MDPI

Article

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

Pranav Srinivas Kumar [1,†,‡*], William Emfinger [1,‡], Gabor Karsai [1,‡], Dexter Watkins [2,‡], Benjamin Gasser [2,‡], and Amrutur Anilkumar [2,‡]

[1] Institute for Software Integrated Systems, Electrical Engineering and Computer Science, Vanderbilt University; {pkumar, emfinger, gabor}@isis.vanderbilt.edu
[2] Mechanical Engineering, Vanderbilt University; {dexter.a.watkins, benjamin.w.gasser, amrutur.v.anilkumar}@vanderbilt.edu
* Correspondence: pkumar@isis.vanderbilt.edu; Tel.: +1-615-414-1561
† Current address: 1025 16th Ave S, Nashville, TN, 37212
‡ These authors contributed equally to this work.
§ This paper is an extended version of our paper published in IEEE International Symposium on Rapid System Prototyping, 2015.

Academic Editor: name
Version March 31, 2016 submitted to Entropy; Typeset by LaTeX using class file mdpi.cls

Abstract: This paper presents ROSMOD, a model-driven component-based development tool suite for the Robot Operating System (ROS) middleware. ROSMOD is well suited for the design, development and deployment of large-scale distributed applications on embedded devices. We present the various features of ROSMOD including the modeling language, the graphical user interface, code generators, and deployment infrastructure. We demonstrate 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.

Keywords: robotics; distributed; real-time; embedded; cyber-physical; timing; analysis; model-driven; development.

1. Introduction

Distributed Cyber-Physical Systems (CPS) are hard to develop hardware/software for; because the software is coupled with the hardware and the physical system, software testing and deployment may be difficult - a problem only exacerbated by distributing the system. All systems require rigorous testing before final deployment, but may not be able to be tested easily in the lab or may not be testable in the real world without first providing the assurances that testing produces, especially with respect to reliability and fail-safety. Examples of these systems include unmanned aerial vehicle (UAV)/ unmanned underwater vehicle (UUV) systems, fractionated satellite clusters [1], and networks of autonomous vehicles, all of which require strict guarantees about not only the performance of the system, but also the reliability of the system.

Developing systems in a design-test-integrate-iterate methodology is standard practice especially for hardware systems, where electrical and mechanical components can be prototyped and unit-tested before performing higher-level system integration testing. This development approach becomes more challenging when developing software however, since there do not exist well-adopted pervasive standard interfaces and design principles for software.

Emerging industry standards and collaborations are progressing towards component-based system development and reuse, *e.g.* AUTOSAR[2] in the automotive industry. As these systems are becoming increasingly more reliant on collections of software and hardware components which must interact, they enable more advanced features, and performance, but must still meet stringent safety requirements and as a consequence require more thorough integration testing. Comprehensive full systems integration is required for system analysis and deployment, and the development of relevant testing system architectures enables expedited iterative integration. Developing these systems manually, by prototyping individual components and composing them can be expensive, time consuming and error prone, so tools and techniques are needed to expedite this process.

Examples of such systems which can be prototyped and tested using this architecture are (1) autonomous cars, (2) controllers for continuous and discrete manufacturing plants, and (3) UAV swarms. Each of these systems is characterized as a distributed CPS in which embedded controllers are networked to collectively control a system through cooperation. Each subsystem or embedded computer senses and controls a specific aspect of the overall system and cooperates to achieve the overall system goal. For instance, the autonomous car's subsystems of global navigation, pedestrian detection, lane detection, engine control, brake control, and steering control all must communicate and cooperate together to achieve full car autonomy. The control algorithms for each of these subsystems must be tested with their sensors and actuators but also must be tested together with the rest of the overall system. It is these types of cooperating embedded controllers which are a distinguishing feature of distributed CPS.

In the field of Robotics, adoption of Robot Operating System (ROS) [3] has become widely prominent. ROS 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* [4].

This paper describes ROSMOD [5], an open source development tool suite and run-time software platform for rapid prototyping of 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, and present a model-driven development workflow with run-time management tools to build and deploy component-based software with ROS. The utility of ROSMOD is demonstrated 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 [6] 2014-2015. We describe the challenges and requirements, the robotic design, the software prototyping and the overall performance that led us to winning this competition.

1.1. ROSMOD Component Model

Software development using ROSMOD is influenced by the principles of Model Integrated Computing[7] and Component-based Software Engineering [8][9]. Large and complex systems can be built by assembling reusable software pieces called *components*. These components are specified by well defined execution semantics and interaction patterns. With this model of a component's behavior, more complex software can be created as the composition of multiple components and a mapping between the input/output ports of the components. ROSMOD components contain executable code that implement functions, manipulate state variables and interact with other components in applications. This model is inspired by our previous efforts with the F6COM Component Model [10].

Here, ROSMOD is focused on improving the software development workflow for ROS applications. ROS provides a light-weight middleware framework for a variety of interactions between processes via a communications broker. However, the development of ROS applications is neither model-driven nor component-based; ROS processes, called *nodes*, are assembled together

into a *package* and interact using the supported set of ROS *ports*. ROS defines certain types of input and output ports that are available with either blocking or non-blocking semantics: *publish-subscribe* interactions enable non-blocking one-to-many one-way transport of data while *client-server* interactions enable blocking send/receive one-to-one interactions. In these interactions, the publishers and clients act as the output ports and therefore trigger subscriber and server input ports. In these interactions, input ports trigger *operations* which can execute arbitrary code. Such nodes can also be triggered by timers for time-triggered operations e.g. periodic camera feeds. It is important to note that ROS imposes no design constraints or principles on the nodes; they can be any process with any number of threads, shared memory, etc. The goals of ROSMOD are to provide a component model with clear execution semantics coupled with design principles for multi-threaded distributed code, and a model-driven development tool suite for rapid prototyping of ROS applications, while receiving the numerous benefits of component-based software engineering.

ROSMOD components, as shown in Figure 1, can contain a variety of ports and timers. Publisher ports publish messages, without blocking, on a message topic. Subscriber ports subscribe to such topics and receive all messages published on the specified topic. This interaction implements an anonymous topic-driven publish-subscribe message passing scheme [11]. Server ports provide an interface to a component service. Client ports can use this interface to request such services. Clients and servers implement a blocking peer-to-peer synchronous remote method invocation (RMI) interaction [12]. Component timers can be periodic or sporadic and allow components to trigger themselves with the specified timing characteristics. ROSMOD components, upon deployment, are dormant. Execution of the application is first initiated by a timer-triggered component. The executor thread of the triggered component wakes up and executes the *trigger operation*. The application code written in this operation may initiate other triggers and component interactions, thus dictating the behavior of the ROSMOD application.

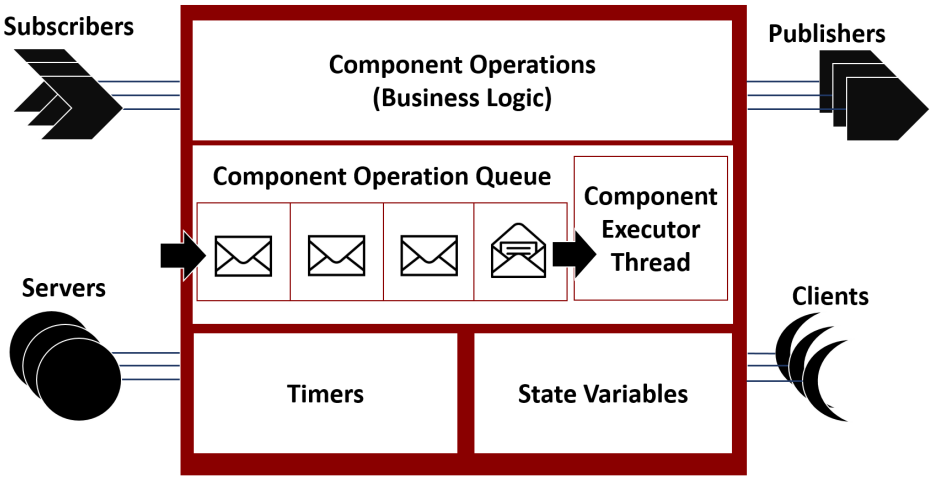

Figure 1. ROSMOD Component

Here, an *operation* is an abstraction for the different tasks undertaken by a component. These tasks are implemented by the component's executor code, written by the developer. Application developers provide the functional, *business-logic* code that implements operations on the state variables. In order to service interactions with the underlying framework and with other components, every component is associated with a *operation queue*. This queue holds instances of operations that are ready for execution and need to be serviced by the component. These operations service either interaction requests (as seen on communication ports) or library-specific requests (from the underlying framework). In each ROSMOD component, the operation queue is processed by a single executor thread. Multiple components may run concurrently but each component's execution is single-threaded. Also, the component operation queue supports several scheduling schemes

including FIFO (first-in first-out), PFIFO (priority first-in first-out) [13] and EDF (earliest deadline first) [14]. Operations in the operation queue are executed using a non-preemptive scheduling scheme. This scheduling scheme means that each operation run by the executor thread is run to completion before the next operation in the queue is processed. The use of this component execution model is to enable composable, single-threaded components which can communicate with each other through the underlying ROS middleware without the developer having to deal with locking shared resources or dealing with other related multi-programming issues that typically introduce difficult to debug errors or other run-time issues. Again, following component-based software design allows these components to encapsulate related functionality into a reusable piece of software that can be shared between projects and systems that require that functionality.

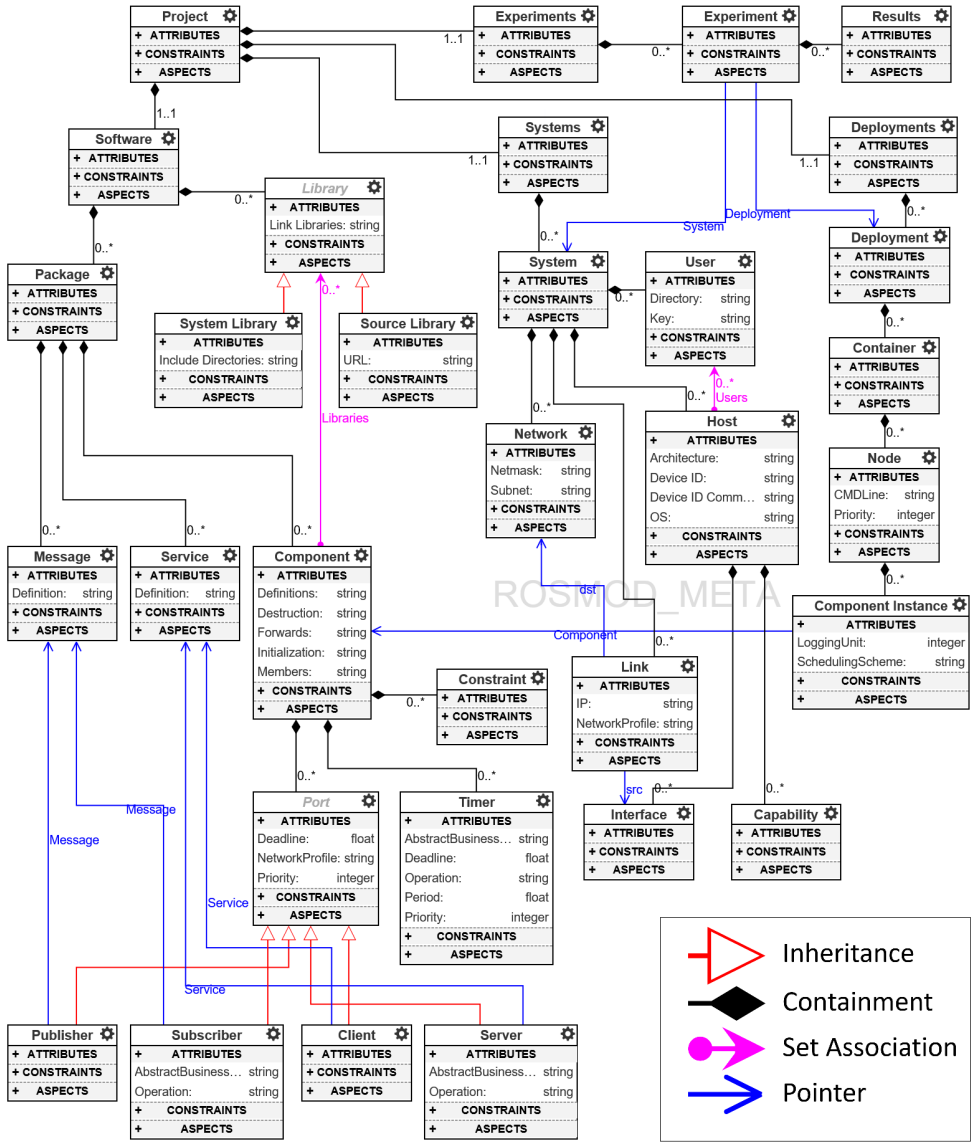

Figure 2. ROSMOD Metamodel. Containment is specified from *src* to *dst* where the source has a containment attribute *quantity*, meaning that *quantity* objects of type *src* can be contained in an object of type *dst*. Pointers are specified as a one to one mapping from source to destination, using the name of the pointer. Sets allow for pointer containment. All objects contain a *name* attribute of type *string*, not shown for clarity. Note: the meta-model is used to create the ROSMOD Modeling Language, but users do not see or interact with it; it is used to enforce proper model creation semantics.

1.2. ROSMOD Modeling Language

To enable the design, development, and testing of software on distributed CPS, we have developed a modeling language specific to the domain of distributed CPS which utilize ROS, the ROSMOD Modeling Language (RML), shown in Figure 2. RML captures all the relevant aspects of the sofware, the system (hardware and network), and the deployment which specifies how the software will be executed on the selected system. Using ROSMOD, developers can create models which contain instances of the objects defined in RML. This approach of using a domain specific modeling language to define the semantics of the models allows us to check and enforce the models for correctness. Furthermore, this approach allows us to develop generic utilities, called *plugins* which can act on any models created using ROSMOD, for instance generating and compiling the software automatically or automatically deploying and managing the software on the defined system. The rest of this section goes into the specific parts of the modeling language, called the Meta-Model, and how they define the entities in a ROSMOD Model.

The top-level entity of RML is a *Project*, which is shown in the upper left of Figure 2. The language supports a variety of modeling concepts that address structural and behavioral aspects for distributed embedded platforms. ROSMOD users can create models of software workspaces, required software libraries, embedded devices, network topologies, component constraints and hardware capabilities. The language also supports code development, particularly with regards to port interface implementations i.e. the execution code for operations owned and triggered by communication ports or local timers. Below, we describe in detail the various aspects of this meta-model and how these concepts are integral to developing distributed CPS and rapid prototyping needs.

1.2.1. Software Model

The *Software* class in Figure 2 models a software workspace. A workspace, following ROS terminology, is a collection of applications that are developed and compiled together into binaries. Thus, each Software class can contain ROS applications, called *Packages*, and *Libraries* required for the applications. Packages consist of *Messages*, *Services* and *Components*. Components contain a set of pointers to Libraries to establish dependence e.g. an *ImageProcessor* component *requires* OpenCV, an open-source computer vision library. Libraries are of two types: Source libraries and System libraries. Source libraries are standalone archives that can be retrieved, extracted and integrated into the software build system with no additional changes. System libraries are assumptions made by a software developer regarding the libraries pre-installed in the system. Here, system refers to the embedded device on which the component is intended to execute.

Messages represent the ROS message description language for describing the data values used by the ROS publish-subscribe interactions. Similarly, *Services* describe the ROS peer-to-peer request-reply interaction pattern. Each service is characterized by a pair of messages, *request* and *response*. A client entity can call a service by sending a request message and awaiting a response. This interaction is presented to the user as a remote procedure call. Each ROSMOD component, as described in the component model (Figure 1), contains a finite set of communication ports. These ports refer to messages and services to concretize the communication interface. Components can also contain *Timers* for time-triggered operation e.g. periodically sampling inertial measurment sensors while operating an unmanned aerial vehicle (UAV). Lastly, components can be characterized by *Constraints*. Currently, ROSMOD constraints are a simple way to establish hardware requirements for operation e.g. fast multi-core processor, quadrature encoded pulse hardware, camera interface etc. When starting such components at runtime, ROSMOD will try to map each component to one of the available devices that satisfies all of the component's constraints. Specifying constraints for a component can be arbitrarily complex but currently we support simple feature-specific constraints e.g. *This component needs a device with atleast 12 open GPIO pins to enable motor control*. In such cases, ROSMOD will simply scan the set of devices in the hardware model and find a candidate host that

provides such a *capability*. More about the deployment infrastructure and the mapping of constraints to capabilities is described in Section 1.4.

All requests received by a component, via subscribers or servers, and all timed triggers can be prioritized. The prioritization is used primarily by the component scheduler and the chosen scheduling scheme. First-in-first-out (FIFO) scheduling of received operations prioritizes based on the arrival time of the requests. Priority-based FIFO resolves conflicts between received operations by using the *Priority* attribute of the relevant ports. Similary, deadline-based schemes like the Earliest-Deadline-First (EDF) scheme uses the *deadline* of the received operation requests to resolve conflicts and operate safely. The scheduling scheme and operation priorities within a component are important choices to make since these choices directly affect the efficiency and safe operation of components. Highly critical operations need acceptable response times for robotic systems to meet quality and timing specifications. This design criteria is our primary motivation for integrating timing and performance characterization techniques into our software specification and tool suite. The integrated performance and timing measurement system allows for the logging and visualization of system execution traces and to show the enqueue, dequeue, and completion of operations being executed by the component executor thread. As these operations trigger each other, these traces provide valuable feedback to the users about the performance and timing characteristics of the system, allowing the determination of performance bottlenecks and resource contention.

1.2.2. System Model

A *System Model* completely describes the hardware architecture of a system onto which the software can be deployed. A ROSMOD Project contains one or more *Systems*. Each System contains one or more *Hosts*, one or more *Users*, one or more *Networks*, and one or more *Links*. A host can contain one or more *Network Interfaces*, which connect through a link to a network. On this link the host's interface is assigned an IP, which matches the subnet and netmask specification of the network. Additionally, a host has a set of references to users, which define the user-name, home directory, and ssh-key location for that user. The host itself has attributes which determine what kind of processor architecture it has, e.g. *armv7l*, what operating system it is running, and lastly a combination of Device ID and Device ID Command which provide an additional means for specifying the type of host (and a way to determine it), for instance specifying the difference between a BeagleBone Black and an NVIDIA Jetson TK1 which both have armv7l architecture but can be separated by looking at the model name in the device tree. Finally, a host may contain zero or more *Capabilities* to which the component constraints (described in the previous section) are mapped. The final relevant attribute is the *Network Profile* attribute of a link. Using the network profile, which is specified as a time-series of bandwdith and latency values, we can configure the links of the network using the Linux TC to enforce time-varying bandwidth and latency. This network configuration is useful when running experiments on labratory hardware for which the network is not representative of the deployed system's network.

1.2.3. Deployment and Experimentation

Deployment refers to the act of starting application processes on candidate hosts, where each host can provide for and satisfy all of the constraints of the processes e.g. general purpose input/output (GPIO) ports, CPU speed etc. Therefore, a deployment is a mapping between application processes and system hosts on which the application processes run. The ROSMOD Deployment Model makes this map a loose coupling to enable rapid testing and experimentation. Each Deployment consists of a set of *Containers*. Each container, conceptually, is a set of processes that need to be collocated. Containers also ensure that no process outside a container is deployed along with the container once it has been mapped to a host. Each container, therefore, contains a set of *Nodes* (ROS terminology for processes). In each node, application developers instantiate one or more components previously defined in the Software model. Following the ROSMOD component model, each such instance maps

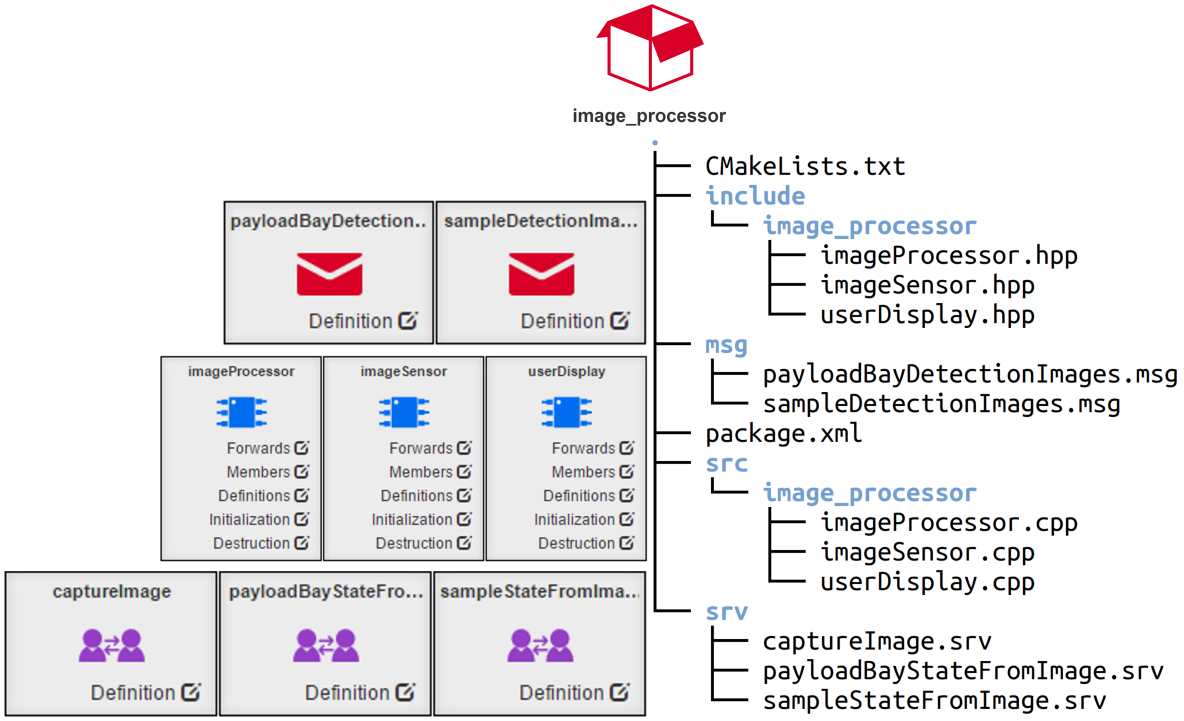

Figure 3. Workspace Code Generation. In this figure, the *image_processor* package and its children messages (red), components (blue), and services (purple) are generated into a catkin package for compilation. The msg and srv files are automatically filled out from the definitions of the messages and services, as are the header and source files for the components.

to an executor thread that executes the operations in the component's operation queue. Note that the same component can be instantiated multiple times even within the same node. Also note the container is not mapped to a specific host within the deployment model, but rather is automatically mapped to a host by the deployment infrastructure within an *Experiment*.

As shown in Figure 2, each project supports a set of *Experiments*. Each experiment has pointers to one System model and one Deployment model. The system model provides the set of available devices and the deployment model provides the set of containers, where each container contains a set of component constraints that need satisfying (the union of all the container's nodes' component constraints). ROSMOD uses these sets to find a suitable mapping and then deploys the containers on the chosen host devices, if a mapping can be found which satisfies all the constraints of all the containers. When the deployment infrastructure selects hosts from the system model for mapping, it first checks to see which of the system model's hosts are 1) reachable, 2) have valid login credentials, 3) have the correct architecture, operating system, and device ID, and 3) are not currently running any other compilation or experiment processes from any other user. In the case that there are no available hosts in the system or the deployment's constraints cannot be satisfied, the infrastructure informs the user. Upon successful deployment of the experiment, the infrastructure automatically stores the specific mapping relevant to the deployed experiment for later management and destruction. When such experiments are stopped, ROSMOD retrieves the component logs from the hosts and displays results. We are currently working on improving our runtime monitoring features to enable real-time component execution plots and network performance measurements at runtime.

Such a loose coupling between the deployment model and the system model, along with the infrastructure automatically mapping the containers to valid, unused hosts enables the execution of the same deployment onto subsets of a large system, for instance running many instances of the same deployment as separate experiments in parallel on a large cluster of embedded devices. Additionally

such a loose coupling enables redeployment onto a completely different system by simply changing the experiment's system model reference; the infrastructure will automatically verify that the new system meets the constraints of the deployment and has available hosts.

1.3. Software Infrastructure

The ROSMOD software infrastructure provides the user with the tools to generate source code corresponding to the software model, compile that source code to produce binaries that can run on the hosts defined in the system models, and also generate documentation for the software and associated libraries.

1.3.1. ROSMOD Workspace

The Software generator in ROSMOD is a plugin that produces a complete ROSMOD workspace. The application software, as defined in the model, is a collection of packages, each with messages, services, components and required libraries. When invoking the workspace code generation, ROSMOD first generates the ROSMOD packages including C++ classes for each component, package-specific messages and services, and logging and XML parser-specific files. Then, for each external library, the URL of the library archive is used to fetch the library and inject this code as a package into the generated workspace. Assuming the library is stable and the target devices have the necessary system libraries, this generated code compiles without errors out of the box. The meta-model for the software is fully specified to include attributes for user code, the business logic for the components' operations, additional class members, etc. Because these pieces of user code are all captured within the model, they are placed into the proper sections of the generated code. For this reason, it is not necessary to alter the generated code before compilation into binaries. If the user chooses, the software infrastructure for ROSMOD will automatically determine the different types of hosts in all the current project's system models, determine which of those hosts are available that match the model specifications and will compile the source code into binaries for those architectures. The compilation is performed as a remote procedure on the host; therefore the compilation will not select hosts which are currently running other compilation or deployment processes. To enable the user to multi-task, compilation is an asynchronous non-blocking process allowing further interaction with the model or other plugins.

Figure 3 shows the generated code tree for an *image_processor* package. This package has three components - *userDisplay*, *imageProcessor*, *imageSensor*, and associated messages and services. For each component, ROSMOD generates a unique class that inherits from a base *Component* class and contains member objects for every component port and timer. Unlike earlier versions of ROSMOD, this generated code is not a skeleton. Rather, it includes all of the operation execution code embedded in the model. So, all changes to this business logic code are done from the model, instead of modifying a skeleton. This ensures code consistency and avoids synchronization issues between the model and the generated code. Furthermore, this enables faster training and use of the ROSMOD tool suite, since users no longer need to know into what folders and files the generated code goes. In the competition version of ROSMOD, a user would have to generate the skeleton code for the workspace, open up the relevant files manually, find the specific places into which to place the business logic for the operations and added members, add extra library code into the generated build files manually, and then inform the infrastructure that the code was ready for compilation. This process was error prone and required detailed knowledge of what files are generated, how to integrate libraries into the build system, and other details that were specific to both ROS' package system and our component implmentation. In the current version of ROSMOD, the user is given direct access to only the code relevant for the specific object (component, message, service) they are editing. Furthermore, the library system now enables the creation and use of libraries without knowledge of how the underlying ROSMOD build system works.

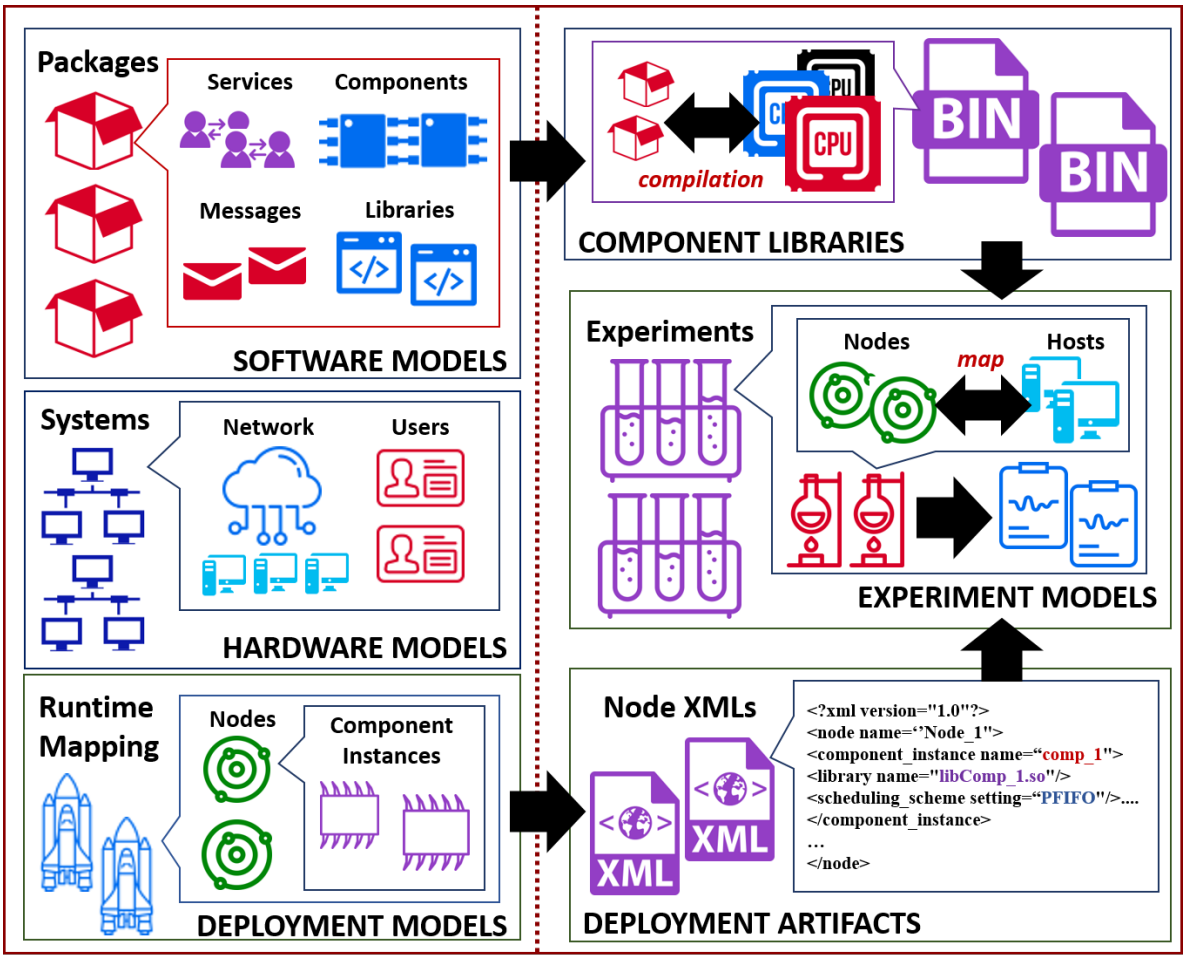

Figure 4. Software Deployment Workflow

1.3.2. Class Documentation

Along with the source code, ROSMOD also supports generation of Doxygen-style documentation for all classes and files used in the software model, including external libraries. All component files, and library directories are scanned and the class hierarchy is constructed from which documentation, both *html* and *latex*, is automatically generated. The generation of documentation allows for better project maintenance, description, and training when multiple people are working on the same project.

1.4. Deployment Infrastructure

The workflow for software deployment is as shown Figure 4. After the user has generated and compiled the software model into binary executables, they can run an experiment that has valid deployment model and system model references. Every ROSMOD workspace is generated with an additional *node* package. This builds a generic node executable that can dynamically load libraries. When the software infrastructure generates and compiles the source code for the software model, the components are compiled into dynamically loadable libraries, one for each component definition along with a single executable corresponding to the generic node package. The first step the deployment infrastructure performs when running an experiment is generating the XML files which contain metadata about each ROS node modeled in the ROSMOD Deployment Model. This metadata 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 deployment 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.

The reason for having the components be libraries that are loaded at run-time by a node executable is 1) to help enforce separation between each component's code, 2) to shorten the compilation time (which may be quite long for embedded processors and complex models), and most importantly 3) to enable the ability for users to change the mapping of component instances to processes without the need to recompile any of the code. Because ROSMOD is focused on the rapid development and deployment of reusable software components, we designed the infrastructure to allow users to experiment with which components are collocated within processes dynamically and rapidly iterate through several scenarios and experiments without having to wait for code compilations between experiments. Finally, such a component-based deployment framework enables unit testing and integration testing in a very well-defined manner for the software components. If an error occurs during the execution of an application on a system, the user can easily and quickly break the deployment down into sub-deployments for unit-testing of only the relevant components to determine the source of the error.

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.

When the user has selected an experiment to run, the deployment infrastructure first determines whether the selected deployment can execute on the selected system. Like the software infrastructure described above, the deployment infrastructure queries the selected system to validate that the system is reachable, conforms to the model, and has available hosts for deployment (i.e. they are not currently running any compilation or deployment processes). Once those available hosts have been determined, the infrastructure attempts to map the deployment's containers to the available hosts based on the constraints and capabilities of the two sets. If the constraints cannot be satisfied by the capabilities of the available hosts, the user is informed and the deployment of the experiment is halted. If the capabilities of the hosts do satisfy the constraints of the containers and their associated components, the deployment infrastructure generates the required XML files for the deployment and copies the XML files and the binaries over to the selected hosts, before starting the relevant processes. Finally, if all of those steps are successful, the infrastructure saves the mapping into the model for user inspection and for later teardown of the experiment.

When the user chooses to end a currently running experiment, the deployment infrastructure verifies that the experiment is still running before stopping the associated processes, copying the relevant log files back, cleaning up the deployment artifacts from the hosts, and removing the saved experiment mapping from the model.

2. Experimental Section

2.1. Case Study: Autonomous Ground Support Equipment (AGSE) Robot

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 [6]. 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. We use this example to demonstrate the utility of our integrated model-driven component-based software tool suite.

2.2. Competition Requirements

The NASA Student Launch Initiative [6] is a research-based competition partnered with NASA's Centennial Challenges, and aims 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 a system termed the Autonomous Ground Support Equipment (AGSE) that independently 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.

2.3. AGSE Project Architecture

The sample retrieval was accomplished using a robotic arm with computer vision to find the sample and identify its orientation. After successfully acquiring the sample, the system will then search for the payload bay, identify its orientation, and place the sample within it. The robot arm itself is a simple crane-style device akin to a pick-and-place robot with a four-pronged gripper as the end effector. It was designed to have a cylindrical workspace in order to most efficiently access the ground around the system and rocket. It starts in a known position and incrementally scans its workspace using a built in camera. Image processing is performed to identify key environmental features such as the sample and payload bay. The control flow in the AGSE software is shown in Figure 5.

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 competition. The volatile nature of these rules combined with the short eight month duration of the build cycle precipitated the need for rapidly adjustable design and fabrication processes. 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.

2.3.1. Distributed Deployment

The AGSE robot is controlled by a distributed set of embedded controllers. Figure 6 shows the high-level design for the deployment architecture. There are three embedded devices, each with its own responsibilities. The reasons for the use of multiple embedded controllers cooperating was two-fold: 1) given the design decision to fully automate the robot to search the workspace for both the sample and the payload bay, we needed an embedded processor capable of performing image-based object detection and 2) given the design of the robot to search a workspace with the given degrees of freedom, we needed an embedded processor with the required available General Purpose Input/Output (GPIO) and Special Function Input/Output (SFIO) pins.

To meet the first requirement, we selected the NVIDIA Jetson TK1, which is an embedded ARM controller with 4+1 ARM cores and 192 CUDA cores, that consumes 10 W or less. We could not use the Jetson to meet the second requirement since it does not support enough GPIO to control the linear actuators and retrieve feedback from them. Furthermore, it lacks SFIO for encoder pulse

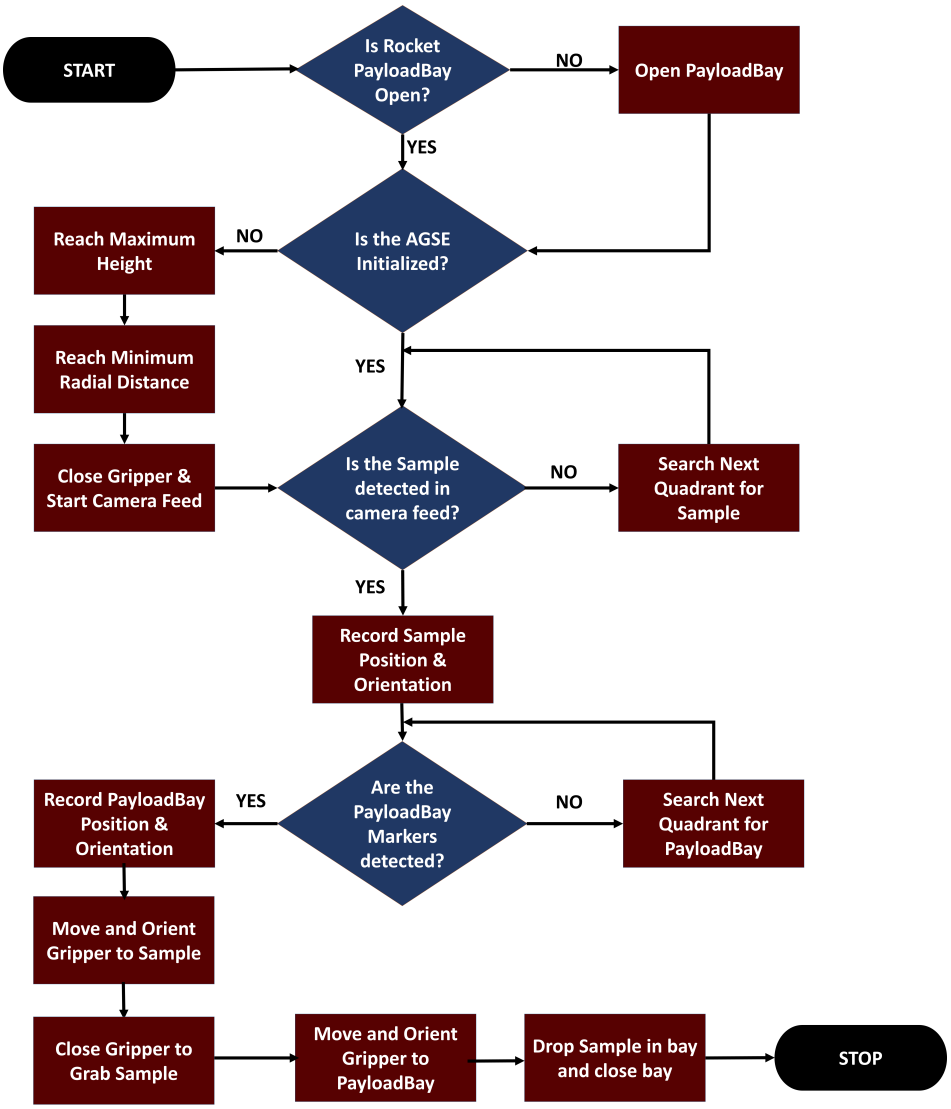

Figure 5. AGSE Control Flow Chart

decoding. Therefore, a BeagleBone Black was selected for the motor control interface board because it has specific hardware for decoding quadrature encoded pulses (QEP) and enough available GPIO for controlling the linear actuators and reading limit switches.

Since one of the secondary requirements of the competition governed pause control and state feedback to the operator of the AGSE (during the competition execution), a second BeagleBone Black was introduced which served to provide mechanical safety switches for pausing the AGSE, LED panels indicating the state of the AGSE, and a touchscreen showing what the AGSE sees as it searches the workspace for the sample and the payload bay. This BeagleBone Black resides the User Interface Panel (UIP).

The NVIDIA Jetson TK1 periodically fetches the latest webcam feed, performs image processing and high-level path planning, and updates a global state machine. The Beaglebone Black (BBB) mounted on top of the robot performs power management, low-level motor control and feedback processing. Lastly, the User Input Panel (UIP) houses a second Beaglebone Black which reacts to user input through switches and provides feedback through touchscreen display and LED panel display. The UIP 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 multiple ROS nodes with ROSMOD component

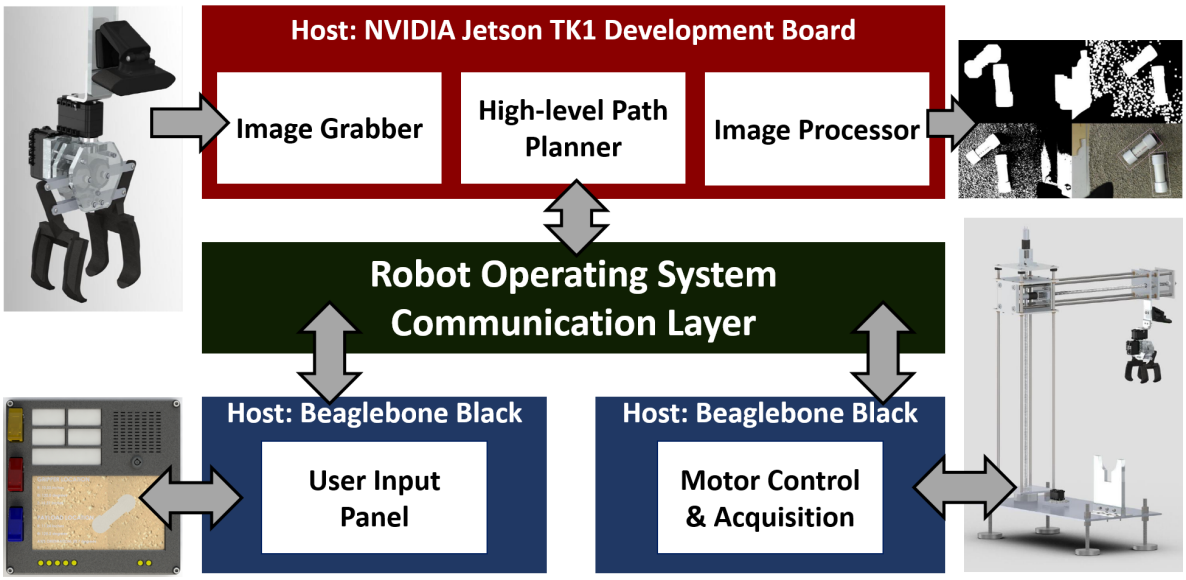

Figure 6. AGSE Package Deployment

executor threads periodically performing algorithmic computations, calculating new robotic paths and communicating to coordinate and maintain the AGSE state.

2.3.2. Software Prototyping with ROSMOD

The AGSE software was iteratively designed and rapid prototyped using our ROSMOD tool suite. 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, service and interacting components. Note that just as in ROS, packages can share messages so that components can subscribe/publish/provide/require messages/services from other packages. Figure 7 shows the component assembly and wiring as per the design. The *radialControl* and *verticalControl* components are responsible for radial and vertical actuation of the AGSE respectively. The *rotationControl* component is capable of controlling three servo motors: (1) the base rotation servo, (2) the gripper rotation servo, and finally the (3) gripper position servo that opens and closes the robot gripper. A *Camera* component deployed on the NVIDIA Jetson TK1 provides a direct interface to the camera. Using the *CaptureImage_Server*, the *ImageProcessor* component receives a snapshot of the camera feed for periodic processing needs. This feed is also used by the *userDisplay* component to display the feed on the user input panel, as shown in Figure 6. The user input panel is the primary interface between the ROSMOD applications and the user. The *userDisplay* component provides information to the user and the *userInputControl* receives data from the user, specifically to read the various control switches on the panel e.g. the pause, alarm, and debug switches. Lastly, a *HighLevelControl* component orchestrates the high-level state transitions and controls the operation of the robotic arm. These transitions include commands such as *find_payload_bay*, *find_sample*, *move_to_target* etc., each of which publishes messages to other components to propagate the motor control commands.

The ROSMOD code generators enabled generation of nearly 60% (6,000+ lines) of the total built code. As mentioned before, much of this code includes port initialization, build system files, callback skeletons, etc. that usually take up a significant amount of development time. As developers, we had to fill in the missing pieces - the business logic of the callbacks, completing the component interaction

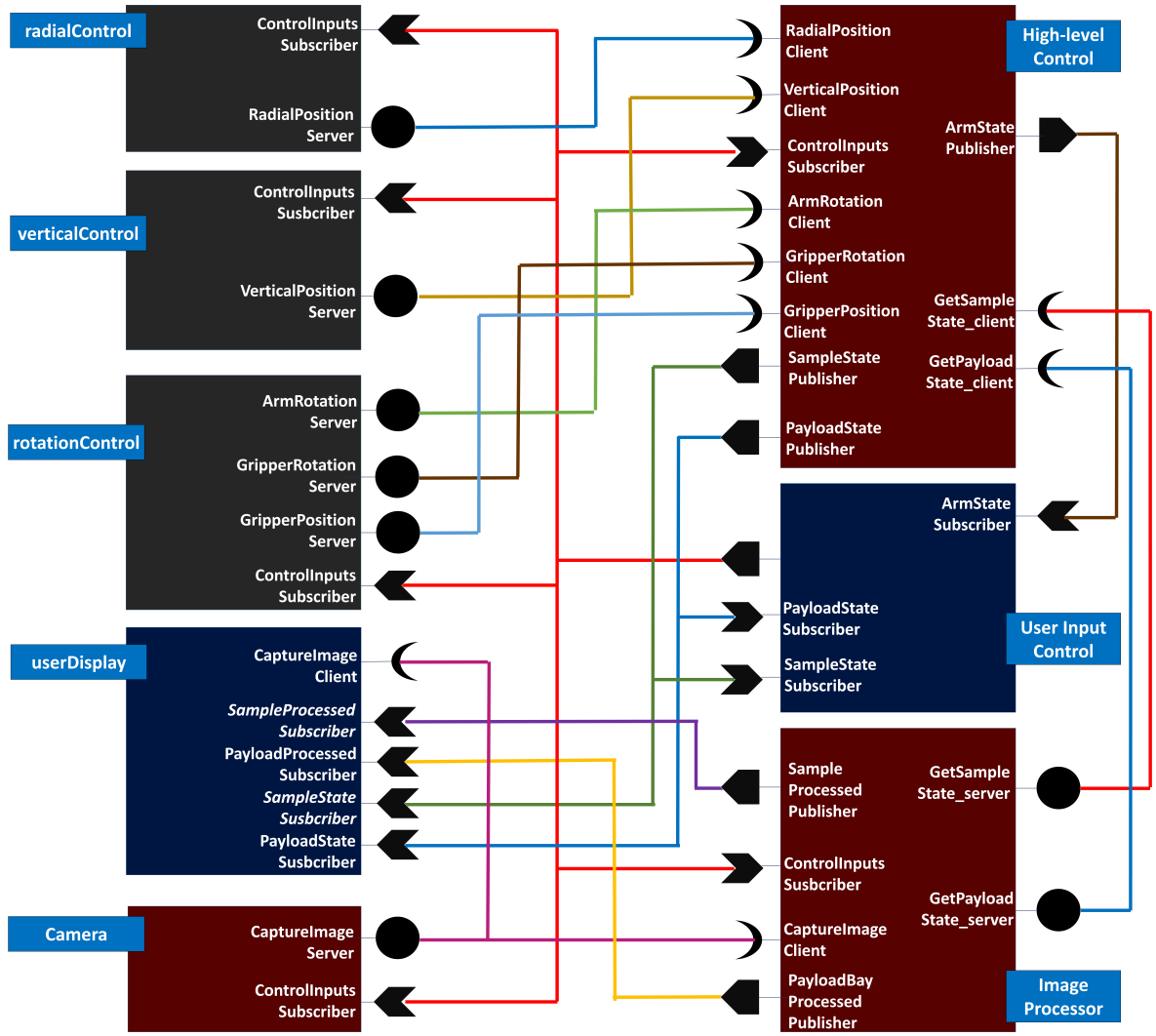

Figure 7. AGSE Component Assembly

loops. This code includes architecture-specific control, e.g. GPIO and encoder readings, LED and switch settings, camera image acquisition, and high-level control.

The final AGSE used in the 2014-2015 NASA SLI competition is shown in Figure 8.

3. Results

3.1. 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

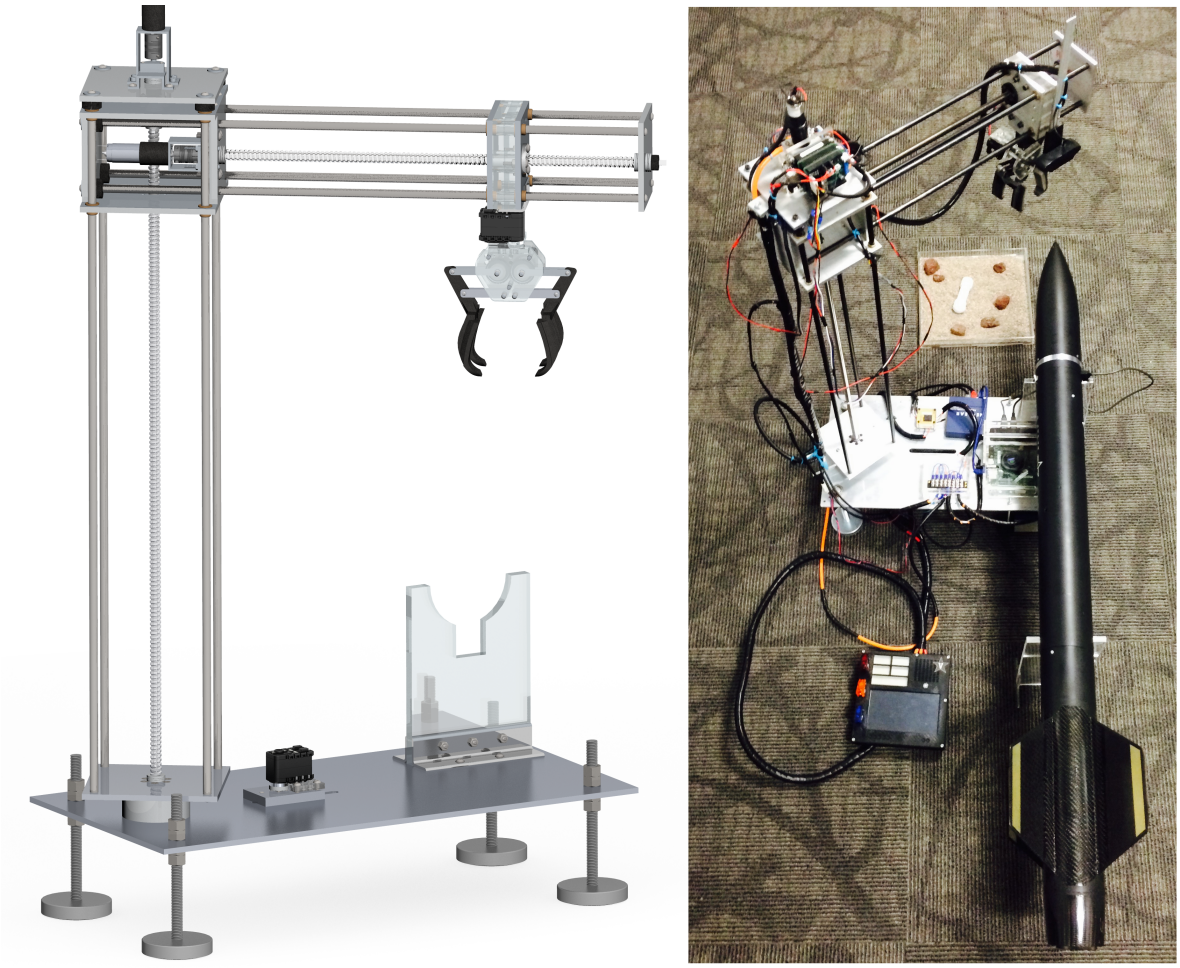

Figure 8. AGSE and rocket used in the 2014-2015 NASA SLI competition. The UIP is shown in the bottom left of the picture, the Motor Control Board is on the top of the arm of the AGSE, and the NVIDIA Jetson is under the rocket.

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 *rotation_controller* component to update the system to use the new hardware. The new control scheme was quickly implemented and the control software was updated to account for the new physical placement of the servo due to its different mounting footprint. After these modifications were made, the AGSE was able to perform at its optimal level during its part of the competition.

4. Discussion

The goal of ROSMOD is to be a model-driven, component-based rapid development, deployment, and experimentation toool suite for distributed CPS. This goal has been the driving force since the beginning of ROSMOD during the 2014-2015 NASA SLI competition with the AGSE. ROSMOD itself was developed alongside the AGSE and in concert with it; as we added features to the modeling language, the user interface, or the generation and deployment infrastructure, we immediately tested them with the AGSE. The focus on the development was always model-driven engineering coupled with component-based software design principles enabling an iterative

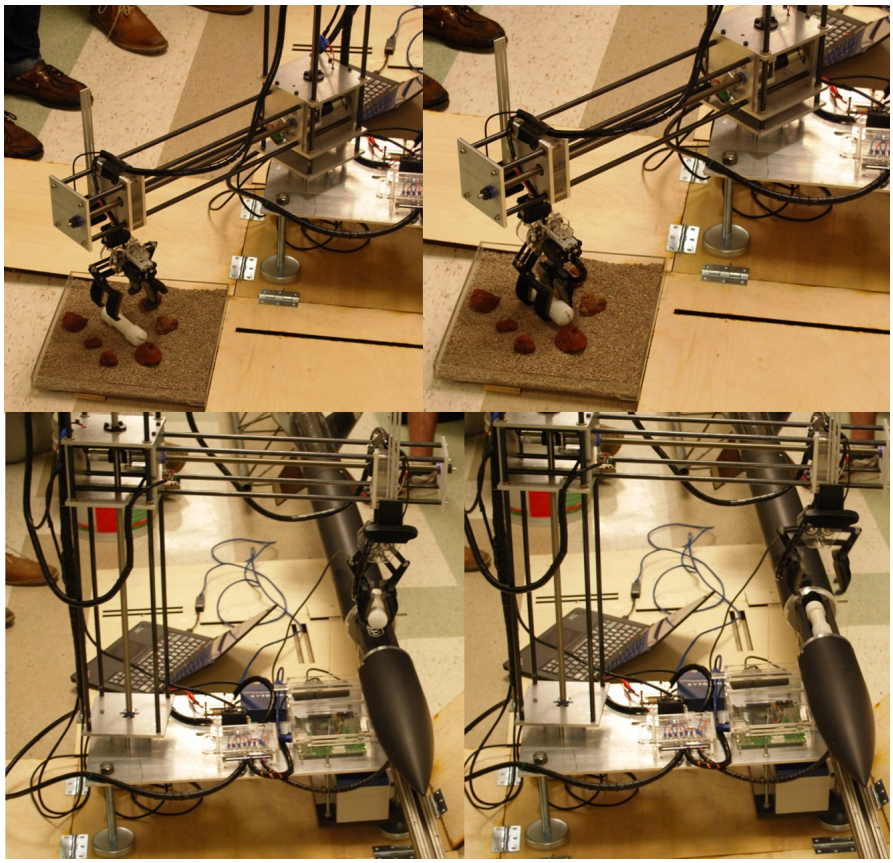

Figure 9. AGSE Calibration and Testing

development cycle for both the AGSE and ROSMOD. Because of the competition's deadlines, we focused on developing the AGSE and ROSMOD through short cycles between design, implement, test, iterate. These deadlines helped focus the core of ROSMOD's architecture towards one of rapid system design, development, and deployment.

Through this development cycle, our team had to maintain a balance between many (sometimes conflicting) design considerations:

- utilizing component-based design to parallelize the development effort into subunits among each member
- evolution of the design in concert with evolution of the implementation: must prototype and test quickly, but the implementation should be usable for the next phase of the design.
- track and plan for materials lead-time, and testing/integration time
- tool support, i.e. mechanical facilities and software development/testing support
- expertise required for each component/subsystem design, development, and testing
- importance of meaningful feedback especially with respect to errors and failures in software/hardware.

As in any software or hardware development, bugs and failures were encountered in the development of both the AGSE hardware and software. However, the use of the ROSMOD infrastructure allowed the causes of those issues to be narrowed down and more quickly resolved. For example, the logging and tracing framework of ROSMOD enabled the resolution of issues stemming from slow processor speed leading to long exeuction times. By using the logging and plotting features of ROSMOD to trace the execution time of the component operations in the AGSE software, we were able to determine that the NVIDIA Jetson was running sub-optimally, and confirmed that the CPU frequency governor was configured by default to dramatically throttle the main ARM cores

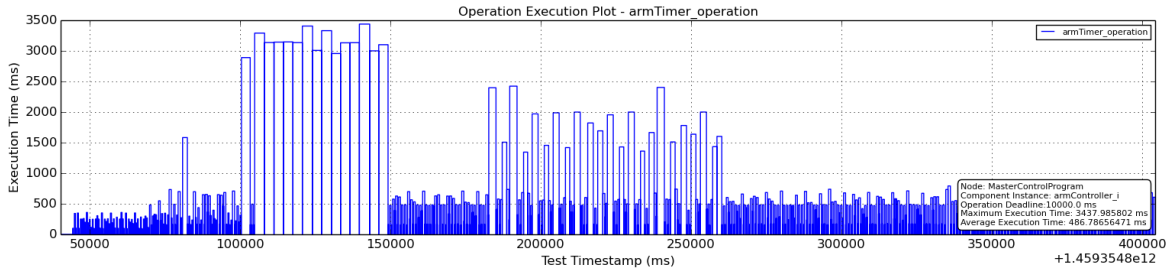

Figure 10. High-level Timer Execution

of the processor. Furthermore, we were able to use the logging framework to show that for some of the client-server connections in the AGSE that were configured to be persistent connections, the connection would drop immediately after being established. After determining this from the trace logs, we reconfigured those client-server connections to no longer be persistent so they would reconnect when required.

One of the main benefits from this timing and performance logging infrastructure was the ability to validate the periodicity of timer interactions in the AGSE. By running the AGSE software and tracking how long each operation takes, we were able to determine that the original configuration of the AGSE timers was too frequent and needed to be reduced by a factor of four. By specifying deadlines on the all the operations (esp. timer, server, and subscriber) we were able to determine the bottlenecks in the system and figure out what parts of the system had deadline violations (i.e. their execution times exceeded their periodicity) and re-adjust the timers accordingly.

Figure 10 shows the execution time plot of the *armTimer_operation* in the high-level controller component. For sake of brevity, we avoid showing all the operational plots. In the context of Figure 7, this timer operation represents the execution of the high-level control state machine. The high-level controller communicates with multiple components over its life-span orchestrating various parts of the overall state machine e.g. initialization, sample detection, payload bay detection etc. Choosing an optimal period for this timer is necessary for many reasons. Firstly, the ROSMOD component operation scheduling is a non-preemptive one i.e. an operation must run to completion before the next request is serviced from the component operation queue. This means that if the timer operation does not complete before its period, the number of waiting requests in component queue start to monotonically rise. This hurts the system as a whole since subsequent requests take longer to complete and the high-level control breaks down. Secondly, inside the timer operation, the controller requires the services of both the motor control and the imaging components, often times via synchronous blocking interactions. This is the second cause of a potential problem as the blocking times are non-deterministic and dependent on the execution state of other embedded boards. For this reason, the AGSE software was deployed and experimented with to identify the optimal period where the response times are manageable. As shown in Figure 10, the high-level controller has two set of spikes in execution time. The first set (between 100 and 150 seconds into the experiment), as confirmed by Figure 11 corresponds to sample detection. The second set of spikes, between 200 and 250 seconds, is the payload bay detection. These plots quickly present the performance of the high-level controller and the worst-case response times during periodic image processing.

Figure 11 presents the performance of the *GetSampleState_Server* and the *GetPayloadState_Server* in Figure 7. These servers are periodically invoked by the high-level controller during its operation, first during sample detection and then during payload bay detection. Each of these servers, on demand, are required to obtain the current camera feed from the Camera component, perform image processing, and return the results to the high-level controller. Thus, the interaction and data flow path is as follows: During sample detection, the high-level controller is periodically triggered to ask the Image Processor component to provide a new result. The Image Processor, in turn, queries the

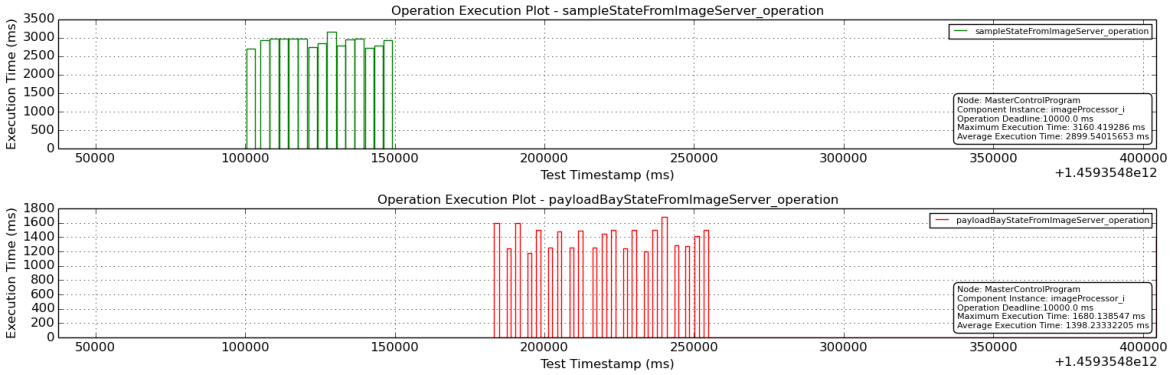

Figure 11. Sample and Payload bay Detectors Execution

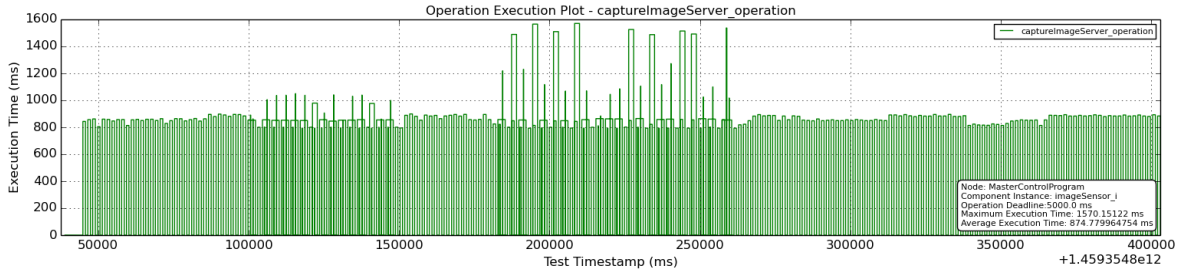

Figure 12. Camera Feed Reponse Times

Camera component for a feed update. The Camera *ImageServer*, as shown in Figure 12 responds to the Image Processor component with a new feed. This image server is also responding to the userDisplay component that receives the camera feed to display to the user.

Wwe were able to use the ROSMOD deployment infrastructure to quickly run separate experiments on the BeagleBone Blacks (single core embedded computers), testing the performance impact of running our motor control components in separate processes or as separate threads in the same process. By simply changing the deployment model and re-running the same code we were able to get ROSMOD timing and performance logs from the experiments to show the extra overhead of process-level context switching on the single-core BeagleBone Black.

The use of ROSMOD's performance, timing, and trace logging (coupled with the plotting utilities for those logs) enabled us to easily visually verify the behavior and performance of the AGSE software or spot anomalies when they occurred. Furthermore, the use of code-generation and automated build and deployment infrastructure meant that far less code had to be inspected for errors when a software or logical error cropped up, and the developers did not have to spend time configuring or debugging the build and deployment systems. Finally, the use of a graphical modeling tool for specifying the entire AGSE project from software to hardware to deployment enabled faster training and communications between team members as well as visual inspection of the software configuration, for instance ensuring that all required components can respond to the user's control inputs or verifying the other interaction patterns and triggering operations for each component.

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 in spite of the evolution of and inclusion of more autonomy.

5. Materials and Methods

This section covers the materials and methods for ROSMOD and the AGSE, both what was used in the competition (2014-2015) as well as the current state of each.

5.1. Competition AGSE

The version of the AGSE software and hardware designs that were used in the 2014-2015 NASA SLI competition can be found open-sourced online[15]. The current version of the AGSE code and hardware designs has been moved and can be found in the Vanderbilt Aerospace Design Lab's AGSE repository[16].

5.1.1. Kinematics

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 orienting wrist for the end effector. A wireframe and workspace rendering of the AGSE can be seen in Figure 13.

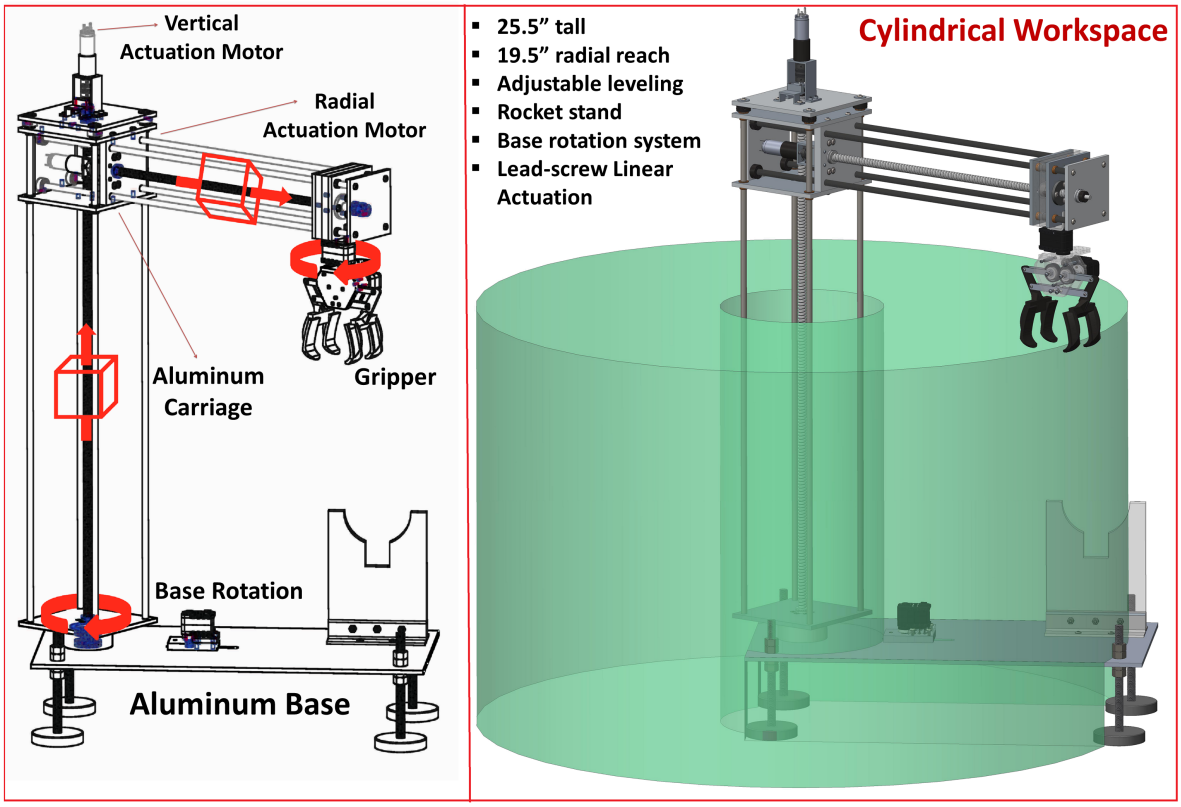

Figure 13. AGSE Mechanical Design

By design, the single revolute and two prismatic joints of the AGSE provide the basis for a cylindrical coordinate system and workspace, reducing the implementation of both forward and reverse kinematics to a trivial exercise of mapping joint position to the corresponding coordinate in the workspace.

$$\begin{bmatrix} \boldsymbol{\theta}_{workspace} \\ \mathbf{r}_{workspace} \\ \mathbf{z}_{workspace} \end{bmatrix} = \begin{bmatrix} \boldsymbol{\theta}_{rev} \\ \mathbf{r}_{pris} \\ \mathbf{z}_{pris} \end{bmatrix}$$

5.1.2. Structure

The AGSE base is comprised of a machined sheet of aluminum upon which leveling legs are mounted to easily account for uneven surfaces. Protruding from the base is the revolute joint, a machined spindle centered within ball bearing cup. Extending upwards along this rotational axis is the vertial join, a lead screw and guide rod assembly driving an aluminum carriage. A similar lead screw-carriage assembly extends from the side of the vertical carriage to provide motion within the horizontal plane. A mounting point for both the gripper and camera hangs from this horizontal carriage.

Underneath the base is a platform for mounting batteries, power regulation and protection circuitry, and on-board computer systems. At the top of the vertical assembly is a mounting point for the actuator control electronics.

5.1.3. Actuation and Sensing

A Dynamixel MX-28T servo is mounted to the revolute joint's spindle to provide rotational movement. Two additional Dynamixel AX-12 servos are used to orient and actuate the gripper. The two lead screw assemblies are driven using 12 V Faulhaber motors mounted directly to the lead screws.

The AGSE guides the end effector motion using the limit switch-encoder combo on its linear actuators, the built in positional feedback from the servo motors, and the optical feedback provided by a camera mounted above the end effector (allowing the AGSE to recognize objects within the reachable area underneath the gripper phalanges). Limit switches are mounted at the minimum and maximum of each lead screw assembly's range of movement, providing hard stops to motor actuation. Quadrature encoders are then used within this range of motion to monitor the position of the vertical and horizontal carriages. The Dynamixel servos controlling the revolute joint and gripper provide position and speed feedback, as well as PID tuning to allow for quick and responsive control.

5.1.4. Power System

The power from the AGSE comes from two 12V 12Ah lead-acid batteries in series, which provide input power to a 12V power regulator. The 12V output from the power regulator passes through a relay which is controlled using a key-switch on the User Interface Panel (UIP). From the relay the 12V runs in parallel to the NVIDIA Jetson TK1, the Motor Control Cape on the Motor Control BeagleBoneBlack, the User Interface Panel power control cape (a stripped down version of the Motor Control Cape with the H-Bridges and other motor related electronics not populated), and finally to the ethernet switch which provides a closed, hard-wired LAN for communications between the boards.

On the Motor Control Board, the 12V input drives 1) the cape's 5V regulator which provides power to the BeagleBone Black and the encoders, 2) the 12V Dynamixel servo motors, and 3) the H-Bridges for the two 12V linear actuators.

On the UIP, the 12V simply powers the 5V regulator which provides power to the BeagleBone Black, the touchscreen LCD, and the display LEDs.

5.1.5. Control

The payload bay and nose section of the rocket rests on the base of the AGSE. This payload bay is motor-controlled by the high-level controller. When the AGSE starts up, the payload bay's on-board Arduino motor controller is commanded to open the nosecone. Once open, the AGSE performs an initialization routine. This routine includes movement of the vertical and radial actuators to a safe height and a retracted state. Then, the AGSE rotates on its base to a servo angle of zero degrees. Once this routine is complete, the robots begins searching for the sample. The camera used for this purpose is a 1920x1080 wide resolution feed; one that enables the robot to sweep wider distances

when searching for the sample and increases the step size of the search i.e. the search steps up by 20 degree increments while searching for the sample.

Figure 14. AGSE Sample Detection - Periodic Image Processing

Periodic sample detection performed by the AGSE uses OpenCV-based image processing algorithms to identify and track the sample in real-time. An Image Processor component periodically fetches the latest feed from the mounted camera and performs a series of filtering tasks. Each *RGB* (Red-green-blue) image frame is converted to both *HSV* (Hue-Saturation-Value) and *Grayscale* image frames. After applying thresholds, filters, erosion and dilation methods, the target sample is extracted from the webcam feed. Once the target sample is detected, as shown in Figure 14, we draw contours around the object and identify its relative position and orientation.

Once the sample is detected, the AGSE records the position and orientation of this sample and begins searching for the payload bay. The payload bay is detected using marker detection methods. Coded markers are stuck on the payload bay to detect the angle at which the sample needs to be dropped into the bay. As shown in Figure 15, by processing the input camera feed, the two markers on the bay are detected and the angle of drop is calculated.

Once the sample and payload bay are fully detected, the AGSE performs a simple pickup-drop transition where it rotates to the position of the sample, grabs the sample, rotates to the position of the payload bay and drops the sample in the bay. The arm control logic also sends a *"Close Payload Bay"* command that actuates the payload motor and closes the nosecone, effectively completing the sample retrieval process.

5.1.6. Communication

The AGSE communicates between its various subsystems using a few different protocols. At the high level, the different control boards of the AGSE communicate using ROS over a wired ethernet LAN. On the motor control board, the servo motors are directly connected to a 3.3V serial port on the GPIO pins of the BeagleBone Black. Because the servo motors communicate at 5V using a 1-wire communications line, the communications transmitted to the servos from the BeagleBone Black passes through a buffer implemented with NPN transistors. Because the servos can be daisy-chained, only one connection is made to the Motor Control Cape and each servo is connected in series through the previous servo. This daisy-chaining greatly simplifies the wire routing and increases the modularity and maintainability of the AGSE motor hardware. Finally, the NVIDIA Jetson

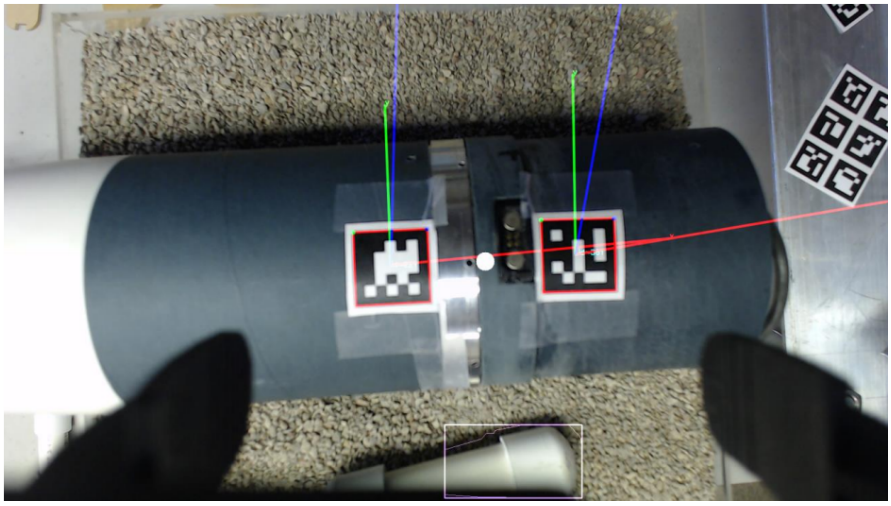

Figure 15. AGSE Payload Bay Detection

communicates to the Payload Bay Arduino to open and close through its own USB port acting as a virtual serial port. This serial connection is implemented as a quick-disconnect USB cable connected to a port on the surface of the rocket.

5.2. AGSE Changes Since the Competition

Post competition, minor mechanical and electrical improvements were made to the AGSE. A new mounting bracket was machined to replace the temporary mounting of the Dynamixel MX-28T that occurred the day of the competition. A new undercarriage made of extruded 20mmx20mm aluminum was fabricated to house the on-board power circuitry and embedded systems. Additionally, new power protection circuitry was made in the form of fuse boards and a dedicated emergency stop button in lieu of the key switch in the UIP.

5.3. Competition ROSMOD

ROSMOD exists as two separate code-bases: 1) the ROSMOD component model implementation, which enables the configuration of the component operation queue to support different scheduling schemes and priorities/deadlines for operations, and 2) the ROSMOD graphical modeling, generation, and deployment tool suite. These two code-bases are referred to as ROSMOD-COM and ROSMOD-GUI, respectively. All ROSMOD related code (old and new) can be found open-sourced online in the ROSMOD Github organization[17].

ROSMOD-COM is a package which extends the functionality of the *ROS Callback Queue* into which timer, subscriber and service operations are placed. To enable the addition of scheduling, priority, and deadline attributes to operations in the queue, all the relevant objects were modified, e.g. Timer, Subscriber, etc. Additionally, the queue was extended to support priority insertion of operations based on their deadline (EDF scheduling) and priority (PFIFO). By using the relevant classes and methods from ROSMOD-COM, ROSMOD components can ensure the proper scheduling of their operations.

The version of ROSMOD that was used in the competition can also be found open-sourced online[5], as the *v0.3-beta* release. Note that the version of the ROSMOD-GUI that was used is under the *gui* folder and is a Python-based program for which the relevant dependencies must be installed following the *README*. This is one of the primary problems with this version. Firstly, ROSMOD requires Linux and the Python programming language and a long list of dependencies. Also, this version, as used in the competition was primarily used for code generation and compilation. The deployment framework had several shortcomings, primarily due to the metamodeling language at

the time. Many of these shortcomings have now been addressed e.g. the concept of an experiment, a much improved network model etc.

5.4. ROSMOD Changes Since the Competition

The current version of ROSMOD is split into two separate repositories under the ROSMOD organization: the new version of the GUI[18], and the current version of ROSMOD-COM[5].

ROSMOD-COM has been updated to be a standalone ROS package so that it no longer has to be compiled inside ROS' source tree. Upon installing ROSMOD-COM, the scheduling schemes and associated configuration and classes become available for use in all packages.

ROSMOD-GUI has been transformed into a more fully integrated development environment which is now platform independent as it runs within the browser. The server runs within a Linux environment and can be locally deployed on a laptop or virtual machine, or on a real server. The change to this new infrastructure was enabled by the switch to using WebGME[19] as the backend for the modeling and server execution. This change also enables collaborative simulataneous model creation and editing between multiple users, with built-in git-sytle versioning, branching, and merging.

Despite the increase in backend complexity and number of features of ROSMOD, the use of ROSMOD and the installation of the backend has become far simpler and completely automated by widely-used packaging tools. The repository has installation instructions and users guides.

Acknowledgments: 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.

Author Contributions: Gabor Karsai conceived and designed the ROSMOD component model. Pranav Srinivas Kumar and William Emfinger were the primary software developers for the ROSMOD toolsuite, the ROSMOD component model, and the AGSE control software. Amrutur Anilkumar and Benjamin Gasser designed the AGSE robot and lead the Vanderbilt Aerospace Design Laboratory. Dexter Watkins, and Benjamin machined and constructed the AGSE robot. William Emfinger performed the AGSE execution time experiments. Pranav Srinivas Kumar analyzed the data and generated execution plots. Pranav Srinivas Kumar, William Emfinger and Dexter Watkins wrote the paper.

Conflicts of Interest: The authors declare no conflict of interest. The founding sponsors had no role in the design of the study; in the collection, analyses, or interpretation of data; in the writing of the manuscript, and in the decision to publish the results.

Abbreviations

The following abbreviations are used in this manuscript:

AGSE: Autonomous Ground Support Equipment
ARM: Advanced RISC Machines
BBB: Beaglebone Black
CNC: Computer Numerical Control
CPS: Cyber-Physical System
CPU: Central Processing Unit
CUDA: Compute Unified Device Architecture
CV: Computer Vision
DARPA: Defense Advanced Research Projects Agency
DOF: Degrees of Freedom
EDF: Earliest Deadline First

FIFO: First-In First-Out
GPIO: General Purpose Input/Output
HSV: Hue-Saturation-Value
IP: Internet Protocol
LAN: Local Area Network
LCD: Liquid Crystal Display
LED: Light Emitting Diode
MAV: Mars Ascent Vehicle
NASA: National Aeronautics and Space Administration
PFIFO: Priority First-In First-Out
PID: Proportional-Integral-Derivative
QEP: Quadrature Encoded Pulses
RGB: Red-Green-Blue
RMI: Remote Method Invocation
RML: ROSMOD Modeling Language
ROS: Robot Operating System
SFIO: Special Function Input/Output
SSH: Secure Socket Shell
SLI: Student Launch Initiative
UAV: Unmanned Aerial Vehicle
URL: Uniform Resource Identifier
UUV: Unmanned Underwater Vehicle
UIP: User Interface Panel
USB: Universal Serial Bus
XML: Extensible Markup Language

References

1. Brown, O.; Eremenko, P. Fractionated space architectures: a vision for responsive space. Technical report, DTIC Document, 2006.
2. Autosar GbR. AUTomotive Open System ARchitecture. http://www.autosar.org/.
3. Quigley, M.; Conley, K.; Gerkey, B.P.; Faust, J.; Foote, T.; Leibs, J.; Wheeler, R.; Ng, A.Y. ROS: an open-source Robot Operating System. ICRA Workshop on Open Source Software, 2009.
4. DARPA Robotics Challenge. http://www.theroboticschallenge.org//.
5. ROSMOD Repository. https://github.com/rosmod/rosmod.
6. NASA Student Launch. http://www.nasa.gov/audience/forstudents/studentlaunch/home/.
7. Sztipanovits, J.; Karsai, G. Model-integrated computing. *Computer* **1997**, *30*, 110–111.
8. Crnkovic, I.; Chaudron, M.; Larsson, S. Component-based development process and component lifecycle. Software Engineering Advances, International Conference on. IEEE, 2006, pp. 44–44.
9. Heineman, G.T.; Councill, B.T. *Component-Based Software Engineering: Putting the Pieces Together*; Addison-Wesley: Reading, Massachusetts, 2001.
10. Otte, W.R.; DUBEY, A.; Pradhan, S.; Patil, P.; Gokhale, A.; Karsai, G.; Willemsen, J. F6com: A component model for resource-constrained and dynamic space-based computing environments. Object/Component/Service-Oriented Real-Time Distributed Computing (ISORC), 2013 IEEE 16th International Symposium on. IEEE, 2013, pp. 1–8.
11. Eugster, P.T.; Felber, P.A.; Guerraoui, R.; Kermarrec, A.M. The Many Faces of Publish/Subscribe. *ACM Comput. Surv.* **2003**, *35*, 114–131.
12. Emmerich, W.; Kaveh, N. Component technologies: Java beans, COM, CORBA, RMI, EJB and the CORBA component model. Software Engineering, 2002. ICSE 2002. Proceedings of the 24rd International Conference on, 2002, pp. 691–692.

13. Lehoczky, J.P. Fixed Priority Scheduling of Periodic Task Sets with Arbitrary Deadlines. RTSS, 1990, Vol. 90, pp. 201–209.
14. Liu, C.L.; Layland, J.W. Scheduling algorithms for multiprogramming in a hard-real-time environment. *Journal of the ACM (JACM)* **1973**, *20*, 46–61.
15. AGSE 2015 Competition Repository. https://github.com/finger563/agse2015.
16. AGSE Repository. https://github.com/vadl/agse.
17. ROSMOD Organization. https://github.com/rosmod.
18. ROSMOD Tool Suite Repository. https://github.com/rosmod/webgme-rosmod.
19. Maróti, M.; Kecskés, T.; Kereskényi, R.; Broll, B.; Völgyesi, P.; Jurácz, L.; Levendovszky, T.; Lédeczi, Á. Next Generation (Meta) Modeling: Web-and Cloud-based Collaborative Tool Infrastructure. MPM@ MoDELS, 2014, pp. 41–60.