# A Compact Task Representation for Hierarchical Robot Control

Luke Fraser*, Banafsheh Rekabdar†, Monica Nicolescu‡, Mircea Nicolescu§,
David Feil-Seifer¶ and George Bebis‖
Department of Computer Science and Engineering, University of Nevada, Reno
Email: *fraser@nevada.unr.edu, †brekabdar@unr.edu, ‡monica@cse.unr.edu, §mircea@cse.unr.edu,
¶dave@cse.unr.edu ‖bebis@cse.unr.edu

***Abstract*— Robot tasks for real-world applications typically involve multiple paths of execution, where the same task can be achieved in different ways. This poses challenges with respect to the representation and execution of such tasks, as enumerating all possible execution paths leads to combinatorial increases in the size of the representation. We present a novel robot control architecture that addresses these challenges. The architecture 1) provides an efficient, compact encoding of tasks with multiple paths of execution, 2) uses the same compact representation as the controller that the robot will use to achieve its goals, 3) allows the robot to dynamically decide which execution path to follow using an activation spreading mechanism that relies on environmental conditions, and 4) provides a mechanism for robustness to changes in the environment during the task execution. We validate our architecture using a humanoid PR2 robot, showing that the robot dynamically selects a path of execution based on the current state of the environment, and is robust to environmental changes.**

## I. Introduction

In real-world applications, the tasks that a robot would have to complete are typically more complex than a sequence of steps that must be performed in a predefined order. Furthermore, the same task could be performed in a wide variety of ways, due in large part to the affordances present in the environment. As an example, an assembly task may have parts during which some steps must be executed sequentially (e.g, an axle must be mounted before the wheel), other parts in which the steps could be executed in any order (e.g., mounting four wheels could happen in any order), and also parts that could be achieved through multiple paths of execution (e.g., could use either wrench1 or wrench2 to tighten the bolts). The tasks could further be structured using a hierarchical representation.

The major challenge for encoding tasks with *no ordering constraints* on their steps and with *multiple paths of execution* is that it leads to a combinatorial increase in the size of the representation, due to the fact that all possible execution paths need to be explicitly encoded in the architecture. While the tasks can, in principle, be encoded using a compact representation, the robot controller needs to expand this representation for actual robot execution. We propose a robot control architecture that enables both a *compact encoding* and execution of the above tasks using the same task representation. Furthermore, through the use of an activation spreading mechanism, the representation allows the robot to dynamically decide which path of execution to follow. The architecture follows a behavior-based paradigm, in which basic robot capabilities are represented as nodes in an interconnected network. Another contribution of the proposed method is the *robustness to environmental changes during the task execution*: as the nodes in the control architecture continuously evaluate the environmental conditions, the activations passed between nodes in the architecture will dynamically change to reflect the most current state of the environment, allowing the robot to switch the order in which the task steps are executed. This is achieved through a particular process embedded with each task node, which enables re-evaluation of conditions and switching to another task step.

The remainder of this paper is structured as follows: Section II describes previous related research, Section III presents our approach for encoding and executing complex tasks, Section IV shows our experimental evaluation and Section V gives a summary of the presented work.

## II. Related Work

The focus of the proposed work is to address challenges that arise in real-world environments, for robots performing service or assistive tasks.

Recent work addresses challenges related to complex task representations using a probabilistic approach for predicting human actions and a cost based planner for the robots response [1]. This work presents a task representation that can encode tasks with multiple paths of execution. The initial representation for the task is a compact *AND-OR* tree structure, but for action prediction and planning, it has to be converted into an equivalent Bayes network, which explicitly enumerates all possible alternative paths.

The task representation we propose is similar to that of hierarchical task networks (HTNs) [2], which have been widely used in automated task planning. While HTN and planning approaches [3][4][5][6] can automatically generate a task network plan, the plan remains unchanged for the duration of the execution, or until there is a need to replan. We use a flexible task network (given by a domain expert), which encodes all possible ways in which that task can be performed. By using a distributed activation spreading approach, the robot dynamically selects which actions to perform in order to achieve the task's goals, based on the most current state of the environment. Without any restrictions, HTNs are more complex than partial order planning (POP) [7], as shown by theoretical studies [8].

In contrast, the proposed activation spreading approach can dynamically and in real-time select an appropriate course of action, out of the multiple options available. Activation spreading has been successfully introduced by [9], in a scenario in which STRIPS-like pre-conditions were used to drive the activations of behaviors for action selection and sequential task execution in a simulated environment. [10] also proposes an activation spreading approach for action selection, focused on solving dynamic requests (of simple structure) with potential deadlines. In contrast with these methods, the representation proposed in this paper uses activation spreading with task representations that can have hierarchical structure, ordering constraints, and alternative paths of execution.

Hierarchical representations that rely on behavior-based architectures have been proposed [11]. This architecture has been employed in the context of learning by demonstration, in order to build task representations that generalize from multiple demonstrations and have alternative paths of execution [12], or that can encode fusion of multiple low-level behaviors [13]. However, these representations are not capable of representing the general types of task structures that are proposed in our work.

The ability of robotic systems to recover from environmental changes during task execution is a key component of architectural robustness. Recently, [14] developed a controller synthesis mechanism that is able to handle violations in the assumptions about the environment present in the task specification. The focus of their work is on the controller synthesis/re-synthesis process and the approach is being applied to mostly sequential tasks. In this work, the robustness to environmental changes emerges from the continuous spreading of activation, in a task representation that encapsulates complex hierarchical constraints.

The contribution of the proposed work is the development of a control architecture that compactly encapsulates complex, hierarchical task constraints, and allows for dynamic action selection and robustness to environmental changes through the use of activation spreading. This addresses limitations of the currently existing approaches for task representation and execution.

## III. Proposed Architecture

This section begins by presenting general terms used within the following sections. These terms define many facets of the functionality of the underlying control architecture. The remainder of the section uses the general terms to fully describe the proposed task representation and architecture.

### A. General Terms

- **Node**: The underlying structure that encompasses all the behaviors in the system. The communication as well as the *update-loop* architecture is maintained in this object. Every behavior inherits from the base node object.
- **Goal node**: $\{goal \in \{THEN, AND, OR\}\}$ A goal node provides the base goal control behaviors of the hierarchical task structure.
- **Behavior node**: A *behavior node* encompasses all the leaf nodes in the task structure and encodes physical behaviors that can be performed by the robot. An example of such a Behavior node would be a *pick spoon behavior*. A *pick spoon behavior* will control the robot arm to pick up a spoon object from the table in front of the robot. A behavior node can arbitrarily break down a task into simple modular components that combine to complete a larger more complex task.
- **Activation Level**: A node's *activation level* is a number provided by a node's parent. It represents the significance and priority placed on the goal of a given node.
- **Activation Potential**: A node's *activation potential* encapsulates a node's perceived efficiency with respect to performing its work. For instance, the longer a node perceives it will take to complete task, the lower its *activation potential* will be. The resulting activation potential is sent from a child to its parent.
- **Preconditions**: The *preconditions* of a node are the goals that must be completed prior to a node changing into a *running* state. In the case of an *AND* node, all children must be in the done state prior to the *AND* node switching to the *running* then *done* state. The *precondition* for an *OR* node is that the child node with the highest *activation potential* be completed prior to the *OR* node switches to the *running* state.
- **Node State**:
  - **Active**: A node becomes *active* when its activation level exceeds a predefined threshold-$\tau_{active}$.
  - **Running**: A node is *running* while it is performing the behaviors action.
  - **Done**: A node is *done* when it has completed the *running* action.
- **Action**: A nodes action is representative of the behavior of the node. In the case of a *pick-behavior* the action of the node is to pick up an object using the robot manipulator.

### B. Task Representation

The control architecture we present below serves the following main roles:

1) Provides an efficient, compact encoding of tasks that have sequential, non-ordering, and alternative paths of execution.
2) Allows the use of this compact representation to execute the robot's goals.
3) Allows the robot to dynamically decide which execution path to follow using an activation spreading mechanism that relies on environmental conditions.
4) Allows the robot to adapt to changes in the environment during task execution.

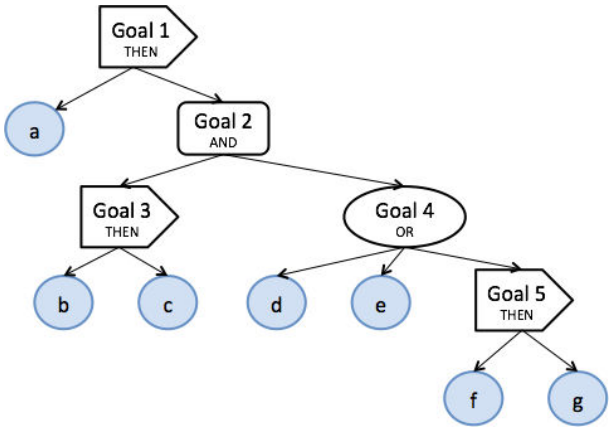



Fig. 1: Representation of task: a **THEN** ((b **THEN** c) **AND** (d **OR** e **OR** (f **THEN** g))).

We developed our representation using a behavior-based paradigm [15], which provides modularity and ease in communication and connectivity between behavioral modules. Our aim is to enable the system to encode tasks that involve temporal sequencing constraints (some steps need to happen before others), alternative ways of execution (any one of multiple options is acceptable), and no temporal constraints (some steps can be executed in any order). All these options could be a part of a single task representation, which could be seen from a task such as: a **THEN** ((b **THEN** c) **AND** (d **OR** e **OR** (f **THEN** g))). To encode such a task we will define two types of nodes in our behavior network. **Behavior nodes** encode a basic behavior that achieves a well-defined goal (such as a, b, c, d, e, f, g above). **Goal nodes** are N-ary trees (i.e., can have from 0 to N children) and encode the three different types of execution constraints mentioned above, as follows:

- **THEN** goal nodes encode sequencing constraints. For example, GoalSeq = a **THEN** b, implies that in order to achieve GoalSeq, the system should execute behavior a, followed by behavior b.
- **OR** goal nodes encode alternate paths of execution. For example GoalAlt = a **OR** b, implies that in order to achieve GoalAlt, the system can execute either behavior a or behavior b.
- **AND** goal nodes encode the option of having no ordering constraints. For example GoalNoOrd = a **AND** b, implies that in order to achieve GoalNoOrd, the system should execute both behavior a and behavior b, but in any order. This also leads to alternative paths of task execution, but in which all the individual components must be performed at some point.

With these types of components, the above task can be represented as shown in Fig. 1. A task can have any type of goal node as a root and there is no restriction on which nodes can be parents of others in the hierarchy. It can be seen that this representation compactly encodes all the task constraints, including all possible paths of execution. This is especially important when there are multiple alternative paths, such as for Goal2: either Goal3 or Goal4 can be performed first; to achieve Goal4, either one of d, e, or Goal5 can be executed. Instead of explicitly enumerating all possibilities [16], we use the most compact form of the task representation. This representation can be more accurately represented by the following prefix encoding: (**THEN** a, (**AND** (**THEN** b c) (**OR** d e (**THEN** f g)))).

### C. Task Execution

The control architecture operates in a publish/subscribe environment provided by ROS [17]. This facilitates and maintains network connectivity and sends messages between parent and child nodes. Each parent node is connected to each of its children and each child is connected to its parent. With these connections, timed messages are sent asynchronously between nodes, allowing each node to run in a distributed fashion in the system. Each node stores the following information:

- type, which can be either **THEN**, **OR**, **AND**, or **BEHAVIOR**
- state, which can be either *active*, *running*, or *done*
- activation level
- activation potential

To execute a task, *activation spreading* messages are sent from the root node of a given task toward its children. At the same time, each node sends *status* messages to its parent node. *Status* messages encode the current state of any given node. This allows decisions to be made both from a top-down perspective of the graph as well as from the bottom-up. This state information maintained in each node is used to perform this distributed top-down and bottom-up activation spreading.

The system uses two types of messages: *activation spreading messages*, which are sent from parent nodes to their children and *state messages*, which are sent from child nodes to their parent. *activation spreading messages* contain the following information: *sender* (the node that sent the message), and *activation level* (the amount of activation that the parent sends to the child with this message).

State messages contain the following information: *active* status (true if the child node is currently performing an action, i.e, actuating a robot arm), *done* status (true if the node has finished its actions and achieved its goal), *activation level* (the current activation level of the child node) and *activation potential* (the node's perceived efficiency for executing its work).

The *activation level* is the primary mechanism for top-down activation spreading in the task network. When the root node sends activation to its children, those children will spread their activation to their children and so on. Based on the node type, each *goal node* will impose a different method to determine which children receive activation and when. The activation levels will spread throughout the system until the task is complete.

The *activation potential* is the primary mechanism for bottom-up activation spreading. It solves a critical limitation of only top-down spreading of activation. It is used to pass real-time information about the feasibility of children

behaviors and is mostly relevant for *OR* nodes and the *mutex* acquisition, this is described below and in Section III-D respectively. It is useful when determining which child should be awarded higher activation or possibly be activated first. In the case of an *OR* goal node deciding which child to activate, the *activation potential* can be used to pick the most efficient child node.

The different types of *goal nodes* send activation messages as follows:

- **THEN** *goal nodes* evaluate the status of their children in the order given by the sequence and send **activation messages** to the first child node in the list whose status is not met. Once that child node signals completion the subsequent child will receive activation. The *activation level* is sent to the first child within an activation message. Each child node will update its *activation level* during each time-step of its update loop, as described in section III-D.
- **OR** *goal nodes* send activation messages to the child with the highest activation potential. Leaf nodes (basic behaviors) will compute and send their *activation potential* to their parents at each time step. This allows for opportunistic task execution, in situations in which the environmental conditions are met for just one (or some) of the alternative pathways. In case of equal activation levels and applicability conditions, the robot will choose one of the options at random. Once a pathway becomes active (detected through messages from the children), the other children will receive zero activation.
- **AND** *goal nodes* send activation messages to all their children equally and at the same time.

### D. Update Loop

**Algorithm 1** Behavior update loop

```
1: if Not Done then
2:   if Is Active then
3:     if Preconditions then
4:       if BEHAVIOR NODE then
5:         Activate :
6:           MutexAcquired ⇒ state ← running
7:       else if GOAL NODE then
8:         Activate :
9:           state ← done
10:      end if
11:    else
12:      SpreadActivation
13:    end if
14:    ActivationFalloff ⇒ α ∗ activation_level
15:  end if
16: end if
```

Each node in the system is running an update loop, which runs at every clock tick, and is responsible for controlling the state and the execution of that node. In order to decide a node's actions, the update loop runs through a series of checks, as shown in Algorithm 1. All behaviors implement the same update loop as follows:

- **If-Done**: Check if the node has completed its task; if yes, do nothing. A node is done when its running process has completed successfully.
- **If-Active**: Check if the node is active; if not, do nothing. A node becomes active when its activation level is above a threshold-$\tau_{active}$
- **If-Preconditions**: Check if the nodes' preconditions are satisfied; if yes, set node to *running*, otherwise spread activation (as described below). Preconditions are the set of conditions that must be met in order for the node to begin its work and they ensure that this happens only after all the required tasks constraints are satisfied.

Based on the results of the above checks, the nodes may take the following actions:

- **Activate**: When all the checks are satisfied, the node will activate itself and proceed to *running*. This action signals a thread to begin performing the node's corresponding behavior. The node will continue to run its update loop unhindered. The work being done is encapsulated by the behavior. In the case of a pick and place behavior, *running* is achieved by taking control of robot arm and picking up and placing an object. A goal node will only set its state to done in this phase of execution as no *running* behavior is defined. If an error occurs the running state will be disabled and the node will return to the active state. Scenarios where this occurs is described in Section III-F.
  Each *behavior* node's activation is restricted by a mutex, which controls the access to the robot effectors. To become active, each node must satisfy a precondition of acquiring the mutex responsible for arm manipulation. The method of acquiring the mutex relies on the behavior's *activation potential*. When multiple *behavior* nodes race to acquire the mutex, access will be given to the node with the highest *activation potential*. This is accomplished as follows: when the mutex receives a lock request, a timer is started to allow other behaviors to bid for control. When the timer ends, the mutex grants the lock to the node that bid with the highest *activation potential* and denies all other behaviors. The denied behaviors will continue to request a lock until it is acquired. This mechanism enforces responsible use of the robot's actuators, as no two behaviors can attempt to use a robot's arm at the same time. As well, this locking mechanism can be expanded to allow for further modularization of the robot's actuators: one mutex for each arm on the PR2 would allow for actions to be performed in parallel, using both arms.
- **Spread Activation**: When the preconditions are not met, a control message with an activation level is sent to the children behaviors (as described above). Children nodes do not become active unless their activation level is above a threshold-$\tau_{active}$. This threshold is a parameter set at run-time.

- **Activation Falloff**: The *activation level* of the node is decreased proportionally by a prescribed amount $\alpha$ at the end of each time-step of the update-loop. This ensures that a node that does not receive activation from a parent will slowly lose activation. The *activation level* is multiplied by the $\alpha$ value parameter at each time-step of the the update loop. This value is in the range between (0,1), exclusive.
- **Publish Status**: A state message is sent to the parent at the end of each update loop. This message encodes the current state of the node. This status message is used to spread *activation potential* so that parent nodes can determine which children to activate.

**Algorithm 2** THEN - Spread Activation

$queue \leftarrow child\_list$
$msg \leftarrow \{activation\_level = 1.0\}$
**if** $queue.front().isDone()$ **then**
  $queue.pop()$
**end if**
$SendToChild(queue.front(), msg)$

In algorithm 2 the activation spreading mechanism of the *THEN* goal node is described. A queue of children is created and as each child finishes its task, it is de-queued and the next child is sent activation. This enforces a sequential activation of the child behaviors from the *THEN* goal node.

**Algorithm 3** AND - Spread Activation

$msg \leftarrow \{activation\_level = \frac{1}{number\_of\_children}\}$
**for** $child \in children$ **do**
  $SendToChild(child, msg)$
**end for**

Algorithm 3 shows the *AND* goal node activation spreading method. This differs from the *THEN* nodes spreading in that all nodes receive activation simultaneously, but by a reduced amount. This allows all child behaviors to activate simultaneously or all race to acquire the arm mutex simultaneously. For *AND* nodes that have a large number of children, this algorithm can be modified to send a fixed, equal activation to all the children, in order to ensure that their activation levels can reach the $\tau_{active}$ threshold.

**Algorithm 4** OR - Spread Activation

$msg \leftarrow activation\_level = 1$
$max\_child \leftarrow max(child.ActivationPotential())$
$SendToChild(max\_child, msg)$

Algorithm 4 shows the *OR* goal node activation spreading method. The *OR* node specifically operates on the *activation potential* of its children. This produces the effect of activating the child with the highest *activation potential*. Only the child with the highest *activation potential* is activated by the *OR* goal node.

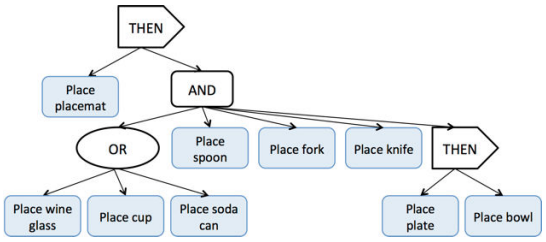


Fig. 2: Task representation for setting up the table

*1) Activation Potential Spreading:* Each goal node has its own method for spreading *activation potential* as can be seen in Algorithms 5, 6, 7. These methods provide a complete understanding of the bottom-up activation spreading framework. As each *behavior node* sends *activation potential* to its parent, each *goal node* will apply a function on these potentials to spread up to the eventual root node of the task tree.

**Algorithm 5** OR - Update Activation Potential

$max \leftarrow max(ActivationPotential(children))$
$activation\_potential = max$

The *OR* goal node only spreads the *activation potential* of the single child with the highest *activation potential*. *OR* nodes will only activate one child. This means the child with the highest *activation potential* is representative of the *OR* node's potential.

**Algorithm 6** AND - Update Activation Potential

$activation\_potential = \frac{sum(child.activation\_potential)}{number\_of\_children}$

The *AND* goal node spreads the average *activation potential* of its children. All children of an *AND* goal node must be executed. This limits the *activation potential* an *AND* node has.

**Algorithm 7** THEN - Update Activation Potential

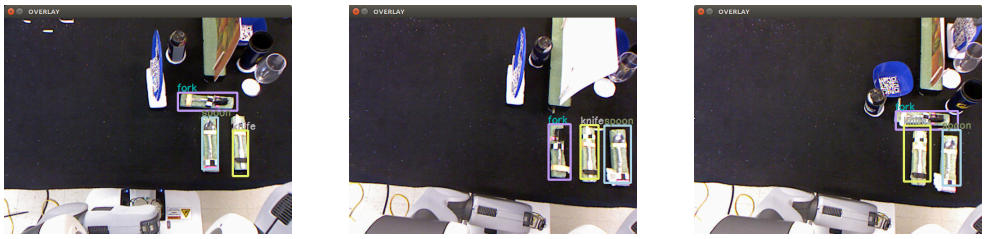
$activation\_potential = \frac{sum(child.activation\_potential)}{number\_of\_children}$

The *THEN* goal node, similar to *AND* goal nodes, spread the average *activation potential* of its children.

### *E. Basic Behavior Representation*

In our particular experimental domain, all the basic behaviors involve object manipulation: picking up and placing objects at particular locations. Therefore, we created a generic *PickAndPlace* node, which encompasses all pick and place behaviors, who inherit their properties from the *PickAndPlace* behavior. Algorithms 8, 9 illustrate the function of this behavior node in the activation spreading framework.

The *PickAndPlace* behavior's *activation potential* is defined as one over the euclidean distance of the object. The *PickAndPlace* 3D arm position represents the PR2's current




(a) Scenario 1 configuration. (b) Scenario 2 configuration. (c) Scenario 3 configuration.

Fig. 3: The three scenario object configurations.

**Algorithm 8** PickAndPlace - Update Activation Potential

1: $\vec{x}_{obj} = \{3D\ object\ position\}$
2: $\vec{x}_{arm} = \{3D\ arm\ position\}$
3: $activation\_potential = \frac{1}{||\vec{x}_{obj} - \vec{x}_{arm}||}$

arm position. The effect of using one over euclidean distance means that the closest object to the PR2 arm will have a higher probability of gaining access to the mutex first given there is a race condition.

**Algorithm 9** PickAndPlace - Precondition

1: **if** $ObjectInView()$ **then**
2: **return** $arm\_mutex.Lock()$
3: **end if**
4: **return** $false$

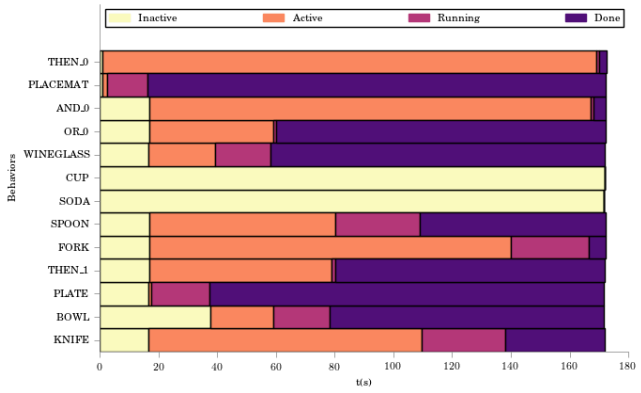
Prior to a node going into the *running* state the mutex must be acquired. Algorithm 9 outlines this method.

### F. Validity Checking

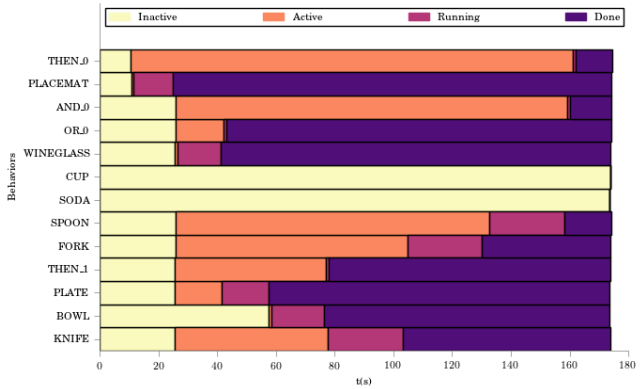
When a *behavior node's* preconditions are satisfied the node's corresponding action will start. The actions are specific for any individual behavior, but the node structure generalizes to any task. In the case of the table setting scenario a *pick-place-cup* behavior is responsible for moving the arm of the robot to the location of the cup and closing the gripper. This action is time-extended and will occur over the span of several seconds. If the state of the environment were to change during this time, the execution of the behavior would fail. The validity checking module is responsible for checking the current state of the environment during the behavior execution and for signaling a reset of the behavior when changes occur that would adversely impact the execution of the behavior. For example, in the case of grasping an object from the table, the validity checking monitors the location of the object and determines if it has changed position during grasp planning. If a significant environmental state change occurs, a signal is sent to stop the behavior's execution and reset the state of the behavior.

## IV. Experimental Evaluation

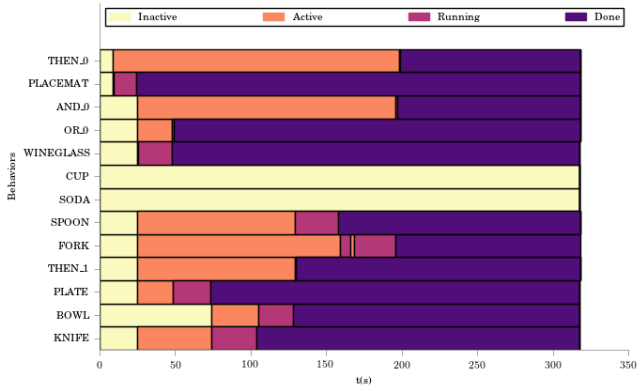
We evaluated our work using a PR2 humanoid robot working on the task of setting up a dinner table. The objects




(a) Scenario 1 execution graph.

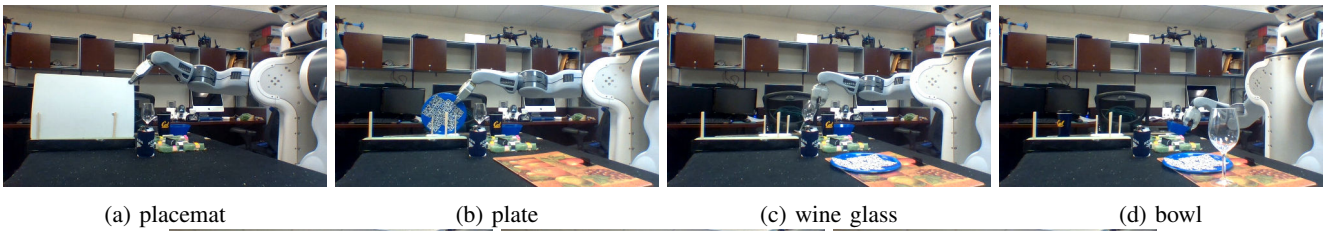



(b) Scenario 2 execution graph.

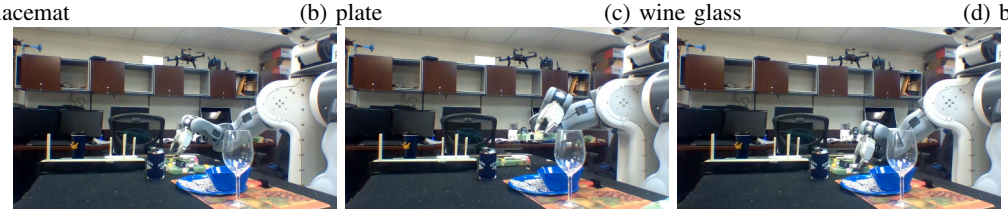



(c) Scenario 3 execution graph.

Fig. 4: Three scenarios with different environment setups where the table objects are placed in different configurations.



(a) placemat (b) plate (c) wine glass (d) bowl



(e) spoon (f) knife (g) fork

Fig. 5: Stages from a task execution.

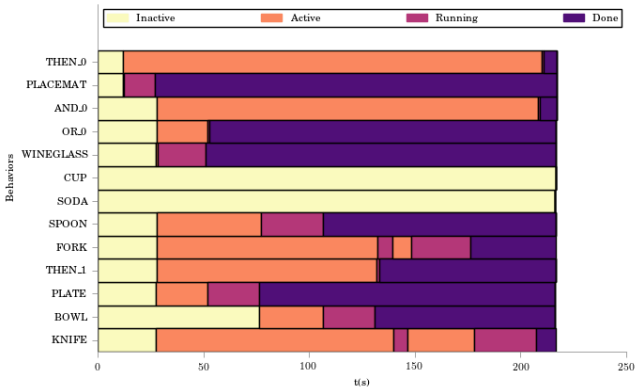



Fig. 6: Scenario 4: Robustness execution demonstration.

that the robot can use for this task include: fork, spoon, knife, wine glass, cup, soda can, placemat, plate, and bowl. A 2D tracking method was used to maintain identity of the dynamic objects in the scene. The TLD tracker in [18] was the algorithm used to track the objects in the scene as it is robust to occlusions. The implementation was from the OpenCV tracking library [19]. The OMPL library was used for arm navigation solutions [20]. [20] is responsible for all arm navigation during the experiments.

We created a task representation that encapsulates all the constraints that occur in a table setting scenario, as shown in Fig. 2. The task structure encodes sequential constraints (e.g., setting up the placemat before the plate), alternative paths of execution (e.g., choose either the wine glass, the cup or the soda can) and steps whose ordering is not important (e.g., placing the spoon, the fork and one of the drinking objects).

For evaluation, we created three different setups, in which the objects are placed in different locations on the table. The layout of the objects for each of these scenarios is shown in Fig. 3. At execution time, the robot uses the locations of the objects and its activation spreading mechanism to dynamically decide which parts of the task to perform and in which order. As currently implemented, the objects that are closer will have a higher activation potential than objects farther away, so the robot will choose to first handle the objects that are at a shorter distance, while still maintaining the ordering constraints represented in the task structure.

For each experiment we recorded the following information: 1) the activation state (*active*, *running*, *done*) for each low-level behavior throughout the duration of the experiment and 2) the times during which the behaviors were in each of these states. The different scenarios were created to show the variety of ways in which the robot could complete the task, utilizing the same task representation and its activation spreading mechanism. Fig. 4 shows the execution results of three scenarios.

The different color bars represent the times during which a particular behavior is in one of the following states: inactive, active, running or done. The intervals corresponding to the *running* state show when a particular pick and place behavior has been executed and are thus indicative of the order in which various task steps have been performed. As seen from the plots of the activity of behaviors, the robot chooses different ways of completing the same task, while obeying the constraints set in the task representation. Due to the constraints set by the top-level *THEN* node, in all scenarios the robot picked up and put the placemat first. In scenario 1, this was followed by the plate, the wine glass, the bowl, the spoon, the knife and the fork. In scenario 2, the order of steps after the placemat was the wine glass, the plate, the bowl, the knife, the fork and then the spoon. For scenario 3, after putting the placemat the robot chose the wineglass, the plate, the knife, the bowl, the spoon and ultimately the fork.

The results show that with the same compact network structure, solely through the activation spreading mechanism and based on the current environmental conditions, the robot will select its own way of accomplishing the task. Fig. 5 shows stages of the task execution for Scenario 1. Each subfigure shows the robot grasping each of the objects prior to setting them in the proper location.

To demonstrate the robustness of the architecture, we interfered with the robot's execution, by moving objects during the time the robot was trying to reach for and

grasp them. The *validity checking* mechanism described in Section III-F detected the change, stopped the currently running behavior, released the arm mutex, and reset the behavior. This is shown in the behavior execution diagram in Fig. 6, where the fork behavior was initially interrupted at approximately 130 seconds into the scenario, when the fork was moved. The robot chose to move on to the knife behavior and revisit placing the fork later. The robot was interrupted again, this time during the knife behavior. The robot decided to complete setting the fork and then set the knife. Since the task network encodes all possible contingencies (through the OR nodes in the representation), our architecture will be able to handle all types of environmental changes that can be handled by choosing an alternate way of performing the task.

## V. Conclusion

We presented a new control architecture that allows for efficient encoding of tasks with multiple paths of execution. The representation used for encoding the task structure serves at the same time as a controller that the robot can use for executing the task. Based on this task representation and an *activation spreading* mechanism within the nodes of the task, the robot can dynamically select which path of execution to perform, based on the current environmental conditions. We validated our approach on a physical humanoid PR2 robot, working on the task of setting the table, in different scenarios that resulted in different ways of execution for the task. Furthermore, we showed that the architecture is robust to failure when environmental changes are occurring during task execution, enabling the robot to pursue alternative ways of executing the task.

## Acknowledgments

This work has been supported in part by ONR award N-00014-15-1-2212 and by NASA EPSCoR under Cooperative Agreement No. NNX11AM09A.

## References

[1] K. P. Hawkins, N. Vo, S. Bansal, and A. F. Bobic, "Probabilistic Human Action Prediction and Wait-sensitive Planning for Responsive Human-robot Collaboration."

[2] K. Erol, J. A. Hendler, and D. S. Nau, "UMCP: A Sound and Complete Procedure for Hierarchical Task-network Planning." in *AIPS*, vol. 94, 1994, pp. 249–254.

[3] A. Koppula, Hema S.and Jain and A. Saxena, *Anticipatory Planning for Human-Robot Teams*. Cham: Springer International Publishing, 2016, pp. 453–470.

[4] T. Asfour, P. Azad, N. Vahrenkamp, K. Regenstein, A. Bierbaum, K. Welke, J. Schrder, and R. Dillmann, "Toward humanoid manipulation in human-centred environments," *Robotics and Autonomous Systems*, vol. 56, no. 1, pp. 54 – 65, 2008, human Technologies: Know-how.

[5] E. Ovchinnikova, M. Wachter, V. Wittenbeck, and T. Asfour, "Multi-purpose natural language understanding linked to sensorimotor experience in humanoid robots," in *Humanoid Robots (Humanoids), 2015 IEEE-RAS 15th International Conference on*, Nov 2015, pp. 365–372.

[6] M. Beetz, U. Klank, I. Kresse, A. Maldonado, L. Mösenlechner, D. Pangercic, T. Rühr, and M. Tenorth, "Robotic Roommates Making Pancakes," in *11th IEEE-RAS International Conference on Humanoid Robots*, Bled, Slovenia, October, 26–28 2011.

[7] J. S. Penberthy, D. S. Weld, and Others, "UCPOP: A Sound, Complete, Partial Order Planner for ADL." *Kr*, vol. 92, pp. 103–114, 1992.

[8] K. Erol, J. Hendler, and D. S. Nau, "HTN planning: Complexity and expressivity," in *AAAI*, vol. 94, 1994, pp. 1123–1128.

[9] P. Maes, "How to do the right thing," *Connection Science*, vol. 1, no. 3, pp. 291–323, 1989.

[10] B. a. Towle and M. Nicolescu, "An auction behavior-based robotic architecture for service robotics," *Intelligent Service Robotics*, vol. 7, pp. 157–174, 2014.

[11] M. N. Nicolescu and M. J. Matarić, "A Hierarchical Architecture for Behavior-Based Robots," in *Proc., First Intl. Joint Conf. on Autonomous Agents and Multi-Agent Systems*, Bologna, Italy, jul 2002, pp. 227–233.

[12] M. N. Nicolescu and M. J. Matari, "Natural Methods for Robot Task Learning: Instructive Demonstration, Generalization and Practice," in *Proc., Second Intl. Joint Conf. on Autonomous Agents and Multi-Agent Systems*, Melbourne, Australia, jul 2003.

[13] M. Nicolescu, O. Chadwicke Jenkins, A. Olenderski, and E. Fritzinger, "Learning behavior fusion from demonstration," *Interaction Studies*, vol. 9, pp. 319–352, 2008.

[14] K. W. Wong, R. Ehlers, and H. Kress-Gazit, "Correct High-level Robot Behavior in Environments with Unexpected Events," in *Proceedings of Robotics: Science and Systems*, Berkeley, USA, jul 2014.

[15] R. C. Arkin, *Behavior-based robotics*. MIT press, 1998.

[16] K. P. Hawkins, S. Bansal, N. N. Vo, and A. F. Bobick, "Anticipating human actions for collaboration in the presence of task and sensor uncertainty," in *2014 IEEE International Conference on Robotics and Automation (ICRA)*. IEEE, may 2014, pp. 2215–2222.

[17] M. Quigley, K. Conley, B. P. Gerkey, J. Faust, T. Foote, J. Leibs, R. Wheeler, and A. Y. Ng, "ROS: an open-source Robot Operating System," in *ICRA Workshop on Open Source Software*, 2009.

[18] Z. Kalal, K. Mikolajczyk, and J. Matas, "Tracking-learning-detection," *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, vol. 34, no. 7, pp. 1409–1422, 2012.

[19] G. Bradski, *Dr. Dobb's Journal of Software Tools*, 2000.

[20] I. A. Şucan, M. Moll, and L. E. Kavraki, "The Open Motion Planning Library," *IEEE Robotics & Automation Magazine*, vol. 19, no. 4, pp. 72–82, December 2012, http://ompl.kavrakilab.org.