

[*] This research refers to the file system implementation of the component repository as the "codebase" during experimental implementation.

b. Users

In the middle of **Figure 17**, the actor icons indicate primary human interaction with the system. There are no exact limitations, but we notice a progression of responsibility from the software engineer and programmer at the top to the technician at the bottom.

In reality, there will be a cyclic interaction among the actors in any prototyping exercise. In fact, a failure to achieve feedback indicates that the process has broken down and is in need of stimulation.

c. Foundation

The underlying and implied foundation to any engineering effort is basic tools and guidelines. These may include editors, XML parsers, design patterns and text files describing requirements and standards. If any of these goes beyond commonly accepted or understood practices, then an explicit description must be included as, at minimum, an annotation to the architecture.

d. Components

Put most simply, components are chunks of code. Ideally, they would be self-contained and independent of other concerns. This ideal definition leads to the potential for code bloat. Consider a mechanical system where a gear is a component. A gear requires a shaft. Imagine the consequences if a vendor were required to include every possible shaft length with the gear. Similarly, components in this effort use messages. Rather than supplying every potential message strategy, the components rely on being supplied with a message object, itself a component of the system.

The upper left hand block of **Figure 17** represents reusable components. These components are stored in the component repository when completed. These components compose the final codes for each node in the prototype. The components are constructed using design patterns referred to in the bottom block. Different types of components will use different design patterns. The design patterns are necessary to insure that the interfaces are of the proper type at composition time. This part of the architecture is not only in the realm of the software engineer, but also indicates a temporal abstraction. The guidelines and components are a prerequisite to the blocks above. This relationship is not a hard one; alternate guidelines can exist in parallel but a single set of guidelines will apply to a single Meta-model. In order for the Meta-model to work, it needs to "know" what components will be available to the final composition. The component repository grows with time, but Meta-models will need to be versioned to take advantage of additional components.

e. Meta-model

The lower left block up represents the GME modeling environment, which begins with a Meta-model. Meta-models define architecture. Software engineers create Meta-models with prior knowledge of the domain or in collaboration with domain experts. Meta-models are the key to domain architecture. The Meta-model encapsulates high-level information about the system. The Meta-model defines components, component relationships and constraints. It is a vehicle to encapsulate software engineering knowledge and facilitate transfer of this knowledge to domain experts. A complete Meta-model is a prerequisite to domain modeling.

f. Domain Model

Within the GME block of the architecture is the domain-specific modeling tool. The environment generates this tool from a Meta-model. Concrete models of the system under construction may be instantiated from the domain-specific modeling environment workspace. There may be one or many domain models created from a single Meta-model. Icons specific to the domain under examination are part of the domain-model generation process. This enables domain engineers to create system models for a variety of scenarios. For instance, the Meta-model may include a communications element. The domain modeler may choose from a variety of concrete communications components like serial, TCP/IP or CAN communication components represented by unique descriptive icons. The domain modeler is also able to select the individual node artifacts that will participate in the system from a collection of high-level domain-specific artifacts. As an example, a mobility platform might be a stylized representation of a mechanical man.

g. Putting it All Together

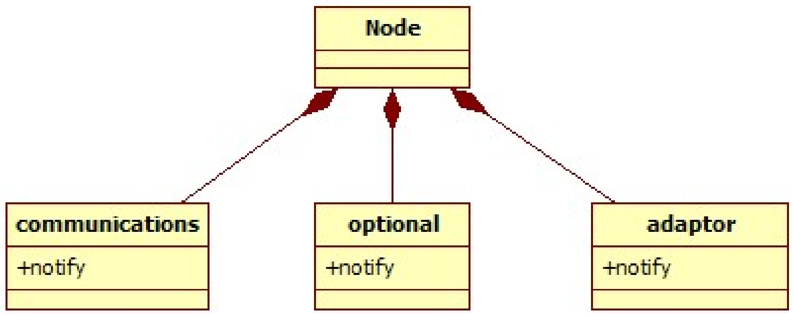
Moving up to the fifth level, Composition / Generation allows the domain expert to create the software elements necessary, again without having to know the software engineering details necessary to accomplish this task. Again, the goal is to separate concerns. This architecture allows the experts to do what they know. Software engineers do not need to learn intricate details of the realization of the system; domain engineers do not need to know the details of the software engineering needed to provide them with this tool. “Architects” create models of buildings, structural engineers flesh out the design depending on location, customer and environmental factors. Tradesmen build the building.

Above the fifth block is yet another dashed line - another separation of responsibility and another temporal break point. Transitioning across this line is not possible until the domain model is completed and the system assembled. It also is the transition from design to a reification of the system. Above the line is the product of interest, software, and documentation necessary for aggregate prototype operation.

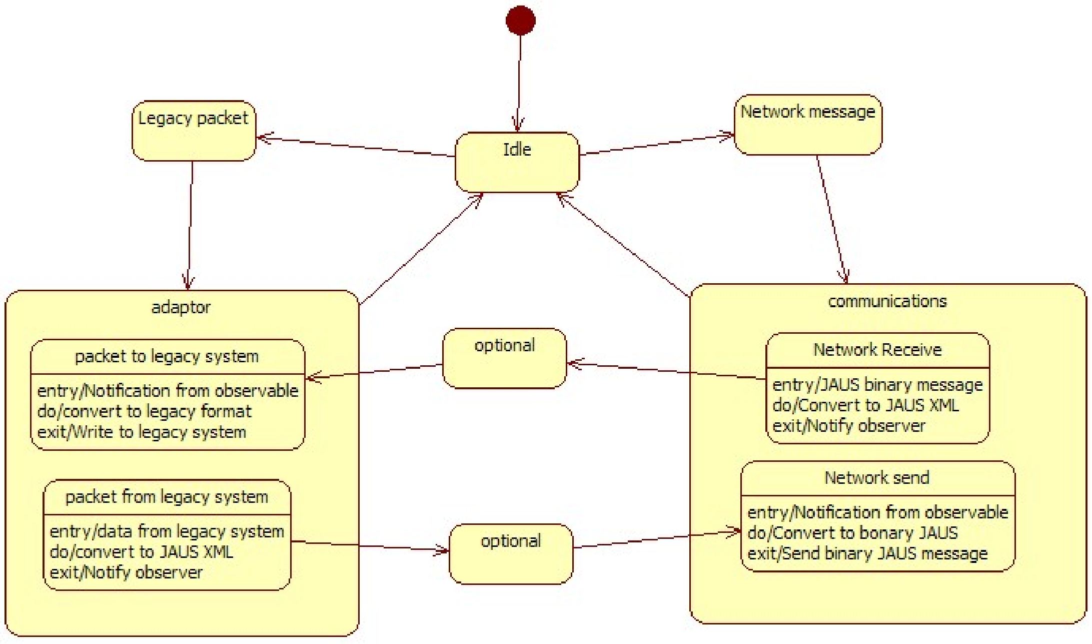
h. Nodes

The top pair of blocks represents the assembled programs and documentation. Codes for the individual nodes are available in the assembled programs block. The aggregate of the node codes is the system artifact. In a prototyping environment, the artifact runs through its paces in a variety of mission scenarios. The final artifact may be a simulation, a hardware-in-the-loop simulation or a pre-production hardware prototype. The block labeled assembled program is an abstraction of a collection of software programs to run a distributed system. In the case of a robotic prototyping system, each node has a run-time architecture associated with it (**Figure 22**).

A later section, describes a design for a robotic prototyping system based on this target run-time architecture.



Class Diagram


Legacy packet
Idle
Network message
adaptor
packet to legacy system
entry/Notification from observable
do/convert to legacy format
exit/Write to legacy system
packet from legacy system
entry/data from legacy system
do/convert to JAUS XML
exit/Notify observer
optional
optional
communications
Network Receive
entry/JAUS binary message
do/Convert to JAUS XML
exit/Notify observer
Network send
entry/Notification from observable
do/Convert to bonary JAUS
exit/Send binary JAUS message

State Chart Diagram

Figure 22. UML Views of the Robotic Prototyping System Node Run-Time Architecture

C. SUMMARY - ARCHITECTURE

The architecture of a system imposes discipline. It defines available elements, how those elements relate to each other and constraints. Architectural views are used to examine system characteristics early in the life-cycle and make changes while change is inexpensive. Later in the life-cycle, architecture views guide maintainers.

Architectures are abstract. They can be used to create similar systems, or can be examined to find the successful aspects of a system to apply to a new system.

Architectures have several levels. This effort first defines a modeling architecture. The modeling architecture allows the software engineers to create Meta-models, which are architectural descriptions of a domain specific modeling environment. Domain engineers, then, create instances of the domain architecture, using the modeling architecture.

THIS PAGE INTENTIONALLY LEFT BLANK

VII. DESIGN OF PROTOTYPING ENVIRONMENT

A. DESIGN ASSUMPTIONS

This effort begins with design assumptions based on experiences gained over the last several years by TARDEC engineers and researchers. The first assumption is that they have a collection of artifacts that they are interested in integrating. These artifacts may include Operator Control Units, Platforms (robots or unattended sensors), mission sensors, proprioceptive sensors, control algorithms, or mission packages (arms, masts etc.), just to name a few. In most cases, these disparate artifacts do not conform to messaging standards, and in most cases, researchers do not have access to the embedded processors to include additional code. Indeed, in most cases, they do not have access to the code, or access to proprietary compilers needed to modify the code. In essence, they want to integrate a collection of black boxes. They have some knowledge of the physical I/O, but they often have to ferret out the software data structures needed to communicate, either from code or by inspection of the run-time communications. In the worst case, they will run all integration software on auxiliary processors.

Creating interfaces to these artifacts can be an expensive and time-consuming effort; effort software engineers would like to retain and reuse. They would also like to make this information and knowledge usable by non-software experts.

The second assumption is that in many government robotics labs, software engineers are a scarce commodity. That is not to say there are no software “people;” these are talented individuals, but they are usually mechanical, computer or electronics engineers with a few programming courses, and are not trained in the intricacies of modern software design and development paradigms.

In summary, the worst and often typical case is that a research team needs to integrate a collection of artifacts that they can only access via external interfaces. The engineers and scientists are usually robotic specialists with a smattering of software knowledge; experienced software engineers and experiences robotics engineers with intensive software engineering experience are in short supply.

The task then is to develop guidelines, methods and tools to:

– Capture Software Engineering Expertise.
– Transfer this knowledge to Domain Engineers.
– Capture software elements for reuse.
– Capture configuration and execution data.
– Provide tools to simplify the integration process.

B. MODEL-DRIVEN ENGINEERING

This design is based on Model-Driven Engineering, a relatively new software development paradigm. Product line development for distributed embedded systems, such as aerospace and automotive, has become extremely complex. Developers spend years mastering platform APIs and usage; even still, they often only come to a complete understanding of a subset of the platforms they develop for regularly. Model-driven engineering focuses on abstractions particular to the application problem space and expresses designs in terms of concepts from that space [34].

Model-driven Engineering combines software components constructed to conform to specific design patterns with Domain-specific Languages. These languages are described in a Meta-model, often graphical, that defines the relationships of abstractions in the domain. Engineers create the Meta-models in UM, the language of the software engineer, and transform them into a constrained design environment, usually using graphical icons that pictorially describe the abstractions in terms understandable by domain engineers.

The domain engineers then create concrete instances of the Meta-model using icons that represent components available for composition of the final product. From the completed design, program generators are able to assemble components and create glue code to allow them to work together.

C. STANDARDS & TOOLS

In many research applications, particularly in the early phases, standards often take a back seat. Engineers are encouraged to think outside the box or standards do not yet exist. For this effort, however, several standards are of the up most importance.

1. Extensible Markup Language (XML)

XML standards are important at several levels. XML representations are both machine- and human-readable. XML representations of models within a tool suite facilitate transitions between different phases of model development and allow the use of automated tools. XML representations of models allow archiving and storage in a neutral format. Finally, XML representation of models is often a prerequisite for transfer between different tools, or between tools running on different operating systems.

Engineers often refer to XML data sets as self-describing data. This feature makes possible run-time configuration of instrumentation. The instrumentation parses a run-time data stream, containing a set of messages, and processes only messages of interest. The user changes configuration of the instrumentation at run-time based on message descriptions in a schema or directly from the messages.

2. Messaging

The Joint Architecture for Unmanned Systems (JAUS) [35], transitioning to an SAE standard, provides a common messaging framework for this effort. JAUS messages provide a path forward to a production system. An overview of JAUS is included in 0.

Different domains may use different message sets. Larger unmanned aerial vehicles frequently use the North Atlantic Treaty Organization’s (NATO) unmanned aerial vehicle (UAV) interoperability standard, Standardization Agreement (STANAG) 4586 [36]. Military war-gaming uses Institute of Electrical and Electronics Engineers (IEEE) Standard 1278, Standard for Distributed Interactive Simulation (DIS) [37].

Common messaging is important to this research since it reduces the level of effort required to bring legacy and experimental artifacts into the prototyping process. Consider a collection of N artifacts, each with a unique communication protocol.

Introducing artifact N + 1 requires potentially creating N new protocol adaptors. On the other hand, using a common message set requires creating one protocol adaptor for each new legacy item (**Figure 23**). Federations of artifacts communicate using a common message set. Wrapping each artifact with an adaptor to the common message set allows it to participate in the federate.

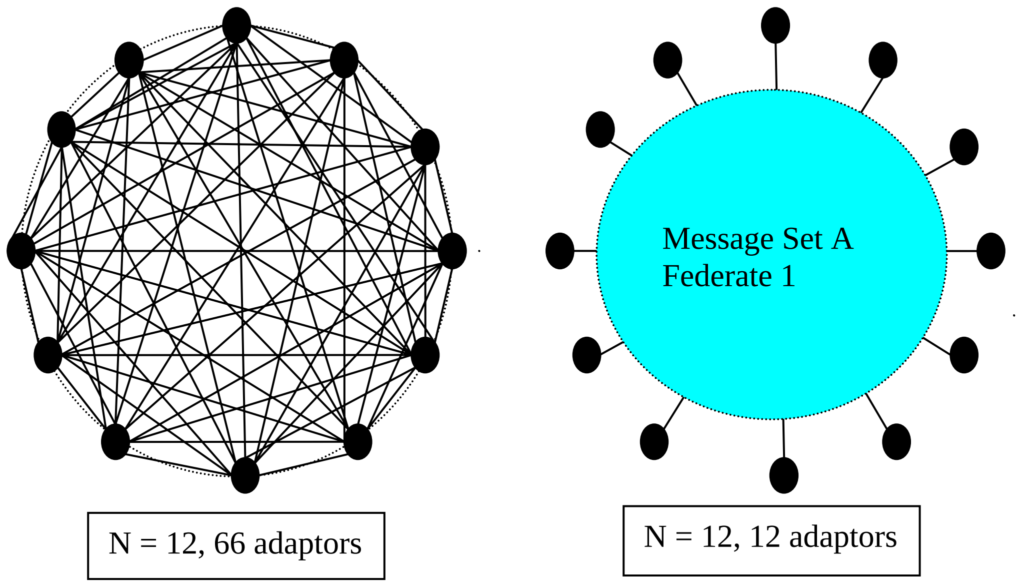


Figure 23. Visualization of the Number of Protocol Adaptors For Ad-Hoc Vs. Common Messaging Scenarios

Different domains may use different message sets. Larger unmanned aerial vehicles frequently use the North Atlantic Treaty Organization's (NATO) unmanned aerial vehicle (UAV) interoperability standard, Standardization Agreement (STANAG) 4586 [36]. Military war-gaming uses Institute of Electrical and Electronics Engineers (IEEE) Standard 1278, Standard for Distributed Interactive Simulation (DIS) [37]. Conceivably, JAUS-capable robotic system prototype experiments will interface with existing federates of artifacts communicating with either of these protocols. In this case, engineers create bridge software to allow an entire federate to participate in the others' domain as if the federate was a single artifact with many facets (**Figure 24**).

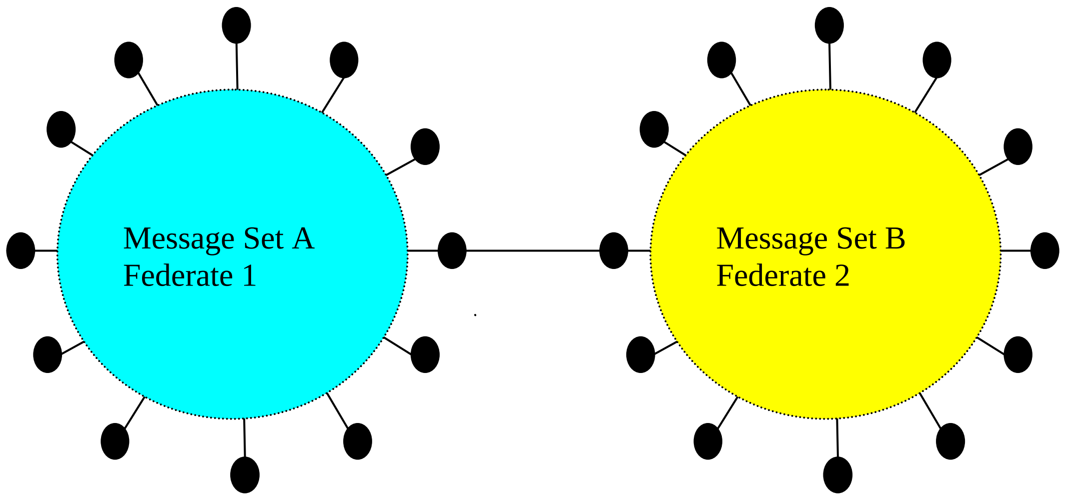


Figure 24. Connecting Two Federations Requires Only One Communications Adaptor

Messages in any distributed system, by necessity, will morph between formats. A system not designed for extensibility contains messages that have two distinct realizations, a binary wire format for speed and an internal format understandable to the target program. This is the case in many proprietary systems, as well as the current definition of a JAUS based system.

The XML enters the picture, first, as a tool to describe messages. In the current implementation of JAUS, XML documents describe the messages; however, an official JAUS XML schema does not yet exist. The logical next step is to create an official JAUS XML schema. This schema is a tool to generate and operate on intermediate XML representations of the JAUS messages. Intermediate XML representations take advantage of XML aware instrumentation tools [38].

Tools can provide:

- Run-time monitors.
- Logging.
- Fault insertion.
- Temporal logic checking.
- Operator Alerts.
- Run-time operational modification.

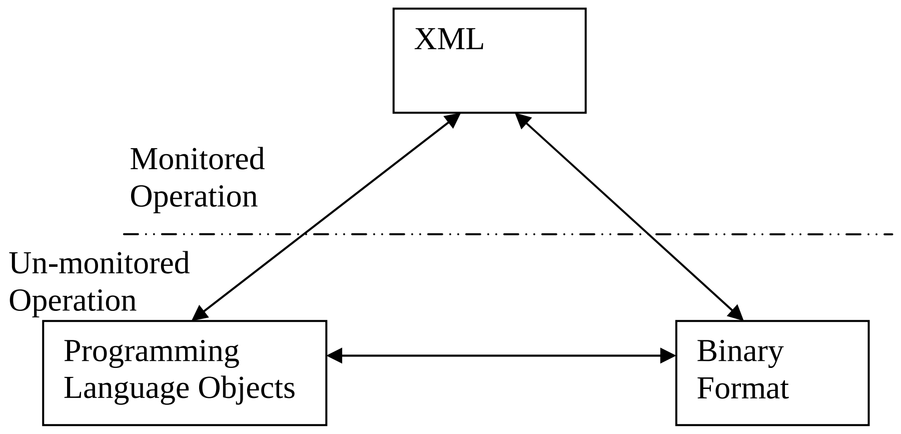


Figure 25. Translating Messages across Three Formats [After Ref. [38]]

Figure 25 depicts the possible formats within a single paradigm. Binary is the choice for on-the-wire format, due to compactness and speed. Programming Language objects are necessary to communicate within an application, and finally XML formats are desirable when a system needs to expect the unexpected. Many current operations do not require the additional translations to and from XML. This is mainly because current scenarios are not readily extensible, or require vendor support for extension. Adding the alternative XML translations opens up a world of opportunities. Not only can developers monitor current activities, but also they can make use of the XML tool set for translation, tool access and readable logs.

Although various arbitrary vehicles in and entering the inventory, do not all play by JAUS messaging, the XML dimension opens the door to a common data model. In his 2006 dissertation, D. Davis [39], concludes, "…there is enough commonality between various vehicles to enable implementation of a single data model suitable for the representation of arbitrary vehicle tasking, messaging and mission results and that XML Schema provides a suitable mechanism for formal definition of this data model." This research recognizes the importance of XML Schema in automating transitions between message formats. Further efforts with JAUS and translations between standards will examine the ability of existing and new XML Schema documents to create translations between protocols.

D. COMPONENTS

Components are the key to software reuse. Software engineers create a collection of components. These components later become a selection of model elements selectable by domain engineers.

Clemens Szyperski writes, "All components exist in a flat universe. This is an important property, as it allows servicing of components without having to know all places where that component has been used [40]." This indicates that components should support a consistent interface and contract.

Both Szyperski, in "Component Software" [41]and Czarnecki and Eisenecker, in "Generative Programming: Methods, Tools and Applications [42]" agree that a component:

– Is a unit of independent deployment; and
– Has no externally observable state.

Szyperski contends that a component is a unit of third party composition, while Czarnecki and Eisenecker relax this requirement. This dissertation work agrees with Czarnecki and Eisenecker, as long as the first two requirements hold. Components may be created internally or externally. Components are simple building blocks combinable in is as many ways as possible.

For the purposes of this design, there are three general classes of components:

- The endpoints, the individual hardware artifacts or simulation artifacts, along with wrappers (software), that at a minimum provides a mechanism to allow artifacts to be connected, are components for our purposes.
- An arbitrary number of optional components to instrument the prototype, induce disturbances, simulate communications protocols, throttle communications speed and/or provide translations to name a few.
- Communications components that connect the nodes to the system. These may be very simple components such as TCP/IP or serial connection code. Alternatively, they may be very complex communication components such as self-organizing mesh networks, TCP/IP networks with additional discovery algorithms or entirely new communications components.

Each of the classes should conform to a common interface to allow automatic construction of the resulting run-time code.

E. DESIGN PATTERNS

Design patterns are high-level abstractions of common design problems. They help designers describe components, or collections of components. By using design patterns, they can develop designs that are extendable and mutable. If they create designs that specify a particular design pattern, they can take advantage of polymorphism, create new behaviors within this pattern later, and reuse the high-level design. This means they can add new artifacts, optional components, or communication components as needed.

To use design patterns effectively, engineers take advantage of common abstract interfaces. The glue code generators defined at the design level bring together a collection of interfaces; the details of the actual implementation below the interface are unimportant to the glue code generator. As an analogy, think of a soda bottling plant. The design for the plant includes a capping machine. The capping machine is concerned with the interface between the bottle top and the cap. It is not concerned with the flavor of the beverage inside the bottle. When modifying the plant design to change from a crimped cap to a screw cap, all the designers need to be concerned with is the interface, they do not need to be concerned with the flavor of the beverage that is being contained.

Design patterns also compare well to composite digital devices. When creating an electronic design, designers refer to reference material of discrete components. These components have a well-defined interface. Designers may be concerned with some of the characteristics of these components dictated by their internal makeup, such as power consumption or latency, but are not usually concerned about the intricate details. What designers are concerned about is the interface. In order to compose a circuit, designers need to know the pin outs and function of the device. They find this in a reference volume or specification sheet from the manufacture. In many cases, there may be more than one manufacture; the internals of the chip may be different, but the interface is common. This allows designers to use tools that can layout the traces on a circuit board.

Design patterns are becoming a similar abstraction for software. A particular design pattern specifies an interface and function of interest to the high-level designer. Component developers do not need to know in what context the design pattern is used; they need to know the function and interface they are creating.

Reference material is becoming available for software design patterns, just as there are reference volumes for electronics. There are several excellent books available for understanding design patterns:

- “Design Patterns, Elements of Reusable Object-Oriented Software [43]” provides a catalog of common general-purpose patterns.
- “Head First Design Patterns [44]” is a very readable introduction to the most common design patterns. It provides detailed examples with UML descriptions and Java code.
- “Pattern-Oriented Software Architecture, Volume 2, Patterns for Concurrent and Networked Objects [45]” provides patterns to solve the often-difficult problems associated with communications in distributed systems.

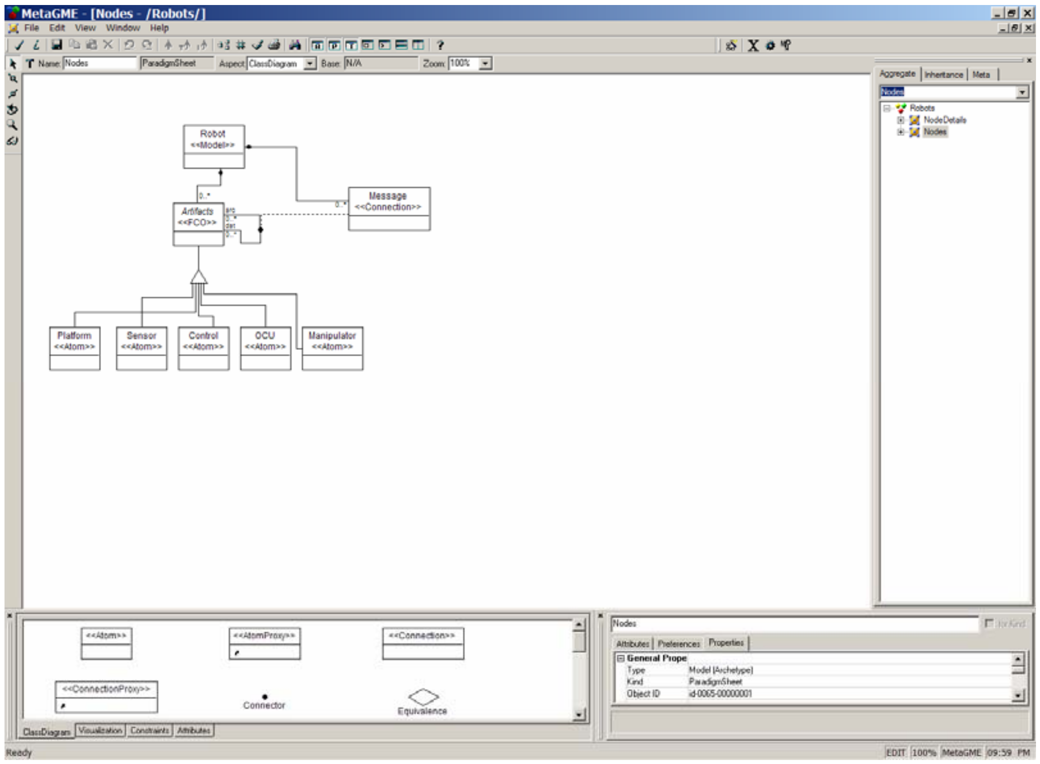
There are also many web-based references, such as Bob Tarr’s lecture Notes, “Design Patterns in Java, http://www.research.umbc.edu/~tarr/dp/fall00/cs491.html [46].”

Three main design patterns are used in this dissertation:

– Adaptor. The adaptor patterns wrap the legacy and research artifacts that represent the physical and control nodes of the robotic system. The input and outputs of the adaptor pattern will be XML representations of JAUS messages. The adaptors translate the JAUS messages to the software and physical formats necessary to the artifact. As an example, the ODIS-T2 robot accepts proprietary data packets via a serial port. The wrapper will convert to and from JAUS message format to ODIS-T2 format; it will also transport the proprietary data packets via a serial link.
– Observer. The observer patterns will be responsible for passing the JAUS message to any instrumentation, modification or other optional component specified in the design. Observers will insure that the appropriate components see and/or operate on each incoming and outgoing message.
– Factory. The components will be composed into a node of the system via factory patterns. The communications components are expected to vary widely, from simple serial to very complex mesh networks with discovery; adaptors will wrap a growing number of legacy components Using factory patterns insures that the system will be extendable. As the component collection grows, only the concrete components will change. Using a factory design pattern will allow composition of nodes using new components and avoid instantiation of application specific classes.
– Researchers may use additional design patterns in conjunction with the main design patterns within components. This will simplify modifying and expanding components as the system matures.

F. META-MODEL

The Meta-modeling environment for this project is the Generic Modeling Environment (GME) [47], an open-source, visual, configurable environment for creating Domain-specific Modeling languages. GME use starts with configuration of the modeling environment, i.e., modeling of the modeling process or creating a Meta-model. The modeling language is UML class diagrams. **Figure 26** shows a simple GME Meta-model for a robot system, the work under discussion.



Type Model (Archetype)
Kind ParadigmSheet
Object ID id-0065-00000001

Figure 26. Generic Modeling Environment, Meta-model of a Robotic System

The Meta-model is a source document. That is, unlike previous CASE models of the 1990's, it is not left behind to get out of synchronization with the implementation. The Meta-model defines a paradigm, a set of rules that will configure the GME for a specific operation.

In the case of **Figure 26**, the top-level object is a model labeled "Robot." Contained in the Robot Meta-model are messages and artifacts. Artifacts are abstract;

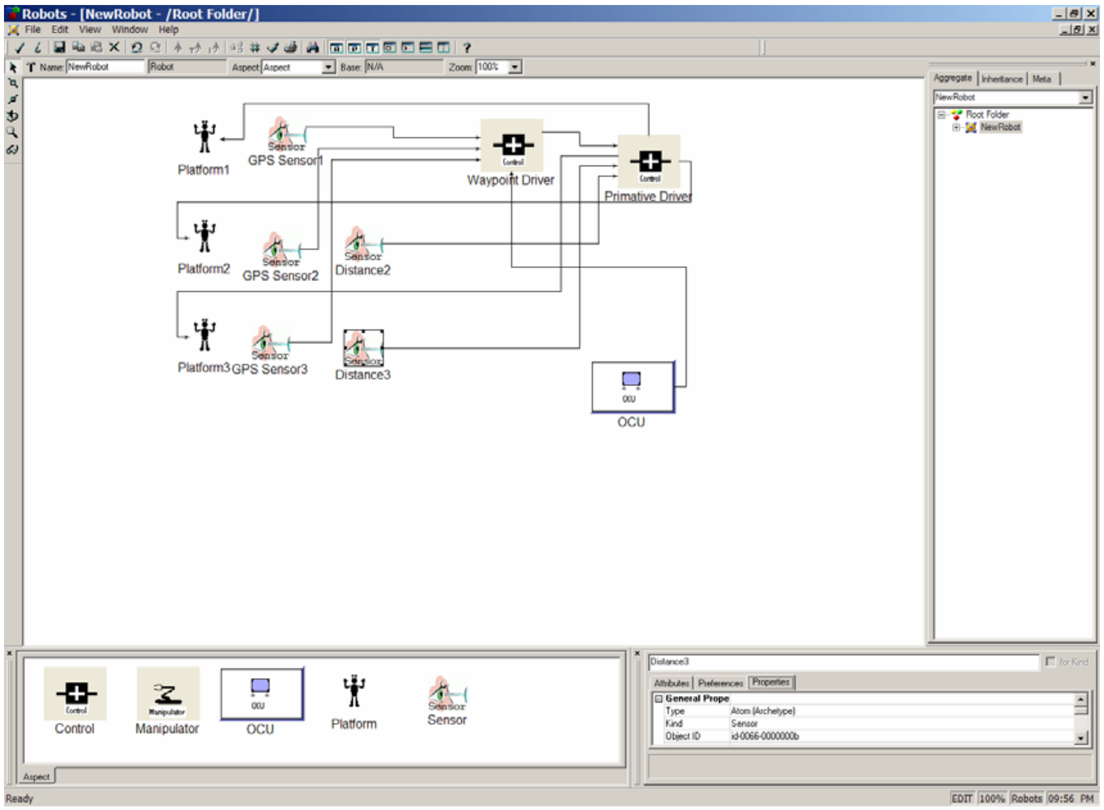
they do not have any implementation. Inherited types, bottom-level objects or atoms define the artifacts. There are five different types of atoms possible to represent artifacts, and a robot model can contain one or more artifacts. Finally, connections between the artifacts are “messages;” artifacts can send or receive zero or more different messages.

G. DOMAIN-SPECIFIC MODEL LANGUAGE

The GME tool generates a Domain-specific Modeling Language (DSML) from a corresponding Meta-model. Note that due to configuration of the Meta-model, domain-specific icons that represent their functionality in terms of the domain of interest, in this case, robots, now represent the artifacts.

A domain engineer manipulates the DSML modeling environment to create a prototype design. The domain engineer selects from a palate of approved abstract artifacts, [controls, sensors, platforms, Operator Control Units (OCU) and manipulators] and creates a model by connecting them in a meaningful way (**Figure 27**). The underlying components that will be used are aspects of the artifacts dragged onto the workspace.

There may be any number of models created by the domain engineers. The limit on the number of models possible is a function of the cardinalities designers and engineers imposed in the Meta-model and the number of component instances available for each artifact. This particular model is of three robots, a leader and two followers. Each has a GPS positioning sensor and the two followers have distance sensors. The waypoint driver control computes waypoints for the two followers based on the input from the five sensors. The waypoint driver passes new messages to the primitive driver to control the two follower robots and receives messages from the OCU to control the leader. For instance, OCU messages influence the waypoint driver to vary the distance.



Distance3

Attributes | Preferences | Properties

General Prope	
Type	Atom (Archetype)
Kind	Sensor
Object ID	id-0066-0000000b

Figure 27. Generic Modeling Environment, Domain-specific Modeling Workspace

The Meta-model-to-model translation facilitates transfer of specific software engineering knowledge to non-software domain engineers. The domain engineers use the domain-specific model to compose models of instances of a "robotic system product line." This allows project leaders to control development and manage differences while leveraging common characteristics of the application domain [48].

H. CODE COMPOSITION / GENERATION

The ultimate goal of this research is to free the domain engineer from the arduous task of creating code for prototype robotic systems. All code needed should be created by software experts and stored in a repository, so that the domain engineer can select the icons that represent collections of code. The domain engineer selects a particular code by completing an annotation in the domain model.

The domain model completed by the domain engineer is represented by an XML file. This file contains all the information needed to recreate the domain model. It also contains all the information needed to compose components and create glue code for the robotic system.

```
…
<atom id="id-0066-0000000c" kind="Control" role="Control"
relid="0x15">
     <name>Primative Driver</name>
     <regnode name="PartRegs" status="undefined">
          <value></value>
         <regnode name="Aspect" status="undefined">
                <value></value>
                <regnode name="Position" isopaque="yes">
       <value>765,72</value>
                         </regnode>
           </regnode>
      </regnode>
</atom>
<connection id="id-0068-00000001" kind="Message"
role="Message" relid="0x5">
     <name>Message</name>
     <connpoint role="src" target="id-0066-00000001"/>
     <connpoint role="dst" target="id-0066-00000004"/>
</connection>
<connection id="id-0068-00000005" kind="Message"
role="Message" relid="0x13">
     <name>Message</name>
     <connpoint role="src" target="id-0066-00000001"/>
     <connpoint role="dst" target="id-0066-00000005"/>
</connection>
<connection id="id-0068-00000008" kind="Message"
role="Message" relid="0x16">
     <name>Message</name>
     <connpoint role="src" target="id-0066-0000000a"/>
     <connpoint role="dst" target="id-0066-0000000c"/>
</connection>
...
```

Table 1. Fragment of XML Code Generated by Domain Model Instance

Table 1 is a fragment of code from the domain model represented in **Figure 27** above. Remember, this is a simplified model created for illustration purposes only, there are no attributes associated with the atoms (icons) or connections (lines). It also does not have lower level components associated with it that would be necessary for complete configuration of the artifacts for use in a prototyping system.

Even so, complete robotic system code could be created from the XML file represented in **Table 1**. One atom is shown, the primitive driver control. Both the primitive driver and the waypoint driver are simply controls in the Meta-model. In this domain model, the XML <name> element differentiates them. The GME environment also assigns them “IDs,” which are used later by the XML <connection>/<conpoint> element to specify the source or destination points of the connection.

Adding additional attributes to the Meta-model will allow additional tuning of the generated/composed code. Enumerated attributes can constrain the domain engineer to a selection that may be a subset of all the components of this type, i.e., a particular set of sensors.

To generate the code, the XML tree is parsed and, in this case, large components (containing predefined wrappers, instrumentation and communications) are written for each of the artifacts. Researchers can configure the components by using the parsed XML tree as input to a compositional script written in PERL. Another possibility is to transfer the XML to a generative environment, such as an ECLIPSE project [49].

I. MODEL-DRIVEN DESIGN CONCLUSIONS

Model-driven Design has great potential to extend the software engineers knowledge to domain engineers. It provides a vehicle for software reuse through the focus on predefined software components. It simplifies the job of the engineer creating the prototype system by allowing him to focus on the task at hand. It also reduces the time and cost required to evaluate a new application or mission

Since the Meta-model is the root of all the design efforts, subsequent activities are traceable to the initial Meta design level. Finally, prototyping with Model-driven design provides path forward for implementation of final systems.

THIS PAGE INTENTIONALLY LEFT BLANK

VIII. EXPERIMENTAL IMPLEMENTATION

A. INTRODUCTION

This section details an experimental implementation of the approach described in the preceding part of this work. Going back to **Figure 17**, this section addresses the implementation of the architecture.

The model that will be examined throughout and defined in the Meta-modeling phase is simple, but common, especially in the early stages of system concept exploration. This section constrains the nodes of a system to having one output stream (they can have as many input streams as necessary). Think of a sensor fusion scenario. DoD researchers might want to create a system control node that integrates Laser Radar (LADAR), visual, and geospatial messages. Each message stream originates on a separate processor or microcontroller. A software control algorithm, along with a message stream from an Operator Control Unit, processes the set of input message streams. The control transmits a single message stream with appropriate fused data.

The first section will cover component concerns, including coding practices, and required design patterns. It will also discuss simple matters, such as directory structure for the component repository. This is mainly collaboration between software engineers, computer scientists and programmers to agree on common practices, like we all drive on the right side of the road in the US.

The second section, Meta-modeling, discusses software modeling. The model is primarily the work of a software engineer, in consultation with a domain engineer. It is also influenced by previously agreed upon practices, design patterns and completed components.

The third section, Domain Modeling, is primarily the responsibility of the domain engineer. When the domain engineer finishes a model, the translation and code composition is dependent on up-front work performed by the software engineer.

Lastly, the codes are distributed to the various nodes for operational evaluation, potentially, to evaluate hardware/software or to evaluate system level procedures.

The most important thing about this section is that it is the first step in an evolving system. TARDEC engineers and researchers want to capture expertise and data as time progresses and be able to use it in new and interesting ways.   Component repository development is an ongoing process; capture as much as possible, use versioning as necessary, and purge when corresponding artifacts are no longer viable.

The component repository, or more correctly, the structure of the component repository, is an input to a Meta-model. Different Meta-models may specify different subsets. Each Meta-model may spawn one to many domain models, depending on the richness of the codebase subset allowed in the Meta-model.

Each Domain Model will produce one set of Node Codes, but researchers will be able to operate the system under evaluation in various scenarios. For instance, if they build the example system, they may operate it first with static obstacles for a first experiment, next they may add dynamic obstacles, and finally, they may introduce systematic faults at one or more nodes of the system to see how it operates under duress. See **Figure 28**.

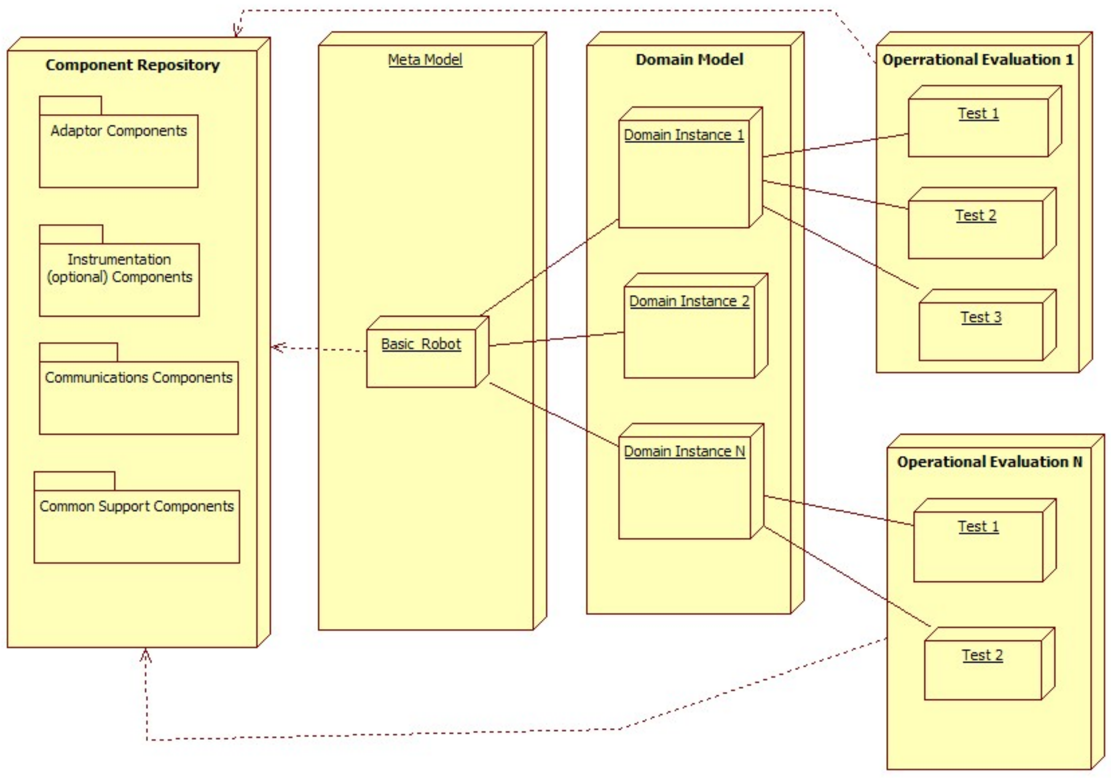


Figure 28. High Level Deployment View, the Relationship between Components, Models and Applications

B. COMPONENTS

1. Component Repository

One of the first implementation concerns is where the components of the system will be stored. This is a relatively simple decision, but it must be defined early to provide a starting point; call it the codebase. Component repository is a generic term used to describe the storage area for components, scripts and related documentation. Ideally, software engineers would be implemented it as a network enabled repository, since a network enabled implementation allows multiple, geographically-dispersed users and contributors to the system. However, that imposes the administrative and security concerns researchers attribute to any shared system. These concerns are not addressed here.

The component repository for this experiment is implemented in a simple file system on a local workstation. The important issue is component repository content, as well as the directory structure. Designers must define directory structure before modeling, since the rule sets that compose the final node codes are dependent on the directory structure.

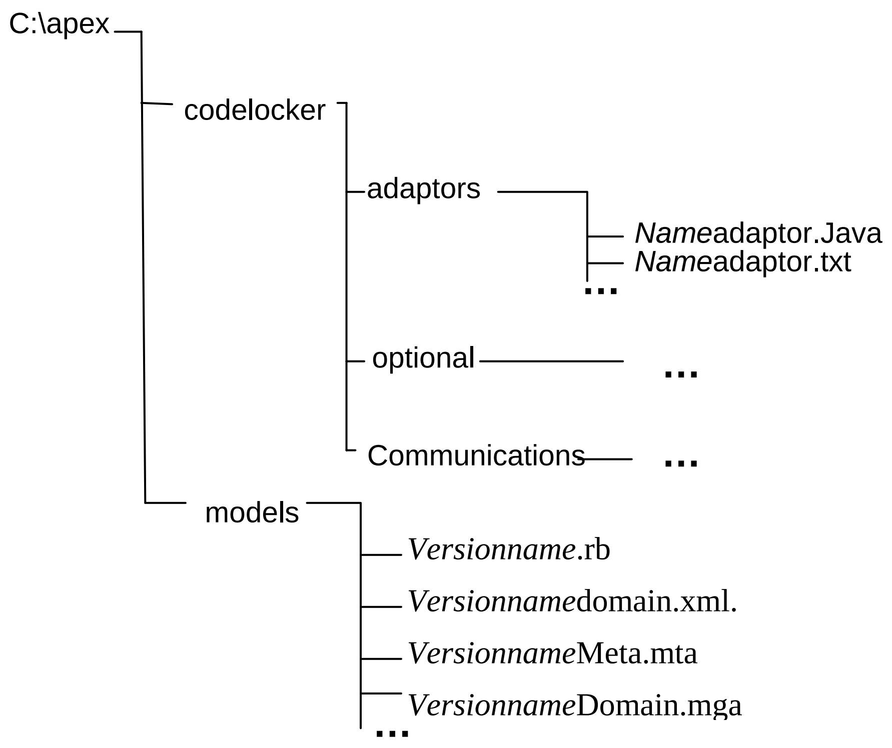


Figure 29 . Code Repository in Windows Directory Structure

Figure 29 shows the directory structure used in this first experiment. The root of the component repository is the directory c:\apex. Two subdirectories are codelocker and models; codelocker is the main and most important subdirectory, while the models subdirectory is just a convenient place to store Meta-models, Domain Models, and XML versions of the domain models as well as Ruby scripts created after a Meta-model is defined.

The codelocker subdirectory contains three subdirectories:

- Adaptors;
- Optional; and
- Communications.

Again, software engineers create these directories as a convenience for humans, in this case to separate different component types. All three of these directories contain two types of files: components and component user documentation.

Rules merge the abstract domain model with concrete Java components. The structure for the codelocker mirrors the atomic or leaf nodes of the feature model; where the structure identifies the concrete components in the models, is parallel to where they are found in the codelocker. This makes reading and writing rules less cumbersome.

Storing files on an isolated computer is not the ideal situation, but is necessary during the research phase of this work. A first-level expansion would be to create a shared file system, but in the TARDEC lab environment, this only allows local users access.

A higher-level solution would be to encapsulate the code objects within a database. The tables that have the code could then have related versioning information and fields to help search for code objects, created within a time period, by a particular group or for a particular artifact.

2. Components

In general, engineers could create components in virtually any language. If they do so, they also need to provide mechanisms to cope with a non-homogeneous system, adding significantly to the environment's complexity. Alternatively, component code can be restricted to a single language. For the work TARDEC engineers and researchers are doing, and for work projected in the near future, they chose the latter alternative. For this experiment, all component codes are in the Java language.

There are many compelling reasons to use Java for prototyping:

- There is a large open-source community associated with Java;

- Compilers and run-time environments are available for a large number of processors;
- It is an object-oriented language; and
- It has an intimate association with XML.

As mentioned above in the directory structure section, engineers separate the components into three categories: Adaptors, Optional and Communications. The Java files here will not be complete implementations. Complete implementations will be reserved for future efforts; the purpose of this experiment is to demonstrate the systems engineering approach described in the first part of this dissertation up to the point of bringing the code together. Creation of Java Libraries will require time and effort.

3. Documentation

In prototyping, we are working with a collection of hardware and software artifacts. Some may be Commercial-Off-The-Shelf (COTS) items. Others may be artifacts remaining from previous research projects. We start with a full compliment of hardware, software and ancillary cables, connectors and power supplies. Over time, however, we see these systems begin to “dissolve;” documentation is separated, cables get lost and software gets misplaced. We can reverse or prevent this phenomena by adding information about these artifacts and their environment to the component repository at a level parallel to the Java components.

Each Java component in the component repository will have associated with it a text file containing documentation, or pointers to documentation. This documentation is not used directly by the tools or modeling environment. The documentation files associated with the components are essentially fragments of a user’s manual. The domain modeler needs to be aware of this documentation, because in some cases, the modeler will need to refer to it for information needed during domain modeling.

Some of the information in the documentation will be component application specific, such as constants that the software engineer needs to consider at model design time, like a legacy artifact with a fixed serial speed. Other information will be necessary at system assembly time, as in the case of an adaptor that requires a special serial cable to connect to a legacy artifact. These files will be collected by the rule set at the same time the components are assembled and placed in the same file structure where the completed wrapper code is stored.

In the case of communications components, the documentation may be quite simple, perhaps the source of the code. In other cases, the communications code may be more complex, as in the case of a component that represents a communications element of a framework that performs discovery, in which case, any assumptions needed to deploy the component are necessary as a guide to the software engineer creating the models in which it will be used. The complexity of documentation for optional components will depend on the complexity of the component. A logger may include only the location of the log file, while a run-time monitor may include a pointer to an operations manual for that monitor.

Adaptors are expected to have a collection of information unique to the artifact they are adapting; special cables have already been mentioned, but locked IP addresses, specific serial ports and other hardware parameters are also candidate information. Designers may need to add pictures, serial numbers and wiring diagrams, and sometimes may version adaptors to reflect changes to hardware configurations. If that is the case, the documentation files should also be versioned to reflect the changes.

4. Design Patterns

As mentioned earlier, design patterns play an important part in implementation of understandable, reusable software systems. We include this discussion here, under components; since this is where structural (adaptor) and behavioral patterns (observer) are used and considered.

The design patterns provide a common working ground for writing rules. In the example, researchers use the observer pattern to pass messages between components within a node. Now they know that intra-process communication only requires adding addObserver messages in the Java main class. Rule developers do not have to concern themselves with pipes or other methods of inter-process communication.

a. Observer Pattern

Designers use the observer pattern [50] to allow components in a node to communicate. The individual components must pass messages amongst themselves. For example, incoming network messages are instantiated in the communications component; the communications component needs to notify the next component in the node that a message is available. In the observer pattern nomenclature, the communications component is a subject, and the next component is an observer. When the state of the subject changes in a meaningful way, the subject notifies all of its observers. The notification can be simple with no argument, or it can pass the message as an argument. In the former case, the observer must request the update after notification; this is sometimes called a pull operation. The latter case is called a push operation because the update information is pushed along with the notification. Here, researchers chose to push the incoming message as an argument of the notification to the observer (**Figure 30**).

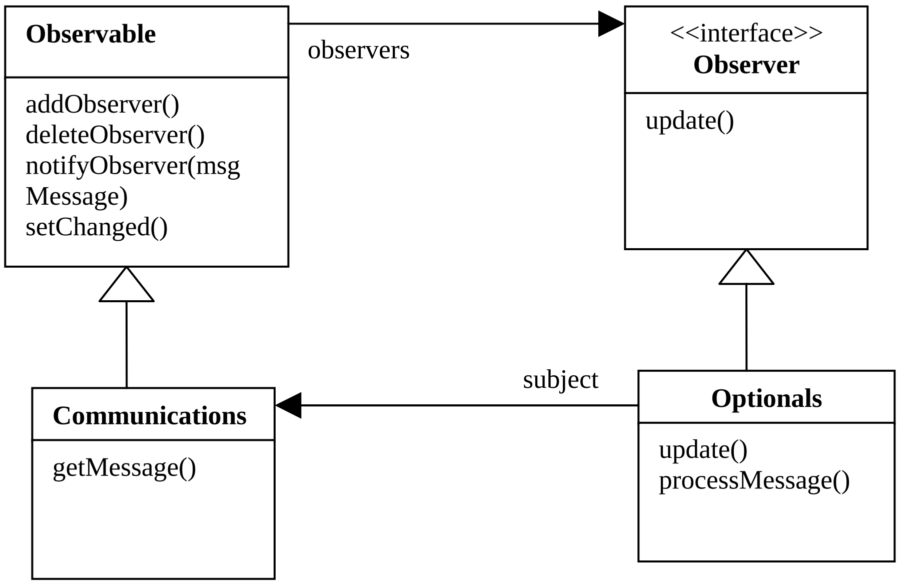


Figure 30. The Observer Base Class, UML Model of the Java Observer Design Pattern

An observer can also be a subject. Many optional components will take on both aspects of the observer pattern, as messages will pass through them. Some optional components will operate on messages. For instance, a throttle or leaky bucket will be used to limit network bandwidth available or to introduce delays in order to simulate real-world network conditions. In other cases, an optional component may change a message or even delete it to introduce errors. Monitor and log programs do not necessarily need to be pass through, but are implemented as such in this experiment as a matter of design choice. Since most adaptor and communications components have two-way communications, most will be implemented using subject/observer patterns (**Figure 31**).

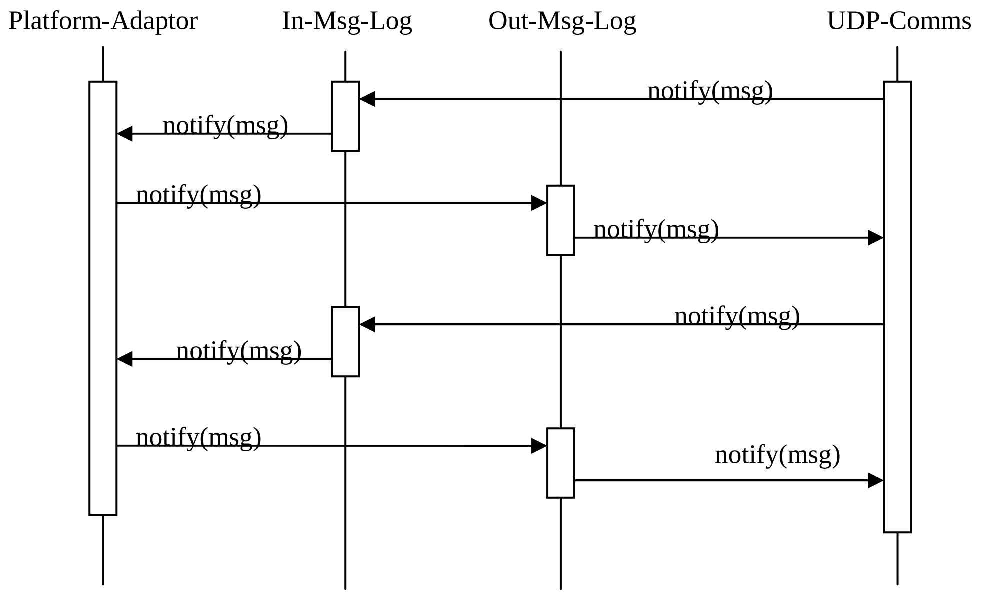
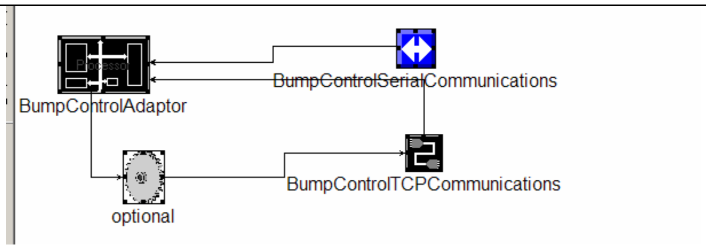


Figure 31. Collaborations for Components with Observer & Subject Roles Noted

The observer pattern is a commonly used construct, and the Java language provides built-in support [50]. The Java.util.Observable class is a base subject class. Here, designers will use this class for all components by having abstract Adaptor, Communications and Optional classes that extend Observer. Specific adaptors will subclass the adaptor class, Communications class and Optional class respectively.

The Java.util.Observer interface is the observer interface. The abstract base classes for the components will implement the Observer interface. The concrete components will inherit both behaviors. The actual use of the behaviors will be coded into the concrete components; that is, when a message has progressed through a component, a state change will be triggered that will cause the message to transfer from one component to another.

Model	 BumpControlAdaptor BumpControlSerialCommunications optional BumpControlTCPCommunications

XML	`<connection id="id-0068-00000006" kind="Connection" role="Connection" relid="0x5" isinstance="no" isprimary="yes" isbound="no">`
	`<name>Connection</name>`
	`<regnode name="autorouterPref" isopaque="yes">`
	`<value>We</value>`
	`</regnode>`
	`<connpoint role="dst" target="id-0066-00000003" isbound="no"/>`
	`<connpoint role="src" target="id-0066-00000004" isbound="no"/>`
	`</connection>`

Ruby	["BumpControlAdaptor,” "optional,” "BumpControlTCPCommunications"]
Java	BumpControlAdaptor.addObserver(optional);

Table 2. Different Levels of Dataflow Representations

The component interactions are governed by having observers register with subjects. This registration process takes place at the main program level, after the individual component objects are instantiated. The registration process is governed by component dataflows defined in the Domain Model. A list of ordered dataflow relationships is extracted from the Domain Model XML file and translated into a set of addObserver methods written to the main Java program by a Ruby rule. **Table 2**, depicts the representations of component dataflow as it progresses through the various transformations, beginning at the domain model where specific dataflows are defined. The first representation is a directional line in the domain model. The domain model is saved as an XML file and GME generates XML describing the dataflow. The relevant data is extracted from the XML domain model and stored in a Ruby array in preparation for generating a Java message. Finally, the dataflow is defined in Java as an Observer addObserver message relating the dataflow source component, “BumpControlAdaptor,” with the destination component, “optional.”

b. Adaptor Pattern

The adaptor pattern [52], also known as a wrapper was the original central idea that launched this work. In the TARDEC lab, as in many robotics labs in DoD, Industry and Academia, researchers and engineers have a variety of platforms, sensors, control codes, and Operator Control Units. Within a system, these items work together; trying to make the individual items work together across different systems is quite another matter.

Army developers have a need to examine concepts rapidly, often as a precursor to entering into a formal development effort. This necessitates rapid prototyping. They are interested in assembling systems from parts of other systems as well as experimenting with control algorithms. Ideally, all the parts would communicate via a common protocol, such as JAUS, but they do not. In practice, programmers can write code that wraps the non-compliant devices and controls to allow them to communicate on a JAUS backbone.

Programmers wrap non-JAUS-compliant artifacts with adaptor subclasses specifically written to convert legacy protocol to JAUS protocol (**Figure 32**). The base Adaptor class interface extends the Observer pattern.

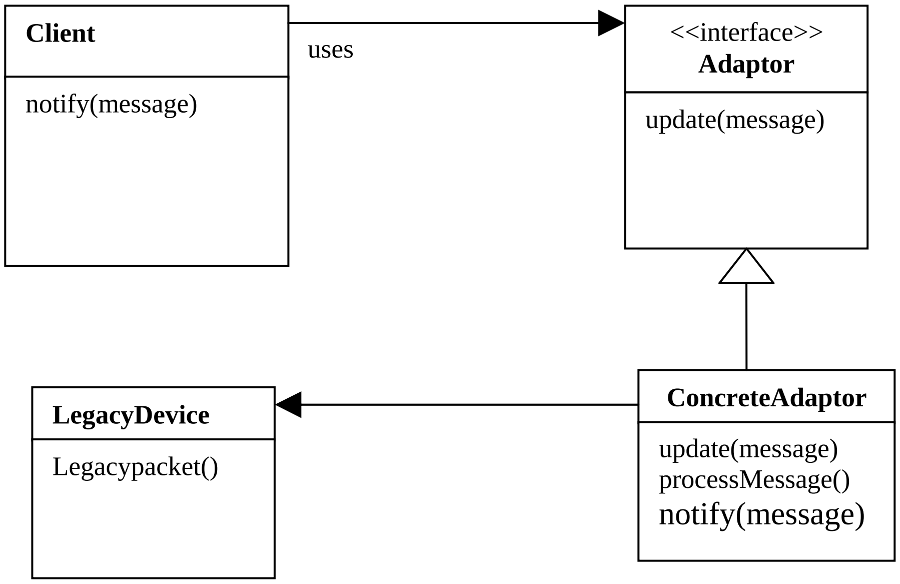


Figure 32. The Adaptor Base Class, UML Model of Adaptor Design Pattern Class Structure

The client passes in a JAUS message object. JAUS message objects are themselves abstract, and implemented via a collection of subclasses. Since JAUS is defined in XML, with a properly defined XML Schema, code to read and write JAUS messages can be created with the Java Architecture for XML Binding **(**JAXB**).** [53] Of course, the class to pass a message to the legacy item must be hand-coded. This may involve several JAUS messages to a particular adaptor since many legacy systems combine concerns in a single packet. An example is the ODIS robot. ODIS communicates by serial packets; the inbound packets, for example contain mobility data, camera control data and digital I/O data. This requires a class that can read more than one type of JAUS messages, package the data into an ODIS packet and send it out via a serial connection, since the ODIS robot's only physical interface to the world is a serial port on its internal microcontroller.

C. META-MODELING

1. Introduction

Modeling is central to this effort. Researchers use Meta-models to *capture* software engineering concerns as paradigms and as a vehicle to *transfer* those paradigms to a domain-specific modeling environment. Domain–specific design environments capture specifications. They can be used to generate code in particular applications. Matlab [54] and Labview [55] are two examples of primarily graphical domain-specific modeling languages. While they are useful in the domains for which they were developed, they are not easy to extend to an arbitrary new domain.

2. Why the Generic Modeling Environment?

When TARDEC researchers started this effort, they decided to use tools readily available to DoD engineers and scientists, wanting to avoid having to provide support across multiple proprietary tools or environments. The first consideration was to use tools that were developed for the DoD under any of a number of contracts. In particular, they were concerned that the tools should not have a significant initial cost and should not have significant recurring support costs. In particular, researchers wanted to use freely available software with little or no license restrictions for government use. The Generic Modeling Environment (GME) is one of the tools that fit the requirements; funding for GME in part came from the DARPA Information Exploitation Office (DARPA/IXO).

GME is a design environment specifically designed to be configurable to a wide range of domains. GME is configured by creating Meta-models that specify a paradigm for modeling in an application domain. The Meta-models are composed of syntactic, semantic, and presentation information, as well as organization, construction, and constraint information. The paradigm created in the Meta-model defines a family of application-specific models. In the GME Environment, the Meta-model of a specific paradigm is used to automatically gene a target domain-modeling environment (**Figure 33**).

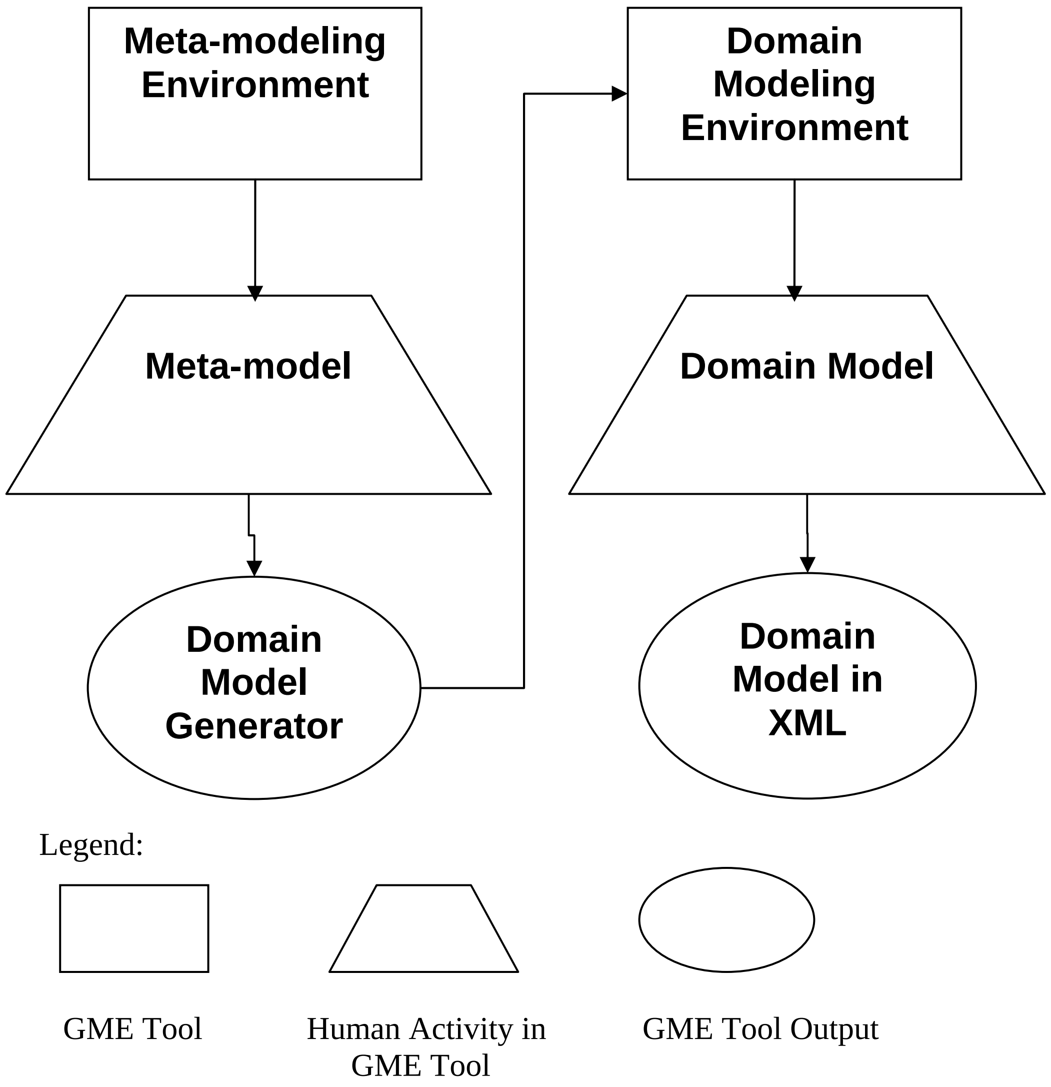


Figure 33. Overview of the Generic Modeling Environment; Software Engineers Create Meta-models, Meta-models Generate Domain Modeling Environment, Domain Engineers Create Domain Models

3. GME Concepts

GME supports a variety of modeling concepts [56] that engineers use to create an architectural description or Meta-model. These concepts include hierarchy, multiple aspects, sets, references and constraints. These concepts, when composed in a meaningful way, specify software architecture [57].

A GME Meta-model is defined as a project, that has a set of folders to help organize complex models. Folders contain models, which are composed of other models, atoms, references, connections and sets. Models, atoms, references, connections and sets are all GME “First Class Objects” (FCO). The number and kind of FCOs that are allowed in a model is determined by the modeling paradigm under construction and is defined by a containment connection. Contained objects can also be defined with an inheritance relationship. Atoms are elementary objects; they represent the lowest-level element of a model hierarchy. GME objects have attributes associated with the basic concept, such as role, name and kind. GME has a facility where additional attributes can be defined during Meta-modeling. The attributes that can be associated with an object include field (text, integer and double), Boolean and enumerated. If the attributes defined are associated with the parent object in an inheritance hierarchy, then the sub-objects inherit those attributes.

Relationships are modeled by creating a connection between two objects; These connections may be defined as directional or bi-directional. Two objects must have the same parent and be visible within the same aspect. Several kinds of connections can be defined in a single paradigm. The connections determine which objects can participate in a particular relationship, and connections can have attributes and cardinality. If it becomes necessary to associate objects in different parts of the model hierarchy, GME provides a Reference object that can be used exactly as other GME FCO. Any FCO except a connection may be referred to by a Reference.

GME models are similar to classes in Java. They can be sub-typed and instantiated as many times as needed. In order to promote reuse and simplify model maintenance, designers restrict changes that propagate down in the model. Attribute values of model instances can be changes, but no parts can be added or deleted. Sub-typed models may have new parts added, but parts from the parent model cannot be deleted.

GME’s Meta-modeling paradigm is based on the Unified Modeling Language (UML). Syntactic definitions are modeled using UML class diagrams, while semantics are specified using the Object Constraint Language (OCL).

4. Building the Meta-model

The scenario chosen for this experiment is relatively simple, but very common. It was chosen to demonstrate how this approach could satisfy real-world concerns and exercise concepts, yet be readily understandable. In this model, researchers are configuring a robot composed of "RoboticSystems." The modeling environment is started by launching GME, and selecting the built in metaGME paradigm from the paradigm menu. **Figure 34** shows the initial GME modeling screen with an abstract class "RoboticSystem" and several allowable sub-classes. These sub-classes are represented as Models, which means they are also container objects and will be composed of additional objects. The sub-classes are attached to the abstract base class by a triangle that is used to represent inheritance in GME's graphical modeling environment.

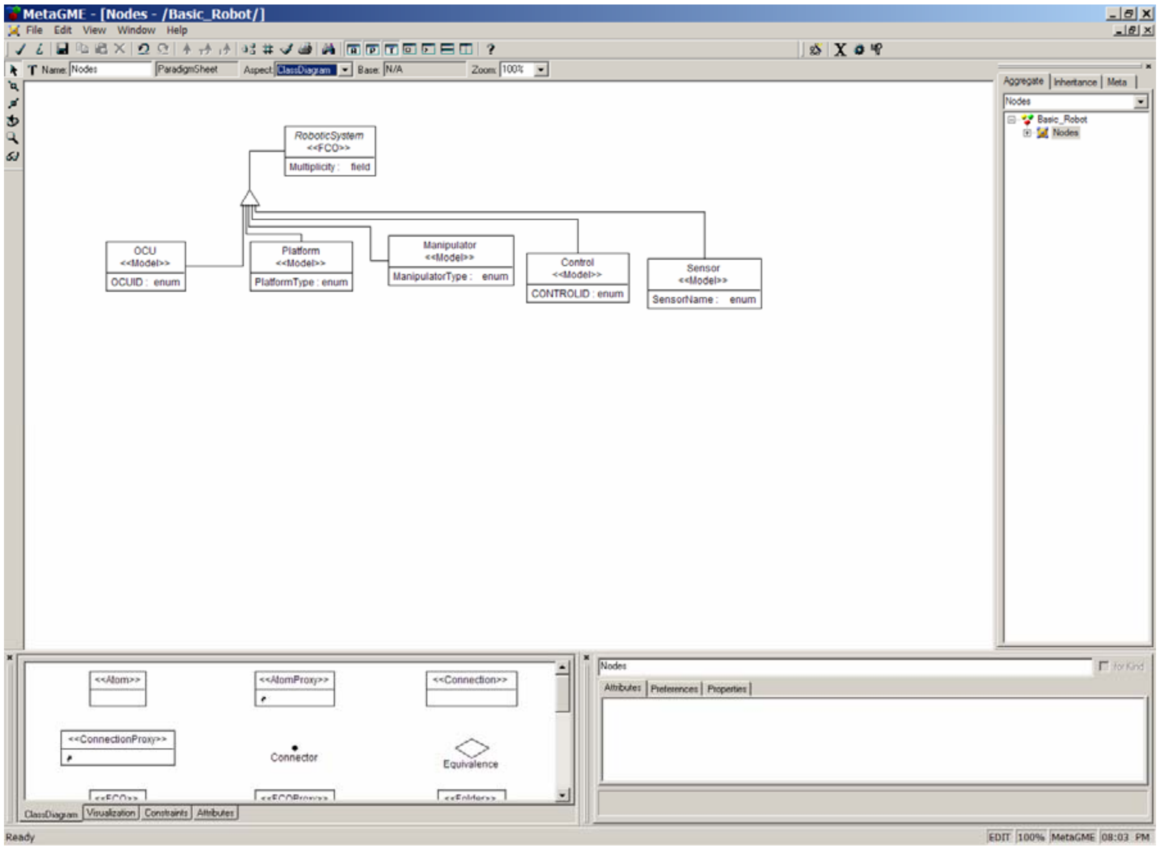
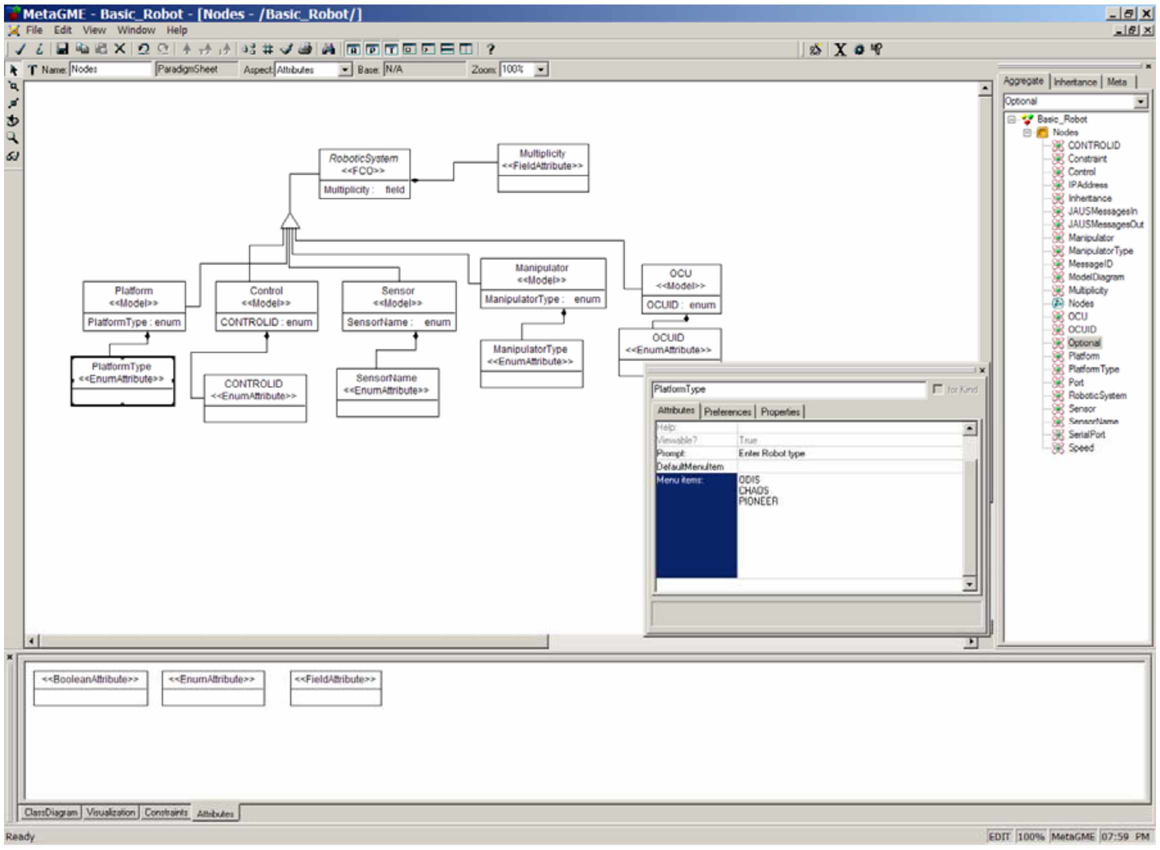


Figure 34. GME Screen Shot of Top Level Classes in the Meta-modeling Environment

The sub-classes included in this model are:

- OCU – Operator Control Unit. This can be any number of OCU’s and the final model may include multiple OCU’s if necessary, e.g., one for the mobility platform and one for the manipulator.
- Platform – This is a robotic mobility platform, or even possibly a simulation of a mobility platform.
- Manipulator – Self-explanatory, a gripper, multiple degree of freedom arm or some new concept to interact with the environment.
- Control – A control is usually a block of software. It may be embedded in a microcontroller. Researchers may be experimenting with a new control, or they may have a necessary control that is not integral to one of the other systems in the model.
- Sensor – Sensors come in two overlapping flavors, Mission Packages and environment sensors. An example of a pure environment sensor might be a bump stop. A vision system on the other hand, might overlap: it could be used to operate the mobility platform, as well as perform a mission function, such as facial recognition. A pure mission sensor might be a chemical detector.



Name: Nodes ParadigmSheet Aspect: Attributes Base: N/A Zoom: 100%

PlatformType

Attributes | Preferences | Properties

Help:
Viewable? True
Prompt: Enter Robot type
DefaultMenuItem
Menu items: ODIS CHAOS PIONEER

Figure 35. GME Screen Shot Showing Software Engineer Adding Attributes to Classes

The next step is to switch to the attributes view (**Figure 35**) and define attributes for the classes. In the Attributes view, modelers annotate the graphical model with information that augments the graphical model. As can be seen, the attributes show up as types in the attributes screen and appear in the lower box of the class diagram. The PlatformType attribute is added to the platform in **Figure 35**. This is an enumerated type used to restrict the Platform. The modeler can use one of the supported types, in this case, ODIS, CHAOS or PIONEER.

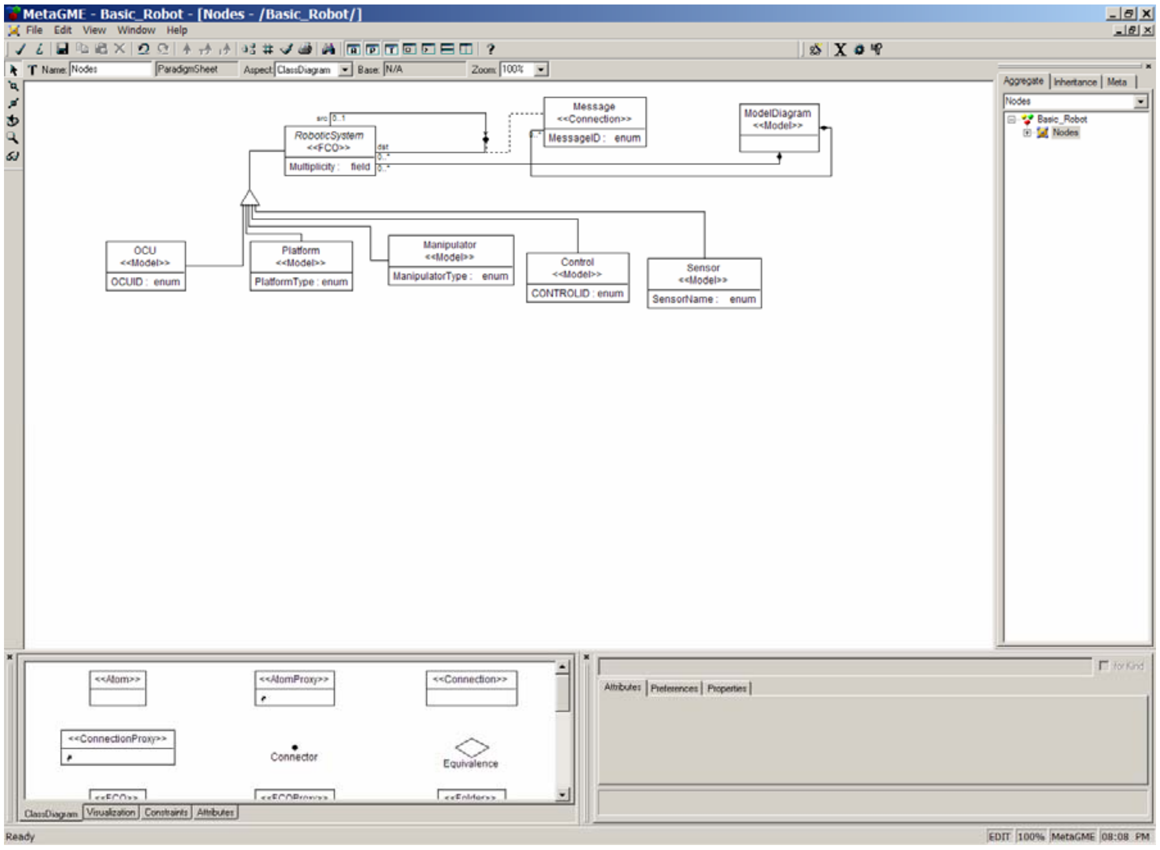
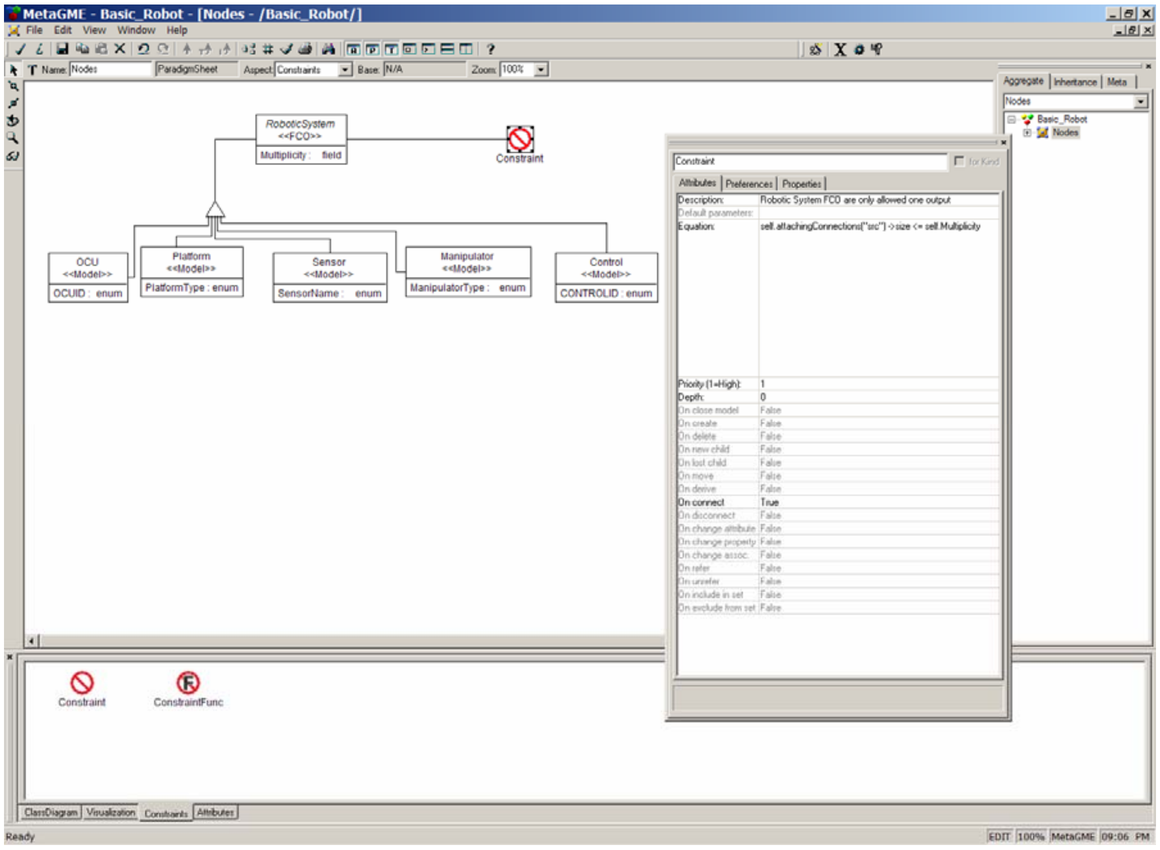


Figure 36. GME Screen Shot after Software Engineer Added Messages and Top Level Container

In **Figure 36** the modelers add a top-level container, ModelDiagram to hold all the model objects and a Connection. Connections express a relationship to associate different objects in the model. In this case, the relationship is message-passing between two objects. Note that connections may also have attributes associated with them. The message is associated with the abstract object, which shows that RoboticSystem objects can pass messages to other robotic systems. The sub-classes inherit this relationship and are allowed to connect message-passing streams between each other. Modeling in this way makes for a cleaner diagram, but also sets up the possibility for a domain modeler to create a message-passing connection from an object to itself. In the context of this model, that construct makes no sense, so the modelers are not too worried about it. The alternative is to model explicitly connections between the different sub-classes.



Constraint

for Kind

Attributes | Preferences | Properties

Description:	Robotic System FCO are only allowed one output
Default parameters:	
Equation:	self.attachingConnections("src") ->size <= self.Multiplicity
Priority (1=High):	1
Depth:	0
On close model	False
On create	False
On delete	False
On new child	False
On lost child	False
On move	False
On derive	False
On connect	True
On disconnect	False
On change attribute	False
On change property	False
On change assoc.	False
On refer	False
On unrefer	False
On include in set	False
On exclude from set	False

Figure 37. GME Screen Shot of Adding an Explicit Constraint to the Meta-model

As mentioned at the beginning of this section, the scenario being modeled calls for one and only one output message stream from each object. This may be somewhat restrictive in production, but at the early prototyping stage, it is acceptable and useful. It also makes for a more understandable example. **Figure 37** is a capture of the constraint aspect screen of GME. A constraint is added to the model to prevent the domain modeler from performing an operation that is not allowed by the paradigm being defined in this Meta-model. If a domain modeler attempts to create more than one source message connection from a given object, GME pops up a message screen disallowing the operation and explaining why. As can be seen, Constraints have attributes as well as a defining equation.

In **Figure 38**, the modeler completes the Meta-model by adding a collection of atomic elements. The atomic elements ultimately compose the individual nodes (OCU, Platform, etc.) of the system. It is very similar to the structure of the higher-level classes. ArtifactComponents is an abstract class and the leaf classes inherit connections from it. Two significant differences are that the sub-classes are all atoms, self-contained objects and there are no constraints associated with the Connection item. Note there is a variety of ways this could be modeled. For instance, Adaptor objects could have been modeled as an inheritance hierarchy. If the adaptor sub-classes had different attributes, this would have been the way to go. However, since they do not in this case, modelers simply add an enumerated attribute to the Adaptor class to select which adaptor they plan to use.

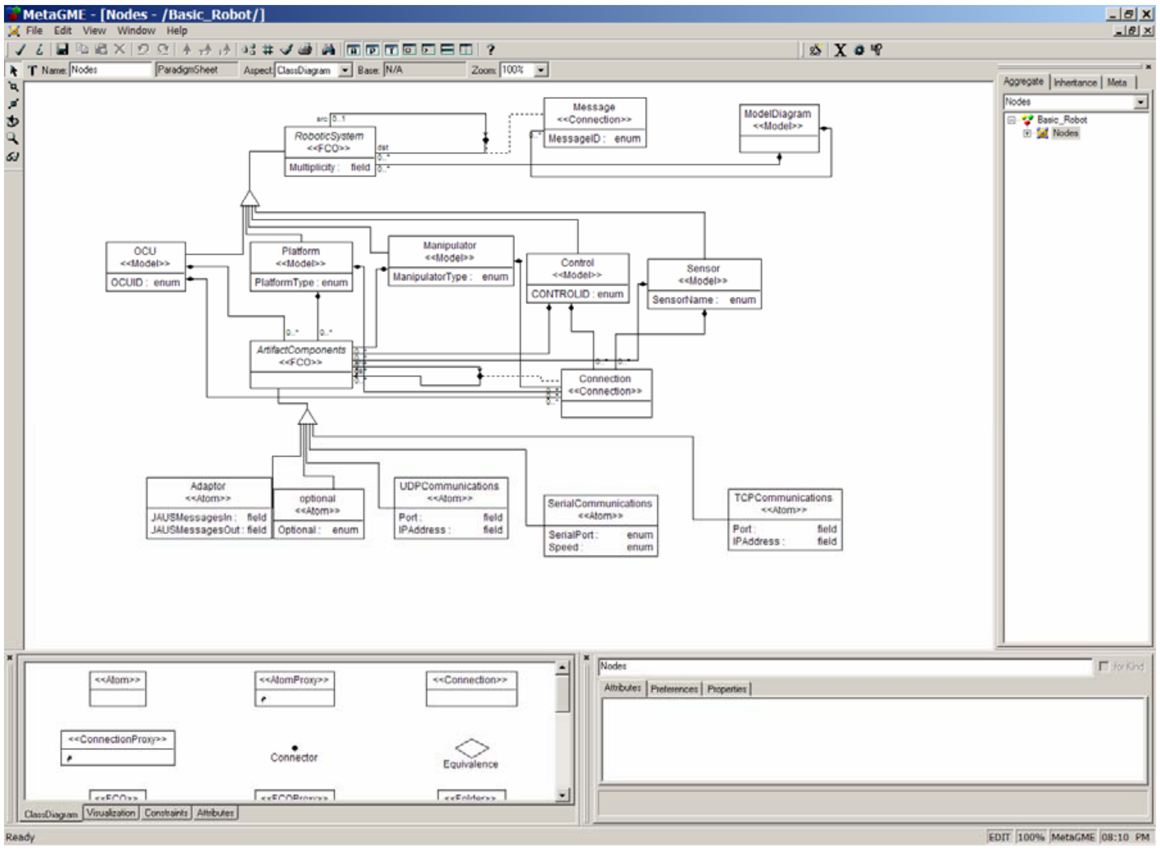


Figure 38. GME Screen Shot of a Complete Meta-Model for a Basic Robot

After completing the Meta-model, they need to prepare a new paradigm for use by the domain engineers. Fortunately, GME does this for the developers. On the top of the GME workspace is a toolbar with a small gear-like icon, the MetaGME Interpreter. Selecting this icon causes the environment to prompt the user to save the current paradigm as an XML file, and then prompts the user to register the paradigm. Once the paradigm is registered, it is available for use in domain modeling. GME automatically save the current workspace when a project is closed, unless specifically requested to abort.

D. DOMAIN MODELING

1. Introduction to the Model

In this section, we build and discuss a domain-specific model-based on the paradigm, experiment-meta, created in section VIII.C.4. The model we are creating is simple, yet plausible. Designers have a tele-operated robot, ODIS. They would like to add some semi-autonomous features to assist the operator. The concern is the robot bumping into things and either damaging itself due to impact, damaging the objects it bumps into or creating a safety hazard by bumping into pedestrians. Conceptually, they know this is a feasible scenario, and realistically need to operate the robot to prove that it will work. Many things can go wrong with the physical system. What size speed bumps will be detected as obstacles? Are there situations where the robot will not be able to recover from detecting an obstacle and need to be manually retrieved? Can adjusting the physical mounting of sensors overcome problems?

Researchers initiate this experiment with four artifacts:

- ODIS robot.
- ODIS OCU.
- Sonar Data.
- Control Algorithm.

They have an ODIS robot and an ODIS Operator Control Unit (OCU). The robot and OCU communicate via a set of proprietary serial packets, over an RF serial link. Both of these items use microcontrollers; researchers cannot alter the code on the microcontrollers, which they have examined and determined the serial packet structure of.

There is a sonar array, and a “sonar data collection system” developed as part a previous project with a local University [58]. The data collection system is i586 based, but designers do not have access to the operating system or the code. The sonar data board has a serial output with a published protocol.

An engineering staff specializes in control algorithms. Given a set of inputs, they will deliver a working prototype of the algorithm in a format of their choice that they only develop on a supercomputer array accessible via DREN. The robotic engineer's job is to analyze the situation, rapidly create a prototype and examine potential solutions, report on feasibility and, if feasible, initiate a development spiral to implement the solution.

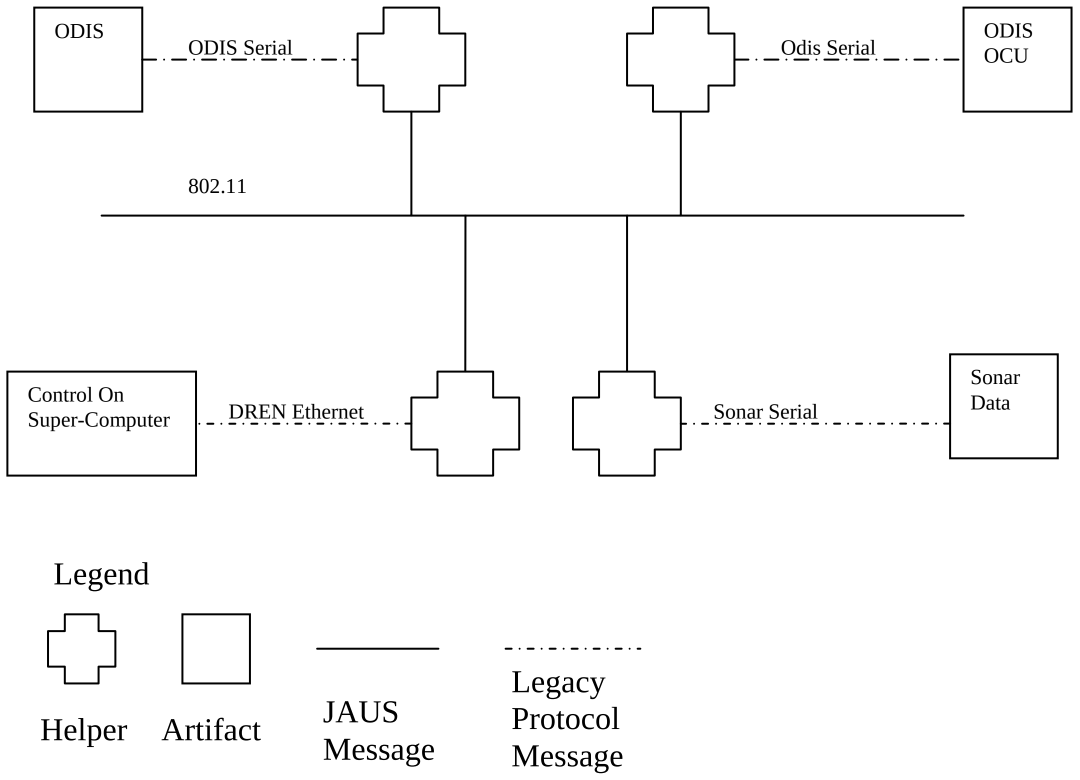
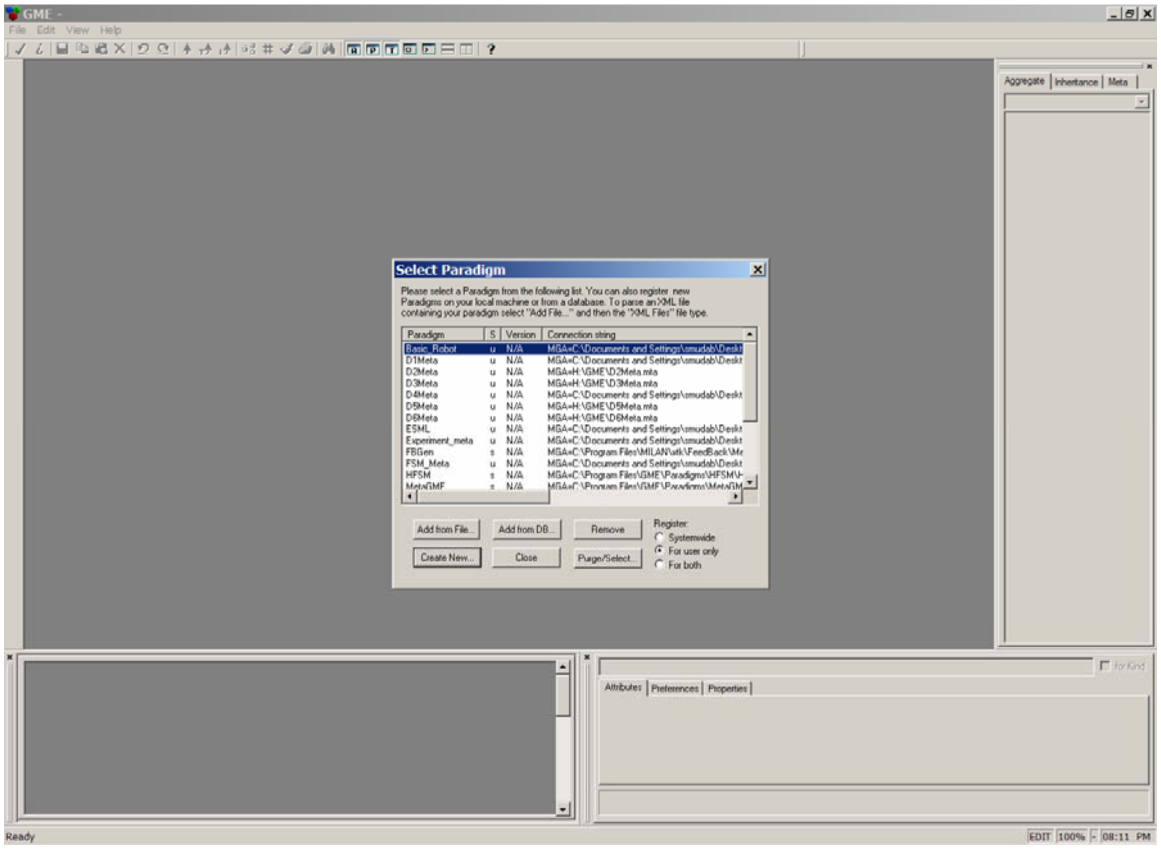


Figure 39. Hardware Block Diagram for Prototyping Environment Instantiation, Elements Include Legacy Artifacts Helper Computers and Network Environments

The hardware solution involves all the artifacts listed above, and a set of helper devices (most likely old laptops or PC 104 stacks) to adapt the artifacts to the prototyping environment. **Figure 39** represents a possible hardware configuration. Depending on access to the computing elements of legacy devices, the number of helper devices may be reduced by running helper node software on legacy devices, or running multiple helper nodes on a single helper device. The domain model output will be the basis to generate/compose the Java codes needed to complete all connections.

2. Configuring a Domain Model



Please select a Paradigm from the following list. You can also register new Paradigms on your local machine or from a database. To parse an XML file containing your paradigm select "Add File..." and then the "XML Files" file type.

Paradigm	S	Version	Connection string
Basic_Robot	u	N/A	MGA=C:\Documents and Settings\smudab\Deskt
D1Meta	u	N/A	MGA=C:\Documents and Settings\smudab\Deskt
D2Meta	u	N/A	MGA=H:\GME\D2Meta.mta
D3Meta	u	N/A	MGA=H:\GME\D3Meta.mta
D4Meta	u	N/A	MGA=C:\Documents and Settings\smudab\Deskt
D5Meta	u	N/A	MGA=H:\GME\D5Meta.mta
D6Meta	u	N/A	MGA=H:\GME\D6Meta.mta
ESML	u	N/A	MGA=C:\Documents and Settings\smudab\Deskt
Experiment_meta	u	N/A	MGA=C:\Documents and Settings\smudab\Deskt
FBGen	s	N/A	MGA=C:\Program Files\MILAN\utk\FeedBack\Me
FSM_Meta	u	N/A	MGA=C:\Documents and Settings\smudab\Deskt
HFSM	s	N/A	MGA=C:\Program Files\GME\Paradigms\HFSM\H
MetaGMF	s	N/A	MGA=C:\Program Files\GME\Paradigms\MetaGM

Register: ( ) Systemwide [✓ For user only] ( ) For both

Figure 40. GME Screen Shot as Domain Engineer Initiates a Domain Model by Selecting the Basic_Robot Paradigm Defined During Meta-modeling

Domain modeling begins by creating a new GME project and selecting the appropriate paradigm. The GME distribution contains four paradigms, each time a new Meta-modeling project is created a new paradigm is generated at completion. As a Meta-model is evolved, it can replace the current version, or the project can be copied to increment the paradigm version. For this experiment, researchers select Experiment_meta, created in the previous section (**Figure 40**).

Selecting the paradigm creates a blank workspace with a root node. To begin domain modeling, designers must select the root node and right click to get a context menu, then select "Insert Model." The only choice is "Model Diagram" (**Figure 41**) which is the top-level model container diagram created in the Meta-modeling process

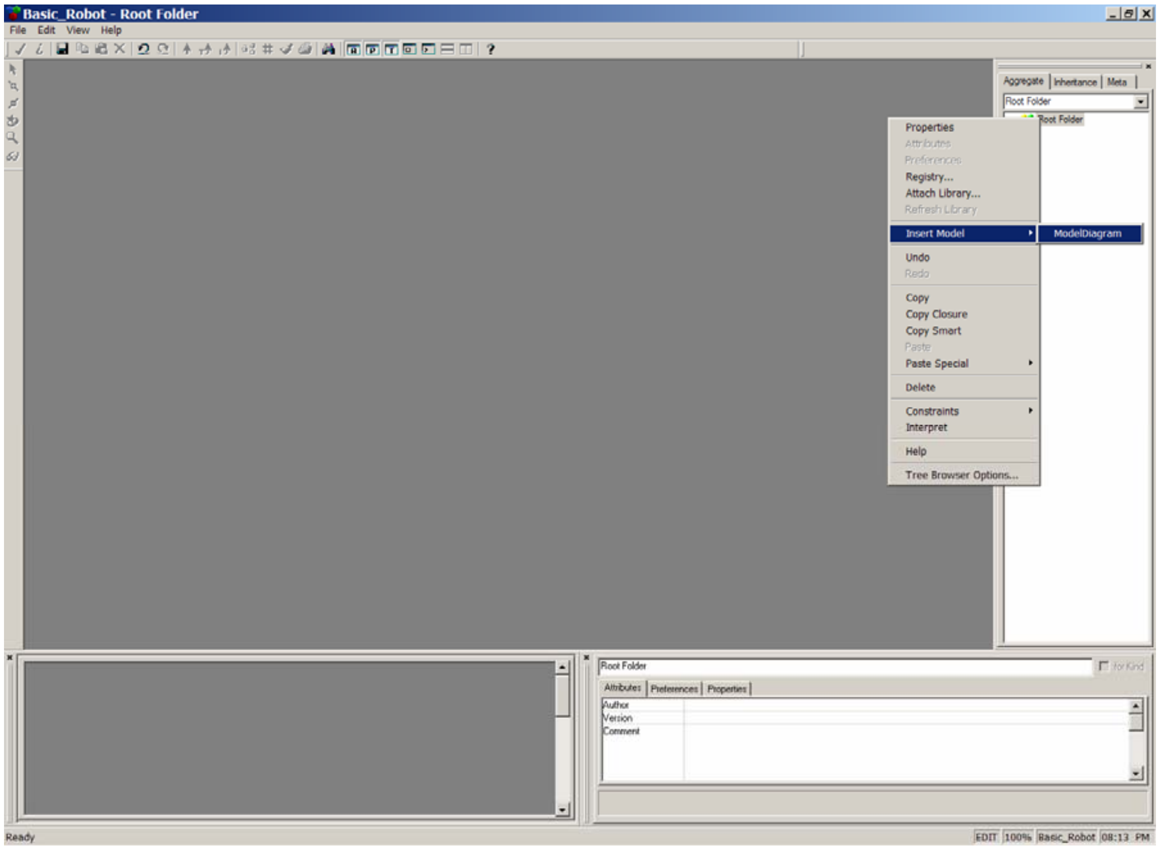


Figure 41. GME Screen Shot as Domain Engineer Creates a New Model using the Basic Robot Paradigm

The result is a new workspace to begin top level modeling of the domain (**Figure 42**).

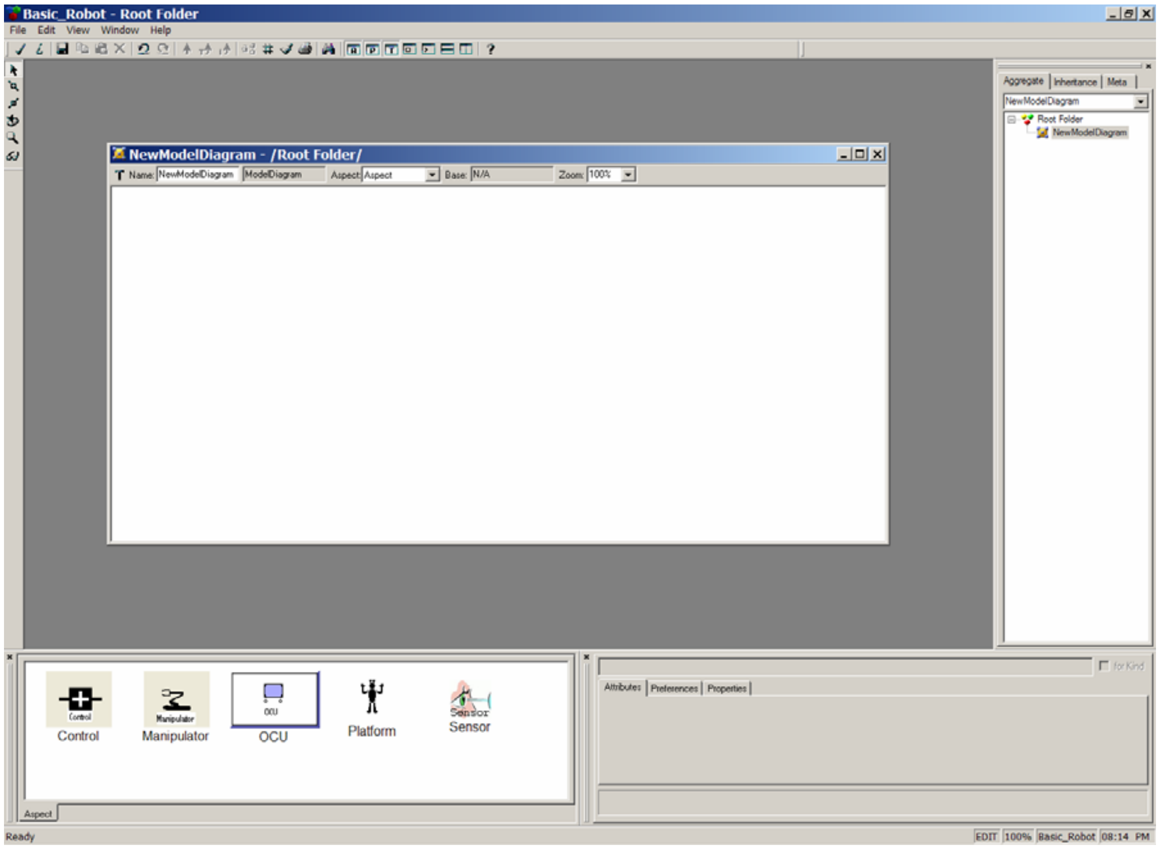


Figure 42. GME Screen Shot of an Initialized Top Level Domain Workspace; note the Domain-specific Icons for Legacy Artifacts

You will notice that in addition to the blank workspace, a set of domain-specific icons appears in the aspect window, labeled Control, Manipulator, OCU, Platform and (not visible) Sensor. Domain engineers drag icons from the aspect window to the workspace that represents the artifacts they wish to model.

The icons have attributes associated with them, both static attributes from the meta-Meta-model and configurable aspects the software engineer added as aspects during Meta-modeling. The static aspects include items such as the display name, which can be changed during domain modeling and other items, such as kind and role, which are fixed at meta-modeling time by the GME environment. Other items modifiable at Meta-modeling time, such as attributes, help define the target domain paradigm, while still other items, such as icon name and color, help define the look and feel of the models.

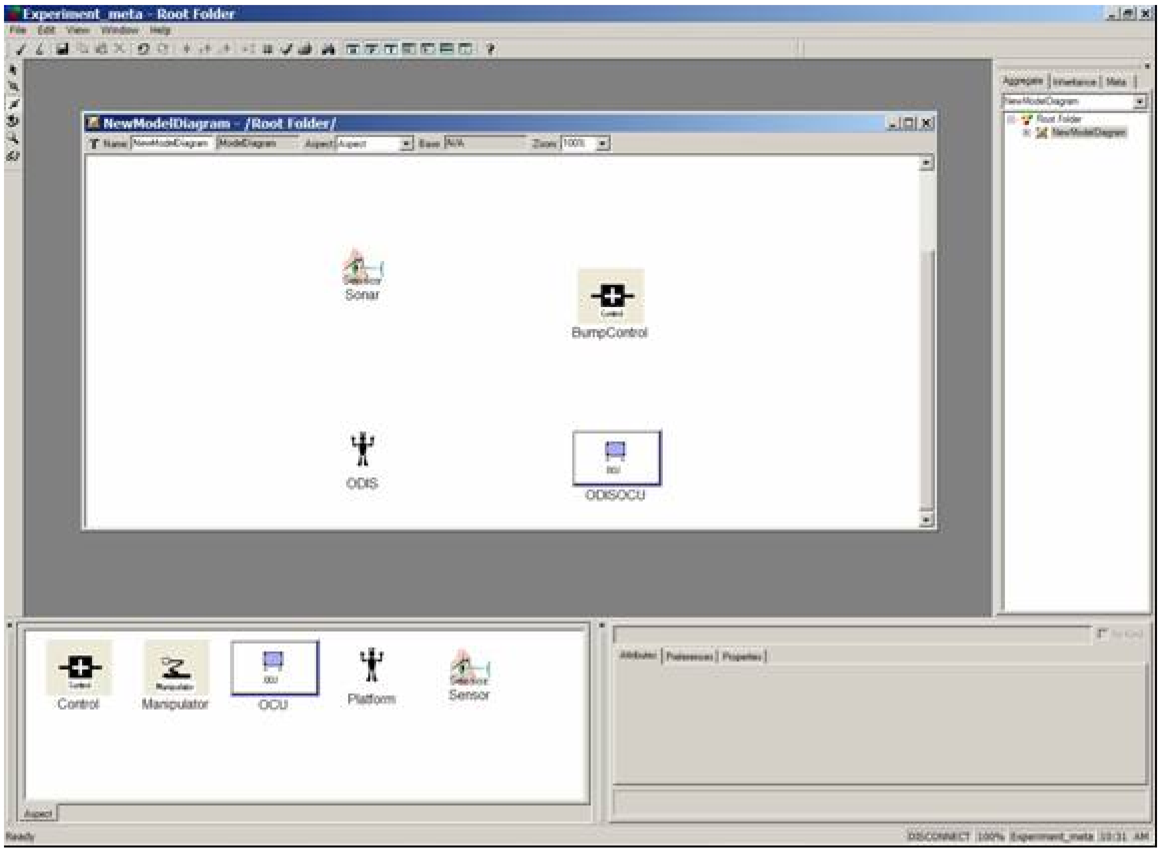


Figure 43. GME Screen Shot as Domain Engineer Selects Artifacts for the Domain Model

Examining the attributes views in **Figure 43** and **Figure 44** shows how some different decisions at Meta-modeling time show up in the domain model. (It also shows how GME windows can be undocked). In **Figure 43** the second line in the attributes window is a string attribute the domain engineer needs to fill in to fix the exact OCU that will be used in the subsequent code composition/generation. In **Figure 45** the user has configured the attribute, in this case "Sensor Name" to Sonar. Notice also in **Figure 45** that the name below the sensor icon has not changed. The "SensorName" attribute is a different element from the name attribute associated with the sensor element in the GME model. Also, note that the domain modeler has begun to customize the names on the top-level diagram by renaming the Platform icon to "ODIS."

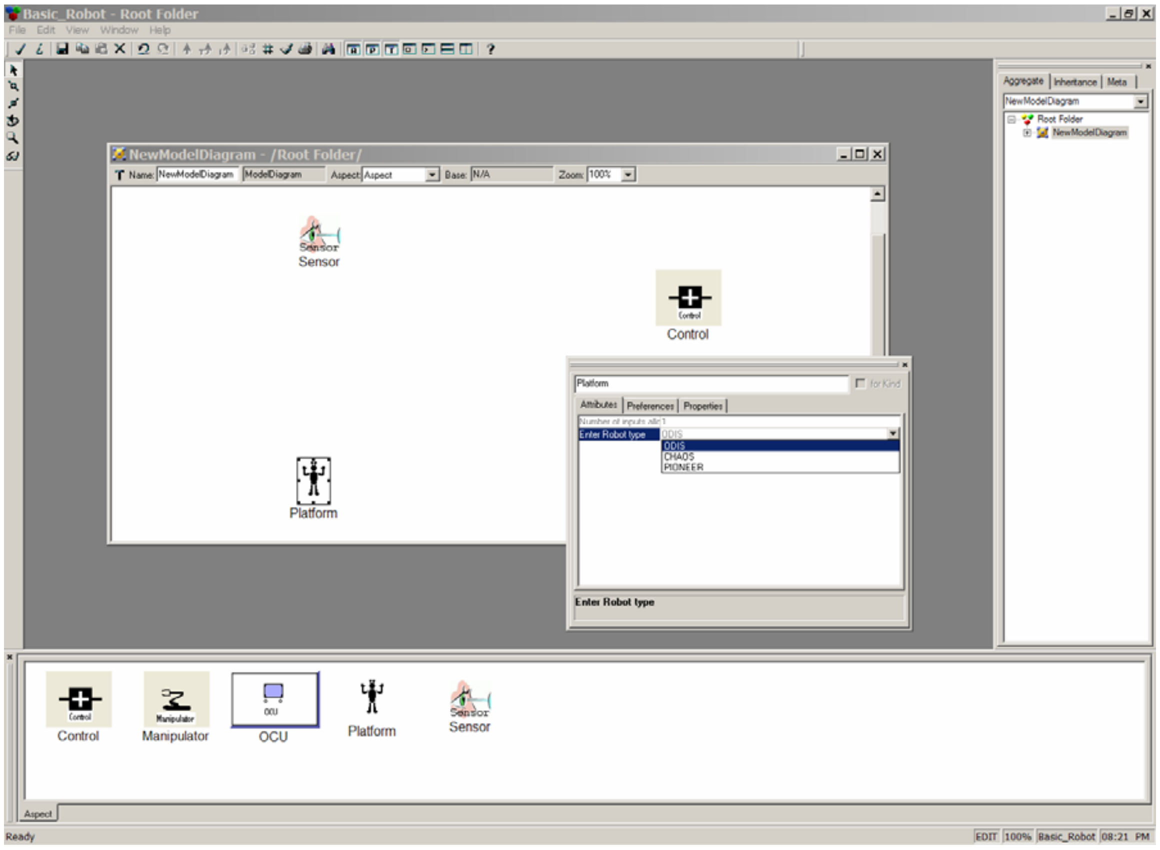


Figure 44. GME Screen Shot as Domain Engineer Selects Artifact Enumerated Attributes

The above discussion of the aspects of a model shows how Meta-modeling decisions affect the domain model. In the case of including a string variable, the domain model becomes very flexible. New components can be added at any time and the same paradigm can be used. In the case of using an enumerated aspect, the domain model is restricted to exactly the components a Meta-modeler allows. Which case to use is a matter of philosophy and the sophistication of the domain modeler. A hybrid case could be included to have an enumerated attribute along with a string attribute, that when filled in overrides the enumerated attribute.

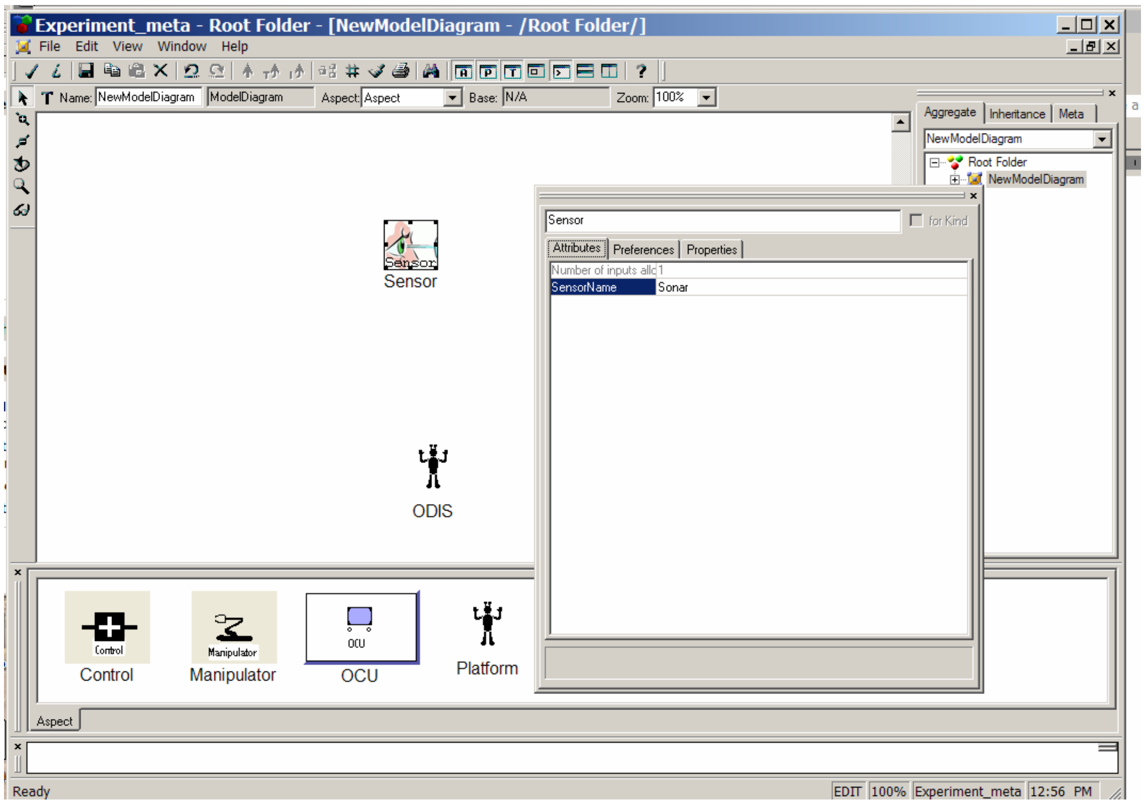
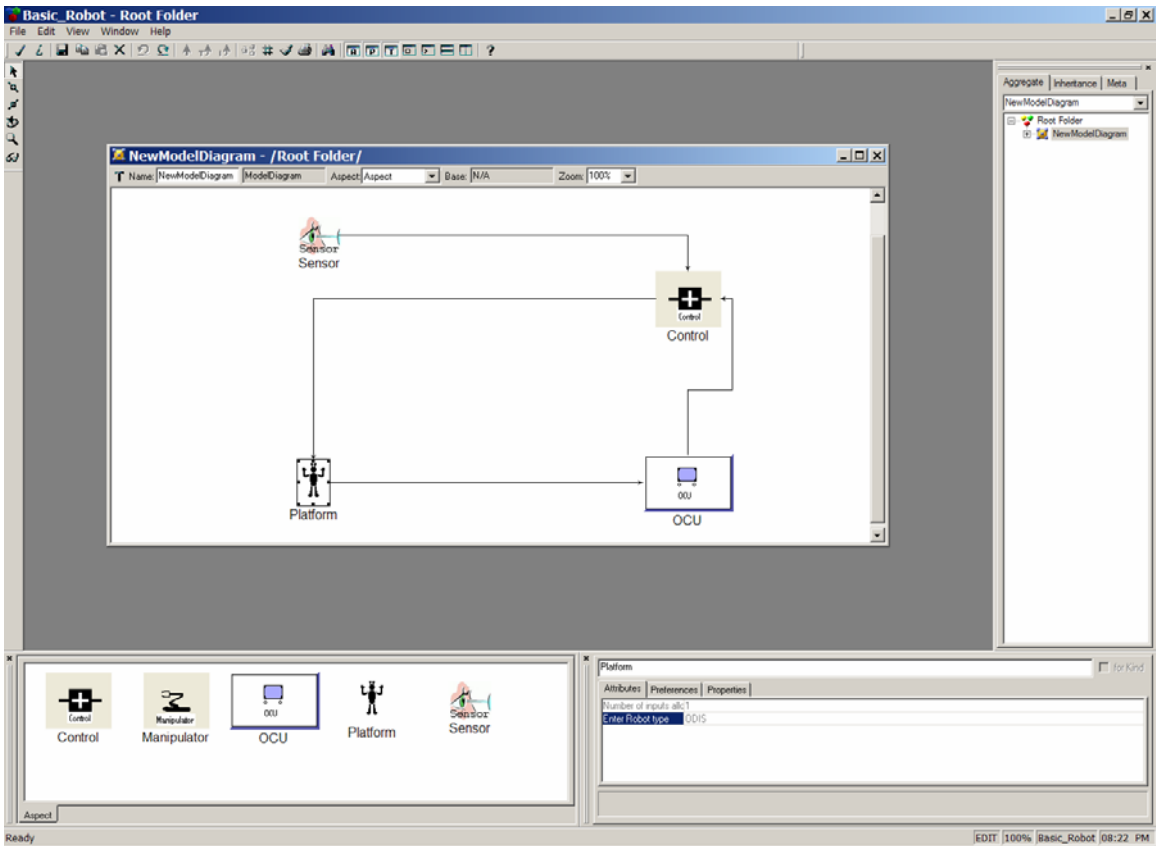


Figure 45. GME Screen Shot as Domain Engineer Inserts an Artifact Text Attribute

Additional concerns for Meta-modelers arise when they attempt to create rules to compose/generate the codes. In the Meta-model, of the top-level models, OCU, Platform, Sensor, Manipulator and Control inherit from a base class. Each of these models has an independent attribute to define which legacy item we are planning to include. This is of course necessary for enumerated items, but notice also that the attribute names are all unique: OCUID, PlatformType, SensorName, ManipulatorType and ControlID respectively. Again, at rule creation time, a common name, such as ItemID, will simplify the rule set needed to interpret the domain model.

Figure 46 shows an essentially complete top-level model. Essentially, there is a dataflow model or cooperating set of state machines, between the icons. The various legacy items are connected via a set of legal connection arrows that can transmit one or more message types, ideally any legal message in the message set. Since this is a prototyping environment, the case may be that some nodes may transmit messages that are not used by connected nodes. What happens then is controlled by the receiving node, and may be one of the reasons the prototype was constructed in the first place.


Basic_Robot - Root Folder
File Edit View Window Help
NewModelDiagram - /Root Folder/
Sensor
Control
Platform
OCU
Aggregate Inheritance Meta
Root Folder
NewModelDiagram
Control
Manipulator
OCU
Platform
Sensor
Aspect
Attributes Preferences Properties
for Kind
Ready
EDIT 100% Basic_Robot 08:22 PM

Name: NewModelDiagram ModelDiagram Aspect: Aspect Base: N/A Zoom: 100%

Platform

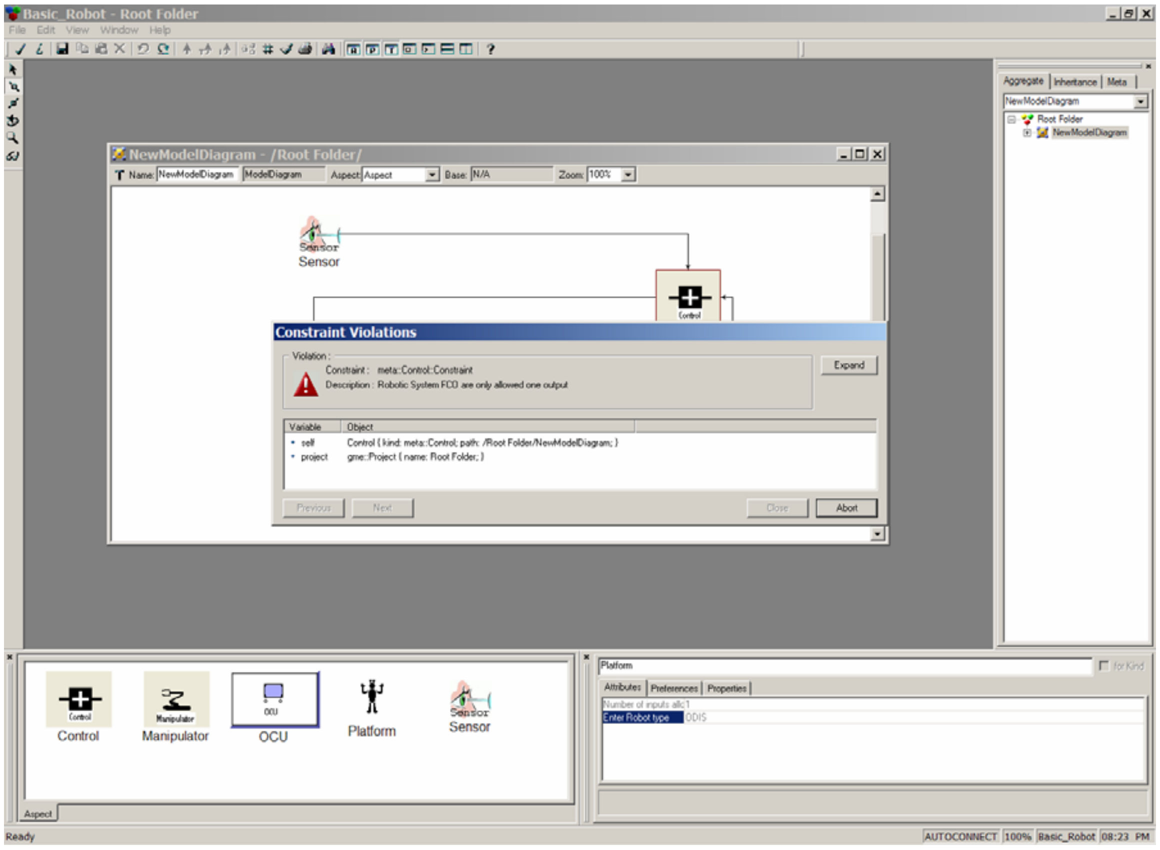
Number of inputs allc 1
Enter Robot type ODIS

Figure 46. GME Screen Shot as Domain Engineer Configures Message Passing

Returning to the model in **Figure 46**, the team has a prototype to examine using sonar for bump stop control. The "BumpControl" may be a simplistic algorithm that simply stops the robot when it is triggered, in which case researchers will probably determine quickly that they have a problem when they have to go down range and shove the robot away from the obstacle. In another case, the "BumpControl" may be a sophisticated non-deterministic algorithm, perhaps based on a neural network, where they want to exercise the scenario in a relevant environment to be reasonably sure it will not be stuck in actual operations. They can also recreate the model by changing out the robot to see if the control algorithm works with different types of mobility platforms.

The last illustration on the top-level model is what happens when we encounter a constraint. In the Meta-model, the software engineer imposed the constraint that no node in the domain model would be able to be the source of more than one message stream. **Figure 47** shows a pop-up message in the GME environment informing the domain modeler that attempting to source a second message stream from the OCU is a constraint violation. The modeler simply selects the only real choice, abort and is able to continue work. When the case is that the domain expert absolutely needs to have a node source two or more message streams, he will have to go back to the Meta-modeling team and have them create a new paradigm for this new case. Like fine wine, computer models come in distinct varieties and improve with age. Eventually, there will be a collection of well-defined paradigms covering several distinct cases, as well as a collection of ever-improving versions of a model.

The top-level model gives, obviously, a top-level view. Usually, researchers build prototypes because they are more interested in what is going on behind the scenes. They want to observe physical as well as hidden aspects that may be associated with the underlying messaging software, and to be able to change certain aspects, such as bandwidth, or be able to introduce faults. They want to include legacy items with a variety of protocols, and to be able to include emerging communications protocols as well as emerging communications devices. For that, they use lower-level models where the actual mechanics of the prototype are configured. In this example, the lower-level models are composed of atomic parts to simplify the discussion. For instance, the optional component is really a placeholder; in future instantiations of this paradigm, the team envisions that at least the optional component will become a model; it will have sub-components that will essentially define a mini-graphical programming environment.


Basic_Robot - Root Folder
File Edit View Window Help
NewModelDiagram - /Root Folder/
Name: NewModelDiagram ModelDiagram Aspect: Aspect Base: N/A Zoom: 100%
Sensor
Control
Expand
Previous
Next
Close
Abort
Aggregate
Inheritance
Meta
Root Folder
NewModelDiagram
Control
Manipulator
OCU
Platform
Sensor
Aspect
Platform
Attributes
Preferences
Properties
Ready
AUTOCONNECT 100% Basic_Robot 08:23 PM

Constraint Violations

Violation :
Constraint : meta::Control::Constraint
Description : Robotic System FCD are only allowed one output

Variable	Object
self	Control { kind: meta::Control; path: /Root Folder/NewModelDiagram; }
project	gme::Project { name: Root Folder; }

Number of inputs allo	1
Enter Robot type	ODIS

Figure 47. GME Screen Shot as Domain Engineer Attempts an Illegal Connection as Defined by Meta-model Constraint

Figure 48 shows a domain model with two sub-models open. The "BumpControl" model window is active and attributes for the "BumpControlAdaptor" are visible in the attributes window. The Meta-model is constructed such that the paradigm has a common set of component atoms for each artifact that can be included in the top-level model. This means the domain modeler will see the same set of parts in the aspect window (lower left) for any top-level model icon that is opened. The "ODIS" sub-model window is also open, and we can see the domain modeler picked and connected a different set of parts to compose that model.

In the "BumpControl" window, the modeler has defined an incoming serial connection, a bi-directional UDP connection, a "BumpControlAdaptor" and an optional component in the outbound data stream. For simplicity, he created this sub-model without constraints and without explicit connections to the connections in the top-level model. This could lead to ambiguities if the domain modeler has no idea what they are doing, so future production versions of the Meta-model will use additional GME concepts to make the model much more robust at domain modeling time.

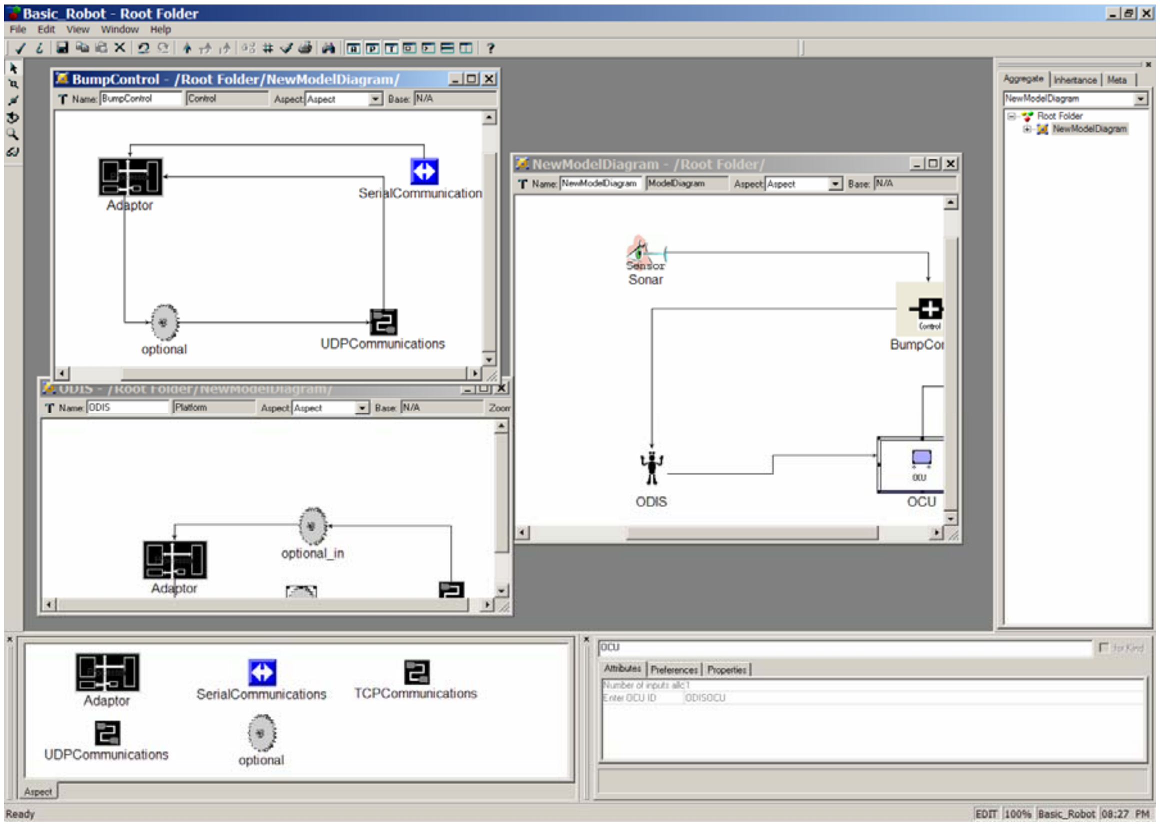


Figure 48. GME Screen Shot as Domain Engineer adds Adaptor, Communications and Optional Software Components that Configure the Artifact Wrappers

The actual adaptor that will be used can be determined by the “TypeID” of the parent model, the name of the adaptor, or from an adaptor artifact: again this is a Meta-modeling concern and should be addressed by the modeling organization’s Standard Operating Procedures. The optional components have considerable potential to expand the capabilities of these tools, especially since models can be copied and saved for reuse in this and other domain models.

The other two artifacts, “ODISOCU” and “Sonar” are configured as above. When all the artifacts’ sub-models are configured, the model is saved and is ready for use. To use the model to generate/compose code, it is next exported to an XML format file. The saved GME model can also be edited to create a new instance.

3. The XML Output from the Domain Model

The completed domain model is saved in XML format for portability. This domain model XML output is the next area of concern for the software engineer. The software engineer creates a set of rules to parse the output, which will allow additional rules to move files, setup the run-time environment and generate top-level code. This is done once, after creating a representative domain model. A properly constructed rule set will correctly interpret any domain model created from the Meta-model's paradigm. GME generates an XML file with an .xme extension to indicate that the XML is domain model output. A sample domain model XML output file, "Basic_Robot_Domain.xme"[*], is available in Appendix C.3.b. This file is used for the remainder of this chapter.

An XML document is a collection of elements. Elements are generally denoted by a start tag with an identifier string in angle brackets , and an end tag in angle brackets with the same identifier string followed immediately by a "/," or in the case of empty element, a single tag in angle brackets with the slash before the closing bracket. **Table 3** illustrates the basic ways an XML document element is constructed.

Note the different ways an element can convey information:

- An empty tag can be a place marker in the XML, or convey that some piece of information is missing, i.e., <author/> might mean that the next group of tags is about the author, or that the author did not fill in the field in the generating application.
- Information related to a tag's attributes is enclosed in double quotes.
- Textual information enclosed in tags is not quoted; if it is quoted, the quotes are part of the text string.

[*] The naming of the files used is somewhat arbitrary, however, file extensions are defined by the GME environment.

Description	Example
Empty XML Tag	<element/>
Simple XML Tag with text	<element> This is an XML Element. </element>
Empty XML Tag with attributes	<element name = "*value*" type = "*value2*" />
Simple XML Tag with text and attributes	<element name = "*value*" type = "*value2*"> This is an XML Element. </element>
Mixed XML Tag with text, attributes and inner element.	<element name = "*value*" type = "*value2*"> This is an XML Element. <element2 name = "*value3*"> This is the text in XML element2 </element>

Table 3. Basic XML Tag Concept for Rule Creation

The output XML file, Basic_Robot_Domain.xme is derived from the paradigm, Basic_Robot_Meta, which defines an allowable structure for combination of features in a Domain model (**Figure 49**). The Meta-model adds additional semantic information, attributes, and syntactic information, constraints, to the feature model. It follows that since the paradigm for the creation of the Domain model is based on a particular feature model, the result will be a domain model conforming to the original feature model. Thus, the high-level feature model carries through the process from Meta-modeling, through domain modeling to rule creation for composing/generating the run-time environment. This is a systems engineering process to enhance and define a series of models and artifacts by structured transformation from a requirement, to a feature model, formally modeled in UML, to a fully executable prototype composed of previously incompatible hardware and software artifacts.

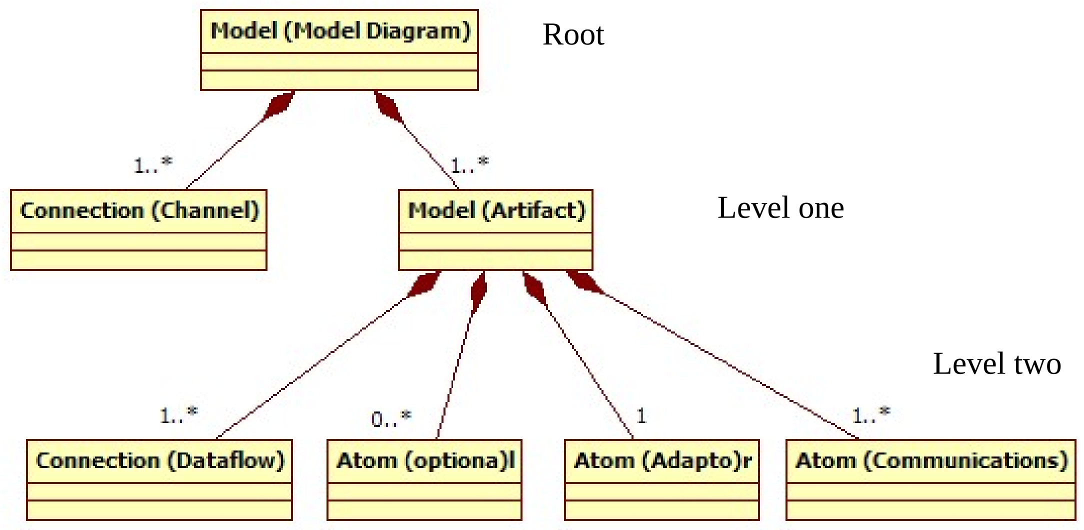


Figure 49. The "Basic_Robot" Feature Model in UML, Showing the Three Levels of the Model and the Cardinality of the Elements

4. Translation

a. Introduction

At every step, the prototyping environment is working with a representation of the basic feature model that spawned the paradigm. At every step, the user are manipulates the representation into a format that is best suited to the task. During translation, a set of Ruby coded rules use XPath extensions to Ruby to extract the information in the XML output of the domain model into a set of Ruby data structures. The Ruby rules are arraigned to parse the XML data and construct arrays and hash tables corresponding to the feature model. The final set of data structures constitutes a set of related knowledge.

b. Frame-Based Knowledge

The feature-model-based paradigm separates knowledge about the system into chunks of information. During parsing, related information; is stored together, in chunks. A frame is a model for a chunk of related information. A frame is divided into slots and each slot contains an element of the instance of the feature. The information in a slot may be instantiated directly during parsing or derived from other information by running a procedure (or firing a rule). Prior to instantiating derived information, the slot is associated with a procedure. A set of frames stores the knowledge defined in the system in a readily manipulated format. [59] Frames exhibit polymorphic behavior by collecting appropriate information, i.e., communications parameter frames will have common slots and specialized slots depending on the type of communication they represent.

Table 4 through **Table** 7 show the frames represented in hash table within the Ruby arrays created from the domain model XML output as well as cross references between object id's. Human understandable text names make the data structures more user friendly. For instance, the GME connections in the XML output only have the object ids of objects with which they are associated. Designers have written rules to search artifacts and component frames for their object ids and then written the corresponding object name to the associated communication frame. **Table 4** is a rendering of an artifact frame. All slots except *comms_destination* are directly associated with a model element in the GME domain model. One can derive comms_destination by searching channel frames for a channel with a source at the artifact associated with this frame, reading the destination id from the channel's destination slot, then searching other artifact frames for the corresponding id, and then finally filling in the comms_destination slot with the data in the found frame's name slot.

name	User Input in Domain Model
id	GME Defined
comms_destination	Derived
messages_in	User Input in Domain Model
messages_out	User Input in Domain Model

Table 4. Artifact Frame

Table 5 and **Table 6** represent communications frames. They have no name of their own because the GME Meta-model, Basic_Robot_Meta does not use names for connection.[*]

destination_name	Derived
destination	GME Defined
source_name	Derived
source	GME Defined

Table 5. Channel Frame

destination_name	Derived
destination	GME Defined
destination_kind	Derived
source_name	Derived
source	GME Defined
source_kind	GME Defined

Table 6. Dataflow Frame

Table 7 is a representation of a component frame. The component frames are composed of different slots based on the value associated with the kind of slot. The top three slots in each frame are the same, and could be considered a base. Alternative data structures are necessary because engineers modeled the components as a group of classes inheriting from a base class. It makes sense that there can be many different extensions of the base class by derived classes.

[*] GME has the capability but assigning identifiers to connections or not is a modeling decision.

name	User Input in Domain Model
kind	Adaptor – Defined in Meta-model
id	GME Defined

Or

name	User Input in Domain Model
kind	TCPCommunications – Defined in Meta-model
id	GME Defined
Port	User Input in Domain Model
IPAddress	User Input in Domain Model
destinationIPport	Derived
destinationIPaddress	Derived

Or

name	UserInput in Domain Model
kind	SerialCommunications – Defined in Meta-model
id	GME Defined
serialport	User Input in Domain Model
serialspeed	User Input in Domain Model

Table 7. Component Frames

c. Rule-base Program Discussion

Feature models are essentially a hierarchy of data items. This suggests a hierarchy of frames, discussed above, and rules. Rules are well suited for hierarchical description and manipulation of declarative knowledge. Software engineers transform declarative knowledge into a production rule by writing the knowledge in the form of a pattern and an action. Rules may call other rules.

One may ask, why choose a rule-based translation over a procedural translation of the graphical domain model? The answer relates to the fact, mentioned in the discussion of Feature Modeling, that there is a grammar associated with a feature model. This grammar suggests generic production rules to process a feature model. The grammar for a feature model specializes to specific instances of a feature model, i.e., domain models. Each specialization may spawn a family of domain-model grammars, with many common attributes and productions.

Rules are, by nature, highly uncoupled. A rule can have multiple results depending on the artifact that the rule operates on. The software engineer takes advantage of the nature of rules when creating a set of outcomes for the associated aspects of a feature model.

Rule languages are domain-specific, extensible, and high-level. This allows directly reusing higher-level rules as parts of the lower-level parts domain model change. Rules capture the semantics of the Meta-model, as well as the semantics of the domain model. Software engineers version previously created rules along with the Meta-model.

The ability to reuse directly or modify rules contributes greatly to the minimization of turnaround time for producing new or new versions of prototypes. In many cases, the software engineer that created the original rule set will still be available, but not always. In either case, the software engineer, working on the revision has a baseline.

d. Ruby

The link with the most subjective variability in this effort was the choice of a language to translate the domain model into an executable application. A high-level language was paramount. Prolog and Lisp stood out as languages with artificial intelligence roots to address rule bases. Perl and Python are scripting languages with needed XML support. All are portable, but different instantiations are not freely available for a wide variety of platforms.

While any of the above languages might have been used, TARDEC designers ultimately chose to use Ruby. Ruby is a freely available, portable language with a large open source community background. It is object-oriented, which makes it suitable for creating reusable, polymorphic objects needed to interpret and translate feature models expressed as GME domain models. Ruby is a loosely-typed language, which allows for simple translation from XML representations of domain models into frames internally represented as a collection of arrays and hash tables.

Ruby has required support for other necessary elements of the prototyping environment; REXML extensions provide XPath and XPointer support. Ruby supports regular expressions as a language feature. Ruby is open source. It has a large contributor community via Ruby Gems. Ruby Gems are another possible way to share Ruby rules created via this project.

E. RESULTS

This experiment exercises the research objectives from concept inception through executable java code. The experimental implementation begins with a software engineer's, in cooperation with a domain engineer, realization of a Meta-model for the proposed series of experiments using the GME tool. A GME function transforms the Meta-model is into a domain model.

During creation of the Meta-model, the software engineer queries the component repository to insure that the appropriate components are available in the repository. Missing or out of date components initiate a programming effort to create or maintain needed components as necessary.

The software engineer creates a set of rules that will translate any instance of the domain model into a set of executable programs necessary to operate a prototype. The rules also collect and compile documentation and instructions necessary to operate the aggregate prototype.

A UML depiction of activities associated with Meta-modeling in **Figure 50** shows the various actors, programs and data stores necessary to generate a domain model.

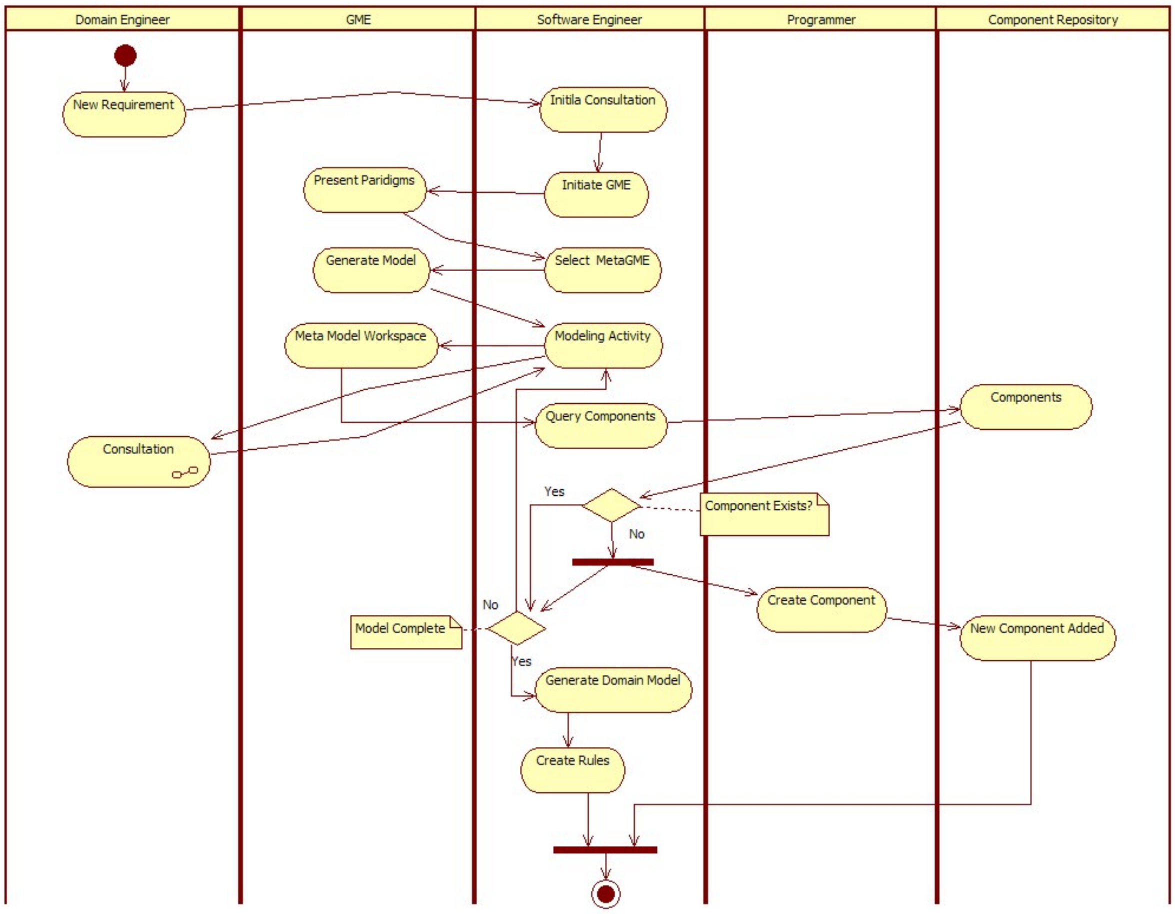


Figure 50. Activity Diagram Showing Meta-modeling and Associated Efforts Necessary to Create a Domain Model

A domain engineer creates a domain model, an instance of the Meta-model in GME. GME outputs the domain model to an XML output file. The domain engineer initiates the translation rule program that composes and generates Java code and an "operator's manual" necessary to run the prototype.

A UML depiction of activities associated with domain modeling in **Figure 51** shows actors, programs and data stores necessary to produce software needed operate a prototype.

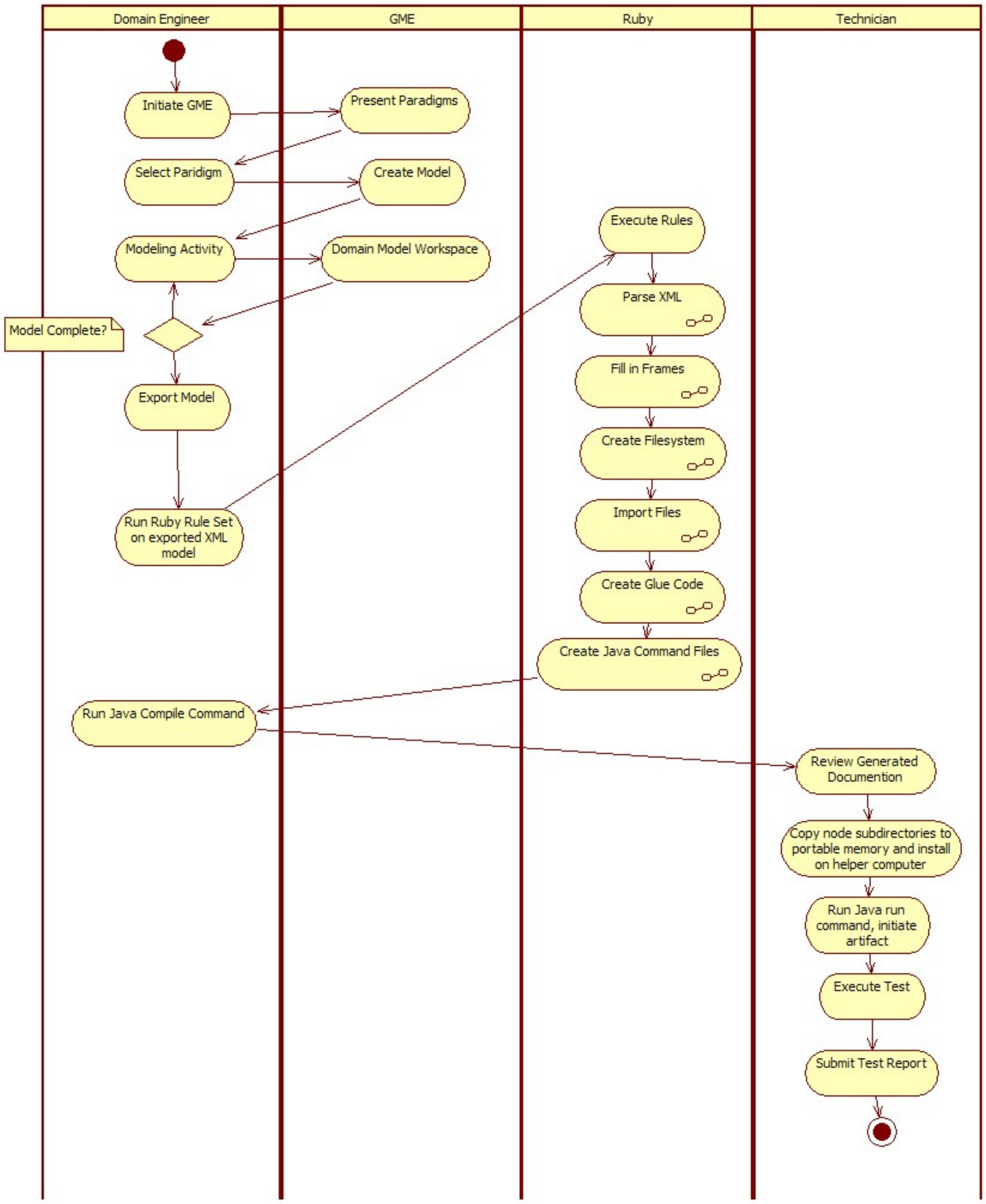


Figure 51. Activity Diagram Showing Domain Modeling and Associated Efforts Necessary to Create a Final Product

1. Software Engineering

The Generic Modeling Environment is a well-documented tool. It comes with a detailed user manual as well as a comprehensive set of tutorials and examples. Developing the "Basic_Robot_Meta" Meta-model was integral to this effort. Developing this model included gaining experience with GME via the tutorials, as well as several trial Meta-models. Approximately six iterations of the Meta-model contributed to an understanding of the necessary components and attributes that appear in the Basic_Robot GME paradigm. Creating the final paradigm used in this experiment was about a one-week part time effort.

After a Meta-model is complete, the software engineer creates a rule set to interpret instantiations of the resulting domain model. In this effort, creating the initial rule set was a two-part effort. The first part was establishing appropriate rules. The second part was coding these rules in Ruby. Both these effort took place over a period of about one year, interactively. This effort required creating several rule sets to correspond to the progression of Meta-models. The "Basic_Robot" rule set contains sixteen rules. Some of these rules are very similar to rules in the first rule set. Others appear only in the final set of rules. Creating rule sets is a function of experience and the size of the Meta-model. Larger Meta-models, with deeper tree structures, will require additional rules; however, the information extraction rules will be similar at each level. This indicates that the time required to create a rule set is not a linear function of the size of the Meta-model.

2. Coding Efforts

The main thrust of this effort is to reduce the time it takes to create a prototype system; in particular, the software needed to establish a federate of legacy hardware and software artifacts. Technicians and users then exercise the federate in a test environment as needed.

It was a several month, part time, effort to create the initial set of software components needed to exercise the system. Counting the hours involved exactly, proved somewhat difficult due to the multitasking nature of the work in the Robotics Lab. Observations of work levels indicate that there is at least forty man-hours associated with creating first two adaptors. Creating the UDPCommunications component was about a twenty-man-hour effort and creating the optional component was about a ten-man-hour effort.

The first two adaptors interface to two legacy simulation items, a simulation of the ODIS robot and a simulation of an Operator Control Unit (OCU). Originally, these two artifacts ran together on a single computer, communicating over the localhost address, 127.0.0.1. The two artifacts communicate using messaging packets designed in an ad hoc fashion, but influenced by the ODIS robot messaging packet scheme. The two adaptors allow the simulation and OCU to be run on different machines and allow insertion of instrumentation in the messaging stream. The adaptors convert the legacy packet format to and from XML JAUS messages. The communications components convert the XML JAUS messages to binary JAUS messages for transmission. The messaging code to convert between JAUS formats and from JAUS XML to legacy format is from the Naval Postgraduate School's Autonomous Underwater Vehicle (AUV) Workbench code distribution [60] written by D. Davis and used unmodified.

The adaptor components are about 400 lines of code each, the simple optional component is about 100 lines of code and the UDP communications component is about 400 lines of code. Components once created once are reused in multiple domain instantiations. The Ruby rule set is about 500 lines of code and is highly reusable, not only in its current application, but individual rules are also reusable for future rule sets.

3. Creating and Operating a Prototype System

The time it took it create the communications, adaptors and optional components and get the first test case running is representative of the previous ad hoc methodology for creating prototypes. If new team members come on board, the software may not be available, or if it is, not understandable. This is particularly true when the replacements are not software centric.

To test the environment, TARDEC engineers took the components for the first prototype implemented, the simulation artifacts, placed them in the component repository, and created a Meta-model that incorporates them as elements.

Two Interns participated in the second phase of the experiment. One is a computer scientist with two years experience in the Robotics Lab. The other is a mechanical engineer with two weeks experience. Both attended a 45-minute briefing on the purpose and scope of the prototyping environment, including Meta-modeling, domain modeling, an overview of the rule set and instructions on how to generate a set of wrappers.

For the experiment, the two Interns used two computers. The first computer ran the GME program, Ruby and the Component Repository, as well as the OCU simulation. The second computer ran the robot simulation. The first to use the environment, the computer scientist, took 25 minutes to get the simulation operational on two computers. The second completed a very similar task in 20 minutes. Note that the participant played the roles of both domain engineer and technician.

This effort is representative of the potential time and effort savings. The TARDEC Robotics lab intends to use these same simulations, along with control artifacts and physical OCU artifacts to examine behavioral control algorithms. Programmers in the TARDEC Robotics lab are creating adaptors for low-cost hand held game controllers. Programmers are also creating adaptors for sensors and behavior controls to examine the feasibility of orbiting an omni-directional drive vehicle within six inches of a stationary object, marinating a predefined relationship normal to the surface of the stationary object.

F. EXPERIMENT CONCLUSION

This approach will reduce time and effort needed for hardware software prototyping by reducing rework. Adaptors for legacy systems will be stored in the component repository. Highly reusable components, such as instrumentation and communications will eliminate time-consuming recoding (often by inexperienced personnel).

The process taken by this effort generates work products at each step of the process. The process generates Meta-models, domain models and corresponding rule sets. These process documents will feed forward into production efforts. Production engineers will have solid documentation of assumption and constraints used in prototype efforts.

These documents are usually not readily available today. Coupled with test reports, they will have a better understanding of issues and requirements.

This process in not currently used at TARDEC, due to the immaturity of the component repository. An effort to add adaptor components for each of the platforms in the TARDEC Robotics Lab inventory is under way. Lab engineers are assembling several stand-alone sensor packages, including an Inertial Navigation System (INS), a scanning LIDAR range sensor and a sonar array; adaptor components will follow.   A new project focusing on current operations is in the planning stages. New sensor and mobility platforms will be entering the inventory. TARDEC plans to include this process during concept and sensor evaluation.

THIS PAGE INTENTIONALLY LEFT BLANK

IX. CONCLUSIONS AND RECOMMENDATIONS

A. CONCLUSIONS

This work provides framework for a systems engineering approach to prototyping robotic systems. It provides tools to an interdisciplinary team, to create a set of work products necessary to create an executable prototype robotic system. The Meta-model work product is directly applicable as an input to a production systems engineering effort.

There are immediate and compelling reasons to invest effort into prototype efforts for robotic systems. There are current needs that are addressable by technology that approaches a viable Technology Readiness Level (TRL). Understanding of operational and logistical issues is paramount to developing technology to operationally acceptable TRL.

This work takes a graphical, model-driven approach. The work defines a chain of responsibilities to take advantage of expertise from different technical fields. The actors in the chain generate a set of reusable models, components, rules and documentation.

B. RECOMMENDATIONS FOR FUTURE WORK

1. Introduction

The next step is implementation of the design environment for prototyping a series of robotic systems in the TARDEC Robotics System Integration Laboratory (SIL). In-house researchers will begin with simple models, robot simulations and very coarse-grained components; the simple models presented earlier will be realized. Continuing, the Meta-models will be refined to include lower level component composition. A set of robotic artifacts (platforms, controls, OCUs etc.) will have their interfaces wrapped to conform to the JAUS standard. TARDEC research engineers will create a collection of instrumentation components, as well as several different communications components - UDP/IP, TCP/IP and serial to begin with.

As the team grows more confident with the Meta-models and domain-specific models, additional artifacts such as mission packages and manipulators will be included both in simulation and physically. This work provides a framework for prototyping robotic systems. There continue, however, to be open-ended concerns to be addressed, at least initially by the TARDEC Intelligent Ground Systems Lab.

2. Component Repository

A robust component repository is a prerequisite to this effort. Initial efforts will focus on Java components. New artifacts require new adaptor components, and new communications protocols require supporting communications components. Revisions to the messaging protocols will also require new versions of message objects.

A new version of a messaging protocol will trigger a maintenance event. Incorporation of the new message class potentially requires versioning of adaptor components. Inspection of adaptor components is necessary to determine if a versioned message affects a particular component.

Maintaining the component repository is an open-ended effort. As time goes on a series of versions of components will emerge. Developing cataloging schemes to keep track of what versions of the components are compatible is an interesting research topic. Methods to search the component repository “intelligently” are another potential research topic.

In addition, there is no reason that components based on other languages could not be included in the future. Rules to deal with conflicts and precedence are necessary, assuming components with different language roots, but the same functions appear in the component repository.

This effort will support at a minimum, team efforts. Team efforts imply shared repositories. An in-house effort has relatively simple requirements, perhaps a mutual file server. Including outside interests requires networked distributions. The situation is not complex as long as the data flow is outward bound. It becomes more complex with community support, but still reasonable. Of course, if proprietary components or intellectual property is involved, the situation becomes more complex.

Database support addresses many concerns when multiple levels of access are required. Database support also imposes administrative requirements, as well as security concerns, hosting requirements and database middleware support. Determining these requirements and suggesting solutions may also be a research topic.

3. Model Collections

Completed models are a product of this process. Meta-models define architectures, and domain models define instances of a Meta-model. To facilitate future model reuse, researchers should consider a searchable model repository.

In a relational repository, domain models are children of the Meta-model that generated them. Versions of similar Meta-models are children of an abstract parent Meta-model. Abstract Meta-models are themselves children of a root element.

Each model element should have auxiliary information associated with it. Meta-models might have requirements documents. Domain models might have sets of physical test results. Each Meta-model should have, at a minimum, a pointer to its associated rule set.

4. Rules

Rules extract information from the domain model and compose sets of code. In the experimental implementation, the rules are straightforward. Software engineers hand-assemble a relatively small set of declarative rules. Other than specifying the location of the XML file containing the domain model output, the rules are not interactive.

Rules, however, *can* be interactive. In the experiment rule set, there is a rule to check serial port compatibility. The rule only determines if the ports configured in the domain modeling process are compatible and outputs an appropriate message. A design decision may allow an additional rule that prompts the user for corrected information when the ports are incompatible.

The experiment rule set suggests that raising the level of abstraction of the rule set to a domain-specific rule language is a distinct possibility. Iteration rules have similar structure as do code generation and composition rules. Parameterized versions of these rules may simplify rule creation.

The frame-based nature of the data generated during domain modeling suggests using frame based expert system paradigms. Previous work in the Ruby community has produced Ruby-based versions of Prolog [61] [62] as well as favorably compared Ruby to Lisp [63]. This suggests that a Ruby-based inference engine is well within the realm of possibility.

5. Operational Environments and Run-time Environments

Initial experiments rely on available hardware platforms, mainly surplus laptop computers, PC104 computers and Velcro. The initial experiments planned at TARDEC include building control and sensor suites for small robots. The experiments will examine necessary maneuvers to position mission packages precisely with respect to an item of interest.

Surplus equipment is adequate for initial experiments in a controlled laboratory environment, but future experiments will require a more robust solution for communication with legacy artifacts in harsh field environments. This is particularly true when working with smaller man-portable platforms and their associated power and weight capacity limits. A variety of small run-time platforms are candidates to act as helper computers, including Sun Microsystems SunSPOTs [64].

Considerations for run-time helper computers include power consumption, support for wireless networking, I/O ports, J2ME [65] support, packaging and environmental specifications. A TARDEC engineers should consider a product survey as a candidate project for a TARDEC engineering co-op work rotation.

6. Optional Component as Compositions

The experimental implementation only defines one optional component. Additional optional components are necessary for instrumentation, fault insertion, bandwidth limiting, assertion checking or any number of other tasks. Currently, optional components are atomic. Future uses for optional components may require custom configurations; this will require engineers to model optional components as GME models, with a selection of sub-components to choose from. An example is an optional component that logs XML messages dependent on one or more attributes. The optional component would have a "child" modeling level to allow the domain modeler to select conditions and actions, such as a loop component or an if-then component.

7. Integration with Other Modeling Environments

As mentioned, the experimental implementation uses the GME tool for many good reasons. The GME tool is not, however, the only modeling tool available. There are proprietary and other open source tools for Meta-modeling. Of particular interest is the Eclipse environment due to its use in other TARDEC projects and Rational Rose, for the same reason.

8. Education Outreach

Future efforts in education have many facets from K-12 to professional education.

National objectives for Science and Technology Education [66] mandate that TARDEC and other DoD researchers perform K-12 outreach in a variety of ways, from school tours, to Engineer-in-the-Classroom programs to after-hours technology programs for high school students held at government facilities. The "First" [67] robotics program and local experience has proven that robotics programs capture and hold students' attention.

Local experience at TARDEC includes working with high school students. Early efforts used Motorola 6805-based microcontrollers and small hand built mobility platforms. Later efforts included Lego Mindstorms™ sets. The Lego Mindstorms™ produced small successes earlier. Early success caused the students to engage and explore more difficult concepts sooner that they might otherwise have done. Many of the students moved on to engineering curriculums in college. Several students pursued Cooperative Education opportunities with TARDEC. They are now staff engineers.

A possible use of the tools and processes developed in this effort is to tailor the process to the K-12 community. Meta-modeling may be beyond younger student's capabilities, but domain modeling and experiment development are certainly possibilities. Students can participate in real experiments using in-house assets.

The next aspect is formal education in engineering. This project presents potential for several aspects of engineering. One is to create a unit on architectural modeling for a specific set of requirements. Another is to examine the effect of structured vs. ad-hoc processes. Continuing, the process described in this dissertation may be a useful tool to remove some of the rote coding required in term projects in robotics and let the students focus on the relationships between premise and outcome. Looking from different perspectives, creating components might be a useful coding exercise, or creating interface specifications for adaptors might be a useful exercise for a software engineering unit.

Lastly, outreach should entail dissemination of concepts created in this effort to members of the immediate DoD robotics community, including DoD and commercial partners. This is a two-level effort. The first is an informational campaign, through presentations and demonstrations, targeted at the immediate community, the JAUS Working Group, the JGRE coordinator and member organizations as well as professional organizations, such as the Association of Unmanned Vehicle Systems International (AUVSI). The second level will include tutorials, examples and a reference distribution. A publicly releasable CD or open source web sites are candidates to make this a reality.

C. LIMITATIONS

As noted earlier, this project is just beginning. There are gaps in the environment capabilities.

The component repository is sparsely populated. The programmers populating the component repository must be sensitive to emerging needs. The component repository is local to the modeler's workstation.

The environment requires software engineers with modeling experience. New engineers do not have significant exposure to software modeling at University. In house On-the-Job Training (OJT) programs are necessary.

Educating and training domain engineers is also necessary. This is a new process, and virtually every new process meets with resistance.

The process is unproven. TARDEC plans several small evaluation projects that will lead to a planned new project. Success with these efforts will demonstrate the utility of the process.

The process is incomplete. There is currently no facility to store physical test plans and results. An in-house effort is under way to identify and characterize the currently available TARDEC and Army data storage assets. An extension to the project will create an experimental results repository.

THIS PAGE INTENTIONALLY LEFT BLANK

APPENDIX A. ABBREVIATIONS AND ACRONYMS

Shorthand	Meaning
AADL	Architecture Analysis and Design Language
AS	Aerospace Standard
AUV	Autonomous Underwater Vehicle
AUVSI	Association of Unmanned Vehicle Systems International
BNF	Backus–Naur Form
CENTCOM	Central Command
CHP	California Highway Patrol
DARPA	Defense Advanced Research Projects Agency
DIS	Distributed Interactive Simulation
DREN	Defense Research and Engineering Network
DSML	Domain-specific Modeling Language
EOD	Explosive Ordnance Disposal
ESQL	Embedded System Control Language
FA	Functional Agent
FCO	First Class Object
FCO	First Class Object
FCS	Future Combat System
GME	Generic Modeling Environment
IED	Improvised Explosive Device
IR	Infrared
IXO	Information Exploitation Office
JAUS	Joint Architecture for Unmanned Systems
JAXB	Java Architecture for XML Binding
JGRE	Joint Ground Robotics Enterprise
KDS	Kuchera Defense Systems
KS	Knowledge Store
LIDAR	Light-Imaging Detection and Ranging
NAFIPS	North American Fuzzy Information Processing Society
NQC	Not Quite C
OCL	Object Constraint Language
OCL	Object Constraint Language
OCU	Operator Control Unit
ODIS	Omni-Directional Inspection System represented by robot vehicle used at Traffic Control Point
OJT	On-the-Job Training
OMG	Object management Group
ONS	Operational Need Statement
POLA	Port of Los Angeles
POLB	Port of Long Beach
QOS	Quality of Service

RCX	LEGO Control Brick
REXML	Ruby XML Processor
RF	Radio Frequency
RS-JPO	Robotic Systems Joint Program Office
SAE	Society of Automotive Engineers
SIL	System Integration Laboratory
SMART	The Army’s Simulation and Modeling for Acquisition, Requirements and Training
SPIE	International Society for Optical Engineering
STANAG	NATO Standardization Agreement
TACOM	Tank-automotive and Armaments Command
TARDEC	Army Tank Automotive Research, Development and Engineering Center
TCP	Traffic Control Point
TLD	Total Level Detector
TRL	Technology Readiness Level
TSA	Transportation Security Agency
TTP	Tactics, Techniques and Procedures
TTS	Tactics, Techniques and Procedures
TVWS	Track Vehicle Work Station
UAV	Unmanned Aerial Vehicle
UDP	User Datagram Protocol
UGV	Unmanned ground vehicle
UGV	Unmanned Ground Vehicle
UML	Unified Modeling Language
UML	Unified Modeling Language
UNS	Urgent Need Statement
USU	Utah State University
XMI	XML Metadata Interchange

APPENDIX B. JOINT ARCHITECTURE FOR UNMANNED SYSTEMS (JAUS)

A. INTRODUCTION

The Robotic Systems Joint Program Office (RS-JPO) at Redstone Arsenal initiated JAUS in the mid 1990's to address shortcomings in Robotic System Development. In particular, there was a concern that the robotic development efforts were uncoordinated and proprietary. Robotic systems were unique, lessons learned difficult to transfer between systems and technology insertion slow. A challenge unique to DoD also existed, working within acquisition guidelines.

The JAUS effort embraced an evolutionary acquisition approach; field affordable technology now, insert new technology as it becomes relevant and affordable. JAUS addresses systems engineering issues, including life cycle cost, schedules, risk management, performance based requirements and open standards.

Government engineers originally led the JAUS development team. Other team members consisted of volunteers from industry and academia. Having a government lead violated DoD's shift away from MIL Standards to Industry Standards. About 2003, the JAUS committee announced that the Society of Automotive Engineers (SAE) agreed to adopt the JAUS effort as Aerospace Standard AS-4. At the current time, the first SAE standards documents are beginning to mature. The JAUS Working Group coexists with the SAE AS-4 committee.

The JAUS documentation is widely available at several Internet accessible sites, the primary being http://www.jauswg.org, [35], however, due to the importance of JAUS to ground robotics, a brief overview is appropriate here.

B. ELEMENTS OF JAUS

JAUS is a collection of six documents; the two main documents are the Domain Model, and the Reference Architecture. The remaining documents are the Standard

Operating Procedure, the Document Control Plan, the Strategic Plan and a draft Compliance Plan. This appendix only discusses the Domain Model and Reference Architecture.

1. Domain Model

The domain model is the user requirements model (**Figure 52**). The domain model encapsulates similar functions into groups called Functional Agents (FA). There are six FAs: Command, Tele-Communications, Mobility, Payloads, Maintenance and Training. The domain model encapsulates similar knowledge into groups called Knowledge Stores (KS). There are four KSs: Vehicle Status, World Map, Library and Log. The domain model also encapsulates similar devices into groups called Device Groups. Each FA and KS has a set of capabilities and services.

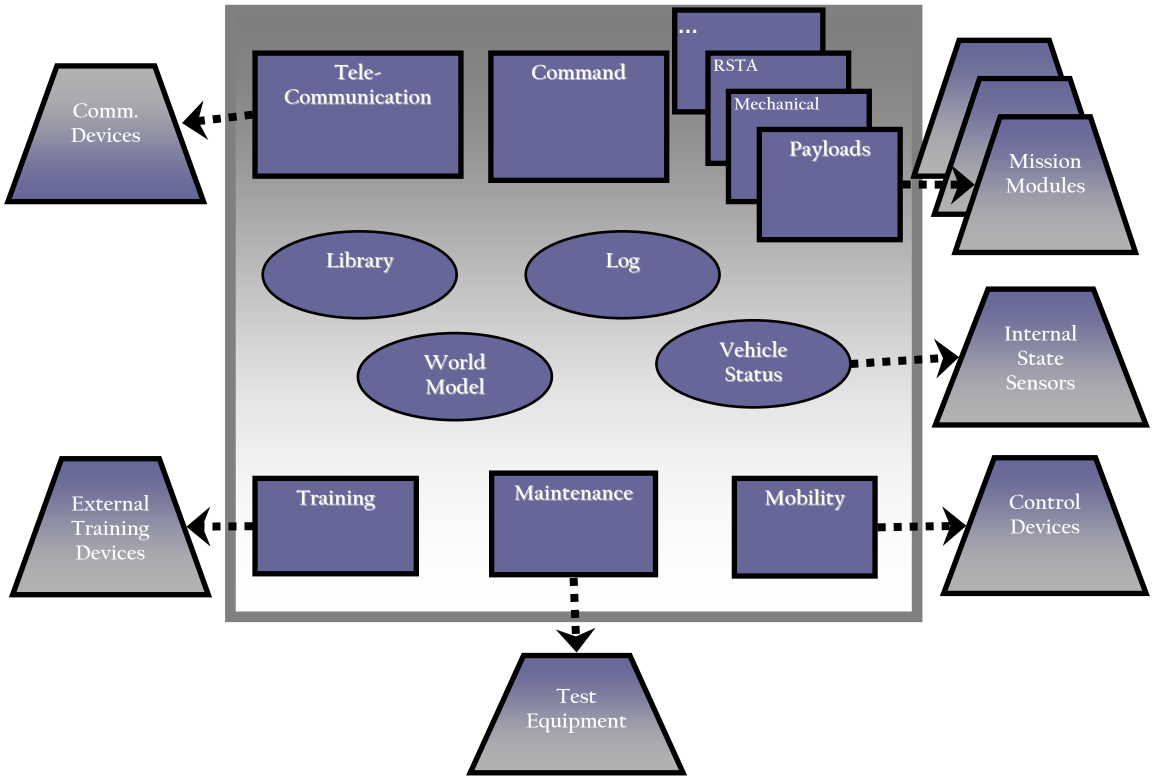


Figure 52. JAUS Domain Model, Model Elements are Enclosed Within the Shaded Rectangle. Functional Agents are Internal Rectangles, while Knowledge Stores are Ovals [From Ref. [35]]

2. Reference Architecture

Within the JAUS reference architecture, a system is composed of a set of operational subsystems, such as, operator control unit, sensors and a mobility platform. Each subsystem has one or more processing nodes; each processing node has one and only one Message Routing Service (MRS). A processing node contains one or more components (**Figure 53**).

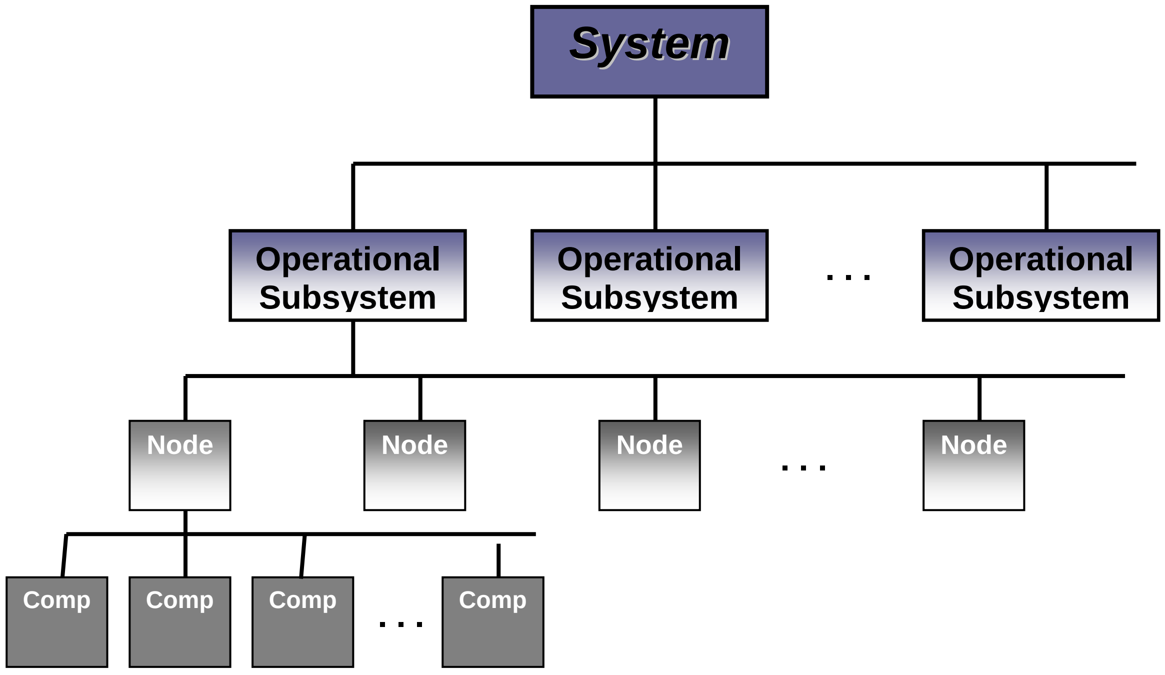


Figure 53. JAUS Reference Architecture Physical Topology [From Ref. [35]]

The JAUS system level refers to a group of artifacts working together. A system may be as simple as and OCU and a mobility platform. On the other hand, the upper level is unbounded; a system may include multiple platforms, multiple communications elements and a collection of control and monitoring stations.

A JAUS subsystem refers to a functional aspect of a system. JAUS subsystems may be composed of distinct items, such as an OCU or a platform, or a subsystem may be a collection of distinct items, a swarm of robots for example.

A JAUS node refers to a “black box” item. A node contains all the hardware and software needed to perform its function. Node examples include controllers, knowledge stores and sensor processors. From a hardware perspective, a node is an entity on the network, although it may contain several internal computing devices. From a software perspective, a node is the software that runs on the node.

A JAUS component is the lowest level of JAUS decomposition. Components perform specific operations. Components communicate via JAUS messages, their interface to other components, and higher-level aspects of the JAUS system.

a. JAUS Components

JAUS currently has five component groups:

Command and control components (**Table 8**) provide a mechanism for integration at the system and sub-system level. Command and control components may send and receive any message to and from any component.

Name	**ID**	**Function**
System Commander	40	Coordinates all activity within a given system.
Subsystem Commander	32	Coordinates all activity within a given subsystem.

Table 8. JAUS Command and Control Components Necessary for System Integration

Communications components maintain (**Table 9**) data links. There is one defined communication component.

Name	**ID**	**Function**
Communicator	35	Maintains all data links to other subsystems.

Table 9. JAUS Communications Component Maintains Data Links

Platform components (**Table 10**) report states in state for a platform, and effect change in state for a platform. All platforms are points in a six degree of freedom space. Driver commands use wrench messages. Wrench messages have separate fields for each of the three linear and three rotational dimensions for both propulsive and resistive effort. Propulsive efforts range from -100% to +100% effort (throttle). Resistive efforts range from 0 to +100% (braking).

Name	**ID**	**Function**
Global Pose Sensor	38	Report the global position and orientation.
Local Pose Sensor	41	Report the local position and orientation.
Velocity State Sensor	42	Report the instantaneous velocity.
Primitive Driver	33	Basic driving, mobility and platform device control.
Reflexive Driver	43	Modify a commanded effort to insure safety or stability.
Global Vector Driver	34	Closed loop control of the desired global heading, altitude and speed.
Local Vector Driver	44	Closed loop control of the desired local heading, pitch, roll and speed.
Global Waypoint Driver	45	Determine the desired wrench of the platform given the desired waypoint(s), travel speed, current platform pose and current velocity state.
Local Waypoint Driver	46	Determine the desired wrench of the platform given the desired waypoint(s), travel speed, current platform pose and current velocity state.
Global Path Segment Driver	47	Perform closed loop control of position and velocity along a path.
Local Path Segment Driver	48	Perform closed loop control of position and velocity along a path.

Table 10. JAUS Platform Components Report on and Control Platform State

Manipulator components (**Table 11**) support both low open and closed loop control of a robotic arm. Coordinate systems are either a global coordinate system, a coordinate system that is relative to the base of the vehicle, or a coordinate system relative to a manipulator (for the case of a manipulator attached to a manipulator).

Name	ID	Function
Primitive Manipulator	49	Remote operation (open-loop control) of a single manipulator system
Manipulator Joint Position Sensor	51	Report the values of manipulator joint parameters
Manipulator Joint Velocity Sensor	52	Report the values of instantaneous joint velocities.
Manipulator Joint Positions Driver	54	Closed-loop joint position control.
Manipulator End-Effectors Pose Driver	55	Closed-loop position and orientation control of the end-effectors.
Manipulator Joint Velocities Driver	56	Closed-loop joint velocity control.
Manipulator End-Effectors Velocity State Driver	57	Closed-loop velocity control of the end effectors.
Manipulator Joint Move Driver	58	Closed-loop joint level control of the manipulator where motion parameters for each joint are specified.
Manipulator End-Effectors Discrete Pose Driver	59	Closed-loop control of the end-effectors pose through a series of specified positions and orientations.

Table 11. JAUS Manipulator Components for Command and Control of a Robotic Arm

Environmental sensor components interact with the environment surrounding the platform (**Table 12**).

Name	ID	Function
Visual Sensor	37	Controls the camera(s) of a subsystem.
Range Sensor	50	Reports range data for the purpose of object detection.

Table 12. JAUS Environmental Sensor Components to interact with the Platform Environment

b. JAUS Messages

JAUS has multiple methods for messaging in the Reference Architecture. Command messages are one way; a response is not necessarily required. A query message requires a response; messages transfer data to queues. A service connection is a specific periodic dataflow; data arriving at a service connection is not queued. Thus, JAUS supports clock driven and event driven as well as hybrid architectures.

JAUS message conform to a set of standards. Knowledge of these standards is especially important when converting between JAUS and actuator or sensor application specific formats.

Textual data uses Latin-1 ISO/IEC 8859 Latin-1 standard character set. Fixed length strings do not require a terminator; the length of the string is its declared length. Variable length strings will use the NUL character as a terminator.

Numeric data representation is shown in **Table 13**:

Data Type	**Size (in Bytes)**	**Representation**
Byte	1	8 bit unsigned integer
Short Integer	2	16 bit signed integer
Integer	4	32 bit signed integer
Long Integer	8	64 bit signed integer
Unsigned Short Integer	2	16 bit unsigned integer
Unsigned Integer	4	32 bit unsigned integer
Unsigned Long Integer	8	64 bit unsigned integer
Float	4	IEEE 32 bit floating point number
Long Float	8	IEEE 64 bit floating point number

Table 13. JAUS Numerical Data Types [From Ref. [35]]

Byte order is *"Little Endian"*. Data streams shall transmit the least significant byte first. Additional standards define platform orientation and Manipulator Linkage Notation.

JAUS defines six message classes. The messages are grouped by "command code". JAUS defines the format of each message.

"Command class" messages effect system mode changes, actuation control, alter the state of a component or subsystem, in other words, initiate some type of action.

"Query class" messages solicit information from another component. Query class messages require an inform class message in reply.

"Inform class" messages transmit information between components.

"Event setup class" messages setup the parameters for an Event Notification message and have a component start monitoring for the trigger event.

"Event notification class" messages communicate the occurrence of an event. Events may be unsolicited, as in the case of exceeding an environmental parameter.

"Node Management class" messages transmit node specific information. Node Specific communications includes configuration information and component registration.

C. SUMMARY

The JAUS is the primary DoD standard for Ground vehicles. It is required in many DoD ground robotic acquisition efforts. JAUS has wide community support, as is evidenced by the membership in the JAUS/AS-4 working groups. The two will proceed in parallel until the SAE AS-4 Unmanned Systems Standard has matured. JAUS will exist as long as legacy acquisition efforts require it.

APPENDIX C. GENERIC MODELING ENVIRONMENT EXPERIMENT XML FILES

A. INTRODUCTION

GME supports several storage formats, a proprietary fast binary format, relational database and XML import and export for Meta and domain models. XML Meta-model representation facilitates tool extension or integration with other tools. It also provides a readable version of the stored Meta-model. The GME designates the Meta-model XML by the file extension .xmp, where GME designates the domain model XML by the file extension .xme. Both the paradigm file and the domain model file are persistent work products for future use in production and analysis.

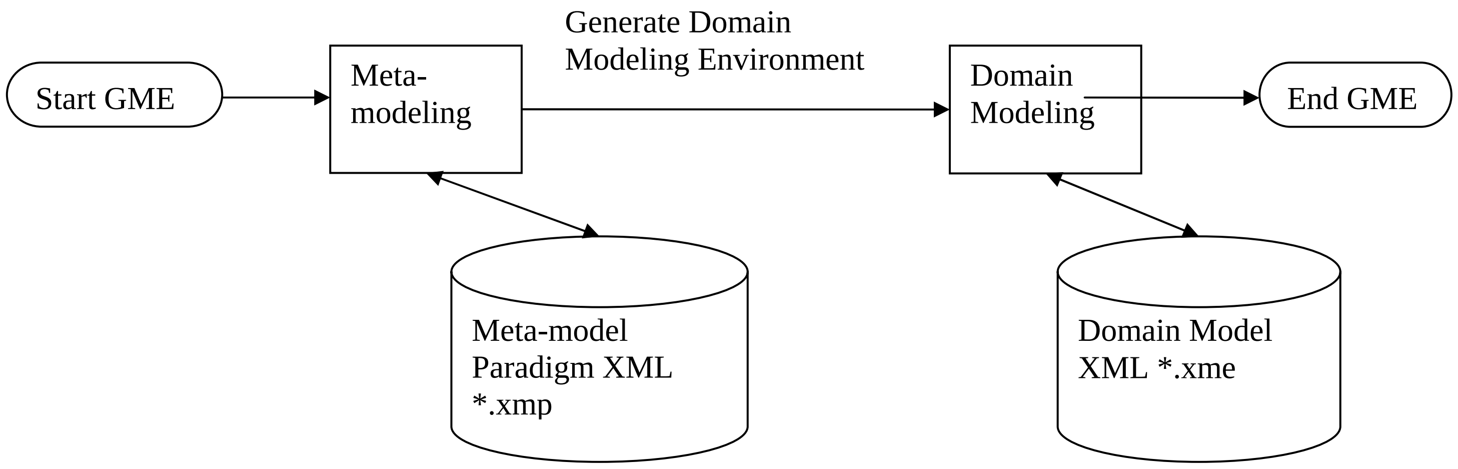


Figure 54. The Generic Modeling Environment Stores Meta-models and Domain Models as XML Files

1. Experiment XML Paradigm File – Basic_Robot.xmp

```
 <?xml version="1.0"?>
<!DOCTYPE paradigm SYSTEM "edf.dtd">

<paradigm name="Basic_Robot" guid="{A3161AA9-3ADF-4DA1-9FF0-384EE576D75F}" cdate="Sat Feb 25 16:41:47 2006" mdate="Sat Feb 25 16:41:47 2006" >

        <comment></comment>

        <author></author>

        <folder name = "RootFolder" metaref = "1000"  rootobjects = "ModelDiagram" >
                <attrdef name="CONTROLID" metaref = "1278" valuetype = "enum" defvalue = "BumpStop">
                        <enumitem dispname = "BumpStop" value = "BumpStop"></enumitem>
                        <enumitem dispname = "Follower" value = "Follower"></enumitem>
                </attrdef>
                <attrdef name="IPAddress" metaref = "1210" valuetype = "string" defvalue = "">
                        <dispname>Enter Ip Address</dispname>
                </attrdef>
                <attrdef name="JAUSMessagesIn" metaref = "1212" valuetype = "string" defvalue = "">
                        <dispname>Enter Input JAUS Messages separated by commas:</dispname>
                </attrdef>
                <attrdef name="JAUSMessagesOut" metaref = "1213" valuetype = "string" defvalue = "">
                        <dispname>Enter Output JAUS Messages separated by commas:</dispname>
                </attrdef>
                <attrdef name="ManipulatorType" metaref = "1279" valuetype = "enum" defvalue = "Oceaneering">
                        <enumitem dispname = "Oceaneering" value = "Oceaneering"></enumitem>
                        <enumitem dispname = "Kuchera" value = "Kuchera"></enumitem>
                        <enumitem dispname = "RE2" value = "RE2"></enumitem>
                        <enumitem dispname = "Turing" value = "Turing"></enumitem>
                        <enumitem dispname = "IRobot" value = "IRobot"></enumitem>
                        <enumitem dispname = "Talon" value = "Talon"></enumitem>
                </attrdef>
                <attrdef name="MessageID" metaref = "1045" valuetype = "enum" defvalue = "M1">
                        <dispname>Enter Message Id</dispname>
                        <enumitem dispname = "M1" value = "M1"></enumitem>
                        <enumitem dispname = "M2" value = "M2"></enumitem>
                        <enumitem dispname = "M3" value = "M3"></enumitem>
                </attrdef>
                <attrdef name="Multiplicity" metaref = "1277" valuetype = "integer" defvalue = "1">
                        <dispname>Number of inputs allowed</dispname>
```

```
</attrdef>
<attrdef name="OCUID" metaref = "1280" valuetype = "enum" defvalue = "ODISOCU">
        <dispname>Enter OCU ID</dispname>
        <enumitem dispname = "ODISOCU" value = "ODISOCU"></enumitem>
        <enumitem dispname = "SIMODISOCU" value = "SIMODISOCU"></enumitem>
        <enumitem dispname = "ODIS-T3OCU" value = "ODIS-T3OCU"></enumitem>
        <enumitem dispname = "Laptop" value = "Laptop"></enumitem>
        <enumitem dispname = "Swabby" value = "Swabby"></enumitem>
</attrdef>
<attrdef name="Optional" metaref = "1042" valuetype = "enum" defvalue = "optional">
        <enumitem dispname = "optional" value = "optional"></enumitem>
</attrdef>
<attrdef name="PlatformType" metaref = "1043" valuetype = "enum" defvalue = "ODIS">
        <dispname>Enter Robot type</dispname>
        <enumitem dispname = "ODIS" value = "ODIS"></enumitem>
        <enumitem dispname = "CHAOS" value = "CHAOS"></enumitem>
        <enumitem dispname = "PIONEER" value = "PIONEER"></enumitem>
        <enumitem dispname = "SIMODIS" value = "SIMODIS"></enumitem>
</attrdef>
<attrdef name="Port" metaref = "1211" valuetype = "integer" defvalue = "">
        <dispname>Enter IP Port</dispname>
</attrdef>
<attrdef name="SensorName" metaref = "1281" valuetype = "enum" defvalue = "SONAR">
        <enumitem dispname = "SONAR" value = "SONAR"></enumitem>
        <enumitem dispname = "GPS" value = "GPS"></enumitem>
        <enumitem dispname = "Explosives" value = "Explosives"></enumitem>
        <enumitem dispname = "Chemical" value = "Chemical"></enumitem>
        <enumitem dispname = "Biological" value = "Biological"></enumitem>
        <enumitem dispname = "Bumper" value = "Bumper"></enumitem>
        <enumitem dispname = "LASER" value = "LASER"></enumitem>
        <enumitem dispname = "LADAR" value = "LADAR"></enumitem>
</attrdef>
<attrdef name="SerialPort" metaref = "1214" valuetype = "enum" defvalue = "1">
        <dispname>Enter Serial Port to connect to:</dispname>
        <enumitem dispname = "1" value = "1"></enumitem>
        <enumitem dispname = "2" value = "2"></enumitem>
        <enumitem dispname = "3" value = "3"></enumitem>
        <enumitem dispname = "4" value = "4"></enumitem>
</attrdef>
<attrdef name="Speed" metaref = "1215" valuetype = "enum" defvalue = "9600">
        <dispname>Select Baud Rate</dispname>
        <enumitem dispname = "9600" value = "9600"></enumitem>
        <enumitem dispname = "19200" value = "19200"></enumitem>
</attrdef>
<atom name = "Adaptor" metaref = "1046"  attributes = "JAUSMessagesIn JAUSMessagesOut">
                <regnode name = "namePosition" value ="4"></regnode>
                <regnode name = "icon" value ="AdaptorIcon.bmp"></regnode>
</atom>
<atom name = "Optional" metaref = "1282"  attributes = "Optional">
                <regnode name = "namePosition" value ="4"></regnode>
                <regnode name = "icon" value ="OptionalIcon.bmp"></regnode>
</atom>
<atom name = "SerialCommunications" metaref = "1216"  attributes = "SerialPort Speed">
                <regnode name = "namePosition" value ="4"></regnode>
                <regnode name = "icon" value ="InOutPort.bmp"></regnode>
</atom>
<atom name = "TCPCommunications" metaref = "1260"  attributes = "Port IPAddress">
                <regnode name = "namePosition" value ="4"></regnode>
                <regnode name = "icon" value ="CommsIcon.bmp"></regnode>
</atom>
<atom name = "UDPCommunications" metaref = "1243"  attributes = "Port IPAddress">
                <regnode name = "namePosition" value ="4"></regnode>
                <regnode name = "icon" value ="CommsIcon.bmp"></regnode>
</atom>
<connection name = "Connection" metaref = "1155" >
                <regnode name = "color" value ="0x000000"></regnode>
                <regnode name = "dstStyle" value ="arrow"></regnode>
                <regnode name = "srcStyle" value ="butt"></regnode>
                <regnode name = "lineType" value ="solid"></regnode>
        <connjoint>
                <pointerspec name = "src">
                        <pointeritem desc = "Adaptor"></pointeritem>
                        <pointeritem desc = "Optional"></pointeritem>
                        <pointeritem desc = "SerialCommunications"></pointeritem>
                        <pointeritem desc = "TCPCommunications"></pointeritem>
                        <pointeritem desc = "UDPCommunications"></pointeritem>
                </pointerspec>
                <pointerspec name = "dst">
                        <pointeritem desc = "Adaptor"></pointeritem>
                        <pointeritem desc = "Optional"></pointeritem>
                        <pointeritem desc = "SerialCommunications"></pointeritem>
                        <pointeritem desc = "TCPCommunications"></pointeritem>
```

```
                              <pointeritem desc = "UDPCommunications"></pointeritem>
                        </pointerspec>
                  </connjoint>
            </connection>
            <connection name = "Message" metaref = "1006" attributes = "MessageID" >
                        <regnode name = "color" value ="0x000000"></regnode>
                        <regnode name = "dstStyle" value ="arrow"></regnode>
                        <regnode name = "srcStyle" value ="butt"></regnode>
                        <regnode name = "lineType" value ="solid"></regnode>
                  <connjoint>
                        <pointerspec name = "src">
                              <pointeritem desc = "Control"></pointeritem>
                              <pointeritem desc = "Manipulator"></pointeritem>
                              <pointeritem desc = "OCU"></pointeritem>
                              <pointeritem desc = "Platform"></pointeritem>
                              <pointeritem desc = "Sensor"></pointeritem>
                        </pointerspec>
                        <pointerspec name = "dst">
                              <pointeritem desc = "Control"></pointeritem>
                              <pointeritem desc = "Manipulator"></pointeritem>
                              <pointeritem desc = "OCU"></pointeritem>
                              <pointeritem desc = "Platform"></pointeritem>
                              <pointeritem desc = "Sensor"></pointeritem>
                        </pointerspec>
                  </connjoint>
            </connection>
            <model name = "Control" metaref = "1049" attributes = "Multiplicity CONTROLID" >
                        <regnode name = "namePosition" value ="4"></regnode>
                        <regnode name = "icon" value ="Control.gif"></regnode>
                  <constraint name="Constraint" eventmask = "0x800" depth = "0" priority = "1">
                        <![CDATA[self.attachingConnections("src") ->size <= self.Multiplicity]]>
                        <dispname>Robotic System FCO are only allowed one output</dispname>
                  </constraint>
                  <constraint name="ValidMessagesrcCardinality3" eventmask = "0x0" depth = "1" priority = "1">
                        <![CDATA[let srcCount = self.attachingConnections( "dst", Message ) -> size in
(srcCount <= 1)]]>
                        <dispname>Multiplicity of objects, which are associated to RoboticSystem as "src" over Message, has to match 0..1.</dispname>
                  </constraint>
                  <role name = "Adaptor" metaref = "1156" kind = "Adaptor"></role>
                  <role name = "Connection" metaref = "1194" kind = "Connection"></role>
                  <role name = "Optional" metaref = "1283" kind = "Optional"></role>
```

```
                <role name = "SerialCommunications" metaref = "1218" kind = "SerialCommunications"></role>
                <role name = "TCPCommunications" metaref = "1261" kind = "TCPCommunications"></role>
                <role name = "UDPCommunications" metaref = "1244" kind = "UDPCommunications"></role>
                <aspect name = "Aspect" metaref = "1284" attributes = "Multiplicity CONTROLID" >
                        <part metaref = "1160" role = "Adaptor" primary = "yes" linked = "no"></part>
                        <part metaref = "1196" role = "Connection" primary = "yes" linked = "no"></part>
                        <part metaref = "1285" role = "Optional" primary = "yes" linked = "no"></part>
                        <part metaref = "1221" role = "SerialCommunications" primary = "yes" linked = "no"></part>
                        <part metaref = "1263" role = "TCPCommunications" primary = "yes" linked = "no"></part>
                        <part metaref = "1246" role = "UDPCommunications" primary = "yes" linked = "no"></part>
                </aspect>
        </model>
        <model name = "Manipulator" metaref = "1051" attributes = "Multiplicity ManipulatorType" >
                        <regnode name = "namePosition" value ="4"></regnode>
                        <regnode name = "icon" value ="Manipulator.gif"></regnode>
                <constraint name="Constraint" eventmask = "0x800" depth = "0" priority = "1">
                        <![CDATA[self.attachingConnections("src") ->size <= self.Multiplicity]]>
                        <dispname>Robotic System FCO are only allowed one output</dispname>
                </constraint>
                <constraint name="ValidMessagesrcCardinality3" eventmask = "0x0" depth = "1" priority = "1">
                        <![CDATA[let srcCount = self.attachingConnections( "dst", Message ) -> size in
(srcCount <= 1)]]>
                        <dispname>Multiplicity of objects, which are associated to RoboticSystem as "src" over Message, has to match 0..1.</dispname>
                </constraint>
                <role name = "Adaptor" metaref = "1163" kind = "Adaptor"></role>
                <role name = "Connection" metaref = "1197" kind = "Connection"></role>
                <role name = "Optional" metaref = "1286" kind = "Optional"></role>
                <role name = "SerialCommunications" metaref = "1223" kind = "SerialCommunications"></role>
                <role name = "TCPCommunications" metaref = "1264" kind = "TCPCommunications"></role>
                <role name = "UDPCommunications" metaref = "1247" kind = "UDPCommunications"></role>
                <aspect name = "Aspect" metaref = "1287" attributes = "Multiplicity ManipulatorType" >
                        <part metaref = "1167" role = "Adaptor" primary = "yes" linked = "no"></part>
                        <part metaref = "1199" role = "Connection" primary = "yes" linked = "no"></part>
                        <part metaref = "1288" role = "Optional" primary = "yes" linked = "no"></part>
                        <part metaref = "1226" role = "SerialCommunications" primary = "yes" linked = "no"></part>
                        <part metaref = "1266" role = "TCPCommunications" primary = "yes" linked = "no"></part>
                        <part metaref = "1249" role = "UDPCommunications" primary = "yes" linked = "no"></part>
                </aspect>
        </model>
        <model name = "ModelDiagram" metaref = "1138" >
                        <regnode name = "namePosition" value ="4"></regnode>
```

```
			<role name = "Control" metaref = "1144" kind = "Control"></role>
			<role name = "Manipulator" metaref = "1145" kind = "Manipulator"></role>
			<role name = "Message" metaref = "1139" kind = "Message"></role>
			<role name = "OCU" metaref = "1146" kind = "OCU"></role>
			<role name = "Platform" metaref = "1147" kind = "Platform"></role>
			<role name = "Sensor" metaref = "1148" kind = "Sensor"></role>
			<aspect name = "Aspect" metaref = "1289" >
				<part metaref = "1150" role = "Control" primary = "yes" linked = "no"></part>
				<part metaref = "1151" role = "Manipulator" primary = "yes" linked = "no"></part>
				<part metaref = "1141" role = "Message" primary = "yes" linked = "no"></part>
				<part metaref = "1152" role = "OCU" primary = "yes" linked = "no"></part>
				<part metaref = "1153" role = "Platform" primary = "yes" linked = "no"></part>
				<part metaref = "1154" role = "Sensor" primary = "yes" linked = "no"></part>
			</aspect>
		</model>
		<model name = "OCU" metaref = "1054" attributes = "Multiplicity OCUID" >
				<regnode name = "namePosition" value ="4"></regnode>
				<regnode name = "icon" value ="OCU.bmp"></regnode>
			<constraint name="Constraint" eventmask = "0x800" depth = "0" priority = "1">
				<![CDATA[self.attachingConnections("src") ->size <= self.Multiplicity]]>
				<dispname>Robotic System FCO are only allowed one output</dispname>
			</constraint>
			<constraint name="ValidMessagesrcCardinality3" eventmask = "0x0" depth = "1" priority = "1">
				<![CDATA[let srcCount = self.attachingConnections( "dst", Message ) -> size in
(srcCount <= 1)]]>
				<dispname>Multiplicity of objects, which are associated to RoboticSystem as "src" over Message, has to match 0..1.</dispname>
			</constraint>
			<role name = "Adaptor" metaref = "1173" kind = "Adaptor"></role>
			<role name = "Connection" metaref = "1201" kind = "Connection"></role>
			<role name = "Optional" metaref = "1290" kind = "Optional"></role>
			<role name = "SerialCommunications" metaref = "1229" kind = "SerialCommunications"></role>
			<role name = "TCPCommunications" metaref = "1268" kind = "TCPCommunications"></role>
			<role name = "UDPCommunications" metaref = "1251" kind = "UDPCommunications"></role>
			<aspect name = "Aspect" metaref = "1291" attributes = "Multiplicity OCUID" >
				<part metaref = "1177" role = "Adaptor" primary = "yes" linked = "no"></part>
				<part metaref = "1203" role = "Connection" primary = "yes" linked = "no"></part>
				<part metaref = "1292" role = "Optional" primary = "yes" linked = "no"></part>
				<part metaref = "1232" role = "SerialCommunications" primary = "yes" linked = "no"></part>
				<part metaref = "1270" role = "TCPCommunications" primary = "yes" linked = "no"></part>
				<part metaref = "1253" role = "UDPCommunications" primary = "yes" linked = "no"></part>
			</aspect>
```

```
        </model>
        <model name = "Platform" metaref = "1056" attributes = "Multiplicity PlatformType" >
                        <regnode name = "namePosition" value ="4"></regnode>
                        <regnode name = "icon" value ="Platform.bmp"></regnode>
                <constraint name="Constraint" eventmask = "0x800" depth = "0" priority = "1">
                        <![CDATA[self.attachingConnections("src") ->size <= self.Multiplicity]]>
                        <dispname>Robotic System FCO are only allowed one output</dispname>
                </constraint>
                <constraint name="ValidMessagesrcCardinality3" eventmask = "0x0" depth = "1" priority = "1">
                        <![CDATA[let srcCount = self.attachingConnections( "dst", Message ) -> size in
(srcCount <= 1)]]>
                        <dispname>Multiplicity of objects, which are associated to RoboticSystem as "src" over Message, has to match 0..1.</dispname>
                </constraint>
                <role name = "Adaptor" metaref = "1180" kind = "Adaptor"></role>
                <role name = "Connection" metaref = "1204" kind = "Connection"></role>
                <role name = "Optional" metaref = "1293" kind = "Optional"></role>
                <role name = "SerialCommunications" metaref = "1234" kind = "SerialCommunications"></role>
                <role name = "TCPCommunications" metaref = "1271" kind = "TCPCommunications"></role>
                <role name = "UDPCommunications" metaref = "1254" kind = "UDPCommunications"></role>
                <aspect name = "Aspect" metaref = "1294" attributes = "Multiplicity PlatformType" >
                        <part metaref = "1184" role = "Adaptor" primary = "yes" linked = "no"></part>
                        <part metaref = "1206" role = "Connection" primary = "yes" linked = "no"></part>
                        <part metaref = "1295" role = "Optional" primary = "yes" linked = "no"></part>
                        <part metaref = "1237" role = "SerialCommunications" primary = "yes" linked = "no"></part>
                        <part metaref = "1273" role = "TCPCommunications" primary = "yes" linked = "no"></part>
                        <part metaref = "1256" role = "UDPCommunications" primary = "yes" linked = "no"></part>
                </aspect>
        </model>
        <model name = "Sensor" metaref = "1059" attributes = "Multiplicity SensorName" >
                        <regnode name = "namePosition" value ="4"></regnode>
                        <regnode name = "icon" value ="Sensor.bmp"></regnode>
                <constraint name="Constraint" eventmask = "0x800" depth = "0" priority = "1">
                        <![CDATA[self.attachingConnections("src") ->size <= self.Multiplicity]]>
                        <dispname>Robotic System FCO are only allowed one output</dispname>
                </constraint>
                <constraint name="ValidMessagesrcCardinality3" eventmask = "0x0" depth = "1" priority = "1">
                        <![CDATA[let srcCount = self.attachingConnections( "dst", Message ) -> size in
(srcCount <= 1)]]>
                        <dispname>Multiplicity of objects, which are associated to RoboticSystem as "src" over Message, has to match 0..1.</dispname>
                </constraint>
                <role name = "Adaptor" metaref = "1187" kind = "Adaptor"></role>
```

```
                        <role name = "Connection" metaref = "1207" kind = "Connection"></role>
                        <role name = "Optional" metaref = "1296" kind = "Optional"></role>
                        <role name = "SerialCommunications" metaref = "1239" kind = "SerialCommunications"></role>
                        <role name = "TCPCommunications" metaref = "1274" kind = "TCPCommunications"></role>
                        <role name = "UDPCommunications" metaref = "1257" kind = "UDPCommunications"></role>
                        <aspect name = "Aspect" metaref = "1297" attributes = "Multiplicity SensorName" >
                                <part metaref = "1191" role = "Adaptor" primary = "yes" linked = "no"></part>
                                <part metaref = "1209" role = "Connection" primary = "yes" linked = "no"></part>
                                <part metaref = "1298" role = "Optional" primary = "yes" linked = "no"></part>
                                <part metaref = "1242" role = "SerialCommunications" primary = "yes" linked = "no"></part>
                                <part metaref = "1276" role = "TCPCommunications" primary = "yes" linked = "no"></part>
                                <part metaref = "1259" role = "UDPCommunications" primary = "yes" linked = "no"></part>
                        </aspect>
                </model>
        </folder>
</paradigm>
```

2. Experiment XML Domain Model File – Basic_Robot.xme

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE project SYSTEM "mga.dtd">

<project guid="{A579F914-33EF-401B-8F7C-E32118CA3970}" cdate="Sun Mar 25 15:01:30 2007" mdate="Sun Mar 25 15:01:30 2007" version="" metaguid="{E692DE2C-
056B-4E0A-85CE-134DECD45012}" metaversion="" metaname="Basic_Robot">
    <name>Root Folder</name>
    <comment></comment>
    <author></author>
    <folder id="id-006a-00000001" relid="0x1" childrelidcntr="0x1" kind="RootFolder">
        <name>Root Folder</name>
        <model id="id-0065-00000001" kind="ModelDiagram" relid="0x1" childrelidcntr="0x8">
            <name>NewModelDiagram</name>
            <model id="id-0065-00000002" kind="Control" role="Control" relid="0x1" childrelidcntr="0x9">
                <name>Control</name>
                <regnode name="PartRegs" status="undefined">
                    <value></value>
                    <regnode name="Aspect" status="undefined">
                        <value></value>
                        <regnode name="Position" isopaque="yes">
                            <value>667,149</value>
                        </regnode>
                    </regnode>
                </regnode>
                <attribute kind="CONTROLID" status="meta">
                    <value>BumpStop</value>
                </attribute>
                <attribute kind="Multiplicity" status="meta">
                    <value>1</value>
                </attribute>
                <atom id="id-0066-00000003" kind="Adaptor" role="Adaptor" relid="0x1">
                    <name>Adaptor</name>
                    <regnode name="PartRegs" status="undefined">
                        <value></value>
                        <regnode name="Aspect" status="undefined">
                            <value></value>
                            <regnode name="Position" isopaque="yes">
                                <value>129,102</value>
                            </regnode>
                        </regnode>
```

```
		</regnode>
		<attribute kind="JAUSMessagesIn">
			<value>M2,M4</value>
		</attribute>
		<attribute kind="JAUSMessagesOut">
			<value>M4</value>
		</attribute>
	</atom>
	<atom id="id-0066-00000004" kind="SerialCommunications" role="SerialCommunications" relid="0x2">
		<name>SerialCommunications</name>
		<regnode name="PartRegs" status="undefined">
			<value></value>
			<regnode name="Aspect" status="undefined">
				<value></value>
				<regnode name="Position" isopaque="yes">
					<value>509,103</value>
				</regnode>
			</regnode>
		</regnode>
		<attribute kind="SerialPort" status="meta">
			<value>1</value>
		</attribute>
		<attribute kind="Speed" status="meta">
			<value>9600</value>
		</attribute>
	</atom>
	<atom id="id-0066-00000005" kind="UDPCommunications" role="UDPCommunications" relid="0x3">
		<name>UDPCommunications</name>
		<regnode name="PartRegs" status="undefined">
			<value></value>
			<regnode name="Aspect" status="undefined">
				<value></value>
				<regnode name="Position" isopaque="yes">
					<value>538,300</value>
				</regnode>
			</regnode>
		</regnode>
		<attribute kind="IPAddress">
			<value>10.0.0.1</value>
		</attribute>
		<attribute kind="Port">
```

```
                <value>3743</value>
        </attribute>
</atom>
<atom id="id-0066-00000006" kind="optional" role="optional" relid="0x4">
        <name>optional</name>
        <regnode name="PartRegs" status="undefined">
                <value></value>
                <regnode name="Aspect" status="undefined">
                        <value></value>
                        <regnode name="Position" isopaque="yes">
                                <value>193,286</value>
                        </regnode>
                </regnode>
        </regnode>
        <attribute kind="Optional" status="meta">
                <value>optional</value>
        </attribute>
</atom>
<connection id="id-0068-00000009" kind="Connection" role="Connection" relid="0x6">
        <name>Connection</name>
        <regnode name="autorouterPref" isopaque="yes">
                <value>Nn</value>
        </regnode>
        <connpoint role="dst" target="id-0066-00000003"/>
        <connpoint role="src" target="id-0066-00000004"/>
</connection>
<connection id="id-0068-0000000a" kind="Connection" role="Connection" relid="0x7">
        <name>Connection</name>
        <regnode name="autorouterPref" isopaque="yes">
                <value>Sw</value>
        </regnode>
        <connpoint role="src" target="id-0066-00000003"/>
        <connpoint role="dst" target="id-0066-00000006"/>
</connection>
<connection id="id-0068-0000000b" kind="Connection" role="Connection" relid="0x8">
        <name>Connection</name>
        <regnode name="autorouterPref" isopaque="yes">
                <value>Ew</value>
        </regnode>
        <connpoint role="dst" target="id-0066-00000005"/>
        <connpoint role="src" target="id-0066-00000006"/>
        </connection>
        <connection id="id-0068-0000000c" kind="Connection" role="Connection" relid="0x9">
                <name>Connection</name>
                <regnode name="autorouterPref" isopaque="yes">
                        <value>Ne</value>
                </regnode>
                <connpoint role="dst" target="id-0066-00000003"/>
                <connpoint role="src" target="id-0066-00000005"/>
        </connection>
</model>
<model id="id-0065-00000003" kind="OCU" role="OCU" relid="0x2" childrelidcntr="0x4">
        <name>OCU</name>
        <regnode name="PartRegs" status="undefined">
                <value></value>
                <regnode name="Aspect" status="undefined">
                        <value></value>
                        <regnode name="Position" isopaque="yes">
                                <value>646,373</value>
                        </regnode>
                </regnode>
        </regnode>
        <attribute kind="Multiplicity" status="meta">
                <value>1</value>
        </attribute>
        <attribute kind="OCUID" status="meta">
                <value>ODISOCU</value>
        </attribute>
        <atom id="id-0066-0000000b" kind="Adaptor" role="Adaptor" relid="0x1">
                <name>Adaptor</name>
                <regnode name="PartRegs" status="undefined">
                        <value></value>
                        <regnode name="Aspect" status="undefined">
                                <value></value>
                                <regnode name="Position" isopaque="yes">
                                        <value>156,177</value>
                                </regnode>
                        </regnode>
                </regnode>
                <attribute kind="JAUSMessagesIn">
                        <value>M4</value>
                </attribute>
                <attribute kind="JAUSMessagesOut">
                        <value>M1</value>
                </attribute>
        </atom>
        <atom id="id-0066-0000000c" kind="UDPCommunications" role="UDPCommunications" relid="0x2">
                <name>UDPCommunications</name>
                <regnode name="PartRegs" status="undefined">
                        <value></value>
                        <regnode name="Aspect" status="undefined">
                                <value></value>
                                <regnode name="Position" isopaque="yes">
                                        <value>450,191</value>
                                </regnode>
                        </regnode>
                </regnode>
                <attribute kind="IPAddress">
                        <value>10.0.0.3</value>
                </attribute>
                <attribute kind="Port">
                        <value>3743</value>
                </attribute>
        </atom>
        <connection id="id-0068-00000011" kind="Connection" role="Connection" relid="0x3">
                <name>Connection</name>
                <regnode name="autorouterPref" isopaque="yes">
                        <value>Nn</value>
                </regnode>
                <connpoint role="src" target="id-0066-0000000b"/>
                <connpoint role="dst" target="id-0066-0000000c"/>
        </connection>
        <connection id="id-0068-00000012" kind="Connection" role="Connection" relid="0x4">
                <name>Connection</name>
                <regnode name="autorouterPref" isopaque="yes">
                        <value>We</value>
                </regnode>
                <connpoint role="dst" target="id-0066-0000000b"/>
                <connpoint role="src" target="id-0066-0000000c"/>
        </connection>
</model>
<model id="id-0065-00000004" kind="Platform" role="Platform" relid="0x3" childrelidcntr="0x8">
        <name>Platform</name>
```

```
<regnode name="PartRegs" status="undefined">
        <value></value>
        <regnode name="Aspect" status="undefined">
                <value></value>
                <regnode name="Position" isopaque="yes">
                        <value>226,379</value>
                </regnode>
        </regnode>
</regnode>
<attribute kind="Multiplicity" status="meta">
        <value>1</value>
</attribute>
<attribute kind="PlatformType" status="meta">
        <value>ODIS</value>
</attribute>
<atom id="id-0066-00000007" kind="Adaptor" role="Adaptor" relid="0x1">
        <name>Adaptor</name>
        <regnode name="PartRegs" status="undefined">
                <value></value>
                <regnode name="Aspect" status="undefined">
                        <value></value>
                        <regnode name="Position" isopaque="yes">
                                <value>128,156</value>
                        </regnode>
                </regnode>
        </regnode>
        <attribute kind="JAUSMessagesIn">
                <value>M4</value>
        </attribute>
        <attribute kind="JAUSMessagesOut">
                <value>M1</value>
        </attribute>
</atom>
<atom id="id-0066-00000008" kind="UDPCommunications" role="UDPCommunications" relid="0x2">
        <name>UDPCommunications</name>
        <regnode name="PartRegs" status="undefined">
                <value></value>
                <regnode name="Aspect" status="undefined">
                        <value></value>
                        <regnode name="Position" isopaque="yes">
                                <value>611,184</value>
                        </regnode>
                </regnode>
        </regnode>
        <attribute kind="IPAddress">
                <value>10.0.0.2</value>
        </attribute>
        <attribute kind="Port">
                <value>3743</value>
        </attribute>
</atom>
<atom id="id-0066-00000009" kind="optional" role="optional" relid="0x3">
        <name>optional</name>
        <regnode name="PartRegs" status="undefined">
                <value></value>
                <regnode name="Aspect" status="undefined">
                        <value></value>
                        <regnode name="Position" isopaque="yes">
                                <value>359,65</value>
                        </regnode>
                </regnode>
        </regnode>
        <attribute kind="Optional" status="meta">
                <value>optional</value>
        </attribute>
</atom>
<atom id="id-0066-0000000a" kind="optional" role="optional" relid="0x4">
        <name>optional</name>
        <regnode name="PartRegs" status="undefined">
                <value></value>
                <regnode name="Aspect" status="undefined">
                        <value></value>
                        <regnode name="Position" isopaque="yes">
                                <value>366,282</value>
                        </regnode>
                </regnode>
        </regnode>
        <attribute kind="Optional" status="meta">
                <value>optional</value>
        </attribute>
</atom>
<connection id="id-0068-0000000d" kind="Connection" role="Connection" relid="0x5">
```

```
                <name>Connection</name>
                <regnode name="autorouterPref" isopaque="yes">
                        <value>Ne</value>
                </regnode>
                <connpoint role="src" target="id-0066-00000008"/>
                <connpoint role="dst" target="id-0066-00000009"/>
        </connection>
        <connection id="id-0068-0000000e" kind="Connection" role="Connection" relid="0x6">
                <name>Connection</name>
                <regnode name="autorouterPref" isopaque="yes">
                        <value>Wn</value>
                </regnode>
                <connpoint role="dst" target="id-0066-00000007"/>
                <connpoint role="src" target="id-0066-00000009"/>
        </connection>
        <connection id="id-0068-0000000f" kind="Connection" role="Connection" relid="0x7">
                <name>Connection</name>
                <regnode name="autorouterPref" isopaque="yes">
                        <value>Sw</value>
                </regnode>
                <connpoint role="src" target="id-0066-00000007"/>
                <connpoint role="dst" target="id-0066-0000000a"/>
        </connection>
        <connection id="id-0068-00000010" kind="Connection" role="Connection" relid="0x8">
                <name>Connection</name>
                <regnode name="autorouterPref" isopaque="yes">
                        <value>Es</value>
                </regnode>
                <connpoint role="dst" target="id-0066-00000008"/>
                <connpoint role="src" target="id-0066-0000000a"/>
        </connection>
</model>
<model id="id-0065-00000005" kind="Sensor" role="Sensor" relid="0x4" childrelidcntr="0x3">
        <name>Sensor</name>
        <regnode name="PartRegs" status="undefined">
                <value></value>
                <regnode name="Aspect" status="undefined">
                        <value></value>
                        <regnode name="Position" isopaque="yes">
                                <value>228,83</value>
                        </regnode>
        </regnode>
</regnode>
<attribute kind="Multiplicity" status="meta">
        <value>1</value>
</attribute>
<attribute kind="SensorName" status="meta">
        <value>SONAR</value>
</attribute>
<atom id="id-0066-00000001" kind="Adaptor" role="Adaptor" relid="0x1">
        <name>Adaptor</name>
        <regnode name="PartRegs" status="undefined">
                <value></value>
                <regnode name="Aspect" status="undefined">
                        <value></value>
                        <regnode name="Position" isopaque="yes">
                                <value>143,102</value>
                        </regnode>
                </regnode>
        </regnode>
        <attribute kind="JAUSMessagesIn" status="meta">
                <value></value>
        </attribute>
        <attribute kind="JAUSMessagesOut">
                <value>M4</value>
        </attribute>
</atom>
<atom id="id-0066-00000002" kind="SerialCommunications" role="SerialCommunications" relid="0x2">
        <name>SerialCommunications</name>
        <regnode name="PartRegs" status="undefined">
                <value></value>
                <regnode name="Aspect" status="undefined">
                        <value></value>
                        <regnode name="Position" isopaque="yes">
                                <value>495,96</value>
                        </regnode>
                </regnode>
        </regnode>
        <attribute kind="SerialPort" status="meta">
                <value>1</value>
        </attribute>
        <attribute kind="Speed" status="meta">
```

```
                <value>9600</value>
            </attribute>
        </atom>
        <connection id="id-0068-00000006" kind="Connection" role="Connection" relid="0x3">
            <name>Connection</name>
            <regnode name="autorouterPref" isopaque="yes">
                <value>Ew</value>
            </regnode>
            <connpoint role="src" target="id-0066-00000001"/>
            <connpoint role="dst" target="id-0066-00000002"/>
        </connection>
    </model>
    <connection id="id-0068-00000001" kind="Message" role="Message" relid="0x5">
        <name>Message</name>
        <regnode name="autorouterPref" isopaque="yes">
            <value>Ne</value>
        </regnode>
        <attribute kind="MessageID" status="meta">
            <value>M1</value>
        </attribute>
        <connpoint role="dst" target="id-0065-00000002"/>
        <connpoint role="src" target="id-0065-00000003"/>
    </connection>
    <connection id="id-0068-00000002" kind="Message" role="Message" relid="0x6">
        <name>Message</name>
        <regnode name="autorouterPref" isopaque="yes">
            <value>En</value>
        </regnode>
        <attribute kind="MessageID" status="meta">
            <value>M1</value>
        </attribute>
        <connpoint role="dst" target="id-0065-00000002"/>
        <connpoint role="src" target="id-0065-00000005"/>
    </connection>
    <connection id="id-0068-00000004" kind="Message" role="Message" relid="0x7">
        <name>Message</name>
        <regnode name="autorouterPref" isopaque="yes">
            <value>Wn</value>
        </regnode>
        <attribute kind="MessageID" status="meta">
            <value>M1</value>
```

```
                    </attribute>
                    <connpoint role="src" target="id-0065-00000002"/>
                    <connpoint role="dst" target="id-0065-00000004"/>
                </connection>
                <connection id="id-0068-00000005" kind="Message" role="Message" relid="0x8">
                    <name>Message</name>
                    <regnode name="autorouterPref" isopaque="yes">
                        <value>Ew</value>
                    </regnode>
                    <attribute kind="MessageID" status="meta">
                        <value>M1</value>
                    </attribute>
                    <connpoint role="dst" target="id-0065-00000003"/>
                    <connpoint role="src" target="id-0065-00000004"/>
                </connection>
            </model>
        </folder>
    </project>
```

APPENDIX D. RULES

A. INTRODUCTION

Rules are instantiated in the Ruby language. The rules operate on the GME domain model XML file (.xme). Part one of this appendix presents each rule in text and pseudo code. It also presents output of each rule, although these intermediate values are internal for the first half of the rules. The second half of the rules builds the codes for operating the prototype.

1. Rule Descriptions

Rule 1. Create Artifact List

Parse the XML file, find the Artifacts from child models of the top-level model and create Artifact frames for them with id, name and kind slots (**Table 14**).

Pseudo Code	Output

artifact = new array
i = 1
for each level 1 model in "basic_robot_domain.xme"
   artifact[i] = new hash
artifact[i].id = id
artifact[i].name = name
artifact[i].kind = kind
i = i + 1
endfor
	Artifact[1].id = "id-0065-00000002"
	Artifact[1].name = ODISOCU
	Artifact[1].kind = "OCU"
	
	Artifact[2].id = "id-0065-00000003"
	Artifact[2].name = ODIS
	Artifact[2].kind = "Platform"
	
	Artifact[3].id = "id-0065-00000004"
	Artifact[3].name = BumpControl
	Artifact[3].kind = "Control"
	
	Artifact[4].id = "id-0065-00000005"
	Artifact[4].name = Sonar
	Artifact[4].kind = "Sensor"

Table 14. Create an Array Artifacts, at each Artifact Create a Hash and Instantiate Information Frames for the Artifacts

Rule 2. Create Channel List:

Parse the XML file, find the Channels from child <connection> tags of the top level model and create Channel frames for them with source and destination slots where source and destination are the “id’s” are directly related to <Connpoint> target and role attributes (**Table 15**).

Pseudo Code	Output

channel = new array
i = 1
for each level 1 connection in
“basic_robot_domain.xme”
 	
channel[i] = new hash
channel[i].source = source
channel[i].destination = destination
endfor
	Channel[1].source = "id-0065-00000005"
	Channel[1].destination = "id-0065-00000004"
	
	Channel[2].source = "id-0065-00000004"
	Channel[2].destination = "id-0065-00000003"
	
	Channel[3].source = "id-0065-00000003"
	Channel[3].destination = "id-0065-00000002"
	
	Channel[4].source = "id-0065-00000002"
	Channel[4].destination = "id-0065-00000004"

Table 15. Create an Array Channels, at each Channel Create a Hash and Instantiate Information Frames for the Channels

Rule 3. Additional Channel information:

For readability, add Channel[Y].source.name to each Channel Frame by going through the Artifacts and comparing their id slot to the Channel[Y].source slot. If they match, add the Artifact name to a sub slot of Channel[Y].source, Channel[Y].source.name. Do same for Channel[Y].destination. (**Table 16**).

Pseudo Code	Output

For each Channel
  For each Artifact
    If Artifact.id = Channel.source,
        Then
        Channel.source.name = Artifact[.name
    Endif
 	
    If Artifact.id = Channel.destination,
        Then
        Channel[.destination.name = Artifact.name
   Endif
 Endfor
Endfor
	Channel[1].source = "id-0065-00000005"
	Channel[1].destination = "id-0065-00000004"
	Channel[1].source.name = Sonar
	Channel[1]. destination.name = BumpControl
	
	Channel[2].source = "id-0065-00000004"
	Channel[2]. destination = "id-0065-00000003"
	Channel[2].source.name = BumpControl
	Channel[2]. destination.name = ODIS
	
	Channel[3].source = "id-0065-00000003"
	Channel[3]. destination = "id-0065-00000002"
	Channel[3].source.name = ODIS
	Channel[3]. destination.name = ODISOCU
	
	Channel[4].source = "id-0065-00000002"
	Channel[4]. destination = "id-0065-00000004"
	Channel[4].source.name = ODISOCU
	Channel[4]. destination.name = BumpControl

Table 16. Match GME Generated ID Fields between Artifacts and Channels to Add Human Understandable Names to the Channel Information Frames

Rule 4. Create Artifact Component Lists:

For each Artifact discovered in Rule 1, parse the XML file, find the child Atoms. Create Component frames for them with id, name and kind slots (**Table 17**).

Pseudo Code	Output
For each level 1 Model in xme file	Component[1,1].id = "id-0066-0000000b"
For each atom in model	Component[1,1].name = ODISOCUAdaptor
Component = hash.new	Component[1,1].kind = "Adaptor"
Component.id = id,	
Componen.name = name,	Component[1,2].id = "id-0066-0000000c"
Component.kind = kind.	Component[1,2].name =
Endfor	ODISOCUTCPCommunications
Endfor	Component[1,2].kind = "TCPCommunications"
	Component[2,1].id = "id-0066-00000007"
	Component[2,1].name = ODISAdaptor
	Component[1,1].kind = "Adaptor"
	Component[2,2].id = "id-0066-00000008"
	Component[2,2].name =
	ODISTCPCommunications
	Component[2,2].kind = "TCPCommunications"
	Component[2,3].id = "id-0066-00000009"
	Component[2,3].name = ODISInoptional
	Component[2,3].kind = "optional"
	Component[2,4].id = "id-0066-0000000a"
	Component[2,4].name = ODISOutoptional
	Component[2,4].kind = "optional"
	Component[3,1].id = "id-0066-00000003"
	Component[3,1].name = BumpControlAdaptor
	Component[3,1].kind = "Adaptor"
	Component[3,2].id = "id-0066-00000004"
	Component[3,2].name =
	BumpControlSerialCommunications
	Component[3,2].kind = " SerialCommunications"
	Component[3,3].id = "id-0066-00000005"
	Component[3,3].name =
	BumpControlTCPCommunicationsl
	Component[3,3].kind = "TCPCommunications"
	Component[3,4].id = "id-0066-00000006"
	Component[3,4].name = optional
	Component[3,4].kind = "optional"
	Component[4,1].id = "id-0066-00000001"

Component[4,1].name = SonarControlAdaptor Component[4,1].kind = "Adaptor" Component[4,2].id = "id-0066-00000002" Component[4,2].name = SonarSerialCommunications Component[4,2].kind = " SerialCommunications"

Table 17. Create Component Arrays for each Artifact, Create a Hash and Instantiate Information Frames for the Components as was Done for the Parent Artifacts

Rule 5. Create Artifact dataflow lists:

Parse the XML file, find the Dataflows from child <connection> tags of the child model (Artifact) of the top level model and create Dataflow frames for them with source and destination slots where source and destination are the “id’s” are directly related to <Connpoint> target and role attributes (**Table 18**).

Pseudo Code	Output
For each Artifact	Dataflow[1,1].source = "id-0066-0000000b"
For each connection	Dataflow[1,1].destination = "id-0066-0000000c"
Dataflow = new.hash	
For each Connpoint	Dataflow[1,2].source = "id-0066-0000000c"
If Connpoint .role = “src”	Dataflow[1,2].destination = "id-0066-0000000b"
Then	
Dataflow.source =Connpoint.target	Dataflow[2,1].source = "id-0066-00000007"
Endif	Dataflow[2,1].destination = "id-0066-0000000a"
If Connpoint role = “dst”	
Then	Dataflow[2,2].source = "id-0066-0000000a"
Dataflow.destination =Connpoint.target	Dataflow[2,2].destination = "id-0066-00000008"
Endif	
Endfor	Dataflow[2,3].source = "id-0066-00000008"
Endfor	Dataflow[2,3].destination = "id-0066-00000009"
Endfor	
	Dataflow[2,4].source = "id-0066-00000009"
	Dataflow[2,4].destination = "id-0066-00000007"
	Dataflow[3,1].source = "id-0066-00000004"
	Dataflow[3,1].destination = "id-0066-00000003"
	Dataflow[3,2].source = "id-0066-00000003"
	Dataflow[3,2].destination = "id-0066-00000006"
	Dataflow[3,3].source = "id-0066-00000006"
	Dataflow[3,3].destination = "id-0066-00000005"
	Dataflow[3,4].source = "id-0066-00000005"
	Dataflow[3,4].destination = "id-0066-00000003"
	Dataflow[4,1].source = "id-0066-00000001"
	Dataflow[4,1].destination = "id-0066-00000002"

Table 18. Create an Arrays of Dataflows associated with each Artifact, at each Dataflow, Create a Hash and Instantiate Information Frames for the Dataflows

Rule 6. Additional Data flow information:

Add Dataflow[X,W].source.name to each Dataflow Frame by going through the Components in the current X and comparing their id slot to the Dataflow[X,W].source slot. If they match, add the Component name to a sub slot of Dataflow[X,W].source, Dataflow[X,W].source.name. Do same for Dataflow[X,W].destination.

Above rules can be modified to capture other information captured in the Component data structures or directly from the XML file (**Table 19**).

Pseudo Code	Output
For each Artifact	Dataflow[1,1].source = "id-0066-0000000b"
For each Component	Dataflow[1,1].source.name = ODISOCUAdaptor
For each Dataflow	Dataflow[1,1].source.kind = "Adaptor"
If Component.id = Dataflow.source	Dataflow[1,1].destination = "id-0066-0000000c"
Then	Dataflow[1,1].destination.name =
Dataflow.source.name= Component.name	ODISOCUTCPCommunications
Dataflow.source.kind = Component.kind	Dataflow[1,1].destination.kind =
Endif	"TCPCommunications"
If Component.id = Dataflow.destination,	
Then	Dataflow[1,2].source = "id-0066-0000000c"
Dataflow.destination.name =	Dataflow[1,2].source.name =
Componen.name	ODISOCUTCPCommunications
Dataflow.destination.kind =	Dataflow[1,2].source.kind = "TCPCommunications"
Component.kind	Dataflow[1,2].destination = "id-0066-0000000b"
Endif	Dataflow[1,2].destination.name = ODISOCUAdaptor
Endfor	Dataflow[1,2].destination.kind = "Adaptor"
Endfor	
Endfor	Dataflow[2,1].source = "id-0066-00000007"
	Dataflow[2,1].source.name = ODISAdaptor
	Dataflow[2,1].source.kind = "Adaptor"
	Dataflow[2,1].destination = "id-0066-0000000a"
	Dataflow[2,1].destination.name = ODISOutoptional
	Dataflow[2,1].destination.kind = "optional"
	Dataflow[2,2].source = "id-0066-0000000a"
	Dataflow[2,2].source.name = ODISOutoptional
	Dataflow[2,2].source.kind = "optional"
	Dataflow[2,2].destination = "id-0066-00000008"
	Dataflow[2,2].destination.name =
	ODISTCPCommunications
	Dataflow[2,2].destination.kind =
	"TCPCommunications"
	Dataflow[2,3].source = "id-0066-00000008"
	Dataflow[2,3].source.name =
	ODISTCPCommunications
	Dataflow[2,3].source.kind =
	"TCPCommunications"
	Dataflow[2,3].destination = "id-0066-00000009"
	Dataflow[2,3].destination.name = ODISInoptional
	Dataflow[2,3].destination.kind = "optional"

	Dataflow[2,4].source = "id-0066-00000009" Dataflow[2,4].source.name = ODISInoptional Dataflow[2,4].source.kind = "optional" Dataflow[2,4].destination = "id-0066-00000007" Dataflow[2,4].destination.name = ODISAdaptor Dataflow[2,4].destination.kind = "Adaptor" Dataflow[3,1].source = "id-0066-00000004" Dataflow[3,1].source.name = BumpControlSerialCommunications Dataflow[3,1].source.kind = " SerialCommunications" Dataflow[3,1].destination = "id-0066-00000003" Dataflow[3,1].destination.name = BumpControlAdaptor Dataflow[3,1].destination.kind = "Adaptor" Dataflow[3,2].source = "id-0066-00000003" Dataflow[3,2].source.name = BumpControlAdaptor Dataflow[3,2].source.kind = "Adaptor" Dataflow[3,2].destination = "id-0066-00000006" Dataflow[3,2].destination.name = optional Dataflow[3,2].destination.kind = "optional" Dataflow[3,3].source = "id-0066-00000006" Dataflow[3,3].source.name = optional Dataflow[3,3].source.kind = "optional" Dataflow[3,3].destination = "id-0066-00000005" Dataflow[3,3].destination.name = BumpControlTCPCommunications Dataflow[3,3].destination.kind = "TCPCommunications" Dataflow[3,4].source = "id-0066-00000005" Dataflow[3,4].source.name = BumpControlTCPCommunications Dataflow[3,4].source.kind = "TCPCommunications" Dataflow[3,4].destination = "id-0066-00000003" Dataflow[3,4].destination.name = BumpControlAdaptor Dataflow[3,4].destination.kind = "Adaptor" Dataflow[4,1].source = "id-0066-00000001" Dataflow[4,1].source.name = SonarControlAdaptor Dataflow[4,1].source.kind = "Adaptor" Dataflow[4,1].destination = "id-0066-00000002" Dataflow[4,1].destination.name = SonarSerialCommunications Dataflow[4,1].destination.kind = " SerialCommunications"

Table 19. Match GME Generated ID Fields between Components and Dataflows to Add Human Understandable Information to the Channel Information Frames

Rule 7. Create Artifact Message Lists:

For each Component atom, if XML kind attribute is “adaptor,” search for an <attribute> tag with of kind = "JAUSMessagesIn.” Insert the <attribute> tag’s value into the corresponding Artifact’s messages_in slot. Do same for kind = "JAUSMessagesOut" (**Table 20**).

Pseudo Code	Output
For each Artifact	Artifact[1].id = "id-0065-00000002"
For each Component	Artifact[1].name = ODISOCU
If Component kind = “Adaptor”	Artifact[1].kind = “OCU”
then	Artifact[1].messages_in = M1
For each attribute in Component	Artifact[1].messages_out = M2
If attribute.kind="JAUSMessagesIn"	
then	Artifact[2].id = "id-0065-00000003"
Artifact.messages_in = attribute.value.	Artifact[2].name = ODIS
Endif	Artifact[2].kind = “Platform”
If attribute.kind="JAUSMessagesOut"	Artifact[2].messages_in = M3
then	Artifact[2].messages_out = M1
Artifac.messages_out = attribute.value.	
Endif	Artifact[3].id = "id-0065-00000004"
Endfor	Artifact[3].name = BumpControl
Endif	Artifact[3].kind = “Control”
Endfor	Artifact[3].messages_in = M2,M4
	Artifact[3].messages_out = M3
	Artifact[4].id = "id-0065-00000005"
	Artifact[4].name = Sonar
	Artifact[4].kind = “Sensor”
	Artifact[4].messages_in = null
	Artifact[4].messages_out = M4

Table 20. Search the Components for Adaptor Types, Register Supported Messages in the Artifact Frames (for future use)

Rule 8. Create Communications Components attribute list:

From the XML file, fill in appropriate slots in the Component data structures for Communication components, i.e., speed and serial port for serial communications, IP address or appropriate IP port for IP communications (**Table 21**).

Pseudo Code	Output

For each Artifact
  For each Component
    If Component kind = SerialCommunications
      For each attribute]
        If attribute.kind = “Speed”
          then
          Component.serialspeed = attribute.value
        Endif
        If attribute. kind = “SerialPort”
        then
        Component.serialport = attribute.value
       Endif
       If attribute.kind = UDPCommunications
        then
      If attribute. kind = “IPAddress”
       then
       Component.IPAddress = attribute.value
       Endif
       If attribute.kind = “Port”
        then
         Component.port = attribute.value
      Endif
     Endif
    Endfor
   Endif
  Endfor
Endfor
	Component[1,1].id = "id-0066-0000000b"
	Component[1,1].name = ODISOCUAdaptor
	Component[1,1].kind = "Adaptor"
	
	Component[1,2].id = "id-0066-0000000c"
	Component[1,2].name = ODISOCUTCPCommunications
	Component[1,2].kind = "TCPCommunications"
	Component[1,2].IPAddress = 10.0.0.3
	Component[1,1].port = 3743
	
	Component[2,1].id = "id-0066-00000007"
	Component[2,1].name = ODISAdaptor
	Component[1,1].kind = "Adaptor"
	
	Component[2,2].id = "id-0066-00000008"
	Component[2,2].name = ODISTCPCommunications
	Component[2,2].kind = "TCPCommunications"
	Component[2,2].IPAddress = 10.0.0.2
	Component[2,2].port = 3743
	
	Component[2,3].id = "id-0066-00000009"
	Component[2,3].name = ODISInoptional
	Component[2,3].kind = "optional"
	
	Component[2,4].id = "id-0066-0000000a"
	Component[2,4].name = ODISOutoptional
	Component[2,4].kind = "optional"
	
	Component[3,1].id = "id-0066-00000003"
	Component[3,1].name = BumpControlAdaptor
	Component[3,1].kind = "Adaptor"
	
	Component[3,2].id = "id-0066-00000004"
	Component[3,2].name = BumpControlSerialCommunications
	Component[3,2].kind = " SerialCommunications"
	Component[3,2].serialspeed = 9600
	Component[3,2].serialport = 1
	
	Component[3,3].id = "id-0066-00000005"
	Component[3,3].name = BumpControlTCPCommunicationsl
	Component[3,3].kind = "TCPCommunications"
	Component[3,3].IPAddress = 10.0.0.1

	Component[3,3].port = 3743 Component[3,4].id = "id-0066-00000006" Component[3,4].name = optional Component[3,4].kind = "optional" Component[4,1].id = "id-0066-00000001" Component[4,1].name = SonarControlAdaptor Component[4,1].kind = "Adaptor" Component[4,2].id = "id-0066-00000002" Component[4,2].name = SonarSerialCommunications Component[4,2].kind = " SerialCommunications" Component[4,2].serialspeed = 9600 Component[4,2].serialport = 1

Table 21. For Each Artifact, Search find the Communications Parameters and Fill in Communications Fields in the Component Frames

Rule 9. Build Node Folders:

Create a file folder for each Artifact using the Artifacts name slot for a folder name (**Table 22**).

Pseudo Code	Output
For each Artifact Create folder with name string "Artifact.name." Endfor	Folder $ARBITRARY\\ $ARBITRARY\\ODISOCU $ARBITRARY\\ODIS $ARBITRARY\\BumpStop $ARBITRARY\\Sonar

Table 22. For Each Artifact, Create Folders in File System

Rule 10. Import Files.

Import configurable and non-configurable files associated with each component into the parent artifact’s folder (**Table 23**).

Pseudo Code

```
For each Artifact
    For each Component
      If Component.kind = (“Adaptor” or “optional”)
         Then
          copy file “Component.name” into folder
                                               “Artifact.name”
         Else
    copy file “Componen.kind\*” into folder
                                               “Artifact.name”
       Endif
   Endfor
Endfor
```

Output

```
$ARBITRARY\ODISOCU:
        ODISOCUMain.java
        ODISOCUAdaptor.java
        ODISOCUConstants.java
        UDPCommunications.java

$ARBITRARY\ODIS:
        ODISMain.java
        ODISAdaptor.java
        ODISConstants.java
        UDPCommunications.java
        Optional.java

$ARBITRARY\BumpStop:
        BumpStopMain.java
        BumpStopAdaptor.Java
        BumpStopConstants.Java
        Optional.java
        UDPCommunications.java
        SerialCommunications.java

$ARBITRARY\Sonar:
        SONARMain.java
        SONARAdaptor.java
        SONARConstants.java
        Serialcommunications.java
```

Table 23. For Each Artifact, Copy the Component Files Matching Model Components from the Component Repository to the Appropriate Folder in the Working Directory

Rule 11. Set Artifact Comms destination element.

For each artifact, search the channel data. If the artifact name field matches the channel source_name field, then set the artifact's comms_destination field to the value of the channel's destination name field. (**Table 24**).

Pseudo Code

```
For each Artifact
  For each channel
     If  channel.source_name = Artifact.name
       Then
        Artifact.comms_destination =
                              channel.destination_name
     endif
    Endfor
Endfor
```

Output

```
[{"comms_destination"=>"ODIS",
  "name"=>"BumpStop",
  "displayed_name"=>"Control",
  "messages_out"=>"M4",
  "id"=>"id-0065-00000002",
  "messages_in"=>"M2,M4"},

 {"comms_destination"=>"BumpStop",
  "name"=>"ODISOCU",
  "displayed_name"=>"OCU",
  "messages_out"=>"M1",
  "id"=>"id-0065-00000003",
  "messages_in"=>"M4"},

 {"comms_destination"=>"ODISOCU",
  "name"=>"ODIS",
  "displayed_name"=>"Platform",
  "messages_out"=>"M1",
  "id"=>"id-0065-00000004",
  "messages_in"=>"M4"},

 {"comms_destination"=>"BumpStop",
  "name"=>"SONAR",
  "displayed_name"=>"Sensor",
  "messages_out"=>"M4",
  "id"=>"id-0065-00000005",
  "messages_in"=>nil}]
```

Table 24. Search the Artifacts to Discover the Name of the Communications Destination Element

Rule 12. We now have enough information to do error checking if desired.

Error checks to make sure serial speeds match.
For each Artifact, check child components for SerialCommunications components. If found, find a Channel frame that has the source matching the Artifact. Find the destination Artifact from the channel destination slot. In the destination Artifact, find the SerialCommunications Component and check the appropriate destination Component slot to insure they match (**Table 25**).

Pseudo Code	Output

```
i= 1
For each Artifact[i]
j = 1
  For each Artifact[j]
     If artifact[i].name != artifact[j].name
       Then
           If Artifact[i].comms_destination =
                                              artifact[j].name
           then
               For each component in artifact[i]
                 If component.kind =
                                    ‘SerialCommunications”
                     Then
                      For each component in Artifact[j]
                          If component.kind =
                                    ‘SerialCommunications’
                           Then
                        If
                         artifact[i].component.serialspeed !=
                           artifact[j].component.serialspeed
                           Then
                             print error message
                          endif
                        If
                         artifact[i].component.serialport !=
                           artifact[j].component.serialport
                           Then
                             print error message
                          Endif
                         Endif
                      Endfor
                  Endif
               Endfor
             Endif
         Endif
    j = j + 1
   endfor
 i = I + 1
Endfor
```
	Since the serial speeds match, the rule succeeds silently.

Table 25. Serial Port Error Checking, Check to Insure that Serial Speeds Match

Rule 13. Configure IP Communications Components by finding the appropriate Component frames and slot values and substituting the values into the template file (**Table 26**).

Pseudo Code	Output

```
i= 1
For each Artifact[i]
j = 1
  For each Artifact[j]
     If artifact[i].name != artifact[j].name
       Then
           If Artifact[i].comms_destination =
                                                       artifact[j].name
           then
               For each component in artifact[i]
                 If component.kind =
                                          ‘UDPCommunications”
                     Then
                      For each component in Artifact[j]
                          If component.kind =
                                          ‘TCPCommunications’
                           Then
        artifact[i].component.destinationIPaddress =
                           artifact[j].component.IPAddress
              artifact[i].component.destinationIPport =
                           artifact[j].component.port
                           Endif
                      Endfor
                  Endif
               Endfor
             Endif
         Endif
    j = j + 1
   Endfor
 i = I + 1
Endfor
```
	Relevant Component frame updates.
	Component[1,2].id = "id-0066-0000000c"
	Component[1,2].name =
	ODISOCUTCPCommunications
	Component[1,2].kind = "TCPCommunications"
	Component[1,2].IPAddress = 10.0.0.3
	Component[1,2].port = 3743
	Component[1,2].destinationIP = 10.0.0.1
	
	Component[2,2].id = "id-0066-00000008"
	Component[2,2].name =
	ODISTCPCommunications
	Component[2,2].kind = "TCPCommunications"
	Component[2,2].IPAddress = 10.0.0.2
	Component[2,2.port = 3743
	Component[2,2].destinationIP = 10.0.0.3
	
	Component[3,3].id = "id-0066-00000005"
	Component[3,3].name =
	BumpControlTCPCommunicationsl
	Component[3,3].kind = "TCPCommunications"
	Component[3,3].IPAddress = 10.0.0.1
	Component[3,3].port = 3743
	Component[3,3].destinationIP = 10.0.0.2

Table 26. Configure IP Component Frame Fields with Proper Communications Constants

Rule 14: Create Constants Files.

Extract information from internal Ruby tables for each artifact in the prototype. Create a file in each node folder, Constants.java and write the appropriate values to each. Constants are used to configure the components for now. (**Table 27**).

Pseudo Code	Output

For each Artifact
  Create a java file
          “artifact.name”Constants
          in the proper working directory
  Open file for writing
  Write the necessary constants
   Close the file
Endfor
	`public class BumpStopConstants {`
	` public static final String`
	`   MESSAGES_IN = M2,M4;`
	` public static final String`
	`   MESSAGES_OUT = M4;`
	` public static final String`
	`    SERIALSPEED = 9600;`
	` public static final String`
	`    SERIALPORT = 1;`
	` public static final String`
	`    DESTINATIONIPADDRESS = 10.0.0.2;`
	` public static final String`
	`    DESTINATIONIPPORT = 3743;`
	` public static final String`
	`    IPADDRESS = 10.0.0.1;`
	` public static final String`
	`    PORT = 3743;`
	`}`

Table 27. Description of Create Constants File Rule

Rule 15: Create Main program.

Create a main program for each node; the main program instantiates components and initiates internal communications via the Java Observer pattern. (**Table 28**).

Pseudo Code	Output

```
For each Artifact
  Create a java file

“artifact.name”Main
          in the proper working
directory
  Open file for writing
  Write the necessary statements
   (instantiate objects, register
observers)
   Close the file
Endfor
```
	```
	public class BumpStopMain {
	  public static void main(String args[]) {
	   BumpStopAdaptor _Adaptor =
	        new BumpStopAdaptor();
	   SerialCommunications
	      _SerialCommunications =
	        new SerialCommunications();
	   UDPCommunications _UDPCommunications =
	        new UDPCommunications();
	   optional _optional =
	        new optional();
	_SerialCommunications.addObserver(_Adaptor);
	_Adaptor.addObserver(_optional);
	_optional.addObserver(_UDPCommunications);
	_UDPCommunications.addObserver(_Adaptor);
	 }
	}
	```

Table 28 . Rule to create a Java Main Program for each Node

Rule 16. Java command line files

Create Java command line files to compile and run each node. Future implementations will include commands to create and execute Java jar files. (**Table 29**).

Pseudo Code	Output
For each Artifact Create a command file to compile javac.bat in the proper working directory Open file for writing Write the command line with the proper Main file name Close the file Create a command file to run java.bat in the proper working directory Open file for writing Write the command line with the proper Main file name Close the file Endfor	javac -classpath ./;../../codelocker/;../../codelocker/jaxb1-impl.jar;../../codelocker/jaxb-api.jar;../../codelocker/jaxb-impl.jar;../../codelocker/jaxb-xjc.jar;../../codelocker/jsr173_1.0_api.jar;../../codelocker/jaus.jar;./../codelocker/bin BumpStopMain.java -d bin java -classpath ./;./jaxb1-impl.jar;./jaxb-api.jar;./jaxb-impl.jar;./jaxb-xjc.jar;./jsr173_1.0_api.jar;./jaus.jar;./bin BumpStopMain

Table 29. Rule to Create DOS .bat Command Files to Execute Java Commands that Compile and Run the Java Wrapper Applications on Each Node

2. Ruby Experiment File – basic_Robot.rb

```
#~ Enter the GME output file (.xme)
puts "Enter filename: "
puts
filename= gets.chomp
puts filename

xml = File.new(filename)

# SciTE Test Case
#~xml = File.new('Basic_Robot_Domain.xme')

#include REXML  necessary Ruby extensions
require 'rexml/document'
require 'pp'
require 'fileutils'
require 'ftools'
doc = REXML::Document.new xml

#-- 1. Create Artifact List:

#-- creates an array of hashes for artifact, adds the artifact IDs
#~ Parse the XML file, find the Artifacts from child models of
#~ the top-level model and create Artifact frames for them
#~ with id, name and kind slots.

artifact = Array.new

i = 0
doc.elements.each('//model/model') { |x|
     artifact[i] = Hash.new
     artifact[i] = {'id' => x.attributes["id"]}
     i = i + 1
}

i = 0
```

```
doc.elements.each('//model/model') { |y|
    y.elements.each('attribute') { |z|
     z.elements.each('value') {|w|
        if z.attributes["kind"] == "CONTROLID"
             then
             artifact[i]['name'] = w.text
             end
        if z.attributes["kind"] == "OCUID"
             then
             artifact[i]['name'] = w.text
             end
        if z.attributes["kind"] == "PlatformType"
             then
             artifact[i]['name'] = w.text
             end
        if z.attributes["kind"] == "SensorName"
             then
             artifact[i]['name'] = w.text
        end
        if z.attributes["kind"] ==  "ManipulatorType"
             then
             artifact[i]['name'] = w.text
             end
             }
             }
             i = i+1
        }
i = 0

doc.elements.each('//model/model/name') { |x|
artifact[i]['displayed_name'] = x.text
i = i + 1
}
```

```
#-- 2. Create Channel  List:

#~ Parse the XML file, find the Channels from child <connection>
#~ tags of the top-level model and create Channel frames for
#~ them with source and destination slots where source and
#~ destination are the "id's" are directly related to <Connpoint>
#~ target and role attributes.

channel = Array.new

i = 0
doc.elements.each("/project/folder/model/connection") { |x|
     channel[i] = Hash.new
     x.elements.each('connpoint') { |y|
           if y.attributes["role"] == "src"
                 then channel[i]['source'] = y.attributes["target"]
           end
            if y.attributes["role"] == "dst"
                 then channel[i]['destination'] = y.attributes["target"]
           end}
     i = i + 1
}

#-- 3. Additional Channel information:

#~ For readability, add Channel[Y].source.name to each Channel Frame
#~ by going through the Artifacts and comparing their id slot to the
#~ Channel[Y].source slot.  If they match, add the Artifact name to a
#~ sub slot of Channel[Y].source, Channel[Y].source.name.
#~ Do same for Channel[Y].destination.

artifact.each { |x| channel.each{ |y|
if x['id'] == y['source']
     then y['source_name'] = x['name']
end
if x['id'] == y['destination']
```

```
     then y['destination_name'] = x['name']
end}
}

#-- 4. Create Artifact Component Lists:

#~ For each Artifact discovered in Rule 1, parse the XML file,
#~ find the child Atoms.  Create Component frames for them with
#~ id, name and kind slots.

component = Array.new

 i= 0
doc.elements.each('/project/folder/model/model') { |x|
  component[i] = Array.new

  j = 0
  x.elements.each('atom') { |y|
    component[i][j] = Hash.new
    component[i][j] = {'id' => y.attributes["id"]}
    component[i][j]['kind'] = y.attributes["kind"]
  j = j + 1
}
j = 0

x.elements.each('atom/name') { |y|
component[i][j]['name'] = y.text
j = j + 1
}
i = i + 1
}

#-- 5 Create  dataflow lists

#~ Parse the XML file, find the Dataflows from child <connection>
#~ tags of the child model (Artifact) of the top-level model and create
#~ Dataflow frames for them with source and destination slots where
#~ source and destination are the "id's" are directly related to <Connpoint>
#~ target and role attribute.

dataflow = Array.new

i = 0
doc.elements.each("/project/folder/model/model") { |x|
     dataflow[i] = Array.new
     j = 0
     x.elements.each("connection") { |y|
     dataflow[i][j] = Hash.new
     y.elements.each('connpoint') { |z|
          if z.attributes["role"] == "src"
               then dataflow[i][j]['source'] = z.attributes["target"]
          end
           if z.attributes["role"] == "dst"
               then dataflow[i][j]['destination'] = z.attributes["target"]
          end}
     j = j + 1
     }
i = i + 1
}

 #-- 6. Additional dataflow information:

#~ For readability, add Dataflow[X,W].source.name to each Dataflow Frame
#~ by going through the Components in the current X and comparing
#~ their id slot to the Dataflow[X,W].source slot.  If they match, add
#~ the Component name to a sub slot of Dataflow[X,W].source,
#~ Dataflow[X,W].source.name.  Do same for Dataflow[X,W].destination.

i = 0
component.each{|w|
     component[i].each {|x|
          dataflow[i].each{ |y|
               if x['id'] == y['source']
                    then y['source_name'] = x['name']
```

```
                        y['source_kind'] = x ['kind']
                  end
                  if x['id'] == y['destination']
                       then y['destination_name'] = x['name']
                       y['destination_kind'] = x ['kind']
                  end

             }
        }
i = i + 1
}

#~ 7. Create Artifact Message Lists:

#~ For each Component atom, if  XML kind attribute is "adaptor,"
#~ search for an <attribute> tag with of kind = "JAUSMessagesIn."
#~ Insert the <attribute> tag's value into the corresponding
#~ Artifact's messages_in slot.  Do same for
#~ kind = "JAUSMessagesOut".

i = 0
doc.elements.each("/project/folder/model/model") { |x|
    x.elements.each("atom") { |y|
         y.elements.each('attribute') { |z|
              z.elements.each('value') {|w|
              if z.attributes["kind"] == "JAUSMessagesIn"
                   then
                   artifact[i]['messages_in'] = w.text
              end

               if z.attributes["kind"] == "JAUSMessagesOut"
                   then
                   artifact[i]['messages_out'] = w.text
              end
              }
         }
    }
```

```
i = i + 1
}

#~ 8. Create Communications Components attribute list:

#~ From the XML file, fill in appropriate slots in the Component data
#~ structures for Communication components, i.e., speed and serial
#~ port for serial communications, IP address or appropriate IP port for
#~ IP communications.

i = 0
doc.elements.each("/project/folder/model/model") { |x|

    j = 0

    x.elements.each("atom") { |y|
         if y.attributes["kind"] == 'SerialCommunications'
              then
              y.elements.each('attribute') { |z|
                   z.elements.each('value') {|w|
                        if z.attributes["kind"] == 'Speed'
                             then
                             component[i][j]['serialspeed'] = w.text
                        end
                         if z.attributes["kind"] == "SerialPort"
                             then
                             component[i][j]['serialport'] = w.text
                        end
              }
         }
         end
         if y.attributes['kind'] == 'TCPCommunications'
              then
              y.elements.each('attribute') { |z|
                   z.elements.each('value') {|w|
                        if z.attributes["kind"] == "IPAddress"
                             then
```

```
                            component[i][j]['IPAddress'] = w.text
                        end
                         if z.attributes["kind"] == "Port"
                            then
                            component[i][j]['Port'] = w.text
                        end
                 }
            }

            end

            if y.attributes['kind'] == 'UDPCommunications'
                 then
                 y.elements.each('attribute') { |z|
                     z.elements.each('value') {|w|
                        if z.attributes["kind"] == "IPAddress"
                            then
                            component[i][j]['IPAddress'] = w.text
                        end
                         if z.attributes["kind"] == "Port"
                            then
                            component[i][j]['Port'] = w.text
                        end
                 }
            }
            end
        j = j + 1
        }
i = i + 1
}

#~ 9. Create Directury structure

#~ Create a file folder for each Artifact using the Artifact's
#~ name slot for a folder name.

#Working Directory hardcoded for now
WORKDIR = 'c:/apex'
```

```
Dir.chdir(WORKDIR)

x = doc.elements["/project[@guid]"]
projectdir = x.attributes["guid"]
projectdir1 = projectdir.sub(/[{]/,'dir')
projectdir = projectdir1.gsub(/[}-]/, '')

if  File.exists? projectdir
     then
     puts "Project Directory exists, removing"
     puts projectdir
     FileUtils.remove_dir projectdir
     puts "removed projectdir"
     end

Dir.mkdir(projectdir)
Dir.chdir(projectdir)

artifact.each{|x|

     d  = x['name']
     Dir.mkdir d
     Dir.mkdir "#{d}/bin"
     Dir.mkdir "#{d}/doc"
}

#~ 10. IMport Files from Code Repository

#~Copy the appropriate files from the repository into the working directory
#~Structure.
#Repository location hard coded for now
CODELOCKER = 'c:/apex/codelocker'

i = 0
doc.elements.each("/project/folder/model/model") { |x|
     j = 0
     x.elements.each("atom") { |y|
        sourcefile = ''
        if y.attributes["kind"] == 'Adaptor'
              sourcefile = ''
              sourcefile = 'adaptors'
              sourcefile = sourcefile.insert(0, '/')
              sourcefile = sourcefile.sub(//, CODELOCKER)
              filename = artifact[i]['name']
              sourcefile = sourcefile.insert(-1, '/')
              sourcefile = sourcefile.insert(-1, filename)
              sourcefile = sourcefile.insert(-1, y.attributes["kind"])
              sourcefile = sourcefile.insert(-1, '.java')

              destfile = 'c:/apex/'
              destfile = destfile.insert(-1, projectdir)
              destfile = destfile.insert(-1, '/')
              destfile= destfile.insert(-1, artifact[i]['name'])
              destfile = destfile.insert(-1, '/')
              destfile = destfile.insert(-1, filename)
              destfile = destfile.insert(-1, y.attributes["kind"])
              destfile = destfile.insert(-1, '.java')

              File.copy sourcefile, destfile

# copy documentation while we are at it.

              sourcefile = sourcefile.sub(/java/,'txt')
              destfile = 'c:/apex/'
              destfile = destfile.insert(-1, projectdir)
              destfile = destfile.insert(-1, '/')
              destfile= destfile.insert(-1, artifact[i]['name'])
              destfile = destfile.insert(-1, '/doc/')
              destfile = destfile.insert(-1, filename)
              destfile = destfile.insert(-1, y.attributes["kind"])
              destfile = destfile.insert(-1, '.txt')
              File.copy sourcefile, destfile

        elsif y.attributes["kind"] == 'Optional'
        sourcefile = ''
        sourcefile = 'Optional'
        sourcefile = sourcefile.insert(0, '/')
        sourcefile = sourcefile.sub(/\//, CODELOCKER)
        filename = 'Optional'
        sourcefile = sourcefile.insert(-1, '/')
        sourcefile = sourcefile.insert(-1, y.attributes["kind"])
        sourcefile = sourcefile.insert(-1, '.java')

        destfile = 'c:/apex/'
        destfile = destfile.insert(-1, projectdir)
        destfile = destfile.insert(-1, '/')
        destfile= destfile.insert(-1, artifact[i]['name'])
        destfile = destfile.insert(-1, '/')
        filename = component[i][j]["kind"]
        destfile = destfile.insert(-1, filename)
        destfile = destfile.insert(-1, '.java')

            File.copy sourcefile, destfile

        sourcefile = sourcefile.sub(/java/, 'txt')

        destfile = 'c:/apex/'
        destfile = destfile.insert(-1, projectdir)
        destfile = destfile.insert(-1, '/')

        destfile= destfile.insert(-1, artifact[i]['name'])
        destfile = destfile.insert(-1, '/doc/')
        filename = component[i][j]["name"]
        destfile = destfile.insert(-1, filename)
        destfile = destfile.insert(-1, '.txt')

            File.copy sourcefile, destfile


    else
        sourcefile =''
```

```
          sourcefile = 'Communications'
          sourcefile = sourcefile.insert(0, '/')
          sourcefile = sourcefile.sub(/\//, CODELOCKER)
          sourcefile = sourcefile.insert(-1, '/')
          sourcefile = sourcefile.insert(-1, y.attributes["kind"])
          sourcefile = sourcefile.insert(-1, '.java')

          destfile = 'c:/apex/'
          destfile = destfile.insert(-1, projectdir)
          destfile = destfile.insert(-1, '/')
          destfile = destfile.insert(-1, artifact[i]['name'])
          destfile = destfile.insert(-1, '/')
          destfile = destfile.insert(-1, y.attributes["kind"])
          destfile = destfile.insert(-1, '.java')

               File.copy sourcefile, destfile

          sourcefile = sourcefile.sub(/java/,'txt')
          destfile = 'c:/apex/'
          destfile = destfile.insert(-1, projectdir)
          destfile = destfile.insert(-1, '/')
          destfile = destfile.insert(-1, artifact[i]['name'])
          destfile = destfile.insert(-1, '/doc/')
          destfile = destfile.insert(-1, y.attributes["kind"])
          destfile = destfile.insert(-1, '.txt')

               File.copy sourcefile, destfile

          end
j = j + 1
}
destdir = 'c:/apex/'
destdir = destdir.insert(-1, projectdir)
destdir = destdir.insert(-1, '/')
destdir = destdir.insert(-1, artifact[i]['name'])
destdir=  destdir.insert(-1, '/')
sourcedir = ''
```

```
     sourcedir = sourcedir.sub(//, CODELOCKER)

     Dir.entries(sourcedir).each  {|name|

          if File.fnmatch('*.jar', name)

               sourcefile = ''
               sourcefile = sourcefile.insert(0, sourcedir)
               sourcefile = sourcefile.insert(-1, '/')
               sourcefile = sourcefile.insert(-1, name)

               destfile = ''
               destfile = destfile.insert(0, destdir)
               destfile = destfile.insert(-1, '/')
               destfile = destfile.insert(-1, name)

               File.copy sourcefile, destfile
          end

          }
 i = i + 1
}

#~ 11.  Set Artifact Comms destination element.

#~ Match artifact names to channel source names.  Set artifact communications
#~ destination to channel destination name.

i = 0
artifact.each { |x|
     j = 0

     channel.each { |y|

          if y["source_name"] == x["name"]
               then
               x["comms_destination"] = y["destination_name"]
          end
```

```
      j = j + 1
      }
i = i + 1
}

#~ 12. Error check to make sure serial speeds match

#~ For each Artifact, check child components for SerialCommunications
#~ components.  If found, find a Channel frame that has the source
#~ matching the Artifact.  Find the destination Artifact from the channel
#~ destination slot.  In the destination Artifact, find the SerialCommunications
#~ Component and check the appropriate destination Component slot to
#~ insure they match.

i = 0
artifact.each{ |x|
      j = 0
      artifact.each { |y|
            if x["name"] != y["name"]
                  if x["comms_destination"] == y["name"]
                        component[i].each { |w|
                        if w["kind"] =="SerialCommunications"
                              component[j].each {|z|
                               if z["kind"] =="SerialCommunications"
                                    if z["serialspeed"] != w["serialspeed"]
                                          then
                                           print('Source component ', w['name'], '\n')
                                           print('destination component ', z['name'], '\n')
                                           print("SERIAL SPEED MISMATCH\n")
                                           print("Between Source Port ", w['serialport'], " on ", w['name'], " and\n")
                                           print("Destination Port ", w['serialport'], " on ", z['name'], "\n")
                                           print("-Please repair \n")
                                          else
                                          print("SERIAL SPEEDS MATCH\n")
                                          print("Between Source Port ", w['serialport'], " on ", w['name'], " and\n")
                                          print("Destination Port ", w['serialport'], " on ", z['name'], "\n")
                                    end
```

```
                                        end
                                        }
                                   end
                                   }
                              end

                         end
                    j = j + 1
                    }
               i = i +1
               }

#~ 13. Set up destination IP addresses and Ports in IP Communication Components

#~ Configure TCPCommunications Components by finding the appropriate
#~ Component frames and slot values and substituting the values into the
#~ proper slots.

i = 0
artifact.each{ |x|
     j = 0
     artifact.each { |y|
          if x["name"] != y["name"]
               if x["comms_destination"] == y["name"]
                    component[i].each { |w|
                    if w["kind"] =="TCPCommunications"
                         component[j].each {|z|
                          if z["kind"] =="TCPCommunications"
                              then
                              w["destinationIPaddress"] = z["IPAddress"]
                              w["destinationIPport"] = z["Port"]
                                  print('Source component ', w['name'], "\n")
                                  print('destination component ', z['name'], "\n")
                                  print(w['destinationIPport'], " on ", w['destinationIPaddress'], " \n")
                             end
                          }
                    end
                    }
```

```
                end
            end
        j = j + 1
        }
i = i +1
}
i = 0
artifact.each{ |x|
     j = 0
     artifact.each { |y|
          if x["name"] != y["name"]
               if x["comms_destination"] == y["name"]
                    component[i].each { |w|
                    if w["kind"] =="UDPCommunications"
                         component[j].each {|z|
                         if z["kind"] =="UDPCommunications"
                                then
                                w["destinationIPaddress"] = z["IPAddress"]
                                w["destinationIPport"] = z["Port"]
                                    print('Source component ', w['name'], "\n")
                                    print('destination component ', z['name'], "\n")
                                    print(w['destinationIPport'], " on ", w['destinationIPaddress'], " \n")
                         end
                         }
                      end
                      }
                end
            end
        j = j + 1
        }
 i = i +1
 }

#~ 14. Create Constants Files

#~ Create a constants files in the working directory for each node to configure
#~components using appropriate values from artifact and component hash tables.
```

```
 i = 0
artifact.each{ |x|
     destfile = 'c:/apex/'
     destfile = destfile.insert(-1, projectdir)
     destfile = destfile.insert(-1, '/')
     destfile = destfile.insert(-1, artifact[i]['name'])
     destfile = destfile.insert(-1, '/')
     #destfile = destfile.insert(-1, x["name"])
     destfile = destfile.insert(-1, 'Constants.java')

     open(destfile, 'w') do |f|
          #f << "public class " << x["name"] << "Constants {\n"
          f << "public class " << "Constants {\n"
           if artifact[i]["messages_in"] != nil
                then
                f << "public static final String MESSAGES_IN = "  << '"' << artifact[i]["messages_in"]  << '"' << ";\n"
           end
            if artifact[i]["messages_out"] != nil
                then
          f << "public static final String MESSAGES_OUT = "  << '"' << artifact[i]["messages_out"]  << '"' << ";\n"
          end
     j = 0
     component[i].each{|y|
          if y["kind"] == "TCPCommunications"
               then
               f << "public static final String DESTINATIONIPADDRESS = " <<  '"' << y["destinationIPaddress"]
                                                                                      <<'"' << ";\n"
               f << "public static final String DESTINATIONPORT = " <<'"'  << y["destinationIPport"] <<'"' << ";\n"
               f << "public static final String IPADDRESS = " <<'"' << y["IPAddress"] <<'"' << ";\n"
               f << "public static final String PORT = " <<'"' << y["Port"] <<'"' << ";\n"
          end

          if y["kind"] == "UDPCommunications"
               then
               f << "public static final String DESTINATIONIPADDRESS = " <<'"' << y["destinationIPaddress"]
                                                                                     <<'"' << ";\n"
               f << "public static final String DESTINATIONPORT = "  <<'"' << y["destinationIPport"] <<'"' << ";\n"
```

```
                f << "public static final String IPADDRESS = " <<'"' << y["IPAddress"] <<'"' << ";\n"
                f << "public static final String PORT = " <<'"' << y["Port"] <<'"' << ";\n"
            end

            if y["kind"] == "SerialCommunications"
                then

                f << "public static final String SERIALSPEED = " <<'"' << y["serialspeed"] <<'"' << ";\n"
                f << "public static final String SERIALPORT = " <<'"' << y["serialport"] <<'"' << ";\n"
            end
        j = j + 1
        }

        f << "}\n"
        f.flush
        end
    i = i + 1
    }

#~ Rule 15. main program

#~ Create Main programs for each node. Instantiate objects and
#~ initiate observers.

i = 0
artifact.each{ |x|
    destfile = 'c:/apex/'
    destfile = destfile.insert(-1, projectdir)
    destfile = destfile.insert(-1, '/')
    destfile = destfile.insert(-1, artifact[i]['name'])
    destfile = destfile.insert(-1, '/')
    destfile = destfile.insert(-1, x["name"])
    destfile = destfile.insert(-1, 'Main.java')

    open(destfile, 'w') do |f|
        f << "public class " << x["name"] << "Main {\n"
        f << "public static void main(String args[]) { \n"
```

```
     j = 0
     component[i].each{|y|
          if y["kind"] == "Adaptor"
               then
                    f << x["name"] << y["kind"] << " " << "_" << y["name"] << " = new "
                                                                      << x["name"] << y["kind"] << "();\n"
               else
                    f <<  y["kind"] << " " << "_" << y["name"] << " = new "  << y["kind"] << "();\n"
            end
     }
     dataflow[i].each{|y|
          f << "_" << y["source_name"] << ".addObserver(" << "_" << y["destination_name"] << ");\n"
     j = j + 1
     }

     f << "}\n"
     f << "}\n"
     f.flush
     end
i = i + 1
 }

#Rule 16. Java command line files

#~Create Java command line files to compile and run each node.

i = 0
artifact.each{ |x|
     destfile = 'c:/apex/'
     destfile = destfile.insert(-1, projectdir)
     destfile = destfile.insert(-1, '/')
     destfile = destfile.insert(-1, artifact[i]['name'])
     destfile = destfile.insert(-1, '/')
     destfile = destfile.insert(-1, 'runjavac.bat')

     open(destfile, 'w') do |f|
          filestring = ''
          filestring = filestring.insert(0, "javac -classpath ")
          filestring = filestring.insert(-1, "./;")
          filestring = filestring.insert(-1, "../../codelocker/;")
          filestring = filestring.insert(-1, "../../codelocker/jaxb1-impl.jar;")
          filestring = filestring.insert(-1, "../../codelocker/jaxb-api.jar;")
          filestring = filestring.insert(-1, "../../codelocker/jaxb-impl.jar;")
          filestring = filestring.insert(-1, "../../codelocker/jaxb-xjc.jar;")
          filestring = filestring.insert(-1, "../../codelocker/jsr173_1.0_api.jar;")
          filestring = filestring.insert(-1, "../../codelocker/jaus.jar;")
          filestring = filestring.insert(-1, "./../codelocker/bin ")
          filestring = filestring.insert(-1, x["name"] )
          filestring = filestring.insert(-1, "Main.java")
          filestring = filestring.insert(-1, " -d bin")

          f << filestring
      f.flush
      end
      destfile = destfile.sub("javac", "java")
      open(destfile, 'w') do |f|
           filestring = ''
          filestring = filestring.insert(0, "java -classpath ")
          filestring = filestring.insert(-1, "./;")
          filestring = filestring.insert(-1, "./jaxb1-impl.jar;")
          filestring = filestring.insert(-1, "./jaxb-api.jar;")
          filestring = filestring.insert(-1, "./jaxb-impl.jar;")
          filestring = filestring.insert(-1, "./jaxb-xjc.jar;")
          filestring = filestring.insert(-1, "./jsr173_1.0_api.jar;")
          filestring = filestring.insert(-1, "./jaus.jar;")
          filestring = filestring.insert(-1, "./bin ")
          filestring = filestring.insert(-1, x["name"] )
          filestring = filestring.insert(-1, "Main")
          f << filestring
       f.flush
     end
 i = i + 1
}
```

THIS PAGE INTENTIONALLY LEFT BLANK

LIST OF REFERENCES

[1] "DoD Integrated Product and Process Development Handbook," Office of the Under Secretary of Defense (Acquisition and Technology), Washington, DC 20301-3000, http://akss.dau.mil/docs/026EV001DOC.doc, 09 May 2007.

[2] Wired Magazine, December, 2004: "More Robot Grunts Ready for Duty" http://www.wired.com/news/technology/0,1282,65885,00.html?tw=wn_tophead_1, 09 April 2007.

[3] W. Smuda, P. Muench, G. Gerhart, K. Moore, "Autonomy and Manual Operation in a Small Robotic System for Real-time Inspections at Security Checkpoints," SPIE Defense & Security Symposium, Orlando, FL, April 2002.

[4] C. LickTeig, W. Sanders, P. Durlach, J. Lussier, "Measuring Human Performance in Battle Command Army AL&T, May-June 2005, pp. 16-20.

[5] L. Ojeda L., J. Borenstein, "FLEXnav: Fuzzy Logic Expert Rule-based Position Estimation for Mobile Robots on Rugged Terrain," Proceedings of the 2002 IEEE International Conference on Robotics and Automation. Washington DC, USA, 10 - 17 May 2002, pp. 317-322.

[6] W. Smuda, L. Freiburger, H. Andrusz, J. Overholt , G. Gerhart, D. Gorsich, "Rapid Infusion Of Army Robotics Technology For Force Protection & Homeland Defense", Army Science Conference, Orlando, FL, December 2002.

[7] "Traffic Safety Facts," 2004, National Traffic Safety Association (NHTSA), http://www-nrd.nhtsa.dot.gov/pdf/nrd-30/NCSA/TSFAnn/TSF2004.pdf, 09 April 2007.

[8] B. Boehm, "A Spiral Model of Software Development and Enhancement," Computer, pp. 61-72, May 1988.

[9] L. Freiburger, W. Smuda, R. Karlsen, S. Lakshmanan, "ODIS the Under-vehicle Inspection Robot – Development Status Update," SPIE Defense & Security Symposium, Orlando, FL, April 2003.

[10] W. Smuda, L. Freiburger, G. Gerhart, L. Mallon, "Robotics for Port Security," SPIE Defense & Security Symposium, Orlando, FL, April 2004.

[11] E. Schoenherr, W. Smuda, "Real-time Autonomous Inspection through Undercarriage Signatures," SPIE Defense & Security Symposium, Orlando, FL, April 2005.

[12] J. Overholt, G. Hudas, C.K. Cheok, "A Modular Neural-Fuzzy Controller for Autonomous Reactive Navigation," NAFIPS 2005, Soft Computing for Real World Applications, Ann Arbor, MI. 22-25 June 2005.

[13] W. Smuda, "Software Requirements Specification (SRS), Track Vehicle Workstation (TVWS), General Test Utility Subsystem (genx)" TARDEC Technical Report, October 1993.

[14] F. Klassner, "A case study of LEGO Mindstorms'™ suitability for artificial intelligence and robotics courses at the college level," Technical Symposium on Computer Science Education, Proceedings of the 33rd SIGCSE technical symposium on Computer science education, Cincinnati, Kentucky, February 27 - March 03, 2002, pp. 8-12.

[15] News 2.0, ACM Queue, Volume 5, Issue 1, February 2007, p. 8.

[16] J. Krikke, IEEE Intelligent Systems, September/October 2006.

[17] Microsoft Robotics Studio, http://msdn.microsoft.com/robotics/, 7 May 2007.

[18] K. Chang, S. Cohen, J. Hess, W. Nowak, S. Peterson, "Feature Oriented Domain Analysis (FODA) Feasibility Study," Technical Report, CMU/SEI-90-TR-21, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA, November 1990.

[19] K. Czarnecki, U. Eisenecker, "Generative Programming: Methods, Tools, and Applications," Boston, MA, Addison-Wesley, 2000.

[20] B. Dawes, "Feature Model Diagrams in text and HTML," http://www.boost.org/more/feature_model_diagrams.htm, 09 April 2007.

[21] "Model-driven Architecture (MDA)," Document number ormsc/2001-07-01 Architecture Board ORMSC1 July 9, 2001, http://www.omg.org/mda, 09 April 2007.

[22] K. Czarnecki, T. Bednasch, P. Unger, U. Eisenecker, “Generative Programming for Embedded Software: An Industrial Experience Report,” Proceedings ACM SIGPLAN/SIGSOFT Conference, GPCE, Pittsburgh, PA, October 2002.

[23] “An SAE AADL Language Overview,” SAE AADL Information Site, http://la.sei.cmu.edu/aadlinfosite/AnSAEAADLLanguageOverview.html, 10 May 07.

[24] R. Pierce, “Leveraging Technology Affinity: Applying a Common Set of Tools and Practices to Information Development,” SIGDOC’05, 21-23 September 2005, Coventry, United Kingdom.

[25] Y. Li, J. Landay, Z. Guan, X. Ren, G. Dai, G., “Sketching Informal Presentations,” *ICMI’03*,5-7 November 2003, Vancouver, British Columbia, Canada.

[26] Luqi, “Software Evolution Through Rapid Prototyping,” IEEE Computer, May 1989, pp. 13-25.

[27] Joint Ground Robotics Enterprise Overview, http://www.ndia.org/Content/ContentGroups/Divisions1/Robotics/Purdy.pdf, 09 April 2007.

[28] Public Law 106-398; 114 Stat. 1654A-38.

[29] SEI Software Architecture, http://www.sei.cmu.edu/ata/ata_init.html, 09 April 2007.

[30] N. Rozanski, E. Woods, “Software Systems Architecture: Working With Stakeholders Using Viewpoints and Perspectives,” Addison Wesley Professional, 2005.

[31] D. D’Souzaand and A.Wills , “Objects Components and Frameworks with UML,” Addison Wesley, 1999.

[32] R. Kazman, **“**Handbook of Software Engineering and Knowledge Engineering,” December 2001, ftp://cs.pitt.edu/chang/handbook/15.pdf, 09 April 2007.

[33] M. Simos, D. Creps, C. Klinger, L. Levine, and D. Allemang, “Organization Domain Modeling (ODM) Guidebook,” Version 2.0. Informal Technical Report for STARS, STARS-VC-A025/001/00, 14 June 1996.

[34] D. Schmidt, "Model-driven Engineering," IEEE Computer, February 2006, pp. 25-31.

[35] "Joint Architecture for Unmanned Systems," http://www.jauswg.org, 09 April 2007.

[36] NATO Working Group, STANAG 4586 "Standard Interface of the Unmanned Control System (UCS) for NATO UAV interoperability." http://www.cdlsystems.com/stanag.html, 10 May 2007.

[37] "IEEE standard for distributed interactive simulation – applicationprotocols," http://ieeexplore.ieee.org/Xplore/login.jsp?url=/iel1/3700/10849/00499701.pdf?arnumber=499701, 10 May 2007.

[38] D. McGregor, D. Brutzman, C. Blais, A. Arnold, M. Falash, E. Pollak, "DIS-XML: Moving to Open Data Exchange Standards," Proceedings of the Simulation Interoperability Standards Organization (SISO) Spring 2006 Simulation Interoperability Workshop, Huntsvill, AL, April 2006.

[39] D. Davis, "Design Implimentation and Testing of a Common Data Model Supporting Autonomous Vehicle Compatibiliy and Interoperabiliyt," Naval Postgraduate School Dissertation, September 2006.

[40] C. Szyperski, "Component Technology - What, Where, and How?" *icse*, p. 684, 25th International Conference on Software Engineering (ICSE'03), 2003.

[41] C. Szyperski, "Component Software, Beyond Object Oriented Programming," Boston, MA, Addison-Wesley, 2002.

[42] K. Czarnecki, and U.W. Eisenecker, "Generative Programming: Methods, Tools, and Applications." Boston: Addison-Wesley, 2000.

[43] E. Gamma, R. Helm, R. Johnson, J. Vlissides, "Design Patterns," Boston, MA, Addison-Wesley, 1995.

[44] E. Freeman, "Head First Design Patterns," Sebastopol, CA, O'Reilly, 2004.

[45] D. Schmidt, M. Stal, H. Rohnert, F. Buschmann, " Pattern-Oriented Software Architecture, Vol 2, Patterns for Concurrent and Networked Objects," West Sussex, England, 2000.

[46] CMSC491D Design Patterns In Java, http://www.research.umbc.edu/~tarr/dp/fall00/cs491.html, 09 April 2007.

[47] Generic Modeling Environment, http://www.isis.vanderbilt.edu/projects/gme, 09 April 2007.

[48] K. Czarnecki, M. Antkiewicz, C. Hwan, P. Kim, S. Lau, K. Pietroszek, "Model-Driven Software Product Lines," OOPSLA '05, San Diego, CA, October 2005

[49] Eclipse, http://www.eclipse.org/, 09 April 2007.

[50] B. Tarr, "Design Patterns in Java, The Observer Pattern," http://www.research.umbc.edu/~tarr/dp/lectures/Observer.pdf, 09 April 2007.

[51] Java^{tm} 2 platform standard edition 5.0 API specification, HTTP://JAVA.SUN.COM/J2SE/1.5.0/DOCS/API/, 09 April 2007.

[52] B. Tarr, "Design Patterns in Java, The Adaptor Pattern," http://www.research.umbc.edu/~tarr/dp/lectures/Adapter.pdf, 09 April 2007.

[53] XML**,** Java Architecture for XML Binding (JAXB), http://Java.sun.com/webservices/jaxb/, 09 April 2007.

[54] Matlab, http://www.mathworks.com/, 09 April 2007.

[55] Labview, http://www.ni.com/labview/, 09 April 2007.

[56] A. Ledeczi, M. Maroti, A. Bakay, G. Karsai, J. Garrett, C. Thomason, G. Nordstrom, J. Sprinkle, P. Volgyesi, "The Generic Modeling Environment," Proceedings of IEEE WISP'2001, Budapest, Hungary, May 2001.

[57] D. Perry, A. Wolf , "Foundations for the study of Software Architecture," ACM Software Engineering Notes, vol 17, no 4, 1992, pp. 40-52.

[58] J. Borenstein, "User Guide for the Micro-Controller Interface Board (MCIB)," University of Michigan, 1999, http://www.eecs.umich.edu/~johannb/MCIB_User_Guide.pdf, 09 April 2007.

[59] C. Marcus, "Prolog Programming," Addison-Wesley, Reading, MA, 1986.

[60] NPS AuvWorkbench, https://savage.nps.edu/AuvWorkbench/install.htm, 24 May 2007.

[61] K. Dale, “Prolog as a Ruby DSL,” http://www.kdedevelopers.org/node/2369, 10 May 2007.

[62] “Logic programming in Ruby: a tiny prolog interpreter and symbolic computation”, http://eigenclass.org/hiki.rb?tiny+prolog+in+ruby, 10 May 2007.

[63] E. Kidd, “Why Ruby is an acceptable LISP,” http://www.randomhacks.net/articles/2005/12/03/why-ruby-is-an-acceptable-lisp, 10 May 2007.

[64] “Project Sun Spot,” http://www.sunspotworld.com/, 10 May 2007.

[65] Sun Developer’s Network, Java ME Technology, http://java.sun.com/javame/index.jsp, 10 May 2007.

[66] The Goals 2000: Educate America Act, Public Law 103-227.

[67] “First,” http://www.usfirst.org/, 10 May 2007.

INITIAL DISTRIBUTION LIST

1. Defense Technical Information Center
   Ft. Belvoir, VA

2. Dudley Knox Library
   Naval Postgraduate School
   Monterey, CA

3. Professor Mikhail Auguston
   Department of Computer Science
   Naval Postgraduate School
   Monterey, CA

4. Professor Luqi
   Department of Computer Science
   Naval Postgraduate School
   Monterey, CA

5. Associate Professor Don Brutzman
   Department of Undersea Warfare
   Naval Postgraduate School
   Monterey, CA

6. Professor Kevin Squire
   Department of Computer Science
   Naval Postgraduate School
   Monterey, CA

7. Dr. Jim Overholt
   Director, Joint Center for Unmanned Ground Systems
   US Army, TARDEC
   Warren, MI

8. MOVES Institute
   Naval Postgraduate School
   Monterey, CA

9. Technical Information Center
   US Army TARDEC
   Warren, MI