Software System for the Mars 2020 Mission Sampling and Caching Testbeds

Kyle Edelberg, Paul Backes, Jeffrey Biesiadecki, Sawyer Brooks, Daniel Helmick, Won Kim,
Todd Litwin, Brandon Metz, Jason Reid, Allen Sirota, Wyatt Ubellacker, Peter Vieira
Jet Propulsion Laboratory
California Institute of Technology
4800 Oak Grove Drive
Pasadena, CA 91101
Kyle.D.Edelberg@jpl.nasa.gov

Abstract—**The development of the Sampling and Caching Subsystem (SCS) of the Mars 2020 Rover Mission is highly dependent on testing of prototype hardware and software operating in explicit conditions as part of integrated testbeds. To achieve relevant integration of hardware and software while maintaining rapid algorithm development capabilities and high testing throughput, the Controls and Autonomy for Sample Acquisition and Handling (CASAH) software system was developed. CASAH is an implementation of the Intelligent Robotics System Architecture (IRSA), which mimics JPL Flight Software (FSW) in that it is divided into modules that run separate processes that communicate via message passing, each module is assigned an owner that is a single developer, and the operator initiates requests via a text-based interface that interprets sequences of commands. IRSA enables a modular breakdown of CASAH that follows that of 2020 Flight Software, so developers can take an algorithm from a module in CASAH and re-code it into the same module in FSW. As deployment of CASAH has grown to ten testbeds - each with different hardware and objectives - bottom-up design decisions have been intentionally made to keep the system lightweight and maintainable by a very small team. To date, CASAH has been used to run 1393 different tests. This work describes CASAH, the testbeds and functionality it supports, the tools used to manage the development and sharing of code, and the features of the software. Lessons learned over the past three years of development and deployment are provided.**

TABLE OF CONTENTS

1. INTRODUCTION

The Mars 2020 rover mission, set to launch in July/August of 2020, is part of NASA's Mars Exploration Program and is being managed by the Jet Propulsion Laboratory. The mission consists of a Mars Science Laboratory (MSL) heritage rover with a new payload that addresses high-priority science goals for Mars exploration, including key questions about the potential for life on Mars. One of the four objectives of the mission is to collect Martian sample cores and deposit them on the surface for potential return to earth by a future mission [1], [2]. This capability is to be achieved via the Sampling and Caching Subsystem (SCS), which is currently under development [3].

The Sampling and Caching Subsystem consists of a rotary-percussive Coring Drill for collecting samples, Robotic Arm for tool placement, a system for managing drill bits and sample tubes known as the Adaptive Caching Assembly, and a Gas Dust Removal Tool for clearing rock abrasions. Development of SCS hardware and software requires prototyping and testing, which is achieved through testbeds. Development testbeds are integrated systems which consist of mechanical hardware, avionics, and software designed to operate in explicitly defined conditions in order to gather data required to refine and characterize the prototypes under test. An example of such a testbed for SCS is the Environmental Development Testbed, which consists of a Robotic Arm and prototype Coring Drill in a 10 foot thermal/vacuum chamber [4]. The high-level driving requirements for the software system for these testbeds are as follows:

i) Enable high throughput testing of prototype hardware across a suite of testbeds.

ii) Provide a framework for rapid algorithm development that can be tested on real hardware and assist with Flight Software.

iii) Allow for non-developers to operate the testbeds.

iv) Collect data products that can be easily parsed and archived.

v) Provide robust fault protection to avoid hardware damage.

Development and operation of Flight Software is complex and rigorous, as it is designed for flight hardware being operated on the surface of Mars. Due to the prototype nature of development testbeds described in this work, FSW itself is not a suitable choice for meeting the above requirements. Thus, the Controls and Autonomy for Sample Acquisition and Handling (CASAH) software system was created in 2014 for SCS development testbeds. CASAH is an implementation of the Intelligent Robotics System Architecture (IRSA), which mimics JPL rover Flight Software [5]. The CASAH implementation is lightweight and shares code and concepts from other research tasks within JPL Robotics that implement the IRSA architecture, such as RoboSimian [6]. Behavior modules, written specifically for each application and responsible for capabilities associated with a subset of physical components (e.g. DRILL, ARM) use data-driven state machines to implement high-level, semi-autonomous

978-1-5386-2014-4/18/$31.00 ©2018 IEEE

functionality. The owner of a behavior modules in CASAH is also the owner of the same module in 2020 Flight Software. This strategy enables developers to rapidly code algorithms and concepts in CASAH, test them in a variety of conditions on prototype hardware, and then, once satisfied with their performance and robustness, re-code them in FSW.

As each testbed demands unique capabilities, the tendency for a single software system that operates all testbeds to become bloated, complicated, and unusable is high. Four key steps have been taken to combat this throughout development of CASAH:

1) Use of standardized avionics across all testbeds to avoid having to redesign or overly-abstract the hardware interface components of CASAH.

2) Re-use of code and libraries where possible without dictating that a developer must do so, through appropriate division of the software repositories.

3) Maintaining a bottom-up, capability-driven approach that is focused on real requirements as opposed to potential future use cases.

4) Active resistance to chasing the 'perfect' software system that never requires modification.

In this work we attempt to provide a retrospective look on CASAH's development, integration and deployment to illustrate the challenges, successes, and lessons learned with developing a software system that supports a variety of complex testbeds, each with different configurations and objectives.

2. SCS Components and Testbeds

As new technology is developed for flight, it must be tested at increasing stages of fidelity. Test data provides invaluable feedback on prototype designs and directly influences subsequent, higher-fidelity prototypes. This applies to both hardware and software, as it is the integration of the two that delivers system capability. Each testbed is designed to test some subset of SCS components at various stages of development and is configured as shown in Figure 1. A description of the different components of SCS is given below, followed by an overview of the testbeds that CASAH supports.

Primary Robotic Arm—The primary Robotic Arm (RA) is roughly two meters in length and consists of five Degrees-of-Freedom (DoF) in a yaw-pitch-pitch-pitch-yaw configuration. It is kinematically similar to the Mars Science Laboratory (MSL) RA. Like MSL, at the end of the arm is a turret with various science and engineering instruments and tools.

Coring Drill—The Mars 2020 Coring Drill is a rotary percussive tool on the turret designed to collect sample cores of the Martian surface for potential return to Earth. It also abrades rocks to create flat areas for science instruments, and can collect regolith. The drill on MSL was designed to collect powder for in-situ analysis, thus the Mars 2020 Coring Drill functionality and design is significantly different.

Adaptive Caching Assembly—The Adaptive Caching Assembly (ACA) handles sample inspection, sealing, and dropoff. It also handles transfer of bits and sample tubes to the Coring Drill. The ACA consists of a smaller three degree of freedom robotic arm (two rotational and one linear DoF), end-effector, sealing mechanism, bit carousel, and various

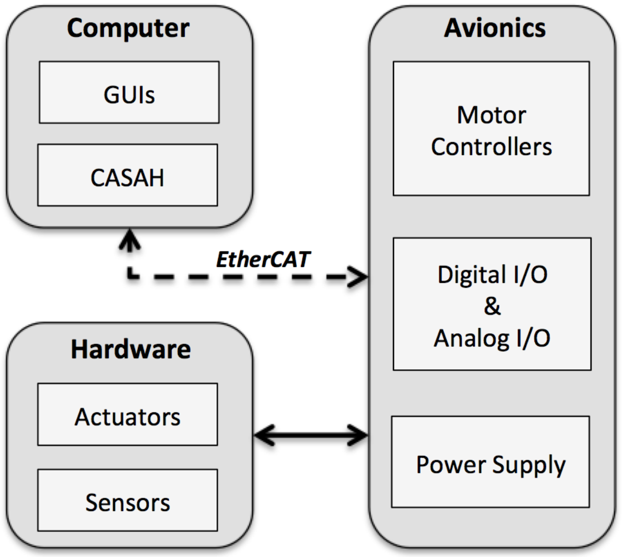


Figure 1. Block diagram representation of Mars 2020 SCS development testbeds supported by CASAH

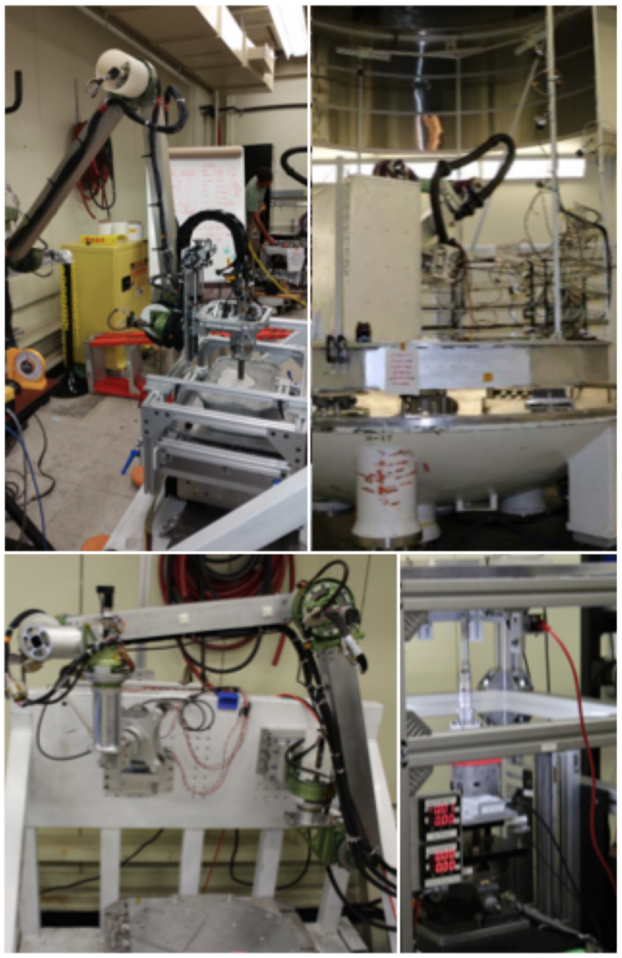


Figure 2. Four of the testbeds that CASAH supports. Clockwise, from top left: (1) Boundary Conditions Testbed (2) Environmental Development Testbed (3) Single Station Testbed (4) Docking Testbed

inspection stations. The arm and end-effector are known as the Sample Handling Assembly (SHA). The majority of the ACA mechanisms, including the SHA, are housed inside the rover.

Gas Dust Removal Tool—The gas Dust Removal Tool (gDRT) is also mounted on the turret and is designed to remove dust from an abrasion by discharging high pressure gas.

To date, there are ten distinct development testbeds that CASAH supports, each with different hardware, software, and objectives. An overview of the testbeds is provided in Table 1.

3. SOFTWARE SYSTEM

CASAH is a robotics software system design to provide high testing throughput on prototype hardware while providing a relevant framework for flight algorithm development. Figure 3 illustrates the different components of CASAH, which are described in detail below.

Architecture

CASAH is an implementation of the Intelligent Robotics Software Architecture (IRSA) [5]. IRSA is a research version of the JPL rover Flight Software architecture. IRSA is similar to flight in that the system is decoupled into modules that run independent processes which communicate via message passing. Behaviors are encoded into behavior modules which interface to the user via commands, and to hardware by creating motion requests to the motor control interface module. In mimicking Flight Software architecture by implementing IRSA, CASAH is directly relevant to SCS Flight Software development. In addition to this benefit, our experience has taught us that the JPL rover Flight Software architecture is also ideal for research and development tasks that are independent of flight projects, such as RoboSimian [6].

Middleware

At the core of the IRSA architecture is the notion of message passing; like in flight, there is strict avoidance of shared memory and thus, processes communicate through messages [7]. The CASAH implementation of IRSA uses an in-house library for message passing called *RSAP*. *RSAP* provides the application Transmission Control Protocol (TCP) and User Datagram Protocol (UDP) interfaces. UDP is used for high rate (e.g., 1kHz) broadcast of state information to other modules, and TCP is used in all other cases because of its high reliability. *RSAP* also provides mechanisms for data logging and message printing.

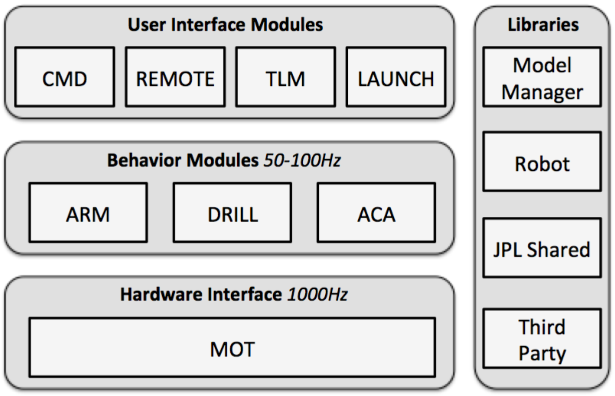


Figure 3. Controls and Autonomy for Sample Acquistion and Handling (CASAH) software system overview

Also as in flight, each module in CASAH has its own command dictionary, which serves as the primary interface definition to each module. An auto-coding tool written in-house, *CMDGEN*, eases this development. For each module, *CMDGEN* users write an XML file that specifies command line messages and replies between modules, as well as their arguments including data types and ranges. Using the user-specified XML input files, the *CMDGEN* python script auto-generates C header files and C codes that declare command and reply message data structures, functions that parse the command line messages and check valid arguments. It also auto-generates a comprehensive HTML command dictionary for operator convenience. *CMDGEN* provides several advantages over manual message coding:

1) *CMDGEN* reduces the development time of the software that support command line messages and replies between modules.

2) *CMDGEN* makes the software less error-prone with less debug time.

3) *CMDGEN* is much simpler to add or modify the command and reply messages. For this reason, JPL rover Flight Software also employs a very similar methodology.

CASAH uses the GNU Build System, also known as Autotools [8]. This system allows for streamlined building without requiring developers to directly edit Makefiles. Like Flight Software, CASAH is primarily written in the C Programming Language.

Modules

MOT—The MOT (short for 'motor') module is responsible for hardware interfacing, actuator and sensor processing, high-rate data logging, and low-level fault protection. MOT supports EtherCAT communication via the in-house driver stack that interfaces to EtherLab's open-source master [13]. The MOT process runs at 1kHz and is assigned real-time priority and a dedicated CPU core to minimize jitter and timer slip. MOT performs trapezoidal motion profiling and operates the Elmo motor controllers in Cyclic Synchronous Position mode which bypasses their internal motion profiler. MOT can receive motion requests directly from the user as well as behavior modules.

MOT records and broadcasts its public state information at 1kHz. MOT monitors actuator position tracking, forces and torques, position and velocity limits, temperatures, and a variety of other telemetry to ensure safe operation of hardware. If MOT detects a deviation beyond a parameterized limit, it initiates its fault response which is to stop all motion and engage an internally latched fault flag which prevents further motion until cleared by the user.

MOT is also responsible for managing actuator positions. On shutdown MOT writes all positions to a text file; when it is started again, it reads from this file. A set of backup files are written at 10Hz to ensure if for some reason the MOT process is not shutdown cleanly, positions are recoverable. When positions are not known (e.g., a new testbed comes online, mechanism is dissembled and reassembled, etc.), the MOT calibration functionality is used to stall the actuator against a known hardstop and set its position.

Table 1. **CASAH-Supported Testbeds for the Sampling and Caching Subsystem**

Testbed	**Hardware Components**	**Period**	**Overview**
1) Boundary Conditions Testbed	Robotic Arm, Coring Drill	08/2014-03/2015	Robotic Arm with flight-like stiffness and a Coring Drill developed in a research task leading up to the Mars 2020 Rover Mission [12]. Testing at ambient conditions to examine factors on collected sample core quality, such as robotic arm pose.
2) Percussion Efficacy and Comminution	Coring Drill	02/2015-Active	Brassboard-level Coring Drill fixed to a commercial feed stage operating in ambient conditions. Due to lack of Robotic Arm, this testbed allows for high-testing throughput of coring, abrasion, and regolith collection.
3) Environmental Development Testbed	Robotic Arm, Coring Drill	04/2015-Active	Robotic Arm with a Coring Drill in a thermal/vacuum chamber, which allows for testing sampling in Mars atmospheric conditions. Started with a Brassboard-level drill, recently upgraded to Engineer Development Unit [4].
4) Ambient Robotic Coring	Robotic Arm, Coring Drill	07/2015-05/2016	Re-purposing of Boundary Conditions Testbed - removed research tool and replaced it with Brassboard-level Coring Drill.
5) Tube Manipulation Testbed	ACA (Linear actuator only)	08/2015-08/2016	Prototype linear actuator and tube gripper for the Sample Handling Assembly. Testing interface interactions.
6) Docking Testbed	Robotic Arm	06/2016-Active	Robotic Arm and prototype docking hardware, for developing and testing self-docking capability required for bit exchange.
7) Surface Prep Operations Testbed	Coring drill, gas Dust Removal Tool	11/2016-Active	Coring Drill fixed to commercial feed stage, housed inside a small low-pressure chamber, with prototype gDRT. Testing rock abrasion and subsequent dust removal at Mars atmospheric conditions.
8) Single Station Testbed	ACA (Linear actuator and end-effector only)	03/2017-Active	Prototype linear actuator and full end-effector. Testbed has swappable ACA stations for testing various interactions.
9) Multi Station Testbed	ACA (no bit carousel or holder)	04/2017-Active	For testing SHA interactions with the different ACA stations.
10) Percussion Mechanism Testbed	Coring drill (percussion only)	08/2017-Active	Percussion mechanism from Engineering Development Unit Coring Drill attached to a dynomometer. For characterization and life testing of the mechanism.

When MOT compiles, two separate server binaries are created. The main binary is for interfacing to real hardware. The second, dubbed 'offline' MOT, compiles against EtherCAT driver stubs that enable it to be run on a development computer with no hardware. This provides developers a simulation mechanism to test some aspects of software before running on real hardware.

Due to the criticality of MOT's functionality, extra care is taken when modifying it. A peer review is performed for all MOT-level changes as well as significant regression testing to ensure proper functionality. The solid foundation that MOT provides enables rapid development of algorithms at the behavior module level.

CMD—The primary interface to CASAH is the CMD module. CMD ingests text files which consists of a list of commands to run in succession. As in flight, this text file is referred to as a sequence. Operators create sequences to perform the desired test (or some subset of it). When CMD receives the sequence, it parses it and sends the first command to the appropriate module. When the request is complete, the serving module responds with a message back to CMD, which steps to the next command and the process repeats. This continues until completion of all commands in the file, or until a failure response is received. When failure occurs, CMD stops sending out commands for execution and responds with a failure back to the operator.

While this linear execution of commands could in theory be expanded to provide more complex functionality (e.g. support of branching logic within the sequence) in practice this has not been needed as the behaviors the commands initiate handle such complexity internally. To prevent failure due to operator error, CMD performs a validity check against the command dictionary prior to execution by using autogenerated code from *CMDGEN*.

Behavior Modules— CASAH contains multiple behavior modules as show in Figure 3. It should be noted that each behavior module may have separate instantiations. For example, there are three distinct DRILL modules in CASAH, as there have thus far been three different versions of the prototype Coring Drill under test. Testbed configurations only build and install the specific behavior module(s) required for the given testbed.

Each behavior module has its own command dictionary which is exposed at the CMD level to the user. Upon receipt of a command, the module initiates the associated behavior using arguments supplied with the command. Behaviors are coded as asynchronous, data-driven hierarchical state machines (HSM) [9]. The HSM initializes controllers within the behavior module, which in turn create motion requests to MOT. The HSM is responsible for determining if the behavior completes successfully or not, and a success or failure message is sent back to CMD based on execution.

An example of a behavior HSM from DRILL, Seek Surface, is shown in Figure 4. When a command is received to run the behavior, the HSM transitions from *Idle* to *Contact* which is a child state of *Seek Surface*. The state *Contact* sends a motion request to MOT to move the Coring Drill feed forward. This moves the bit into contact with the rock, and when the force measured by DRILL is above a threshold, the HSM transitions to *Apply WOB*. This state initiates a closed-loop force controller to cyclically update a feed motion request to achieve the desired force, at which point the HSM transitions to *Backoff*. This state moves the feed to retract the drill bit from the rock, then *Rotate Spindle* moves the spindle a small amount and the process repeats. When this has been repeated a parameterized number

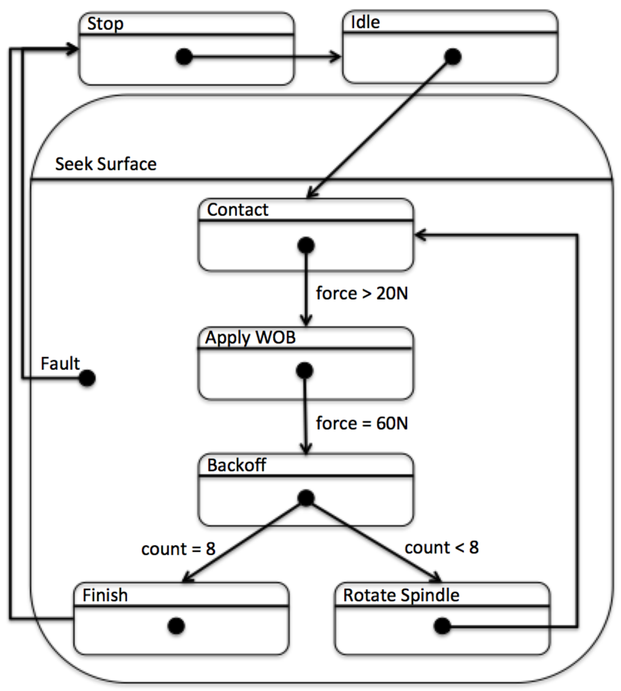


Figure 4. **Example behavior from DRILL module. Seek Surface is used to estimate the highest point of the rock surface as well as its variability**

of times, the HSM transitions to *Finish*, which records the highest surface location as well as surface variability (the maximum difference between feed positions at which the desired force was achieved for each spindle position), then to *Stop*. Once motion has settled, a success message is sent back to CMD, and the HSM transitions back to *Idle*. If a fault occurs at any point during execution, the HSM transitions to *Stop*, sends a failure message to CMD, then transitions back to *Idle*.

The behaviors, associated controllers, and algorithms they encode are the components of CASAH that we are transferring to Flight Software for the Mars 2020 Rover Mission. The Seek Surface behavior described above is an example of this, as it is currently being written into the real Flight Software DRILL module. This process is made possible by 1) running CASAH with relevant hardware and testing 2) the similarity of CASAH's architecture - IRSA - to JPL rover flight software and 3) personnel developing CASAH behaviors are also implementing the same behaviors in flight. By intentional design, this process of transference is not a direct port and thus, requires re-writing code. The prototyping, run-time, and intuition is what is gained by writing and running these behaviors in CASAH.

Graphical Interfaces—There are three graphical interfaces to CASAH. The first is LAUNCH, which is used to start and stop the various processes in CASAH. The second is REMOTE, which is a robotic visualization tool built on top of OpenGL. REMOTE is primarily used for monitoring robot state, but can also be used to send a limited set of commands, including joint space and Cartesian motion requests to ARM. The third is TLM, which is a server that broadcasts state information from all processes to TLMVIZ, a live telemetry display tool developed in-house to interface with CASAH.

Libraries

CASAH relies on several third party libraries. To ease deployment and assist with debugging, we explicitly chose to stay with open source software. A few examples of such libraries are: Bullet, used for real-time collision detection to prevent self-collision of robotic manipulators [10]; OpenGL, the Open Graphics Library used by REMOTE for graphical robot state display [11]; and EtherLab, IgH's open source EtherCAT master [13].

There are two libraries internal to CASAH. While the concept of each is shared across other projects within the JPL Robotics Section, the code itself is not. The first is Model Manager (MM). This library provides interfaces for performing kinematic and loads-related calculations. We create a robot model for each testbed that can either come from parsing a URDF or similar format that defines the testbeds kinematic chain of articulated bodies used by MM. This allows the core of MM to work for multiple testbeds. The second is Robot, which contains enumerations of all actuators and sensors for the testbed, as well as position, velocity, and acceleration limits. Behavior modules and MOT rely on the Robot library.

CASAH also shares libraries across other projects within the JPL Robotics Section that use IRSA. These include the aforementioned middleware as well as drivers, toolkits, controllers, and several other components of robotics software that is convenient to reuse across applications. A description of how these libraries, known as 'externals', are shared is given in the Code Management section below.

Code Management

CASAH development started with three developers in February 2014. The team has grown and re-shaped since then, with on average four to five people making frequent contributions at any given time. CASAH also grew from supporting a single testbed to ten, and the level of system fidelity has increased significantly. Management of the CASAH code has thus changed in multiple manners since its inception. A description is provided below.

Version Control—Version control of CASAH has morphed as the system has developed and expanded. Initially CASAH was managed using Subversion (SVN) on an internal server operated by the JPL Robotics Section [14]. Very early in development, we switched to git for version control due to git's numerous advantages over SVN, including streamlined branching, merging, and committing [15]. The repository still lived on the internal server until 2015, when we migrated to JPL's Github enterprise. Github provides a web interface convenient for collaboration and does not preclude command-line interfacing to the repository.

Externals and Internals— CASAH shares code with other projects within the Robotics Section, including RoboSimian, SmalBoSSE, Surrogate, and several planetary sampling research tasks. The in-house libraries shared between these tasks are referred to as externals. Each external has its own git repository and owner. Some examples of externals include our EtherCAT driver stack that interfaces to Etherlab's master, message passing library (*RSAP*), and command auto-coder (*CMDGEN*). Externals are designed to be lightweight and parameterizable by the application. No external is dependent on another, so they can be individually pulled into a project as needed by the application.

The second mechanism of code sharing between CASAH and other tasks is through 'internals'. Like externals, internals are their own repository which can be pulled into the main project. The distinction is that they are used to run a process and therefore are not considered libraries. Internals consist of all the core code needed to compile a certain process and are how we share modules in a semi-generic fashion. They have heavy dependencies, including certain externals and project structure, as well as project-specific code for parameterizing the internal. An example is the MOT module. The core of MOT is the message handling, actuator and sensor processing, data logging, fault protection, and hardware interfacing. All of this functionality is part of the MOT internal; CASAH then supplies a small set of files which the internal compiles against to define the bus topology, sensors, actuators and their parameters, etc. Internals allow for code sharing at the module level without overly constraining each project.

Testbed Configuration, Branching, and Tagging— Because CASAH originally only worked for a single testbed there were several possibilities regarding how to best support new testbeds with different hardware and functionality. Our experience has taught us that the price paid for arbitrary generality and abstraction is an unmaintainable software system, so we sought to avoid blindly supporting all testbeds. Our first attempt at a targeted approach was for every testbed to be a separate branch; ARM, MOT, etc. could all be parameterized in this manner for the specific testbed. Each branch only contained the behavior modules required for that testbed. Since the bulk of the code across each branch was in fact the same, this required developers to cherry pick bug fixes and updates that affected the common code into their specific testbed branch. We found that this solution was error-prone and not scalable.

This led to the concept of testbed 'configuration', specified during the build process. The CASAH code contains all behavior modules and configuration files for every testbed, including things like kinematic definitions of all the different manipulators for each testbed. But during the configure step of the AutoMake build procedure, the developer specifies which testbed to compile for. This tells the build system to compile only the modules needed for the testbed as well as select testbed-specific files for configuration. This concept allowed us to parse out testbed-specific code from each module, while retaining commonality across configurations.

The concept of testbed configuration allowed us to work from a single, unified CASAH master; all development of CASAH occurs on a side branch, and when the work is complete, it is merged back into the single master. The master should always be stable and up-to-date. Thus changes which could affect system performance may have to be tested by the developer on relevant hardware prior to the merge. Once the merge is complete the updated version can be tagged, if desired. Only tagged releases are installed on the operator account of the testbed computers. The release tag name is stored in the data logs created by operators when running all tests.

Development of internals and externals also occurs on branches, which are merged to their master and tagged when the update is complete. Part of the CASAH repository is two scripts: one for pulling in tagged version of internals, and one for externals.

4. Hardware-Software Integration and Testbed Bring-up

As a robotics software system, CASAH is tightly coupled with electrical and mechanical hardware. From its conception, CASAH was designed to integrate only with the set of hardware required for Mars 2020 SCS testbeds. No attempt was made to architect CASAH such that it could interface to any arbitrary set of hardware; in fact, such top-down philosophy was explicitly avoided in favor of a bottom-up, capability-driven approach. In this way, the complexities and challenges associated with an overly abstract and generic system were avoided. A description of the hardware CASAH integrates with is given below.

Avionics

CASAH was developed in conjunction with the avionics for the Mars 2020 SCS testbeds. EtherCAT was selected as the communication bus due to its low jitter and high bandwidth [13]. Elmo DC Gold Whistle motor controllers (100Volts, 1-20Amps) were chosen for running actuators. The Whistle supports a wide range of motor and feedback types, and our robotics staff has experience with them across several projects. Beckhoff EtherCAT modules are used for all non-actuator I/O, including reading force-torque sensor strain gauges, toggling digital outputs, etc. The Whistle's high level of flexibility and Beckhoff's diverse catalog allows for adaptability at the low-level without increasing overall system complexity, as the bus type and drivers are standardized and common across all testbeds.

A proof-of-concept avionics box consisting of Gold Whistles, Beckhoff modules, and power converters was created for operating a five degree of freedom robotic arm and coring tool in 2013 [16]. This informed the design of the avionics system, dubbed the 'Blue Box', which has been used by the Mars 2020 SCS development testbeds since 2014.

Computer

CASAH is developed and deployed on 14.04 Ubuntu Linux Operating System. The testbed computer is either a Dell T7600 or T7810 with an Nvidia Quadro K2000 graphics card, 2.4Ghz Intel Xeon E5-2630 processor, 32 GB of RAM, and 2TB 7200RPM hard drive. The low-latency Linux kernel, version 3.16, is used for reduced jitter and to allow for setting process priorities. Each computer has two Ethernet ports: one for EtherCAT communication and the other for JPL intranet. A single computer is used for both real-time control and operator interfacing, thus, all of CASAH's processes are run locally.

Mechanical Hardware

While every testbed has a different hardware configuration, there is inherent commonality which CASAH's implementation exploits to reduce complexity. The first is that all actuators are either rotational or translational, and consist of brushless DC motors with incremental encoders or hall effect sensors for commutation and position feedback. The second is that no testbeds have mobile bases, and thus, there is no need to track or estimate a moving robot frame of reference. Third, no testbed uses exteroception for state estimation or feedback.

Development and Integration Process

There are several steps performed by the software team to bring a new testbed online. Prior to hardware arrival, the testbed is assigned a software owner who follows the general process outlined in Figure 5. A description of each phase is given below:

Phase I. *Initial Development* When a testbed is conceived, it is created for a specific set of objectives. The software team translates these objectives (e.g., test end-to-end coring with a brassboard drill on a robotic arm) into the software functionality required. A developer creates a testbed configuration to support the given hardware and objectives, and behavior modules are created / modified as needed. Functionality is tested in offline simulation mode to shake out as many bugs as possible prior to integration.

Phase II. *Partial Integration* As avionics, actuators, and sensors become avaliable, partial integration can begin. A significant component of this process is hardware bring-up.

Motor Tuning—Each unique motor has to go through a 'tuning' process, during which the non-volatile Elmo parameters used to operate the motor are determined and programmed into the motor controller. This process is performed using the Elmo Application Studio (EAS), a closed-source Windows program. Some of these parameters are dictated by the motor specification (commutation feedback type, current limits, speed limits, etc) and thus are set directly by the engineer. Other parameters, such as gains used by Elmo's internal control loops, are determined by EAS automatically via various system identification routines. Once the motor tuning process is complete, the motor can be jogged in free space. Data sets of current, speed, voltage, etc. are collected and stored for future comparison.

It should be noted that the tuning process is performed with the motor de-coupled from the mechanism it drives. This is in part because 1) in certain situations, the motion induced by

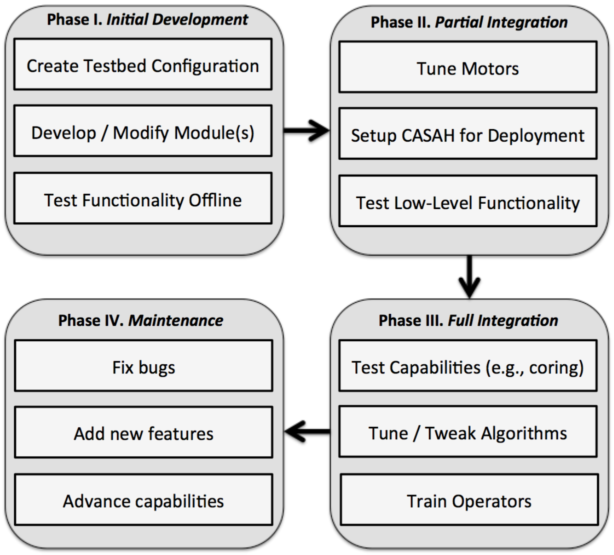

Phase I. *Initial Development*
Create Testbed Configuration
Develop / Modify Module(s)
Test Functionality Offline
Phase II. *Partial Integration*
Tune Motors
Setup CASAH for Deployment
Test Low-Level Functionality
Phase III. *Full Integration*
Test Capabilities (e.g., coring)
Tune / Tweak Algorithms
Train Operators
Phase IV. *Maintenance*
Fix bugs
Add new features
Advance capabilities

Figure 5. Phases of CASAH development for a given testbed at various stages of hardware-software integration

EAS system ID is not feasible once the motor is integrated and 2) the large majority of our applications use high gear ratios and thus the inertial load has little impact on the motor dynamics. The other advantage is that it eases the debugging process to bring things online in an incremental manner. Once the motor is tuned, it can be integrated into the mechanism. The Elmo parameters are burned to the motor controller. A text-file version is stored in the CASAH repository as backup. On average, this process takes approximately four hours per unique actuator.

Setting up CASAH—To ease the setup of the testbed computer, a master hard drive with the specific OS, kernel version, packages, and various other configurations is used to rapidly create a hard drive clone. As each testbed has a different set of Beckhoff modules and Elmo motor controllers, some configuration of CASAH at the avionics interface level is required for each application. Once the computer is functional, the version of CASAH specific for the new testbed can be compiled and tested. Initial tests are done to confirm CASAH can communicate with all the avionics and that valid data is being received. On average, this process takes roughly one day.

Low-level Testing— Once CASAH is communicating with the avionics and motors have been tuned, basic free-space motion of each mechanism can be performed. This verifies the tuning process as well as the gear ratios, encoder/hall counts per revolution, speed limits, etc. that are hard-coded into the configuration of CASAH for the specific testbed. Mechanisms with restricted ranges are calibrated by moving the actuator to stall against the hardstop at a known location. Sensor values are also verified at this stage, typically by checking data as produced by CASAH against a ground-truth measurement (e.g. place a known weight on a force sensor and verify the change measures correctly). Low-level fault protection, such as position restrictions, maximum temperatures, and force/torque limits are also tested. The timeline for this process depends on the testbed complexity, but on average takes approximately two days.

Phase III. *Full Integration* Once partial integration is complete, and the testbed has been fully assembled, full functionality can be tested. High-level capabilities are tested in nominal conditions first. As these behaviors are testbed-specific, there is no universal procedure to follow for bringing them online with hardware. This process may include tuning force controllers, verifying coordinated multi-axis motions, and testing end-to-end autonomous behaviors. The time it takes to complete this step can be two weeks or more. Baseline data sets are recorded for future reference. Operators are trained in order to handoff operations from the software team to the operations team.

Phase IV. *Maintenance* After testbed operations have been handed off, the software team remains on call for any issues that arise during testing. This can include both software, hardware, and system-level issues, as often untangling the root cause can require detailed parsing of data logs and source code. If the cause is software, a bug fix is performed and a new software tag is installed on the operations computer. New features are also sometimes added, either to modify an approach or to streamline operations, as well as advanced capabilities.

Lessons Learned from Hardware-Software Integration

(1) It is ideal to have a spare of every sensor and actuator. This allows for parallel bring-up / mechanical assembly. It also helps with debugging.

(2) To save on schedule, test components incrementally as they are ready, as opposed to waiting for the whole testbed being constructed to test for the first time.

(3) If they exist, system-level and interface issues are most commonly caught at the hardware-software integration stage. Time should be budgeted accordingly.

(4) It is worth being diligent and taking baseline measurements of the system, from low-level hardware and sensor characterization to high level algorithmic-based behaviors. This information becomes invaluable when issues arise down the line.

(5) The software team should have continuous access to a dedicated set of relevant avionics and development hardware in order to test bug fixes and upgrades. An example of such a setup is shown in Figure 6.

(6) The more tightly coupled the hardware and software teams are, the better.

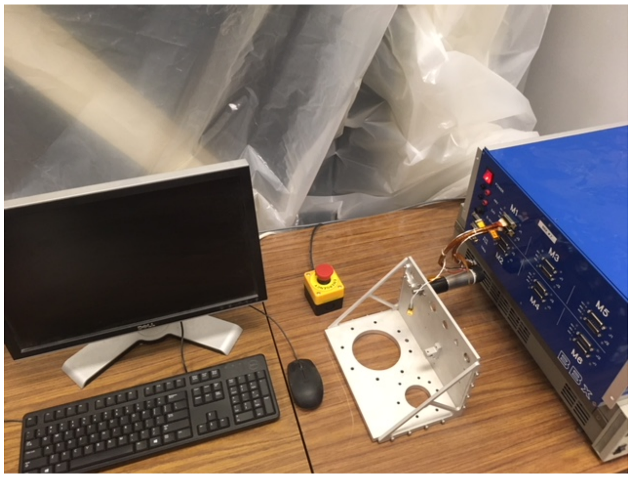


Figure 6. Software development station consisting of a 'Blue Box' avionics system, computer, and representative actuator for low-level checkout

5. Deployment on Testbeds

To date, CASAH has supported a total of ten different development testbeds for the Sample Caching Subsystem. CASAH was first deployed in August 2014, and has since been used to perform nearly 1400 tests. The number of tests performed on each testbed is given in Table 2.

The data from all these tests is collected and stored in MATLAB file format on a shared drive. The data is analyzed and parsed to generate results that answer questions stemming from the various testbed objectives.

Significant Issues and Resolutions

As run-time on CASAH has grown, issues have inevitably been discovered. When trouble occurs, testbed operators document the issue and contact the software team to assist with root-cause analysis and resolution. In certain cases, an operational work-around (such as resetting from an unexpected fault condition) is feasible; in other cases, software system updates may be required. Table 3 lists a subset of development and run-time issues we have run into to provide insight into the types of problems that have been discovered and how they were resolved.

6. Summary

The Controls and Autonomy for Sample Acquisition and Handling software system was created in 2014 for Mars 2020 SCS development testbeds. CASAH is an implementation of IRSA, and thus mimics JPL Rover Flight Software. By doing so, the algorithms developed in CASAH and integrated with the Robotic Arm, Coring Drill, Adaptive Caching Assembly, and Gas Dust Removal Tool prototypes can be re-implemented in Flight Software with little risk or iteration required. The CASAH development process has followed a bottom-up approach to ensure the system remains lightweight and manageable by a small team. We experimented with various mechanisms for code management, such as version control, in order to strike the right balance of proper code sharing without over-burdening developers.

Table 2. Number of tests run on each testbed using CASAH, as of 10/2017

Testbed	Number of Tests
1) Boundary Conditions Testbed	93
2) Percussion Efficacy and Comminution	426
3) Environmental Development Testbed	282
4) Ambient Robotic Coring	136
5) Tube Manipulation Testbed	105
6) Docking Testbed	41
7) Surface Preparation Operations Testbed	85
8) Single Station Testbed	159
9) Multi Station Testbed	21
10) Percussion Mechanism Testbed	45

Table 3. Select CASAH issues and resolutions from past three years of deployment

Date	Issue	Resolution
08/2014	MOT timer slip during force control	Found that when DRILL was running force-control, MOT would often overrun its loop timer. Found that MOT was printing continuously at each request it received from DRILL at 100Hz. Modified MOT's logic to only print on first motion request.
03/2015	Some uncontrolled motion would occur when E-stop pressed	This affected a robotic arm for a testbed. If moving when the E-stop was pressed, the arm would fall under gravity a small amount before brakes closed. Determined to be a hardware limitation, but required adding a 'soft' E-stop feature, whereby a physical E-stop would toggle a digital input that CASAH would read and use to initiate a smooth motion ramp-down.
06/2015	Separate branches for each testbed infeasible for developers	Created testbed configuration files and integrated with build process. Changed CASAH to consist of single, unified master branch.
07/2015	MOT timer slip during graphics processing	Found that Nvidia graphics driver was clashing with MOT process at the OS level. Selected alternate graphics card with specific open-source driver that eliminated the clash. Retrofitted all computers with this card, and to ensure correct driver version started the master hard drive cloning system.
07/2015	Externals versions not being managed or tracked	Moved all externals from SVN to git on JPL's GitHub. Changed CASAH to pull in specific tag numbers of all externals via a version-controlled script.
08/2015	Hard drives getting full on operation computers	Added notification to alert operator that hard disk is getting full. Dropped MOT's data logging rate for MOT and behavior modules to 1Hz when no motors have been active for more than 60 seconds.
10/2015	Could not use Beckhoff module that had digital inputs and outputs	A Beckhoff module that had both inputs and outputs did not work with our EtherCAT drivers. Found bug in driver design, requiring major overhaul. Updated driver stack, performed significant testing, then switched CASAH to support new design.
11/2015	Still getting timer slips in MOT process	By tracing where timer slip was occurring, found fflush in several parts of low-level message printing. Added option to disable fflush, setting default to disable it for all CASAH modules.
05/2016	MOT would not shut down	Determined cause was persistent fault on the motor controller inhibiting MOT's state machine from allowing it to terminate. Added persistence counter and modified MOT's shutdown logic accordingly.
08/2016	Testbed operations computer kept locking up	Found that multiple instantiations of telemetry display were running. Found that CASAH script used to start the display was not checking if instance was already running in the background, which could happen if not closed properly. Updated script to alert operator if display is already running when they try to start it.

CASAH has been used to perform nearly 1400 tests across 10 ten different testbeds, providing invaluable data to inform both hardware and software design. Hardware has been characterized and test data has informed refined prototypes and flight designs. Algorithms and behaviors developed and tested in CASAH are currently being written into Flight Software for the 2020 Rover Mission. And aspects of system performance - including any potential issues that can only be revealed through testing - has been characterized because of the relevant manner in which hardware and software has been integrated and tested.

ACKNOWLEDGMENTS

The authors would like to acknowledge the numerous individuals who developed the prototype hardware and ground support equipment for the testbeds. We would also like to acknowledge Marc Pomerantz and Darren Lee who developed additional Graphical User Interfaces tools that interface to CASAH to ease operations. Finally, we would like to extend a special acknowledgment to the testbed operators and the data processing team, who have been the end users of CASAH and its products. They have provided invaluable feedback to us.

The research described in this publication was carried out at the Jet Propulsion Laboratory, California Institute of Technology, under a contract with the National Aeronautics and Space Administration (NASA).

References

[1] "Mars 2020 Mission Overview." [Online]. Available: https://mars.nasa.gov/mars2020/mission/overview/. Accessed Oct. 9, 2017.

[2] R. Mattingly and L. May, Mars Sample Return as a campaign, IEEE Aerospace Conference, March 2011.

[3] "Flight Projects - Mars 2020 Rover." [Online]. Available: https://www-robotics.jpl.nasa.gov/projects/Mars2020.cfm. Accessed Oct. 9, 2017.

[4] L. Chu, K. Brown, K. Kriechbaum, "Mars 2020 Sampling and Caching Subsystem Environmental Development Testing and Preliminary Results," IEEE Aerospace Conference, March 2017.

[5] P. Backes, K. Edelberg, P. Vieira, et. al, "The Intelligent Robotics System Architecture Applied to Robotics Testbeds and Research Platforms," IEEE Aerospace Conference, March 2018 (Submitted).

[6] S. Karumanchi, K. Edelberg, I. Baldwin, et. al, "Team RoboSimian: Semi-autonomous Mobile Manipulation at the 2015 DARPA Robotics Challenge Finals," Journal of Field Robotics: Special Issue on the 2015 DARPA Robotics Challenge Finals, 2016.

[7] "JPL Institutional Coding Standard for the C Programming Language Version 1.0," March, 2009.

[8] "GNU Automake." [Online]. Available: https://www.gnu.org/software/automake/manual. Accessed Oct. 9, 2017.

[9] M. Samek, "Practical Statecharts in C/C++: Quantum Programming for Embedded Systems," CRC Press, 2002.

[10] "Bullet Collision Detection and Physics Library." [Online]. Available: http://bulletphysics.org/Bullet/BulletFull/. Accessed Oct. 9, 2017.

[11] "OpenGL: The Industry's Foundation for High Performance Graphics." [Online]. Available: https://www.opengl.org/. Accessed Oct. 9, 2017.

[12] P. Backes, J. Aldrich, D. Zarzhitsky, K. Klein, P. Younse, "Demonstration of Autonomous Coring and Caching for a Mars Sample Return Campaign Concept," IEEE Aerospace, March 2012.

[13] "IgH EtherCAT Master for Linux." [Online]. Available: http://www.etherlab.org/en/ethercat/. Accessed Oct. 9, 2017.

[14] "Apache Subversion: Enterprise-class centralized version control for the masses." [Online]. Available: https://subversion.apache.org/. Accessed Oct. 9, 2017.

[15] "git: Distributed Even if Your Workflow Isn't." [Online]. Available: https://git-scm.com/. Accessed Oct. 9, 2017.

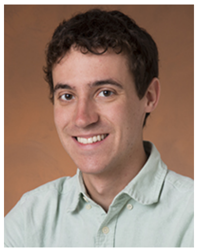
[16] P. Backes, R. O'Flaherty, D. Helmick, et. al. "Tube transfer using the sampling arm for Mars sample caching," IEEE Aerospace Conference, March 2014.

Biography

Kyle Edelberg *has been a robotics engineer in the Robotic Manipulation and Sampling group at JPL since obtaining his B.S. and M.S. in Mechanical Engineering from UC Berkeley in 2013. Kyle is the algorithms and software lead for the Mars 2020 Rover Coring Drill. Kyle's interests lie in delivering end-to-end capabilities for mobile manipulation systems operating in real-world conditions.*

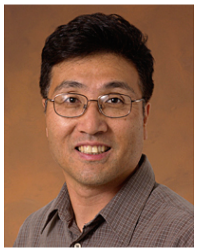
Paul Backes, Ph.D. *is the Group Supervisor of the Robotic Manipulation and Sampling group at Jet Propulsion Laboratory, where he has been since 1987. He received the BSME degree at U.C. Berkeley in 1982 and Ph.D. in Mechanical Engineering from Purdue University in 1987. His awards include NASA Exceptional Engineering Achievement Medal (1993), JPL Award for Excellence (1998), NASA Software of the Year Award (2004), IEEE Robotics and Automation Award (2008), and NASA Exceptional Service Award (2014).*

Jeffrey Biesiadecki *has been a software engineer at NASA's Jet Propulsion Laboratory since 1993, after completing his Master's degree in Computer Science at the University of Illinois, Urbana-Champaign. He designed and implemented the core motor control and non-autonomous mobility flight software for the Mars Exploration Rovers and Mars Science Laboratory rover, and was a rover driver for both missions, responsible for command sequences that tell the rover where to drive and how to operate its robotic arm on the surface of Mars. He is presently leading the development of the Mars 2020 rover Sampling and Caching Subsystem flight software.*



Sawyer Brooks *is a robotics software and systems engineer in the Robotic Manipulation and Sampling Group at JPL. He completed his B.S.E in Mechanical Engineering and his M.S.E in Robotics at the University of Pennsylvania in 2015. His research interests are in robust autonomous systems for science and in-space assembly. He develops docking autonomy and is the software lead for the gas Dust Removal Tool on Mars 2020.*

Daniel Helmick *is a former member of the Robotic Mobility group at Jet Propulsion Laboratory, California Institute of Technology. He received his B.S degree in Mechanical Engineering from Virginia Polytechnic Institute and State University and his M.S. in Mechanical Engineering with a specialization in controls from Georgia Institute of Technology in 1996 and 1999 respectively. He worked on robotics research and flight projects at JPL*



Won Kim *received the Ph.D. degree in electrical engineering and computer sciences from the University of California, Berkeley, in 1986. Since 1988, he has been in Jet Propulsion Laboratory. In 2009, he received NASA Exceptional Engineering Achievement Medal for the successful maturation, validation, and deployment of the Visual Target Tracking capability for Mars Exploration Rover (MER) mission. He developed several flight software modules for Surface Sampling and Science (SSS) system of Mars Science Laboratory (MSL) rover mission. He is currently the robotics system engineer for Mars InSight Mission and a robotic system analyst for the Sample Handling Assembly (SHA) of Mars 2020 mission.3*

Todd Litwin *received a B.S. in Applied Physics from Harvey Mudd College in 1979. He has been with JPL for more than 37 years. He has designed and written flight, ground, and robotics software for several flight projects and many research tasks. His first job at JPL was monitoring the Earth's ionosphere in support of spacecraft navigation for the Voyager, Viking, and Pioneer projects.*

Brandon Metz *received BS and MS degrees in Electrical Engineering from the California State Polytechnic University in Pomona, CA. He has worked on various flight projects at JPL including the Mars Science Laboratory and Mars 2020, focusing on mechanisms and motor control. He also spent two years working on the Canada France telescope in Hawaii. When not figuring out how to make robots move very precisely, he can be found hacky sacking and surfing.*

Jason Reid *has been a robotics technologist in the manipulation and sampling group at JPL since earning his Ph.D. from UC Berkeley in 2012. He is the current robotic arm systems engineer for Mars Science Laboratory and the robotic arm analyst for the Mars 2020 project. Jason has interests in mobile manipulation including vision based robotic arm pose estimation.*

Allen Sirota *is the technical group supervisor of the Robotic Actuation and Sensing Group at NASA Jet Propulsion Laboratory, California Institute of Technology, where he has been since 1983. He received his B.S. degree in Electronics Engineering from the University of California, Los Angeles, in 1976. Allen Sirota received the 1997 NASA Exceptional Engineering Achievement Medal for his contributions to the Mars Pathfinder Sojourner Rover mission.*

Wyatt Ubellacker *is a robot technologist in the Manipulation and Sampling Group as part of the Mobility and Robotics Systems Section at JPL. Within this group, Wyatt has worked on software and control of robotic systems spanning sample acquisition and handling to high degree of freedom planning and coordination. Prior to this, Wyatt received his B.S. in Mechanical Engineering (concentration in Controls, Instrumentation, and Robotics) from the Massachusetts Institute of Technology in 2013 and his M.S. in the same program in 2016.*

Peter Vieira *has been a robotics technologist in the Robotic Manipulation & Sampling group at JPL since completing his M.S. in Electrical & Computer Engineering at Georgia Institute of Technology, where he was a main team member of their DARPA Robotics Challenge team, Hubo. Pete is the algorithms and software lead for the Mars 2020 Rover Adaptive Caching Assembly and the Comet Surface Sample Return task. He focuses on delivering end-to-end capabilities for mobile manipulation platforms in real-world applications.*