Crossing Boundaries: Multi-Level Introspection in a Complex Robotic Architecture for Automatic Performance Improvements

Evan Krause[1] and **Paul Schermerhorn**[2] and **Matthias Scheutz**[1]
Human-Robot Interaction Laboratory
Tufts University[1] and Indiana University[2], USA
{ekrause,mscheutz}@cs.tufts.edu, pscherme@indiana.edu

Abstract

Introspection mechanisms are employed in agent architectures to improve agent performance. However, there is currently no approach to introspection that makes automatic adjustments at multiple levels in the implemented agent system. We introduce our novel multi-level introspection framework that can be used to automatically adjust architectural configurations based on the introspection results at the agent, infrastructure and component level. We demonstrate the utility of such adjustments in a concrete implementation on a robot where the high-level goal of the robot is used to automatically configure the vision system in a way that minimizes resource consumption while improving overall task performance.

Introduction

Self-adjusting agent architectures based on introspection typically either employ component-specific introspection mechanisms or attempt to integrate all levels of self-reasoning and self-adjustment into a single system-wide mechanism (Morris 2007; Haidarian et al. 2010; Sykes et al. 2008; Georgas and Taylor 2008). Component-specific mechanisms have the advantage that they are modular and that components with these mechanisms can be integrated into existing architectures, but they typically lack knowledge about the whole system and how to best manage it. System-wide approaches, on the other hand, break encapsulation by requiring observable data and component implementations from all levels to be exposed to the system, and thus can create performance bottlenecks, aside from making the integration of new components difficult and reducing the opportunity for component reuse.

We believe that using either approach alone misses important opportunities for exploiting synergies of the two approaches to improve agent performance and, therefore, propose to integrate introspection mechanisms at all levels of an agent system. Specifically, we introduce a conceptual framework for multi-level introspection in cognitive agents and describe a concrete instantiation of the framework in a complex robotic architecture which demonstrates the integration of introspection at various levels of organization and their interaction across abstraction boundaries.

Copyright © 2012, Association for the Advancement of Artificial Intelligence (www.aaai.org). All rights reserved.

Motivation and Background

Over the past decade there has been a growing effort to develop various agent systems with self-adjusting mechanisms to enable robust and long term operation in the face of software, hardware and environmental uncertainty. Aaron Morris, for example, presents in his thesis on subterranean autonomous vehicles a system-wide introspection layer that combines observations from both the infrastructure and component-level of the robotic architecture into a single data signature to represent introspective state (Morris 2007). Haidarian et al. (2010) have been developing a domain-independent general introspection mechanism they have dubbed the Meta-Cognitive Loop (MCL). MCL attempts to generalize introspection by boiling down all potential anomalies into a set of general-use anomaly-handling strategies. This approach has only been integrated into systems at the infrastructure level and does not address how or if MCL is intended to be integrated in a distributed fashion.

Edwards et al. (2009) propose a layered hierarchical self-adjusting system that not only allows components to observe and manage themselves (ending the need for all components to report to a central manager), but also allows the adaptive components to themselves be managed by other higher level adaptive components. Unfortunately, there appears to be no method for observed data from low-level components to be used by high-level adaptive components unless the high-level components are continually monitoring the low-level data. There is, in effect, no way for low-level components to notify high-level components when additional assistance is needed.

A great deal of this work has focused on failure detection and recovery (Sykes et al. 2008; Haidarian et al. 2010; Morris 2007), and system reconfiguration to accomplish a high-level goal (Georgas and Taylor 2008; Edwards et al. 2009), and while these are important aspects of any successful system, introspection mechanisms also provide the opportunity for systems to self-adjust to achieve better task performance. A few recent approaches have utilized self-adjusting mechanisms for performance improvement (e.g., Perez-Palacin and Merseguer; **?** (2011; **?**)), but the application to cognitive systems remains largely unexplored.

Self-adjusting mechanisms are either implemented by means of a single system-wide mechanism and operate only at the infrastructure level, or are realized as highly special-

ized mechanisms for specific structural-level components. While both types of introspection are integral to any successful architecture, we claim that such mechanisms need to be simultaneously integrated at *all* levels of an agent system in a granular and distributed fashion, so that synergies between the introspection mechanisms and their ways of making adjustments at the different levels can be utilized to produce more robust systems than any of these mechanisms alone would be able to produce.

A Three-Level Introspection Framework

Architectures of cognitive agents consist of various, typically heterogeneous components and the way they are connected (e.g., planners, action execution, vision processing, etc.). Similarly, architectures of agent infrastructures (sometimes also called MAS middleware) consist of various heterogeneous components for managing control and information flow in distributed systems (e.g., white and yellow page services, discovery agents, etc.). Both kinds of architectures have components that are computational processes that perform operations specific to their functional roles in the agent system: in cognitive architectures, components might perform analyses of images or sounds, parsing of natural language expressions, or truth maintenance in a knowledge base; in infrastructures, specialized components might perform service discovery, communication and load balancing functions, while others implement functionality specific to the agent architecture implemented in the middleware. The crucial idea of multi-level introspection is to integrate component-level, infra-structure-level, and agent-level introspection. The first is performed inside every computational component, the second is performed via specialized components, and third is performed inside some specialized components (e.g., a reasoner).

Moreover, introspection at all three levels follows the same three phases necessary to adjust components, their arrangements, and agent goals: *self-observation, self-analysis, and self-adjustment* (Anderson and Perlis 2005). The self-observation phase is responsible for taking note of any data that will be useful for analysing the state of operation. Self-analysis uses the observed data along with a set of domain-specific expectations to make hypotheses about the current state of things (e.g., an expensive process is no longer needed), and the self-adjustment phase makes adjustments based on the findings of the self-analysis (e.g., suspend that process until it is needed again). Based on the introspection level, different adjustment operations are appropriate: at the component level, component-internal algorithms are selected to optimize component operation, while at the infrastructure level system configurations are adjusted to better utilize computational resources; finally, at the agent-level, goals and subgoals together with their priorities are adjusted to improved the agents' task performance.

Component-Level Introspection

Individual components in an agent architecture perform highly-specialized tasks that may operate independently or in tandem with one or more other components (e.g., a sentence parser component in a cognitive architecture might rely on the output of a speech recognition component, or a service discovery component in the infrastructure might rely on the yellow page component).

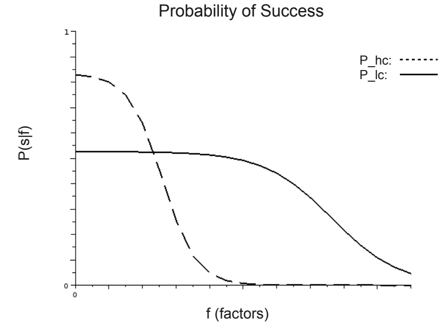

Figure 1: Example relationship between a high-cost algorithm P_{hc}, and a low-cost algorithm P_{lc}.

More formally, a component consists of a set of algorithms, $A = \{a_0, a_1, ..., a_n\}$, where each a_i is capable of independently completing a given task. The probability of a successful outcome s for a particular algorithm a_i can be expressed as $P(s|f)$, where f is an array of contributing factors, and can include such things as robot velocity, room noise, sensor quality, and/or light levels. Depending on the complexity of a particular a_i, $P(s|f)$ could be derived analytically, found experimentally (e.g., by determining false-positive and false-negative rates under various f), or more abstractly by monitoring various properties of each algorithm (e.g., running time, loop time, number of iterations, time of last iteration, time of last client use). An example relationship between a high-cost and low-cost algorithm can be seen in Figure 1. From here it is easy to see that:

$$P_{hc}(s|f) > P_{lc}(s|f) \text{ for } \{0 \leq f < cf\}, \text{and}$$

$$P_{hc}(s|f) < P_{lc}(s|f) \text{ for } \{f > cf\}$$

where cf is some critical threshold at which $P_{hc}(s|f) = P_{lc}(s|f)$.

Furthermore, it is often the case that each algorithm within a set A offers important tradeoffs between cost, speed, and effectiveness. A low-cost speech recognizer, for instance, might perform well in low-noise environments, but become ineffective in noisy rooms, while an alternate recognizer might provide adequate results but at the price of high resource use. Thus, the general goal of an introspection policy at the component-level will be to maximize $P(s|f)$ while also trying to minimize cost $C(a_i)$. Applying this policy π to accomplish a particular component task, π is evaluated against the current state of the component, including the currently selected algorithm, a_i, to determine the appropriate course of action, which can be a change of algorithm selection and/or a goal request to a higher level component.

Even though introspection at the component level is often highly specialized and tailored to the specific needs and functions of each component, we have defined a general

high-level policy that can be implemented in any component to improve system performance.

Infrastructure-Level Introspection

Complex agent architectures, especially for robots, are often implemented in an infrastructure or middleware (e.g., Scheutz et al. (2007)), which itself has an architecture, not unlike the agent architecture, with knowledge about the underlying operating environment such as available hardware, communication between components, distribution of components across multiple hosts, and resource management.

Infrastructure-level reflection is responsible for system state, and while it is often the case that infrastructure-only reflection can effectively monitor and respond to these concerns, communication from other levels of reflection can allow substantial performance and utility improvements at this level. For example, a component may be able to detect that it is performing poorly but have no way to address the problem, while the infrastructure has no way of telling how any given component is performing even though it is capable of restarting or replacing the component. With inter-level communication, introspection at the component-level can provide the needed information to the infrastructure.

Agent-Level Introspection

Cognitive agents typically have knowledge about the world, their own capabilities, and their own goals (e.g., including which goals to achieve, the time-frame in which each goal needs to be achieved, and the relative importance of concurrent goals). This knowledge is used by various deliberative mechanisms (planners, reasoners, etc.) to make decisions and generate actions that lead to the accomplishment of the agent's goals. *Agent-level introspection* is then largely responsible for observing the progress of plans, detecting when plans are acceptable, when they need to be regenerated, or when certain goals are unattainable and need to be abandoned altogether. Note that agent-level introspection is thus not concerned with the overall system state (e.g., whether a certain component is operating correctly, or whether there are adequate system resources).

Information from introspection mechanisms at the other levels can be crucial for timely and successful adjustment at the agent level. For example, assume the goal of a robot is to (1) navigate from way-point A to way-point B and (2) take a photograph of the location. If the camera component is inoperable, agent-only introspection would only detect a failure after it had successfully reached way-point B and attempted to use the camera. If, on the other hand, introspection mechanisms responsible for monitoring the camera were able to report to the agent-level mechanism, the failure could be detected both before the robot set out toward its destination, and during task execution (i.e., while traveling to way-point B, but before attempting to use the camera). The robot may be able to recover in either scenario, but it is easy to see how communication between introspection levels could mean the difference between success and failure (e.g., when additional time constraints are imposed).

Validation Experiments

We have implemented the introspection framework in the robotic DIARC (distributed integrated affect, reflection, cognition) architecture (Scheutz et al. 2007) to be able to demonstrate the utility and effectiveness of the multi-level introspection. Figure 2 shows a sample DIARC implementation, including the levels of introspection that are most relevant to each component. We have chosen a simple task and scenario in which each of the three levels of introspection (agent, infrastructure, and component) engages in the three phases of introspection described above (self-observation, self-analysis, and self-adjustment). In this scenario (schematically depicted in Figure 3), a robot is instructed to move down a corridor (starting at position A), to find the "Research Lab" (D3). The location of the target room is unknown, so the robot must visually identify signs affixed to doors; these signs are all the same shade of blue with the room name in black letters.

Agent-level introspection in DIARC is performed by the DIARC goal manager, which selects the best available action to achieve the agent's goals in its current state (Schermerhorn and Scheutz 2010). The goal AT(ROBOT,RESEARCH LAB) and the actions working toward it are monitored to ensure progress, and new actions are selected based on agent and world states. When other goals are instantiated, the goal manager resolves resource conflicts in favor of the higher-priority goal.

Infrastructure-level introspection in the employed ADE infrastructure (Scheutz 2006) is handled mainly by a dedicated infrastructure component, the ADE system registry. The registry monitors the system, including the resource utilization on the computers hosting the components. If the load becomes unbalanced, the registry can improve overall performance by migrating a component from a high-load host to an idle host—assuming that no hardware constraint (e.g., a connected camera) requires the component to run on a specific host.

Since aspects of the agent- and infrastructure-level introspection mechanisms have already been described elsewhere, as indicated above, we focus here on component-level introspection in a particular DIARC component: the DIARC vision component. The vision component features multiple detectors for a variety of object types, and uses introspection to drive a policy for choosing between them. In particular, for the current task, the set of algorithms A includes two detectors capable of detecting the target sign: a_1, a blob detector that scans captured frames for regions matching a specified RGB color range and size, and a_2, a SIFT-based detector that compares SIFT features extracted from frames with a database of features for the target sign. a_2 is more accurate than a_1 with respect to successfully detecting blue "Research Lab" signs in a camera frame (i.e., $P_{a_1}(s) < P_{a_2}(s)$), but is also more costly in terms of reduced frame rates and high CPU and GPU utilization ($C(a_1) < C(a_2)$).

Policy Options

A simple static policy would indicate which of the two detectors to instantiate, either to maximize $P(s)$ or to min-

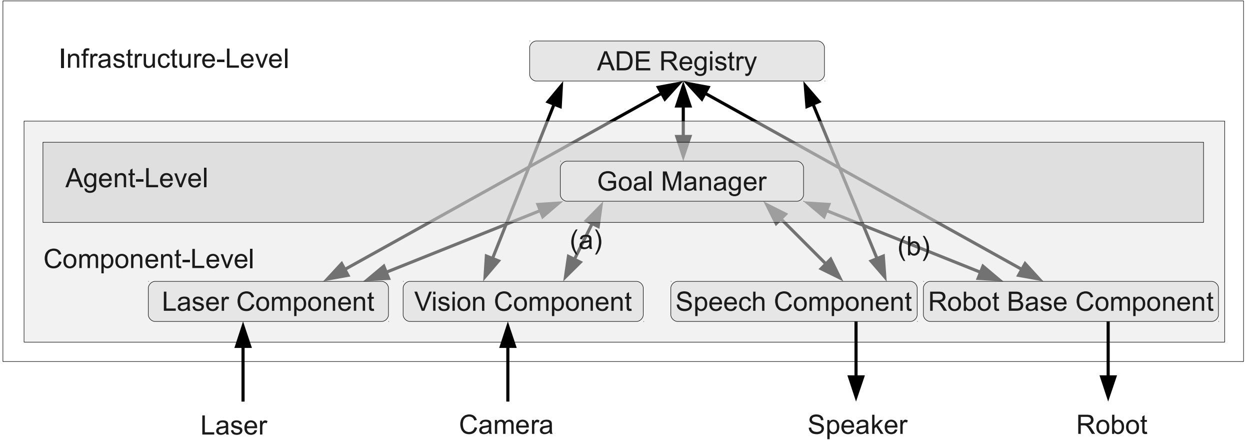

Figure 2: An example DIARC configuration used in the validation experiments. (a) represents the connection used by the Vision Component to submit the "pause motion" goal, and (b) is used to carry out the goal *if* it is accepted by the Goal Manager.

imize $C(a)$. A somewhat more sophisticated policy would include additional information about f, the known set of factors that contribute to each detectors probability of success. The most important such factor in the validation scenario is v, the velocity at which the robot is moving down the corridor, because it affects the likelihood that enough of a sign is captured in a frame for a_2, the SIFT detector, to identify it given its reduced frame rate. If the designer knows the velocity at which the robot will travel, the conditional probabilities $P_{a_1}(s|v)$ and $P_{a_2}(s|v)$ can be used to determine which detector to use.

Component-level introspection in the vision component can improve performance over such static selections by using run-time information to select the best detector given the circumstances. The velocity may not be known in advance, for example, in which case the vision component could monitor it and change detectors dynamically according to a policy π. In the more complex example presented here, selecting a detector in advance is impossible, because neither detector has both acceptable accuracy and acceptable cost; a_1 is too susceptible to false positive detections, while the impact of a_2 on system load is to high to allow other components to operate effectively. Component-level introspection allows the vision component to monitor operating conditions and switch between the algorithms to approximate a single detector with accuracy on a par with a_2, but at an acceptable level of cost. This works because the inaccuracy of the color blob detector a_1 with respect to "Research Lab" signs is constrained to false positives. It is, in fact, very good at detecting "Research Lab" signs, but not at distinguishing them from other signs in the environment. In contrast, the high-accuracy detector a_2 will have very few false positives, in addition to very few false negatives (assuming v is such that the signs are likely to be fully captured in a frame processed by a_2). If a "Research Lab" sign is present, both a_1 and a_2 are very likely to detect it, and if a_2 detects something, it is very likely to be a "Research Lab" sign.

Because a_1 and a_2 meet these constraints, the policy can take advantage of the cost/accuracy trade-off by running the detector a_1 under normal circumstances and switching to to a_2 when a_1 detects something; a_2 is used, in effect, only to verify the judgement of a_1. The policy π is, thus, defined in terms of simple "if-then-else" rules based on the outcomes of running detectors: if a_1 detects a target, a goal is instantiated that generates component-local actions (e.g., reconfiguring the set of running detectors) and possibly messages to other components in the system (e.g., submitting a goal to the goal manager, in this case to pause the motion to allow a_2 a chance to examine the scene). If a_2 subsequently fails to detect the target sign, π prescribes a switch back to the low-cost detector to free up resources for other components. Hence, π can be viewed as a mapping from detector-value pairs onto the Cartesian product of the set of local actions and the set of external actions available to the component.

Policy Evaluations

Three configurations were compared: static blob-only detection (C1), static SIFT-only detection (C2), and dynamic detector switching using introspection (C3). The characteristic trajectory for each configuration is shown in Figure 3 (top). There are three doors with signs (D1, D2, and D3) in this section of the hallway; D3 is the goal ("Research Lab"). The CPU utilization for each is given in Figure 3 (bottom).

In (C1), the blob detector a_1 is activated at the start of the evaluation run (A in Figure 3 (top)). CPU utilization is low throughout the run, but because there are multiple signs of the same color present, (C1) fails to locate (D3), finishing much earlier than the others due to a false-positive identification of the target sign at (D1)—the goal AT(ROBOT,RESEARCH LAB) appears to have been achieved.

In (C2), the more expensive SIFT-detection a_2 is acti-

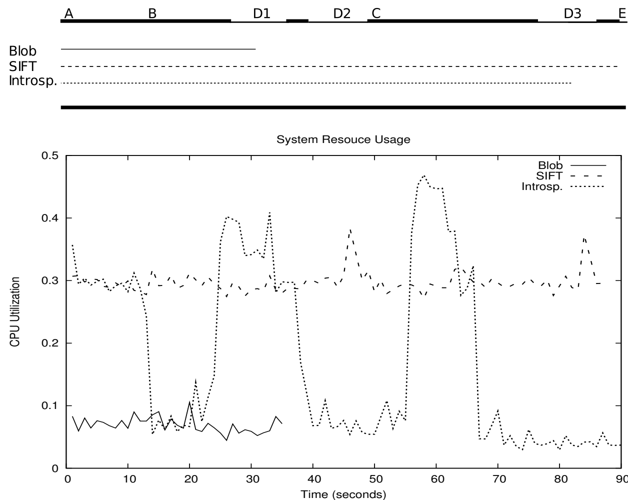

Figure 3: Schematic of evaluation runs (top) and CPU utilization (bottom) for (C1) static blob, (C2) static SIFT, and (C3) dynamic introspection configurations. The robot starts in the center of the hallway in all conditions; the three trajectories are offset here for display purposes.

vated at the start (A), leading to consistently higher CPU utilization. This configuration is not fooled into stopping at (D1) or (D2), but the run ends in failure at (E) after the robot fails to identify (D3) and drives past. Because the sign is in the frame of the side-mounted camera for a relatively short amount of time, the slower SIFT method fails to achieve a tracking lock (note that, although the failure could be avoided by traveling at a slower velocity, the CPU utilization would be the same and the overall duration of task would be longer).

The analysis policy π for (C3) is designed with two competing goals in mind: providing reliable results to client ADE components, and minimizing resource consumption. π consists of four rules:

R0 enable a_2 when an external request to identify target signs is received

R1 when a_2 is active for 10 seconds without detecting the target sign, switch to a_1

R2 when a_1 indicates a possible match, switch to a_2 and submit a "pause motion" goal to the goal manager

R3 when no external request for target signs has been received for 10 seconds, deactivate sign detection

The robot begins at (A) with neither sign-detector active, but immediately begins receiving periodic requests (from actions being administered by the goal manager) to look for signs and, in accordance with (R0), instantiates a_2. Via self-observation, the component determines that no sign is detected by the time 10 seconds have passed and (B) is reached, so it determines that an adjustment is desirable and, by (R1), switches to a_1; CPU utilization drops from (C2) levels to (C1) levels.

At (D1), the component observes a positive result from a_1. The adjustment dictated by (R2) switches to detector a_2, causing an increase in CPU utilization, somewhat higher than (C2) levels for the first few seconds while the detection and tracking threads are initialized.[1] Because the changeover is not instantaneous, the initialization routine uses cross-component communication to submit a "pause motion" goal to the goal manager.[2] No match is detected at (D1) or (as the robot moves on) at (D2), and after the component observes the 10 second period without a positive detection event, adjusts according to (R1), causing a_2 to be replaced again by a_1 at (C) and CPU utilization to drop.

At (D3) the sign is matched by a_1, and (R2) the detector exchange process is triggered again (along with the resulting increase in CPU utilization). This time, the match is verified and the result is returned to the goal manager in response to the most recent request. The robot has arrived at its destination, so it turns toward the door and announces its arrival. No further requests to look for doors are sent by the goal manager, and once the time threshold has been reached, (R3) is triggered and the sign-detection subsystem is deactivated entirely.

The CPU results in Figure 3 clearly demonstrate the benefits of introspection when the blob-based detector is active. Moreover, despite the switching overhead, the average CPU utilization across the run for the introspection version (M=0.177, sd=0.144) was much lower than for the SIFT-only version (M=0.297, sd=0.017), although not as low as the blob-only configuration (M=0.069, sd=0.013). The advantages of introspection would become even more apparent in terms of *shorter overall runtime* and *lower overall CPU utilization* during the course of longer tasks. A video is available at `http://tiny.cc/introspect`, showing each of the three conditions in action. Note that, in order to highlight the various transitions described above, verbal indications have been temporarily inserted when the policy dictates a change.

Discussion

The validation results establish the potential benefits of multi-level introspection in agent systems. Due to the three-level design, policies at these levels can be effective (even if not optimal) across a wide range of scenarios. This is because the effectiveness of a particular policy is a measure of how much it contributes to achieving the goals of the entity

[1]This overhead became apparent during the course of these evaluations; we have since implemented a suspend/resume mechanism to replace the start/stop process that eliminates the restart penalty.

[2]There is no guarantee that this goal will be accepted; its priority is based on the value of the goal that triggered it (in this case the AT(ROBOT, RESEARCH LAB) goal), while its urgency is typically higher, given the short time frame within which it must be performed. Hence, it will typically be granted priority over the triggering goal, but another goal may have higher priority and require the motor resource. In that case, the pause goal will fail and an accurate reading will not be made—the same as in the other conditions.

for which it is designed. So, for example, infrastructure-level policies are dependent on the goals of the infrastructure (to facilitate communication, ensure reliability, etc.). Similarly, the vision component's goals have to do with its local state (e.g., maximizing frame rates, response times, etc.). The important thing to notice is that, because these levels have (at best) limited access to the *agent-level* goals, their goals are defined with reference only to their own infrastructure- or component-internal states. For example, locating the research lab is not a goal of the vision component, even though its policies, to the extent that they are effective, should contribute to that goal. Hence, well-formed policies at these levels will be applicable and effective across a wide variety of agent-level goals and scenarios.

While the distributed nature of our approach allows for largely self-sufficient and modular introspection mechanisms, an important aspect is also the ability of these mechanisms to reach across vertical and horizontal boundaries when necessary. Cross-level communication in the system can take many forms, but is achieved via a limited set of mechanisms. The goal manager responsible for agent-level goals is a DIARC component, so other components can submit requests just as they could to any other component, as illustrated in the validation example when the vision component sent the "pause motion" request. Although it would definitely be possible for the vision component to have established a connection with the robot base component and sent the request there directly, that would require somewhat more detailed knowledge of the interfaces implemented by the relevant component, not to mention knowledge of which component is the relevant one. Instead, the vision component sent the request to the goal manager. Whether the robot was a wheeled base (as in this case), a UAV, or a biped robot, whether it was moving or already stationary, whether a simple motion command was sufficient to stop the robot or a goal needed to be cancelled in a motion-planning component, all were unknown and irrelevant to the vision component. The vision component neither had nor needed to know about any of the other components in the system. The goal manager maintains detailed knowledge of most of the components in the system so that other components do not have to. Because of this, the vision component needed to know about only the `submit goal` interface in the goal manager. Moreover, because this is actually cross-level communication between the component and the agent levels, the agent level is given the opportunity to evaluate how the requested action will affect agent-level goals and determine whether identifying the door is of greater importance than continuing down the corridor.

Ideally, it will be possible for policies to be constructed automatically, with little or no human intervention (e.g., specifying possible performance measures), and this is currently one limitation of our system. This will likely entail extensive data collection, evaluating performance under as many relevant condition variations and usage patterns as possible (e.g., lighting conditions, for a vision component) and possibly some kind of learning procedure to categorize different actions according to the performance measures (e.g., high accuracy, low resource use) and determine how various conditions (e.g., how frequently a particular request is received) contribute to the success or failure of a particular policy. In that case, the programmer could simply specify the goals and their relative priorities, and the system could generate a policy that would attempt to achieve those goals. Moreover, because cross-level communication is possible, this would also open the door for run-time "tweaking" of other components' policy parameters, effectively enabling dynamically switchable policies based on the needs and goals of higher level architecture components.

Conclusion and Future Work

We argued for the utility of employing multi-level introspection mechanisms in agent systems to automatically improve agent performance and presented a three-level introspection framework. Comprising agent, infrastructure, and component levels, the framework acknowledges the distinct goals, information, and responses available at these different levels, in addition to the need for mechanisms that allow for information sharing with or requesting aid from other levels and components of an architecture.

We described a concrete instantiation of the framework in the DIARC robotic architecture and presented examples demonstrating the utility of introspection on a physical robot. Specifically, we showed that component-level introspection can be integrated both independently of, and cohesively with, higher-level mechanisms to improve system performance during a simple robotic task by lowering system resource use and overall time to completion.

Future work will investigate how and the extent to which policies at the different levels can be learned automatically during task performance.

Acknowledgements

This work was in part funded by ONR MURI grant #N00014-07-1-1049.

References

Anderson, M. L., and Perlis, D. R. 2005. Logic, self-awareness and self-improvement: The metacognitive loop and the problem of brittleness. *Journal of Logic and Computation* 15(1):21–40.

Edwards, G.; Garcia, J.; Tajalli, H.; Popescu, D.; Medvidovic, N.; Gaurav, S.; and Petrus, B. 2009. Architecture-driven self-adaptation and self-management in robotics systems. In *Proceedings of the 2009 ICSE Workshop on Software Engineering for Adaptive and Self-Managing Systems*, 142–151. Washington, DC, USA: IEEE Computer Society.

Georgas, J. C., and Taylor, R. N. 2008. Policy-based self-adaptive architectures: a feasibility study in the robotics domain. In *Proceedings of the 2008 international workshop on Software engineering for adaptive and self-managing systems*, SEAMS '08, 105–112. New York, NY, USA: ACM.

Haidarian, H.; Dinalankara, W.; Fults, S.; Wilson, S.; Perlis, D.; Schmill, M.; Oates, T.; Josyula, D.; and Anderson, M. 2010. The metacognitive loop: An architecture for building robust intelligent systems. In *PAAAI Fall Symposium on Commonsense Knowledge (AAAI/CSK'10)*.

Morris, A. C. 2007. *Robotic Introspection for Exploration and Mapping of Subterranean Environments*. Ph.D. Dissertation, Robotics Institute, Carnegie Mellon University, Pittsburgh, PA.

Perez-Palacin, D., and Merseguer, J. 2011. Performance sensitive self-adaptive service-oriented software using hidden markov models. In *Proceedings of the second joint WOSP/SIPEW international conference on Performance engineering*, ICPE '11, 201–206. New York, NY, USA: ACM.

Schermerhorn, P., and Scheutz, M. 2010. Using logic to handle conflicts between system, component, and infrastructure goals in complex robotic architectures. In *Proceedings of the 2010 International Conference on Robotics and Automation*.

Scheutz, M.; Schermerhorn, P.; Kramer, J.; and Anderson, D. 2007. First steps toward natural human-like HRI. *Autonomous Robots* 22(4):411–423.

Scheutz, M. 2006. ADE - steps towards a distributed development and runtime environment for complex robotic agent architectures. *Applied Artificial Intelligence* 20(4-5):275–304.

Sykes, D.; Heaven, W.; Magee, J.; and Kramer, J. 2008. From goals to components: a combined approach to self-management. In *Proceedings of the 2008 international workshop on Software engineering for adaptive and self-managing systems*, SEAMS '08, 1–8. New York, NY, USA: ACM.