"Talk to me!": Enabling Communication between Robotic Architectures and their Implementing Infrastructures

James Kramer
Department of Computer Science and Engineering
University of Notre Dame
Notre Dame, IN 46556, USA
jkramer3@cse.nd.edu

Matthias Scheutz and Paul Schermerhorn
Cognitive Science
Indiana University
Bloomington, IN 47401, USA
{mscheutz,pscherme}@indiana.edu

Abstract— **Complex, autonomous robots integrate a large set of sometimes very diverse algorithms across at least three levels of system organization: the *agent architecture*, the *implementation environment*, and the *hardware devices*. Insofar as a distinction is maintained between them, the levels serve different purposes and thus exhibit different characteristic strengths and weaknesses. Exchanging information among organizational levels can be used to mitigate the shortcomings of one level by making use of the strengths of another.**

In this paper, we highlight the roles, characteristics, and relations between the *infrastructure* and the *architecture* of complex robots, describing a novel form of integration that results from enabling the exchange of information between these two levels, which otherwise is maintained internally. The information from the infrastructure is especially amenable for use by the architecture to achieve a higher level of robustness and system awareness. We demonstrate the functionality and utility of the proposed mechanisms in a set of experiments in which failures of architectural components are induced on an actual robot engaged in a joint human-robot team task.

I. Introduction

Complex autonomous robots, especially those interacting naturally with humans, rely on a large number of different functional modules and capabilities that need to operate concurrently in a coordinated fashion. There are three levels at which coordination is required: (1) the *agent architecture*, where the information and control flow among components, as well as each component's functionality, is specified; (2) the *infrastructure*, which provides the implementation environment (often a *virtual machine*) for an architecture; and (3) the hardware environment, on which the virtual machine runs, providing both computing capabilities and access to the various devices that operate the robot's sensors and effectors.

Coordination among components at each of the levels is targeted at different purposes: the architecture aims at making an agent perform a set of specified tasks as well as possible; the infrastructure provides the computational resources required by the architectural components' algorithms while interacting with the hardware devices; and the hardware devices provide the sensor information and perform the effector operations required by the architecture. As a result, both the information available to each level and how it is used to adjust to real-world events are also different. For example, an architecture might have knowledge about how to cope with failures in vision processing due to bad lighting, while the infrastructure might have mechanisms that monitor the CPU load to guarantee the vision component will get enough CPU time to function properly. The hardware level, finally, might have mechanisms to recover from communication errors between the framegrabber and the main computer.

In many cases, the information generated, processed, and stored at a level is relevant only to that level. For example, the choice of one particular CPU among many will likely not matter to the architecture, so long as the required computation completes. Similarly, the type of knowledge structure used by the architecture does not concern the infrastructure, so long as adequate memory is available. However, there are cases where information is relevant across organizational levels. For instance, the architecture might switch to sonar readings if informed by the infrastructure that the laser rangefinder has failed. Similarly, the infrastructure might reduce system load by shutting down non-essential components. Communication between levels in such cases can be be highly beneficial, if not *necessary*, for task success.

In this paper, we explicitly distinguish between a robotic architecture and its implementing infrastructure, resulting in a novel view of information exchange between them that grants a higher level of system awareness to the architecture that can lead to significantly improved handling of resource allocation, component failures, etc. The rest of the paper is organized as follows: we start with a summary of the architecture and infrastructure levels, isolating their salient properties from a systems perspective. We then discuss the sharing of information across levels that would positively impact both performance and robustness. The subsequent experimental evaluation focuses on failure detection and recovery, showing the benefits of information exchange between the infrastructure and the architecture.

II. The Distinction between Architectures and Infrastructures

As presented here, an infrastructure is *not* part of an architecture, but rather provides functionality and tools that aid implementation. This separation permits different agent architectures–even cognitive architectures [1]–to use the same framework. We first describe the differing purposes of infrastructure and architecture, then list some characteristics of each, and finish with a brief comparison to related work.

A. Distinct Purposes

An *agent architecture* is the blueprint of a system's functional organization, specifying its functional break-down, component operations, and the interactions among components. Traditionally, architectures for complex deliberative agents, particularly cognitive architectures, have not been designed with physical implementations (e.g., on a robot) in mind (e.g., [2], [3], but see [4]). Consequently, they do not provide mechanisms for distributed, fault-tolerant computing under real-time constraints with limited resources. Rather, the implicit assumption is that computational resources are always sufficiently available and that the speed of execution is secondary (oftentimes, architectural updates occur in "logical-time", entirely decoupled from real-time).

While it has always been necessary in robotics to be sensitive to the computing environment's limitations, only recently have there been efforts to provide a computing infrastructure that is decoupled from and independent of the actual architecture.[1] Such infrastructures provide a "run-time" implementation environment for an architecture–that is, a *framework* for connecting and executing architectural components–that handles the real-time, real-world constraints of the robotic hardware. In such frameworks, e.g., [6], [7], [8], [9], particular attention is given to appropriate hardware abstractions, sensory processing routines, simulators, networking protocols, and various other tools that facilitate the design and implementation of robotic architectures.

B. Distinct Characteristics

Considerations for distinguishing between an infrastructure and an architecture include: *hardware knowledge* (i.e., the types of sensors, effectors, platforms, operating systems, etc., supported by the system, in addition to their abstractions and programming interface), the *operational time frame* (e.g., obstacle avoidance must be immediately responsive, but planning a path to a goal location can be delayed), *update frequency* (e.g., sensor readings are taken periodically, while planning is sporadic), and the degree of *task-specific knowledge* (e.g., direct motor control versus spatial relations involved in navigation). While these issues are a concern at each level to some degree, their expressions (generally) take on divergent characteristics, as shown in Table I.

TABLE I
INFRASTRUCTURE AND ARCHITECTURE CHARACTERISTICS

Property	Infrastructure	Architecture
Operation principles	low-level	high-level
Task knowledge	none	detailed
Hardware knowledge	detailed	minimal
Timing	immediate	delayed
Control frequency	periodic	sporadic

[1]In Brook's subsumption architecture [5]–the first "behavior-based" architecture–the infrastructure assumptions about asynchronously executing components with unreliable communication links were effectively part of the "architectural design."

First, consider agent architectures: they often utilize abstract representations of the world (i.e., entities and their properties), have detailed knowledge of high-level tasks (i.e., goals and how to achieve them), are able to formulate plans (i.e., make use of pre- and post-conditions), learn from experience, and may incorporate sophisticated techniques for choosing one task over another. Planning to achieve goals generally encompasses a relatively long time frame, perhaps on the order of seconds or minutes for a single, concrete step in pursuit of a goal (e.g., "move to a location" or "ask for more information"), but potentially ranging from minutes to hours for a high-level and possibly open-ended task (e.g., "give a tour" or "monitor the hallway"). As a result, action control is sporadic, occurring on an as-needed basis.

On the other hand, an infrastructure necessarily has low-level access to the execution environment, with detailed knowledge of the available hardware, communication channels between components, and possible distribution over multiple CPUs. It may also be responsible for coordinating system operation, often including management of parallel and asynchronous process execution, monitoring of hardware and software components, and detecting and recovering from errors. Maintaining system operation is a continuous process requiring immediate reaction to changing conditions, often on the scale of fractions of a second.

C. Architecture Versus Infrastructure

The divergent characteristics given in the previous section result from decoupling the infrastructure from an agent architecture. Doing so is not *necessary*; an architecture may entirely address any or all of them. For instance, a single, monolithic system would encompass all of them and achieve the highest possible degree of integration.

The primary motivations for separating the infrastructure from an architecture are to make robot programming faster, easier, and more efficient. For instance, Vaughan *et al* [10] describes the use of "device abstractions" to increase portability and reusability. An infrastructure not only reduces complexity, but also removes the programming effort otherwise required to implement all the processes and activities provided to an architecture. Furthermore, an infrastructure eliminates, to the extent possible, an architecture's need for "hardware-awareness." That is, a concern at the architecture level is that components are *able* to communicate, not *how* they do so; resources must be allocated, but their *locations* are likely unimportant; components must provide data in a timely fashion, but their functional roles do not dictate the specific details of internal operation.

Many existing architectures include infrastructural functionality in their descriptions. For instance, the typical three-layer architecture found in [11] includes defining characteristics of the "controller" layer: algorithmic time and space bounds, bandwidth, failure detection, and internal state. More recently, various *model-based* architectures ([12], [13]) have been proposed that construct a monolithic system representation to mediate between reactive and deliberative layers, monitoring system operation, allocating resources,

and validating the exchanged commands/data. The impact of accounting for these items can be greatly reduced (similar to using "device abstractions") by moving implementation details from the architecture into an infrastructure.

III. Benefits of Integration

Once the benefit of using an infrastructure to implement architectures is accepted, a natural reaction might be to move to the opposite extreme, from monolithic control systems to strictly separate infrastructure and architecture. Yet, the monolithic approach does have its own advantages, primary among which is the availability of information that may be masked by strict separation. The ideal approach will strike a balance between the two extremes, taking advantage of the strengths of each. This gives rise to the questions: what are the potential types of exchange between infrastructure and architecture? How can they be integrated? More specifically, in what areas can their different characteristics be utilized?

A. Two Examples

One area that can benefit from architecture/infrastructure integration is *system configuration*. The architecture has specific information about the goals it is attempting to achieve, including what low-level capabilities are required, the time-frame in which results are required, and the importance of particular tasks, potentially with alternatives. The infrastructure has information about the resources required to fulfill those needs, including hardware availability, host configuration, and status of components.

System configuration can be viewed as the matching of capabilities required for a task (or set of tasks) with those available. A particular architecture (e.g., a robot receptionist) might need speech recognition and production, visual processing, and affect recognition and expression capabilities, while another (e.g., for autonomous exploration) might need mobility, localization, and other sensors. The same infrastructure can be used for both, where the architecture requests a particular configuration. If the request cannot be satisfied, the infrastructure can relay *why* (i.e., what part of the request cannot be met), allowing an alternate choice.

An ability to change system configurations allows *resource optimization*. The architecture has information about task requirements, their order, and potentially their duration; in short, a specific plan over some length of time. In addition to its knowledge of hardware devices, the infrastructure can determine their connectivity, response time, and status.

At a coarse-grained level, exchanging information permits components to be started or stopped as needed. The addition of a rudimentary scheduler can allow the infrastructure to autonomously configure the system in a dynamic way, ensuring that components are available for the architecture when needed. Furthermore, if the infrastructure is able to maintain information about system resources (e.g., CPU load, available memory, or battery level), finer-grained control is possible, dynamically changing the configuration to best manage resources to still meet the requirements. Finally, the infrastructure may be able to determine whether any configuration exists that can meet the real-time deadlines.

B. Failure Recovery

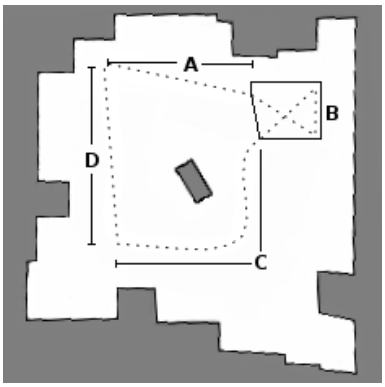
Many scenarios require autonomous robots operate for extended periods of time. The need for fault-tolerance in robotic applications is demonstrated by a simple observation: *given a long enough time-frame, any system will inevitably fail*. Failures at the infrastructure and architecture levels exhibit different characteristics that result in different types of recovery, if recovery is possible.

Some example failures that may occur at the infrastructure level include: hardware breakage (e.g., sensor/effector malfunction, faulty wiring, unseated card), operating system halts, software component crashes (e.g., uncaught exception, illegal state), unresponsive software components (e.g., due to an infinite loop or deadlock), and network failure (e.g., broken sockets, out-of-range wireless). The characteristics of these failures fit those listed for infrastructures in Table I: to consistently detect arbitrary failures, monitoring must be continuously active and span the entire lifetime of the application; detection is possible shortly after the failure; an abrupt and immediate disruption in system operation occurs.

Within an infrastructure, failure recovery can be handled in two ways: propagation, with the expectation that it will be resolved elsewhere, or direct intervention, either taking steps to reconnect components that are still operational, substituting redundant or replicated components for those that failed, or restarting the failed components, potentially requiring migration. While specific mechanisms (e.g., message-passing, thrown exceptions, heartbeats, etc.) may yield slightly better or worse performance relative to one another, infrastructural recovery will retain its fundamental nature; that is, recovery is limited: to the extent the infrastructure's knowledge is limited to the execution environment (i.e., the computational infrastructure and the robotic hardware), it is general-purpose and uninformed. Even if failed components can be recovered, it might not be enough to bring the system back into an operational state.

Failures at the architecture level are of a different nature; some examples include: user input error, pursuit of an unachievable goal, and interruption due to cascading infrastructure-level failures. These types of failures match the architecture characteristics found in Table I: failures tend to be task-specific; an individual task usually spans an extended time-frame (especially relative to lower level failures); as a consequence of the time-frame, failures are event-driven and thus sporadic.

Architecture-only failure recovery amounts to task reformulation, either by meeting the goals in a different way (e.g., substituting components that provide similar data) or by aborting the current task in lieu of one that can be achieved. Again, however, recovery is limited: the high-level and task-oriented knowledge involved likely avoids the general mechanisms necessary to identify and resolve low-level problems. With architecture-only recovery, the process is oriented at achieving the specific goal being pursued, not



	Recovery Type	Mode at Time of Failure	Component Recovered?	Completion State
1.	Infra.	Autonomous	×	Failure
2.	Infra.	Autonomous	√	Success
3.	Infra.	Interactive	×	Failure
4.	Infra.	Interactive	√	Failure
5.	Arch.	Autonomous	× / √	Abort
6.	Arch.	Interactive	× / √	Abort
7.	Both	Autonomous	×	Abort
8.	Both	Autonomous	√	Success
9.	Both	Interactive	×	Abort
10.	Both	Interactive	√	Success

Fig. 1. The "Explore, transmit, return to base" task. **Left:** A map of the area, showing the task phases and one possible trajectory. **Right:** Experimental conditions and resultant task completion state.

at the overall system state; failure is detected only when the current task is interrupted; recovery is initiated on an as-needed basis, potentially continuing an unachievable task, identified only when an unavailable dependency is finally encountered. The architecture may be able to adapt its task in some way (e.g., backtrack to a point where the goal is once again achievable) or it may have to simply abort; although the latter is undesirable, the former may lead to unacceptable penalties in terms of time and resource consumption.

A system with both types of recovery can use the strengths of each to mitigate the other's shortcomings: infrastructural recovery maintains general system health, while architectural recovery resolves task-specific issues; infrastructural recovery continuously monitors for failure, while architectural recovery resolves issues as required; infrastructural recovery attempts to resolve any and all failures in an uninformed way, while architectural recovery alters plan or goal achievement to adjust to specific failures.

A further improvement can be had over simply enabling recovery at both levels by allowing *communication* of recovery status between levels. Communication may be "lazy"; for instance, the architecture, only when attempting to use a component and subsequently detecting its failure, might query the infrastructure for the status of the component's recovery process. The infrastructure might respond that the component is *unrecoverable*, in which case the architecture would proceed with its goal modification. Alternately, the infrastructure might respond that recovery has been initiated but is *not yet complete*, allowing the architecture to *delay* its goal pursuit until low-level recovery finishes. However, exchanges can also be *proactive*; a level may be *notified* of recovery status as it changes, in effect giving a *warning* that a failure condition *may* be impending.

An important point to recognize is that the information's utility is contingent upon a level's properties: because the infrastructure's recovery mechanisms are always active and occur in a short time-frame, in most cases no benefit is gained by architecture notification. But the converse is not true; depending on the goal, it may be the case that the architecture would not detect the failure until some point *far in the future*. Proactive failure notification by the infrastructure allows the architecture to introspectively examine its current task for dependencies and alter its current plan.

IV. Experimental Validation: Integrating ADE and DIARC

For evaluation purposes, a series of experiments were conducted that demonstrate various combinations of failure recovery types. A primary experimental condition was to use a task that had already been tested, inducing failures *during task execution*. Specifically, we augmented the DIARC (Distributed, Integrated Affect Reflection Cognition) functional architecture [14], [15] and the ADE (APOC Development Environment) infrastructure [9], [16], [17].

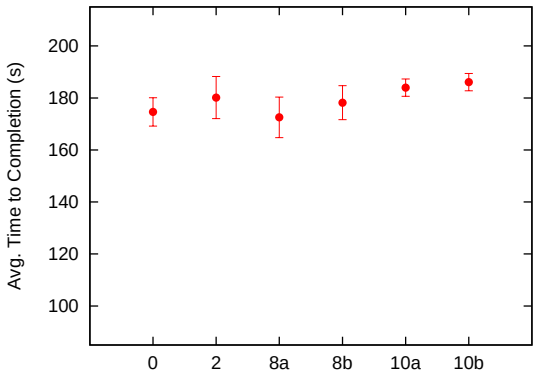
A. Experimental Setup

Experiments consist of a joint human-robot exploration task in which a subject works in tandem with a real robot, simulating a hypothetical space scenario. The goal of the team is to gather data about a nearby area, transmit it to a remote location (an orbiting spacecraft), and return to the starting point as quickly as possible. The possible task completion states are: *Success* (both transmission and return succeed), *Failure* (both transmission and return fail), and *Abort* (transmission fails, but return succeeds).

The left side of Figure 1 shows a map of the experiment environment, depicting the phases of the task and one possible robot path (the dotted line) for *Success*. The robot's initial location is in the upper left quadrant of the map; when the task starts, the human directs the robot to the exploration area in the upper right quadrant using natural language (phase A). A command causes the robot to enter an autonomous, fixed action sequence for exploration (phase B); when operating in autonomous mode, the robot neither listens for nor executes commands from the human. Once phase B completes, the robot again enters interactive mode and accepts commands from the human, who directs it along a path towards the transmission area (phase C). Teammates are interdependent during this phase: the human knows the general vicinity from which to transmit, but must rely on the robot's sensors to find the precise location; the robot requires direction from the human to find that location. After

Experiments ending in *Success* (Component Recovered)

Condition	Recovery Type	Mode at Time of Failure	Avg. Time	Std. Dev.
0.	Any/All	*n/a*	174.62	5.44
2.	Infra.	Autonomous	180.15	8.11
8a.	Both	Autonomous	172.54	7.81
8b.	Both	Autonomous	178.18	6.53
10a.	Both	Interactive	183.95	3.34
10b.	Both	Interactive	186.07	3.33



Experiments ending in *Abort* (Component Not Recovered)

Condition	Recovery Type	Mode at Time of Failure	Avg. Time	Std. Dev.
5.	Arch.	Autonomous	117.11	4.08
6.	Arch.	Interactive	147.21	3.66
7a.	Both	Autonomous	111.88	3.51
7b.	Both	Autonomous	93.16	2.58
9a.	Both	Interactive	145.31	2.26
9b.	Both	Interactive	146.98	8.82

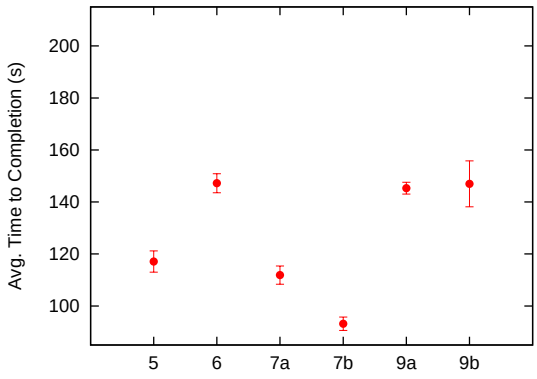


Fig. 2. Average time to task completion. The "Condition" column reflects the line from the table in Figure 1; notification is indicated by appending an "a" (disabled) or "b" (enabled) to the line number. **Left:** Tasks resulting in *Success* state; for comparison's sake, condition 0 presents performance when no component failure occurs. **Right:** Tasks resulting in *Abort* state. No results are presented for Tasks ending in the *Failure* state.

successful transmission, the robot again enters autonomous mode, planning and following a path back to the starting location (phase D).

Experiments consider *catastrophic component failure*; that is, inducing an unclean failure of the "Speech Recognition" component in either phase B or C. There are three experimental variables: (1) *active recovery type* (*infrastructure*, *architecture*, or *both*), (2) *operation mode* when failure occurs (*autonomous* or *interactive*), and (3) whether component recovery is possible. Failures in B occur 3 seconds after entering autonomous mode; while the infrastructure detects the failure when it happens, the architecture does not–because no interaction is taking place, speech recognition is inoperative. Only upon re-entering interactive mode will the architecture attempt to access the speech component and detect the failure. Failures in C occur 18 seconds after re-entering interactive operation; both architecture and infrastructure detect them as they occur.

The table on the right of Figure 1 lists the variables' combinations and possible outcomes (the cases where neither recovery type is active always result in *Failure* and are not shown). In lines 1-4, only infrastructure recovery is activated. A temporary failure while in autonomous mode is the only case not resulting in *Failure*: the infrastructure recovers the component before the architecture detects failure. Lines 5 and 6 reflect the cases where only architectural recovery is active; because infrastructure recovery is inactive, failures are always permanent. In each case, the task is aborted–although transmission is impossible, the robot can return. Finally, lines 7-10 encompass the cases where both types of recovery are enabled; component recoverability dictates the completion state. When the architecture detects a failure, it obtains information from the infrastructure about the recovery status: permanent failures lead to *Abort*, while temporary failures allow the architecture to wait for recovery, then continue.

Because it does not affect the task completion state, the additional condition of *proactive communication* is not explicitly shown in Figure 1. However, while not affecting overall task success or failure, the *notification* of failure can be exploited to improve task performance in some cases, which are presented as part of the results section.

B. Experimental Results

Figure 2 summarizes the experimental results, which are divided into two groups: the left shows the *Success* outcomes (i.e., both transmission and return were completed), while the right shows the *Abort* outcomes (i.e., the robot was able to return, but did not complete data transmission). Data appears in the same order as the table in Figure 1, where the values in the first column reflect a line number from the other table (*Failure* outcomes, lines 1, 3, and 4, are not shown, as the completion time would be ∞ in each case). Because notification is not explicitly shown in Figure 1's table, it is indicated by appending a letter to the line number where necessary (an "a" or "b" represents *disabled* or *enabled*, respectively). Below each table is a graph of average completion times of five runs and their standard deviations. While experimental conditions were kept as uniform as possible, some variance is inevitable when working with a real robot (e.g., internal system timing, slight path variations that affect travel distances, natural language pauses, etc.).

First, consider experiments resulting in *Success*. All exhibit similar completion times, and are comparable to task performance when no failure occurs; the distance travelled by the robot is approximately the same for each run, and travel

time dominates other factors. However, one difference of note is attributable to "Mode at Time of Failure". When the failure occurs in autonomous mode, infrastructure recovery completes before the robot re-enters interactive mode (i.e., attempts to use the speech component), so task completion time depends solely on infrastructure recovery time. However, when failure occurs during the interactive phase, both types of recovery are engaged. The architecture queries the infrastructure about failure status and is informed that recovery is in progress; whereas the task would otherwise be aborted, the system is able to wait for recovery to complete, then finish the task.

The *Abort* experiments (i.e., the robot returns to its starting location, but does not complete transmission), exhibit more pronounced effects from different combinations of recovery types and communication. The graph shows three distinct horizontal "bands" of similar completion times (in descending order): (1) items 6, 9a, and 9b, (2) items 5 and 7a, and (3) item 7b. For conditions in band (1), failure occurs during interactive mode and is detected at approximately the same time and position for every run; the *Abort* phase follows a nearly identical trajectory each time. This is similar for items in band (2), where failure is detected just after the autonomous phase completes (about 18 seconds earlier than detection in band (1), and closer to the home base).

The sole data point in band (3) would fall into band (2), except that in this condition *failure occurs in autonomous mode with active notification*. This allows a decision at the architecture level to abort immediately (i.e., during, rather than after, phase B), saving approximately 10 seconds of execution time compared with band (2). In contexts where resources are severely constrained, or for long-running tasks where substantial time and effort would be expended before noticing the failure, notification would be of great benefit.

V. Conclusion

We have introduced a novel communication methodology between components of a robotic architecture and its implementing software infrastructure. The information exchange enabled by the mechanism between the two levels can be used for a variety of purposes to improve system robustness and performance. Here we specifically focused on failure recovery and experimentally demonstrated that the achieved level of integration leads to more robust systems by yielding high levels of reliability under a variety of failure conditions. Specifically, we demonstrated (1) that enabling both architectural and infrastructural failure recovery allows the system to make "correct" choices in a wider range of failure conditions than possible with either alone (e.g., to avoid unnecessarily aborting task execution in some cases), and (2) that information exchange (i.e., notification) can be used to improve task performance under some circumstances (e.g., when the architecture cannot immediately detect the need to respond to a component failure). While specific performance improvement is somewhat dependent on implementation details, the demonstrated results are due to the characteristics of infrastructure and architecture recovery. Overall, the proposed mechanisms provide a critical enabling technology for robust long-term interaction and sustainable robot autonomy.

Next steps will include an even closer interaction and tighter integration between architectural (i.e., task-specific) and infrastructure-level (i.e., system-state) knowledge, allowing for internal reasoning about resources in the infrastructure that can influence task scheduling and actions of the architecture in a resource-sensitive way. We believe that by actively monitoring and managing computational resources and hardware device information, the infrastructure might be able to provide the architecture with a rudimentary form of "bodily awareness" that could provide important information about the state of the robot for maintaining system health and decision-making about task selection.

References

[1] J. Trafton, N. Cassimatis, M. Bugajska, D. Brock, F. Mintz, and A. Schultz, "Enabling effective human-robot interaction using perspective-taking in robots," *IEEE Transactions on Systems, Man and Cybernetics*, vol. 25, no. 4, pp. 460–470, 2005.

[2] J. Laird, A. Newell, and P. Rosenbloom, "SOAR: An architecture for general intelligence," *Artificial Intelligence*, vol. 33, pp. 1–64, 1987.

[3] A. Rao and M. Georgeff, "BDI-agents: from theory to practice," in *Proceedings of the First Intl. Conference on Multiagent Systems*, 1995.

[4] R. Ichise, D. Shapiro, and P. Langley, "Learning hierarchical skills from observation," in *Proc. of the Fifth International Conference on Discovery Science*, 2002, pp. 247–258.

[5] R. Brooks, "A robust layered control system for a mobile robot," *IEEE Journal of Robotics and Automation*, vol. 2, no. 1, pp. 14–23, 1986.

[6] M. Montemerlo, N. Roy, and S. Thrun, "Perspectives on standardization in mobile robot programming: The carnegie mellon navigation (CARMEN) toolkit," in *IROS 2003*, vol. 3, 2003, pp. 2436–2441.

[7] B. Gerkey, R. Vaughan, and A. Howard, "The Player/Stage project: Tools for multi-robot and distributed sensor systems," in *Proceedings of the 11th International Conference on Advanced Robotics*, 2003, pp. 317–323.

[8] C. Côté, D. Lètourneau, F. Michaud, J. Valin, Y. Brosseau, C. Raievsky, M. Lemay, and V. Tran, "Programming mobile robots using RobotFlow and MARIE," in *Proceedings IEEE/RSJ International Conference on Robots and Intelligent Systems*, 2004.

[9] V. Andronache and M. Scheutz, "Integrating theory and practice: The agent architecture framework APOC and its development environment ADE," in *Autonomous Agents and Multi-Agent Systems*, 2004, pp. 1014–1021.

[10] R. Vaughan, B. Gerkey, and A. Howard, "On device abstractions for portable, resuable robot code," in *Proceedings of IROS 2003*, Las Vegas, Nevada, Oct 2003, pp. 2121–2427.

[11] E. Gat, "On three layer architectures," in *Artificial Intelligence and Mobile Robots*. AAAI Press, 1998.

[12] B. Williams, M. Ingham, S. Chung, and P. Elliott, "Model-based programming of intelligent embedded systems and robotic space explorers," *Proceedings of the IEEE*, vol. 9, no. 1, pp. 212–237, 2003.

[13] B. Lussier, R. Chatila, F. Ingrand, M. Killijian, and D. Powell, "On fault tolerance and robustness in autonomous systems," in *Proceedings of the 3rd IARP-IEEE/RAS-EURON Joint Workshop on Technical Challenges for Dependable Robots in Human Environments*, 2004.

[14] M. Scheutz, P. Schermerhorn, C. Middendorff, J. Kramer, D. Anderson, and A. Dingler, "Toward affective cognitive robots for human-robot interaction," in *AAAI 2005 Robot Workshop*, 2005.

[15] M. Scheutz, P. Schermerhorn, J. Kramer, and C. Middendorff, "The utility of affect expression in natural language interactions in joint human-robot tasks," in *Proceedings of the 1st ACM International Conference on Human-Robot Interaction*, 2006, pp. 226–233.

[16] J. Kramer and M. Scheutz, "ADE: A framework for robust complex robotic architectures," in *IROS 2006*, Bejing, China, 2006.

[17] M. Scheutz, "ADE - steps towards a distributed development and runtime environment for complex robotic agent architectures," *Applied Artificial Intelligence*, vol. 20, no. 4-5, 2006.