Reflection and Reasoning Mechanisms for Failure Detection and Recovery in a Distributed Robotic Architecture for Complex Robots

Matthias Scheutz and James Kramer
Artificial Intelligence and Robotics Laboratory
Department of Computer Science and Engineering
University of Notre Dame
Notre Dame, IN 46556, USA
Email: {mscheutz,jkramer3}@cse.nd.edu

***Abstract*— Complex robots that interact naturally with humans require the integration, coordination and maintenance of many diverse software components and algorithms. An architecture that incorporates explicit knowledge about the relationships among these components and the overall system state can be used for introspection and consequently to reason about the best configurations of the computing environment under changing conditions; potential uses include maintaining the system's integrity, promoting its health, and providing the ability to dynamically reconfigure system components (e.g., after component failure).**

In this paper, we describe a rudimentary reasoning system, part of our Distributed Integrated Affect Reflection Cognition (DIARC) architecture for human-robot interaction, that can autonomously perform failure detection, failure recovery, and system reconfiguration of distributed architectural components to ensure sustained operation and interactions. We demonstrate the functionality and utility of the proposed mechanisms on a robot, where architectural components are forcefully removed by hand and automatically recovered by the system *while* the robot is continuing its interactions with humans as part of a joint human-robot task.

I. Introduction

The design of architectures for complex robots poses a number of challenges, including *structural* (C1), *control* (C2), and *infrastructure* (C3) challenges. Structural challenges include the selection of algorithms and components (e.g., for natural language processing, discourse management, multi-modal communication, etc.) and their organization within the architecture to allow for proper information exchange. Control challenges occur in the context of coordinating the interactions of components, which typically execute in parallel and possibly asynchronously. Infrastructure challenges include mechanisms for distributing components over multiple CPUs and/or hosts to achieve the required real-time performance while ensuring that individual software or hardware failures do not disrupt system operation.

Work on robot architectures typically focuses on (C1) and (C2), while work in distributed computing typically focuses on (C2) and (C3). And even though both fields address some of the control challenges, there is little to no overlap, as the control problems occur at different levels. We believe that it is critical for complex robots to address all three challenges together, for several reasons: (1) to be able to achieve the integration of a large number of diverse components within one architecture, (2) to manage and operate these components over the long term in a coordinated fashion, and (3) to react coherently to contingencies (such as software or hardware failures) at different levels of the system.

In this paper, we address all three challenges with our complex *Distributed*, *Integrated*, *Affect*, *Reflection*, and *Cognition* architecture, DIARC. After briefly describing DIARC, we focus on a novel reflection and reasoning mechanism that improves the reliability of highly distributed architectures by detecting and recovering from hardware and software failures. We demonstrate the utility of the proposed mechanism on a robot implementation of DIARC in a joint human-robot task where components of the architecture and/or hardware are purposely shut down and successfully recovered–*during task performance*–without human intervention.

II. Autonomic Computing Mechanisms for Sustained Robot Operation

Complex robots, especially ones that interact naturally with humans, require the integration, coordination and maintenance of many diverse software components and algorithms. For seamless operation of the components, the integration requires designers to explicitly address several *structural* (C1), *control* (C2), and *infrastructure* (C3) challenges. Yet, issues related to (C1) make architecture modification and extension very difficult, potentially requiring new communication interfaces, data representation formats, and connection methods. Difficulties related to (C2) such as the adjustment and/or adaptation of component operation and timing, input-output formats, and component functionalities may affect other components with a cascade effect. Issues concerning (C3) include architecture brittleness and difficulties in component monitoring and fault detection, resulting in the nearly impossible integration of mechanisms for dynamic, automatic recovery of failed components and system (re)configuration in reaction to unforeseen contingencies. Consequently, *ad-hoc* integration, while possibly a first step in the prototyping phase of architecture design, is not a viable option for sustained autonomous operation of social robots or robot companions.

Robots that need to operate for extended periods of time will require mechanisms that ensure robust and reliable functioning (e.g., run-time system modification, failure detection and recovery, and automatic system (re)configuration capabilities), which can be all viewed as aspects of *autonomic computing* [2] intended to enhance a system's ease of use and availability. The necessity of including autonomic computing principles in the design of a robotic architecture that extend the robot's time of operation is based on a simple observation: in the long term, *all systems will inevitably fail*, often for reasons that cannot be controlled by the designer. The challenge, therefore, is not to anticipate all possible problems, causes, and solutions at design time, but to design mechanisms that can detect and resolve such problems at run-time. This conclusion is succinctly expressed in [12]: "We consider errors by people, software, and hardware to be facts, not problems that we must solve, and fast recovery is how we cope with these inevitable errors".

The idea, then, is to equip the robot with general mechanisms for detecting component failures and recovering from them quickly, while allowing component-specific functionality to use them. We will, in the following, provide an overview of a complex architecture for human-robot interactions (e.g., as part of a mixed human-robot team) that incorporates knowledge about the relationships among components and the overall system state, then demonstrate the utility of the autonomic computing mechanisms in the context of a human-robot team task.

III. A Proposal for a Robust Complex Architecture for Human Robot Interaction

Architectures for complex, mobile, autonomous robots that have reliable and natural interaction with humans require, at the very least *appropriate interaction capabilities* (R1), including natural language capacity (speech recognition and speech production), dialogue structure (knowledge about dialogues, teleological discourse, etc.), affect recognition and expression (for speech as well as facial expressions), and mechanisms for non-verbal communication (via gestures, head movements, gaze, etc.), as well as mechanisms for *mobility* (R2), including obstacle detection and avoidance, path planning and navigation, map making and localization, and others. Such complex architectures also require *mechanisms for ensuring robust interactions* (R3), including recovery from various communication failures (acoustic, syntactic, semantic misunderstandings, dialog failures, etc.) as well as software and hardware failure recovery (crashes of components, internal timing problems, faulty hardware, etc.).

To meet requirements (R1), (R2), and (R3) we have defined DIARC, an architecture that integrates typical cognitive tasks (such as natural language understanding and complex action planning and sequencing) with lower level activities (such as multi-modal perceptual processing, feature detection and tracking, and navigation and behavior coordination) [17], [16], [6] . To support DIARC in meeting requirement (R3), all DIARC components are implemented as "servers" in the JAVA-based infrastructure framework `ADE`, which provides robust, reliable, fault-tolerant middle-ware services for the distribution of complex robotic architectures over multiple computers and their parallel operation, including monitoring, error detection, and recovery services [14], [7].

A. DIARC -A Distributed Integrated Affect Reflection Cognition Architecture

DIARC provides several features that are not readily found in other robotic architectures. First, DIARC is a complete knowledge-based architecture that meets (R1) and can be employed in human-robot interactions without any structural modification. Task knowledge and thus appropriate behaviors for the task can be represented as both declarative and procedural knowledge, expressed in the form of *scripts* that contain information about action sequences, events, and associated goals and outcomes. Moreover, DIARC is built on the robust multi-agent infrastructure *agent development environment* `ADE` (described in the next section) that treats components of the robot architecture as agents in a multi-agent system (MAS), thus allowing the distribution of architectural components over multiple hosts and providing support for the automatic detection of component faults and for subsequent error recovery, meeting (R3). The utility of DIARC for human-robot interaction has been evaluated in a variety of experiments, both qualitatively and quantitatively, in constrained and unconstrained scenarios (i.e., lab experiments with humans and natural interactions at various AAAI robot competitions) [18], [17], [16].

Figure 1 shows a "3-level" view of the DIARC architecture implemented in `ADE`, which is also used for all experiments reported later. The top level shows parts of the functional decomposition of DIARC in terms of sensory, perceptual, deliberative, action, and effector components. The middle level shows the mapping between the high-level functional components and their implementation in the `ADE` framework. Boxes at this level depict autonomous computing components that operate in parallel and communicate via several types of communication links (discussed below). The bottom level depicts the hardware in the system; components are located above the host on which they execute.

B. The `ADE` Infrastructure

`ADE` [1], [6], [15] is an agent system that combines support for the development of complex individual architectures with the infrastructure of a multi-agent system. One of the goals in developing `ADE` is to allow easy expansion of computing power in the system, such that architecture designers do not have to be concerned with many low-level aspects of agent deployment and execution, instead allowing them to focus on the functional roles and relations of components in the architecture. Space constraints make a full account of `ADE` impossible (for a more detailed description, see [15]); here we only describe the aspects of the framework related to failure detection and recovery.

The basic component in `ADE` is an `ADEServer`, which is comprised of one or more computational processes that

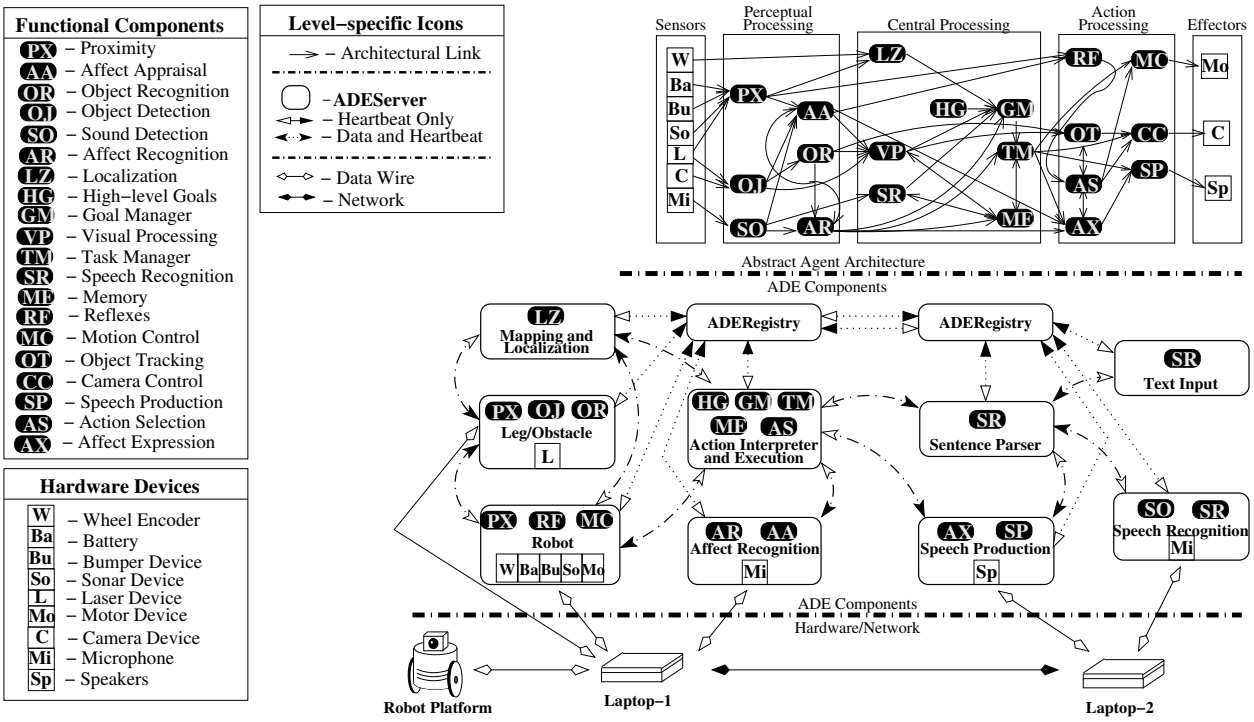

Fig. 1. The DIARC architecture (top) implemented in ADE (middle) running on the robot hardware (bottom), see text for details.

serve requests. Accessing those services is accomplished by obtaining a *reference* to the (possibly remote) ADEServer, referred to as an ADEClient. The ADERegistry, a special type of ADEServer, mediates connections among ADEServers and the processes that use their services. In particular, an ADERegistry organizes, tracks, and controls access to ADEServers that register with it, acting in a role similar to a *white-pages service* found in multi-agent systems. The ADERegistry provides the backbone of an ADE system; all components must register to become part of the architecture. A set of ADE components may contain multiple ADERegistrys that mutually register with one another, providing both redundancy and the means of maintaining distributed knowledge about the system.

All connected components of an implemented architecture (i.e., ADEServers, ADEClients, and ADERegistrys) maintain communication links during operation, consisting of periodic *heartbeat* signals indicating that a component is still functioning. An ADEServer sends a heartbeat to the ADERegistry with which it is registered, while an ADEClient sends one to its originating ADEServer. The receiving component periodically confirms heartbeat reception; if none arrive, the sending component receives an error, while the receiving component times out. An ADERegistry uses this information to determine the status of ADEServers, which in turn determine their accessibility. Similarly, an ADEServer uses heartbeats to determine the status of its ADEClients, which in turn determine if the ADEServer's services remain available.

In Figure 1, a dotted line indicates a client/server connection over which a heartbeat is sent, where the solid arrowhead indicates the originating ADEServer and the empty arrowhead indicates the component receiving the ADEClient. A dashed/dotted line is used to represent a connection over which both a heartbeat signal and other data is transferred. Hardware devices used by an ADEServer are depicted by a set of labeled squares within a rectangle. Two relations between the bottom and middle levels are shown: (1) ADEServers are placed in vertical columns above the host on which they execute and (2) connections between hardware devices and the ADEServers that use them are indicated by solid lines that cross the separating line. The relation between the middle and top levels consists of darkened ovals that represent a functional architectural component of an agent within an ADEServer's rectangle.

C. Reasoning in the Infrastructure

ADERegistrys provide the backbone of agent architectures implemented in ADE, for they contain information about the entire system, including both active and uninstantiated ADEServers, known hosts, relationships among components, etc. For instance, on system start-up, part or all of an architecture layout is stored in a *configuration file*, which an ADERegistry uses to start the various components. This information is then entered into a knowledge base; as the configuration file is read, items are *asserted* as facts. During system operation, facts can be *retracted*, replaced, or newly discovered facts about the system state can be added. The knowledge can be used for both *introspection* and *reasoning* about components, the system's status, and the computational environment in which the system resides.

Description	Predicate
Hardware device	`device(D)`
`ADEHost`	`host(H)`
`ADEServer`	`server(S)`
Device available	`has(H, [D...])`
Device required	`requires(S, [D...])`
Can `ADEServer` S be located on `ADEHost` H?	`canhost(H, S) :- host(H), server(S), has(H, C), requires(S, D), contains(C, D).`
Can `ADEServer` S be moved to `ADEHost` H?	`canmove(S, H) :- host(H), server(S), free(H, D), requires(S, C), contains(D, C).`

Fig. 2. Sample subset of facts and rules stored in the knowledge base.

The top of Figure 2 shows a subset of the facts available to the infrastructure, while the bottom of the figure shows some rules used for failure recovery that rely on those facts (all expressed in Prolog). Specifically, we are concerned here with the facts and policies involved in failure recovery of one or more architectural components, either due to individual component failure or catastrophic hardware failure. As described above, all components in an `ADE` system maintain *heartbeat* signals; a missing heartbeat indicates a failed component. In the case of an `ADEServer`, this causes an attempt at reconnection, in addition to receiving notification that can potentially be used to react in a component-specific way to the failed server (e.g., if the "Speech Recognition" component fails, rendering the robot unable to understand commands, the motor can be shut off for safety reasons). Notification is also sent on reconnection, allowing an `ADEServer` to internally re-adjust its operation. In the case of an `ADERegistry`, the failure detection event causes it to enter a failure recovery procedure, during which attempts are made to restart the component. These failure recovery procedures are part of an `ADERegistry`, and are executed automatically.

If multiple `ADERegistry`s are present in a system, then each registry will be responsible for its locally registered components, but will in addition provide another level of redundancy, as it will register with all the other registries as well. This will ensure that the registry can be recovered if it fails as long as at least one other registry is functional. Hence, in such a system with multiple registries, complete failure is only possible if all registries become dysfunctional at the same time (a very unlikely event).

The reasoning module becomes particularly useful when multiple `ADEServer`s must be restarted. For recovery, the reasoning module can be used to assign an `ADEHost` for each failed `ADEServer`, using rules that describe possible actions the system can perform. System constraints may lead to situations in which recovery of one server interferes with or makes it impossible to recover another server. For example, when recovering a distributed architecture, if a server with moderate demands $S1$ is started on host $H1$, followed by recovery of a computationally demanding server $S2$ on the same host, performance of both servers may be adversely affected. A reasoning module may determine that $S1$ should be relocated to host $H2$ that perhaps has a higher load than $H1$ initially, but which will lead to a better average load distribution throughout the system if only $S2$ is placed on $H1$ (putting $S1$ on $H2$ instead). In essence, this is a form of *system optimization*; the same mechanisms can be applied either at start-up or dynamically at run-time, providing autonomous tuning of an agent's operation.

More importantly, representations of facts about the system state provide new possibilities for introspective reasoning to individual components in the architecture, such as task planners, reasoning components, and the like. It is now possible for a component to request information about the states of other components in the distributed `ADE` system for various purposes. For example, a navigation planner could request information about available sensors and thus adjust its planning behavior dynamically based on system state (e.g., a robot that had both sonars and laser range sensors at the beginning of an operation but subsequently lost the laser due to a hardware failure might still be able to complete its mission if the navigation subsystem is able to take inputs from the sonar sensors instead of the lasers, even though its preferred input is laser data – we will demonstrate a simple form of this component substitution in the experiment section). Another example is a task planner that, based on updated information about problems with the sound subsystem, determines that certain tasks requiring spoken natural language interactions cannot be achieved. It can then decide whether to drop the task, find alternatives, or simply notify an operator (e.g., via email) about the problem [8].

The current implementation of the reasoning module uses a Prolog reasoning engine, which is itself encapsulated in an `ADEServer` that can be distributed or relocated. A different reasoning engine can be substituted, so long as it implements the `assignhost(S)` function. For instance, a cognitive architecture like SOAR [9] might replace Prolog; in doing so, it would gain the ability to reflect on and reason about its own architectural structure and the rules that govern system management.

D. Related Work

In the introduction, three "challenges" were identified: (C1) structural, (C2) control, and (C3) infrastructure; we attempt to provide a very brief representative selection of related research. Player/Stage [4], CARMEN [11], and RobotFlow/FlowDesigner/MARIE [3] are robotic development systems that focus on (C1) and (C2). An examination of the requirements for preserving system health in a hostile environment that considers a reflective architecture, focusing on (C3) and using an immune system analogy, can be found in [5]. [13] addresses challenges (C2) and (C3) by integrating "intelligent sensors" in architectural components such that

self-health awareness permeates the system, from influencing the control system to providing external situational awareness used to affect the agent's goals. Probably the most related work is found in [10], which attempts to address all of the challenges. However, in addition to relying on a single, centralized program for component management, it does not appear to incorporate a reasoning module.

IV. Experimental Validation

To evaluate the utility of ADE's error detection and recovery mechanisms, which rely on introspection and reasoning about infrastructure configurations, we conducted experiments with an assistive robot that has to interact with humans using natural language in a joint human-robot task. The experiments here are extensions of previous work that demonstrated recovery from *catastrophic failures*, where an entire host went down [6]. All components running on that host were restarted on another host and the robot was able to continue its mission with a short delay. However, the main weakness of the previous demonstration was that if the ADERegistry itself was located on a host that subsequently failed, the whole system was rendered useless. Moreover, if hardware (e.g., a sound device) failed that made the execution of components (e.g., speech recognition) impossible, there was no way to substitute functionality for the missing component. In such cases, the robot, while partially operational, was unable to complete the task (e.g., because it could not take commands from the human operator).

The experiments here are intended to address both problems: (1) by using multiple ADERegistrys, the robot can recover even from failures of critical infrastructure recovery components, and (2) by using introspection on the type of a component, the reasoner can substitute a component with similar functionality (e.g., a text input server) for a malfunctioning one (e.g., a speech recognizer with broken microphone input). Moreover, we show a general, statistically significant speed-up of the recovery process over our previous experiments due to the integrated reasoner that can quickly determine the most appropriate action to take based on the type of failure.

A. Experimental Setup

The task chosen for the evaluation takes place against the backdrop of a hypothetical space scenario [17]. A mixed human-robot team on a remote planet needs to determine the best location in the vicinity of the base station for transmitting information to the orbiting space craft. Unfortunately, the electromagnetic field of the planet interferes with the transmitted signal and, moreover, the interference changes over time. The goal of the team is to find an appropriate position as quickly as possible from which the data can be transmitted. The robot is dependent on its human teammate for direction, supplied through natural language commands, while the human is dependent on his robotic teammate for "field strength" readings that cannot be obtained through other means. The specific goal of the team is to find a viable transmission location and send the data, at which point the task is accomplished.

Fig. 3. The robot used for the experimental evaluation.

Experiments consider simulated *software and hardware failures*, in particular, the effects of failures of architectural components and hardware devices *during task performance*.

For the experimental evaluation, we used an ActivMedia Peoplebot (shown in Figure 3) with a pan-tilt-zoom camera, a SICK laser range finder, three sonar rings, and two onboard PC laptops with 1.3GHz and 2.0GHz Pentium M processors. Both laptops run Linux with a 2.6.x kernel and are connected via an internal wired Ethernet. A wireless interface on one laptop enables system access from outside for the purpose of starting and stopping operations. Figure 1 shows the initial configuration of DIARC for all experiments and the assignment of ADE components to the two PCs.

B. Experiments and Results

We consider three failure scenarios: (1) *recoverable component* failure, (2) *recoverable registry* failure, and (3) *irrecoverable component failure with component substitution*, in addition to (4) a baseline experiment without failures.

For the first scenario, the "Speech Production" component is manually terminated via an OS "kill" signal, and is immediately recovered by the registry that is responsible for it. In the second scenario, a registry is terminated the same way, and is recovered by the other registry. Upon recovery, it connects back up with the components for which it is responsible. In the third scenario, the "Speech Recognition" component is brought down and an audio-device failure is simulated so that a restart of the component is not possible, and there is no other host with the necessary hardware audio input device. Upon fault detection, the reasoner determines that another component in the system, the "Text Input" component, can assume the functional role of the "Speech Recognition" (as it provides the same method interface that other currently running components require, and connects that component instead).

We conducted 15 experimental runs in each of the four conditions and used the time it took the human to navigate the robot from an initial condition through the obstacle environment to the transmission point as objective performance measure. We conducted a four-way ANOVA with *condition* as independent and *time-to-task completion* as dependent variables on the resulting run-times, and found no main effect ($F(3, 56) = .17, p = .91$), indicating that error detection and recovery was performed quickly enough so as to not impact overall task performance, as evidenced by the lack of a statistically significant difference among the four conditions.

V. Discussion and Conclusion

While the above experiments demonstrate that ADE's reflection and reasoning mechanisms can achieve detection and recovery of failed components without any impact on task performance in the above task, it is clear that the extent to which this will be true in other tasks will intrinsically depend on the nature of the failure – a component (like a speech synthesizer) not used at the time of failure might be recovered in time for its next use versus the whole effector subsystem failing during a navigation task. We have demonstrated ADE's capabilities qualitatively using several failure events (of different architectural components in DIARC) in different scenarios in our lab (e.g., similar to the ones mentioned at the end of Section III-C), both in situations where recovery was possible and impossible. Whenever recovery was possible in principle, ADE was able to recover from failures and the robot was able to resume its mission.

The above experiments are part of our ongoing attempts at *quantifying* the utility of the employed error detection and recovery mechanisms in a variety of scenarios. While the fact that recovery can be accomplished autonomously and during task performance is certainly desirable for robots that have to perform tasks autonomously without direct supervision or without the option of human intervention in the operation if problems occur, it is even more important to demonstrate, as we have done here, that the impact on task performance can be potentially very small, which is a highly desirable property in many situations (e.g., in time-critical tasks). To our knowledge, there are currently no other robotic architectures that have demonstrated this kind of online failure detection and recovery *on a robot* when major parts of the computational infrastructure fail, demonstrating the full recovery from failure *during a human-robot team task without human intervention* with only negligible effects on the overall task performance.

Finally, it is worth pointing out that the proposed mechanisms enable new possibilities for both architecture and infrastructure to share knowledge and use each other's capabilities to improve the robustness and sustainability of robot operations. For example, we are currently investigating the utility of giving the infrastructure access to the current goals of the high-level cognitive system to both dynamically determine that certain components may be shut down because they are not needed and to (re)start components in anticipation of their use. Conversely, on noticing that the system's health is deteriorating, the high-level system may decide to seek out its human operator or plan to go in for maintenance servicing (a function already possible in DIARC). We believe that it is this integration of architecture and infrastructure that will ultimately ensure the sustained, long-term operation of intelligent autonomous robots, and thus provide an enabling technology for the long-term interactions, as required for assistive robots and robot companions.

References

[1] 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*, pages 1014–1021, 2004.

[2] D. Bantz, C. Bisdikian, D. Challener, J. Karidis, S. Mastrianni, A. Mohindra, D. Shea, and M. Vanover. Autonomic personal computing. *IBM Systems Journal*, 42(1):165–176, 2003.

[3] 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.

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

[5] C. Kennedy and A. Sloman. Reflective architectures for damage tolerant autonomous systems. Technical Report CSR-02-1, University of Birmingham, School of Computer Science, 2002.

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

[7] J. Kramer, M. Scheutz, J. Brockman, and P. Kogge. Facing up to the inevitable: Intelligent error recovery in massively parallel processing in memory architectures. In *International Conference on Parallel and Distributed Processing Techniques and Applications*, Las Vegas, 2006.

[8] J. Kramer, M. Scheutz, and P. Schermerhorn. On integrating form and function: Multi-level dynamic failure recovery for autonomous robots. submitted.

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

[10] N. Melchior and W. Smart. A framework for robust mobile robot systems. In *Proc. of SPIE: Mobile Robots XVII*, volume 5609, 2004.

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

[12] D. Patterson, A. Brown, P. Broadwell, G. Candea, M. Chen, J. Cutler, P. Enriquez, A. Fox, E. Kiciman, M. Merzbacher, D. Oppenheimer, N. Sastry, W. Tetzlaff, J. Traupman, and N. Treuhaft. Recovery-oriented computing (ROC): Motivation, definition, techniques, and case studies. Technical Report UCB//CSD-02-1175, CS, UC Berkeley, 2002.

[13] K. Reichard. Integrating self-health awareness in autonomous systems. *Robotics and Autonomous Systems*, 49(1-2):105–112, November 2004.

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

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

[16] M. Scheutz, P. Schermerhorn, J. Kramer, and D. Anderson. First steps toward natural human-like hri. *Autonomous Robots*, page forthcoming, 2007.

[17] 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*, pages 226–233, 2006.

[18] 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.