An Architectural Style Perspective on Dynamic Robotic Architectures

John Georgas and Richard Taylor

***Abstract*— We focus on the problem of developing robotic architectures which are well-suited to supporting runtime evolution, rather than specific evolution techniques. Based on the insights provided by current robotic architectures, we elaborate on their characteristics and how well they support the quality of evolvability. Combined with insights from architectural styles developed by the software engineering community, we outline the RAS architectural style: a layered, component- and connector-based, event-based style intended to provide architecture-level support for evolvable robotic architectures.**

I. INTRODUCTION

The domain of robotic systems is one in which the importance of systems capable of reacting to unpredictable conditions has long been recognized. To achieve this goal, the robotic architectures research community has developed *situation-* and *task-driven* architectures which take environmental unpredictability into account. These task-oriented architectures focus on explicitly specifying the environmental conditions under which a task is applicable and explicitly accounting for the determination of a task's success or failure. What these architectures, however, do not account for is evolving systems at runtime in order to modify the set of already existing robot behaviors or add new ones without suspending operation – a feature refer to as *runtime evolution*.

The software engineering community has developed architectural techniques to support the development of systems with such runtime evolution capabilities, primarily focused on *dynamic architectures*. These architectures are models of software systems which are changeable at runtime with model changes dynamically reflected on the system's implementation. These research efforts are focused on making the architectural model the locus for easier and more understandable change enactment.

In this paper, we pursue an examination of various robotic architectures currently in use and investigate their combination with insights from the software architecture community. The goal is to elaborate on an integration of research from these two domains aimed at better supporting runtime evolution in robotic architectures; we focus on fundamental architectural support which fosters runtime adaptation rather than discussing specific methods for achieving it.

This work sponsored in part by NSF Grants CCF-0430066, and IIS-0205724.

J. Georgas is with the Informatics Department, Donald Bren School of Information and Computer Sciences, University of California, Irvine, Irvine, CA 92697, USA jgeorgas@ics.uci.edu

R. Taylor is with the Faculty of Informatics Department, Donald Bren School of Information and Computer Sciences, University of California, Irvine, Irvine, CA 92697, USA taylor@ics.uci.edu

To this end, we begin by examining three robotic architectures: the canonical SUBSUMPTION architecture, three-layer (3L) approaches as exemplified by the 3T architecture, and the reactive concentric (RC) architecture of the WITAS project. For each of these robotic architectures, our discussion addresses the foundations of the architecture and then continues with a discussion of the consequences these concepts have in the context of runtime evolution – much of the historical insight into these architectures and their relationships comes from [1].

This paper continues with a discussion of how *architectural style* insights combine with the needs of the robotics domain in order to elaborate on a requirements statement for dynamic robotic architectures. The fundamental contribution of this paper is the presentation of the RAS architectural style, which combines insights from these fields of study to better support robotic architectures which are more flexible and changeable at runtime. The main features of the RAS style include: the strong use of *components* and *explicit connectors*, limited and *directional event-based* communication, *explicit layering* which is preserved during implementation, and a *dynamically changeable* architecture. These characteristics are specifically intended to promote the effective construction of runtime evolvable robotic systems, and the style has already been used in the construction of prototype self-adaptive robotic architectures.

II. BACKGROUND AND RELATED WORK

The section will set the architectural context for the remaining discussion by presenting the basic notions of software architecture, architectural styles, and dynamic software architectures.

A. Software Architecture

Software architecture models most commonly capture a software system's structure expressed through interconnected *components*, which are responsible for the software system's computation [2]. Other perspectives on software architecture also explicitly model the *connectors* which manage and enable communication between components. Architectures clearly identify the high-level elements present in a software system, and make their interconnections explicit.

Architectures are captured through *architecture description languages* (ADLs), which are syntactically precise languages for describing the architectural structure of systems. Software engineering research has produced a variety of ADLs, usually revolving around specific modeling concerns. Architecture development environments, such as ARCHSTUDIO [3], focus on providing support for using architectures

and ADLs in software development by providing tools for the specification, visualization, maintenance, analysis, and deployment of architectures.

It is important to note that for architectural models to provide their full utility to software development, they have to be more than just descriptive models which have no bearing to the actually implemented architecture. Architecture must be prescriptive in nature, and model the architecture of the final product in order for the qualities pursued by the architecture and the analyses performed at the architectural level to have any bearing to the software system being developed.

B. Architectural Styles

Another important concept in software architecture is that of the *architectural style*: a codification of design decisions which are applied to the construction of systems [4]. Architectural styles may capture domain knowledge or be designed to promote qualities which are of particular importance: the C2 style [5], for example, was specifically intended for the development of GUI systems. Architectural styles are a critical concept in achieving high-level reuse of both specific architectural solutions as well as domain expertise.

C. Dynamic Software Architectures

Dynamic architectures focus on techniques which allow systems to be changed at runtime by modifying the architectural model of these systems [6]. Architecture-based runtime evolution focuses on coarse-grained changes enacted on architectural elements and is aimed at providing flexibility and dynamism, so that evolution does not necessarily require foresight during development but can also be enacted long after system deployment.

Focusing on this level of abstraction allows for better cognitive grasp over the process and a clear understanding of the interconnections which may be relevant in specific contexts. Additionally, this approach focuses attention on high-level issues rather than being mired in low-level concerns critical to enactment but conceptually less important. These approaches rely on development infrastructure and architecture-to-implementation mappings to ensure that modifications are reflected on the system's implementation. This is the perspective adopted by this paper: the driver of change in system behavior is a change in the topology of the system's software architecture.

III. ROBOTIC ARCHITECTURES

This section will discuss three robotic architectures from the research literature in the context of runtime evolution. We will examine each of the SUBSUMPTION, 3L, and RC architectures from an architectural perspective, looking at their general architectural foundations and discussing how their characteristics support architecture-based runtime evolution.

A. Subsumption

The SUBSUMPTION architecture [7] was the first significant departure from the *sense-plan-act* (SPA) paradigm of robotic software construction and achieved early success over SPA. Despite the benefits of SUBSUMPTION, it suffered from serious drawbacks particularly centered on its scalability and reliability in large and complex systems; these drawbacks hindered its adoption and motivated the development of alternative architectural solutions.

1) Architectural Foundations: SUBSUMPTION architectures are, in essence, data-flow style architectures with flows originating from sensors and terminating at actuators. Interposed in these flows are the architecture's components, which respond to events in the environment with requests for action. SUBSUMPTION architectures adopt a layered decomposition and capture behaviors as independent components or component compositions which operate in parallel; layers group components together according to their relative complexity and sophistication. Communication between these components is facilitated by the *inhibition* and *suppression* operations: with these operations, higher-level modules are allowed to inhibit the communication of modules as well as replace normal communication with data flows of their own.

2) Runtime Evolution: While no explicit support for runtime evolution could be found in the SUBSUMPTION literature, these architectures hold a powerful lesson: encapsulating behaviors in independent modules communicating asynchronously with no assumptions about the reliability of communications results in highly-modular systems. In SUBSUMPTION architectures, the topology of the architectural composition reflects the behavior of the system itself and modifying behavior is a matter of altering the topology. This focus on architectural elements is a key factor enabling architecture-based runtime evolution.

One important drawback of SUBSUMPTION, however, is that it does not support principled layering of components. The topological location of a component is no indication as to which layer it belongs to, and modules of one layer can interact with modules from any other layer. The existence of layers and component layer membership is wholly conceptual and neither evident in nor enforced by the architecture. The communication mechanisms used also impose another important restriction: components wholly override lower-level modules, which prevents component interactions based on the fusion of information. These drawbacks limit the granularity of reuse to individual components as well as limiting the kinds of interactions which can be used during construction and evolution: both these limitations hinder flexibility and evolvability.

B. Three-Layer Architectures (3L)

3L architectures strongly adopt layers as an abstraction mechanism based on differences in overall component tasks and treatment of state information. 3L architectures and their variants (differences lying mostly in specific implementation choices) have risen to become a very common strategy for robotic architecture construction – the specific exemplar which will be used for this discussion is the 3T architecture [8].

1) Architectural Foundations: As the name would suggest, three-layer architectures focus on constructing robotic

control systems by decomposing the system into three layers: a *skill* layer, a *sequencing* layer, and a *planning* layer.

The skill layer focuses on defining a collection of low-level behaviors which do not maintain long-term state and are responsible for actions which depend on quick reaction. The sequencing layer is responsible for composing behaviors from the skill layer into chains of actions to achieve tasks: the sequencing layer encapsulates much of the behavioral complexity involved in building robots. The sequencing layer is also usually limited to maintaining historical state information, but without attempts to forecast future conditions. The planning layer is the most abstract of the layers and focuses on operations which take a long time to complete relative to other included behaviors; this layer encapsulates long-term goals and performs time-consuming tasks such as generating complete plans of action and environmental projections.

2) Runtime Evolution: While no support for runtime evolution was found in the 3L literature, the strongest benefit of these architectures for this activity lie in their defining architectural characteristic: their use of practically realized layers and the graceful interconnection of these layers through providing input to rather than taking over the operation of other layers. This strong use of abstraction in actual implementations enables independent operation between components as well as between layers, and supports modularity and incrementality in both.

The most significant drawback of 3L architectures for architecture-based runtime evolution is their treatment of the sequencing layer. In the architectures studied for this paper, the sequencing layer is implemented through special-purpose language scripts which are interpreted at runtime. Though this implementation choice is not included in the conceptual basis for 3L architectures, the practical adoption of this method to the exclusion of others leads us to consider it here as a drawback of the methodology in general. The use of these special-purpose languages has a significant architectural consequence: the behaviors they embody are not evident nor can they be affected through architectural means. Differences in behavior in 3L robots are not evident from the software architecture itself, but are primarily expressed in the special-purpose code executed by the sequencing layer. As the architectural topology is not the primary driver for behavioral characteristics, 3L architectures do not naturally lend themselves toward architecture-based runtime evolution.

C. Reactive Concentric (RC)

RC architectures [9] adopt the notion of a *task procedure* (TP) as the central abstraction: these encapsulate the sequence of activities that must take place for a behavior to be achieved. While conceptually maintaining the notion of 3L-style control layers, RC architectures do not exhibit layering in their implementations and instead adopt a middleware-centric approach based on CORBA.

1) Architectural Foundations: Conceptually, RC architectures adopt the *control* layer for providing low-level services, the *deliberative* layer for providing time-consuming services, and the TP layer which forms the link between the two; this layering mirrors the 3L separation into skill, deliberative, and sequencing layers. In terms of the actual architectural implementation, however, this conceptual division is abandoned in favor of a loosely coupled, event-based bus architecture that supports non-layered interactions. The notion of layers, then, is not one that is evident in the architectural topology, as all components essentially belong to a single layer. TPs, which are managed by a special-purpose module, are specified in a special purpose *Task Specification Language* (TSL), which is used to generate stubs forming the basis of custom-coded TP implementations.

2) Runtime Evolution: While – once again – there is no explicit support discussed in the literature for runtime evolution, RC architectures have a number of features which make them well-suited to it. RC architectures exhibit a component-based development methodology with event-based communication: these features allow for the development of systems which exhibit a high degree of flexibility and decoupling, which are critical qualities for dynamic architectures. This methodology also supports the runtime initiation and composition of services, which provide the necessary infrastructure for runtime evolution.

However, RC architectures also exhibit a number of drawbacks. The division between layers of control in an architecture is a purely conceptual one not evident in the architecture's implementation. RC architectures also mirror one of the drawbacks of 3L systems: the TP layer is implemented through the use of special-purpose artifacts which are not architecturally visible. When examining an RC architecture, what can be seen is a coarse-grained component responsible for the enactment of an unknown number of task procedures with architecturally undeterminable content. This low architectural visibility of behaviors has the consequence of making RC architectures unsuitable for the application of architecture-based evolution techniques.

IV. Dynamic Architectures for Robotic Systems

This section addresses the intersection of principles on system construction from the software architecture domain – as codified in architectural styles – and the domain of robotic system development – as exemplified in the preceding discussion of the Subsumption, 3L, and RC architectures. First, we discuss the nature of the problem space and the requirements an architectural style must address to be effective, and then we present and discuss an architectural style formulation which composes features from robotic architectures as well as existing architectural styles.

A. Problem Space

The problem space of interest is dynamic robotic architectures which can support runtime evolution. This class of systems can support the modification of their behavior – at runtime and without service interruption – in response to a variety of stimuli. A full discussion of the methods through which this evolution can be achieved is beyond the current scope, but our particular research approach can be found in [10]. More specifically, we are interested in

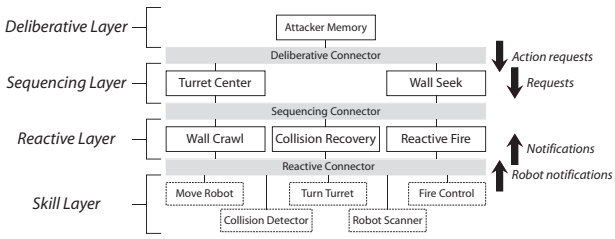

Fig. 1. A partial illustration of the *ArchWall* robot architecture, with annotations pointing out important characteristics of the RAS architectural style.

systems which provide runtime evolution capabilities in an architecture-centric manner in which changes in behavior are brought about through changes in the system's architecture. The benefits of this strategy in this domain mirror those of the general case of architecture-based evolution: architectural models provide a more expressive, understandable, modular, and reusable basis for system evolution than lower-level approaches, such as those centered on source-code artifacts.

An architectural style specifically targeted toward this problem space, therefore, must support both the effective construction of robotic systems themselves as well promote qualities which allow for their runtime change. More specifically, support for the following qualities is critical:

- *Evolvability*. Given the focus on runtime change, it is critical that the architectural style used supports the easy evolution of architectures.
- *Incrementality*. It is important to support both incremental development as well as incremental evolution.
- *Reuse*. A high degree of reuse in terms of both components and component groupings is important in reducing development and evolution effort.

B. The RAS *Architectural Style*

This section proposes a new architectural style named RAS intended for the development of dynamic robotic architectures. The development of this style – already in use in our research activities – is aimed toward providing architecture-level support for the qualities discussed in the immediately preceding section; its development is based on a study of robotic architectures already discussed as well as results from architectural styles stemming from software engineering research.

We focus the discussion of this style on the example found in Fig. 1, which illustrates a part of the architecture of the *ArchWall* robot constructed according to the RAS style. This specific architecture was developed to operate within the ROBOCODE [11] infrastructure in the context of research into the development of self-adaptive robotic systems; while not embodied in a physical robot, this simulation-based platform allows us to focus on the fundamental software engineering concerns of robotic architectures. In ROBOCODE, robots engage in combat with other competing robots in a battlefield where the objective is to destroy opponents while ensuring survival. This simple example robot follows the battlefield's walls, searches for and moves toward a wall if it finds itself far from one, rotates the turret so that it is always facing the center of the battlefield, fires at any target it scans, and maintains information about robots which have injured it to adjust firing strength accordingly. The *ArchWall* robot also performs a number of evolutions through architectural modifications: it changes its targeting and scanning strategies at runtime in response to its level of energy and the number of active opponents in the battlefield.

The major characteristics of the style, as illustrated and annotated in the example of Fig. 1, are:

- *Component-Based*. A fundamental premise of this style is its component-based nature: systems are built through the composition of independent components each encapsulating a single task or behavior. Components are unaware of the existence or identity of other components in the architecture. Followed by both the SUBSUMPTION and, to a lesser extent, the RC approach, this supports the high degree of modularity, incrementality, and reuse necessary to create systems with high evolvability.
- *Explicitly Layered*. The RAS style adopts explicit layering of architectures and partly follows the conceptual delineation and naming conventions of 3L architectures. Four high-level layers compose RAS architectures: the *deliberative*, *sequencing*, *reactive*, and *skill* layers. These layers are mainly differentiated by the requirements on their reaction time, their treatment of state information, and the complexity of their interactions with other system components. While the style can support further layering within each of the above mentioned layers, these four layers form the conceptual and implementation basis for RAS-style architectures. It is important to note that the layering is also reflected in the actual implementation of the system rather than just the conceptual design. The explicit separation of the control architecture in independent layers – supported and encouraged through communication conventions – further promotes modularity and understandability as well as incrementality and reuse at the granularity of layers rather than just components.
  The skill layer – which has different meaning in the context of the RAS style than in 3L systems – contains components which form the foundational capabilities of the robot: components belonging to this layer are the interface between the control software and the hardware actuators and sensors which gather information and execute mechanical actions. In the example of Fig. 1, the *Robot Scanner* component emits notifications of other robots detected by the scanner, while the *Turn Turret* component performs turret movement in response to directives. This layer is closely tied to the capabilities of the hardware platform, and remains relatively static when the system is evolved. The skill layer "pushes" information to the architecture through notifications of robot state changes and also provides information in

response to explicit requests by other components.

The reactive layer is composed of elements which address actions the robot must take immediately after the receipt of information from the skill layer which indicates the need for action; this layer captures the quick "reflexes" of the robotic system and also forms a core of functionality that higher layers can access – this corresponds to the 3L skill layer. Components belonging to this layer do not maintain state information but atomically react to events as they are received: the *ReactiveFire* component of the illustrative architecture of Fig. 1, for example, immediately fires at enemies as they are detected and has no memory of past firing actions. This non-reliance on state optimizes the reaction time of components in this layer, while the capability to maintain state and historic information is relegated to higher layers of the architecture through the emission of notifications which indicate which actions were taken.

The sequencing layer contains components which are characterized by more complexity in the tasks and behaviors they embody and address actions which are not simply reactive in nature but may involve more complex modes of interaction between layers. The *Wall Seek* component of the Fig. 1 architecture, for example, attempts to locate the nearest wall for the robot to move to by explicitly requesting information about the current location of the robot and the size of the battlefield before making its decision; this request-response interaction is one that is unavailable to components of the reactive layer. Sequencing components may also maintain short-term state information which is used in the course of the component's computation, but – similarly to the conventions of 3L architectures – does not attempt to make predictions about the future. Components in this layer have access to any capabilities and services provided by all lower layers.

Finally, the deliberative layer of the RAS style is comprised of components which have the capability of storing long-term state information and may be concerned with forecasting future state – this corresponds to the 3L planning layer. Components of this layer have full access to all provided services in the architecture, and may engage in complex interactions with these components involving multiple message exchanges. In the specific example of Fig. 1, the *Attacker Memory* component stores information about attackers and directs the system to fire more strongly at those enemies. While not predictive in nature, this component is an example of the long-term state maintenance aspect of the RAS deliberative layer.

- *Event-based Communication*. The RAS style adopts the event-based communication paradigm of the Event Based Integration (EBI) style. Communication between components is accomplished through the exchange of four types of events: an *action request* that communicates to the skill layer that an action – either atomic or composite in nature – must be taken; a *request* that is used to request information or an action from components in the architecture in layers other than the skill layer, a *robot notification* which conveys information originating in the skill layer, and a *notification* which conveys state or operational information originating in components not in the skill layer. These events are initiators of action through implicit invocation.

  The RAS style also adopts directional message conventions similar to those of the C2 architectural style: requests may only be transmitted to layers below the originating layer while notifications can only be transmitted to layers above: for example, requests can be sent from the sequencing layer to the reactive layer but notifications cannot. While limiting, these restrictions are important in that they provide architectural support for the layering of components, and they promote careful design of the relationships between components to more easily identify the appropriate layer for each. Additionally, event-based communication further promotes modularity and reduced coupling between components, and therefore enhances the evolvability of RAS-style architecture.
- *Connector-based Communication*. Another characteristic of the RAS style is its reliance on explicit connectors. Each of the four layers discussed in the previous paragraphs is separated from the others by an explicit layer connector, as illustrated in Fig. 1. Connectors are architectural elements responsible for communication: in the RAS style, these connectors broadcast messages to all connected elements in accordance to the directional conventions already described. These connectors, which may also be included as part of intra-layer arrangements, also act as points of access into the inner workings of the architecture and provide support for explicit monitoring of communication. These facilities provide an increased degree of modularity and visibility into the operation of robot architectures, particularly when compared to the opaque nature of the communication in the SUBSUMPTION and 3L architectures.
- *Dynamic Architecture*. The RAS style adopts the dynamic architecture paradigm described in Section II: the architectural model of the system is maintained independently from its implementation and evolution is achieved through modifications to the architectural model, which are then translated to consistent changes in the system's running implementation. This approach – supported by facilities of the ARCHSTUDIO environment – provides for a higher degree of visibility and couches system evolution at a level of abstraction appropriate to an architecture-based approach.

In our research into self-adaptive systems, we have applied the RAS style in the construction of ROBOCODE architectures with the capability to evolve at runtime. Most importantly, the application of the style was critical in enabling runtime adaptation itself: conventionally constructed ROBOCODE robots preclude runtime adaptation. In addition

to the enablement of runtime adaptation, RAS also provided the necessary degree of modularity and reusability to allow the modification of behavior through changes to only the components relating to those behaviors while allowing the remainder of the architecture to remain unchanged.

C. Stylistic Implications

The characteristics of this style are intended to support the needs of the problem domain. Modularity is a fundamental characteristic supporting evolvability: the component- and event-based nature of the style combined with explicit connectors are features intended to support this quality. A high degree of decoupling is also a consequence of the use of event-based communication rather than utilizing direct interconnections between components. The adoption of layers – both conceptually as well as practically in the actual implementation of the system – promotes incrementality in the development of each layer as well as enabling reuse at a coarser granularity than just components. Finally, explicit connectors also increase the visibility of the architecture and the communication taking place between the components of the system, as these connectors can be used for event monitoring and logging.

Adopting the stylistic conventions of the RAS style, however, also carries the potential for detrimental effects. The adoption of an event-based communication paradigm, for example, means that there is the potential for message loss and the consequent loss of possibly important information. This potentially lossy communication between components also implies that service fulfillment is not guaranteed: components may make requests that are never responded to either because the message was never received or because the recipient component elected to not service the request. Event-based systems may also exhibit poor scalability as the number of events emitted by components grows too large for the system to handle; however, the layered and directional event flow nature of the architectural style proposed partially alleviates this issue, as the effect of "event storms" is lessened by the fact that components are partially shielded from sets of system events. The reactive nature of the robotics domain also makes problems that event-based solutions exhibit with large-scale data transfers less likely.

V. Conclusion

Our examination of robotic architectures currently in use revealed a number of areas where robotic architectures can be improved for use with runtime evolution by being informed by the successes of software engineering. The most important weaknesses we identified are a lack of explicit architectural layering and clearly encapsulated behaviors, and the use of special-purpose elements which make understanding systems and evolving them at runtime challenging. We believe that these shortcomings can be addressed through the combination of insights from robotic architectures and general purpose architectural styles. To that end, this paper outlines our proposal of the RAS architectural style, which is currently in use in the context of our research into self-adaptive robotic architectures. The main features of RAS are a strong component-based development focus, explicit layering that is maintained during implementation, event-based implicit invocation of services with directional flow of event types, the adoption of explicit connectors, and the use of dynamic architecture techniques. These features are intended to support the development of systems which are amenable to runtime evolution by providing explicit support for such evolution as well as promoting important qualities in this context, such as modularity, incrementality, and reuse.

Our future work in this direction will be centered on the further elaboration of stylistic elements which continue to support the runtime evolution of robotic architectures. We intend to further examine the literature on robotic architectures for additional features to be integrated into RAS, as well as conduct further studies in order to evaluate the style on larger and more complex systems for applicability and scalability. We also plan on examining how the current elaboration of the style can be augmented with further support for reuse not only at the level of components or layers, but also through the reuse of specific component groupings across layers. Finally, we intend to deploy RAS-style architectures on actual robotic platforms in order to examine the effects of the stylistic principles in the real world arena.

References

[1] E. Gat *et al.*, "On three-layer architectures," *Artificial Intelligence and Mobile Robots*, 1997.

[2] D. E. Perry and A. L. Wolf, "Foundations for the study of software architecture," *ACM SIGSOFT Software Engineering Notes*, vol. 17, no. 4, pp. 40–52, 1992.

[3] *ArchStudio, An Architecture-based Development Environment*, Institute for Software Research, University of California, Irvine, http://www.isr.uci.edu/projects/archstudio/.

[4] M. Shaw and P. Clements, "A Field Guide to Boxology: Preliminary Classification of Architectural Styles for Software Systems," *Proceedings of the 21st International Computer Software and Applications Conference*, pp. 6–13, 1997.

[5] R. N. Taylor, N. Medvidovic, K. M. Anderson, J. E. James Whitehead, J. E. Robbins, K. A. Nies, P. Oreizy, and D. L. Dubrow, "A component- and message-based architectural style for gui software," *IEEE Transactions on Software Engineering*, vol. 22, no. 6, pp. 390–406, 1996.

[6] P. Oreizy, "Open architecture software: a flexible approach to decentralized software evolution," Ph.D. dissertation, University of California, Irvine, 2000.

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

[8] R. Bonasso, R. Firby, E. Gat, D. Kortenkamp, D. Miller, and M. Slack, "Experiences with an architecture for intelligent, reactive agents," *JETAI*, vol. 9, no. 2-3, pp. 237–256, 1997.

[9] P. Doherty, P. Haslum, F. Heintz, T. Merz, P. Nyblom, T. Persson, and B. Wingman, "A distributed architecture for autonomous unmanned aerial vehicle experimentation," *Proceedings of the 7th International Symposium on Distributed Autonomous Robotic Systems*, 2004.

[10] J. C. Georgas and R. N. Taylor, "Towards a knowledge-based approach to architectural adaptation management," in *WOSS '04: Proceedings of the 1st ACM SIGSOFT workshop on Self-managed systems*. New York, NY, USA: ACM Press, 2004, pp. 59–63.

[11] M. A. Nelson, "Robocode," 2006, http://robocode.sourceforge.net.