By Vincent Berenz and Stefan Schaal

Reactive Programming for Orchestrating Robotic Behavior

The Playful Software Platform

For many service robots, reactivity to changes in their surroundings is a must. However, developing software suitable for dynamic environments is difficult. Existing robotic middleware allows engineers to design behavior graphs by organizing communication between components. But because these graphs are structurally inflexible, they hardly support the development of complex reactive behavior. To address this limitation, we propose Playful, a software platform that applies reactive programming to the specification of robotic behavior.

Playful's front end is a scripting language that is simple (only five keywords) yet results in the runtime coordinated activation and deactivation of an arbitrary number of higher-level sensory–motor couplings. When using Playful, developers describe actions on various levels of abstraction via behavior trees. During runtime, an underlying engine applies a mixture of logical constructs to obtain the desired behavior. These constructs include conditional ruling, dynamic prioritization based on resource management, and finite state machines. We have successfully used Playful to program an upper-torso humanoid manipulator to perform lively interaction with any human approaching it.

Robotic Behavior Specification

High-level behavior specification is the offline setup of information exchange between software components so that a robot autonomously performs a desired behavior online. Existing robotic middleware tackles this by enabling the creation of behavior graphs that arrange communication between components.

Digital Object Identifier 10.1109/MRA.2018.2803168
Date of publication: 10 May 2018

1070-9932/18©2018IEEE

A limitation of such behavior graphs is their inflexibility. Typically, the structure of the graph is fixed. Therefore, runtime changes in the robot's behavior can occur only through changes in the states of the nodes. This results in increased node-code complexity. Consequently, the robotic behavior is no longer expressed solely by the structure of the graph, i.e., a profound knowledge of the code embedded with the nodes is required to relate the graph to the behavior displayed by the robot. As the size of the graph grows and the complexity of the nodes increases, this relationship becomes intractable. Because the code encapsulated by the nodes does not necessarily refer exclusively to a functionality reusable across behaviors, but may also include logic related to the specifics of the target behavior, there is a direct negative impact on code reusability. And because the logic of the behavior is shared between the structure of the graph and the code encapsulated by the nodes, the efforts required to modify or extend the existing graphs is considerable.

Playful enables the encoding of robotic behavior via behavior trees that support runtime structural modifications, such as 1) changes in the activation status of nodes, 2) online creation of branches, and 3) online setup of information flows between nodes. A first consequence of these runtime structural modifications is that these trees may express reactive behaviors. For example, Playful is suitable for implementing the sensory–motor couplings required to grasp a moving object while looking at it. The second consequence is that the logic of the robotic behaviors designed via Playful no longer needs to be partially delegated to nodes; it may be fully expressed by the tree. As exemplified in the "Complex Behaviors" section, this has a positive effect in terms of code reuse and behavior refactoring.

When using Playful, developers design reactive trees whose structure will change during runtime. To simplify the definition of such behavior trees, Playful supports, as a front end for developers, a novel dedicated scripting language based on the reactive programming paradigm. We chose this paradigm because it allows for specifying logic via expressive statements relatively close to natural language. At startup, the script provided by the user is interpreted, the corresponding behavior tree is created, and Playful's underlying engine ensures that the specified logic is applied and the sensory–motor couplings are suitably coordinated. Reactive programming, while popular for creating graphical interfaces, is rarely used for robotics (see the "Related Work" section). To our knowledge, the Playful scripting language is the first application of such programming to the definition of behavior trees.

This article shows that applying reactive programming to the design of dynamic trees amounts to a high-level scripting language that allows the design of reactive behavior trees using descriptive statements. In the "Complex Behaviors" section, we show that Playful supports the creation of complex dynamic behaviors via the association of reusable components of modest size. We also show how these behaviors can be significantly extended via the addition of only a few lines of script. This work focuses on Playful as a convenient tool for creating first-order reactive and lively behaviors; interfacing with higher-order planning and reasoning software has not yet been validated. In the "Future Work" section, we discuss how such interfacing could be done and report on related early results.

A dedicated website for download and tutorials is available [15]. A support video [16] for this article as well as an overview video [17] can be found online.

Related Work

Reactive programming is an asynchronous programming paradigm concerned with data streams and the propagation of change. Elliot and Hudak first formulated it [1], and it has been used mostly for the development of graphical user interfaces. The researchers in [2] and [3] proposed its use for robot control in the form of languages embedded in Haskell. These languages handle both streams of continuous values and discrete events without regard for their rates, allowing the creation of commands at higher levels of abstraction. While the latter languages focus on lower-level control, Playful applies reactive programming to another domain of robotics: that of orchestrating behavior trees. Our software platform monitors the activation of higher-level modules communicating with a middleware—e.g., the Robot Operating System (ROS). In this context, a combination of streams is solved by the middleware, as with the use of publishers and subscribers. Playful operates with different concepts (e.g., it does not use monades and signals) and provides complementary constructs more suitable for describing overall behavior (e.g., online activation and modification of behavior trees).

Three-layer architectures, and notably modern hybrid architectures, are robotic software creations supporting the integration of deliberation and reactive execution. Executive control often refers to the middle component of three-layer architectures, for which the concept of task is central [4]. A task refers to the execution of a potentially complex action characterized by a start and an end. Typically, a task's performance can be monitored, and its success or failure evaluated on completion. While both approaches have been designed for supporting reactive execution, they differ drastically, as the notion of task does not exist in Playful. The latter's behaviors are encoded in tree structures defining mappings between the state of the world and module activation patterns. Thus, Playful and three-layer architectures are suitable for different application domains. Executive control, as implemented in three-layer architectures, is suitable for applying sequences of actions generated online by planners. In this article, on the other hand, we present Playful only for rapid prototyping of reactive behaviors and its possible interfacing with other higher-level decision-making systems not yet tested and only briefly discussed in the "Future Work" section.

Following Target-Drive-Means (TDM), a software product for declarative specification of high-level robotic behavior

described in [5], Playful relies on a light robotic architecture implementing a simple shared memory and resource management system. It also supports runtime instantiation and deletion of modules via the trigger mechanism [6]. The platform is inspired by TDM and implements a similar declarative programming paradigm, one improved with an original scripting language as well as support for state machines and tree structures.

In many respects, Playful is similar to cognitive robot abstract machine (CRAM) plan language (CPL), with which it shares some core features: a dedicated language and a synchronizing of parallel behaviors, with support for reactive execution targeting real, physical robots [7]. The two software items nevertheless rely on fundamentally different approaches. CPL is a plan language that is explicitly goal oriented. On the other hand, Playful does not rely on planning, as it is based on a mixture of logic, with no explicit declaration of goals. In CRAM, CPL has also been tightly integrated into knowledge processing for robots, which provides first-order knowledge representation and reasoning. In the "Future Work" section, we discuss the possibility of interfacing Playful with reasoning-enabled software via evaluation.

Playful encodes behaviors using tree structures, similar to some frameworks used for robot soccer [8]. Our platform differs from these by the mixture of logic it implements: a combination of conditional ruling, dynamic prioritization based on resource management, and finite state machines.

Playful's scripting language is used to encapsulate subbehaviors as branches of a behavior tree. Because of the reactive programming paradigm applied, these branches may run in parallel, and Playful's syntax is particularly suitable for specifying parallel actions. The work in [9] introduced scripting commands for parallel execution as nonblocking commands in URBI and have since been implemented in robotic operating systems as post commands (Softbank Robotics NaoQi) or via monitoring libraries, such as actionLib (the ROS monitoring library). But in those, parallel commands are created in the context of imperative scripts, where sequential executions are monitored by classic statements. In contrast, Playful uses parallel commands in the context of reactive programming. Our platform does not use any of the statements used by imperative programming languages, such as `while`, `if`, or `for`. It is based on novel statements that are suitable for reactive programming: `whenever`, `targeting`, `priority of`, and `switch to`. `If` is also a keyword used by Playful, but is adapted to reactive programming.

At runtime, the Playful engine reevaluates the behavior tree at a fixed frequency, leading to activation and/or deactivation of its branches. This differs from the approaches in which logic is encoded in edges, and module activation relies on traversing the tree during operation [10].

The leaf nodes of Playful's tree are Python modules that run their own thread, control access to resources, and manage start and stop calls. Our software platform provides an application programming interface (API) for the creation of new modules, and is in this respect similar to PyRobot [11].

Reactive Programming

The reactive programming paradigm, as implemented by Playful, declares the conditional activation of `a`

```
a whenever e
```

This declarative statement calls for the activation of `a` whenever `e` is evaluated to true, independently of the execution status of the rest of the program. When using reactive programming, `e` is continuously reevaluated, and the activation status of `a` is updated accordingly. By contrast, using an imperative programming paradigm, conditional activation of a routine `a` is monitored via instructions such as

```
if(e){
   a;
}
```

Based on such an instruction, `a` will be activated if `e` is true at the moment the expression is evaluated. The difference between imperative programming and reactive programming is striking. Typically, imperative scripts focus on sequential execution, monitored via classic statements such as `while`, `for`, and `if`. In the following imperative example, a_2 may activate only after a_1:

```
if(e1){
   a1;
}
if(e2){
   a2;
}
```

In contrast, reactive programming results in a purely declarative approach in which the commands in the script do not presume any execution order. Activations of `a` and `b` are asynchronously monitored, and the order of these two statements has no importance:

```
a1 whenever e1
a2 whenever e2
```

As the evaluation status of e_1 and e_2 change, the program above results in four activation patterns: 1) neither a_1 nor a_2 is activated, 2) only a_1 is activated, 3) only a_2 is activated, and 4) both a_1 and a_2 are activated. If a_1 and a_2 are modules communicating with a robotic middleware, these changes in activation patterns will shape the behavior of the robot. Reactive programming therefore naturally supports mapping from the world state to patterns of module activation. In addition, it allows the natural expression of behaviors that may overlap in time.

Playful Scripting Language

Conditional Activation

In the previous section, we already presented a reactive programming statement: `a` is called a *node* and `e` an *evaluation*. In Playful, evaluations are arbitrary functions programmed using the Python scripting language.

Online Prioritization

Statements may be prioritized as follows:

```
a1 whenever e1
a2 whenever e2
```

When both e_1 and e_2 evaluate to true, this program commands a_1 and a_2 to be simultaneously activated. If a_1 and a_2 control the same resource (e.g., the same robotic joint), they may conflict. The keyword `priority of` can be used to specify which node has prioritized access to the resource.

```
a1 whenever e1, priority of f1
a2 whenever e2, priority of f2
```

If f_1 evaluates to a number higher than f_2, a_1 is activated and a_2 is inactivated. In case a_1 and a_2 do not compete for access to the same resource, the `priority of` keyword has no effect.

State Machine

Playful supports state machines. For example,

```
a1 switch to a2 if e1
a2 switch to a1 if e2
```

Nodes a_1 and a_2 may never activate simultaneously, and they will alternatively activate depending on the runtime evaluation of e_1 and e_2. At startup, if e_1 evaluates to true, a_1 will activate. Using state machines in Playful scripts allows the platform to deviate from a purely reactive programming paradigm, as the order statement of the script is of importance.

Behavior Tree

A node can be declared as a list of nodes, themselves associated with evaluations. For example,

```
a1 whenever e1
a1:
a1a whenever e1a, priority of f1a
a1b whenever e1b, priority of f1b
```

This defines a tree in which a_1 is a node, and a_{1a} and a_{1b} its child nodes. Nodes a_{1a} and a_{1b} may be leaf nodes, or they may be declared a list of nodes, resulting in behavior trees of arbitrary depth. Leaf nodes are instances of user-developed Python objects. The code they encapsulate may use the API provided by the robotic middleware to receive sensor data or send control commands.

Configuration

Some nodes and some evaluations may be configurable. Configuration values can be provided:

```
a whenever e | key = value
```

If the code encapsulated by `a` and/or `e` accepts `key` as an argument, it will be configured accordingly.

Targeting

The keyword `targeting` is a multifaceted term that simultaneously commands the creation of new branches at runtime and enables continuous configuration of the created branches. Through `targeting`, developers can attach branches to a type of perception. For example, setting a look node to target objects of ball type will set the robot to look at detected balls. A new branch will be created for each detected ball. Each new branch will continuously reconfigure itself during runtime, according to the changing properties of the associated ball.

The targeting system relies on Playful's shared memory. This memory supports data exchange via the pull/push paradigm: code encapsulated by leaf nodes may push or pull schemes to or from the memory. Schemes are instances of arbitrary objects consisting of a set of properties. When a scheme is pushed to the memory for the first time, the memory attributes a key to it. The Python API provides a pull function. The code encapsulated in nodes can pass this memory key to this function to extract from the memory the information encapsulated by the scheme. The `targeting` keyword is used to relate a tree branch to a specific memory key, as in

```
a1 |out = s
targeting s: a2 whenever e2, priority
of f2
```

During runtime, node a_1 pushes schemes of type `s` to the memory. When a new memory key related to a scheme of type `s` is created, `targeting` commands

- the creation of a new instance of a_2 and its related evaluations; if a_2 is declared as a list of child nodes, these are also instantiated, creating a full tree branch
- that the memory key is passed as an implicit argument to all nodes and evaluations of the newly created branch.

Schemes may also be deleted from the memory, and the Playful Python API provides a function for doing so. In such a case, the branches relating to the memory key of the deleted scheme are removed from the tree. By default, the Playful engine never deletes schemes, even when the corresponding objects are not currently perceived, as it allows the system to act on objects that are not currently in the field of vision.

Passing a memory key as an argument to a branch is in many respects similar to passing a pointer to a function in the C++ programming language. During runtime, the code related to the targeted branches may use the key to query the memory and use the extracted information for self-reconfiguration. As schemes may be pushed and pulled to and from the memory at high frequencies, this may result in efficient sensory–motor couplings.

In the background, data exchange relies exclusively on pull/push calls. But in effect, the `targeting` keyword results in a flow of information from the memory to all nodes and evaluations corresponding to the targeted branch. Via the `targeting` keyword, these information flows are only implied by the programmer and implemented transparently by the system at runtime.

Targeting is Playful's less intuitive concept. It is based on the triggering system, which we previously implemented in TDM [5], here extended to behavior trees. For details related to shared memory, the pull/push data exchange paradigm, and encapsulation of data into schemes, we invite the reader to consult [6]. We provide concrete instances in the "Examples" section.

Formal Syntax

Playful's syntax is formally presented in Algorithm 1. Items in parentheses refer to content that has to be specified by the developer, and items in brackets indicate optional content. In bold are the supported statement keywords. Evaluations are functions that should either return a Boolean (when associated with `whenever` or `if`) or a float (when associated with `priority of`). A node is defined as a list of nodes associated with keywords for dynamic reconfiguration and exchange of memory keys (`targeting`), evaluations for setting rules of activation (`whenever, switch to`), and rules of prioritization (`priority of`). Leaf nodes are instances of Python objects, which interact with the middleware. Higher levels of the tree encode the logic that monitors the leaf nodes' activation status. They do so through evaluations and resource management (e.g., forbidding two leaf nodes from simultaneously controlling the same robot joints). If no evaluation is used and no resource conflict is detected, each leaf node runs continuously, concurrent with all of the other running leaf nodes.

The activation status of a node propagates to all of the nodes of its subtree, including the leaf nodes. If a given node is deactivated, all of the nodes in its subtrees are deactivated. If a node is activated, the monitoring of the activation status of its underlying nodes is delegated to their evaluations. Thus, a change in the activation status of a node in the tree implies a change in the pattern of activation of its underlying leaf nodes. And because leaf nodes interface with the middleware, a change in the patterns of activation of the leaf nodes shapes the behavior performed by the robot.

Python for Evaluations and Leaf Nodes

Evaluations and leaf nodes are to be programmed using the Python scripting language. Evaluations can be any arbitrary Python statement that evaluates to either a Boolean (when associated with the keywords `whenever` or `if`) or a float (when connected with `priority to`). The Playful interpreter also supports simple logic and arithmetic. For example, an evaluation may consist of the negation of a Python function (keyword `not`) or the addition of a function with a number. Leaf nodes correspond to Python objects, which must implement the specific interface required for communication with the Playful engine (e.g., it must implement a `run` function). Leaf nodes may be considered our platform's primitives, and the behavior tree the encoding of the logic used to orchestrate their activation during runtime. Each leaf node is spawned in a dedicated thread running at its dedicated frequency. Playful provides an API for receiving start or stop commands from the engine, accessing the shared memory, and reserving or releasing resources.

Playful's Back End

The scripting language is the front end of the Playful software platform, which is completed by the engine back end. The latter consists of two parts. First, there is the interpreter, which creates the behavior tree and relates nodes and evaluations to their corresponding Python code. Second, there is the engine, which at runtime

- instantiates new branches when the memory is updated with a new key, according to the usage of the `targeting` keyword
- calls evaluations code, i.e., branches that are conditionally evaluated as false (`whenever` keyword) or as not active (`switch to` keyword) are sent signals to deactivate; other nodes are sent signals to activate

Algorithm 1:

```
(node name):
[
  [targeting (memory key):](node name)[, whenever (evaluation)][, priority of
    (evaluation) if (evaluation), […]][, switch to (node name)
    if (evaluation), […]]
  […]
]
```

Algorithm 2:

```
ball_chase, switch to sit if   battery_low
sit, switch to ball_chase if battery_high

ball_chase:
    ball_detection | out=ball
    targeting ball: following, whenever time_ago<2, priority of 2
    searching, priority of 1

    following:
      looking
      walk_to, whenever far

      looking:
          look_at, whenever time_ago<1, priority of 2
          head_search, priority of 1

    searching:
      turning
      head_search
```

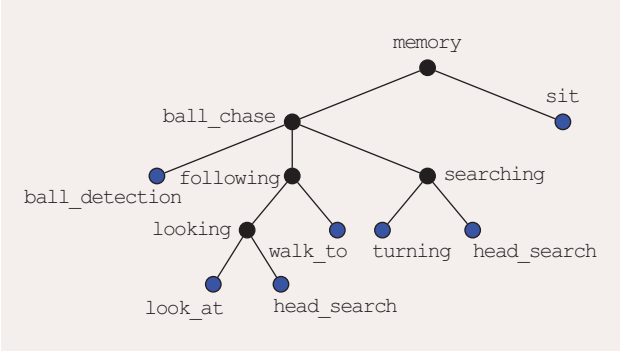

Figure 1. A tree representation of the script presented in Algorithm 2. Nodes in blue correspond to leaf nodes.

- sorts activated leaf nodes (based on the `priority of` keyword and the tree structure), and grants or revokes access to resources based on this order.

Examples

Algorithm 2 shows a Playful script for a toy example corresponding to a dynamic ball-searching and following behavior. The corresponding tree representation is displayed in Figure 1. We implemented this script in Nao, a small humanoid robot commercialized by Softbank Robotics, and the related behavior can be seen in the support video [17].

Script Description

The higher level of behavior is shown in

```
ball_chase, switch to sit if battery_low
sit, switch to ball_chase if battery_high
```

This implements a state machine that has the robot chasing the ball until its battery gets low (`ball_chase`), in which case it sits (`sit`). When recharged, the robot resumes the chasing behavior. `ball_chase` is declared as a list of three nodes:

```
ball_chase:
    ball_detection | out=ball
    bargeting ball: following, whenever
      time_ago <2, priority of 2
    searching, priority of 1
```

When ball chasing is activated, the vision module for ball detection is activated and pushes schemes describing detected balls to the memory. In this case, a ball is described using three properties: its color (to differentiate one ball from another), its position, and a time stamp. When a new ball is detected, the targeting command creates a `following` branch related to it—i.e., a pulling of position and color information from the memory via the Python API will return the most recent information corresponding to the targeted ball. This branch is activated only when the ball has been recently detected, as commanded by the `time_ago <2` evaluation. When activated, because of its higher priority score, it blocks searching, which is of lower priority. Only at startup, when no ball has been detected yet, or when a ball has not been detected for more than 2 s, the lower priority `searching` branch may activate. Not using priorities results in an undefined behavior, as both `following` and `searching` would, at times, be set to activate without informatio n about which should take precedence. This results in the robot alternating between following and searching for the ball.

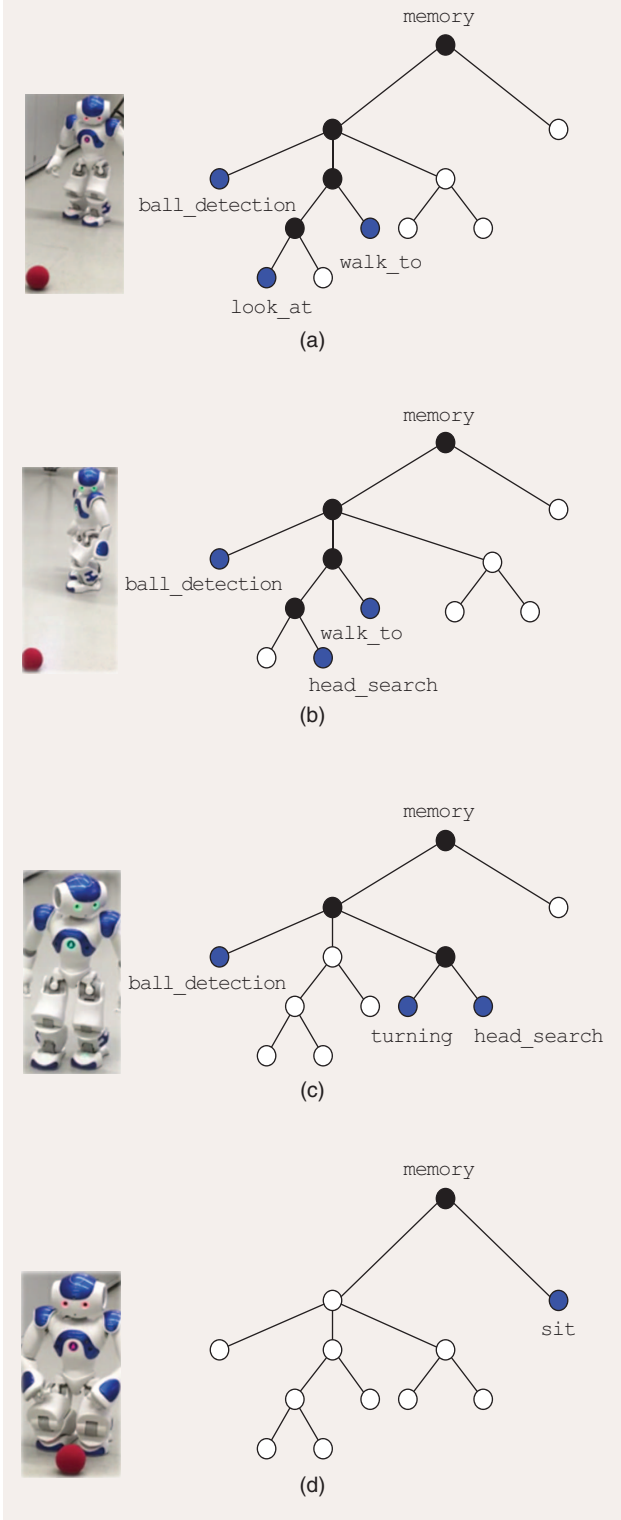

Figure 2. Some possible patterns of activation of the tree presented in Figure 1 where the robot is (a) walking toward the ball when it sees it; (b) walking toward the last known position of the ball while searching for it with its head; (c) performing a full body search; and (d) sitting in a safe position when its battery runs low. Black is used to indicate activation of nodes and blue activation of leaf nodes. For brevity, only activated leaf nodes are labeled.

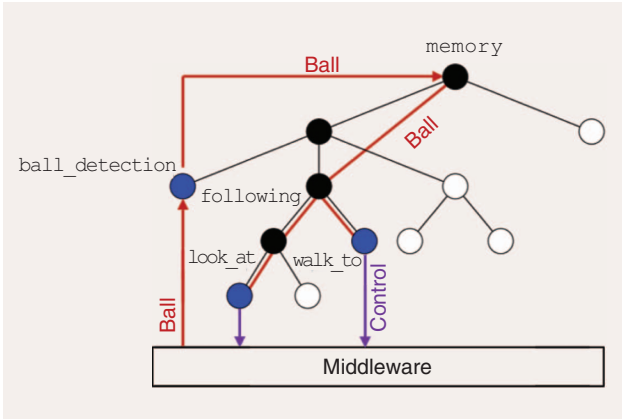

Figure 3. The flows of information are monitored during runtime to implement sensory–motor couplings, here for having the robot tracking the ball.

In turn, `following` corresponds to looking at the ball while walking toward it (as long as the distance between the ball and the robot is significant, as expressed by the evaluation implementing the Python function `far`):

```
following:
    looking
    walk_to, whenever far
```

and looking at the ball corresponds to either centering the detected ball in the center of vision (`look_at`) or performing a rapid head-search motion (`head_search`):

```
looking:
    look_at, whenever time_ago <1,
      priority of 2
    head_search, priority of 1
```

It can be noted that the Python code corresponding to leaf nodes and evaluations can be reused not only across programs but also several times in the same script. For example, the `head_search` node is used in two branches: `looking` and `searching`.

The full behavior is robust, as the robot either 1) walks toward the ball when it sees it, 2) keeps walking toward the last known position of the ball while searching for it with its head, 3) stops walking and performs a full body search, or 4) sits in a safe position if at any time its battery runs low. Related patterns of activation in the behavior tree are displayed in Figure 2.

Reactivity

The keyword `targeting` relates sensory information corresponding to the detected ball to the `following` node. Figure 3 shows the resulting flows of information that are applied at runtime when the `following` subtree is activated. Information encoded by the

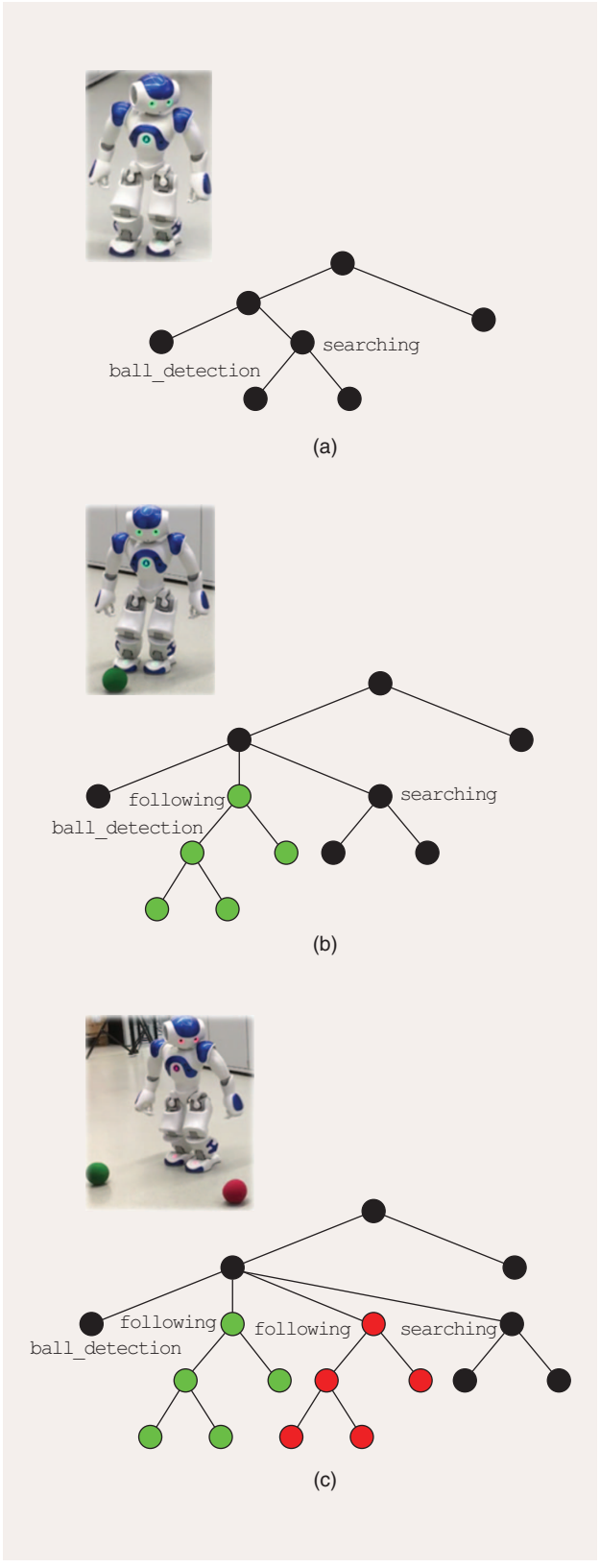

Figure 4. An illustration of dynamic topology updating. (a) The topology of the behavior tree at runtime prior to any ball detection by the robot. (b) The robot detects a green ball, and a corresponding `following` subtree is instantiated (in green). (c) A red ball is detected, resulting in the instantiation of a competing `following` subtree (in red).

`ball_detection` leaf nodes are pushed to the shared memory. The targeting keyword then links the `following` branch to the related memory item. By virtue of the resulting information flows, the leaf nodes `look_at` and `walk_to` dynamically reconfigure their parameters. Evaluations along the subtree may also relate to the targeted memory item; e.g., `time_ago` refers to the latest time stamp of the memory item.

Adaptation for Multiple Targets

As presented in the "Targeting" section, the keyword `targeting` commands the online creation of subtrees. That keyword conditions the runtime instantiation of targeted subtrees to the creation of their related memory key. Figure 4(a) shows the instantiation of the tree at startup. Because no corresponding memory key has yet been created, the `following` subtree has not been instantiated. In Figure 4(b), the robot detected a green ball, and a corresponding `following` subtree has been instantiated.

If the `ball_detection` node differentiates between balls (e.g., using color information), it pushes related information to several memory keys. The memory will host a separate memory key for each ball detected. The `targeting` keyword then results in online instantiation of several subtrees, one per memory key. The script presented in Algorithm 2 can easily be modified to support an environment with multiple balls (the updated code is in bold):

```
targeting ball: following, whenever time_
    ago <2, priority of
    2 + 1/distance
```

This modification sets the robot to follow the closest ball, as reevaluated continuously during runtime. In Figure 4(c), the robot detected a red ball, and a second `following` branch has been instantiated.

Filtering

Leaf nodes can be used to implement filters. For example, the original script shows

```
targeting ball: following, whenever
    time_ago <2, priority of 2
```

The following modification implements filtering:

```
targeting ball: filter | out=filtered_ball
targeting filtered_ball: following,
    whenever time_ago <2, priority of 2
```

This syntax is agnostic to the specifics of the filter implemented. In our example, the filter combines knowledge about the perceived ball with the odometry functionality provided by the middleware to implement a Kalman

filter. While the syntax is trivial, it enforces suitable information flows, presented in Figure 5. Because Playful relies on schemes for data exchange (see the "Targeting" section), it is possible to create generic filters that can be applied over a wide range of objects. For example, the filter used in this section could be targeted by any scheme having a `position` and `time_stamp` property. It can be noted that, via the `targeting` keyword, it is possible to relate branches to different levels of filtering. A branch, for instance, may relate to raw data (`targeting ball`), while another relates to filtered data (`targeting filtered_ball`).

Refactoring

Playful's declarative programming paradigm allows for rapid refactoring of existing programs. These advantages have been exposed for TDM [6] and experimentally validated in [5]. Here, we briefly show through an example how they apply to Playful. In Algorithm 2, the robot looks at the ball it is approaching. The behavior would be more robust if it could share attention between all of the detected balls. The robot could better reevaluate the relative distance to all balls, and therefore make a better-informed decision of which ball to approach. The original script is

```
targeting ball: following, whenever
    time_ago <2, priority of 2
following:
    looking
    walking_to, whenever far
```

The following shows how a simple refactoring of the original script results in the robot alternating in looking at each ball:

```
targeting ball: walking_to, whenever
    time_ago <2 and far,priority of 2
targeting ball: looking, priority of 1
    + time_not_seen
```

The longer the robot did not see a specific ball, the higher the priority of the related look action. This results in the robot looking alternatively at each ball.

Core Reuse

Because of its tree-based organization, full Playful applications may be trivially reused as leaf nodes of larger application trees. For example, the full tree of the ball-tracking behavior may be saved in a file `ball_tracking.play`, which may be reused in other programs. This feature provides a strategy for the development of robotic applications of increasing complexity. Full applications may be created and debugged, saved as leaf nodes, and then combined. The following section shows examples of applications developed using this approach.

Complex Behaviors

A ball-grouping behavior executed by Nao and an interactive manipulation behavior performed by Apollo, a real-time-controlled upper-torso humanoid robot, have been obtained exclusively through the combination of evaluations and leaf nodes of modest size (three to 72 lines of code, averaging 21). This indicates that complexity is well managed and spread over simple reusable components, rather than centralized in a few complex leaf nodes. Spreading of complexity across reusable components enables developers to extend behaviors via recombination of these components. An extension of Apollo's behavior in the "Extended Behavior" section provides a concrete example.

Ball-Grouping Behavior Executed on Nao

The goal of the application considered in this section is to keep two balls together, either by kicking one toward the other (if both balls are on the floor) or by grasping a ball and bringing it to the other (if one of the balls is presented to the robot by a person). The application is executed on a Nao robot, with no supplementary external sensing. Figure 6 shows the robot executing the behavior, which is also included in the support video [16].

Figure 7 includes the layout of the tree when the robot has detected only one ball. The online creation of subtrees results

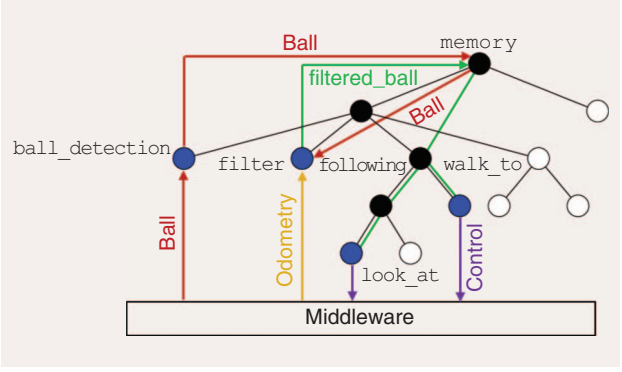

Figure 5. A filter is added to the tree.

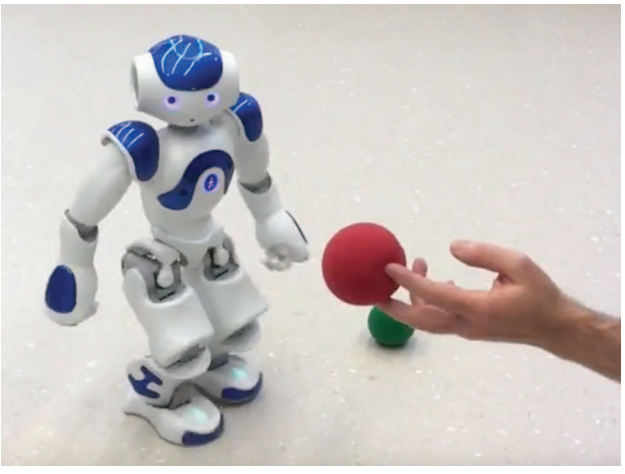

Figure 6. A Nao robot executing the ball-grouping behavior.

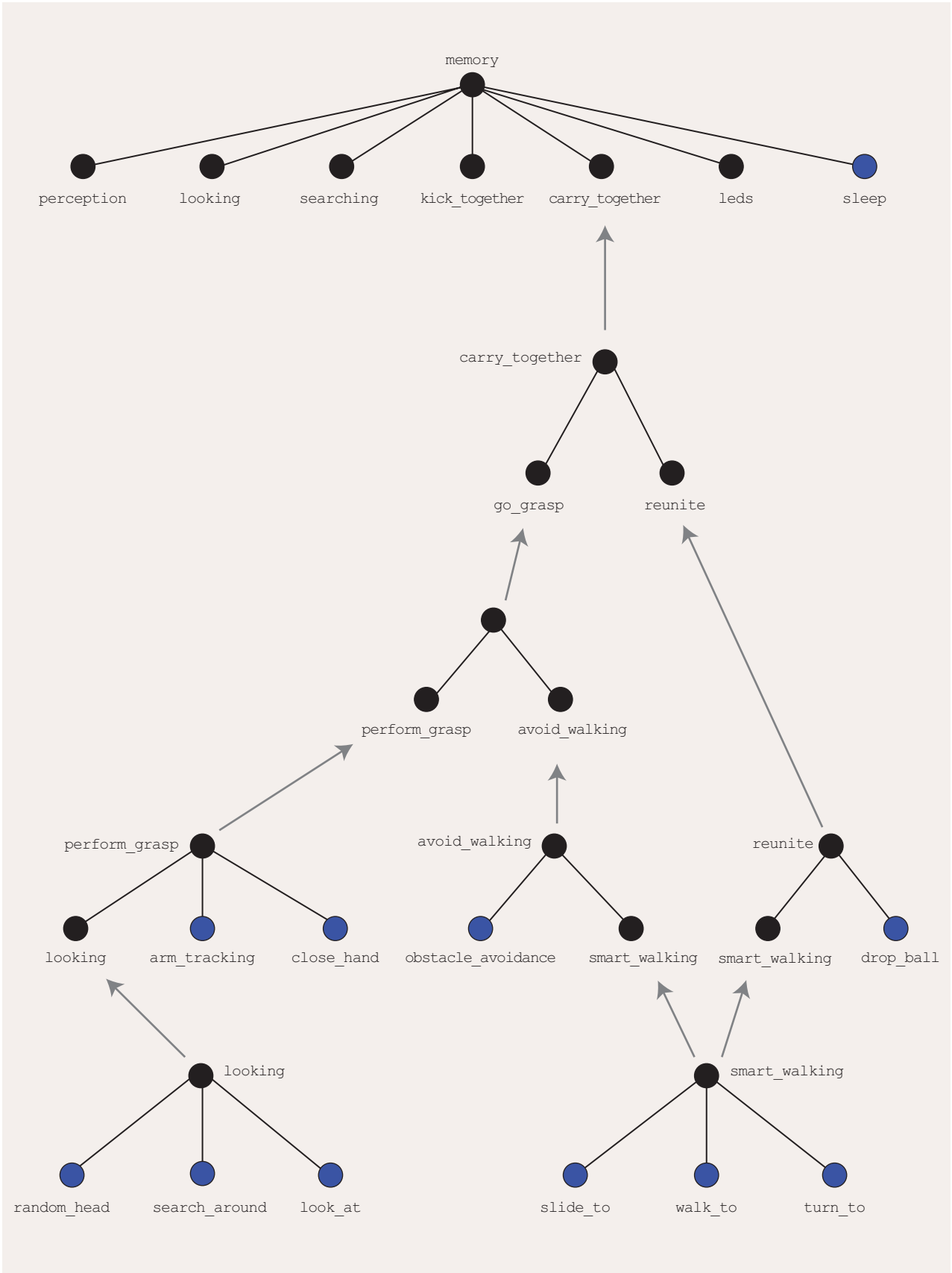

Figure 7. The iterative development of the `carry_together` subtree of the reuniting application for Nao. In blue are leaf nodes. From bottom to top, trees of small size are created and debugged and then used as leaf nodes of trees of higher levels of abstraction. Other nodes of the full application, e.g., `kick_together`, are developed using a similar approach (but are omitted here for brevity).

in a considerably larger behavior tree at runtime once Nao detects a second ball. We developed this tree by first creating and debugging simple trees, then reusing these as leaf nodes of other simple trees, which we also debugged, and so forth until we obtained the desired complex behavior.

The tree has Nao sharing attention between the two balls. The robot uses virtual targets to which to walk for either aligning itself with the two balls to kick one in the direction of the other or for walking toward one while avoiding the other and performing a reactive arm-tracking behavior based on inverse kinematics.

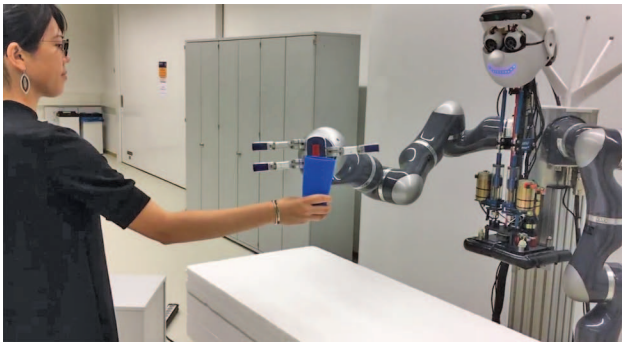

Figure 8. Apollo executing an interactive manipulation behavior.

Interactive Manipulation Task Executed on Apollo

Software Stack

Apollo is a fixed-base manipulation platform equipped with seven-degrees of freedom Kuka LWR IV arms, three-fingered Barrett Hands, and a red, green, blue depth camera (Asus Xtion) mounted on an active humanoid head (Sarcos). Its operating system, simulation laboratory (SL), runs over a real-time-patched Linux kernel (Xenomai), and performs torque control using an inverse dynamics controller to track the desired joint state [12]. The controllers implemented in SL are designed to remain stable in the context of rapid behavior switching, which is a requirement for safe interfacing with Playful. The ROS is used for broadcasting RGB images and point clouds collected from the head-mounted camera, applying sensor processing using the point cloud library and the OpenCV libraries, and calculating transformations between various reference frames. ROS nodes broadcast sensory information, transforms, and joint states. This approach corresponds to standard robotics practices.

The Playful tree is added above the ROS. The sensory leaf nodes subscribe to the underlying topics, and the motor leaf nodes publish the desired joint states, which are then transmitted to the underlying SL controller via the ROS in real time.

Original Behavior

We used Playful to develop a human–robot interaction application. We controlled the right arm of the robot to mirror the motion of the arm of the closest person standing in front of it. When presented with a cup, the robot grasps it and places it at the location indicated by the person. As the cup moves, new grasping postures are dynamically recomputed. Both for the sake of keeping objects centered in the field of vision and giving an impression of liveliness, the robot alternates its gaze between the presented cup, the person, and the person's hand.

A picture of Apollo performing the application is shown in Figure 8, and the behavior is also shown in the support video [16]. As can be seen, the robot is highly interactive. However, we effectively managed the complexity with reusable components of modest size. All of the code and modules used are based on well-known accessible libraries, part of the roboticist's standard tool kit. We were able to achieve behavioral richness and reactivity because of Playful's efficient code organization and expressiveness.

Extended Behavior

We exclusively extended the behavior by reusing existing nodes and evaluations. This extension has the robot dropping the cup it has in hand if any other cup is presented to it. To obtain the desired results, the nodes for arm tracking, motion planning, and evaluations that assess the relative positions of objects were applied in a new branch. This extension required the addition of only six lines to the Playful script and did not require development of any new Python component. Yet it extends the behavior considerably. This extended behavior can be found in the support video [16]. A video of Apollo running this application when interacting with various guests is shown online [18].

We were able to achieve behavioral richness and reactivity because of Playful's efficient code organization and expressiveness.

Future Work

Playful is suitable for the creation of first-order reactive behavior. Our efforts are now directed toward evaluating Playful as an interface between higher-level decision-making systems and reactive execution. A standard approach consists of using planners to generate sequences of tasks to execute and monitor. Our platform is different.

In Playful, all of the logic is implemented via evaluations. As evaluations correspond to arbitrary Python code, they may be used as bridges with any higher-level software that supports interfacing with Python. Via evaluations, other software may thus command activation or deactivation of subtrees, modify priorities, or change the state of the running program.

To evaluate this approach, we have interfaced Playful with a case-supported principle-based behavior paradigm, a

system for deciding which action a robotic agent should accomplish based on ethical principles [13]. The target application is elder care, and the target robotic system is TiaGO (PAL Robotics). Proof of concept is currently performed using Nao. The ethical engine continuously reevaluates the situation as observed by the robot by connecting to Playful's shared memory. Via the evaluations it interfaces with, it sets the priority of the higher-level subtrees based on their ethical desirability. Details may be found in [14], a presentation video is online [19], and the latest results are being prepared for publication. The success of this approach would enhance the reactivity displayed by robots without sacrificing the framework's ability to interface with other advanced decision-making algorithms.

All of the code and modules used are based on well-known accessible libraries, part of the roboticist's standard tool kit.

Conclusions

Playful is a software creation for behavior engineering whose front end is a scripting language based on a reactive programming paradigm. It supports the development of simple, clear, and well-structured code that brings robots to life. Unlike other complex software and robotic architectures, Playful is easy to use and deploy. It uses a five-keyword scripting language that supports the rapid definition of dynamic behavior trees of arbitrary complexity. Playful not only allows for specifying the desired logic, but it expresses the flow of information that should be applied at runtime. As a result, reactive behavior can be created using expressive instructions. Playful is agnostic regarding the middleware and the robotic platform it runs on, making it a convenient tool for a wide range of applications.

Acknowledgments

This research was supported in part by the Max Planck Society, National Science Foundation (grants IIS-1205249, IIS-1017134, and EECS-0926052), the Office of Naval Research, and the Okawa Foundation.

References

[1] C. Elliott and P. Hudak, "Functional reactive animation," in *Proc. Int. Conf. Functional Programming*, 1997, pp. 263–273.

[2] J. Peterson, P. Hudak, and C. Elliott, "Lambda in motion: Controlling robots with Haskell," in *Practical Aspects Declarative Languages (PADL)*, G. Gupta, Ed. Berlin, Heidelberg: Springer-Verlag, Jan. 1999, pp. 91–105.

[3] P. Hudak, A. Courtney, H. Nilsson, and J. Peterson, "Arrows, robots, and functional reactive programming," in *Advanced Functional Programming* (Lecture Notes in Computer Science, vol. 2638), J. Jeuring and S. L. P. Jones, Eds. Berlin, Heidelberg: Springer-Verlag, 2003, pp. 159–187.

[4] D. Kortenkamp and R. Simmons, "Robotic systems architectures and programming," in *Springer Handbook of Robotics*, B. Siciliano and O. Khatib, Eds. Berlin, Heidelberg, Germany: Springer-Verlag, 2008, pp. 187–206.

[5] V. Berenz and K. Suzuki, "Targets-Drives-Means: A declarative approach to dynamic behavior specification with higher usability," *Robotics Auton. Syst.*, vol. 62, no. 4, pp. 545–555, 2014.

[6] V. Berenz, F. Tanaka, K. Suzuki, and M. Herink, "TDM: A software framework for elegant and rapid development of autonomous behaviors for humanoid robots," in *Proc. IEEE-RAS Int. Conf. Humanoid Robots*, Oct. 2011, pp. 179–186.

[7] M. Beetz, L. Mösenlechner, and M. Tenorth, "CRAM: A Cognitive Robot Abstract Machine for everyday manipulation in human environments," in *Proc. IEEE/RSJ Int. Conf. Intelligent Robots and Systems*, Oct. 2010, pp. 1012–1017.

[8] T. J. de Haas, T. Laue, and T. Roefer, "A scripting-based approach to robot behavior engineering using hierarchical generators," in *Proc. IEEE Int. Conf. Robotics and Automation*, May 2012, pp. 4736–4741.

[9] J.-C. Baillie, "URBI: Towards a universal robotic interface," in *Proc. 4th IEEE/RAS Int. Conf. Humanoid Robots,* 2004, pp. 33–51.

[10] A. Marzinotto, M. Colledanchise, C. Smith, and P. Ogren, "Towards a unified behavior trees framework for robot control," in *Proc. IEEE Int. Conf. Robotics and Automation*, May 2014, pp. 5420–5427.

[11] S. Lemaignan, A. Hosseini, and P. Dillenbourg, "PYROBOTS: A toolset for robot executive control," in *Proc. IEEE Int. Conf. Intelligent Robots and Systems*, Sept. 2015, pp. 2848–2853.

[12] S. Schaal. (2009). The SL simulation and real-time control software package. Univ. Southern California, Los Angeles. [Online]. Available: http://www-clmc.usc.edu/publications/S/schaal-TRSL.pdf

[13] M. Anderson and S. L. Anderson, "Toward ensuring ethical behavior from autonomous systems: A case-supported principle-based paradigm," *Ind. Robot: An Int. J.*, vol. 42, no. 4, pp. 324–331, 2015.

[14] M. Anderson, S. L. Anderson, and V. Berenz, "Ensuring ethical behavior from autonomous systems," in *Proc. Artificial Intelligence Applied Assistive Technologies and Smart Environments Workshop*, Phoenix, AZ, Feb. 12, 2016.

[15] Playful. [Online]. Available: playful.is.tuebingen.mpg.de

[16] V. Berenz. (2017, Sept. 15). Playful: Reactive programming for orchestrating robot behavior. [Online]. Available: https://youtu.be/Kb6O0KKXKp8

[17] V. Berenz. (2017, May 14). How to create a reactive robot application for Nao (with Playful). [Online]. Available: https://youtu.be/784eL6uSKbk

[18] V. Berenz. (2017, Oct. 12). Guests playing with Max Planck Apollo (long). [Online]. Available: https://youtu.be/ivJ8ZdUPOPo

[19] Machine Ethics. (2017, Dec. 8). A value driven agent [press]. [Online]. Available: https://youtu.be/V8utFzn7DJk

Vincent Berenz, Autonomous Motion Department, Max Planck Institute for Intelligent Systems, Tübingen, Germany. E-mail: vberenz@tuebingen.mpg.de.

Stefan Schaal, Autonomous Motion Department, Max Planck Institute for Intelligent Systems, Tübingen, Germany, and Computational Learning and Motor Control Lab, University of Southern California, Los Angeles, United States. E-mail: sschaal@usc.edu.