Programming with Behaviors

Sudheer Apte
03-February-2017

Deploying an industrial robot is a major project. Even after selecting a robot, attaching end-of-arm tooling, and integrating other equipment in the work cell, you still need many iterations of planning, design, simulation, and testing of the robot application. Only established manufacturers with long-term automation needs and access to experts can afford the trouble, expense, and time for this project.

Smaller-volume, high-mix shops cannot justify these large automation deployment projects. The target user in such shops is a small team of generalists, usually a single person, who needs to do everything and is not an expert.

Everybody understands that industrial robots need to be "easy to use" to reduce the time, risk, and level of expertise needed to deploy. But what does that mean, exactly?

In this paper, I describe a proven architecture called a behavior engine and its unique user interface, which together allow non-experts to quickly build powerful robot applications.

A behavior engine provides a framework on which expert programmers can develop special-purpose building blocks. A generalist user can use the framework and these building blocks to build a robot application.

The ideas in this paper are a result of behavior-based robotics research and practical experience with building a behavior engine for an industrial robot. In a tutorial section below, I explain what a behavior engine is and how it makes robot applications easy to develop.

Then, in an example section, I describe how a behavior engine works and some example building blocks to create a few applications.

Finally, in a design section, I describe how a behavior engine should work internally and what enhancements need to be made to the concept.

1 Introduction

Robot applications and smarter robots

The whole user experience of programming and integrating robots into the work cell today is poor. One reason is that we try to simplify the task for the **robot,** instead of for the **user.**

Say a robot needs to pick up a workpiece. We make the user present it exactly in the same position every time, so that the gripper can reliably pick it up. The robot is programmed to move through a pre-recorded sequence of motions. What could be simpler?

But the cost of this simplicity is borne by the user, who becomes responsible for "part presentation." The user has to design special dispensers or fixtures to hold the workpieces, so that they can be spoon-fed to the robot.

But this kind of precisely repeatable application becomes **brittle.** The slightest variation causes the robot to miss. If something in the work cell moves out of place, or if the robot's mounting position or platform gets loose, the application can fail.

The user can "touch up" the robot's recorded positions to make the application work again. But it remains vulnerable to variations in the environment. Once the user has gotten the application working, he is reluctant to change the program or anything in the work cell, afraid to break it.

Manufacturing shops that are producing short runs of parts don't want to spend too much time setting up the work cell precisely. They want to take existing work cells, where people are doing the work, and try to automate them as quickly as possible.

They really need the robot to be more capable and smarter. For example, if the robot could use a vision system to locate the part, then it could pick up the part even if it appeared in a slightly different position or orientation. Then the user would find it easier to program the robot application. The application would be more robust to variations, and the user would also find it easier to modify.

But such a solution would make the deployment much more complex and expensive. Cameras are cheap, but to integrate a vision system you need to program low-level programming interfaces (APIs) suitable only for expert developers. This means expensive and scarce external consultants and integrators.

As a practical matter, for a robot application that needs to run only for a few weeks or months, nobody is going to bother with a vision system today.

Part presentation is just one example. Users are forced at every turn to trade off ease of use against complexity and cost.

The users who need to deploy the robot are not expert programmers. They need ready-made **building blocks** that they can directly use to build robot applications.

What do we expect from these building blocks?

Besides being pre-integrated with sensors, the robot needs to be more **self-aware** about the robot's kinematic limitations. For example, when hand-guiding, the user should not have to be careful with joint limits, singular poses, and self-collisions. The robot should automatically provide force feedback to stay away from problematic configurations. This kind of smarter hand-guiding is an example of a building block available at edit-time, which can be used for defining robot tasks or declaring the geometry of objects in the work cell.

When executing motions, too, the built-in building blocks should incorporate this kind of knowledge. For example, when picking up a part that is rotationally symmetrical, like a bottle, the robot should be able to exploit the **geometric redundancy** in the task to optimize the motions. For example:

- To easily reach every corner of the workspace, the robot should optimize motions to increase the arm's reach.

- For sanding or polishing applications where the robot needs to exert pressure in certain configurations, the robot should maximize that parameter.

The user should be able to somehow indicate that the workpiece is symmetrical about a certain axis, and the robot motion building block should be able to take care of the rest.

Similarly, the robot should be made aware of objects in its work cell if needed, so that when planning its motions, it could automatically **avoid collisions.** The work cell geometry could be captured and reused in different applications in the same work cell.

Other building blocks should include common requirements in the manufacturing context. For example, knowledge of **grid patterns** is needed for filling and emptying trays and pallets. Recognizing and locating **fiducial markers** (landmarks) is needed for self-calibration and correction. **Force sensing** and feedback is needed for tasks that are based on force conditions like tugging at cables to ensure they are secure.

And finally, **real-time** reactions are needed for tasks that require coordination of motion and I/O within guaranteed deadlines, like welding or dispensing glue, where the robot's motion must be able to follow the intended operation.

Algorithms have been developed for gas metal arc welding to measure variations in the current passing through the arc, which indicate changes in the length of the arc, and use this to estimate the stick-out position relative to the seam.

If this kind of arc tracking algorithm has been implemented in the torch assembly, it can provide the estimated stick-out position as feedback to the robot, which can use this estimated difference and move the torch to maintain a constant stick-out. These robot motions must react to the feedback within milliseonds.

All these technologies and others are well understood. We know how to program applications that use them; what is missing is a way to make them accessible to generalist users so that they can use them in *their* robot applications.

These building blocks need a **framework** into which they can be integrated, so that users can easily create, modify, and debug robot applications, and these applications can handle run-time exceptions and errors without low-level coding. Such a framework will make the building blocks accessible and usable, so that the user gets a smarter robot.

Such a framework is what we're discussing in this paper.

Our solution is called a **behavior engine,** and the visual user interface it enables is called a **behavior tree.**

This framework will make robots smarter and more capable. Industrial robots today are programmed and deployed almost in the same way as the first Unimate was back in 1961. The behavior engine with its user interface promises to transform this process and make robots accessible to a wider audience.

This document will explain what the underlying principles are. Then it will show how we have adapted behaviors for programming industrial robots, and how we should develop the concept further in the future.

To explain what exactly a behavior engine is and how it works, I will need to cover some basic background about computer programs and operating systems.

It will take you about twenty minutes to read through this document slowly. But you will find that it is time well spent.

2 Tutorial

What is a Program? Sequential model

Programming students are taught that a program is a list of instructions loaded into a computer's memory. This model of a "stored-program computer" from Alan Turing's time before World War II is based on a computer with memory cells that can store numbers. Each memory cell has a numeric address, its "location."

Data and instructions are both stored in memory locations. Some instructions read data, perform mathematical operations on the data, and write the results back into memory.

> **Note:** Some computers store program instructions in the same kind of memory that stores data; other computers have different memory areas for instructions and data. The distinction is not important for our discussion. We'll assume that our computer stores both in the same memory area.

The computer starts at the first instruction, at memory location zero, executes it, then increments a "program counter" memory cell to point to the next instruction, executes that, and so on.

Some instructions change the sequential flow in which instructions are executed. For example:

- A conditional branch instruction can skip the next instruction if some condition is true, for example, if one number is smaller than another number.
- A jump instruction changes the program counter so that the computer continues execution at some different memory location.

Using these instructions, the program can loop and branch.

Instead of writing programs directly, students are taught text-based programming languages like Java or C++ that build abstractions like "functions" and "modules" on top of this basic model. They write their programs in more structured and easier-to-understand text documents, called "source code." They can use tools—compilers, linkers, interpreters— and pre-written programs called libraries, to convert the source code into a running program.

To do anything useful, a program running on a computer needs to *communicate* with other devices: other computers, disk drives, mice, keyboards, touch screens, and so on. The program you write has to at least produce some *output* so that you can see what it did, and usually it also needs to *input* some data from somewhere.

Beginning progammers are taught to output data like this:

```
printf(”hello world!”);
```

The above line in their C language source code will make the resulting program output the greeting message on the program's output display screen and then continue.

A computer has wires sticking out of it that are hooked up to external devices like displays, and the computer converts the greeting message into signals that the display shows as characters.

But there's one wrinkle that beginning students aren't told about: To produce output, the computer needs more features beyond this sequential model of programming.

A failure to communicate

Inside the computer, the wires are "mapped" to certain memory locations. The computer can encode and decode the signal information as numbers in these memory locations.

- If a program writes a number into a certain special memory location, the computer will decode the number and send a signal out of the corresponding wire. These are output signals.
- Similarly, if the program reads a number from a certain special memory location, by decoding the number, it can find out what signal is being received on the corresponding wire. These are input signals.

You can read a "spec sheet" that comes with the computer to learn about these details. Once you know these mappings, then sending and receiving signals becomes a matter of writing and reading from memory locations.

When you touch your smartphone screen, the screen sends electrical signals over wires to the computer inside. The electrical signals carry information about the location where your finger touched the screen and how fat your fingerprint is. The computer encodes this information as numbers in memory locations.

This method of reading and writing input and output signals raises a simple question: let's say a user touches her smartphone screen. How does the app know that it needs to read the special memory locations?

The app is a program. If the program reads from the memory locations too early or too late, then it will miss the signals. How can the app make sure to read while the user is touching the screen?

The real world runs at its own pace and does not care about your program. Even if the program reads the memory locations many times, it's unlikely that the input signals will arrive at the exact convenient time so that the instructions reading the memory locations encounters them. The computer needs to have some mechanism to make the app read the encoded numbers from memory.

Similarly, for sending output signals, too, there's a similar timing problem.

The computer usually needs to send multiple signals to the external devices. For example, to output the characters `hello` to the display, it needs to send five signals one after the other.

A computer has a particular *clock speed* that paces each program instruction. For example, my old Android phone can run over a billion cycles per second. If the program just wrote to the same memory location again and again using successive instructions, the computer would barely get enough time to decode the number and put the signal out on the wire, before the next instruction ran. The receiving display device would not be able to use these signals so quickly.

The program somehow needs to be able to introduce a delay between successive memory write operations, so that the output signals stay on the wire long enough for the receiving device to process them.

Dealing with the real world, it turns out, is not so simple. There are serious timing issues even when everything goes according to plan.

The sequential model of programs is incomplete. Computers are capable of two more tricks that let programs communicate with the outside world.

Program fragments, interrupts, and timers

The first trick is **interrupts.** Whenever an input signal comes in on a wire, the computer will be interrupted. It will temporarily stop running its program and do something special.

The input signal wire is mapped not only to a memory location where the signal will be encoded as a number, but also to another memory location where a program instruction should be. When interrupted, the computer will temporarily change the program counter to jump to the instruction at that memory location.

Starting at that memory location, you write a program fragment, called a *handler.* The handler is your chance to read the numbers that encode the touch and do something with the information, such as scrolling the screen or pushing a button. When the handler has finished executing, the computer will put back the original program counter to resume the main program.

The second trick is **timers.** The computer comes with an internal device called a timer. When the program writes a number into its special memory location, the device starts counting down that many clock cycles, and when the timer goes off, the computer will be interrupted just as it is for incoming signals.

In other words, the program is not just a sequence of instructions, but a set of program *fragments,* each one of which is a sequence of instructions. The programmer needs to load these fragments into the computer in such a way that a “main” program will lie from memory location zero onwards, while other fragments will start at the correct place for the “handlers” that the computer uses for the various interrupts.

Normally the computer will execute the instructions in the main program. But whenever a signal comes in or a timer goes off, the main program will be interrupted and one of the handler fragments will be executed.

Using the signal interrupts, you can be sure to receive incoming signals.

Using the timer interrupts, you can delay your output signals so that each one lasts for a certain amount of time. You send the first signal and set a timer. In the handler for the timer, you send out the second signal, then set the timer again for the third signal, and so on.

Programming a computer is thus not just about nice mathematical algorithms. Communicating with external devices, and the timing and bookkeeping chores it requires, make it more complicated, messy.

But all is not lost. Beginning students need not despair. For most typical applications, the external devices that a computer is hooked up to are fairly standardized: displays, keyboards, mice, storage drives, and networks. For these standard "peripheral" devices, you don't need to program at such a low level. Special programs called *operating systems* make it easy to use these devices from your programs.

How an operating system simplifies your life

Computers today run a modern operating system like Linux or Windows, which can run multiple programs simultaneously. Modern smartphones, too, are equipped with powerful computers and modern operating systems: Android is based on Linux, and iOS is based on several other varieties of UNIX.

On all these computers, the operating system is the main program that is always running. Your program is only a guest, to be loaded and unloaded as needed. These guest programs are referred to as "application programs" or "apps."

The operating system performs two key functions for you.

The first one is that it **manages all the hardware** and communications. All those wire mappings, special memory locations, timers, and the handler programs? You no longer need to worry about those— they are all taken over by the operating system. The operating system presents you with a set of special program fragments called *system routines* that perform useful functions.

When your program needs to input or output data, you just jump to a system routine that does it for you and then jumps back to your program. Programming students are taught about these system routines and how to use them.

Each operating system contains dozens of special routines for sending outputs and receiving inputs of various kinds. Programming languages and tools already know about these routines and make them easy to use from your high-level program "source" documents.

The second ability of the operating system is that it can run **multiple programs** at once. Running multiple programs at the same time, called *multitasking,* is important for writing powerful applications, and for fully utilizing today's fast computers.

Here's how multitasking works. Many programs are loaded into memory. One of them is running. The operating system will periodically interrupt the running program and start running another program.

The formerly-running program is "suspended." Its program counter is remembered so that it can be resumed later. The new program is allowed to run for a little while.

By rapidly switching between different programs, the operating system can effectively let all of them run simultaneously.

The app thinks it is running on its own computer. When the operating system suspends and later resumes the program, it gives it back its original program counter so that it can continue running its instructions, oblivious that it was ever suspended.

Note: Modern computers often have two, four, or more processors, each of which acts as a computer. All these computers share the same memory; the operating system can schedule programs on all the processors at once. For example, my phone has eight of them. This kind of *multiprocessing* is not important for our discussion.

Because of this ability to suspend and resume, the operating system has complete control over your program. It can make your program think it is running sequentially *even when performing input and output.*

Here is how the `printf` in your source document really works.

At the point where your app needs to send `hello world!` to the screen, it jumps to a system routine. The operating system suspends your app and remembers that it is waiting for the output to complete.

The system routine sends the entire sequence of signals to the display device with the appropriate delays. After receiving these signals, the display device sends the computer an input signal acknowledging receipt of the characters. This process takes a long time. In the meantime, several other programs have had a chance to run.

Finally, the operating system interrupts the currently-running program and wakes up your app. As far as the app is concerned, it jumped into an operating system routine to send output, and now the output is complete, the app is running again, and it is as if the app never stopped.

This operating system facility is called "blocking I/O." It allows the programmer to pretend that their program is just a sequence of operations. Receiving and sending signals is reduced to calling operating system routines. The messy reality of program fragments and handlers is, to a large extent, eliminated.

A couple of other nice facilities become available, too:

- Programs can communicate with each other by reading and writing into shared memory locations, which the operating system facilitates.
- Besides scheduling programs, the operating system can start or stop them. For example, if a program tries to access memory it is not supposed to, the operating system can stop it.

This simplified programming model with modern operating systems has been a big boost to programmer productivity— provided your computer is using the standard peripherals, and your app is doing regular business logic.

Programming robots using an operating system

For controlling a robot or other automation equipment, too, you have to write programs on the computer called the "controller" that is the brain of the robot. But programs that control robots and similar devices are not well served by standard operating systems.

For one thing, the devices that robot programs want to communicate with are not standard peripherals and are not supported by the operating system's system routines. To deal with them, your programs must perform input and output communication the old-fashioned way, with interrupts and handlers.

But even if we were to write new system routines to support these devices, it would still not be enough, because of the peculiar needs of robot applications.

A robot application needs to manipulate its physical environment and the robot's own body. The robot application needs to constantly be aware of the current state of the robot and its environment. It needs to start or stop its component programs as the state changes.

For example, when an automatic vacuum cleaner bumps into an obstacle, it might stop cleaning and start running an evasive maneuver that will move it away. After it has escaped from the obstacle, it can resume its cleaning program.

Operating systems, computer languages, and development tools are not designed for these kinds of applications. It is *possible* to run a robot application on a regular operating system; the programmer needs to use non-blocking I/O and also write more programs to command the operating system to switch programs as needed. Working around the operating system is a source of complexity for a robot application.

The same operating system that was developed to make the programmer's life easier is now getting in the way of writing a good robot application.

> **Note:** I am simplifying a bit. Not all apps are sequential. Many regular computer applications need to be robust and concurrent. There are tools and infrastructures to write such applications, for example, Erlang/OTP, which is used to develop telephony and similar distributed applications, and Node.js, used to develop concurrent servers. Those tools are usually considered advanced and specialized, and again, they are targeted toward professional programmers.

To let users program a robot, we have to provide a better infrastructure than a general-purpose operating system. A behavior engine is such an infrastructure.

Behaviors in robotics research

The term "behaviors" has been in use in robotics research for many decades. A program that makes a robot sense and react under a particular set of circumstances is called a "behavior."

Robotics researchers wanted to build **intelligent robots** that react to their environment. Instead of following a fixed procedure, these robots are opportunistic. For example, a Mars rover must be able to autonomously avoid obstacles to reach a given end location. Or, a home vacuum cleaner must be able to detect the edge of the stairs and avoid falling off. A "behavior-based" approach lets the researchers build such robust robots.

Industrial robots, of course, are different. They operate within a controlled environment and are not exploring other planets or dealing with completely unexpected situations. At the same time, the behavior-based approach developed for intelligent robots is based on principles that are valuable for industrial robots, too. These principles can be adapted to make industrial robots much smarter and easier to deploy and debug.

The term "behavior tree" comes from the video game industry, where it is a user interface paradigm that lets game designers program intelligent non-player characters (NPCs) without doing general programming. We would like to offer a similar facility to the target users of industrial robots.

Experience has shown that to write an intelligent robot application, you need to be able to schedule, start, and stop these behaviors flexibly and dynamically based on the needs of the application.

For the vacuum cleaner example, when its bump sensor has been depressed indicating it has hit an obstacle, it needs to stop its cleaning behavior and start the evasive behavior that tries to back off and reposition itself. After it has escaped from the obstacle, it needs to resume its cleaning program.

Just as programs are scheduled by an operating system, behaviors, too, need to be scheduled. The differences are that:

- A sequential program expects input and output to happen as part of its execution sequence, whereas a behavior expects to always be aware of the state of its robot and environment, and to react to any changes immediately.
- An operating system uses a scheduling policy based on time slices. For example, a typical Linux policy is "round-robin with priorities" which means that each program will get a turn to run for a certain period, and programs that have a higher priority will get more chances to run.

  In contrast, a robot application needs to control the scheduling of all its behaviors. The robot application decides which behaviors to run and which other ones to suppress.

Historically, researchers trying to build intelligent robots have written behavior programs and tried different ways to perform this kind of behavior scheduling. Various scheduling methods have advantages and disadvantages. See the history section at the end of this document for a short summary.

But if you want non-programmers to build robot applications, then the behavior programs and their scheduling policies must be **easy to configure.** This requirement is of paramount importance.

When a generalist user needs to create a robot application, we would like to give her a collection of pre-written behaviors. She should be able to configure these behaviors for the particular robot and environment. She should be able to control the scheduling policies so that the specific robot application works effectively.

Programming industrial robots with a behavior engine

Most industrial robot vendors have developed their own special-purpose languages: KAREL (FANUC), RAPID (ABB), or KRL (KUKA). The user writes a robot application as a source text document in the vendor's language. A special program called an *interpreter,* supplied by the vendor, runs on the controller. This interpreter reads the source document and executes the robot application.

A behavior engine-based programming interface works completely differently. There is no source document, language, or interpreter visible to the user; instead, a component called a behavior engine runs on the controller. The user can instantiate building blocks of various kinds using a graphical UI. These building blocks comprise the robot application.

The behavior engine is a sophisticated program that pretends to be a computer. On this pretend computer, many programs can be running at the same time. These are the behaviors.

The user interface presents to the user a powerful set of building blocks and special facilities to create and manipulate them:

- A palette of various types of "action nodes" that represent behaviors that can do useful things like commanding the robot or gripper. The user can instantiate nodes of any of these types and make them part of her application.
- Parent-child groupings of nodes for applying scheduling policies to the behaviors.
- Parameters and wizards for configuring the behaviors.
- A "blackboard" that represents the current state of the robot and its environment.
- A mapping facility to integrate signals from external devices into the blackboard and make them available to the robot application.
- A simulation facility for external devices, useful for developing and debugging the application logic.
- A "gallery" of auxiliary objects that represent reusable knowledge about the particular work cell, work pieces, or other objects. For example, if a vision system is trained to recognize a particular part or a fiducial landmark marker, then this knowledge can be represented by an object called a "snapshot."

Using these building blocks, the user can create instances of behaviors, configure them, and set their scheduling policies in an intuitive way.

In the programming UI, the robot controller shows a graphical "behavior tree" containing nodes. This tree represents the entire set of behaviors and scheduling policies that comprise the robot application.

Instead of writing text files full of configuration parameters and scheduling policies, the user manipulates and edits the graphical tree and edits the individual nodes.

The user can run the behavior tree, pause, resume, and stop and reset it to the beginning. She can save the entire tree, giving it a name. When she loads the tree again, it recreates all the behaviors and scheduling policies that were originally created.

Blackboard and behavior tree

The **blackboard** is a dictionary of variables whose values represent the current state of the robot and its environment. It is updated as the robot and its environment changes. For example, the joint angles of the arm, forces sensed at various joints (if the robot can sense forces), and the current state of the gripper, all appear as variables. Any specific geometric features in the work cell also appear as variables, for example, the pick location where the robot should pick up incoming parts.

Besides the state, the blackboard also contains **tokens** for each **resource** in the robot that has to be coordinated between different behaviors, so that, for example, two behaviors will not try to move the arm at one time. There is one token for the arm, another token for the gripper, etc.

At run time, all behaviors and scheduling policies can see the blackboard and use the current state to decide what to do.

The **action behaviors** are represented by one or more nodes in the tree. The user can instantiate nodes of any type of action behavior and add them to the tree. When the tree is running, it schedules the nodes to run according to its scheduling policies. For example:

- A **Send** node is a behavior that queues an output command to be sent, and then finishes.
- A **Set** node is a behavior that queues an update for one or more blackboard variables representing the state of the robot or its environment, and then finishes.
- A **Reserve** node is a behavior that updates the blackboard to reserve a resource like the arm or the gripper. There is a similar, inverse, **Relinquish** node that gives up a reservation.

  This token reservation mechanism allows different behaviors to coordinate the use of a common resource like the arm or the gripper.
- A **Wait** node is a behavior that runs until the robot and environment reach a desired state. For example, the desired state might be: "the gripper is in the released position AND a part is present on the conveyor." This description is called a "wait condition," and it is expressed by some combination of blackboard variable values. When this condition is satisfied, the Wait behavior finishes.
- A **Move** node represents a behavior that commands the robot to move to a target position. This node takes parameters that specify constraints on the motion: the target position, the maximum speed, and so on. The behavior keeps running until the motion completes, and then the behavior finishes.
- A **Grip** node represents a behavior that commands the gripper at the end of the arm to close. The behavior keeps running until the gripper is closed, and then finishes. A similar **Release** node does the reverse.

These behaviors can be scheduled using other nodes that represent **scheduling behaviors,** i.e., scheduling policies that apply to their child nodes. For example:

- A **sequence** node is a scheduling policy that schedules each of its children in sequence one by one, waiting for each one to finish running before starting the next one, until all of them have finished executing. Each child, when started, inherits any reserved tokens from the previous child.
- A **loop** node is a scheduling policy similar to a sequence node, except that when the last child has finished, it evaluates a condition and if it's true, it again starts the first child.
- A **parallel** node is a scheduling policy that starts multiple child behaviors and lets them all run simultaneously. A parallel node can be configured with a sub-policy to decide when to stop running. Here are some examples:

  If given a **run one** parallel policy, the parallel node stops all the children as soon as any one child finishes, and then it itself finishes.

  If given a **priority** parallel policy, it designates one of the children as a special priority child. When the priority child has finished, then it stops all the other children and itself finishes.

  Regardless of sub-policy, if given a **goal condition,** a priority node lets all its children run until that condition becomes true or until all the children finish, whichever happens first. If the goal condition becomes true first, then the parallel node stops all the children. Then the parallel behavior finishes.

These scheduling behaviors themselves can be scheduled by their own parent scheduling behaviors to be active or inactive. In this way, the behavior engine scheduler is itself hierarchical and user-configurable.

The word "behavior" can mean either a compound behavior like one of these scheduling behaviors, or an atomic behavior like one of the action behaviors above.

The user can edit the nodes to configure parameters for the behaviors, for example the target position of a Move.

At first glance, the GUI looks like you are editing a program represented in a tree form. We encourage this impression by calling the tree a "program," and offering document-like operations to open, close, edit, and save these trees.

But you are not editing a program as the word is usually understood. Instead, you are configuring *multiple* programs, or behaviors, and editing their scheduling policies.

When you load a tree, you are installing a new set of behaviors and scheduling policies on the pretend computer. When you pause and resume the tree, you are halting and resuming the pretend computer. When you reset the tree so that it starts from the beginning, you are rebooting the pretend computer.

The behaviors are all independent, so the user can run a sub-set of them for debugging, and view any blackboard variables as their values change. The user can pause execution and examine the values. The user can also modify the values and resume the program to test particular scenarios or to simulate changes to the environment.

Integrating signals into commands and state

A big difference between a behavior engine and a computer running an operating system is in how it processes input signals.

As you will recall, a program running on an operating system performs input or output by jumping to an operating system routine, which suspends the program and then encodes or decodes signals to and from numbers in memory locations. But behaviors are different. They don't get blocked because of input or output, and they don't deal directly with the concept of signals and encodings.

The behavior engine models the outside world in terms of **states** and **commands.** Behaviors deal with states and commands, not directly with input and output signals. This is a higher-level integration with the outside world than what an operating system provides.

When you integrate external equipment with the robot controller, you not only figure out how to decode the individual input signals coming from the equipment, but also what the signals mean to the state of the world as seen by the behavior engine.

The blackboard has a **mapping** facility for a user to integrate external input and output (I/O). The user can map inputs to states (variables) in the blackboard, and outputs to commands.

This will become clearer when we consider an example below.

3 Examples

Integrating a testing machine

Picture a testing machine that tests electronic boards. It looks like an old CD or cassette player. As soon as you insert an electronic board into its tray and push the tray into its slot, a yellow light comes on and the machine starts testing. A few minutes later, the test is complete. The test machine automatically ejects the tray, and another light on the machine indicates whether the board passed or failed. You can pick up the board from the tray.

A person might spend their whole day inserting boards into such a machine and dropping them into a "pass" or a "fail" bin.

The test machine supports automation through electrical signals. To automate the testing process, you hook up wires on the back of the machine that will carry the various indications as signals to the robot controller.

Let's make the example more concrete. The machine comes with a spec sheet that says two wires, SLOT and TRAY, will carry either high or low voltages indicating the machine slot and tray states. Similarly, the testing state will be carried by three other wires, TESTING, PASSED, and ERROR.
The spec sheet has a table like this, showing the meanings of the "high" and "low" signals on the five wires:

	SLOT	TRAY	TESTING	PASSED	ERROR
high	full	inserted	testing	passed	error
low	empty	ejected	not testing	failed	no error

You study this spec sheet and decide how to represent the states of the testing machine in the blackboard.

Some of these combinations will probably never happen in real life. For example, if you push the tray in without any board in it, the machine immediately ejects the tray. So, "empty" and "inserted" will never be true at the same time, and this case can be ignored.

You decide to define two blackboard variables "slot state" and "testing state" to capture the state of the machine, with three and four possible values, respectively:

- **slot state** can be full, empty, or ejected.
- **testing state** can be testing, passed, failed, or error.

Using a mapping facility in the UI, you map each possible variable value to the appropriate input signal.

There is no one right or wrong answer; you need to represent enough of the state for what the application needs.

The behavior engine converts input signals into blackboard variable assignments according to your mapping. As you integrate the testing machine with the robot controller, you can test these mappings by operating the machine and viewing the blackboard. This facility helps you verify your understanding of the spec sheet (and it might also uncover bugs in the device).

These assignments become visible to the behaviors when they run. As far as the behaviors are concerned, the blackboard somehow gets updated when they are not looking, and depending on how the state changed, some of their state conditions become true and cause behaviors to be triggered.

Input signals are meaningful to a behavior only in how they indicate a change in the state of the world. Thus, a Wait behavior is not waiting for a certain input signal; instead, it is waiting for a certain condition to become true in the state of the world.

Similarly for output signals, the external devices that need these signals are modelled as taking some "commands," corresponding to certain output signals that they need to be sent. The testing machine above does not need to take any commands from the robot. But perhaps a camera might expect a command to "take picture."

Just as you defined the states and their mappings to the input signals, you define the commands and their mappings to the output signals. Once defined, the commands can be sent using the "Send" action behavior. Again, you can test these command mappings by running a "Send" behavior and observing the result on the external equipment.

In practice, industrial robots often need to sense the world using inputs, but only rarely do they need to send commands as outputs for external equipment. That kind of orchestration usually falls to PLCs, programmable logic controllers. Still, the facility is available when it's needed.

The behavior engine can handle input and output signals not just using low-level electrical signals on wires, but also as higher-level computer-to-computer communications using TCP/IP or industrial protocols. There are thus many ways to integrate external devices with the robot application.

Next, we'll look at how to use the above testing machine to create a machine tending application.

Basic machine tending application

A worker picks up electronic boards one by one, inserts each one into the above testing machine, and drops it into a "pass" or a "fail" bin.

To automate this task, as we mentioned at the beginning of this document, the user is responsible for presenting the incoming boards to the robot.

The user constructs a mechanism that corrals incoming boards on the conveyor and holds one in a fixed position so that a robot can pick it up. When a board is ready to pick up, a part presence sensor in the mechanism sends the robot controller a high signal, and if no board is ready, it sends a low signal.

To build the robot application, the user first integrates this equipment into the behavior engine by mapping its two signals to states in the blackboard. The user defines a variable, "incoming board", and maps the possible values "available" and "none" to the high and low signals from the presence sensor.

Next, it's time to integrate the test machine. The user maps its signals to the two blackboard variables as described above.

Then, the user creates a behavior tree consisting of one long sequence: Wait for a part to be available, pick up the part, and insert it into the machine.

The user tests this sequence to make sure it properly picks up and inserts the part. As soon as the part is inserted, the machine begins to run its test, sending signals to the controller, which should show up in the blackboard as slot state "full" and testing state "testing."

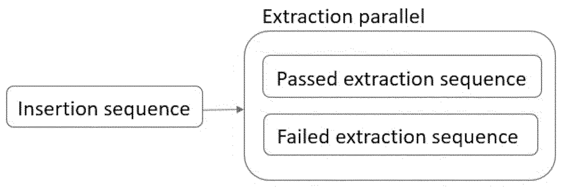
The user defines an extraction sequence for the "passed" case: wait for the testing state to become "passed," and then extract the part and drop it into the success bin.

The user can try running this sequence behavior. The robot should wait for testing to complete, and then it should start moving as soon as the board passes testing. Or, the user can directly set the variable to "passed" in the UI to force the Wait node to be satisfied.

For the "failed" case, the user can create a similar sequence by copying the "passed" sequence and modifying the wait condition to wait for "failed" instead. This new sequence can be tested by setting the same testing state variable to "failed."

The user makes both these sequences children of a "run one" parallel node. When you run the parallel node, then it will run both sequences, which will wait on their initial Wait nodes. In effect, the robot app will wait for either "passed" or "failed" to happen, and then run the appropriate moves. In either case, the parallel node will then finish, stopping the other sequence that didn't finish.

The user creates an overall sequence node and puts the insert sequence and this parallel node as children, in that order. The sequence will run like this:



The entire app can be built by putting this overall sequence inside a loop, so that it keeps testing more and more boards as they become available. The app can now be tested.

But let's make the task more realistic: since the robot is spending most of its time waiting for the test to complete, we could increase throughput if we put **multiple testers** in the cell. While one tester was testing a part, the robot could be loading or emptying other testers.

By measuring how long it takes the robot to perform the insertions and extractions, how long it takes for the tester to test a part, and how rapidly incoming parts are arriving, the user can figure out how many testers are needed to keep the robot busy.

This enhancement will make the robot application more interesting.

Machine tending: multiple testers

Let's say there are five testers, *T*1, *T*2, . . . *T*5. The user will need to integrate all of them into the blackboard, and then create five insertion and ten extraction behaviors (five passed-part extractions and five failed-part extractions).

The behavior tree allows these fifteen behaviors to be arranged together easily as children of a parallel node. No "programming" is needed.

You create five insertion sequences (for *T*1, *T*2, etc.). Each looks like this:

- Wait node, condition: *a part is available AND arm token is free AND T*1 *is empty.*
- Reserve arm token.
- Sequence of arm moves to pick up the part and insert it into *T*1, open the gripper, withdraw the arm, and relinquish the arm token.

These five insertion sequences can be made children of a parallel node that finishes as soon as any of its children finishes. The first Wait to be triggered will cause its insertion behavior to win. The other insertion behaviors will not trigger because they are unable to acquire the arm token.

You put the parallel node inside a loop. Each time through, if any of the testers are empty and a part is available, then one of the five insertions will execute.

As the insertions execute, one by one, all five testers will soon be filled and begin testing their part (depending on the exact time taken to run the tests and the insertion movements).

Just like in the basic machine tending case, you create five passed-extraction sequences that extract and drop the part into the pass bin, and five failed-extractions. Each passed-extraction will wait for *T*1 *has passed AND the arm token is free,* and so on. These are similar to the basic application case above, except that they also wait for the arm token to be free.

You make all of these sequences children of the same parallel node. One of these will trigger when its tester has passed or failed its part, and it will extract the part and drop it into the appropriate bin.

Thus, you can build a five-tester machine tending application as a loop containing one parallel node, which contains fifteen children. There is no need to create a sequence to separate the insertions from the extractions, because the combination of the arm token and the *passed* clause in the wait condition ensures that only one of the fifteen children gets to run at a time.

You can test each sequence as you build it, and you continue to test the application incrementally as you build it from the pieces.

In practice, machine tending applications can appear in great variety. It is common to devise a custom multi-gripper mechanism, for example, a T-shaped tool that contains two separate grippers, so that one gripper can extract the part from the tester and the other gripper can quickly swing and insert a second part. For such multi-gripper tools, you need separate command signals to be configured.

But all such varieties of machine tending applications can be easily handled in robot applications built using a behavior tree, with just a few simple building blocks.

Examples requiring complex building blocks

The behaviors available to instantiate in the tree can be simple or complex. The machine tending example above needed only simple behaviors.

The more complex behaviors provide more powerful building blocks for the user, so that they can accomplish a robot task with minimal effort. For example, in factory automation, you often need to pick or place items in a pallet or tray. For this, the behavior tree provides a complex scheduling behavior, a **pattern.**

The main configuration parameters for a pattern are the geometric extent of the tray or pallet and the number of equally-spaced rows and columns. The pattern behavior also takes a few geometric parameters. The most basic parameter is an "approach offset," which is a certain translation to be applied to the gripping position of each part in the pallet. The idea is that in order to pick up a part at a certain grid location, the robot should first move to a location a few inches above it, and then approach vertically for the pick.

The pattern behavior will repeatedly invoke a set of moves for each item in the pallet. The "approach offset" is a special case. The general case is to supply sequences of Move behaviors to execute, which will read the item location from a variable on the blackboard and use it as a parameter. For each item, the pattern behavior will set the value of this variable to the grip location of the item, and then execute these sequences.

The pattern behavior is not just a simple loop that translates the geometry of the pick location. Recall our discussion of "self-aware kinematics" at the beginning of this document.

If you blindly executed moves from each item that simply translated the pick coordinate frame used for the previous item, then some of those moves will succeed, but others will tend to fail when the robot stretches too far in some joints. The pattern behavior incorporates knowledge of the robot kinematics and plans a pick trajectory that is likely to succeed even if the tray is large with respect to the reach of the robot.

The self-awareness also extends to task redundancy. If the items to be picked are rotationally symmetrical like round bottles, then the pattern node can use the extra degree of freedom to optimize the trajectory of each move.

This is an example of a smart building block that is easy to use but that hides inside it many details and decisions that the user does not need to make.

To integrate the robot with external systems, the behavior engine comes with more building blocks. For example, say we wanted to address the limitation of a "brittle" robot application created by precise part presentation. Say we decided to incorporate a **camera** with a **vision system** on the arm, and avoid having to create a precise part presentation system.

If you move the arm so that the camera is pointed down on to a table, and then command the camera to take a picture, its vision system can run its vision algorithms to recognize a part lying on the table. Alternatively, the camera can be mounted at a fixed location in the work cell.

In either case, the vision system can send a message to the robot controller, either saying that it failed to recognize a part, or that it succeeded and one is visible at a certain location and orientation.

But to perform this recognition, the vision system first needs to be trained for the parts you need to pick up.

You can train this vision system to recognize a part by using a wizard in the controller UI to outline the part and define the reference coordinate location of the part. You need to regulate the amount and direction of light incident on the table and adjust the imaging parameters like brightness and contrast. You need to help the vision system locate the part you need by outlining it on a picture the camera has taken and indicating the reference coordinate frame of the part.

Once the teaching has been completed, the UI shows a new auxiliary "snapshot" object in the gallery. This object encapsulates the configuration of the vision system that enables it to recognize the particular kind of part, for a restricted set of lighting conditions in the work cell.

This snapshot object can be used as a parameter in a node called a Vision Pick node. When the Vision Pick behavior runs, it uses the snapshot object to make the vision system recognize the same part again. If successful, the vision system provides the part's location and orientation, which the snapshot object writes to variables on the blackboard. The Vision Pick behavior uses these variables to schedule and execute a series of move and grip behaviors.

The Vision Pick behavior is an example of a complex, predefined behavior that uses an auxiliary "snapshot" object to maintain and use the knowledge that a vision system has after it has been trained to recognize a particular part. From the user's point of view, the "snapshot" object makes it easy to configure the Vision Pick. It separates the process of training the vision system from the process of programming the robot's task.

Snapshots are an example of auxiliary objects relevant to vision systems. These objects are presented to the user in a "snapshot gallery."

A **gallery** contains lists of auxiliary objects of various types needed to use special capabilities for building the robot application. For example, grippers, vision systems, and other external equipment might require special knowledge to be captured, saved, and reused. Auxiliary objects in the gallery let the user create and use this knowledge. These will also be explained below.

These kinds of auxiliary objects are reusable abstractions. When you instantiate them, they:

- Receive information from external equipment;
- Maintain their own state in the blackboard; and
- Can send commands to external equipment.

These objects provide the behavior engine with a home to encapsulate integrations with external systems. The behavior engine has a standard place where it can be extended to perform specialized communciation of this type, which we will discuss later in the design section.

From the user's point of view, these objects also provide a convenient handle for any work cell-specific knowledge accumulated for the external systems. These objects are saved persistently so that they can be reused along with robot applications.

The behaviors and state mappings available can also exploit any unique capabilities of the robot arm. For example, if the arm is force-sensitive, then special behaviors that can apply a certain amount of force in various directions can be offered, and the user can map the forces being sensed into blackboard variables, where they can be used in wait-conditions. You can create robot tasks that stop when the arm encounters a certain force.

The powerful built-in behaviors that a behavior tree incorporates give the robot application programmer unparalleled power. Its behavior tree UI is well matched with the building blocks in the behavior engine.

Summary

I have described how a behavior engine running on a robot controller forms a framework for several kinds of building blocks. Using these building blocks, a behavior tree and its UI supports the entire **robot deployment workflow.** A generalist user can quickly integrate the robot into its work cell and then build, debug, and run robot applications.

We have walked through examples of a machine tending application built using this framework and a collection of simple behaviors with a simple set of building blocks. We have also briefly described other applications that will require more complex building blocks.

Any behavior, particularly a complex one, can have integration-time features, edit-time features, run-time features, and debug-time features. To help the user to use these features, the framework offers auxiliary objects in the "gallery" so that the user can store any knowledge the behavior needs, as we saw in the snapshot auxiliary object above.

The next section of this document is a design discussion. It goes into a little more detail about some practical aspects of the behavior engine architecture. It also talks about how to extend the behavior engine to add new building blocks.

4 Design

Tick cycle and phases

Conceptually, the behavior engine interleaves the blackboard updates and the behavior executions during successive phases in a "tick cycle."
The four phases of each cycle of the behavior engine are:

- Take all queued input signals and perform blackboard variable assignments for them. Mark all the wait-conditions that have become true.
- Take all the behaviors that are allowed to run, including the ones whose wait conditions have become true, and run them for one tick cycle. Mark all the behaviors that have finished.

  During this cycle, some of the behaviors might issue commands or update blackboard variables. Queue these.
- Take all queued commands from all the behaviors and dispatch them to the outside world as output signals.
- Take all queued blackboard updates from all the behaviors and perform blackboard assignments.

Each behavior is not a sequential program but a set of per-tick fragments. The tick phases make sure that the behaviors all see a consistent view of the state of the world during one tick cycle.

The tick phases also provide logical places for extending and integrating the behavior engine with APIs, signals, and commands for external equipment.

- Any API calls that the behavior engine makes to external equipment are a result of queued commands from the behaviors; these calls happen during that phase of the tick cycle.
- Any information coming from external equipment is treated similarly to input signals. It is queued and processed during the appropriate phase of the tick cycle.

All these external APIs and other communication methods are made available to the user to configure using the built-in mapping facility. This means that when integrating external equipment like vision systems into the behavior engine, you have to provide facilities so that the user can map the configuration of the external system to blackboard variables (for incoming updates to the state) and to commands (for outgoing messages to the external systems).

Sometimes this configuration is complex and creates specific knowledge that needs to be persisted. That's what the gallery objects are for.

Some considerations for this ticking method:

- How frequently should the engine tick? It depends on the speed of the controller computer. Too frequent a tick rate will drag down the controller CPU. A few dozen Hz is typical.

- Is the ticking really necessary? We should try to use Node.js or Erlang to implement behaviors that are event-based and not tick-based. This will be a more natural way to write concurrent behaviors and would avoid the inefficient ticking.

  But if we do that, we will have to employ additional mechanisms to be able to start, stop, and trigger the behaviors at will. We will also need additional mechanisms to make sure that the behaviors see a consistent view of the state of the world.

  We would also have to make sure that the user still gets appropriate mapping facilities to properly interleave incoming and outgoing communications with the behavior execution.

  Also see below for what we need for simulation.

It is harder to program a fragmented, per-tick program for a behavior, than to just write a sequential program end to end. Fortunately, this burden falls only on the expert programmers who create behaviors, not on the generalist user who uses the behaviors to build applications.

Simulation

As we have said before, a behavior engine lets the user modify and test the robot application incrementally during development and during modification. Some portions of the robot application need to be tested before other portions have been developed. For testing, the user also needs to be able to **simulate** things in the work cell.

The user can write behavior trees to simulate different parts of the environment including external equipment. In effect, the behavior engine can pretend to be not just one computer running the behaviors that comprise the robot application, but also *multiple other computers,* each running a part of the environment.

The environment-simulating trees run in each tick cycle alternating with the robot behavior tree.

During the phase when the behavior engine normally processes input signals, the environment-simulating trees get a chance to update the state of the robot blackboard. Similarly, when the robot behaviors issue commands to external equipment, then during the command phase, these appear as inputs to the environment-simulating trees, where a kind of “Wait” node is available to process them.

In this way, the robot behaviors are completely independent of whether their environment is real or simulated.

This feature is of very high value. The ability to simulate the environment while developing the program, and to test individual behaviors and groups of behaviors, becomes more and more important as we support bigger and more powerful robots and complex work cells.

Real-time processing

The best way to implement the behavior engine has been by using libraries designed to make it easy to emulate multiple concurrently running programs, for example Node.js or Erlang/OTP. We have found Node.js to be a good platform to implement behavior engines.

This means that the behaviors running in this engine are not real-time.

Of course, some behaviors need to react to the world in real time. For example, the "Move" behavior needs to control the arm by exchanging input and output signals with it at a high rate, something like 1 kHz. To solve this problem, we run the Move logic inside a separate component, not the behavior engine.

The bulk of the Move behavior is implemented in a *real-time component.* The Move behavior uses a remote procedure call to start the high-frequency control loop that drives the arm motion. When the motion completes, the behavior gets a call and finishes.

The same logic applies to I/O that needs to react to strict deadlines, and to combinations of motions and I/O that need to trigger each other within strict time limits. Any behaviors that need real-time processing must be implemented within a real-time component.

An explanation of what we mean by "real-time component" and why it is significant, is in an appendix section.

Extensions and SDKs

How to build a partner ecosystem that can extend the robot system to add their own equipment and expertise? How to enable third-party equipment vendors to integrate easily with the robot? How to enable value-added resellers (VARs) to offer their own, custom apps using our robot system as a platform?

Clearly, we must offer a way to extend our building blocks, and also to add new building blocks.

This question goes to the heart of what we are offering to the user. From the user's point of view, the *abstractions* that the existing building blocks are built upon should be reused as much as possible, instead of introducing new abstractions.

Different kinds of system extensions need their own mechanism. Take grippers. Say our existing gripper abstraction is a simple one that only supports "open" and "close" commands, and a few different states ("open," "closed," "opening," and "closing," say).
Here are some considerations in supporting new grippers:

- If the new gripper also conforms to this simple abstraction, then a simple API can be provided to the third-party vendor to map their unique I/O signals to the same states and commands that the system already supports for grippers. This is a very quick integration that reuses all the existing objects and UI. We could even offer a built-in UI wizard that lets the user read the spec sheet for the new gripper and map the I/Os by herself.

- If an external gripper offers more capabilities, for example, if it provides feedback on the width of the jaw, or if it can detect whether a part has been gripped or not, then really we need to understand these new capabilities and define a more powerful abstraction to cover these. We need to create the corresponding building blocks and UI to support this abstraction. Once the new abstraction is available in the system, we have reduced this case to the previous case, and can now support this vendor's and other vendors' grippers.
- Another option for supporting the new abstraction is to open up a wider API for third-party vendors to define their own abstractions and UI. To make this new abstraction work with our system, we will need to expose interfaces to our blackboard, I/O mapping system, and perhaps other pieces, too. The external vendors might need to define new auxiliary objects in the gallery. They would certainly need to be able to extend the UI.

  This option is a much more extensive SDK and requires much more work. It has many more moving parts, some supplied by us, others by the third-party vendor, that might break in the field. These bring with them a corresponding customer support burden and the possibility of poor customer satisfaction.
- Meanwhile, regardless of which option we select for supporting the new abstraction, we can still offer "downward-compatible" support for the new gripper. That is, the third-party vendor can use the existing, simple, open-close abstraction, and map the more complex gripper to this simpler abstraction. This will allow users to use their gripper, although not all the features of it. So, the choice for the user is more subtle than "can I or can't I use the new gripper?" The choice is more like, "can I use the extra features of the new gripper?" In some cases, users might be perfectly happy to use the common sub-set that maps to the existing abstraction.

The key point to remember is that having an extension point implies a particular abstraction already existing in the base system.

An "extension point" happens when a third-party vendor's system can be mapped to an existing abstraction. This is the best way to extend the system, because it reduces the cognitive load on the user, and at the same time saves integration time, reusing all the existing building blocks, UI, and code that have already been developed and tested.

If an external system offers more powerful features than the base system does, and it requires a change to the abstraction, then this implies a more extensive API that will be more difficult for partners to integrate and will have more opportunities to break in the field.

Some kinds of extensions are difficult to add. For example, take the seam-tracking algorithms developed for gas metal arc welding. In order to react to external I/O from this algorithm and use it to modify the motion path, you need to somehow deploy logic into the real-time component. This requires careful design of an API. See also an experiment in the Appendix section on the real-time core, for hints on how this might be solved in the future.

5 Appendix

Appendix: Background on real-time processing

A program running on a robot controller needs to react to the world as it changes. Such a program needs to be started and stopped based on the needs of the robot application, and while it is running, it needs to attend to incoming and outgoing I/O signals immediately.

A multi-tasking operating system complicates the writing of a robot application. If a program can get interrupted and re-awakened after an arbitrary amount of time at the whim of an operating system, then it is difficult or impossible to program a robot using such a program.

To support "real-time" applications, operating systems do provide special facilities to guarantee time slices to particular programs. These facilities need to be enabled when installing and configuring the operating system, and the programs need to be written in such a way that they will complete a meaningful amount of work within a time slice.

Using these time slices is a good thing, but it still doesn't give the programmer full control over the timing of the programs. Far from it.

Over the decades, programmers and their development tools and libraries have learned to depend on many facilities that operating systems offer. For example, when they need to use more memory, programs usually use libraries that call the operating system to do it on their behalf.

But these facilities were all developed for regular applications. They work by suspending the calling program for arbitrary amounts of time.

For real-time programs to guarantee that they will finish their work within a time slice, they have to give up many of these facilities and resort to doing the low-level work themselves. This kind of programming tends to get intricate even for full-time programmers; it is simply not a suitable activity for someone who is just trying to deploy a robot.

Appendix: Behavior engine as a real-time core

Some kinds of action behaviors have to be initiated and monitored remotely. But this method could be greatly improved if we made the engine itself real-time.

We experimented with writing a behavior engine "core" that implements hierarchical scheduling, a blackboard, and a real-time ticking heartbeat. This could be embedded in a larger behavior engine. The other components of such an engine would be:

(a) A Node.js-based "shell" that helps the user view and manage the behaviors and scheduling inside the core. The shell serves the UI and maintains a symbol table.

(b) An interface for other programs to feed inputs to the core and take outputs out of the core. This interface could be a real-time method that would allow other real-time components to be driven directly from the engine core. Maybe a real-time protocol running on UNIX-domain socket pair, or shared memory.

(c) A management stream that allows the "shell" to communicate with the core.

In this kind of architecture, the core would be the "run-time" behavior engine. The "shell" part would be needed only for development and debugging. Once a robot program was developed, it would be possible to remove the management shell and UI, and just leave the core running on the robot, using a much smaller and energy-efficient computer. Such an architecture could be used to develop behavior-tree-like controls for all kinds of automation equipment.

Appendix: Scheduling options for behaviors

The heart of the behavior engine is the ability to start, stop, and schedule the behaviors. Some scheduling options are listed below, to suggest enhancements in this area:

- In Rodney Brooks's original subsumption architecture, he tried to make behaviors communicate with each other to suppress or subsume each other. You specified these policies using a language which his compiler would convert into a program for an 8-bit microcontroller.
- In Damian Isla's game engine used for Halo 2, the behavior engine was a program that ran about a few dozen times a second. Each behavior played a non-player character (NPC). The game designer specified how the NPC should behave. At every tick, the behavior engine made all of them progress for a short while.
- In typical behavior trees used for controlling robot arms, the behavior engine is implemented on Node.js as a ticking program, similar to an animation loop in a web browser.

  The individual behaviors are then Javascript programs running as one or more thread fragments in each Node.js event loop. Each thread fragment runs completely during each tick, and the ticking gets delayed if it takes too long. The developers have to break the behavior up into thread fragments so that each fragment runs within a very short time.

Appendix: A history of behaviors

A short history of the idea of behaviors in robotics and gaming.

- In 1987, Prof. Rodney Brooks at MIT proposed an architecture he called the "subsumption" architecture for building reactive, intelligent robots. The software consisted of a bunch of programs called "behaviors." The robot's computer executed all the behaviors at once, but some of them were temporarily suppressed or "subsumed" while others did their work.
  The original subsumption architecture was implemented in a compiler that read a description of your robot's behaviors and generated code for a microcontroller (no operating system was running on it). But it was difficult for a robot programmer to decide which behaviors to suppress when. The scheduling method was not easy to modify.

- In 2002, at Brooks's company iRobot, engineers wrote behaviors as individual programs and scheduled them using hand-written policies. The Roomba vacuum cleaner automatically explored and cleaned a room while avoiding obstacles. iRobot says it has sold over twenty million Roombas to date.
- In 2005, for creating artificial agents inside computer games, my colleague Damian Isla built a graphical tree so that the game designer could easily schedule different behaviors for non-player characters like guards patrolling a castle wall.
- Later in 2015 at Rethink Robotics, we used the same idea to build Intera for the Sawyer 7-DOF cobot.