What am I doing? Automatic Construction of an Agent's State-Transition Diagram through Introspection

Constantin Berzan
Department of Computer Science
Tufts University
Medford, MA 02155, USA
constantin.berzan@tufts.edu

Matthias Scheutz
Department of Computer Science
Tufts University
Medford, MA 02155, USA
mscheutz@cs.tufts.edu

ABSTRACT

Infrastructures for implementing agent architectures are currently unaware of what tasks the implemented agent is performing. Such knowledge would allow the infrastructure to improve the agent's autonomy and reliability. For example, the infrastructure could detect abnormal system states, predict likely faults and take preventive measures ahead of time, or balance system load based on predicted computational needs. In this paper we introduce a learning algorithm to automatically discover a state-transition model of the agent's behavior. The algorithm monitors the communication between architectural components, in the form of function calls, and finds the frequencies at which various functions are polled. It then determines the states according to what polling frequencies are active at any time. The two main novel features of the algorithm are that it is completely *unsupervised* (it requires no human input) and *task-agnostic* (it can be applied to any new task or architecture with minimal effort).

Categories and Subject Descriptors

I.2.9 [**Artificial Intelligence**]: Robotics; I.2.6 [**Artificial Intelligence**]: Learning

General Terms

Algorithms, Experimentation

Keywords

introspection, state-transition model, unsupervised

1. INTRODUCTION

The architectures of robotic agents are often implemented in some middleware or software infrastructure [7]. The infrastructure's purpose is to abstract over hardware details and provide various advanced services to the architecture, such as automatic distribution of components over different computational resources, location-independent service discovery, communication with remote components, and various others. Infrastructures might have mechanisms to monitor their distributed network of components (e.g. to automatically restart crashed components) [10]. But infrastructures do not "know" what tasks the implemented agent is performing. Such knowledge could improve an agent's reliability and autonomy, especially in long-term sustained operations. For example, the infrastructure could detect abnormal system states, predict likely faults and take preventive measures ahead of time, or balance system load based on predicted computational needs.

The challenge is to obtain the knowledge needed to predict the agent's behavior. One possibility is for the designer to explicitly represent all possible system states. This is clearly difficult even for fairly small systems, such as a robot that performs navigation tasks. Moreover, such a description might fail to specify how the agent reacts to environmental contingencies, such as the appearance of an obstacle. Hence, in addition to the overall description of the system, some kind of learning component is necessary to integrate information about how environmental factors influence the agent's behavior.

Since the infrastructure would need an online learning component anyway, the other possibility would be for the infrastructure to *discover* the entire operation of the implemented agent, without any need to specify abstract system behavior or relevant system states. This means that the infrastructure would have to extract state information from the agent's internal, *subjective* perspective only, since it does not have access to any external, objective information such as the agent's global coordinates.

In this paper we introduce an unsupervised agent-centric learning algorithm to automatically discover a model of the agent's behavior. The algorithm monitors the communication patterns among architectural components, and builds a state diagram that reflects the agent's task model. We start with some background on related approaches for learning behavioral models. We then introduce our proposed approach, and demonstrate its operation in several robotic tasks implemented in the ADE infrastructure [10]. We show that the state diagrams generated automatically from the agent's internal perspective can nicely match state diagrams created manually from an external observer's perspective. We then discuss properties and shortcomings of the proposed method, provide a summary of our contributions, and outline future steps for improving results and performing larger-scale evaluations.

2. BACKGROUND

Conventional methods for modeling the behavior of an autonomous agent are based on observing the agent from

Appears in: *Proceedings of the 11th International Conference on Autonomous Agents and Multiagent Systems (AAMAS 2012)*, Conitzer, Winikoff, Padgham, and van der Hoek (eds.), 4-8 June 2012, Valencia, Spain.

an external perspective. This *external approach* is based on methods used by ethologists in describing animal behavior [5]. First, we observe the agent and determine a set of low-level actions that it performs. For a rodent, these could include resting, walking, grooming, eating, and drinking [6]. Once the low-level action repertoire is defined, we describe the environmental conditions that cause the agent to switch between actions. This way we obtain a state diagram of the agent's behavior, where the states are the low-level actions, and the transitions are the observable conditions that cause action switches.

The behavior of an autonomous agent is typically represented as some variation of a Hidden Markov Model (HMM). If the set of states is known in advance and we have a training sequence of observations, we can learn the parameters of a HMM using algorithms such as Baum-Welch [1, 9]. For example, Guillory et al. [5] learn a model of the agent's behavior using Input/Output HMMs. Their approach requires as input a set of possible perceptions for the agent, and human-labeled example trajectories. Similarly, Delmotte and Egerstedt [2] learn simple control programs from externally-observed data. Goldberg and Mataric [4] present an algorithm for learning Augmented Markov Models (AMMs), which are similar to HMMs, but allow each state to produce a single symbol. Their algorithm takes a sequence of symbols and processes it online to update an initially empty AMM.

The strength of the external approach is that the state labels are directly meaningful to human beings, because the model was built based on observed behavior. On the downside, the external approach always requires some amount of domain-specific knowledge, such as a model of the agent's perceptions, or labels for an execution trajectory. Labeling can be difficult when the observer's action abstraction does not directly correspond to the agent's internal action representation. For example, a robot may use several different actions to follow a corridor, depending on its current goal. But an external observer might treat all these actions as the same. Similarly, an external observer might discriminate among multiple actions (e.g. approach wall, turn away from wall) even though the agent's control system does not discriminate among them (e.g. because the agent uses a potential-based approach for traversing hallways). As a result, the state-transition diagram will contain states that have no counterpart in the agent control system. Furthermore, the external approach treats the agent like a black box, and will miss unobservable state changes (such as perceiving a door and storing its location for future exploration). The external approach will also miss any behaviors exhibited outside of the observation period. And, in general, it might not be possible to observe the agent in certain situations (e.g. a cleaning robot in the sewage system).

To overcome some of these complications we turn to the *internal approach*, where the observer is the infrastructure in which the agent control system is implemented. Unlike any external observer, the infrastructure has exact information about component interactions. The agent is no longer a black box. On the other hand, the infrastructure usually has no information about the functional role of these components in the agent architecture. Wallace [12] discusses the advantages of self-assessment (internal monitoring) as opposed to external monitoring for detecting runtime errors. Other unsupervised approaches for acquiring a model of the world and self are being explored in the field of developmental robotics [13, 11]. Our work is most similar to the robot-introspection work of Fox et al. [3], which learns a HMM of the robot's behavior from raw sensor data. They start with a set of human-labeled states, which they then refine. Our method requires no labeling whatsoever for model construction, but only for evaluation. Furthermore, the variables and sensory features used by Fox et al. have to be chosen by hand for each task, whereas the log data we use requires no human pre-processing. Also, Fox et al. require multiple runs of the same task to learn the model, whereas our method allows building a model from a single run.

The ADE infrastructure mediates communication between the agent's components. During task execution we can record the interaction between components, in the form of function calls. The patterns in the recorded call log can be used to define states, which can then be organized in a state-transition model. These states will reflect the agent's control system, and as discussed above, they will not necessarily correspond to states described by an external observer. In general, the best we can hope for from the internal approach is that it will come reasonably close to an external model built from human observations. We are looking for a middle ground between a model that is overly specific (too many states) and one that is overly general (too few states). If this goal can be achieved, the internal approach will have several advantages over the external approach. First, internal modeling does not require any human labeling effort. This means that it can easily be applied to new tasks or architectures, and it can, in principle, run online while the robot is performing its task. Second, the model corresponds closely to the actual control flow of the robot, so there is no risk of "cheating" by imposing structure that is not really there (which an external observer might be tempted to do). Finally, and most importantly, the robot itself knows what state it is in, and it can use its own model to make predictions. This can enable the robot to balance load or to predict failures before they happen.

3. METHOD

Our state extraction algorithm is based on the following key observation: Throughout the execution of the task, the architecture polls various functions at regular intervals. For example, while going through a hallway, the robot might poll the `checkMotion` function, but not the `getLaserReadings` function. When entering a door, the robot might poll `getLaserReadings`, but not `checkMotion`. We associate each distinct polling pattern with a state. The states detected this way are grounded solely in subjective data collected internally by the robot.

To explain our method, we use a short example task, where the robot turns inside a room, and then exits through the door and into the hallway. The input data is an execution log, containing a list of function calls. Each call is identified by a time stamp, a component and function name, and a set of arguments (which we currently ignore). Figure 2a shows some sample log entries.

We first determine the possible *polling frequencies* for each function. Then, we identify the time intervals when each polling frequency is active (we call these the *instances* of a given polling frequency). We define a *state* as a set of active polling frequencies. We use the detected instances to determine the states, and the state-transition history. Finally, we prune the state-transition history to remove superfluous states, and we use the pruned transition history to construct a state diagram. We proceed to describe this process in greater detail, together with the parameters that each step requires. It is helpful to follow Figure 2 while reading the rest of this section.

3.1 Polling frequencies

We start by computing the time difference between pairs of consecutive calls to the same function. If there are n calls to function `f`, we get $n-1$ values, which we call the *cadence values* for `f`. If some caller polls `f` every 100ms, we expect to see a cluster of cadence values around 100ms. Figure 2b shows a histogram plot of the cadence values for our example task. We can see clusters for `getLaserReadings`, `checkMotion`, `updateMoveToRel`, and `getPlan`. To extract these clusters, we make a single-linkage hierarchical clustering[1] of the cadence values for each function. We then put two cadence values in the same flat cluster if the difference between them is less than a threshold D_{max}. If a cluster is supported by less than N_{min} cadence values, we discard it. Figure 2c shows the resulting clusters (the clusters are ranges of values, and we identify them by their center). In general, there can be more than one cluster for any given function. We refer to a function (e.g. `getLaserReadings`) together with one of its clusters (e.g. 327ms) as a *polling frequency* (e.g. `getLaserReadings` at 327ms).

This step has the following parameters:

- D_{max}: maximum difference between adjacent cadence values for them to belong to the same cluster
- N_{min}: minimum number of cadence values to constitute a polling frequency

3.2 Instances of each polling frequency

In the next step, we want to find all *instances* when a given polling frequency is active. For example, for the polling frequency "`f` at 200ms," we want to find all time intervals (t_{begin}, t_{end}) when `f` is being polled at 200ms. To find these intervals, we sweep over the logged calls in the order they occurred, keeping track of which polling frequencies are possible at each point. If over an interval of time, we see at least C_{min} calls to `f`, and the time difference between each consecutive pair of calls is within a tolerance T of 200ms, we save an instance for this polling frequency. Since calls that are slightly off-time happen often, we forgive an early or late call, if the next call is on time. Figure 2d shows a timeline of the logged calls. Figure 2e shows the detected instances for each polling frequency.

This step has the following parameters:

- C_{min}: minimum number of calls to constitute an instance
- T: tolerance with respect to a polling frequency (for a call to be counted towards an instance)

3.3 States and the state diagram

We define a *state* to be a set of active polling frequencies. We extract states from the instances detected in the previous step. At each point in time, a given set of polling frequencies is active, and if this set does not correspond to an existing state, it becomes a new one. Figure 2f shows the states and state-transition history discovered this way. Because instances of different polling frequencies are never perfectly aligned, this process results in a lot of superfluous states, in which very little time is spent. We therefore prune the state-transition history, discarding all state visits shorter than V_{min}. The time spent in a discarded state is redistributed to the previous and next state in the transition history. Figure 2g shows the states and state-transition history after pruning.

This step has a single parameter:

- V_{min}: minimum time spent in a state (for pruning)

Finally, we use the pruned state-transition history to build a state diagram of the task, shown in Figure 1. We indicate the start state with an arrow, and the final state with a double border.

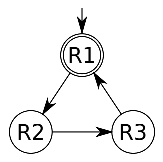


Figure 1: The state diagram obtained for the example task.

We use this simple way of constructing a diagram to illustrate the results of our state-extraction method. The state-transition history obtained in the final step (Figure 2g) is simply a sequence of states, which we could use to train a HMM or AMM if desired.

4. EXPERIMENTAL RESULTS

Evaluation is one of the main difficulties of robot introspection. Because the learned model is grounded in the robot's subjective observations, there is no "ground truth" that we can use for a direct comparison. Instead, as pointed out by Fox et al. [3], we are forced to compare the learned model with a human observer's *interpretation* of what the robot actually did. We cannot escape the limitation that interpretations may differ across observers.

We evaluate our algorithm by looking for bisimilarity between the constructed state diagram D_R and another state diagram D_O, representing a human observer's interpretation of the robot's behavior. Bisimilarity (or bisimulation) is a relationship between two state-transition systems that behave in the same way, in the sense that one system simulates the other and vice-versa. Park [8] provides a technical definition in the context of automata theory.

We have tested our state-extraction algorithm on three different tasks, using the ADE simulator. In the Boxes task, the robot moves objects from several source boxes into a destination box. In the Hallway task, the robot traverses a hallway looking for wounded people in every open room. The Combined task is a composition of the two tasks above. The robot traverses a hallway and performs the Boxes task inside each room.

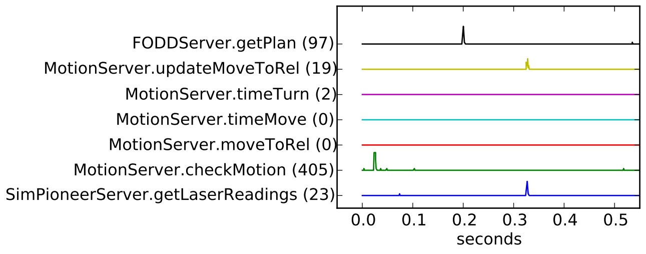
We used the following parameters in our experiments: $D_{max} = 5$ms, $N_{min} = 10$, $C_{min} = 3$, $T = 20$ms, $V_{min} = 1.3$s. For each task, we performed the following steps:

1. Run the task, generating the log and recording a screencast of the simulator window.

[1]We also tried k-means, RANSAC, and Gaussian Mixture Model fitting, but the simple hierarchical clustering worked best.

system time		component name	function name	arguments
1306509565603	CALL:	com.motion.MotionServer	moveToRel	0.0020581365076 0.890346846488
1306509565878	CALL:	com.adesim.SimPioneerServer	getLaserReadings	
1306509573546	CALL:	com.motion.MotionServer	checkMotion	1306509573522

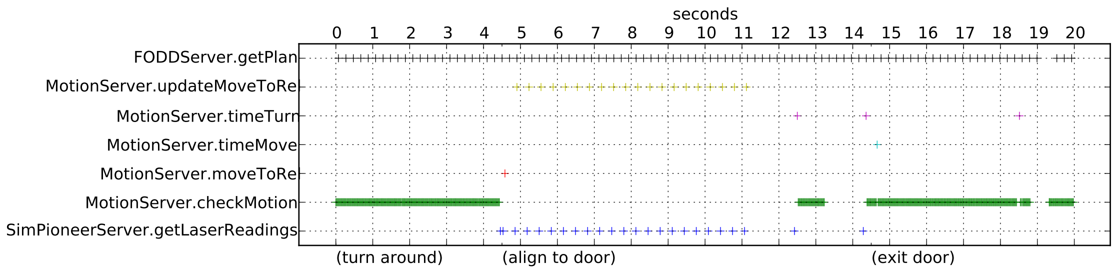
(a) Sample log entries



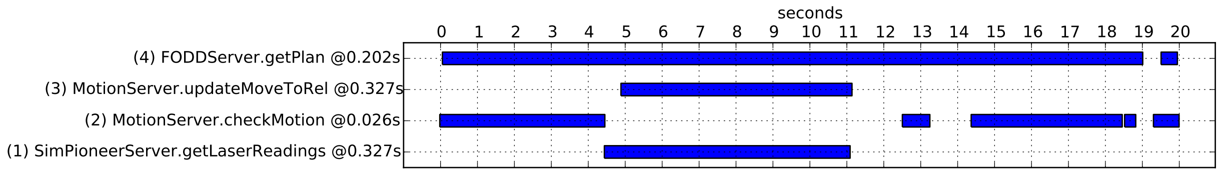
(b) Cadence histogram. The number of cadence values for each function is shown in parentheses.

function	polling frequencies
getLaserReadings	327ms
checkMotion	26ms
updateMoveToRel	327ms
getPlan	202ms

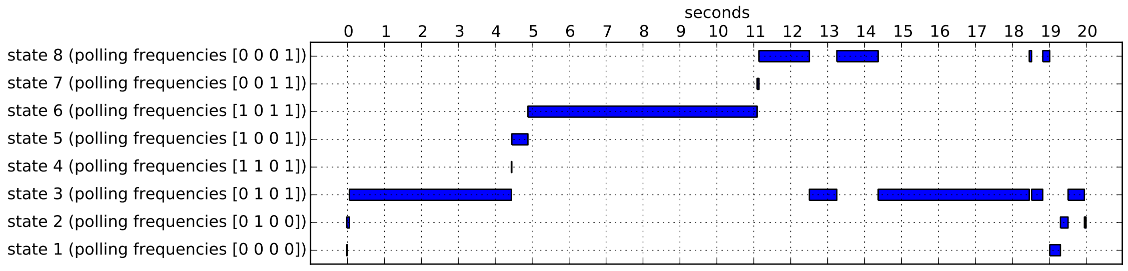
(c) Detected polling frequencies



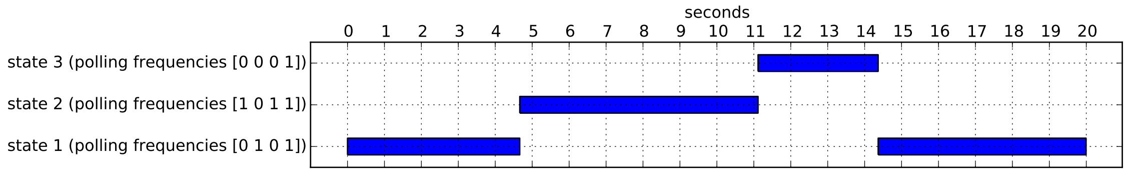
(d) Timeline of function calls, with the observed events shown on the bottom (the algorithm does not use these labels)



(e) Detected instances for each polling frequency



(f) States and state-transition history before pruning. The states are identified by a set of active polling frequencies. For example, state 3 is marked [0 1 0 1], which means the second (`checkMotion` at 26ms) and fourth (`getPlan` at 202ms) polling frequencies are active.



(g) States and state-transition history after pruning

Figure 2: Our method, described in section 3. From the log, we obtain the cadence values (b), which we cluster to obtain the polling frequencies (c). We then traverse the log (d), find the instances of each polling frequency (e), determine states based on what polling frequencies are active (f), and finally prune the state-transition history (g).

2. Watch the screencast and create a state diagram based on the behavior we observe. We call this the observer's state diagram, D_O.
3. Run the state-extraction algorithm on the robot's log, obtaining an unlabeled state diagram.
4. Label the state diagram given by the algorithm, by watching the screencast and observing the robot's behavior during each of the detected states. We call the result the robot's state diagram, D_R.
5. Create an expanded state diagram, D_E, such that every state in D_E corresponds to exactly one state in D_O, and exactly one state in D_R. (We view D_O and D_R as two different ways to decompose the robot's behavior into states, and D_E as their common denominator.)
6. Show that D_R and D_O are bisimilar, by defining a function f mapping states in D_O to states in D_E, and a function g mapping states in D_R to states in D_E.

4.1 The Boxes Task

In this task, the robot is in a room with a destination box and several source boxes, which may be empty or contain a single object. The robot first does a 360-degree sweep to determine the location of the boxes. Then it visits each source box, and if it finds an object inside, it carries the object to the destination box. We stop the run after the robot visits four source boxes. Figure 3 depicts the robot's environment and trajectory.

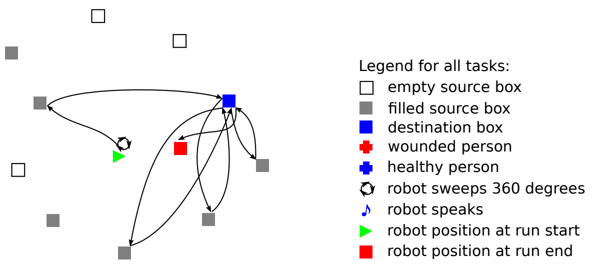


Figure 3: Map of the Boxes task, with legend.

The robot's state diagram D_R has two states, depicted in Figure 6a, and the observer's state diagram D_O has six states, shown in Figure 6b. For this task, the expanded state diagram D_E is identical to D_O. The full state-transition histories for D_R and D_O are given in Figure 6c. The bisimilarity between D_R and D_O is given by:

$$f = \{R_1 \mapsto \{O_1, O_3, O_5\}, R_2 \mapsto \{O_2, O_4, O_6\}\}$$

4.2 The Hallway Task

In this task, the robot traverses a hallway with several rooms. Upon seeing an open door, the robot enters the room and does a 360-degree sweep, looking for wounded people. If it finds a wounded person, the robot pauses and sends a spoken message to the operator. It then exits the room and continues traversing the hallway. We stop the run after the robot visits three rooms. Figure 4 depicts the robot's environment and trajectory. (Please consult the legend on the right side of Figure 3.)

The robot's state diagram D_R has six states, depicted in Figure 7a. The observer's state diagram D_O has five states, depicted in Figure 7c. The expanded state diagram D_E has eleven states, shown in Figure 7b. The full state-transition histories for D_R, D_E, and D_O are given in Figure 7d. The bisimilarity between D_R and D_E is given by:

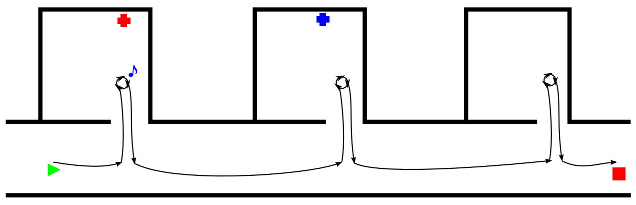


Figure 4: Map of the Hallway task.

$$f = \{R_1 \mapsto \{E_1\}, R_2 \mapsto \{E_2\}, R_3 \mapsto \{E_3\}, R_4 \mapsto \{E_4, E_9\}, R_5 \mapsto \{E_5, E_7, E_{10}\}, R_6 \mapsto \{E_6, E_8, E_{11}\}\}$$

and the bisimilarity between D_O and D_E is given by:

$$g = \{O_1 \mapsto \{E_1, E_2\}, O_2 \mapsto \{E_3, E_4, E_5, E_6\}, O_3 \mapsto \{E_7\}, O_4 \mapsto \{E_8\}, O_5 \mapsto \{E_9, E_{10}, E_{11}\}, \}$$

4.3 The Combined Task

This task is a composition of the Hallway and Boxes tasks. The robot traverses a hallway with several rooms. Upon seeing an open door, the robot enters the room and performs the Boxes task: It does a 360-degree sweep to find any boxes, and carries any objects from source boxes to the destination box. When finished (or if no source boxes are found), the robot exits the room and continues traversing the hallway. The run starts with the robot sweeping the first room, and it ends after the robot exits the third room. Figure 5 depicts the robot's environment and trajectory. (Please consult the legend on the right side of Figure 3.)

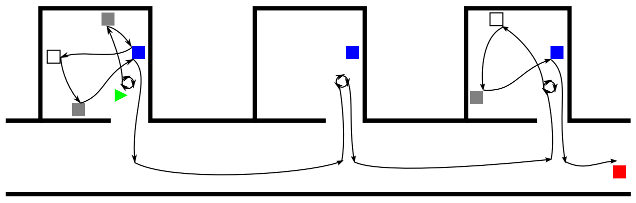


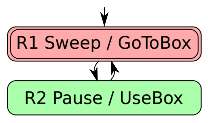
Figure 5: Map of the Combined task.

The robot's state diagram D_R has six states, depicted in Figure 8a. The observer's state diagram D_O has ten states, depicted in Figure 8c. The expanded state diagram D_E has sixteen states, shown in Figure 8b. The full state-transition histories for D_R, D_E, and D_O are given in Figure 9. The bisimilarity between D_R and D_E is given by:

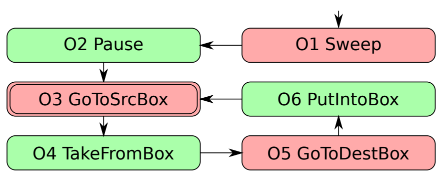
$$f = \{R_1 \mapsto \{E_2, E_4, E_6, E_9, E_{15}\}, R_2 \mapsto \{E_1, E_3, E_5, E_7, E_{10}, E_{16}\}, R_3 \mapsto \{E_8, E_{14}\}, R_4 \mapsto \{E_{11}\}, R_5 \mapsto \{E_{12}\}, R_6 \mapsto \{E_{13}\}\}$$

and the bisimilarity between D_O and D_E is given by:

$$g = \{O_1 \mapsto \{E_1\}, O_2 \mapsto \{E_2\}, O_3 \mapsto \{E_3\}, O_4 \mapsto \{E_4\}, O_5 \mapsto \{E_5\}, O_6 \mapsto \{E_6\}, O_7 \mapsto \{E_7\}, O_8 \mapsto \{E_8, E_9, E_{10}\}, O_9 \mapsto \{E_{11}, E_{12}\}, O_{10} \mapsto \{E_{13}, E_{14}, E_{15}, E_{16}\}\}$$



(a) D_R: robot's state diagram

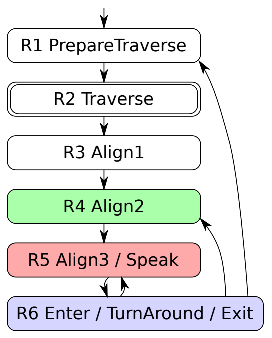


(b) D_O: observer's state diagram

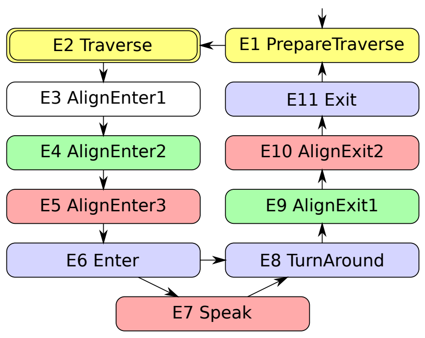
R_1	R_2	R_1	R_2	R_1	R_2	R_1	R_2	R_1	R_2	R_1	R_2	R_1	R_2	R_1	R_2	R_1	R_2	R_1
O_1	O_2	O_3	O_4	O_5	O_6	O_3	O_4	O_5	O_6	O_3	O_4	O_5	O_6	O_3	O_4	O_5	O_6	O_3

(c) State-transition history in D_R (top) and D_O (bottom). Time moves from left to right (not drawn to scale).

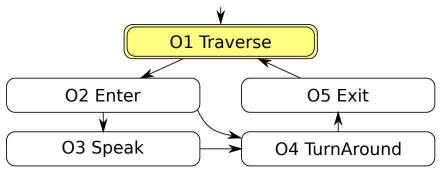
Figure 6: State diagrams for the Boxes task. The start state has a loose arrow coming into it. The end state has a double border. To illustrate bisimilarity, the states corresponding to R_1 are shaded in red, and the states corresponding to R_2 are shaded in green.



(a) D_R: robot's state diagram



(b) D_E: expanded state diagram



(c) D_O: observer's state diagram

R_1	R_2	R_3	R_4	R_5	R_6	R_5	R_6	R_4	R_5	R_6	R_1	R_2	R_3	R_4	R_5	R_6	
E_1	E_2	E_3	E_4	E_5	E_6	E_7	E_8	E_9	E_{10}	E_{11}	E_1	E_2	E_3	E_4	E_5	E_6	E_8
O_1		O_2				O_3	O_4	O_5			O_1		O_2				O_4

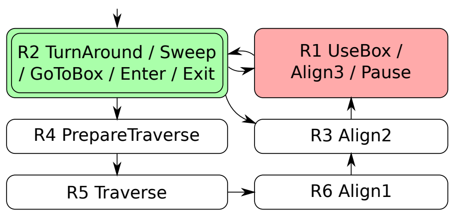
...

...

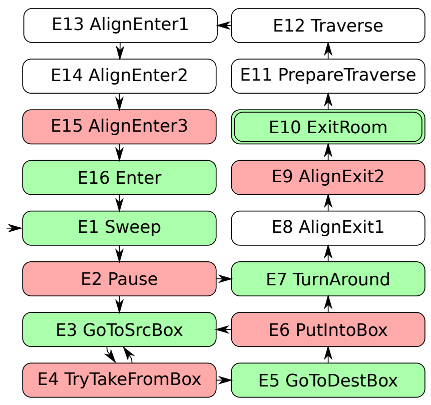
R_4	R_5	R_6	R_1	R_2	R_3	R_4	R_5	R_6		R_4	R_5	R_6	R_1	R_2
E_9	E_{10}	E_{11}	E_1	E_2	E_3	E_4	E_5	E_6	E_8	E_9	E_{10}	E_{11}	E_1	E_2
O_5			O_1		O_2				O_4	O_5			O_1	

(d) State-transition history in D_R (top), D_E (middle), and D_O (bottom). Time moves from left to right (not drawn to scale).

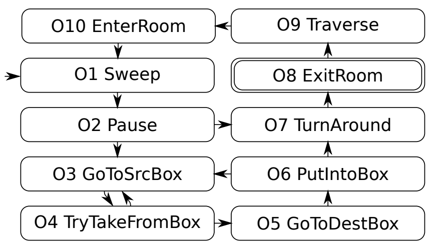
Figure 7: State diagrams for the Hallway task. We illustrate bisimilarity by shading some states in D_R and D_O, and their corresponding states in D_E (R_4 in green, R_5 in red, R_6 in blue, and O_1 in yellow).


R2 TurnAround / Sweep / GoToBox / Enter / Exit
R1 UseBox / Align3 / Pause
R4 PrepareTraverse
R3 Align2
R5 Traverse
R6 Align1

(a) D_R: robot's state diagram



(b) D_E: expanded state diagram



(c) D_O: observer's state diagram

Figure 8: State diagrams for the Combined task. We illustrate bisimilarity by shading R_1 in red and R_2 in green, showing the corresponding states in D_E.

R_2	R_1	R_2	R_1	R_2	R_1	R_2	R_1	R_2	R_1	R_2	R_1	R_2	R_3	R_1	R_2	R_4	R_5	R_6	R_3	R_1	R_2		
E_1	E_2	E_3	E_4	E_3	E_4	E_5	E_6	E_3	E_4	E_5	E_6	E_7	E_8	E_9	E_{10}	E_{11}	E_{12}	E_{13}	E_{14}	E_{15}	E_{16}	E_1	...
O_1	O_2	O_3	O_4	O_3	O_4	O_5	O_6	O_3	O_4	O_5	O_6	O_7	O_8			O_9		O_{10}				O_1	

	R_1	R_2	R_3	R_1	R_2	R_4	R_5	R_6	R_3	R_1	R_2		R_1	R_2	R_1	R_2	R_1	R_2	R_1	R_2	R_3	R_1	R_2
...	E_2	E_7	E_8	E_9	E_{10}	E_{11}	E_{12}	E_{13}	E_{14}	E_{15}	E_{16}	E_1	E_2	E_3	E_4	E_3	E_4	E_5	E_6	E_7	E_8	E_9	E_{10}
	O_2	O_7	O_8			O_9		O_{10}				O_1	O_2	O_3	O_4	O_3	O_4	O_5	O_6	O_7	O_8		

Figure 9: State-transition history for the COMBINED task, in D_R (top), D_E (middle), and D_O (bottom). Time moves from left to right (not drawn to scale).

5. DISCUSSION

We start by discussing how the extracted state diagrams relate to the real world. As noted before, the only information used by our algorithm is the function-call log given by the infrastructure. This means that the algorithm has no knowledge of the *meaning* of a function call, and so cannot derive the meaning of a state, either. Consider the state diagram D_R extracted by the algorithm for the HALLWAY task (Figure 7a). We can imagine a state diagram that has the same number of states, the same arrows, and the same start and end state. But if the states have completely different labels, the semantics of the state diagram would also be radically different. This is the price we pay for being completely agent-centric: Our algorithm has no way to get at the semantics of states.

Despite this limitation, we were able to show how the extracted state diagrams D_R are bisimilar to state diagrams made by an external observer D_O. We did this by building a more fine-grained ("expanded") state diagram D_E, and showing that D_R and D_O are both bisimilar to D_E. This correspondence is encouraging, because it shows that the states detected introspectively often correspond to distinct observable behaviors. Note that bisimilarity is a rather strong requirement, and that we would need something weaker if some state transitions were not detected (e.g. if we usually detect $R_a \to R_b \to R_c$, but sometimes we just detect $R_a \to R_c$).

Comparing the extracted state diagrams with those made by an external observer illustrates some limitations of introspection. For example, in the BOXES task our algorithm was unable to distinguish between the PAUSE, TAKEFROMBOX, and PUTINTOBOX states, and lumped the three together in R_2 (Figure 6a). The reverse can also happen: In the HALLWAY task, our algorithm split the observer's single state ENTER into the sequence $R_3 \to R_4 \to R_5 \to R_6$. This shows that parts of the extracted state diagram can be either more coarse-grained or more fine-grained than what an observer sees.

We now turn to a discussion of our method. Our original idea for extracting states was to slide a fixed-size observation window over the call log, count the number of calls to each function in each observation, and cluster the resulting vectors. This turned out to work very poorly, because the fixed-size window introduced discretization errors, states with a short time span tended to be ignored, and observations taken at the transition between two states gave rise to spurious clusters. Moreover, the number of clusters had to be known in advance. We developed our current method after we realized that most functions were being polled at regular intervals, and that we could take advantage of the polling patterns to distinguish between states. (One could presumably design an architecture that performs no polling whatsoever; in that case our algorithm would need to be adapted.)

Our method has five parameters: two for finding the polling frequencies, two more for detecting the instances of each polling frequency, and one for pruning the state-transition history. For our experiments, we set the parameters by hand, noticing how they affect the algorithm's behavior on the three tasks. For example, if D_{max} or T are too small, then we would most likely miss slow polls (e.g. one call every second). If they are too large, then we might detect spurious polling frequencies and instances. If N_{min} or C_{min} are too large, we might miss states that the robot visits for only a short time. If they are too small, then we would most likely have many spurious state transitions. Finally, V_{min} has to strike a balance between pruning too much (joining states that should be distinct), and pruning too little (leaving spurious states). We believe that the optimal setting for these parameters depends on variables in the infrastructure, such as the call latency.

We briefly sketch the time complexity of each part of our algorithm. The hierarchical clustering to find polling frequencies takes $O(N_c^2 \log N_c)$ time and $O(N_c^2)$ space, where N_c is the number of calls to a given function. Therefore, for N_f, functions that are being polled, finding all polling frequencies takes $O(N_f \cdot N_c^2 \log N_c)$ time. After that, assuming N_{pf} polling frequencies were detected, finding the instances naively takes $O(N_{pf} \cdot N_c)$ time. For N_i instances, extracting the states takes $O(N_i)$ time. If the resulting state-transition history has N_t transitions, pruning it takes $O(N_t^2)$ naively, and can be improved to $O(N_t \log N_t)$ by using a heap.

The initial clustering of cadence values appears to be the bottleneck. In our experiments, the cadence values had millisecond precision. To keep the running time reasonable, it was sufficient to reduce the number of duplicate cadence values to N_{min} (this does not affect the results). For convenience, we used single-linkage hierarchical clustering, and we then created flat clusters based on a distance threshold. It should be possible to replace this expensive part of the algorithm with simply sorting the cadence values, and then traversing them, finding the flat clusters directly, and avoiding the hierarchical clustering altogether. Such an approach would take $O(N_c \log N_c)$ time per function. Based on this back-of-the-envelope analysis, we expect our algorithm to scale well with increasingly complex tasks.

Finally, we discuss how our method could help with fault detection and load balancing. (We plan to evaluate these proposals in future work.) Suppose the robot performs a task for which it has already built a state diagram. If it encounters a previously unseen state transition, the robot could signal to the operator that something unexpected is happening. Using introspection alone, the robot would be unable to distinguish between a fault (in which case it should

state	visit	time spent	
	count	mean	std
R_1	4	3.68 s	1.22 s
R_2	4	14.33 s	7.09 s
R_3	3	2.18 s	0.64 s
R_4	6	4.81 s	1.75 s
R_5	7	2.37 s	0.93 s
R_6	7	17.11 s	6.93 s

Table 1: Mean and standard deviation of the time spent in each state of the HALLWAY task.

notify the operator) and a new state (in which case it should update its internal model). The only way to clarify the situation is to ask the operator: "Am I doing something wrong?" or "Is this supposed to happen?" Spending too much or too little time in a state may also indicate a failure. To detect this, the robot could maintain statistics about the time spent in each state. Table 1 shows the statistics collected during the HALLWAY task. The standard deviations are high, indicating that the time spent in a state depends on environmental features (e.g. the length of hallway between two rooms), and not just on what the robot is doing. This suggests that applying the unmodified AMM-learning algorithm of Goldberg and Mataric [4] could be problematic, because a state with high variance would be split in two.

To perform load balancing, the robot needs to know what components of the architecture are active in each state. We can obtain this information directly from the logs. Knowing its current state and the possible next states, the robot can instantiate components on different hosts to achieve load balancing. It can also conserve energy by suspending a component if it is unlikely to be used in the near future.

6. CONCLUSIONS AND FUTURE WORK

The main contribution of this paper is an algorithm to extract a state diagram of an agent's behavior from the communication patterns of its architectural components. The algorithm monitors function calls between components, and finds the frequencies at which various functions are polled. It then determines the states according to what polling frequencies are active at any time. Unlike external approaches to modeling robot behavior, our algorithm is completely *unsupervised* (it requires no human input) and *task-agnostic* (it can be applied to any new task or architecture with minimal effort).

We evaluated the algorithm in three robotic example tasks. We demonstrated that the state diagrams extracted by the algorithm are bisimilar to state diagrams made by an external observer who watches the robot perform its task. We also discussed how our algorithm can be used for fault detection and load balancing.

The most immediate direction for future work is to make the algorithm work online, which would allow the robot to learn during task execution and take advantage of what it has learned so far. To turn the current version into an online algorithm, we would need to detect new polling frequencies and instances iteratively, without maintaining a full history of every call so far. Using the online version, we intend to demonstrate fault detection and load balancing based on an actual robot.

Another extension to our algorithm is to consider "rare" calls: occasional calls to functions that occur without polling. This should enable us to distinguish among states that are very similar otherwise (e.g. TAKEFROMBOX and PUTINTOBOX). We are also looking into extending our algorithm by incorporating sensor data and function-call arguments, and by building a probabilistic state-transition model, while remaining completely unsupervised and task-agnostic. Finally, it would be interesting to see how much better the state-transition model can get if we are allowed to occasionally ask the operator questions, such as "What am I doing right now?" or "Am I doing the same thing that I was doing a minute ago?".

7. REFERENCES

[1] E. Charniak. *Statistical Language Learning.* Language, Speech, and Communication. MIT Press, 1996.

[2] F. Delmotte and M. Egerstedt. Reconstruction of low-complexity control programs from data. In *43rd IEEE Conference on Decision and Control*, volume 2, pages 1460–1465, 2004.

[3] M. Fox, M. Ghallab, G. Infantes, and D. Long. Robot Introspection through Learned Hidden Markov Models. *Artificial Intelligence*, 170(2):59–113, 2006.

[4] D. Goldberg and M. J. Mataric. Augmented Markov Models, 1999.

[5] A. Guillory, H. Nguyen, T. Balch, and C. L. Isbell. Learning executable agent behaviors from observation. In *Proc. of the Fifth International Joint Conference on Autonomous Agents and Multiagent Systems*, 2006.

[6] H. Jhuang, E. Garrote, X. Yu, V. Khilnani, T. Poggio, A. D. Steele, and T. Serre. Automated home-cage behavioural phenotyping of mice. *Nature communications*, 1(6):68, 2010.

[7] J. Kramer and M. Scheutz. Robotic development environments for autonomous mobile robots: A survey. *Autonomous Robots*, 22(2):101–132, 2007.

[8] D. Park. Concurrency and automata on infinite sequences. In P. Deussen, editor, *Theoretical Computer Science*, volume 104 of *Lecture Notes in Computer Science*, pages 167–183. Springer Berlin / Heidelberg, 1981. 10.1007/BFb0017309.

[9] L. Rabiner. A tutorial on hidden Markov models and selected applications in speech recognition. *Proceedings of the IEEE*, 77(2):257 –286, feb 1989.

[10] M. Scheutz. ADE - Steps towards a distributed development and runtime environment for complex robotic agent architectures. *Applied Artificial Intelligence*, 20(4-5), 2006.

[11] D. Stronger and P. Stone. Towards autonomous sensor and actuator model induction on a mobile robot. *Connection Science*, 18(2):97–119, June 2006.

[12] S. A. Wallace. S-assess: a library for behavioral self-assessment. In *Proceedings of the fourth international joint conference on Autonomous agents and multiagent systems*, AAMAS '05, pages 256–263, New York, NY, USA, 2005. ACM.

[13] J. Weng, J. McClelland, A. Pentland, O. Sporns, I. Stockman, M. Sur, and E. Thelen. Artificial intelligence. Autonomous mental development by robots and animals. *Science*, 291(5504):599–600, January 2001.