ROSMOD: A Toolsuite for Modeling, Generating, Deploying, and Managing Distributed Real-time Component-based Software using ROS

Pranav Srinivas Kumar, William Emfinger,
Amogh Kulkarni and Gabor Karsai*
* Institute for Software Integrated Systems,
Electrical Engineering and Computer Science, Vanderbilt University,
Nashville, TN 37235, USA
Email:{pkumar, emfinger, kulkaras, gabor}@isis.vanderbilt.edu

Dexter Watkins, Benjamin Gasser,
Cameron Ridgewell and Amrutur Anilkumar†
† Mechanical Engineering, Vanderbilt University
Nashville, TN 37235, USA
Email:{dexter.a.watkins, benjamin.w.gasser,
cameron.p.ridgewell, amrutur.v.anilkumar}@vanderbilt.edu

***Abstract*—This paper presents ROSMOD, a model-driven component-based development tool suite for the Robot Operating System (ROS). ROSMOD is well suited for the design, development and deployment of large scale distributed applications on embedded hardware devices. We present the various features of ROSMOD including the modeling language, the graphical user interface, code generators and deployment infrastructure. We describe the utility of this tool with a real-world case study - An Autonomous Ground Support Equipment (AGSE) robot that was designed and prototyped using ROSMOD for the NASA Student Launch competition, 2014-2015.**

I. Introduction

Robot Operating System (ROS) [1] is a meta-operating system framework that facilitates robotic system development. ROS is widely used in various fields of robotics including industrial robotics, UAV swarms and low-power image processing devices. The open source multi-platform support of ROS has made it a requirement in several DARPA robotics projects including the *DARPA Robotics Challenge* [2].

ROS enables the deployment of a network of interacting ROS *nodes*, that communicate using the ROS middleware infrastructure. ROS nodes can contact each other using different types of interaction patterns including synchronous remote method invocation (RMI) and asynchronous message passing publish-subscribe interactions. A ROS application is a packaged set of ROS nodes that communicate through a ROS Master, where a ROS Master is a single discovery and communications broker that facilitates node-node flow setup.

This paper describes ROSMOD [3], an open source development tool suite and run-time software platform for rapid prototyping component-based software applications using ROS. Using ROSMOD, an application developer can create, deploy and manage ROS applications for distributed real-time embedded systems. We define a strict component model, a model-driven development workflow and run-time management tools to build and deploy component-based software with ROS.

The utility of ROSMOD is described using a real-world case study - An Autonomous Ground Support Equipment (AGSE) robot that was designed, prototyped and deployed for the NASA Student Launch Competition [4] 2014-2015. We describe the challenges and requirements, the robotic design, the software prototyping and the overall performance that lead us to winning this competition.

The sections in this paper are organized as follows. Section II describes the ROSMOD tool suite: The component model, modeling language, graphical user interface, code generators and deployment infrastructure. Section III describes our AGSE robot and evaluates ROSMOD. Section IV describes our current efforts to improving ROSMOD support with analysis tools and run-time reconfiguration solutions. Section V presents related efforts in the field of robotics, both using ROS and other similar middleware platforms. Finally, Section VI presents concluding remarks.

II. ROSMOD Tool Suite

A. Component Model

Software development using ROSMOD is influenced by the principles of Component-based Software Engineering [5][6]. Large and complex systems are built by assembling reusable software pieces called *components*. ROSMOD Components contain executable code that implement functions, manipulate state variables and interact with other components in the applications. This model is inspired by our previous efforts with the F6COM Component Model [7]. The architecture of a ROSMOD component is shown in Figure 1.

ROSMOD components can contain (1) publishers, (2) subscribers, (3) clients, (4) servers and (5) timers. Publisher ports publish (without blocking) messages of a message topic, and Subscriber ports subscribe to a message topic. Client ports require the services offered by server ports. Server ports provide services that can be used by the external world. Lastly, periodic and sporadic timers are used to trigger components. The semantics of these communication patterns are dictated by ROS.

To facilitate interactions with other components in an ordered manner, each component has a *Component Message Queue*. This queue holds requests (i.e. messages) received from other interacting components or from the component's timers. These requests are processed by a single thread (per component) called the *Component Executor Thread*. This thread is

therefore responsible for executing all triggered callbacks e.g. subscriber callbacks, server callbacks and timer callbacks.

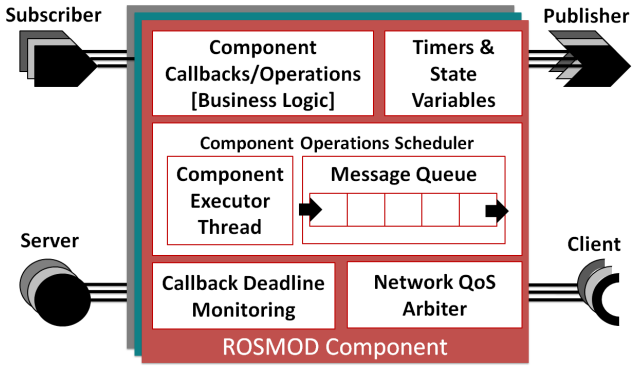

Fig. 1: ROSMOD Component

Figure 2 shows a simple Client-Server component interaction. An Image Processor component is periodically triggered by a timer. At each timer expiry, this component, using its client port, makes a blocking remote procedure call to a Camera component. This service request is enqueued onto the Camera's message queue, and, when it reaches the front of the queue, the Camera component executor thread executes the corresponding server callback, returning the response to the Image Processor. This message queue-based interaction is also true for timers; when the timer in the Image Processor expires, a timer callback request is enqueued onto its message queue and eventually processed.

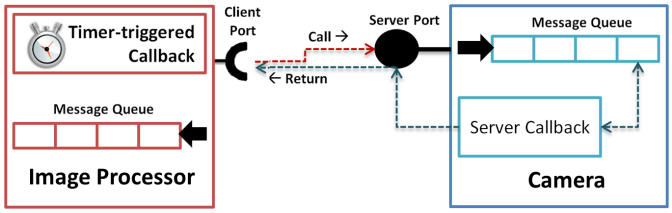

Fig. 2: Component Interactions

It must be noted here that in each component, the message queue is processed by a single executor thread. Multiple components may run concurrently but each component's execution is single-threaded. Also, the component message queue supports several scheduling schemes including FIFO (first-in first-out), PFIFO (priority first-in first-out) and EDF (earliest deadline first). Requests in the message queue are processed using a non-preemptive scheduling scheme. This means that each callback/operation run by the executor thread is run to completion before the next one (waiting in the message queue) is processed. These rules are strictly applied to all ROSMOD components.

The single-threaded component execution is an important choice as it allows robust application development that is free of race conditions. Application integrators can avoid using synchronization primitives and locking mechanisms while developing code and this greatly simplifies design. These choices also more easily enable support for non-functional properties such as fault isolation and tolerance, operation timeliness and component lifecycle management.

B. Modeling Language

ROSMOD Projects are built using the ROSMOD Modeling Language. With this language, ROS users can create models of ROS workspaces, hardware topologies, deployment plans and more. The tool suite provides a Graphical User Interface to build these models but the state and configuration properties of the project are saved in a set of text files (models) that follow a strict set of grammatical rules, written using Antlr 4 [8]. Figure 3 shows the metamodel of the textual modeling language as a UML [9] class diagram.

1) ROS Background: ROS workspaces are high-level containers for source code which may contain one or more ROS packages. ROS packages are containers which may include (1) one or more ROS message definitions for asynchronous publish/subscribe, (2) one or more ROS service definitions for synchronous RMI, and (3) one or more ROS Nodes, which are processes that can communicate with each other using predefined ROS messages or ROS services. In this way a ROS package can be thought of as an application, and a ROS node is a process in that application.

2) Software Model: The ROSMOD Software model represents a ROS workspace. Each model consists of one or more ROS packages. Each ROS package contains definitions to (1) messages, (2) services and (3) components. The component assembly is derived from the interacting component ports. Ports that are associated with callbacks e.g. subscribers, contain both a *priority* and a *deadline* property to facilitate the scheduling schemes in the component model.

3) Hardware Model: Hardware models completely describe the hardware architecture of the system. Here, the user describes the different hardware hosts available for deployment, including their properties such as IP address, username and SSH keys. These properties allow the user to directly map executables to hardware in a specified network and allow the deployment infrastructure to manage all remote operations and help ensure security between applications. Deployment models refer to such predefined hardware models when mapping processes to hardware devices. Current work aims to improve on this hardware model by adding concepts for subnets, network interface controllers (NIC) and network links between hardware devices to more accurately represent the network topology.

4) Deployment Model: ROSMOD Deployment models contain the specifications for ROS nodes (executable processes). Each ROS node is ranked by a process priority and is mapped to a specific hardware device on which it will be executed. Each ROS node contains instances of software components that control its behavior. These component instances refer to specific components defined in the Software Model. At run-time, each node creates one executor thread per component instance before beginning its interaction with the rest of the application.

a) Group Assignment: As shown in Figure 3, each Component Instance can have *port overrides*. These definitions override the ports previously defined in the Software Model. By assigning certain ports to a *group*, a logical grouping of component ports is achieved. This ensures a strong coupling between ports.

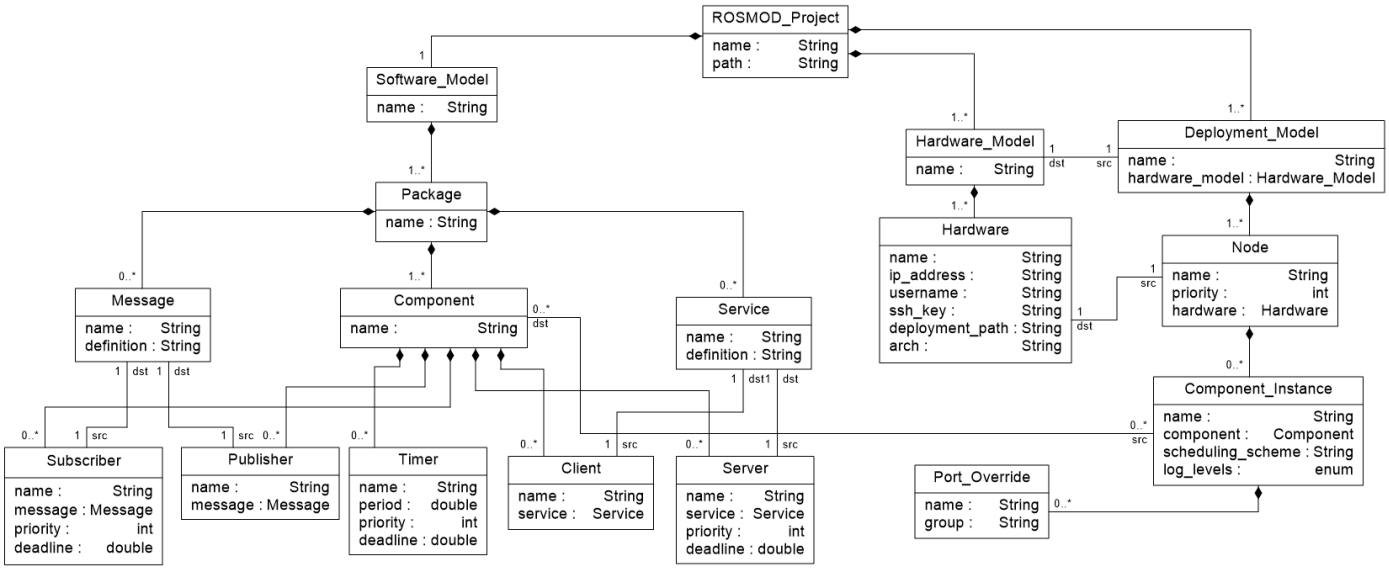

Fig. 3: ROSMOD Project

Suppose an *ImageProcessor* client required a *Camera* service, and this service was provided by two servers - *LowResCamera* and *HighResCamera*. Upon deployment, ROS typically couples the client with the server that advertises first. Although this is the default behavior, users can ensure a strong coupling between ports e.g. the ImageProcessor client connects only to the LowResCamera. This coupling overrides the default behavior, as seen in the Software Model.

C. Graphical User Interface

For large-scale applications, editing text files to describe ROSMOD Projects can be difficult and error prone, especially when referencing model objects defined in multiple files. To ease this development, we have built a Python-based graphical user interface, providing a rendering platform to quickly prototype models and understand the relationship between model elements e.g. component ports.

This integrates well with our deployment infrastructure which opens up interfaces required to build ROS workspaces and deploy node executables. Users can build packages, copy deployment files, *start a deployment*, monitor running ROS nodes and open component instance-specific logs, all from the ROSMOD GUI.

D. Generators

There are currently two classes of generators in ROSMOD.

1) Skeleton ROS Workspace: The workspace generators produce a prototype skeleton ROS workspace. This includes (1) C++ classes for each ROSMOD component, (2) package-specific messages and services, (3) Logger and XML parser-specific files and (4) build system files, all organized following ROS package organization guidelines. Figure 4 shows a sample generated code tree. This is the *motor_control* package used in our AGSE robot. There are three components - *radial_actuator_controller*, *vertical_actuator_controller* and a high-level *servo_controller*. The same figure shows the generated code specific to this package, organized following ROS package guidelines.

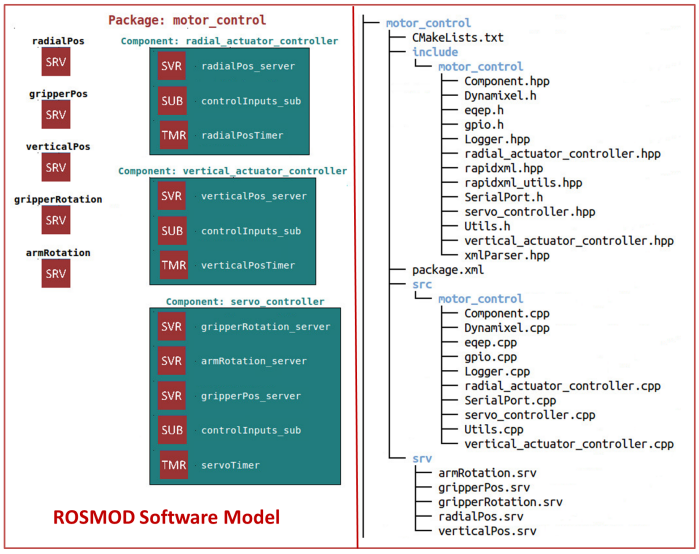

Fig. 4: Workspace Code Generation

In each package, the generated code includes code preservation markers around all callbacks and build system files so that users can quickly add new pieces of code which are guaranteed to be preserved after regeneration. This means that users can, for example, (1) generate a ROS workspace for a ROSMOD Software model, (2) add *business logic* code in the generated skeleton callbacks, (3) go back to the models and add additional ports to specific ROSMOD components as required, and (4) regenerate the ROS workspace ensuring preservation of business logic code and selective code additions while accounting for the newly introduced ports. Therefore, users do not need to complete the ROSMOD models to begin C++ code development, enabling on-the-fly feature

additions to ROS applications.

2) Deployment-specific XML files: The XML generators produce a batch of configuration files per deployment model. These files are fed to the node executable at run-time to easily change its behavior. Deployment-specific XML files contain properties of component instances in each ROS node, as seen in the deployment model in Figure 3. It is typically desired to have knobs to easily tweak component properties e.g. message queue scheduling scheme, logging levels etc. at run-time without having to rebuild the ROS application.

E. Deployment Infrastructure

The workflow for software deployment is as shown Figure 5. Every ROS workspace is generated with an additional *node* package. This builds a generic node executable that can dynamically load libraries. Once the generators generate the ROS workspace and deployment XML files, users complete application development and build their ROS workspace. The build process generates dynamically loadable libraries, one for each component definition along with a single executable corresponding to the generic node package. The generated XML files contain metadata about about all ROS nodes modeled in the ROSMOD Deployment Model. This includes the component instances in each node and the appropriate component libraries to be loaded. Based on the XML file supplied to the node executable, the node will behave as one of the ROS nodes in the model. This allows for a reusable framework where a generic executable (1) loads an XML file, (2) identifies the component instances in the node, (3) finds the necessary component libraries to load and (4) spawns the executor threads bound to each component.

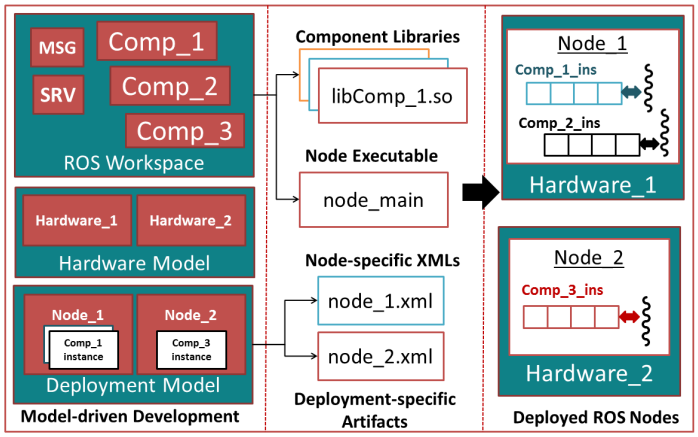

Fig. 5: Software Deployment Workflow

In the above architecture, the deployment needs three primary ingredients: (1) the generic node executable, (2) dynamically loadable component libraries, and (3) an XML file for each ROS node in the deployment model. For each new node added to the deployment model, by merely regenerating the XML files, we can establish a new deployment. The ROS workspace is rebuilt only if new component definitions are added to the Software Model. This architecture not only accelerates the development process but also ensures a separation between the Software Model (i.e. the application structure) and deployment-specific concerns e.g. component instantiation inside ROS nodes.

III. Case Study: Autonomous Ground Support Equipment

This section briefly describes an Autonomous Ground Support Equipment (AGSE) robot that we designed, built, and deployed for the 2014-2015 NASA Student Launch Competition [4]. Special emphasis is given to the value of a rapid system prototyping methodology in the design process and how it allowed the AGSE to overcome many of the challenges and problems encountered during the competition.

A. Competition Requirements

The NASA Student Launch Initiative [4] is a research-based competition partnered with NASA's Centennial Challenges in order to stimulate rapid, low-cost development of rocket propulsion and space exploration systems. Both collegiate and non-academic teams participate in the 8-month competition cycle composed of design, fabrication, and testing of flight vehicles, payloads, and ground support equipment.

The purpose of the 2014-2015 competition was to simulate a Mars Ascent Vehicle (MAV) and to perform a sample recovery from the Martian surface. The requirements for this simulation were twofold: (1) Design and deploy an AGSE robot that autonomously retrieves a sample off the ground and stores it in the payload bay of a rocket, and (2) launch the rocket to an altitude of 3000 ft. before safely recovering the sample.

While the driving requirements of the competition were fixed, many of the minor rules regarding AGSE performance, behavior, and safety requirements evolved and were augmented throughout the course of the eight month design cycle. The volatile nature of these rules precipitated the need for rapidly adjustable design and fabrication processes. For this purpose, the mechanical design of the AGSE followed a modular, quick-to-build approach and ROSMOD was used for software development in order to quickly make on-the-fly adjustments to system behavior.

B. Mechanical Design

The AGSE is a 4-DOF robot utilizing a revolute base joint to rotate the robot body, two prismatic joints to move vertically and horizontally, and a final revolute joint providing an orientation wrist for the end effector to orient a gripper. A wireframe and workspace rendering of the AGSE can be seen in Figure 6.

The AGSE base is comprised of a machined sheet of aluminum, offering a secure mounting point for the upper robotic segments. Above this foundation level, a vertical lead screw, powered by a top-mounted DC motor, drives an aluminum carriage assembly up and down the central rotational axis. A similar lead screw-carriage assembly extends from the side of the vertical carriage to provide motion within the horizontal plane. The combined motion of these joints produces an open-cylindrical workspace. The radially actuating carriage connects to the end effector gripper via a wrist servo motor, which allows the AGSE to interact with the payload and rocket bay.

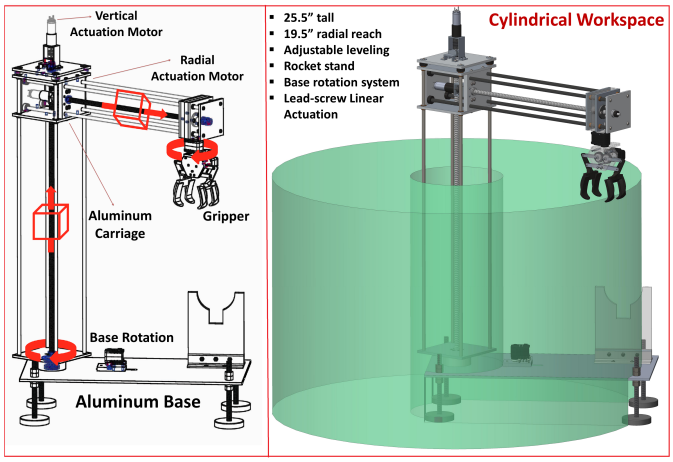

Fig. 6: AGSE Mechanical Design

One significant set of challenges to the construction of the AGSE were time, machinist skill level, and the facilities available to the group's workforce. The team consisted mainly of undergraduate workers with limited machining experience and no access to CNC machinery. Due to this constraint, the mechanical design and software needed to accommodate generous tolerance allowances in component machining. The system also needed to be robust enough to recover from the failure of a component, such as that detailed in Performance Assessment.

The short eight month duration of the design cycle, from initial planning to evaluation, meant that the AGSE system had to undergo rapid development. As such, an iterative, modular, design-build-test approach was implemented in order to concurrently develop as many components of the hardware and software systems as possible. An initial AGSE prototype was conceptualized from off-the-shelf components and the mechanical and software systems were built in parallel, integrated, and tested. These preliminary results were then used in future development to produce a more ideal structure with greater positional accuracy and system robustness. Due to the modular nature of the system's design, it was not necessary to immediately build a completely new second system, so incremental improvements could be made on a specific subsystem (such as the robot's gripper, any single degree of freedom, image processing, motor control, etc.) as the design evolved.

C. Distributed Deployment

The AGSE robot is controlled by a distributed set of embedded controllers. Figure 8 shows the high-level design for the deployment architecture. There are three embedded devices, each with its own responsibilities.

An NVIDIA Jetson TK1 periodically fetches the latest webcam feed, performs image processing and high-level path planning, updating a global state machine. A Beaglebone black (BBB) mounted on top of the robot performs power management, low-level motor control and feedback processing. Lastly, a *User Input Panel* houses a second Beaglebone Black, reacting to user input e.g. pause switch, touchscreen mode changes etc. This last controller is also responsible for keeping the user informed about the real-time state of the AGSE and the current webcam feed. Each of these controllers host ROS multiple nodes with ROSMOD component executor threads periodically performing algorithmic computations, calculating new robotic paths and maintaining the global state machine.

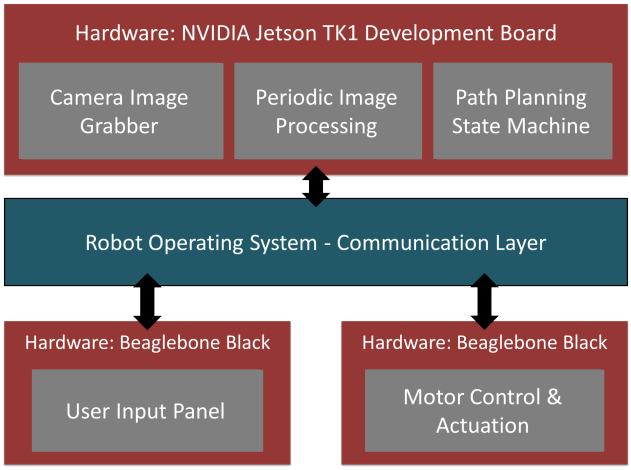

Fig. 8: AGSE Package Deployment

D. Software Prototyping with ROSMOD

The AGSE software [10] was iteratively designed and rapid prototyped using ROSMOD. Figure 7 shows the fully constructed ROSMOD models for the AGSE. The Software Model consists of 8 components spread across three ROS packages - motor control, high-level state machine control and image processing. Each package is characterized by its local set of messages, services and interacting components.

The deployment model shows the various ROS nodes in the final system. Each node instantiates one or more components defined in the Software model e.g. the *positioning* node executes two component threads, one behaving as a vertical_actuator_controller component and the other behaving as a radial_actuator_controller component. Each of these nodes is then deployed on one of the hardware devices modeled in the hardware model.

The ROSMOD code generators enabled generation of nearly 60% (6,000+ lines) of the total built code. For systems like the AGSE with a medium-to-large set of interacting components, a small change to the message structure, port coupling, or functional dependencies can require a cascading and exponentially increased set of code changes. Typically, without ROSMOD, such code-breaking changes can cause a few days of work to fix and the build system can become brittle. This process can be cumbersome and error prone, especially when structural changes to software are frequent.

However, using ROSMOD's code generation and preservation features, such changes can be countered with a few seconds of code regeneration. As developers, we had to fill in the missing pieces - the business logic of the generated callbacks, completing the component interaction loops. This code includes architecture-specific control, e.g. GPIO and encoder readings, LED and switch settings, camera image acquisition, and high-level control. Although the overall software was

Fig. 7: AGSE ROSMOD Model

frequently redesigned and tweaked, a large portion of this code required at most a few weeks of development and testing - a time frame that would not have been met without ROSMOD.

E. Performance Assessment

At the competition, the Vanderbilt AGSE was able to complete the sample retrieval process in approximately 4.5 minutes. The recovery process, as shown in Figure 9, was successful, with payload and rocket bay recognition occurring quickly and efficiently. The AGSE was able to grasp the payload using only two of its four padded end effector phalanges, and successfully deposited the payload within the rocket bay. This operation received high marks from the NASA officials and earned the competition's *Autonomous Ground Support Equipment Award*.

System robustness was validated on the day of competition when a key component failed and was able to be quickly replaced with a different part with no detriment to system performance. The Dynamixel AX-12A servo controlling the base rotational degree of freedom of the AGSE suffered an irreparable failure of its gearbox and had to be removed from the robot. A backup of the servo was not readily available, and a different model servo by the same company had to be swapped in instead. This new model, a Dynamixel MX-28T, while having similar performance as the old servo, had a different communication protocol and mounting footprint, as well as a more complex control scheme.

The component-based nature of ROSMOD allowed quick modifications of the business logic of the *servo_controller* package to update the system to use the new hardware. The new control scheme was quickly implemented and the new physical placement of the servo due to its different mounting footprint was accounted for in software. After these modifications were made, the AGSE was able to perform at its optimal level during its part of the competition.

The rapid prototyping facilitated by ROSMOD and the ROS infrastructure enabled the development of an overall *smarter* robot. The software requirements for autonomy were matched by the ROSMOD code generators such that developers had to spend little time setting up the build system and interaction patterns. The speed of development was drastically improved and the *business logic* code, i.e. the core of the implementation of the system behavior, could be made more robust.

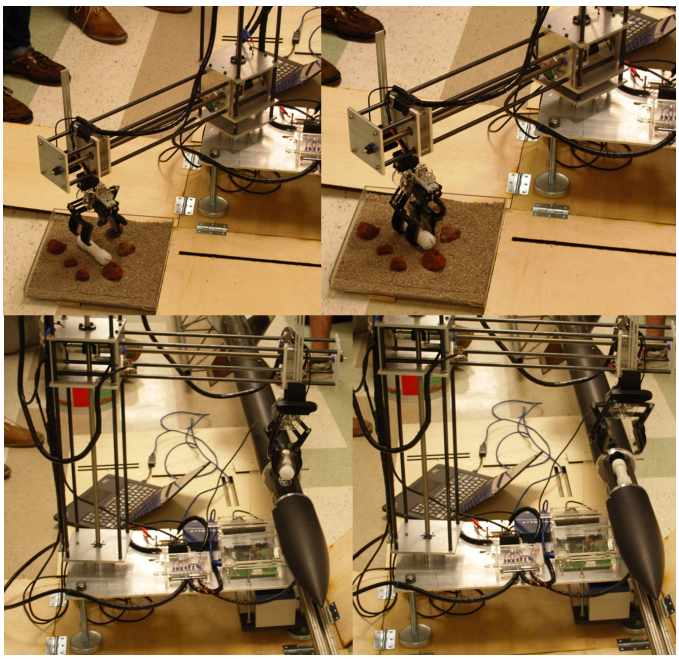

Fig. 9: AGSE Calibration and Testing

IV. Future Work

Building robust, stable and deterministic applications using ROSMOD requires both design-time analysis and runtime management with potential reconfiguration solutions. Our previous work on timing analysis [11][12] and network QoS analysis [13] for component-based distributed real-time systems has given us good insight into the verification and validation requirements, challenges and general design principles for such

systems. Since these methods are model-based and generic, we plan to completely integrate these tools into ROSMOD such that any deployment model can be studied at design-time for potential timing anomalies and network capacity and admittance issues.

To facilitate this, current work on the ROSMOD runtime package aims at introducing deadline monitoring on a per-component basis. The goal of this monitoring thread is to detect deadline violations in all callbacks and operations handled by the component executor thread. This helps in identifying inefficient priority assignments and schedule feasibility issues, which are typical demands for mission-critical real-time systems.

V. Related Efforts

ROS has seen rapid development in the last few years and the number of tools/packages released with ROS grows with every release. In the recent past, several projects and tool suites have established an integration with ROS. We discuss two such infrastructures.

The current version of Matlab's Simulink [14] supports the *Robotics System Toolbox* [15] (RST), using which ROS developers can model ROS workspaces as Simulink blocks, generate executable code, while also exploiting Matlab's large suite of simulation and analysis tools. Unfortunately, the RST was not available to us under our existing academic license. Aside from the costs, our experience shows that it is fairly difficult to modify the Matlab generated code, as well as the modeling language (Simulink), or, for instance, enforce specific scheduling policies, or support the component model constructs that are possible in ROSMOD.

Some infrastructures, like OPRoS [16], provide model transformations and integration tools to couple existing component models and suites with ROS. In case of OPRoS, the framework enables development of OPRoS applications that can communicate with ROS applications. The model transformations help users use the OPRoS platform to develop ROS applications but the expected semantic behavior of the applications, as seen in these models, may not necessarily match the runtime system. Also, such tool suites are relevant to ROS users only if they are also interested in using OPRoS and do not directly focus on ROS development.

VI. Conclusions

Robot Operating System (ROS) is one of few preferred and widely accepted platforms to develop large-scale robotics applications. ROS provides a light-weight middleware framework and a variety of command-line tools to quickly prototype and execute interacting processes. However, much of the development process with ROS can still be automated and there is a need for development tool suites that are easily accessible and that, using model-driven development principles, can provide for a rich set of design-time features such as modeling, code generation, analysis, deployment and process monitoring along with run-time support for general fault management.

The ROSMOD toolsuite presented in this paper supports all these capabilities, as shown in our case study. However, one of the most important aspects of such development tools is the often neglected support for analysis, which is especially relevant in safety-critical real-time systems. Our current work aims at integrating our existing model-driven analysis tools into ROSMOD to support not only a wider range of systems but also a larger user group.

Acknowledgment

This work was supported by DARPA under contract NNA11AB14C and USAF/AFRL under Cooperative Agreement FA8750-13-2-0050, and by the National Science Foundation (CNS-1035655). The activities of the 2014-15 Vanderbilt Aerospace Club were sponsored by the Department of Mechanical Engineering and the Boeing Corporation. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of DARPA, USAF/AFRL, NSF, or the Boeing Corporation. Lastly, the authors would also like to thank the following undergraduate students on the AGSE design team for their invaluable work: Connor Caldwell, Frederick Folz, Alex Goodman, Christopher Lyne, Jacob Moore and Cameron Ridgewell.

References

[1] M. Quigley, K. Conley, B. P. Gerkey, J. Faust, T. Foote, J. Leibs, R. Wheeler, and A. Y. Ng, "Ros: an open-source robot operating system," in *ICRA Workshop on Open Source Software*, 2009.

[2] "DARPA Robotics Challenge," http://www.theroboticschallenge.org//.

[3] "ROSMOD Repository," https://github.com/finger563/rosmod/.

[4] "NASA Student Launch," http://www.nasa.gov/audience/forstudents/studentlaunch/home/.

[5] I. Crnkovic, M. Chaudron, and S. Larsson, "Component-based development process and component lifecycle," in *Software Engineering Advances, International Conference on*. IEEE, 2006, pp. 44–44.

[6] G. T. Heineman and B. T. Councill, *Component-Based Software Engineering: Putting the Pieces Together*. Reading, Massachusetts: Addison-Wesley, 2001.

[7] W. R. Otte, A. DUBEY, S. Pradhan, P. Patil, A. Gokhale, G. Karsai, and J. Willemsen, "F6com: A component model for resource-constrained and dynamic space-based computing environments," in *Object/Component/Service-Oriented Real-Time Distributed Computing (ISORC), 2013 IEEE 16th International Symposium on*. IEEE, 2013, pp. 1–8.

[8] T. Parr, *The Definitive ANTLR Reference: Building Domain-Specific Languages*, 2007.

[9] D. Bell, "Uml basics: An introduction to the unified modeling language," 2003.

[10] "AGSE Repository," https://github.com/finger563/agse2015/.

[11] P. S. Kumar, A. Dubey, and G. Karsai, "Colored petri net-based modeling and formal analysis of component-based applications," 2014, p. 79–88. [Online]. Available: http://ceur-ws.org/Vol-1235/paper-10.pdf

[12] ——, "Integrated analysis of temporal behavior of component-based distributed real-time embedded systems," 2015. [Online]. Available: http://conferences.computer.org/isorc/2015/papers/8781b019.pdf

[13] W. Emfinger, G. Karsai, A. Dubey, and A. Gokhale, "Analysis, verification, and management toolsuite for cyber-physical applications on time-varying networks," in *Proceedings of the 4th ACM SIGBED International Workshop on Design, Modeling, and Evaluation of Cyber-Physical Systems*, ser. CyPhy '14. New York, NY, USA: ACM, 2014, pp. 44–47. [Online]. Available: http://doi.acm.org/10.1145/2593458.2593459

[14] "Simulink," http://www.mathworks.com/products/simulink/.

[15] "Robotics system toolbox," http://www.mathworks.com/help/robotics/index.html.

[16] C. Jang, B. Song, S. Jung, and S. Kim, "A heterogeneous coupling scheme of opros component framework with ros," 2012.