The SMACH High-Level Executive

Jonathan Boren and Steve Cousins

Personal robotics applications often require the integration of hundreds of components. In robot operating systems (ROSs), such subsystems and primitive capabilities are usually encapsulated in ROS nodes. Even with encapsulation and well-documented messaging interfaces, writing maintainable code to make a large set of ROS nodes to act together to solve a problem is difficult. Solution strategies range from writing code in big if/else cascades and nested switch statements to using more powerful inference and task-planning systems. In this column, we introduce an approach based on nested state machines that has proven very effective at building real-ROS applications.

Complex Appications

Over the past couple of years, we have been exploring the trade-offs between task scripting and task planning for high-level control in robot applications written on top of ROS. Scripting approaches let the programmer not only say exactly what the robot should do, but also require the programmer to explicitly describe recovery logic for all failure modes. Although these methods can be rapid for developing small applications, they do not scale well. When failures arise, robots are not like pure software systems: they cannot just reset the state of the world and retry. As a result, autonomous robotics applications require a large amount of additional work to describe how to recover from these failures in addition to the application's nominal execution. Furthermore, our experience has shown that maintaining, extending, and fixing such scripts over time makes it more and more challenging to analyze or model the application.

On the other end of the spectrum, instead of explicitly describing which actions to execute in an imperative programming language, more autonomy can be given to the robot to plan and execute tasks. There exist model-based task planning and inference systems based on classic artificial intelligence (AI), constraint satisfaction, and model checking. The model, in this case, describes constraints and relations relevant to the set of actions at the robot's disposal. These systems aim to shift the burden of solving the application-specific problems from the developer to the autonomous system.

Digital Object Identifier 10.1109/MRA.2010.938836

Rapid Development Needs

Ultimately, we want autonomous robots to do useful tasks in unconstrained human environments. Model-based task planning and inference systems have the potential to allow the robot to recover from unexpected failures if said failures can still be represented within the model. We have significant experience in this space using the personal robot 2 (PR2) at Willow Garage.

One such system is teleoreactive executive (TREX), which was originally developed at the Monterey Bay Aquarium Research Institute for hybrid deliberative/reactive mission planning on autonomous underwater vehicles. This system has also been used extensively as the executive for previous PR2 applications, such as the PR2 Milestone 2 benchmark, in June 2009 [2]. While the system scaled well to satisfy our computational needs, the known development strategies and design patterns did not.

Model-based executives have the potential to prevent undesired conflicts and produce unexpected solutions to a problem, but leaving too much to be decided by the autonomous system can impede development of the application. Models can be over- or underconstrained, leading to undesired behavior, or a failure to plan entirely. Since a full task plan can only be realized once the planner receives the model and the sensor feedback, it is hard to design models that will produce the desired behavior without numerous model design iterations. Simulators can make the iteration cycle faster, and in ROS, any executive can be run in simulation as before being run on physical hardware. Even with simulation, however, this design process is counterproductive when we know exactly how to describe what the robot should do in terms of a task-level flowchart. We found that we could most efficiently construct and maintain models in TREX with state-machine design patterns [1]. In this case, the task planner is working primarily as a model constraint checker and resource manager, with task planning only happening at the highest level.

SMACH

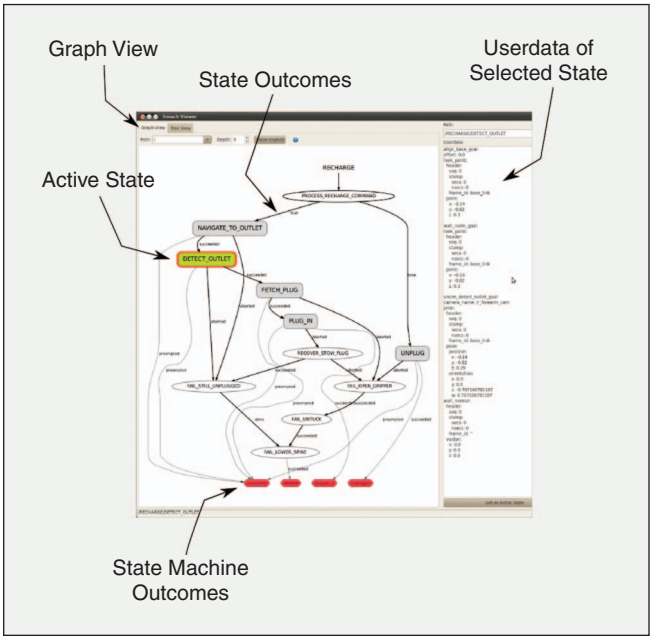
Since it seemed like these tasks, while complex, are well-defined and could be described explicitly, we started to investigate the possibility of a multiexecutive solution to

task-level robot control. In this case, these well-defined tasks could be planned out explicitly, while the less structured ones could be handled at a higher level with a different system. The first step in exploring this strategy was to create an architecture for developing robust midlevel executives. Not only should these executives be able to be controlled by a higher level task-planning system, but they should also be able to be built very rapidly for doing closed-loop systems testing.

We began developing a Python application programming interface (API) based on hierarchical concurrent state machines. We chose Python because of its shallow learning curve and native ROS bindings. The library is called SMACH, a contraction derived from "State MACHine" that is pronounced like "smash." At its core, SMACH is a ROS-independent library that can be used not only to build hierarchical and concurrent state machines but also any other task-state container that adheres to the provided interfaces. While the SMACH core is a ROS-independent library, a considerable amount has been written in the `smach_ros` package for communicating with ROS systems, such as topics, services, and `actionlib` actions.

The core SMACH library is lightweight and, along with logging and utility functions, provides two main interfaces: `State` and `Container`.

SMACH `States` represent "states of execution," each with some set of potential outcomes. SMACH `States` implement a blocking `execute()` function, which runs until it returns a given outcome.



SMACH containers are collections of one or more states, which implement some execution policy. The simplest such execution policy is the `StateMachine`. A SMACH state machine can be visualized as a state-flow diagram, where nodes are states of execution (the robot doing something), and edges represent transitions from one state to another state via a given outcome. SMACH state machines are also `States`, themselves, so they can be composed hierarchically. This means

that SMACH state machines also have outcomes of their own. These outcomes are treated like other transition targets (like states) in the state machine.

Another simple execution policy is the SMACH `Concurrence`. Unlike a `StateMachine`, which executes one state at a time in series, the `Concurrence` executes more than one state simultaneously. Concurrences are also states as well, and their outcomes can be determined by one of several outcome policies defined at construction.

Data Driven

SMACH is not the first architecture to allow users to define hierarchical, concurrent state machines, as these are very old concepts [3], and people often implement their own state machine API, markup language, or model. State machines built with SMACH, however, can diverge from formal state machines with some features unique to SMACH. Each SMACH container has a locally scoped dictionary of user data that can be accessed by each of its child states. This allows states to access data that was written by previously executed states. While this makes analysis more complex, it also makes the system far more powerful since not only can data be passed around, but data can also be accumulated from various states to inform a branch later in execution. This means that the "full" state of a SMACH tree at any given time is the union of the active task-level states in each container and the contents of dictionary of user data in each container.

Since SMACH is written in Python, any type of Python object can be stored in a container's user data dictionary. This includes, for example, ROS message types. Most of the widely used planning and execution frameworks opt to use their own languages for describing either plans or information used to generate plans. While these languages are usually better designed for this role, it means that support for user-defined types that can be processed by lower level systems requires defining and binding data structure translation functions where the executive interacts with these systems. SMACH's ability to directly manipulate these structures allows us to coordinate not only tasks with SMACH but also their associated input arguments and result data.

ROS Interfaces

While a developer can create a custom SMACH state class that executes arbitrary Python code, there are several parametrized state classes that make it even easier to compose lower level systems in ROS. Some of these include, but are not limited to:

- `ServiceState`: It is a state that represents the execution of a ROS service call. This state is parametrized by the service call name, type, and an optional pair of callback functions for generating a request and processing the service response.
- `MonitorState`: It is a state that monitors a given ROS topic. It can take a user-defined callback function that gets executed with each message received.
- `SimpleActionState`: It is a state that represents the execution of a ROS `actionlib` action. This is one of the most used task-abstraction layer in ROS. This state can be given goal generation and result processing callbacks similar to `ServiceState`, as well as other goal and result policies.

Visualization

In addition to ROS tools for visualizing and analyzing data flow over the network, SMACH adds a tool for analyzing high-level systems at run-time. Task-level failures are difficult to debug, since they often happen at the system integration points, so this places a large burden on the execution framework to provide adequate debugging and visualization tools.

We have developed an introspection system that renders the structure of a SMACH plan, highlights the executing states at run-time, and lists the contents of the user data dictionary for a given container. This interface allows a developer to quickly identify errors in specifying connections between different states and observe immediately what the executive is trying to do.

Since the graph shown in the SMACH viewer maps directly onto the structure of the running code, it is easy to catch errors and quickly repair them.

While there is a visualization interface for SMACH, SMACH is not visual programming. One can easily follow how a SMACH plan will execute by looking at the task graph, but the plans that are described with SMACH tend to grow to be too complex to lay out by hand.

Open Source

SMACH has already been used in several projects involving the PR2, where time was critical, and proved useful for both iterative development and debugging. These projects included autonomous recharging, opening doors, and three one-week "hackathons": playing billiards, integrated table clearing/cart pushing, and fetching drinks from a refrigerator. More information about and tutorials on SMACH can be found on the ROS wiki www.ros.org/wiki/smach.

References

[1] C. McGann, E. Berger, J. Bohren, S. Chitta, B. P. Gerkey, S. Glaser, B. Marthi, W. Meeussen, T. Pratkanis, E. Marder-Eppstein, and M. Wise, "Model-based, hierarchical control of a mobile manipulation platform," in *Proc. ICAPS Workshop Planning and Plan Execution for Real-World Systems*, Thessaloniki, Greece, 2009.

[2] W. Meeussen, M. Wise, S. Glaser, S. Chitta, C. McGann, P. Mihelich, E. Marder-Eppstein, M. Muja, V. Eruhimov, T. Foote, J. Hsu, R. B. Rusu, B. Marthi, G. Bradski, K. Konolige, B. P. Gerkey, and E. Berger, "Autonomous door opening and plugging in with a personal robot," in *Proc. IEEE Int. Conf. Robotics and Automation (ICRA)*, 2010, pp. 729–736.

[3] N. J. Nilsson, "Hierarchical robot planning and execution system," Stanford Res. Inst., AICPub76:1973, Apr. 1973.