electronics

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, Nashville, TN 37235, USA; emfinger@isis.vanderbilt.edu (W.E.); gabor@isis.vanderbilt.edu (G.K.)

2 Mechanical Engineering, Vanderbilt University, Nashville, TN 37235, USA; dexter.a.watkins@vanderbilt.edu (D.W.); benjamin.w.gasser@vanderbilt.edu (B.G.); amrutur.v.anilkumar@vanderbilt.edu (A.A.)

* Correspondence: pkumar@isis.vanderbilt.edu; Tel.: +1-615-414-1561

† Current address: 1025 16th Ave S, Nashville, TN 37212, USA

‡ 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, Amsterdam, The Netherlands, 8–9 October 2015.

Academic Editors: Amer Baghdadi, Gabriela Nicolescu and Frédéric Rousseau
Received: 1 April 2016; Accepted: 23 August 2016; Published: 7 September 2016

Abstract: This paper presents the Robot Operating System Model-driven development tool suite, (ROSMOD) an integrated development environment for rapid prototyping component-based software 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: rapid; model-driven; development; robotics; distributed; real-time; embedded; cyber-physical; systems

1. Introduction

Robotics is an experimental science; one that involves integration between heterogeneous fields such as electronics, mechanics, computer science, control theory and artificial intelligence. Some of the technologies used by these various fields, e.g., simulation engines [1,2], embedded single-board computers [3,4], programming languages [5–7], algorithms [8–11], modeling paradigms [12–14] etc., are rapidly changing over time to keep up with advances in science, information technology and electronics. During the design, development and deployment of robotic systems, the components that belong to the various sub-systems, e.g., image processing software, servo motors, power distribution system etc., must work congruously to achieve a common goal. The integrated software in robotics must be flexible and robust enough to handle variabilities during a development cycle such as changes in sensors, control logic or performance modes. Hardware replacements/changes may be necessary during the lifetime of a robot, that can sometimes neither be foreseen during the initial design nor be simple.

One way to reasonably tackle the variabilities in such tightly integrated systems is to develop a model-driven tool that formalizes and promotes rapid system-level design, development, integration, and reuse; one that enables quick changes during design-time and run-time. During the life-cycle of a robotics project, *models* of the robotics software, and hardware are refined until the robot performs satisfactorily enough to meet both its design specification and user's expectations. Model-driven component-based software engineering (CBSE) and development [15–19] has become an accepted practice for tackling software complexity in large-scale embedded robotic software. CBSE tackles escalated demands with respect to requirements engineering, high-level design, error detection, tool integration, verification and maintenance. The widespread use of component technologies in the market has made CBSE a focused field of research in the academic sectors. Applications are built by assembling together small, tested component building blocks that implement a set of services. These building blocks are typically built from design models and captured in UML [20] class diagrams, or imported from other projects/vendors and *connected* together via exposed interfaces, providing a "black box" approach to software construction. This approach also treats software verification in a more modular fashion; the various software components can be verified individually and then composed together to derive a functional system.

The component model that is described in this paper also backs this idea of analysis and verification; the model provides guidelines to users for code development and exhibits well-defined execution semantics that makes the run-time application behavior more amenable to analysis. Though not in the scope of this paper, we have published in previous work some timing [21–23] and network analysis methods [24–26] that have examined the timing and network properties of composed component assemblies, using component models such as ROSMOD, with the component execution semantics presented here.

Along with the component model, this paper also describes the ROSMOD infrastructure, which includes a modeling language, a graphical user interface, code generators, deployment and run-time management features. We discuss the utility of ROSMOD in a real-world case study, the Autonomous Ground Support Equipment (AGSE) robot, that was designed and built by the Vanderbilt Aerospace Design Laboratory, which competed and won the NASA Student Launch Competition (2014–2015).

1.1. Changes Since RSP 2015

The work presented in this paper is an extension of a ROSMOD prototype framework [27]. The work presented here is a fully revised, re-designed, implementation of the core principles; the design decisions regarding the modeling language, artifact management, and deployment and monitoring infrastructure are derived from our experiences with this prototype. The goal of this re-implementation, and therefore also this paper, is to present a framework that aims to be optimized for rapid system development from the perspective of the developer. The prototyped ROSMOD work was implemented with Python, using a GUI development framework called wxPython [28]. This implementation used a simple textual language with an interactive graphical rendering environment, but also had several caveats that often lead to state corruption, and made true collaborative development impossible. Also, this framework did not allow developers to start *jobs* asynchronously through the various aspects in the framework, e.g., it is impossible in this prototype to start multiple experiments at the same time in the same instance of ROSMOD, i.e., the developer has to complete tests/experiments sequentially.

In this paper, we present a fully re-designed ROSMOD that relies on a stable in-house modeling environment, called WebGME [29]. This ROSMOD framework supports asynchronous job scheduling, collaborative code development and model editing features, a more complete meta-model with various integrated analysis features, and a more robust deployment and run-time management infrastructure. Lastly, we have chosen to integrate the code development and compilation process within the graphical user interface in order to keep the framework self-sufficient. This means that code development, code generation, compilation, process deployment, run-time monitoring, and execution time plot generation can all be done within the ROSMOD framework, i.e., a web page; this level of integration is completely streamlined and new to ROSMOD.

1.2. Terms Used in this Paper

- Project: A collection comprised of a software model, a set of system models, a set of deployment models, and a set of experiments.
- Package: A package refers to an application, i.e., a software sub-system.
- Component: a re-usable software building-block of an application.
- Component Instance: an instance of a component, properties of which can be overridden by the instance.
- Port: a communications interface connecting one component to other components.
- Client: a port providing blocking, synchronous request-response interactions.
- Server: a port receiving and responding to client requests.
- Publisher: a port providing anonymous one to one or one to many message passing interactions.
- Subscriber: a port receiving data from one or more publishers.
- Timer: a mechanism used to periodically or sporadically trigger a component.
- Host: a single computer running a Linux-based operating system.
- System: a collection of networked hosts.
- System Model: a model of system
- Node: a POSIX process that executes a collection of components in parallel.
- Deployment Model: A mapping between component instances and nodes
- Experiment Model: A mapping between a deployment model and a system model
- Experiment: the act of executing nodes in a subset of system hosts, specified by the experiment model.

2. Related Research

Software engineering tools and practices for robotics is a large, rapidly evolving field encompassing many disciplines. Much of the progress in this field can be found in the IEEE International Conference on Simulation, Modeling, and Programming for Autonomous Robots (SIMPAR) [30], the Workshop on Domain-Specific Languages and models for Robotic Systems (DSLRob) [31], and the Journal of Software Engineering for Robotics (JOSER) [32].

In the field of Robotics and embedded systems, a large number of component models have been presented and evaluated in literature. These component models fall under one of two categories: (1) component models that implicitly adopt a specific middleware, or programming language and (2) component models that are deliberately general purpose, enabling its use over a wider range of applications. Here, we discuss a few state of the art component models that have been developed and improved over the last 10–15 years and designed for robotic system development and integration.

The CORBA Component Model (CCM) [33] is one such model, pushed by the Object Management Group [34] as a standardized middleware specification to promote distributed real-time systems. CCM components are defined through an object-based component definition language which uses an IDL [35] for interface description. Fully defined components are implemented, where the implementation describes how the component pieces interact with each other. The resulting implementations are packaged into shared libraries and linked dynamically by the run-time system. CCM also provides a deployment mechanism to deploy the components in a CORBA component server; deployment instructions are specified by XML files. CCM components can interact with other components using *ports*. CCM supports four basic types of ports: facets, receptacles, event sources, and event sinks. Facets are *provided* interfaces that present functionality to other interacting components. Provided interfaces define operations that can be invoked either synchronously or asynchronously. A receptacle is a *required* interface; using receptacles, components can connect with other component instances and invoke operations. Event sources and sinks provide event-based communication between component instances. A component can either publish or subscribe to events. To address the issue of CORBA having a heavy memory/storage footprint, UCM [36] is being designed and developed as an alternative OMG standard providing interoperability with CCM and other component models/middlewares.

Similarly, SmartSoft [37–39] is a component model for robotic software, that uses communication patterns and is characterized by dynamic run-time configuration of components. SmartSoft aims to provide a generic software architecture without enforcing a specific robot architecture. There exists a CORBA-based implementation called CORBA SmartSoft that uses the TAO ORB [40] implementation of CORBA, as well as an implementation based on ACE [41]. A SmartSOft component is implemented as a set of threads and interacts with other components using well-defined communication patterns. Components can be dynamically configured at run-time to exhibit specific communication patterns.

Orca [42] is an open-source component model for component-based software development for mobile robots. The Orca framework attempts to impose very few constraints on developers, i.e., developers are free to design components that can provide or require any set of interfaces and the implementations for these interfaces can be provided as necessary. There are also little to no architecture constraints during the composition of Orca components. The Orca component model relies on a middleware that can handle the issues of communication protocols and provide a means for interactions between components. Orca implementations have been released with both CORBA and the Internet Communication Environment (ICE) [43]. Similar to other component models here, interface descriptions and interactions are configured using XML configuration files, per component.

The Open Robot Control Software (OROCOS) [44] is a component-based software framework for robotic control software development in both real-time and non-real-time operating systems. OROCOS supports event-driven interaction between components, where each component consists of a set of properties, ports, operations, and a single activity, i.e., a task with a defined execution frequency. Similar to CCM, the system is configured by a *deployer* (using XML) which declares all OROCOS components and port connections. The execution semantics of each component is described using an internal finite state machine, including initialization, pre-operation, start, update, and clean up states. The OROCOS framework is available both standalone and as part of the Robot Operating System (ROS), a middleware platform used by the ROSMOD platform presented in this paper.

In the past, our team has designed, implemented and tested the F6COM/DREMS component model [45] for tackling software engineering challenges in distributed real-time embedded systems. Each DREMS component supports four basic types of ports for interaction with other collaborating components: Facets, Receptacles, Publishers and Subscribers. A component's *facet* is a unique interface that can be invoked either synchronously via remote method invocation (RMI) or asynchronously via asynchronous method invocation (AMI). A component's *receptacle* specifies an interface required by the component in order to function correctly. Using its receptacle, a component can invoke operations on other components using either RMI or AMI. A *publisher* port is a single point of data emission and a *subscriber* port is a single point of data consumption. Communication between publishers and subscribers is contingent on the compatibility of their associated topics (i.e., data types). More details on this component model can be found in [45].

ROS [46] is a free, open-source 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 and language-neutral design of ROS has made it a requirement in several DARPA robotics projects including the *DARPA Robotics Challenge* [47]. The ROS component model defines nodes, packages, messages, topics and services. Nodes represent the software components and are deployed as processes.

ROS 2.0 [48], while originally proposed to build off of other communications middlewares such as ZeroMQ [49], has been designed to leverage DDS [50] as the underlying communications framework. These pre-existing communications libraries provide the distributed communications functionality which made ROS useful in robotics, and provide much of the interaction pattern specification and implementation relevant for specifying a component model.

The BRICS [51] project aims to provide structure and formalization for developing robots and for increasing interoperability of robot hardware and software components through well defined interfaces on its component model [52] and by providing an integrated development environment, BRIDE [53], based on the Ecplipse [54] platform. BRICS, through BRIDE, supports model-based and component-based code generation for various robotics libraries such as OROCOS and ROS.

The current version of Matlab's Simulink [18] supports the *Robotics System Toolbox* [55] (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 [56], 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.

In this paper, we describe ROSMOD, an open-source development tool suite and run-time software platform for rapid prototyping of component-based software applications using ROS. This software organization consists of several software pieces: (1) A Model-driven development environment; (2) plugins for code generation, software deployment, and run-time management; (3) visualizers for user-defined views of the models and artifacts, e.g., plots generated from run-time logs; and finally (4) the run-time platform: the ROSMOD component model that is distributed as an installable ROS package. The component model in ROSMOD is similar to F6COM in the execution semantics and threading model of the component, i.e., each component has an executor thread, a message queue to receive requests/messages from other components, and the component execution scheduler—the non-preemptive nature of the operation scheduling in each component. The reason for this choice is also to be able to directly apply some formal timing and network analysis methods we have developed in the past, based on F6COM and the DREMS infrastructure, with little to no changes in the analysis models. Additionally, the non-preemptive operation scheduling means that component developers do not have to write locking or synchronization primitives for data shared between operations in the same component.

ROSMOD relies on ROS for serialization, message passing and discovery. Since much of the back-end communication is ROS, ROSMOD also uses ROS's description language for message and service data structures. However, it must be noted that this is not a strong binding. The modeling language, and rapid system integration methods presented in this paper are not strictly tied to ROS, aside from some naming conventions. The code generation and deployment framework in ROSMOD is loosely coupled with the models and therefore, much of this can be swapped out as needed. We are currently in the process of supporting other middleware technologies, e.g., ZeroMQ [49] and OMG DDS [57], and using serialization methods such as Google Protocol Buffers [58], though this integration work isn't with in the scope of this paper.

3. ROSMOD Component Model

The ROSMOD component is a refinement of the ROS component model. In this section, we detail some of the aspects of both component models, describe the differences and present the execution semantics of ROSMOD. Many of the design decisions in ROSMOD, are motivated by the need to keep the model of the software simple and easy to prototype/analyze.

The ROS component model defines *messages*, *topics*, *services*, *packages*, and *nodes*. Nodes represent the software components in ROS and are deployed as independent processes on a local machine or on a set of networked computers. Nodes communicate with each other by sending/receiving messages. In ROS, messages are strictly-typed data structures, defined by language-independent IDL. Complex messages can be constructed by using the primitive data types supported by the IDL, that are composed to arbitrary depth. Upon compilation, ROS messages are translated into serializable data structures. Topics support asynchronous communication and Services support synchronous communication between nodes.

Nodes can send messages to other nodes by *publishing* data to a given topic. A node that is interested in this data can *subscribe* to the appropriate topic. When a new message is received by the subscribing node, a callback function is asynchronously called to handle and receive the message in the user-space. Alternatively, a ROS node can perform a synchronous method invocation on another node using services. A service is characterized by two typed messages: the request message, and the response message. The client node populates the request message and calls the service; the server node receives the request and responds to the client by populating the response message. ROS nodes are grouped into packages. A package is an application that can be compiled, and executed, i.e., a package represents the minimal compilable unit of ROS. Packages can also be used to wrap external libraries.

In ROSMOD, software 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 [59]. 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 [60]. Component timers can be periodic or sporadic and allow components to trigger themselves with the specified timing characteristics. These aspects of ROSMOD are quite similar to ROS. The main difference between ROS and ROSMOD is the threading model and the execution semantics of software components. As described earlier, the software components in ROS are ROS nodes which are independent processes. In ROSMOD, each component is represented by an executor thread and a component operation queue, as shown in Figure 1.

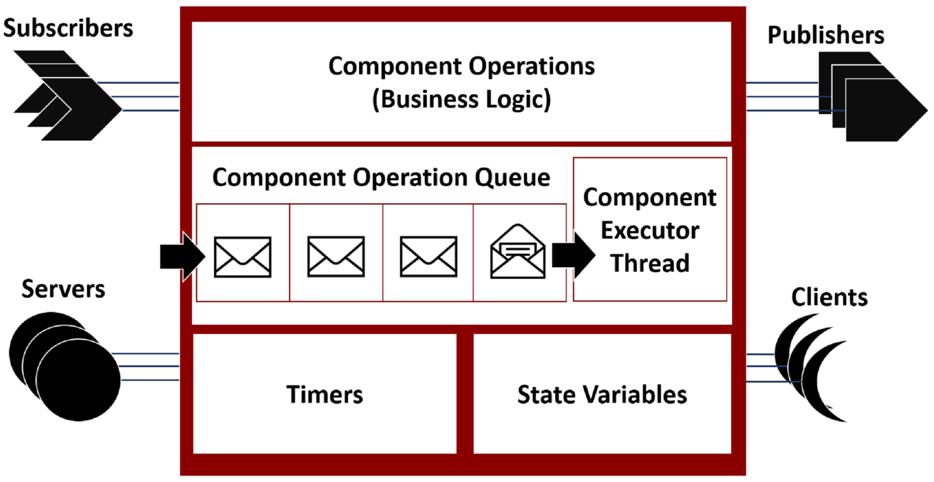


Figure 1. ROSMOD Component.

ROSMOD Component Execution Semantics

An *operation* is an abstraction for the different tasks undertaken by a component. These tasks are implemented by the component's source code written by the developer. Application developers provide the functional, *business-logic* code that implements operations on local state variables and inputs received on component ports. For example, a PID controller [61] could receive the measured value of a process variable from a *sensor* component, and using the relevant gains and set points (desired value for the process variable), calculate a new state to which an *actuator* component should progress the system.

In order to service interactions with the underlying framework and with other components, each component has a *message queue*. This queue holds operation requests received from another component, i.e., messages, service requests, and timer activations. Each request is characterized by a priority and a deadline. Priority refers to the relative importance of one operation over another within the scope of the component. Deadline refers to the worst-case duration from the arrival of a triggering event to the completion of the response operation. If the execution of the operation takes beyond its deadline to complete, then a *deadline violation* is said to have occurred.

Each ROSMOD component has a separate execution thread that handles the execution of all component operations. This executor thread picks the next available request from the message queue and executes the operation to completion, i.e., the operation scheduling is non-preemptive. So, all operations in the queue, regardless of priority, need to wait for the currently executing operation to complete. Allowing only a single executor thread per component and enforcing a single-threaded non-preemptive scheduling scheme on the operations helps avoid the need for synchronization primitives for internal state variables and establishes a more easily analyzable system. It is true that multi-threaded solutions to operation scheduling would avoid starvation, i.e., operations in the queue will not have to wait forever if the currently executing operation is blocked on a resource. However, the ROSMOD execution semantics is still the more predictable design as it is more easily analyzable. The non-determinism in multi-threaded executions causes a tree of possible behaviors, leading to a common analysis challenge called state space explosion [62]. Keeping the operation execution to a single thread per component bounds the overall number of threads in the system leading to a more tractable analysis. However, since these operations are executed without preemption (by other operations within the component), there are scenarios in which the component, set of components, or system can deadlock due to cyclic communications dependencies. A simple example is a component whose timer initiates a blocking client call to a server which is hosted by the same component; because the timer operation is blocked from finishing by the client, the server operation in the queue will never be serviced so the client and thus the timer will never unblock. To enable developers to check their designs for such cyclic communications dependencies, component assemblies can be automatically analyzed to ensure that there will be no deadlocks or operation deadline violations. These analysis techniques and tools are outside the scope of this paper, and have been published in previous work [21–23].

Figure 2 shows the execution semantics of a component operation executed on the component's executor thread. Simplifying assumptions include that this component is the only component thread executing on the CPU, assuming a single core CPU. This is a simple scenario showing how a single operation on a single component is affected by the operation scheduling semantics. The wait times of the remote request are also worsened by OS scheduling non-determinism—when multiple components are scheduled concurrently, fixed-priority thread scheduling is enforced.

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. It is important to note that ROS imposes no design constraints or principles on the nodes; a node can be any process with any number of threads. The component model in ROSMOD, i.e., the use of a single executor thread and a non-preemptive operation queue, enable a stricter variant of the base ROS component model—one where each component is essentially a thread, multiple components are grouped in a process (ROS node).

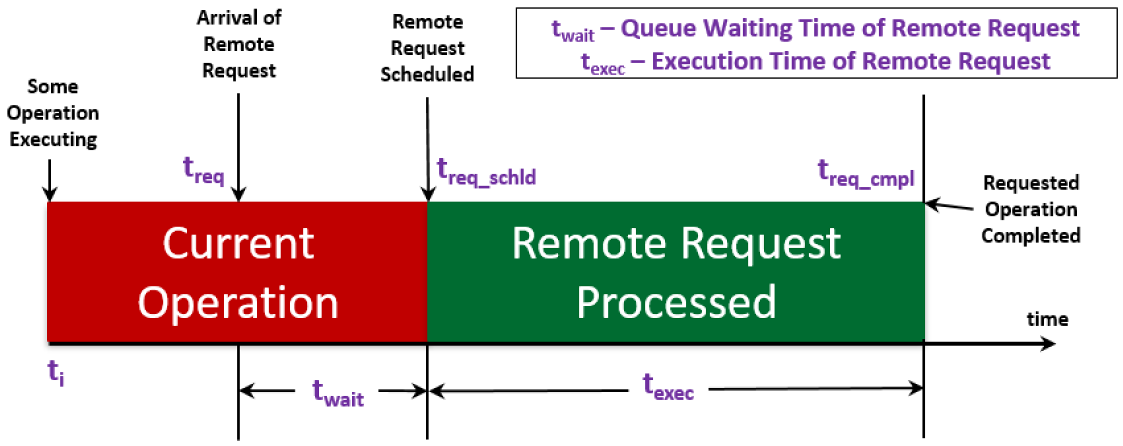


Figure 2. Component Operation Execution Semantics: This figure shows the ROSMOD operation scheduling on an incoming operation request. At some time t_i, the component executor thread is busy executing an operation—component operations can be triggered into execution by the (1) expiry of a timer; (2) the arrival of a subscription message; or (3) the arrival of a service request. t_req represents the arrival time of a remote request. t_wait is the wait time of this request in the message queue while the current operation is still executing. t_req_schld is the time stamp at which the current operation completes executing. At this time, the remote request is finally scheduled for execution. t_req_cmpl is the time stamp at which the remote request completes. The overall time taken by the component to respond to this request is calculated as: $t\_wait + t\_exec = t\_req\_cmpl - t\_req$.

4. ROSMOD Modeling Language

To enable the rapid design, development, and testing of software on robotics and distributed CPS in general, we have developed a domain-specific modeling language (DSML) [63], the ROSMOD Modeling Language (RML), that begins to address the heterogeneous needs of distributed robotic systems engineering. A DSML is a programming language or specification language that, using appropriate notations and abstractions, offers the expressive power focused on, and usually restricted to, a particular domain, e.g., distributed real-time systems.

The ROSMOD meta-model is captured from the developer mode of our graphical interface for model specification, which allows viewing and modification of the meta model and its rules which govern the construction of ROSMOD models. All objects contain a *name* attribute of type *string*. Any object shown in the meta figures which has an italics and grayed-out name is a pure *abstract* object, which cannot be instantiated in a model and merely serves as the base class for other instantiable classes. Note: the meta-model interface 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. RML captures all the relevant aspects of the software, 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. Figure 3 provides a reference legend for all meta-model figures in this section.

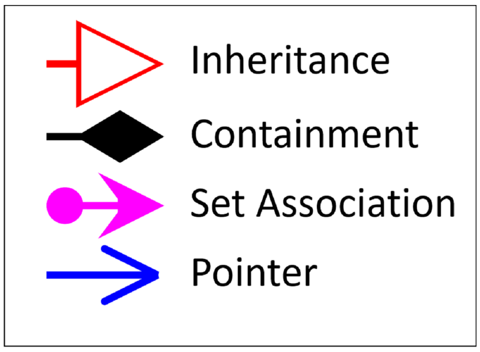


Figure 3. Legend for the relationships in ROSMOD META. Containment is specified from *src* to *dst* where the source has a containment attribute *quantity*, meaning that *quantity* objects of type *src* can be contained in an object of type *dst*. Pointers are specified as a one to one mapping from source to destination, using the name of the pointer. Sets allow for pointer containment.

The top-level entity of RML is a *Project*, which is shown in all of the following meta-model related figures, Figures 4–7. The meta-model has been broken up into separate figures for clarity and ease of understanding. 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.

It should be noted that the capability to specify the meta-model within the same tool and interface as the models themselves allow developers and users (if they choose) to extend or modify the meta-model based on their needs or experiences. Support for this kind of dynamic meta-model specification has been critical in the development of ROSMOD, particularly for enabling *rapid* evolution of the platform and faster feedback when any issues or new requirements are encountered. As such, this meta-model is not provided as a final, gold-standard model which we as developers are promoting for the design and development of *all* distributed CPS. Rather, this meta-model has arisen from our combined development and deployment experiences over the past few years with various tools. Further, this paper and the tool we present are designed to show the utility of such a framework for rapidly evolving projects, tool-suites, designs, and implementations.

4.1. Software Model

The *Software* class in Figure 4 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.

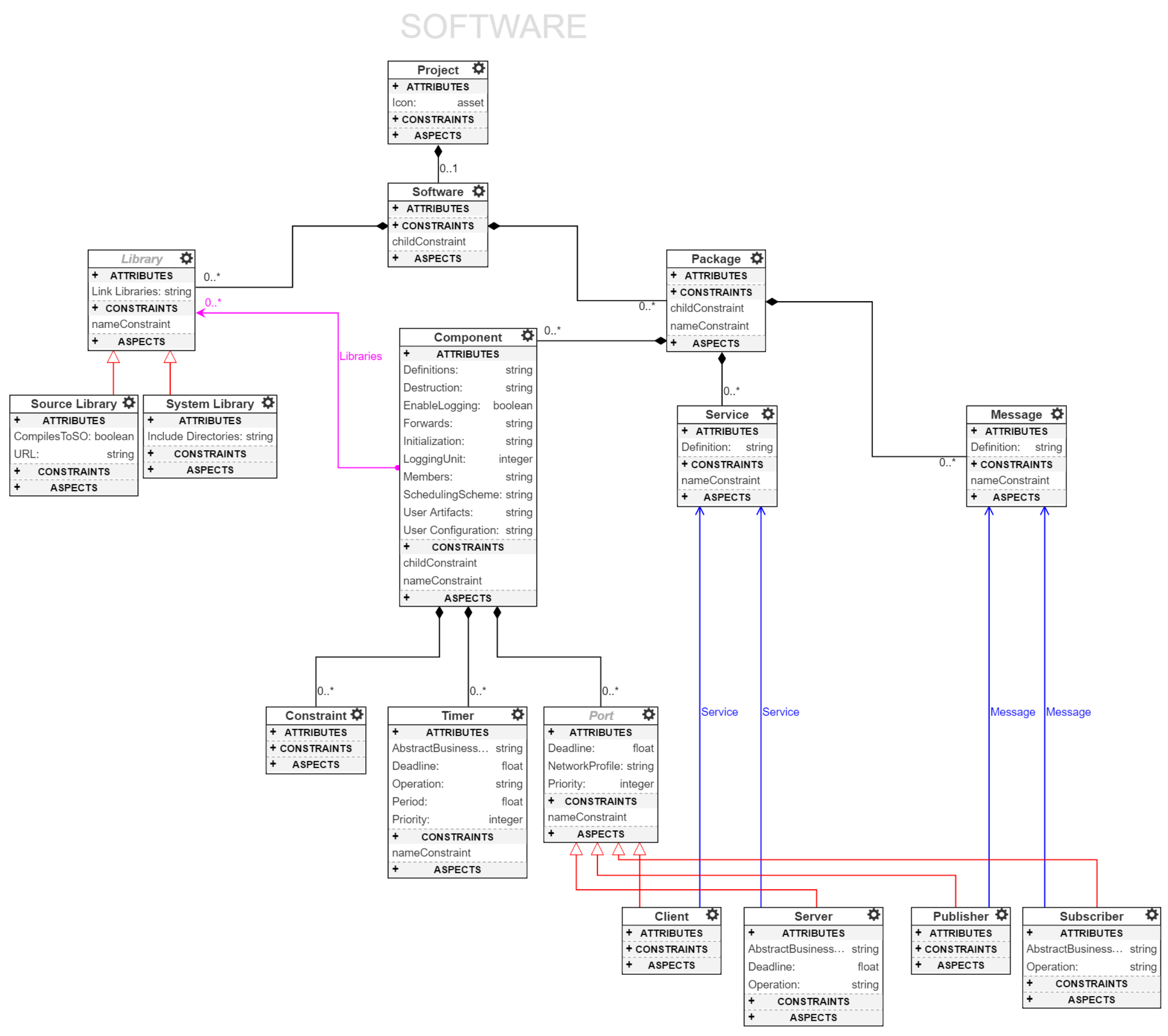


Figure 4. ROSMOD software metamodel. The software metamodel defines how the software is modeled. Since ROS is used as the middleware implementation, we have chosen to keep ROS naming conventions where ever possible.

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 measurement 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 at least 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 5.6.

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. Similarly, 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.

4.2. System Model

A *System Model*, see Figure 5, 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 bandwidth and latency values, we can configure the links of the network using the Linux TC [64] to enforce time-varying bandwidth and latency. This network configuration is useful when running experiments on laboratory hardware for which the network is not representative of the deployed system's network.

4.3. 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, Figure 6, 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 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*.

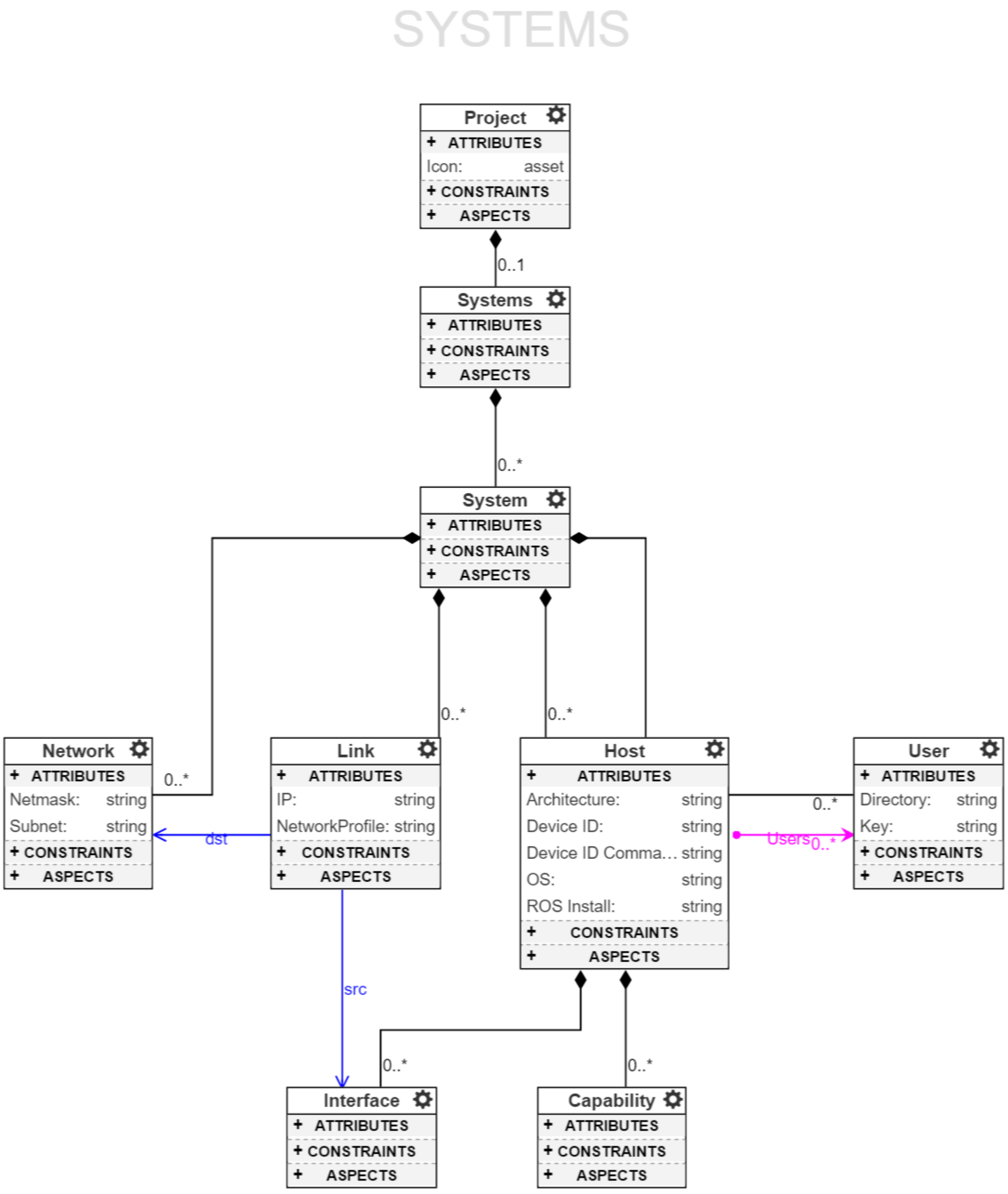


Figure 5. ROSMOD system hardware meta-model. System models define the hardware architectures, user roles, and networks for a (possibly distributed) CPS. A system is composed of a collection of hosts, each of which may have a different processor architecture, and each of which may have a collection of user accounts. Hosts may be connected to one or more communications networks.

As shown in Figure 7, 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 (4) 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.

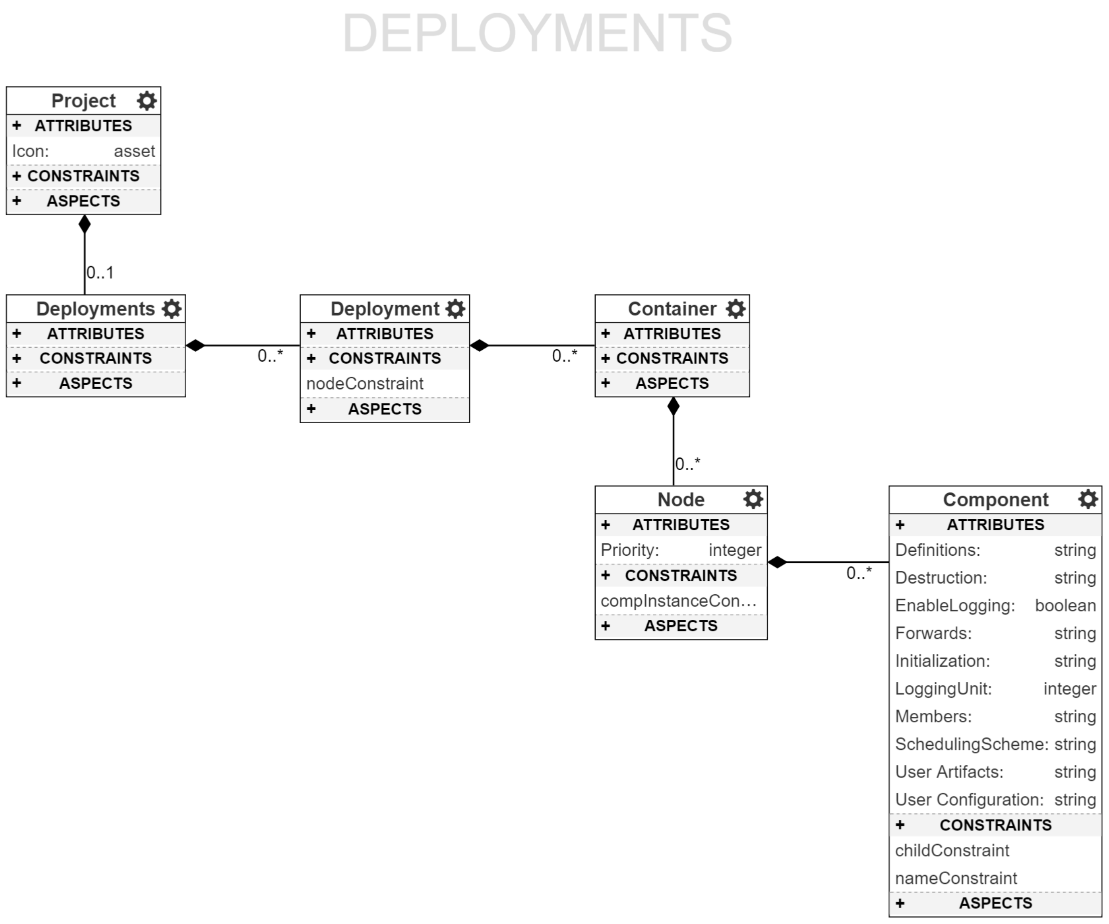


Figure 6. ROSMOD deployment metamodel. A deployment is a collection of containers, where a container is a set of colocated nodes. Users instantiate one or more component definitions from the software model into a node.

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.

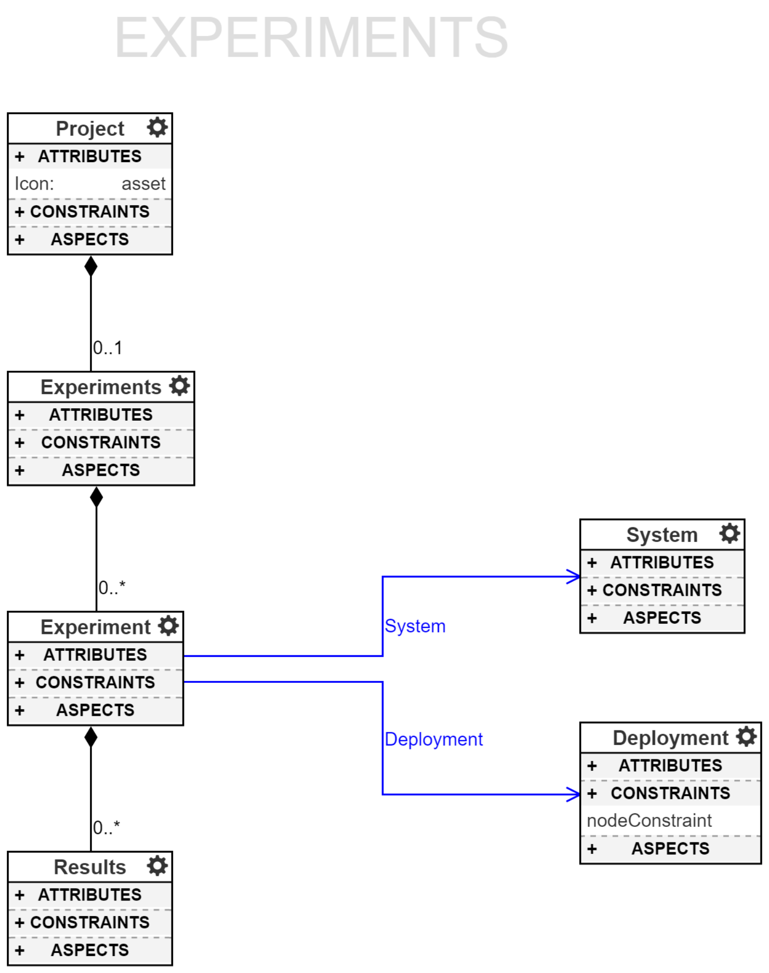


Figure 7. ROSMOD experiment meta-model. An experiment is a direct mapping between a deployment model and a system model. Before the experiment is executed, the constraints of the component instances of the deployment model are checked against the capabilities of the selected system. Similarly, the number of containers in the deployment model must be less than or equal to the number of hosts in the system model.

5. ROSMOD Infrastructure

In this section, we describe the aspects of the ROSMOD tool suite which allow it to function as an IDE for rapidly designing and developing distributed CPS. ROSMOD is designed to be quick for new users (i.e., system modelers, developers, and experimenters) to set up and use for designing and executing experiments on their systems. Secondly, ROSMOD models, and by extension the run-time C++ code, are built in an environment that is amenable to collaborative editing, i.e., multiple users who are logged into the same modeling environment from different parts of the world, and editing different aspects of the same models in a project. This sort of model-driven development is integral to achieving the main goals of ROSMOD—rapid system design and development that is easily accessible to a team of developers. To this end, we have decided to use WebGME [29], as it has proved to be easier to setup and use in collaborative environments where the engineers and developers that work in the project are often from different fields, e.g., mechanical engineers, aerospace engineers and electrical engineers, and also from different physical locations. A WebGME development environment is setup for ROSMOD, and hosted on a server machine as a web page; ROSMOD users simply log on to this web page and start working without having to setup a development environment on their individual machines. The administration of the host server machine is the only work that needs to be done in order to maintain a stable development environment. Thus, although it must be acknowledged that an integrated development environment like Eclipse [54] is powerful and widely popular, we argue that for developers, using a WebGME-style workflow is much more accessible and effortless for collaborative model-driven development.

5.1. WebGME

Before describing the exact structure of the interface and the backend architecture that powers the ROSMOD IDE, we first describe the new framework for web-based collaborative, versioned model editing on which ROSMOD is developed: WebGME [29,65,66]. WebGME provides the infrastructure and UI for enabling rapid meta-modeling and modeling, and is constantly being developed and updated with monthly releases.

The WebGME platform provides the infrastructure which enables collaboration between multiple users and organizations on and across multiple WebGME projects. WebGME handles and allows configuration of the authentication and user permissions within projects and organizations. In addition, each WebGME server instance manages the projects within a model database; within the database, changes to the models are committed and versioned in the same way as other VCS, like Git, enabling branching, merging, and viewing change history.

For access to the models' data within the database, the server exposes a set of APIs both WebSocket-based and REST APIs that enable simultaneous connections from client browsers or other user-created programs. All changes to the underlying model are versioned, and in the case of conflicts or simultaneous changes, automatic branching ensures users see consistent states. A diagram detailing the architecture of WebGME's client and server interfaces is shown in Figure 8.

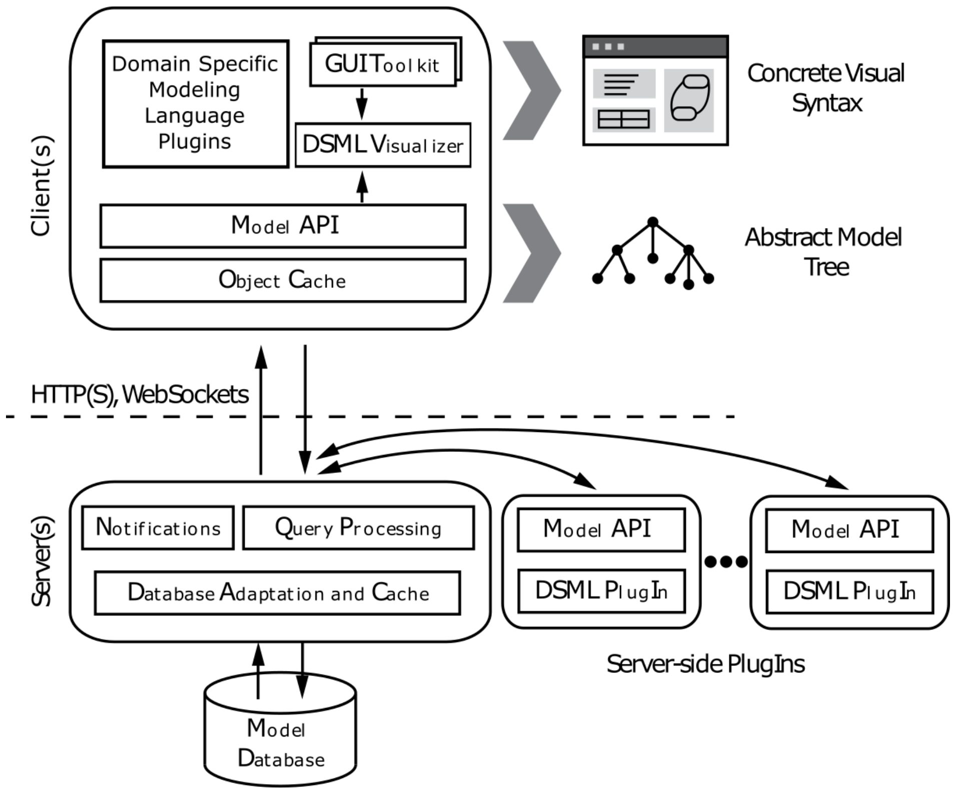
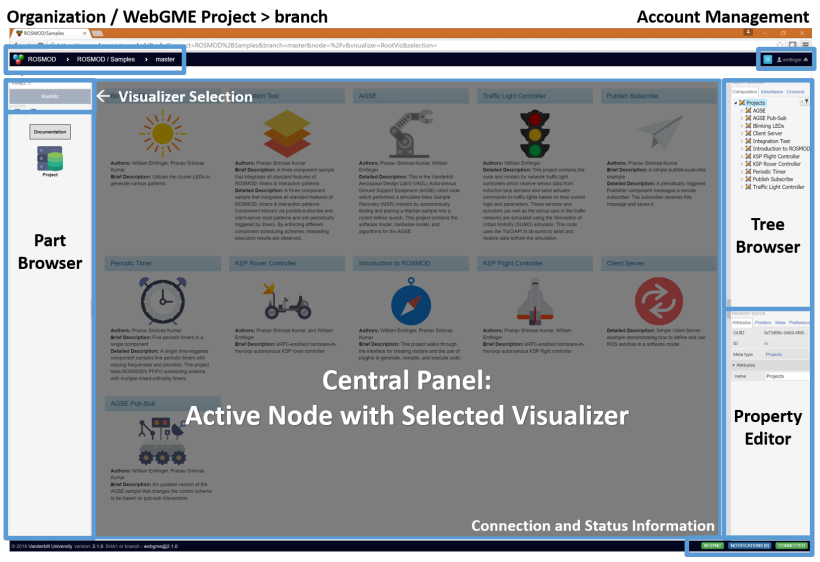


Figure 8. Architectural overview of WebGME and its server, browser, client, plugin, and visualizaion systems.

Finally, the WebGME platform allows users to configure their specific server deployments and projects with custom WebGME components: Layouts, Plugins, Visualizers, and Add-Ons. Users can quickly prototype and customize these custom WebGME components, for which the boilerplate code can be easily generated using an open-source command-line utility *webgme-cli* [67]. These components can be built into the server deployment or contained in separate repositories and installed automatically using Node.js' *npm* tool.

5.2. ROSMOD

The ROSMOD IDE and infrastructure exist on top of WebGME as a set of WebGME components that work with the meta-model described in Section 4. The purpose of each component is to enable rapid iteration of system design and experimentation. As such, ROSMOD provides a set of automation tools which perform asynchronous tasks such as code compilation and experiment deployment. In addition to the automation tools, ROSMOD contains a set of visualizers which provide an interface for viewing and editing specific parts of the model, such as for editing code attributes or viewing plots of experiment trace logs. Figure 9 shows the front-end workspace when logged into ROSMOD. As with the rest of ROSMOD, these WebGME components can be extended or replaced as needed by other developers.

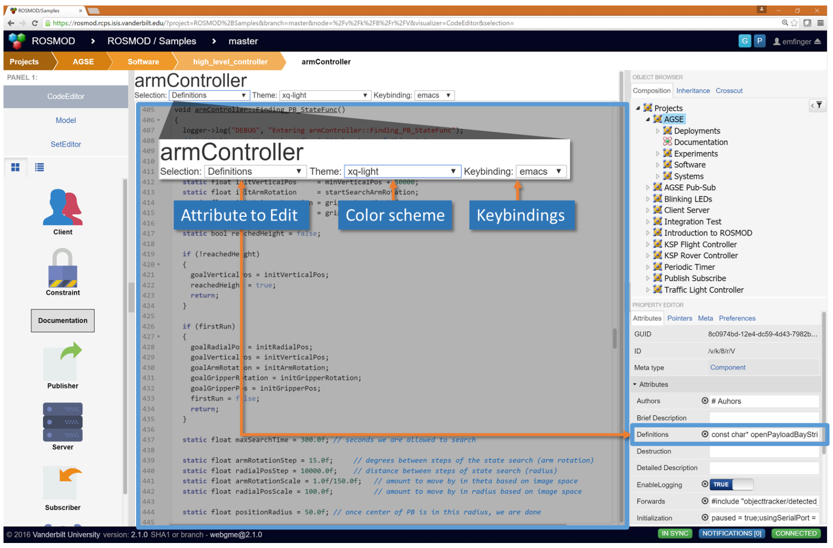


Organization / WebGME Project > branch
Account Management
ROSMOD > ROSMOD / Samples > master
Visualizer Selection
Part Browser
Tree Browser
Central Panel: Active Node with Selected Visualizer
Property Editor
Connection and Status Information

Figure 9. ROSMOD Projects Front Page. The UI layout of the ROSMOD WebGME deployment is comprised of the panels and widgets shown above.

5.3. Code Editor

The code editor, as shown in Figure 10, allows users to view and edit the various types of code and multi-line strings that exist as attributes of different model objects. The code editor supports syntax highlighting for languages such as C++, markdown, javascript, JSON, etc., while also allowing the user to configure their display and input perferences by setting the color scheme and the keybindings. Additionally, the code editor supports (regex-enabled) search and replace, code folding, code completion, and several other features. When the user edits the code, the code editor waits until the user has not made any changes for a configurable amount of time (default is one second) and then batches the changes together into a single commit pushed into the model database. In this way the code editing is automatically versioned and the code is always in sync with the model (since it is part of the model).



https://rosmod.rcps.isis.vanderbilt.edu/?project=ROSMOD%2BSamples&branch=master&node=%2Fv%2Fk%2F8%2Fr%2FV&visualizer=CodeEditor&selection=

armController
Selection: Definitions Theme: xq-light Keybinding: emacs

```
void armController::Finding_PB_StateFunc()
{
  logger->log("DEBUG", "Entering armController::Finding_PB_StateFunc");

  static float i itVerticalPos     = minVerticalPos + 50000;
  static float i itArmRotation     = startSearchArmRo ation;
                               n = gri
                                 = gri

  static bool re chedHeight = false;

  if (!reachedHe ght)
  {
    goalVertical os = initVerticalPos;
    reachedHeigh  = true;
    return;
  }

  if (firstRun)
  {
    goalRadialPo  = initRadialPos;
    goalVertical os = initVerticalPos;
    goalArmRotat on = initArmRotation;
    goalGripperR tation = initGripperRotation;
    goalGripperP s = initGripperPos;
    firstRun = f lse;
    return;
  }

  static float maxSearchTime = 300.0f; // seconds we are allowed to search

  static float armRotationStep = 15.0f;     // degrees between steps of the state search (arm rotation)
  static float radialPosStep = 10000.0f;    // distance between steps of state search (radius)
  static float armRotationScale = 1.0f/150.0f;   // amount to move by in theta based on image space
  static float radialPosScale = 100.0f;          // amount to move by in radius based on image space

  static float positionRadius = 50.0f; // once center of PB is in this radius, we are done
```

GUID	8c0974bd-12e4-dc59-4d43-7982b…
ID	/v/k/8/r/V
Meta type	Component
Attributes	
Authors	# Auhors
Brief Description	
Definitions	const char* openPayloadBayStri
Destruction	
Detailed Description	
EnableLogging	TRUE
Forwards	#include "objecttracker/detected
Initialization	paused = true;usingSerialPort =

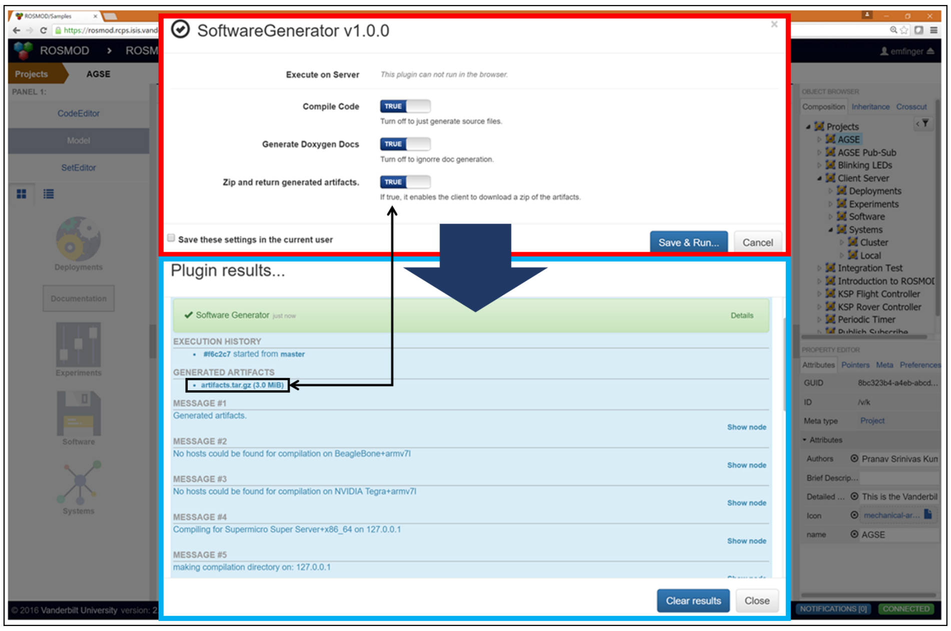
Figure 10. Code Editor Visualizer. The visualizer allows users to edit the relevant code, configuration, and documentation attributes for the active model object.

5.4. Code Generation

The Software generator in ROSMOD is a WebGME plugin (Figure 11) 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. 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. The code attributes, and their meanings is shown in Table 1. 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. The code compilation infrastructure of 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.

Table 1. Attributes stored in the component model and how they map to users' code. Each attribute contains information supplied by the user which gets inserted into the generated code.

Object Type	Attribute	Meaning
Component	Forwards	Header includes and forward declarations - This text is injected in the top of the component's header file
	Members	Private members and methods in the component's class
	Definitions	Global variables and functions available for use by the component
	Initialization	Initialization code—This is the first operation executed by the component executor thread
	Destructor	Cleanup code in the component's destructor
Timer	Operation	Callback function bound to a component timer
Publisher	N/A	Publisher object usable in component code
Subscriber	Operation	Callback function bound to a component subscriber - This operation handles messages received by the subscriber port
Client	N/A	Client object usable in component code
Server	Operation	Callback function bound to a component server— This operation handles requests received by the server


SoftwareGenerator v1.0.0

Execute on Server — *This plugin can not run in the browser.*

Compile Code: TRUE — Turn off to just generate source files.

Generate Doxygen Docs: TRUE — Turn off to ignorre doc generation.

Zip and return generated artifacts.: TRUE — If true, it enables the client to download a zip of the artifacts.

☐ Save these settings in the current user

Plugin results...

✓ Software Generator just now

EXECUTION HISTORY
- #f6c2c7 started from master

GENERATED ARTIFACTS
- artifacts.tar.gz (3.0 MiB)

MESSAGE #1
Generated artifacts.

MESSAGE #2
No hosts could be found for compilation on BeagleBone+armv7l

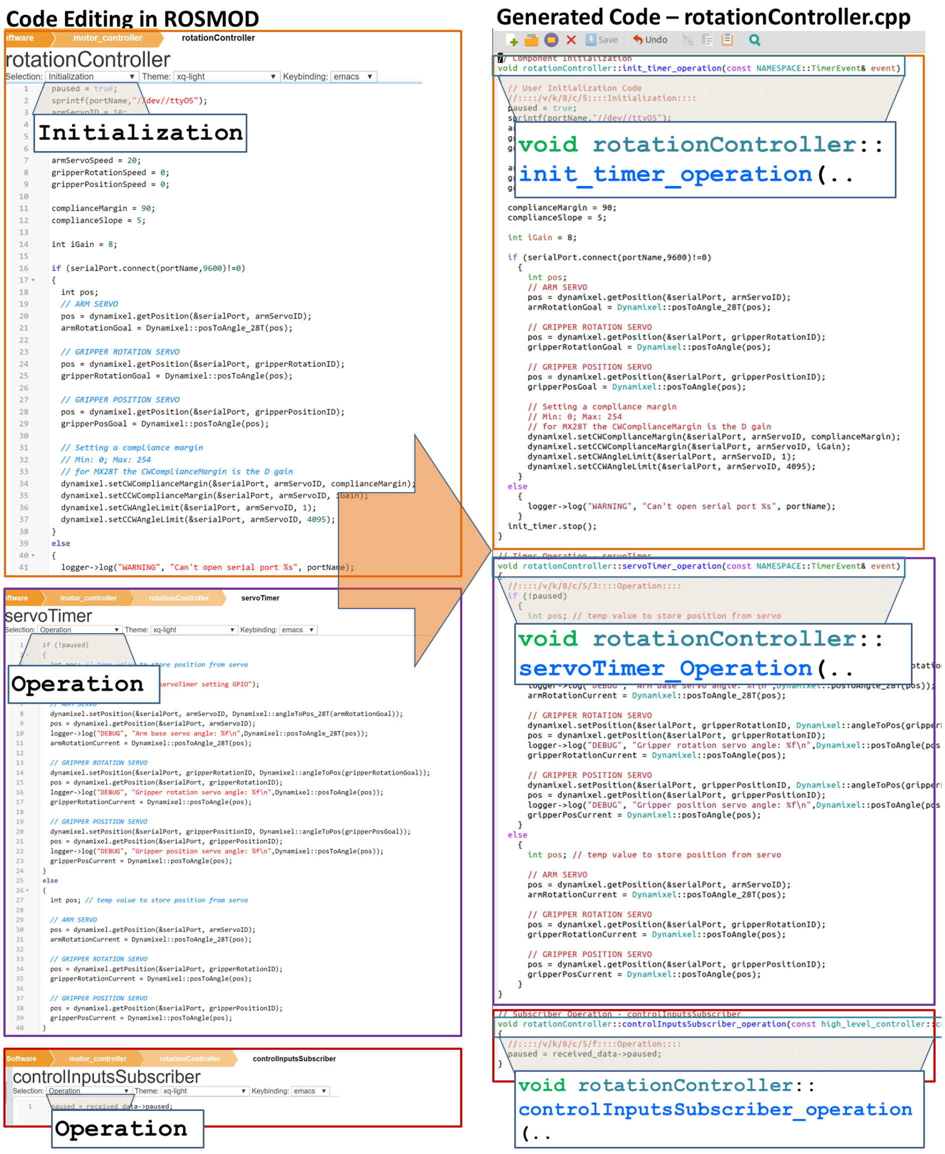
MESSAGE #3
No hosts could be found for compilation on NVIDIA Tegra+armv7l

MESSAGE #4
Compiling for Supermicro Super Server+x86_64 on 127.0.0.1

MESSAGE #5
making compilation directory on: 127.0.0.1

Figure 11. Software Generator Plugin and Results. After the user runs the software generator plugin and it completes, it will notify the user and can open the results page, which is shown in this figure for clarity. If the user requests, the plugin will return the generated source, binary, and documentation files for the software.

Figure 12 shows an example of the generated code for a component and some of the code attributes in the model from which it is generated. 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 and 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 either inform the infrastructure that the code was ready for compilation or compile it manually. This process was error prone and required detailed knowledge of not only what files are generated, but also how to configure the build system, and other details that were specific to both ROS' package system and our component implementation. 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 use of libraries without knowledge of how the underlying ROSMOD build system works. Such an infrastructure is beneficial when needing to support simultaneously multiple middlewares and build systems by freeing application developers from the requirement to train on the specific implementations of the middlewares and their build systems.



Code Editing in ROSMOD
Generated Code – rotationController.cpp
Software
motor_controller
rotationController
servoTimer
controlInputsSubscriber
Initialization
Operation
Operation
Save
Undo
void rotationController::init_timer_operation(..
void rotationController::servoTimer_Operation(..
void rotationController::controlInputsSubscriber_operation(..

rotationController

Selection: Initialization Theme: xq-light Keybinding: emacs

```
paused = true;
sprintf(portName,"//dev//ttyO5");
armServoID = 10;

armServoSpeed = 20;
gripperRotationSpeed = 0;
gripperPositionSpeed = 0;

complianceMargin = 90;
complianceSlope = 5;

int iGain = 8;

if (serialPort.connect(portName,9600)!=0)
{
  int pos;
  // ARM SERVO
  pos = dynamixel.getPosition(&serialPort, armServoID);
  armRotationGoal = Dynamixel::posToAngle_28T(pos);

  // GRIPPER ROTATION SERVO
  pos = dynamixel.getPosition(&serialPort, gripperRotationID);
  gripperRotationGoal = Dynamixel::posToAngle(pos);

  // GRIPPER POSITION SERVO
  pos = dynamixel.getPosition(&serialPort, gripperPositionID);
  gripperPosGoal = Dynamixel::posToAngle(pos);

  // Setting a compliance margin
  // Min: 0; Max: 254
  // for MX28T the CWComplianceMargin is the D gain
  dynamixel.setCWComplianceMargin(&serialPort, armServoID, complianceMargin);
  dynamixel.setCCWComplianceMargin(&serialPort, armServoID, iGain);
  dynamixel.setCWAngleLimit(&serialPort, armServoID, 1);
  dynamixel.setCCWAngleLimit(&serialPort, armServoID, 4095);
}
else
{
  logger->log("WARNING", "Can't open serial port %s", portName);
```

servoTimer

Selection: Operation Theme: xq-light Keybinding: emacs

```
if (!paused)
{
  int pos; // temp value to store position from servo
  servoTimer setting GPIO");

  dynamixel.setPosition(&serialPort, armServoID, Dynamixel::angleToPos_28T(armRotationGoal));
  pos = dynamixel.getPosition(&serialPort, armServoID);
  logger->log("DEBUG", "Arm base servo angle: %f\n",Dynamixel::posToAngle_28T(pos));
  armRotationCurrent = Dynamixel::posToAngle_28T(pos);

  // GRIPPER ROTATION SERVO
  dynamixel.setPosition(&serialPort, gripperRotationID, Dynamixel::angleToPos(gripperRotationGoal));
  pos = dynamixel.getPosition(&serialPort, gripperRotationID);
  logger->log("DEBUG", "Gripper rotation servo angle: %f\n",Dynamixel::posToAngle(pos));
  gripperRotationCurrent = Dynamixel::posToAngle(pos);

  // GRIPPER POSITION SERVO
  dynamixel.setPosition(&serialPort, gripperPositionID, Dynamixel::angleToPos(gripperPosGoal));
  pos = dynamixel.getPosition(&serialPort, gripperPositionID);
  logger->log("DEBUG", "Gripper position servo angle: %f\n",Dynamixel::posToAngle(pos));
  gripperPosCurrent = Dynamixel::posToAngle(pos);
}
else
{
  int pos; // temp value to store position from servo

  // ARM SERVO
  pos = dynamixel.getPosition(&serialPort, armServoID);
  armRotationCurrent = Dynamixel::posToAngle_28T(pos);

  // GRIPPER ROTATION SERVO
  pos = dynamixel.getPosition(&serialPort, gripperRotationID);
  gripperRotationCurrent = Dynamixel::posToAngle(pos);

  // GRIPPER POSITION SERVO
  pos = dynamixel.getPosition(&serialPort, gripperPositionID);
  gripperPosCurrent = Dynamixel::posToAngle(pos);
}
```

controlInputsSubscriber

Selection: Operation Theme: xq-light Keybinding: emacs

```
paused = received_data->paused;
```

```
// Component Initialization
void rotationController::init_timer_operation(const NAMESPACE::TimerEvent& event)
{
  // User Initialization Code
  //::::/v/k/8/c/5::::Initialization::::
  paused = true;
  sprintf(portName,"//dev//ttyO5");

  complianceMargin = 90;
  complianceSlope = 5;

  int iGain = 8;

  if (serialPort.connect(portName,9600)!=0)
    {
      int pos;
      // ARM SERVO
      pos = dynamixel.getPosition(&serialPort, armServoID);
      armRotationGoal = Dynamixel::posToAngle_28T(pos);

      // GRIPPER ROTATION SERVO
      pos = dynamixel.getPosition(&serialPort, gripperRotationID);
      gripperRotationGoal = Dynamixel::posToAngle(pos);

      // GRIPPER POSITION SERVO
      pos = dynamixel.getPosition(&serialPort, gripperPositionID);
      gripperPosGoal = Dynamixel::posToAngle(pos);

      // Setting a compliance margin
      // Min: 0; Max: 254
      // for MX28T the CWComplianceMargin is the D gain
      dynamixel.setCWComplianceMargin(&serialPort, armServoID, complianceMargin);
      dynamixel.setCCWComplianceMargin(&serialPort, armServoID, iGain);
      dynamixel.setCWAngleLimit(&serialPort, armServoID, 1);
      dynamixel.setCCWAngleLimit(&serialPort, armServoID, 4095);
    }
  else
    {
      logger->log("WARNING", "Can't open serial port %s", portName);
    }
  init_timer.stop();
}

// Timer Operation - servoTimer
void rotationController::servoTimer_operation(const NAMESPACE::TimerEvent& event)
{
  //::::/v/k/8/c/5/3::::Operation::::
  if (!paused)
    {
      int pos; // temp value to store position from servo

      armRotationCurrent = Dynamixel::posToAngle_28T(pos);

      // GRIPPER ROTATION SERVO
      dynamixel.setPosition(&serialPort, gripperRotationID, Dynamixel::angleToPos(gripper
      pos = dynamixel.getPosition(&serialPort, gripperRotationID);
      logger->log("DEBUG", "Gripper rotation servo angle: %f\n",Dynamixel::posToAngle(pos
      gripperRotationCurrent = Dynamixel::posToAngle(pos);

      // GRIPPER POSITION SERVO
      dynamixel.setPosition(&serialPort, gripperPositionID, Dynamixel::angleToPos(gripper
      pos = dynamixel.getPosition(&serialPort, gripperPositionID);
      logger->log("DEBUG", "Gripper position servo angle: %f\n",Dynamixel::posToAngle(pos
      gripperPosCurrent = Dynamixel::posToAngle(pos);
    }
  else
    {
      int pos; // temp value to store position from servo

      // ARM SERVO
      pos = dynamixel.getPosition(&serialPort, armServoID);
      armRotationCurrent = Dynamixel::posToAngle_28T(pos);

      // GRIPPER ROTATION SERVO
      pos = dynamixel.getPosition(&serialPort, gripperRotationID);
      gripperRotationCurrent = Dynamixel::posToAngle(pos);

      // GRIPPER POSITION SERVO
      pos = dynamixel.getPosition(&serialPort, gripperPositionID);
      gripperPosCurrent = Dynamixel::posToAngle(pos);
    }
}

// Subscriber Operation - controlInputsSubscriber
void rotationController::controlInputsSubscriber_operation(const high_level_controller::c
{
  //::::/v/k/8/c/5/f::::Operation::::
  paused = received_data->paused;
}
```

Figure 12. Workspace Code Generation. On the left are different views of code specific to the aspect of the model being viewed. On the right is a part of the generated source file incorporating those model elements. If users choose to input code through the GUI, they never need to view the generated files.

5.5. Documentation Generation

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.

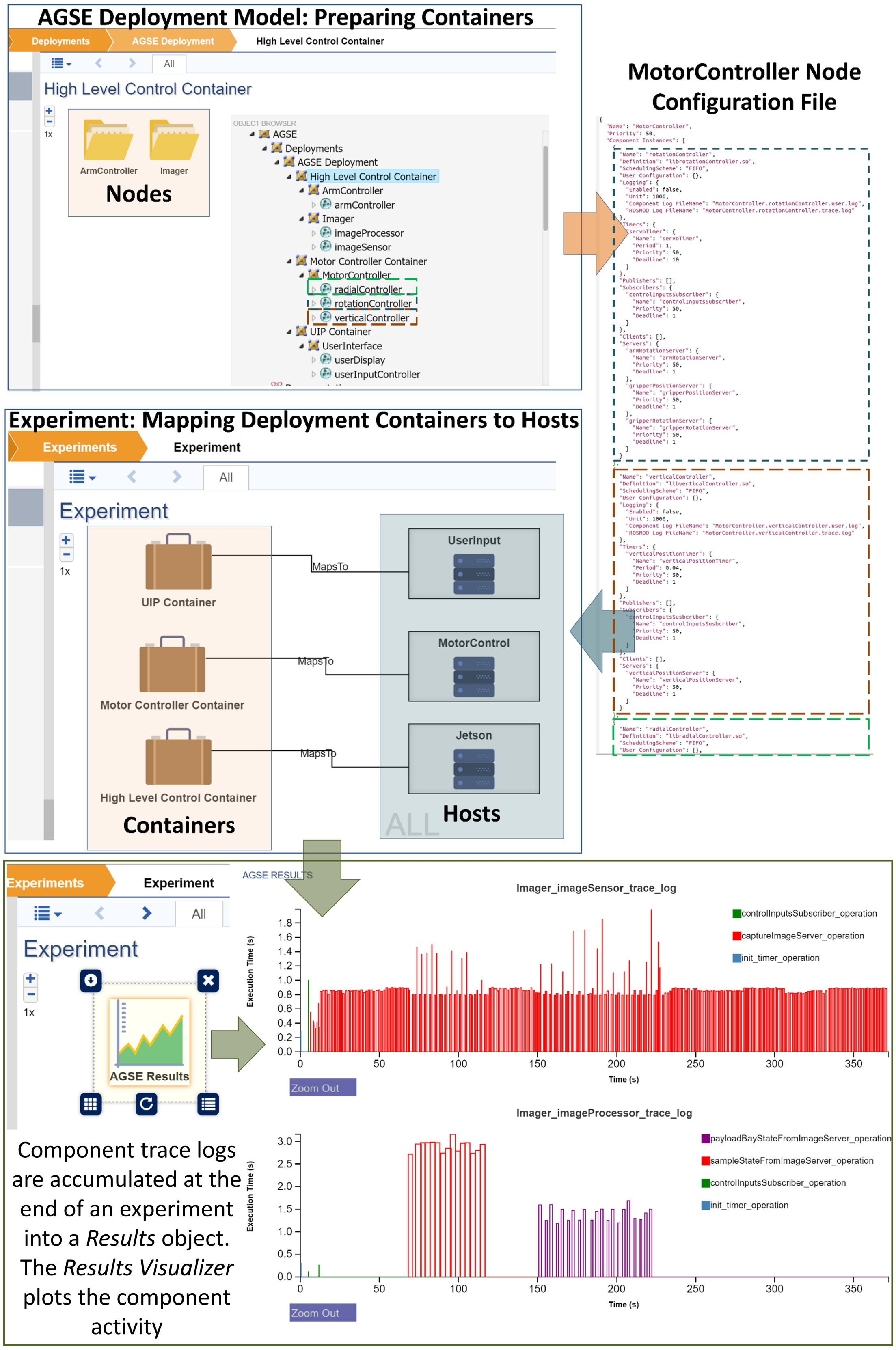
Model documentation is also supported, from which ROSMOD can generate *html* and LaTeX source files. Model documentation is displayed as rendered *markdown* in the model visualizers and can be added to any object in the model. This type of documentation is useful for archiving the design principles behind the model, and for quickly training new users both on how to use ROSMOD as well as what the model and its software+systems are designed to do.

5.6. Deployment

The workflow for software deployment is as shown Figure 13. 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 JSON files which contain meta-data about each ROS node modeled in the ROSMOD Deployment Model. This meta-data includes the component instances in each node and the appropriate component libraries to be loaded. Based on the JSON 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 JSON 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 grouping of component instances within nodes and nodes within containers 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 nodes dynamically and rapidly iterate through several scenarios and experiments without having to wait for code compilations between experiments. This capability is especially important when dealing with embedded devices for which execution of multiple nodes might incur a significant performance overhead. 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) a JSON file for each ROS node in the deployment model. For each new node added to the deployment model, by merely regenerating the JSON 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.


AGSE Deployment Model: Preparing Containers
Deployments
AGSE Deployment
High Level Control Container
All
High Level Control Container
ArmController
Imager
Nodes
OBJECT BROWSER
AGSE
Deployments
AGSE Deployment
High Level Control Container
ArmController
armController
Imager
imageProcessor
imageSensor
Motor Controller Container
MotorController
radialController
rotationController
verticalController
UIP Container
UserInterface
userDisplay
userInputController
MotorController Node Configuration File
Experiment: Mapping Deployment Containers to Hosts
Experiments
Experiment
All
Experiment
UIP Container
Motor Controller Container
High Level Control Container
Containers
MapsTo
MapsTo
MapsTo
UserInput
MotorControl
Jetson
ALL
Hosts
Experiments
Experiment
All
Experiment
AGSE Results
AGSE RESULTS
Imager_imageSensor_trace_log
Execution Time (s)
Time (s)
controlInputsSubscriber_operation
captureImageServer_operation
init_timer_operation
Zoom Out
Imager_imageProcessor_trace_log
payloadBayStateFromImageServer_operation
sampleStateFromImageServer_operation
controlInputsSubscriber_operation
init_timer_operation
Zoom Out
Component trace logs are accumulated at the end of an experiment into a *Results* object. The *Results Visualizer* plots the component activity

```
{
  "Name": "MotorController",
  "Priority": 50,
  "Component Instances": [
    {
      "Name": "rotationController",
      "Definition": "librotationController.so",
      "SchedulingScheme": "FIFO",
      "User Configuration": {},
      "Logging": {
        "Enabled": false,
        "Unit": 1000,
        "Component Log FileName": "MotorController.rotationController.user.log",
        "ROSMOD Log FileName": "MotorController.rotationController.trace.log"
      },
      "Timers": {
        "servoTimer": {
          "Name": "servoTimer",
          "Period": 1,
          "Priority": 50,
          "Deadline": 10
        }
      },
      "Publishers": [],
      "Subscribers": {
        "controlInputsSubscriber": {
          "Name": "controlInputsSubscriber",
          "Priority": 50,
          "Deadline": 1
        }
      },
      "Clients": [],
      "Servers": {
        "armRotationServer": {
          "Name": "armRotationServer",
          "Priority": 50,
          "Deadline": 1
        },
        "gripperPositionServer": {
          "Name": "gripperPositionServer",
          "Priority": 50,
          "Deadline": 1
        },
        "gripperRotationServer": {
          "Name": "gripperRotationServer",
          "Priority": 50,
          "Deadline": 1
        }
      }
    },
    {
      "Name": "verticalController",
      "Definition": "libverticalController.so",
      "SchedulingScheme": "FIFO",
      "User Configuration": {},
      "Logging": {
        "Enabled": false,
        "Unit": 1000,
        "Component Log FileName": "MotorController.verticalController.user.log",
        "ROSMOD Log FileName": "MotorController.verticalController.trace.log"
      },
      "Timers": {
        "verticalPositionTimer": {
          "Name": "verticalPositionTimer",
          "Period": 0.04,
          "Priority": 50,
          "Deadline": 1
        }
      },
      "Publishers": [],
      "Subscribers": {
        "controlInputsSusbcriber": {
          "Name": "controlInputsSusbcriber",
          "Priority": 50,
          "Deadline": 1
        }
      },
      "Clients": [],
      "Servers": {
        "verticalPositionServer": {
          "Name": "verticalPositionServer",
          "Priority": 50,
          "Deadline": 1
        }
      }
    },
    {
      "Name": "radialController",
      "Definition": "libradialController.so",
      "SchedulingScheme": "FIFO",
      "User Configuration": {},
```

Figure 13. Software Deployment. Node configuration files are generated from the deployment model and copied to the hosts on which they will run. The experiment deployment plugin automatically generates suitable container to host mappings.

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 JSON files for the deployment and copies the JSON files and the binaries over to the selected hosts, before starting the relevant processes. Figure 14 shows an example constraint-to-capability mapping established for a set of motor control components in a robot. 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.

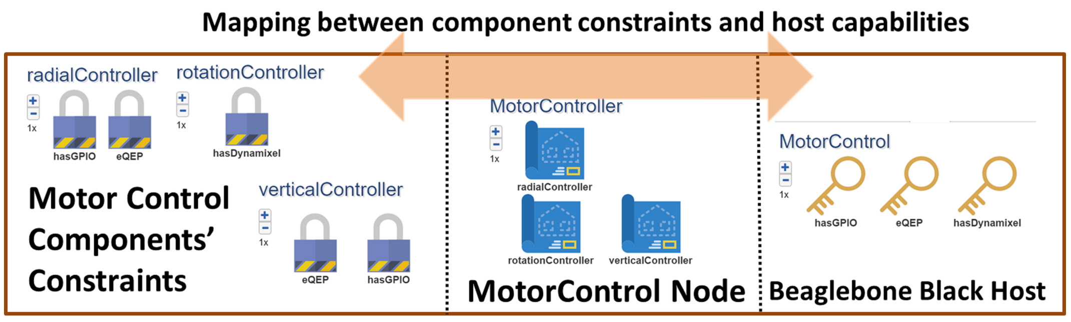


Figure 14. Mapping established between Component Constraints and Host Capabilities.

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 (where they are stored in the model and are available for download by the user), cleaning up the deployment artifacts from the hosts, and removing the saved experiment mapping from the model.

5.7. Plotter

The plotter allows the users to view the results of an experiment deployment by displaying interactive plots of the trace logs (which the components can be configured to generate) from the experiment that show the execution timings for the operations of the components in the deployment. By visualizing these logs, users can visually inspect the timing characteristics of the operations to see what the average and worst case execution times were, as well as to verify ordered execution between dependent operations. This automatic generation and plotting of execution traces from selected components in a deployment allows users to quickly visually verify that operations are being triggered according to the right stimulus, for instance verifying that a subscriber operation is triggered after a timer operation which contains a publish. Some example plots are shown at the bottom of Figure 13, above.

6. Experimental Section

6.1. 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 [68]. 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.

6.2. Competition Requirements

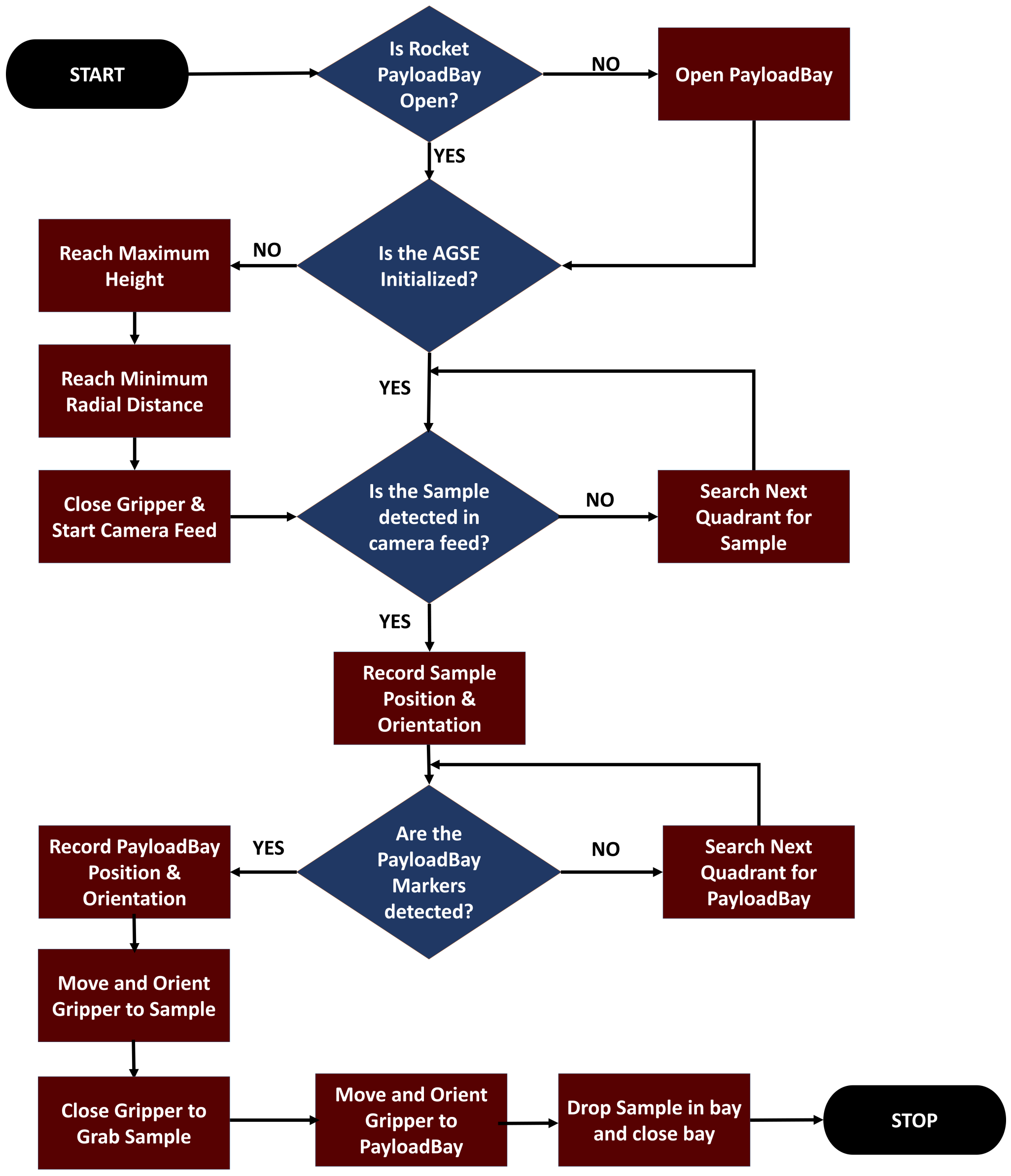
The NASA Student Launch Initiative [68] is a research-based competition partnered with NASA's Centennial Challenges, and aims to stimulate rapid, low-cost development of rocket propulsion and space exploration systems. Both collegiate and non-academic teams participate in the 8-month competition cycle 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.

6.3. 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 15.

While the driving requirements of the competition were fixed, many of the 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.



START
Is Rocket PayloadBay Open?
NO
Open PayloadBay
YES
Is the AGSE Initialized?
NO
Reach Maximum Height
Reach Minimum Radial Distance
Close Gripper & Start Camera Feed
YES
Is the Sample detected in camera feed?
NO
Search Next Quadrant for Sample
YES
Record Sample Position & Orientation
Are the PayloadBay Markers detected?
YES
Record PayloadBay Position & Orientation
NO
Search Next Quadrant for PayloadBay
Move and Orient Gripper to Sample
Close Gripper to Grab Sample
Move and Orient Gripper to PayloadBay
Drop Sample in bay and close bay
STOP

Figure 15. AGSE Control Flow Chart. This control flow is implemented as a state machine in the High-level control component, which uses messages and services to communicate goals and current state to the other lower-level control components.

6.3.1. Distributed Deployment

The AGSE robot is controlled by a distributed set of embedded controllers. Figure 16 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 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).

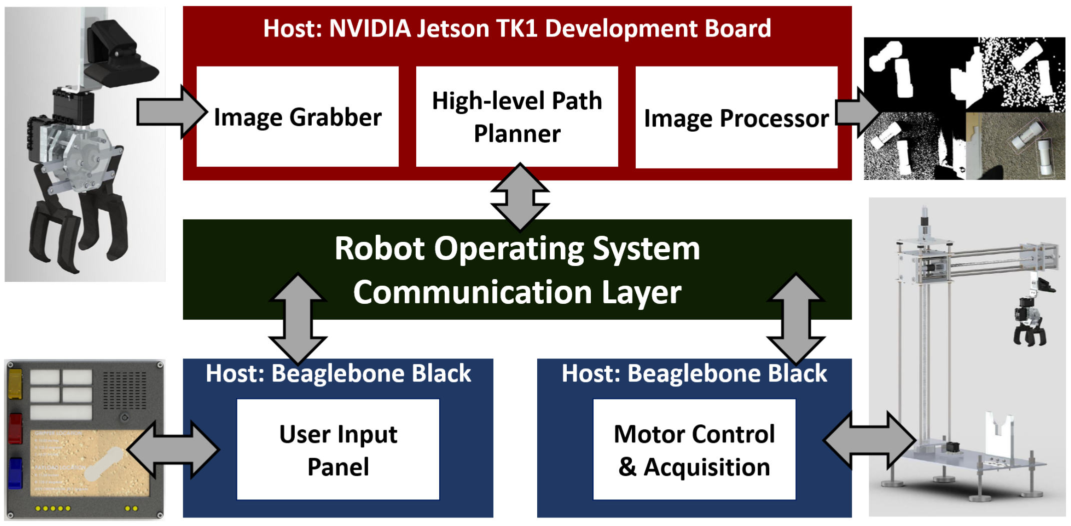


Figure 16. Autonomous Ground Support Equipment (AGSE) Package Deployment.

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 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 multiple ROS nodes with ROSMOD component executor threads periodically performing algorithmic computations, calculating new robotic paths and communicating to coordinate and maintain the AGSE state.

6.3.2. 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 17 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 16. 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.

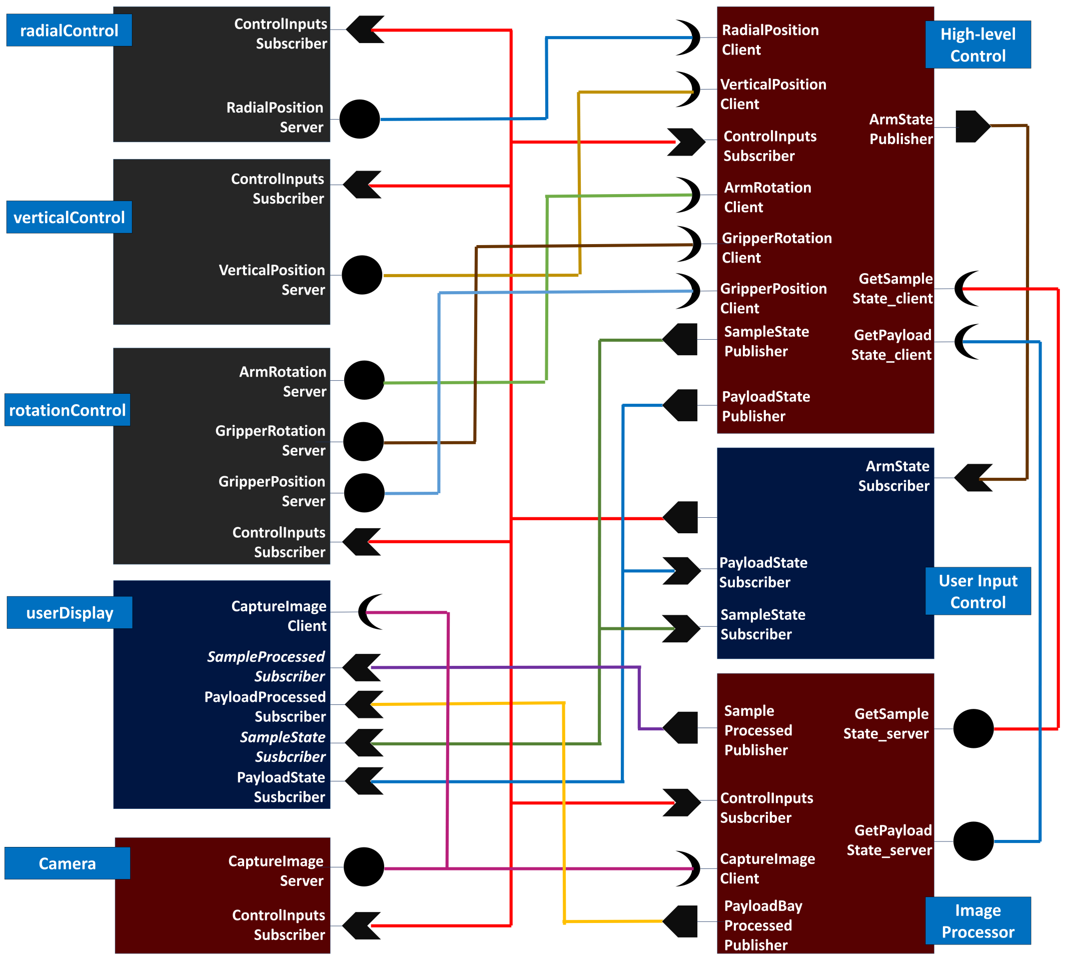


Figure 17. AGSE Component Assembly.

The ROSMOD code generators enabled generation of nearly 60% (6000+ 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 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 18.

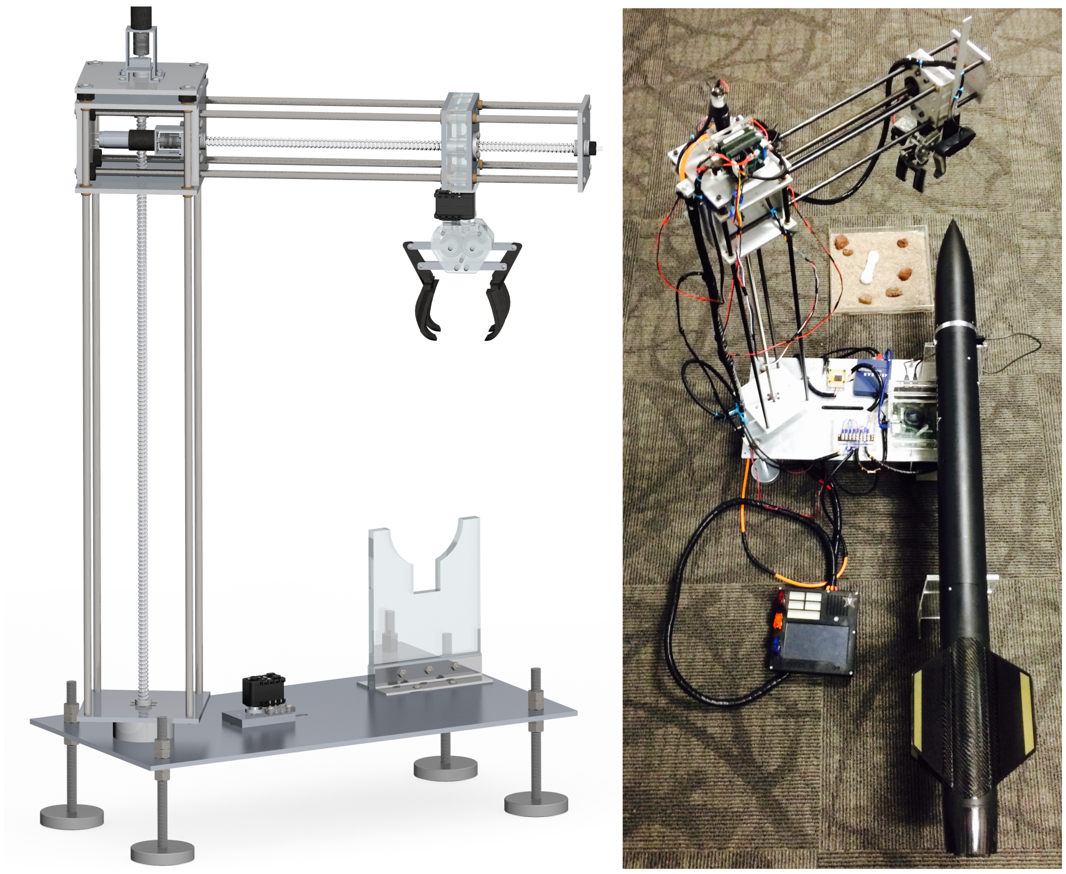


Figure 18. AGSE and rocket used in the 2014–2015 NASA SLI competition. The UIP is shown in the bottom left of the picture, the Motor Control Board is on the top of the arm of the AGSE, and the NVIDIA Jetson is under the rocket.

7. Results

The success of the AGSE given the limited number of developers, the timeline, and that the developers were also developing ROSMOD simultaneously is the principal result as evidence of the suitability of ROSMOD for rapid prototyping, development, and deployment. Over the course of 4 months, 2 software/electronics students developed the ROSMOD toolsuite software, AGSE electronics, and AGSE software from no design or implementation to a fully realized, working implementation which completed and won the NASA Student Launch Competition. The remainder of this section will be devoted to a performance assessment of the AGSE hardware and software; a more in-depth discussion on the design choices and consequences follows in Section 8.

Performance Assessment

At the competition, the Vanderbilt AGSE was able to complete the sample retrieval process in approximately 4.5 min. The recovery process, as shown in Figure 19, 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*.

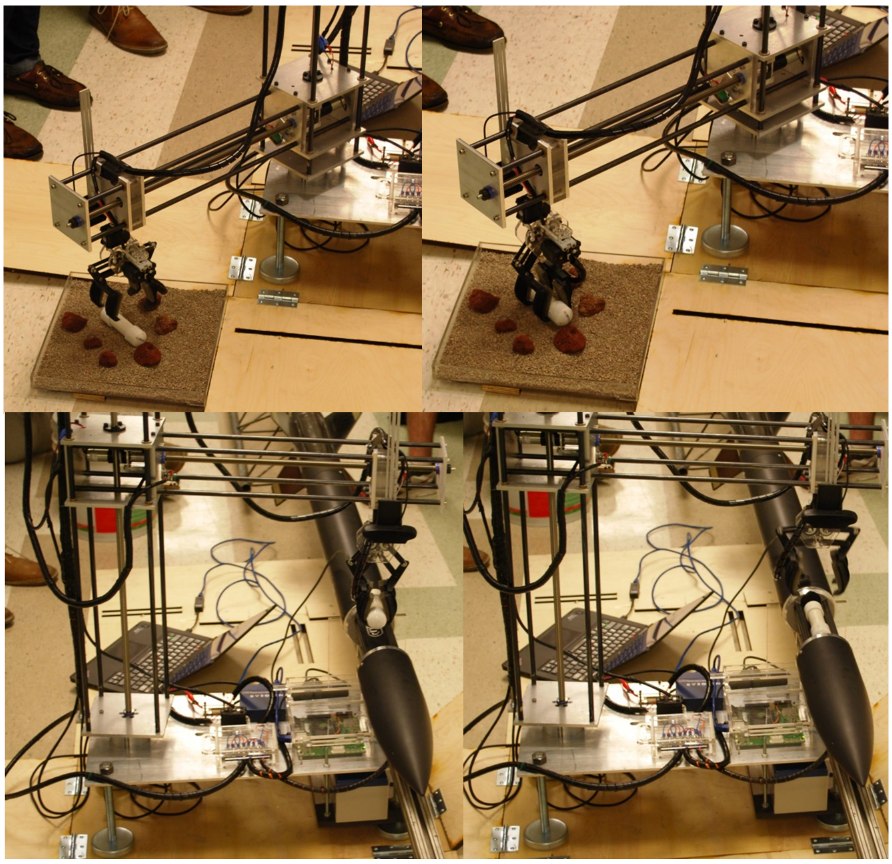


Figure 19. AGSE Calibration and Testing.

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 *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.

8. Discussion

During the development of the AGSE and ROSMOD and since then, the main takeaway for enabling rapid, iterative design, development, and experimentation of these kinds of distributed CPS has been rapid evolution of the meta model and the associated tool suite for being able to match the needs of the system and the users. Here, we discuss the various aspects of the iteration process.

8.1. Model-Integrated Code Development

Guidelines [69] for research and development in software engineering involve version control of the code into a repository, generally with feature-testing and debugging branches for experimental code. Model-driven engineering [70] has also becoming more commonplace in robotics, where models define much of the software architecture and its configuration, while developers provide the core business-logic (functional code) of the software. Maintaining a level of synchronization between models and code artifacts injected by developers is an important challenge to address in such model-driven environments. ROSMOD has chosen to incorporate the core business-logic code as attributes of objects in the model, i.e., the developer writes code, that is also attributed to some object in the model. This keeps the model both complete and up to date. Such a design has both advantages and difficulties. The inclusion of the business logic code into the model has the benefit of decreasing complexity of the software development cycle, especially as it relates to training new users on the infrastructure. In paradigms which generate a source code tree from a model and the developer must open the generated source files and finalize the proper code blocks with their business logic, the developer must learn which files and code blocks are relevant for their code; this mapping between the model objects and the generated code may change (especially if the run-time infrastructure changes) therefore the developer must be trained and proficient. By removing the need for the developer to inspect the generated source tree, we standardize the interface by which developers interact with and modify the code.

However, the transition from direct file access and modification to editing within the ROSMOD toolsuite does require developer training and may (for now) force the developer to use tools and practices with which they are not familiar or comfortable. Developers still need to learn the code blocks that are relevant for specific model objects (a mapping which they need to know regardless of the tool being used) and they must be trained on how to navigate between those code blocks within the ROSMOD toolsuite. However, ROSMOD is an extensible toolsuite with customizable visualizations for navigation and editing. Also important is the ability for developers to debug errors in their code, a task for which a dedicated IDE is the best fit. However, ROSMOD does parse build errors to determine exactly which line of which attribute of which object in the model contains the error, and points the user to that location. Additionally, modifications to deployments and their parameters and using the plotted trace logs also helps developers debug the functions, sequencing, and timing of their component code. Currently ROSMOD does not support debugging with the use of a dedicated debugger or using GDB, though that is a limitation of the current implementation, not of the infrastructure.

8.2. Extensible Meta Model

One of the main ways of managing software and system complexity when designing and developing distributed CPS is to find the component or tool which provides capabilities that align most with the needs of the project and system. In the same way, the meta-model/modeling language used to develop the software and system models should fit the needs of the project and system being developed. To enable such a matching between the needs of the project and the capabilities of the tool, ROSMOD has been developed and designed to facilitate easily customizing the meta model from which the models are constructed. As an example consider the AGSE: for this specific project, a complex communications network meta-model is not needed (esp. when quickly prototyping) and would only serve to complicate the model and development cycle. However, a tool that only supports such a simple communications network meta-model restricts other projects which may require more detailed modeling for analysis or deployment purposes. To manage such design complexity, ROSMOD enables the customization of the meta-model (and associated automation infrastructure) to suit the specific application or domain for which it is being used.

Such a design decision can front-load the design process with more work involved setting up and configuring the meta-model and the associated infrastructure. This extra complication can start to be addressed by using the automatic installation of other WebGME components into ROSMOD. The goal with such an infrastructure is for users to install open-source meta-models and infrastructural tools that closely match their intended domain and perform minor customizations to the installed meta-models and tools. Such a design and development process is actively being researched.

8.3. Extensible Infrastructure

Just as the meta-model should be extensible and customizable to the specific domain for which ROSMOD is being used, so too should the infrastructure facilitate extension, modification, and replacement if necessary. Even once the meta-model has been finalized, there may be a need to further customize the automation tools such as compilation or deployment for the specific system or systems which are being managed. As a simple example, take the remote procedure utilities used by ROSMOD's current compilation and deployment infrastructure. These utilities require the use of authenticated secure shell (SSH) connections with support for specific shells on the remote ends. There exist systems for which ROSMOD would be beneficial that do not support such remote access and execution tools. In such a case, since the remote execution and monitoring facilities supported by ROSMOD are modularized and replaceable, the developers managing such systems can relatively easily swap out these execution procedures with the procedures supported by the target systems. Similarly, developers may have their own cross-compilers for different architectures they are targeting; extending the current tool suite to support these other cross-compilers merely requires extending or swapping out the existing WebGME components (which are Node.js classes), possibly even using the existing software compilation code as a base.

Once again, such extensibility comes with a cost of training and complexity with respect to easily integrating with the existing code-base, learning the interfaces, and developing replacement code which provides the same or new desired functionality.

8.4. Service-Based Infrastructure

Another aspect ROSMOD addresses for rapid system design and development is the setup, configuration, and training required for new developers to use the tool suite. With traditional tool infrastructures, installation of the tool suite, its dependencies, and any associated (if needed) compilation and deployment infrastructures is required on every developer's computer (which may be further restricted to a specific version of a specific operating system). The configuration and training on those tools and infrastructures may be lengthy and complex, and may further reduce the number of available developers who have the capacity to fully integrate with such an infrastructure. By developing the ROSMOD tool suite and infrastructure in a service-oriented fashion, where ROSMOD's development and deployment tools are provided as a service to the developers, no installation or configuration process is required once the ROSMOD server has been set up and configured. Clearly the setup and configuration of the server requires the installation and configuration of the required dependencies (many of which are automatically managed by *npm*), but the managing of those tools and their versions can be handled in a centralized fashion by a very small subset of the overall developer base. Since there is no such installation or configuration for the majority of the developers, they only have to learn the minimum required skills for developing the models, writing the code, and running experiments and deployments on the available systems. Again, such training is expedited by in-place documentation which can walk the developers through the different parts of the tool and how to use it. From a management perspective, this centralization of the tool suite and provision as a service enables tighter control for maintaining and updating relevant dependent tools and infrastructure (such as cross-compilers or available systems).

8.5. Collaborative Infrastructure

By using a centralized service-based toolsuite that is backed by a versioned database, ROSMOD can enable rapid collaboration between developers, allowing them to easily view and edit shared projects simultaneously yet still enable them to operate independently on their own private projects. By using this centralized, automatically versioned database, developers' work is automatically versioned without them having to integrate versioning tools into their workflow. Again, by enabling these integrations automatically, we decrease the amount of training required for developers while still ensuring proper development practices and versioning of all work.

Of course, this direct integration is not without its drawbacks, as such tight integration makes more complex or fine-grained versioning and repository management harder, since currently the versioning subsystem of the infrastructure cannot be easily extended or swapped out should other projects have very stringent requirements on the tools and methods they use for versioning.

8.6. Structured Programming of Nodes

One of the main design decisions we made early on with the ROSMOD component model and execution semantics was to structure the programming of ROS nodes by executing components within nodes. One of the main driving forces behind this decision was the thread overhead ROS nodes incur for communication to other ROS nodes. The communicating threads created by a ROS node are shared between all components on the node, so collocation of components within a node decreases the overhead per component. Additionally, we provide programming guidelines for components which advise programmers to not create threads within component code unnecessarily. By imposing this programming practice, developers communicate shared data using the component interfaces and can rely on the component execution semantics to ensure that they do not have to write data locking code. Additionally, by compiling component code into libraries that are loaded by the actor executable based on the deployment configuration, users can run experiments to determine how the trade-off between collocation of components within nodes and distributed among multiple nodes affects the timing properties of their components (since context switching between processes incurs a significant overhead on embedded devices).

8.7. Software Complexities and Component Interactions

Software components exist as a way for developers to segment the design and development problems into a manageable form where only certain related concerns are grouped together into a component. These groupings of concerns into component functionality provide a mechanism for iteratively and systematically developing larger integrated applications from a suite of software components which provide specific functionalities and interfaces. One of the primary benefits to this practice is component re-use. If a component's functionality and interface are defined well enough (meaning both specific to the functions required and general enough to be used in multiple systems), then by defining and developing the component once, developers can simply re-use that same component when the same or similar functionality is needed again. In this way, the later developers who are re-using the component can treat it as if it were a *black-box*, for which they do not have to be concerned with what is inside the box, i.e., how the functionality is implemented. However, since these are interacting components, such a view is ultimately incomplete. Clearly how often the component produces output interactions (for other components) and how quickly it responds to input interactions from other components is important to overall system behavior. Furthermore, this behavior of the component is dependent not just on the implementation (i.e., the software) of the component, but also the hardware on which the component runs (or to which it may be connected) and the other software components with which the component is communicating. To combat the complexities of these integration and reusability challenges, the ROSMOD component model has been developed from the start to be analyzable with respect to timing and resource requirements [21–26]. In this way, as the developer configures the properties of the component, e.g., periodicity of its timers, the composed system in which that component interacts with other components can be analyzed. Of course this analysis must (and does) take into account various aspects of how the components' operations are scheduled (both within the components and on the processor) as well as the behavior of the components. However, by developing the component model in such a way that it is amenable to these analysis techniques, we enable developers to meaningfully handle the complexities arising from the composition of these components and the re-use of these components.

8.8. ROSMOD and AGSE Design Iterations and Lessons

The goal of ROSMOD is to be a model-driven, component-based rapid development, deployment, and experimentation tool 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 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 execution 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 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 20 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 17, 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 20, the high-level controller has two set of spikes in execution time. The first set (between 100 and 150 s into the experiment), as confirmed by Figure 21 corresponds to sample detection. The second set of spikes, between 200 and 250 s, 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.

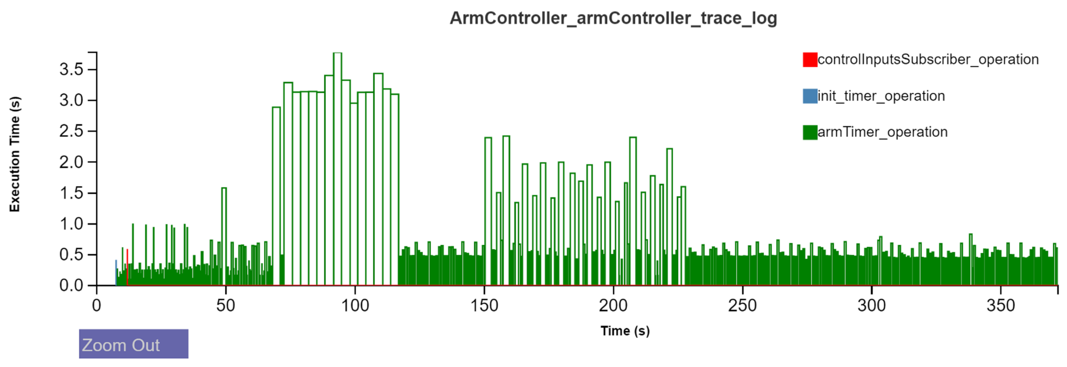


Figure 20. High-level Timer Execution.

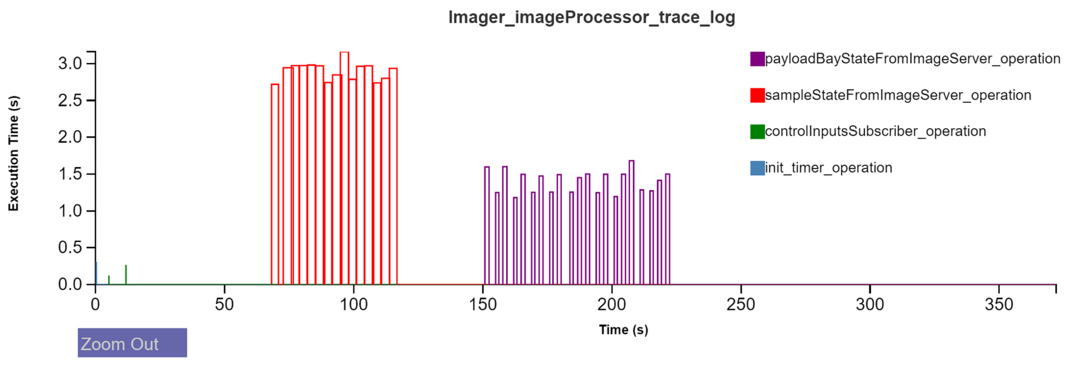


Figure 21. Sample and Payload bay Detectors Execution.

Figure 21 presents the performance of the *GetSampleState_Server* and the *GetPayloadState_Server* in Figure 17. 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 Camera component for a feed update. The Camera *ImageServer*, as shown in Figure 22 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.

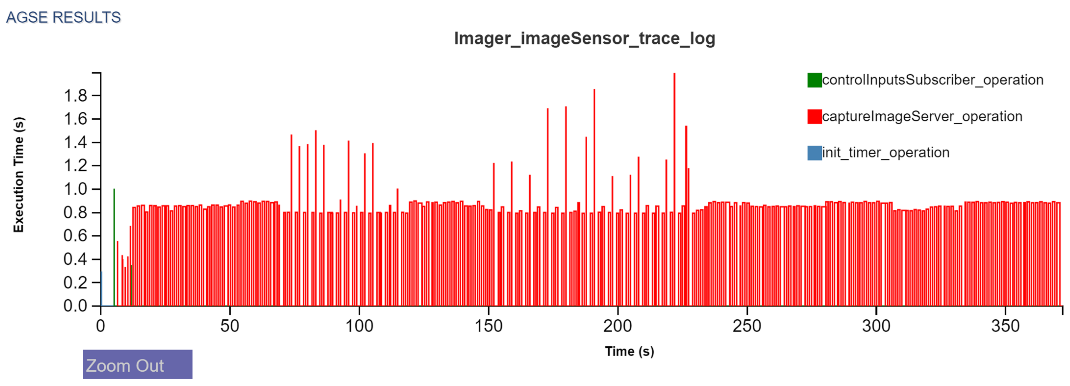


Figure 22. Camera Feed Reponse Times.

We 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.

9. 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. The specific mechanical and software components (as well as from where they were sourced and when) can be found at the end of this section in Tables 2 and 3.

Table 2. Mechanical Parts Used.

Item	Vendor	City	State	Contry
Low-Carbon Steel Rod	McMaster-Carr	Douglasville	GA	USA
Nylon Flanged Bearing	McMaster-Carr	Douglasville	GA	USA
Mini hi Precision SS Ball Bearing	McMaster-Carr	Douglasville	GA	USA
Steel Thrust Ball Bearing Steel Washers	McMaster-Carr	Douglasville	GA	USA
Black-Oxide Alloy Steel Socket Head Cap Screw 1/4"-28; 1/2"	McMaster-Carr	Douglasville	GA	USA
Flat Point Socket Set Screw Type 316 Stainless Steel, 8-32, 3/16"	McMaster-Carr	Douglasville	GA	USA
Black-Oxide Socket Head Cap Screw Alloy Steel, M2, 6mm	McMaster-Carr	Douglasville	GA	USA
Clamp-on Helical Flexible Shaft Coupling Aluminum, 3mm Dia	McMaster-Carr	Douglasville	GA	USA
Corrosion Resistant Dowel Pin , 3/8"; 1"	McMaster-Carr	Douglasville	GA	USA
Aluminum U-Bolt 5/16"-18 for 2-1/2" Pipe	McMaster-Carr	Douglasville	GA	USA
Steel Ball Bearing Dbl Sealed, NO. R2 for 1/8" Shaft Dia, 3/8" OD	McMaster-Carr	Douglasville	GA	USA
Low-Carbon Steel Rod 3/8" Diameter	McMaster-Carr	Douglasville	GA	USA
Black Oxide Steel Drive Shafts (3/8" OD, 36" length)	McMaster-Carr	Douglasville	GA	USA
3mm to 5mm Helical Shaft Coupling	McMaster-Carr	Douglasville	GA	USA
4mm to 6mm Helical Shaft Coupling	McMaster-Carr	Douglasville	GA	USA
Lead Screw Radial Bearings (1/4" ID, 1/2" OD)	McMaster-Carr	Douglasville	GA	USA
Turntable Thrust Bearing- Cage Assembly	McMaster-Carr	Douglasville	GA	USA
Turntable Radial Bearings	McMaster-Carr	Douglasville	GA	USA
Lead Screw Thrust Bearings (3/8" ID, 13/16 OD")	McMaster-Carr	Douglasville	GA	USA
Multipurpose 6061 Aluminum L-bracket (4ft)	McMaster-Carr	Douglasville	GA	USA
Bolt (10-24 1-1/4" Length) (pack of 50)	McMaster-Carr	Douglasville	GA	USA
Nut (10-24, Locknut, Low Profile) (pack of 100)	McMaster-Carr	Douglasville	GA	USA
Bolt (8-32 1/2" Length) (pack of 25)	McMaster-Carr	Douglasville	GA	USA
Bolt (M2-0.4 8mm Length) (pack of 25)	McMaster-Carr	Douglasville	GA	USA
Bronze Bushings (3/8" ID, 1/2" OD) w/ Graphite	McMaster-Carr	Douglasville	GA	USA
Turntable Thrust Bearing- Washer	McMaster-Carr	Douglasville	GA	USA
Teflon Washers	McMaster-Carr	Douglasville	GA	USA
4-40 Bolts (1/2 in length)	McMaster-Carr	Douglasville	GA	USA
4-40 Bolts (1-3/4 in length)	McMaster-Carr	Douglasville	GA	USA
3/4in Standoff for 4-40 bolt	McMaster-Carr	Douglasville	GA	USA
4-40 Nut (low profile)	McMaster-Carr	Douglasville	GA	USA
4-40 tooth lock washer	McMaster-Carr	Douglasville	GA	USA
Faulhaber 2237S012CXR Motors	Micromo	Clearwater	FL	USA
3/4" 6061 Aluminum Plate	Loftis Steel	Nashville	TN	USA
1/4" 6061 Aluminum Plate	Loftis Steel	Nashville	TN	USA
3/4" 3′x3′ Acrylic Sheet	Plastic Supply	Nashville	TN	USA
Lead Screw - 1/2"-10, 36" length	Haydon Kerk	Waterbury	CT	USA
Lead Screw - 1/2"-10, 24" length	Haydon Kerk	Waterbury	CT	USA
Lead Nut - 1/2"-10	Haydon Kerk	Waterbury	CT	USA
Dynamixel AX-12A Servo	Trossen Robotics	Downers Grove	IL	USA
Dynamixel MX-28T Servo	Trossen Robotics	Downers Grove	IL	USA
Linear Actuator Mounting Bracket	Trossen Robotics	Downers Grove	IL	USA
LED - RGB Clear Common Cathode	Sparkfun	Niwot	CO	USA
LED - Super Bright Red	Sparkfun	Niwot	CO	USA
LED - Super Bright Blue	Sparkfun	Niwot	CO	USA
LED - Super Bright Yellow	Sparkfun	Niwot	CO	USA
Encoder Optical Gap 3CH 500CPR	Digikey	Thief River Falls	MN	USA
Codewheel 3CH 500CPR 1/4"	Digikey	Thief River Falls	MN	USA
7" LCD DispCapew/tch	Mouser Electronics	Mansfield	TX	USA
Jetson TK1	NVIDIA	Santa Clara	CA	USA
Beaglebone Black	Adafruit	New York	NY	USA

Table 3. Software Libraries Used.

Item	Version	Company	City	State	Contry	Year
ROS	Indigo	Willow Garage	Menlo Park	CA	USA	2015
WebGME	2.3.1	Vanderbilt University	Nashville	TN	USA	2016
Node.js	4.5	Joyent	San Francisco	CA	USA	2016
webgme-rosmod	1.0	Vanderbilt University	Nashville	TN	USA	2016
rosmod	0.3-beta	Vanderbilt University	Nashville	TN	USA	2016
Aruco	1.2.5	University of Cordoba	Cordoba	Cordoba	Argentina	2015
OpenCV	2.4	Itseez	San Francisco	CA	USA	2015
Jetson OS	R19 - Ubuntu	NVidia	Santa Clara	CA	USA	2015
BBB OS	Debian Jessie	Texas Instruments	Dallas	TX	USA	2015

9.1. 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 [71]. 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 [72].

9.1.1. 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 23.

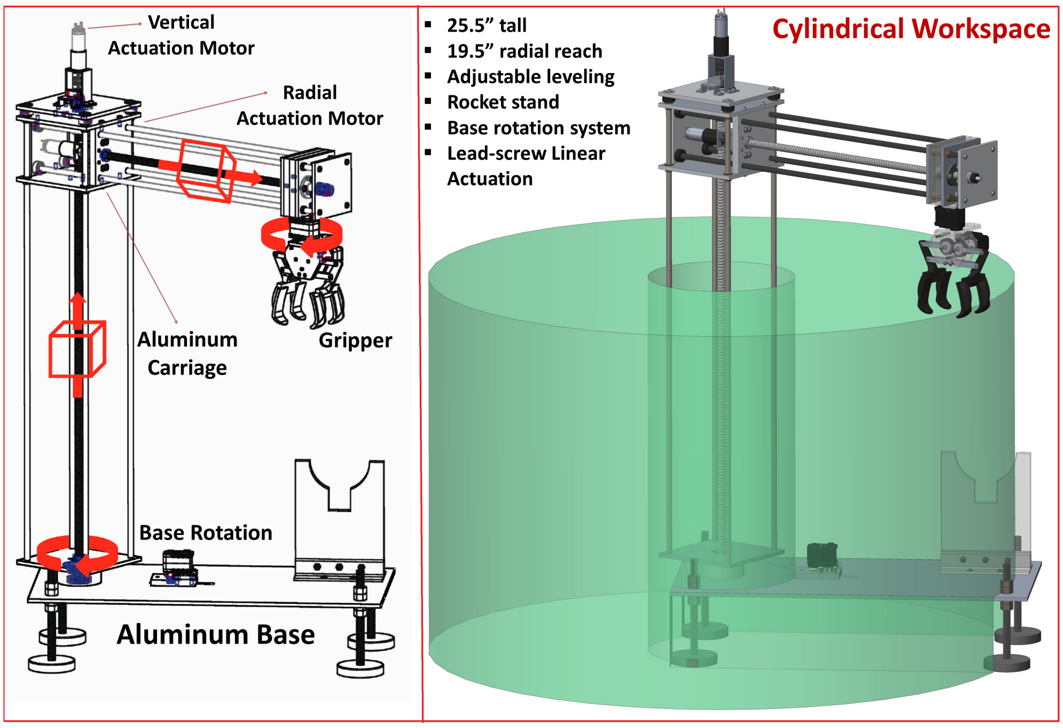


Figure 23. AGSE Mechanical Design.

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.

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

9.1.2. 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.

9.1.3. 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.

9.1.4. Power System

The power from the AGSE comes from two 12 V 12 Ah lead-acid batteries in series, which provide input power to a 12 V power regulator. The 12 V 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 12 V 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 12 V input drives (1) the cape's 5 V regulator which provides power to the BeagleBone Black and the encoders; (2) the 12 V Dynamixel servo motors; and (3) the H-Bridges for the two 12 V linear actuators.

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

9.1.5. 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 1920 × 1080 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.

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 24, we draw contours around the object and identify its relative position and orientation.

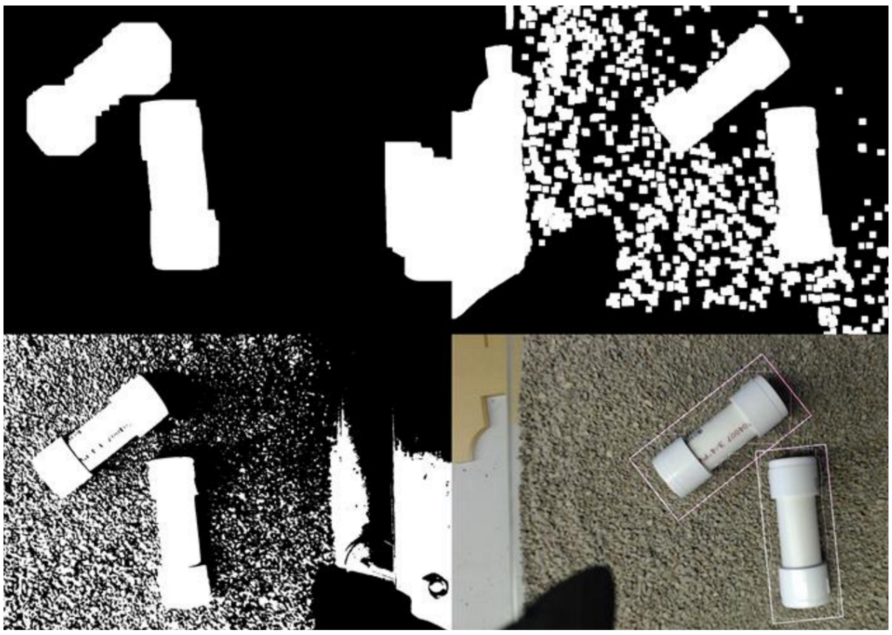


Figure 24. AGSE Sample Detection - Periodic Image Processing.

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 25, 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.

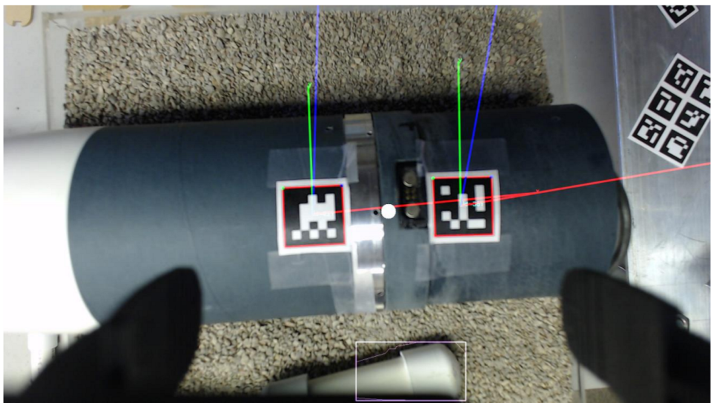


Figure 25. AGSE Payload Bay Detection.

9.1.6. 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 3.3 V serial port on the GPIO pins of the BeagleBone Black. Because the servo motors communicate at 5 V 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 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.

9.2. 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 20 mm $\times$ 20 mm 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.

9.3. 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 [73].

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-source online [74], 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.

9.4. 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 [75], and the current version of ROSMOD-COM [74].

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 in a Linux environment and can be locally deployed on a laptop, virtual machine, or in the cloud. The change to this new infrastructure was enabled by the switch to using WebGME [29] as the back-end platform for the modeling and server execution. This change also enables collaborative model creation and editing between multiple users, with built-in Git-style version control, branching, and merging.

Despite the increase in back-end complexity and number of features of ROSMOD, the use of ROSMOD and the installation of the back-end has become far more streamlined and completely automated by widely-used packaging tools. The repository provides installation instructions and frequently updated users guides.

10. Conclusions

In this paper we have presented a new technique and toolsuite for rapidly prototyping systems. This toolsuite is provided as a service to the developers to minimize the amount of time and effort they spend installing, configuring, and maintaining the toolsuite and its dependencies. This service-based toolsuite also allows for a greater degree of platform independence without adding the cost of additional overhead as with traditional forms of platform independence. Under this paradigm, the required development infrastructure is managed in a centralized fashion, with the development tools being provided as a service to the developers through a web-based platform. The design philosophy of an easily, rapidly changable meta-model/meta-language coupled together with reconfigurable and extendable infrastructure was presented and realized with a fully-functional toolsuite. A rapidly prototyped and evolved meta-model for distributed robotics systems experimentation was presented as our domain use-case for this toolsuite and an example NASA award-winning autonomous robotics system was presented which was developed following this philosophy and using this toolsuite. Finally, we discussed the tradeoffs of desiging the toolsuite in this way and presented the required fully open-source materials and methods needed to fully replicate our work, and any who wish may go to those sources and use our (evolving) web-based deployment of this toolsuite freely.

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 2014–2015 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. Robot by Edward Boatman from the Noun Project.

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.

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
GUI	Graphical User Interface
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
UML	Unified Modeling Language
USB	Universal Serial Bus
XML	Extensible Markup Language

References

1. Echeverria, G.; Lassabe, N.; Degroote, A.; Lemaignan, S. Modular open robots simulation engine: Morse. In Proceedings of the 2011 IEEE International Conference on Robotics and Automation (ICRA), Shanghai, China, 9–13 May 2011; pp. 46–51.
2. Cook, G.E.; Biegl, C.; Springfield, J.F.; Fernandez, K.R. An intelligent robotics simulator. In Proceedings of the Conference Record of the 1994 IEEE Industry Applications Society Annual Meeting, Denver, CO, USA, 2–6 Octorber 1994; pp. 1793–1800.
3. Hamblen, J.O.; van Bekkum, G.M. An embedded systems laboratory to support rapid prototyping of robotics and the internet of things. *IEEE Trans. Educ.* **2013**, *56*, 121–128.
4. Bräunl, T. *Embedded Robotics: Mobile Robot Design and Applications with Embedded Systems*; Springer Science & Business Media: New York, NY, USA, 2008.
5. Vogt, P. Language evolution and robotics: Issues on symbol grounding. In *Artificial Cognition Systems*; Hershey, PA, USA, 2006; pp. 176–209.
6. LaLonde, W.; Thomas, D.A.; Johnson, K. Smalltalk as a programming language for robotics? In Proceedings of the 1987 IEEE International Conference on Robotics and Automation, Raleigh, NC, USA, 31 March–3 April 1987; pp. 1456–1462.
7. Frenger, P. Forth as a robotics language: Part two. *ACM Sigplan Not.* **1997**, *32*, 19–22.
8. Husty, M.L. An algorithm for solving the direct kinematics of general Stewart-Gough platforms. *Mech. Mach. Theory* **1996**, *31*, 365–379.
9. Lin, M.C.; Canny, J.F. A fast algorithm for incremental distance calculation. In Proceedings of the 1991 IEEE International Conference on Robotics and Automation, Sacramento, CA, USA, 9–11 April 1991; pp. 1008–1014.
10. Krishnanand, K.; Ghose, D. Glowworm swarm based optimization algorithm for multimodal functions with collective robotics applications. *Multiagent Grid Syst.* **2006**, *2*, 209–222.
11. Campion, G.; Bastin, G.; Dandrea-Novel, B. Structural properties and classification of kinematic and dynamic models of wheeled mobile robots. *IEEE Trans. Robot. Autom.* **1996**, doi:10.1109/70.481750.
12. Koenig, N.; Howard, A. Design and use paradigms for gazebo, an open-source multi-robot simulator. In Proceedings of the 2004 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS 2004), 28 September–2 Octorber 2004; Volume 3, pp. 2149–2154.
13. Nakaoka, S.; Nakazawa, A.; Kanehiro, F.; Kaneko, K.; Morisawa, M.; Hirukawa, H.; Ikeuchi, K. Learning from observation paradigm: Leg task models for enabling a biped humanoid robot to imitate human dances. *Int. J. Robot. Res.* **2007**, *26*, 829–844.
14. Sciavicco, L.; Siciliano, B. *Modelling and Control of Robot Manipulators*; Springer Science & Business Media: New York, NY, USA, 2012.
15. Beydeda, S.; Book, M.; Gruhn, V. *Model-Driven Software Development*; Gruhn, V., Ed.; Springer: Heidelberg, Germany, 2005; Vol. 15.
16. Heineman, G.T.; Councill, W.T. Component-based software engineering. In *Putting the Pieces Together*; Addison-Westley: Boston, MA, USA, 2001.
17. Clemens, S.; Dominik, G.; Stephan, M. *Component Software: Beyond Object-Oriented Programming*; Addison-Westley: Boston, MA, USA, 1998.
18. Simulink. Available online: http://www.mathworks.com/products/simulink/ (accessed on 31 August 2016).
19. Autosar GbR. AUTomotive Open System ARchitecture. Available online: http://www.autosar.org/ (accessed on 31 August 2016).
20. Bell, D. *UML Basics: An Introduction to The Unified Modeling Language*; IBM Global Services, Rational Software: North Castle, NY, USA, 2003.
21. Kumar, P.S.; Dubey, A.; Karsai, G. Colored Petri Net-based Modeling and Formal Analysis of Component-based Applications. In Proceedings of the 11th Workshop on Model Driven Engineering, Verification and Validation MoDeVVa 2014, Valencia, Spain, 30 September 2014; p. 79.

22. Kumar, P.S.; Dubey, A.; Karsai, G. Integrated Analysis of Temporal Behavior of Component-based Distributed Real-time Embedded Systems. In Proceedings of the 2015 IEEE International Symposium on Object/Component/Service-Oriented Real-Time Distributed Computing Workshops (ISORCW), Auckland, New Zealand, 13–17 April 2015.
23. Kumar, P.; Karsai, G. Experimental Validation of Timing Analysis for Component-based Distributed Real-time Embedded Systems. In Proceedings of the 2016 IEEE International Symposium on Real-time Computing Object/Component/Service-Oriented Real-Time Distributed Computing Workshops (ISORCW), York, UK, 17–20 May 2016.
24. Emfinger, W.; Karsai, G.; Dubey, A.; Gokhale, A. 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, Berlin, Germany, 14–17 April 2014; ACM: New York, NY, USA, 2014; pp. 44–47.
25. Emfinger, W.; Karsai, G. Modeling Network Medium Access Protocols for Network Quality of Service Analysis. In Proceedings of the 18th IEEE Symposium on Real-Time Computing (ISORC), Auckland, New Zealand, 13–17 April 2015; pp. 44–47.
26. Emfinger, W.; Karsai, G. Analysis of Routed Networks with Time-Varying Delays and Capacities. In Proceedings of the 2016 IEEE International Symposium on Real-time Computing (ISORC), Object/Component/Service-Oriented Real-Time Distributed Computing Workshops (ISORCW), York, UK, 17–20 May 2016.
27. Kumar, P.; Emfinger, W.; Kulkarni, A.; Karsai, G.; Watkins, D.; Gasser, B.; Ridgewell, C.; Anilkumar, A. ROSMOD: A Toolsuite for Modeling, Generating, Deploying, and Managing Distributed Real-Time Component-Based Software Using ROS. In Proceedings of the 2015 International Symposium on Rapid System Prototyping, Amsterdam, Netherlands, 8–9 October 2015.
28. Rappin, N.; Dunn, R. *wxPython in Action*; Manning Publications, 2006. Available online: http://manning-content.s3.amazonaws.com/download/3/a958da7-d76c-4c22-b3d6-69ebd4168a21/rappin_ch09.pdf (accessed on 31 August 2016).
29. 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. In Proceedings of the 8th Multi-Paradigm Modeling Workshop, Valencia, Spain, 30 September 2014; pp. 41–60.
30. IEEE International Conference on Simulation, Modeling, and Programming for Autonomous Robots. Available online: http://www.simpar2016.org (accessed on 31 August 2016).
31. Domain-Specific Languages and Models for Robotic Systems. Available online: http://www.doesnotunderstand.org/public/DSLRob2015.html (accessed on 31 August 2016).
32. Journal of Software Engineering for Robotics. Available online: http://www.joser.org (accessed on 31 August 2016).
33. Wang, N.; Schmidt, D.C.; O'Ryan, C. Overview of the corba component model. In *Component-Based Software Engineering*; Addison-Wesley Longman Publishing Co., Inc.: Boston, MA, USA, 2001; pp. 557–571.
34. CORBA, O.; Specification, I. *Object Management Group*; Joint Revised Submission OMG Document Orbos/99-02; Available online: http://www.omg.org/spec/CORBA/ (accessed on 31 August 2016).
35. Nestor, J.R.; Wulf, W.A.; Lamb, D.A. *IDL-Interface Description Language: Formal Description*; Department of Computer Science, Carnegie-Mellon University: Pittsburgh, PA, USA, 1981.
36. Unified Component Model. Available online: http://www.omgwiki.org/ucm/doku.php (accessed on 31 August 2016).
37. Schlegel, C.; Wörz, R. The software framework SMARTSOFT for implementing sensorimotor systems. In Proceedings of the 1999 IEEE/RSJ International Conference on Intelligent Robots and Systems, Kyongju, Korea, 17–21 October 1999; Volume 3, pp. 1610–1616.
38. ACE Based SmartSoft. Available online: http://www.servicerobotik-ulm.de/drupal/?q=node/31 (accessed on 31 August 2016).
39. ACE Based SmartSoft Source Code. Available online: https://sourceforge.net/projects/smartsoft-ace/ (accessed on 31 August 2016).
40. Schmidt, D.C.; Natarajan, B.; Gokhale, A.; Wang, N.; Gill, C. TAO: A pattern-oriented object request broker for distributed real-time and embedded systems. *IEEE Distrib. Syst. Online* **2002**, *3*, 1–20.

41. Schmidt, D.C. *The ADAPTIVE Communication Environment: An Object-Oriented Network Programming Toolkit for Developing Communication Software*; Department of Computer Science, Washington University: St. Louis, MO, USA, 1993.
42. Brooks, A.; Kaupp, T.; Makarenko, A.; Williams, S.; Orebäck, A. Orca: A component model and repository. In *Software Engineering for Experimental Robotics*; Springer: Heidelberg, Germany, 2007; pp. 231–251.
43. Feldmann, B. The Internet communication environment (ICE)-virtual university as virtual community. In Proceedings of the 2002 International Conference on Computers in Education, Auckland, New Zealand, 3–6 December 2002; Volume 2, pp. 1510–1511.
44. Bruyninckx, H. Open robot control software: the OROCOS project. In Proceedings of the 2001 ICRA IEEE International Conference on Robotics and Automation, Seoul, South Korea, 21–26 May 2001; Volume 3, pp. 2523–2528.
45. Otte, W.; 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. In Proceedings of the 2013 IEEE 16th International Symposium on Object/Component/Service-Oriented Real-Time Distributed Computing (ISORC), Paderborn, Germany, 19–21 June 2013; pp. 1–8.
46. 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. In Proceedings of the ICRA Workshop on Open Source Software, Kobe, Japan, 12–17 May 2009.
47. DARPA Robotics Challenge. Available online: http://www.theroboticschallenge.org// (accessed on 31 August 2016).
48. ROS 2.0 Design. Available online: http://design.ros2.org/ (accessed on 31 August 2016).
49. Hintjens, P. *ZeroMQ: Messaging for Many Applications*; O'Reilly Media, Inc.: Sebastopol, CA, USA, 2013.
50. Innovations, R.T. RTI Data Distribution Service. Available online: http://www.rti.com/products/dds/index.html (accessed on 31 August 2016).
51. Bischoff, R.; Guhl, T.; Prassler, E.; Nowak, W.; Kraetzschmar, G.; Bruyninckx, H.; Soetens, P.; Haegele, M.; Pott, A.; Breedveld, P.; et al. BRICS-Best practice in robotics. In Proceedings of the Robotics (ISR), 2010 41st International Symposium on and 2010 6th German Conference on Robotics (ROBOTIK), Munich, Germany, 7–9 June 2010; pp. 1–8.
52. Bruyninckx, H.; Klotzbücher, M.; Hochgeschwender, N.; Kraetzschmar, G.; Gherardi, L.; Brugali, D. The BRICS component model: A model-based development paradigm for complex robotics software systems. In Proceedings of the 28th Annual ACM Symposium on Applied Computing, Coimbra, Portugal, 18–22 March 2013; pp. 1758–1764.
53. Bubeck, A.; Weisshardt, F.; Verl, A. BRIDE-A toolchain for framework-independent development of industrial service robot applications. In Proceedings of the 41st International Symposium on Robotics ISR/Robotik 2014, Munich, Germany, 2–3 June 2014; pp. 1–6.
54. Eclipse, I. The Eclipse Foundation. Available online: https://eclipse.org/downloads/ (accessed on 31 August 2016)
55. Robotics System Toolbox. Available online: http://www.mathworks.com/help/robotics/index.html (accessed on 31 August 2016).
56. Jang, C.; Song, B.; Jung, S.; Kim, S. A Heterogeneous Coupling Scheme of OPRoS Component Framework with ROS. In Proceedings of the 2012 9th International Conference on Ubiquitous Robots and Ambient Intelligence (URAI), Daejeon, South Korea, 26–28 November 2012.
57. Pardo-Castellote, G. Omg data-distribution service: Architectural overview. In Proceedings of the 2003 Proceedings 23rd International Conference on Distributed Computing Systems Workshops, Providence, RI, USA, 13–16 October 2003; pp. 200–206.
58. Google Protocol Buffers. Available online: https://developers.google.com/protocol-buffers/ (accessed on 31 August 2016).
59. Eugster, P.T.; Felber, P.A.; Guerraoui, R.; Kermarrec, A.M. The Many Faces of Publish/Subscribe. *ACM Comput. Surv.* **2003**, *35*, 114–131.
60. Emmerich, W.; Kaveh, N. Component technologies: Java beans, COM, CORBA, RMI, EJB and the CORBA component model. In Proceedings of the 2002 ICSE 2002 Proceedings of the 24rd International Conference on Software Engineering, Orlando, FL, USA, 22–25 May 2002; pp. 691–692.

61. Åström, K.J.; Hägglund, T. *Advanced PID Control*; ISA-The Instrumentation, Systems, and Automation Society: Research Triangle Park, NC, USA, 2006.
62. Lin, F.J.; Chu, P.; Liu, M.T. *Protocol Verification Using Reachability Analysis: The State Space Explosion Problem and Relief Strategies*; ACM SIGCOMM Computer Communication Review; ACM: New York, NY, USA, 1987; Volume 17, pp. 126–135.
63. Van Deursen, A.; Klint, P.; Visser, J. Domain-Specific Languages: An Annotated Bibliography. *Sigplan Not.* **2000**, *35*, 26–36.
64. Linux Traffic Control (TC). Available online: http://linux.die.net/man/8/tc (accessed on 31 August 2016).
65. WebGME Online. Available online: http://www.webgme.org (accessed on 31 August 2016).
66. WebGME Github. Available online: http://www.github.com/webgme/webgme (accessed on 31 August 2016).
67. WebGME Command Line Tool. Available online: http://www.github.com/webgme/webgme-cli (accessed on 31 August 2016).
68. NASA Student Launch. Available online: http://www.nasa.gov/audience/forstudents/studentlaunch/home/ (accessed on 31 August 2016).
69. Runeson, P.; Höst, M. Guidelines for conducting and reporting case study research in software engineering. *Empir. Softw. Eng.* **2009**, *14*, 131–164.
70. Schmidt, D.C. *Model-Driven Engineering*; IEEE Computer Society: Washington, DC, USA, 2006; Volume 39, p. 25.
71. AGSE 2015 Competition Repository. Available online: https://github.com/finger563/agse2015 (accessed on 31 August 2016).
72. AGSE Repository. Available online: https://github.com/vadl/agse (accessed on 31 August 2016).
73. ROSMOD Organization. Available online: https://github.com/rosmod (accessed on 31 August 2016).
74. ROSMOD Repository. Available online: https://github.com/rosmod/rosmod (accessed on 31 August 2016).
75. ROSMOD Tool Suite Repository. Available online: https://github.com/rosmod/webgme-rosmod (accessed on 31 August 2016).