Policy-Based Architectural Adaptation Management: Robotics Domain Case Studies

John C. Georgas[1] and Richard N. Taylor[2]

[1] Department of Computer Science, Northern Arizona University
Flagstaff, AZ 86011 USA
John.Georgas@nau.edu

[2] Institute for Software Research, University of California, Irvine
Irvine, CA 92697 USA
taylor@ics.uci.edu

Abstract. Robotics is a challenging domain that exhibits a clear need for self-adaptive capabilities, as self-adaptation offers the potential for robots to account for their unstable and unpredictable deployment environments. This paper focuses on two case studies in applying a policy- and architecture-based approach to the development of self-adaptive robotic systems. We first describe our domain-independent approach for building self-adaptive systems, discuss two case studies in which we construct self-adaptive Robocode and Mindstorms robots, report on our development experiences, and discuss the challenges we encountered. The paper establishes that it is feasible to apply our approach to the robotics domain, contributes specific examples of supporting novel self-adaptive behavior, offers a discussion of the architectural issues we encountered, and further evaluates our general approach.

1 Introduction

One of the major current challenges in software engineering is the development of self-adaptive systems, which are systems that are able to change their behavior in response to changes in their operation or their environment for a variety of goals. In contrast to more specialized terms such as self-healing or self-optimizing, we use the term self-adaptive to inclusively refer to this class of systems, without consideration for their specific adaptive goals.

In addition to self-adaptive software, we are also keenly interested in robotic systems. These systems are amalgams of software and hardware that are highly resource constrained, commonly deployed in environments out of reach of human operators, exhibit a high degree of dependence on events in their environment, and commonly required to perform functions to which there can be little interruption. The domain characteristics naturally motivate the inclusion of self-adaptive capabilities that are currently lacking. The focus of this paper is the intersection of the robotics domain with self-adaptive software, as we see both a driving need for such capabilities in robotic systems as well as a fruitful application domain for self-adaptive technologies.

B.H.C. Cheng et al. (Eds.): Self-Adaptive Systems, LNCS 5525, pp. 89–108, 2009.
© Springer-Verlag Berlin Heidelberg 2009

The challenge of integrating self-adaptive capabilities into robotic systems is a two-front battle: First, the system itself must be built in such a manner as to be conducive to adaptation by virtue of its very design and construction. Only then can adaptive behavior be integrated into the system. In our research efforts, the fact that the construction of the system must support adaptation implies that modularity is one of the fundamental qualities. This quality is the key enabler that allows adaptation to take place in a fine-grained manner rather than adaptation through wholesale replacement. In addition, due to the fact that adaptive needs are virtually impossible to fully and correctly predict during design, we also posit that adaptive behavior must be built in a way that is flexible and modifiable at runtime.

Much of our previous work has been dedicated to the development of architecture-based self-adaptive systems, and we identify a clear parallel in the capabilities this work provides and the needs of self-adaptive robotic systems. More specifically, we have developed notations and tools that support the design and development of policy- and architecture-based self-adaptive systems that are modular and have the ability to change adaptation policy specifications during system runtime [1]. Our goals with the specific work described in this paper are to:

- establish the feasibility of integrating our policy- and architecture-based self-adaptive system research with robotics domain;
- develop novel self-adaptive capabilities in robotic systems that did not previously exhibit them; and,
- probe into the difficulties and pitfalls of such an integration effort.

This paper is a report of our work to date toward these goals and our experiences in striving to meet them. We describe two case studies we performed in developing self-adaptive robotic systems, beginning with our work in the Robocode system – a robotic combat simulator and development framework – and continuing by discussing the construction of an autonomous Mindstorms NXT robot. For each of these systems, we demonstrate practical self-adaptive solutions to practical domain challenges. Neither of these domains previously considered – much less provided support for – self-adaptive capabilities.

The key contributions of our work in this intersection between self-adaptive architectures and robotic systems are:

- verifying the feasibility of integrating robotics and architecture-based self-adaptive techniques;
- providing examples of novel self-adaptive capabilities in our case-study domains;
- uncovering an important architectural mismatch between architecture-based adaptation and current robotic domain practice; and
- demonstrating that our policy language is adequate for expressing robotic adaptations.

The remainder of the paper offers background information, presents our overall approach to self-adaptive systems, discusses our case studies, and concludes with future work and final remarks.

2 Background and Related Work

This section begins with a discussion of representative robotic architectures, paying particular attention on their support for runtime change. We also discuss related approaches to architecture-based self-adaptive systems.

2.1 Robotic Architectures

One of the first robotic control system architectures to gain wide acceptance was the *sense-plan-act* architecture (SPA) [2]. In SPA, control is accomplished through the *sense* component that gathers information from sensors, the *plan* component that maintains an internal world model used to decide on the robot's actions, and the *act* component that is responsible for executing actions.

SPA architectures, however, scale poorly as robotic systems grow in complexity and scope, and SUBSUMPTION [3] was developed to address these scalability issues. This architecture abandons world models and adopts layered compositions of reactive components. Communication between these components takes place through the *inhibition* and *suppression* of inputs and outputs of lower level components by higher level ones. While the component-based approach of this architecture allows for improved scalability and modularity, the supported modes of communication prove very limiting.

Most current robotic systems are heavily influenced by three-layer (3L) architectures, first described in [4]. These hybrid architectures separate robotic systems into three explicit layers and mix reactive and planning modes of operation: The *reactive* layer captures behaviors that quickly react to sensor information, the *sequencing* layer chains reactive behaviors together and translates high-level directives from the *planning* layer into lower-level actions; the *planning* layer is responsible for deciding on long-term goals.

Despite their differences, these robotic architectures share a commonality in their lack of support for runtime adaptation, discussed in more detail in our paper to a workshop attached to the International Conference on Robotics and Automation (ICRA) [5]. These architectures simply do not consider this concern in their design and therefore do not exhibit the necessary qualities to be amenable to the direct application of self-adaptive techniques – the minimally amenable, perhaps, is the SUBSUMPTION architecture, due to its focus on independent components.

This lack of support for the architectural qualities that promote ease of runtime change is the motivation for the development of the RAS architectural style, also described in [5]. The style combines insights from event-based architectural styles such as C2 [6] and the SUBSUMPTION and 3L robotic architectures, and is aimed at supporting the development of robotic architectures that are modular

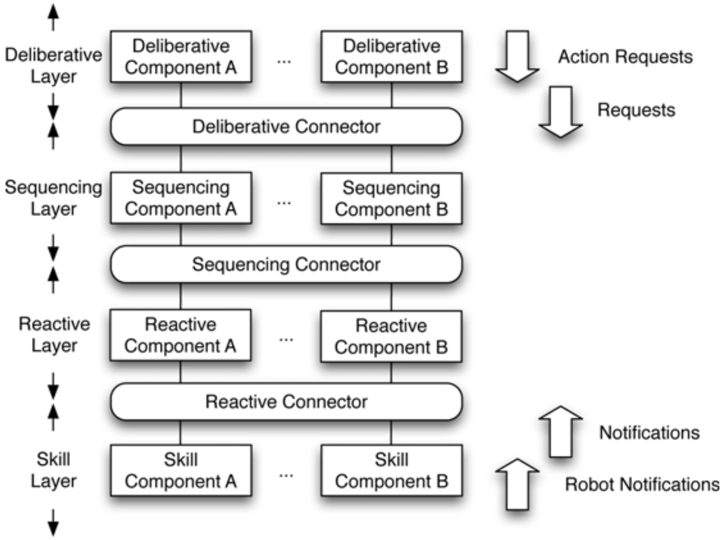

Fig. 1. An illustration of the RAS architectural style, showing the style's layers and event types

and incrementally evolvable while fostering component reuse. Figure 1 outlines the style, which is:

- component-based, with no shared memory between components;
- explicitly-layered into *skill*, *reactive*, *sequencing*, and *deliberative* layers[1] with components belonging to layers based on their complexity and maintenance of state information;
- event-based with communication taking place between components of the architecture through *requests* and *notifications*, sensor information being transmitted by *robot notifications*, and actions being enacted through *action requests*, and;
- connector-based, with independent connectors separating layers and facilitating communication and distribution.

The robotic systems we build in our case studies are built according to the principles of this style, which provides the basis for the construction of robotic systems that foster modularity and, therefore, are more easily modifiable using architecture-based means.

2.2 Self-Adaptive Architectures

Some related work takes a formal approach to the specification of architectures and the artifacts governing adaptation. The work based on Community [7], for example, models architectures as abstract graphs while the approach based

[1] While the RAS style uses similar layer names as 3L architectures, there are differences between the two; the reader is referred to [5] for further details.

on the Darwin [8] architecture description language (ADL) focuses on the self-assembly of systems according to a formally specified set of constraints representing invariant architectural properties. Other approaches are more focused on providing practical tool support for developing self-adaptive architectures. The Rainbow system [9] adopts a style-based approach and focuses on the specification of styles for specific domains along with style-specific adaptations and constraints tailored for the domain's needs. Our own work is a descendant of such a tool-based approach [10], which conceptualized architecture-based adaptation but left many questions about how to implement adaptive behavior unanswered.

There is also work in the intersection of robotic systems and architecture-based approaches: Applied to sophisticated hardware platforms, the Shage framework [11] supports the definition of adaptive strategies managed by a controlling infrastructure, but focuses only adaptations that replace components with alternatives providing similar services. Kramer and Magee have also discussed self-adaptive robotic architectures through the application of a conceptual framework strongly influenced by 3L architectures and focused on self-assembling components using a formal statement of high-level system goals [12]. In contrast, the approach described here embodies a fundamental trade-off away from formal specifications of system behaviors in order to achieve a higher degree of flexibility and support for the runtime change of adaptation policies without necessitating the re-generation of adaptation plans.

3 Approach

Before discussing the case-studies, we present here the high-level approach we use for developing self-adaptive robotic systems. As this paper is focused on the application of our approach to robotic architectures, we keep the discussion minimal; a more extensive discussion of the overall approach appears in [13].

The core of our policy-based approach to architectural adaptation management (PBAAM) appears in Fig. 2. Self-adaptive systems in this approach consist of three fundamental parts: an architectural model specifying the system's structure, a set of adaptation policies capturing how the structure changes, and executable units of code corresponding to each architectural element. These three artifacts are managed at runtime by elements of the PBAAM infrastructure: the *Architecture Model Manager* (AMM), the *Architecture Adaptation Manager* (AAM), and the *Architecture Runtime Manager* (ARM) respectively. Self-adaptive systems are further augmented by a configuration graph model that maintains information about the history of a system's adaptations, as well as a body of architectural constraints that are intended to preserve core system capabilities. These artifacts are managed at runtime by the *Architectural Runtime Configuration Manager* (ARCM) and the *Architecture Constraint Manager* (ACM) respectively.

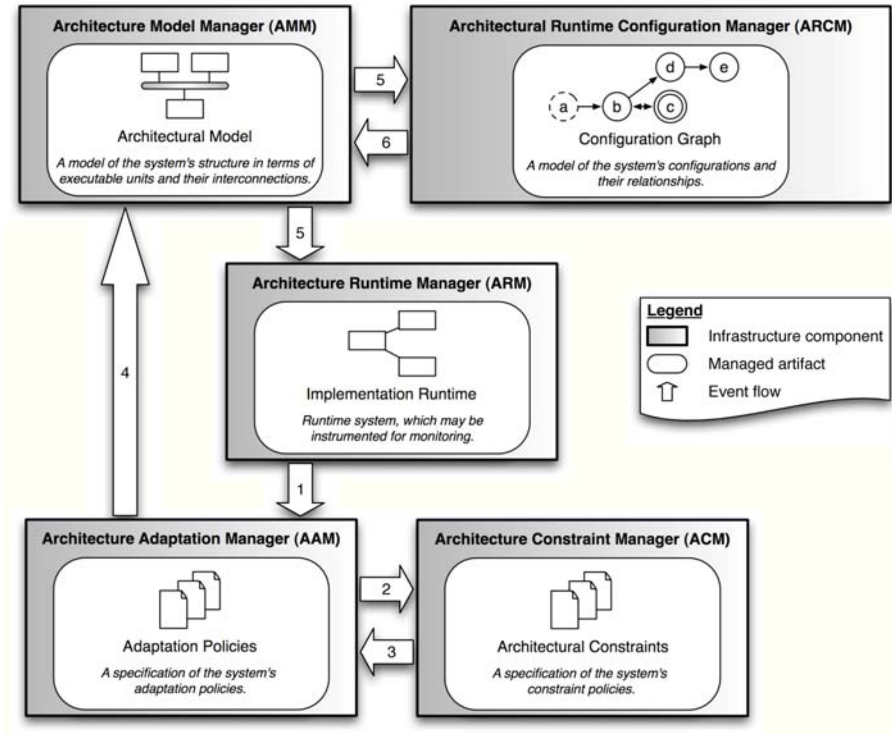

Fig. 2. The tools and activities of the PBAAM approach

3.1 Adaptation Policy Specification

One of the fundamental abstractions in our approach is the adaptation policy: a policy is an encapsulation of the system's reactive *adaptive behavior* and indicates a set of actions that should be taken in response to events indicating the need for these actions. The basic building blocks of adaptation policies are *observations* and *responses*. Observations encode information about a system and responses encode system modifications. Given the architecture-based focus of our approach, responses are limited in the kinds of actions they can perform: they are restricted to operations which change the structure of software architectures and, in essence, reduce to additions, removals, connections, and disconnections of architectural elements.

We specify the structure of adaptation policies using a xADL 2.0 schema [14] that extends the core schemas of the ADL and lays out policy structure. The basic definition of a policy appears below, with XML namespace information removed for the sake of brevity:

```
<xsd:complexType name="AdaptationPolicy">
  <xsd:sequence>
    <xsd:element name="description" type="Description"/>
    <xsd:element name="observationList" type="ObservationList"/>
    <xsd:element name="responseList" type="ResponseList"/>
  </xsd:sequence>
  <xsd:attribute name="id" type="Identifier"/>
</xsd:complexType>
```

Each adaptation policy is characterized by a unique identifier and may contain an optional, human-readable textual description that indicates its purpose to designers. Each policy also contains a list of *observations* and a list of *responses*: when this list of observations is fully satisfied, the entire set of responses is enacted. It is very important to note that this policy schema is highly extensible and can be customized to fit the needs of specific projects. One of the extensions currently under development, for example, extends the notion of policies with support for expressing observations in terms of exhibited behaviors while modeling responses as a set of desired behaviors, where STATECHART models [15] are used to specify behavior.

3.2 Architectural Adaptation Management

The AAM is the element responsible for the runtime management of the specified adaptation policies. When the self-adaptive system is first instantiated, the AAM loads the set of policies and initiates their runtime evaluation: as policies are added and removed from the policy specification, the AAM updates the set of active runtime policies to reflect these changes. The current incarnation of the AAM adopts an expert system for the runtime management of policies. In a very straightforward manner, policies are translated to executable *condition-action* rules and then managed using an expert system shell. More specifically, we adopt the Java Expert System Shell (JESS) [16] for this task, which provides us with a well-tested and efficient platform for the runtime execution of policies.

In coordination with the ARM – an existing element of the ARCHSTUDIO environment that supports runtime evolution and predates our work – the AAM drives architectural change by enacting modifications to the system's architectural model. The ARM's primary responsibility is to ensure that changes enacted to the architectural model are also enacted on the runtime system itself.

Alongside these tools that implement a reactive self-adaptation loop, two additional tools provide capabilities related to constraint and configuration management. Before changes are actually enacted, the ACM ensures that these changes would not violate a body of architectural constraints. These constraints enforce a variety of desired architecture structural properties, ranging from component membership to architectural connectivity. Only those changes that do not violate these constraints are allowed to be enacted by the AMM. The ARCM tool is responsible for monitoring architectural changes as they take place, and recording them in a configuration graph. Nodes in this graph correspond to and

maintain information about architectural configurations, while edges represent adaptations between these configurations. Each edge maintains bi-directional *diff* information, which is used by ARCM to enact explicit user commands to apply rollback or rollforward operations on the architecture (an early description of ARCM appears in [17]).

3.3 Activity Flow

Referring to the activity flows indicated in Fig. 2 by numbered arrows, the adaptation process begins when observations about the running system are collected and transmitted to the AAM (flow labeled 1). These observations are gathered through independent probing elements or through self-reporting components and encapsulate what is known about the system. This information forms the basis for evaluating adaptation policies managed by the AAM.

Any triggered responses are first communicated to the ACM (indicated by the flow labeled 2), which ensures that these suggested responses do not violate the active architectural constraints. Modifications that are deemed allowable are communicated back to the AAM, and then finally enacted by being transmitted to the AMM (event flows 3 and 4, respectively). Both the ARM and ARCM are notified of any changes enacted on the architectural model (event flow 5). The ARM then ensures that these changes are reflected on the executing system, while the ARCM builds the system's configuration graph; ARCM also allows manual modifications triggered by the user (event flow 6).

4 Case Studies

This section offers specific details about two case studies in integrating self-adaptive capabilities with robotic systems. For each of the ArchWall and Archie systems, we will discuss our experiences and call out some of the difficulties we encountered and lessons we learned in providing novel self-adaptive capabilities in domains that did not previously support them.

4.1 Robocode

Our first study was performed using the Robocode[2] system. Initially developed as a Java teaching tool, Robocode is now an open-source system under active development that provides a robotic combat framework and simulator which is used to pit robotic control systems in battle against each other.

Robocode Background. Robocode provides a customizable simulated battlefield into which robots are deployed: the objective of robots is to remain alive while destroying their competitors. Each robot may move, use its radar to detect other robots, and use its gun to fire at opponents. The constraint of primary importance for each robot is the amount of remaining energy. While all robots

[2] http://robocode.sourceforge.net

begin a battle with the same level of energy, energy is depleted by being hit by bullets or colliding with other robots or walls. Energy can also be invested into firing bullets at other robots, but a multiple of this invested energy is recovered by successfully hitting. The goal of each robot, then, is to preserve its own energy by both firing wisely as well as avoiding collisions and enemy fire.

From a software development perspective, the ROBOCODE API provides builders with basic robot control capabilities: movement and steering, control for the robot's scanner and weapon, and support for notifications of battlefield events. How each robot responds to these events is the challenge of ROBOCODE development, and the robots developed by the community vary from the very simple to the very sophisticated. It is important to note that in development for the simulator, a robot is programmed and compiled as a single static unit of code that is then executed by the battle simulator; there is no consideration or support for runtime adaptation.

Architecting ARCHWALL. The core architecture of the ARCHWALL robot follows the guidance and constraints of the RAS architectural style (discussed in Section 2.1) and appears in Fig. 3.

In contrast to ROBOCODE development that focuses on robots being monolithically implemented as a single unit of source code, ARCHWALL is comprised of a number of independent components and connectors that are distributed between the JAVA and ROBOCODE environments. These components embody a number of behaviors and are arranged in RAS layers. At the lowest level, the skill layer captures the basic tasks that a ROBOCODE robot can perform: ARCHWALL can scan for other robots, turn its turret, fire at enemies, detect collisions, and control its speed and direction. Using these fundamental facilities, the reactive layer implements a simple collision recovery strategy of stopping and moving

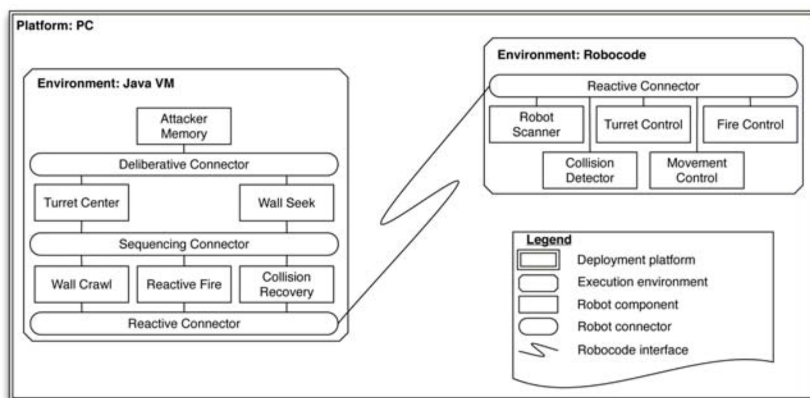

Fig. 3. The architecture of the ARCHWALL robot, showing the layered arrangement of components and connectors implementing robot behaviors, as well as the environments and platforms providing the execution context

away from collisions, firing at any enemy robot detected, and moving along a wall if the robot is near one. The sequencing layer contains components that ensure the turret is always pointed toward the center of the battlefield, and direct the robot to move to the nearest wall if it has not already done so. Finally, the deliberative layer maintains data about enemy robots that attack ArchWall. In accordance with the RAS style, components are arranged in layers according to the timeliness of their responses to events, and their maintenance of state.

The curved connection in Fig. 3 indicates the special-purpose interface we constructed in order to integrate the PBAAM infrastructure with the Robocode simulator (the integration also involved a number of modifications to the simulator framework itself). PBAAM requests for robot actions are translated into the appropriate Robocode API calls, while notifications of simulator events are translated into architectural events and transmitted to the robot's architecture.

With this architecture, ArchWall first seeks a battlefield wall and then follows it for the duration of the battle, embodying the movement behavior of a type of Robocode robots referred to as "wall-crawlers." The turret is always kept pointed toward the center of the battlefield, and ArchWall fires at any robot it detects. This initial set of behaviors is sufficient for the robot to compete in basic battles: in our testing experiences, the robot tends to rank between positions four and six in a battle against ten opponents selected from the set of sample Robocode robots that are distributed with the simulator.

Self-Adaptive ArchWall. The basic behaviors exhibited by ArchWall, while sufficient to be competitive, are certainly not optimal under a wide variety of battlefield conditions. For example, while firing at any enemy robot scanned is perfectly acceptable when the robot's energy is high, it would be helpful to exhibit a more careful firing strategy as the robot's energy is depleted. Similarly, while targeting the center is useful when there are many opponents on the battlefield, it becomes less desirable as the battle progresses and the number of enemies is lessened.

To address these shortcomings, we developed a self-adaptive version of ArchWall by placing the architecture of the robot under the management of the PBAAM toolset. This augmented architecture appears in Fig. 4; the core architecture of the robot appears in unshaded components, while the PBAAM tools are shaded. Along with the system, we deployed a number of adaptation policies addressing the need to change the firing and targeting behaviors of ArchWall as the conditions of the battle change. One policy, for example, states (in an abridged form for brevity):

```
<AdaptationPolicy id="ReplaceFiring">
  <ObservationList>
    <StringObservation>
      <StringObservationContent>
        (energy_report {energy < 60}) </StringObservationContent>
      </StringObservation>
```

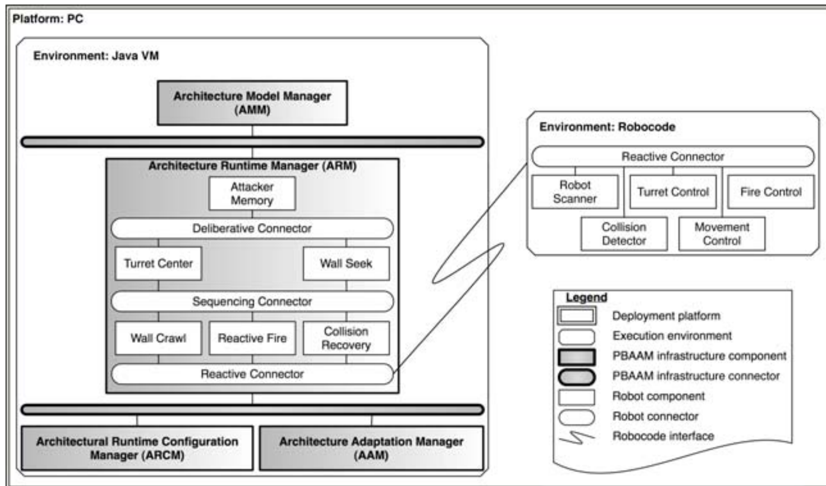

Fig. 4. The PBAAM-managed architecture of the ArchWall robot, showing the robot's architecture as well as the PBAAM tools that enable self-adaptive behavior

```
    <ResponseList>
      <RemoveComponentResponse>
        <RemoveComponent> ReactiveFire </RemoveComponent>
      </RemoveComponentResponse>
      <AddComponentResponse>
        <AddComponentIdentifier>
          DistanceReactiveFire
        </AddComponentIdentifier>
        <AddComponentType>
          DistanceReactiveFire_type
        </AddComponentType>
      ...
    </AddComponentResponse>
  </ResponseList>
</AdaptationPolicy>
```

This policy replaces the firing strategy used by ArchWall when the energy of the robot drops below the indicated threshold by replacing one component with another: *Distance Fire*, which only fires at enemies that are nearby in an attempt to maximize the chances of hitting (and, thereby recovering the invested energy) takes the place of *Reactive Fire*.

Additional adaptation policies also change the way in which the robot moves and scans for opponents as fewer enemy robots remain: in total, the ArchWall robot contains four independent adaptation policies which modify its behavior in different ways. Overall, the addition of the adaptation policies improves the

performance of the robot: the adaptive version of ArchWall tends to rank between positions two and four, while even coming in first on some test runs.

Most importantly, however, is the fact that each adaptation policy is completely independent of the architecture to which it is applied and could be added, removed, or modified during runtime as the robot continues to operate. Furthermore, the architecture and components of the robot are entirely adaptation unaware: None of the components needs to be modified for the transition from the non-adaptive to the adaptive version of ArchWall, and the only changes are those made through architectural means.

Developing the ArchWall robot clearly established the feasibility of integrating architecture- and policy-based self-adaptive software methods in robotic systems by providing novel support for developing self-adaptive Robocode robots. While this framework is admittedly limited to simulation, from a software engineering perspective it exhibits many of the same challenges that developing a self-adaptive system for any other robot does: coherently organizing and relating robot behaviors, for example, and dealing with multiple sources of input in deciding on which actions to perform.

The effort also gave us experience in dealing with an important architectural mismatch between the Robocode framework and the PBAAM infrastructure: Like most robotic system frameworks, robots supported by Robocode are developed synchronously by sequencing behaviors through their explicit ordering in the source code. This way of building systems conflicts with the asynchronous nature of our approach. As this asynchronous and modular nature is a fundamental enabler of runtime change, reconciling this mismatch was necessary and required effort in the design and implementation of each behavior in order to compensate. Each component had to be constructed in a state-based way – that is not necessary in other applications we have applied our approach to – that maintains information about the state of the interactions it is engaged in with components to which it has dependencies. This explicit maintenance of state for inter-component interactions is a key enabler for the integration of our work with robotics; decoupling this interaction modeling from components and isolating it at the architectural level is an area we are interested in pursuing in our future work.

4.2 Mindstorms NXT

We performed the second case study using the LEGO Mindstorms NXT development kit[3]. Released in the summer of 2006, this is another in LEGO's line of kits to support easily accessible and affordable development of robotic systems that has found great traction in academic settings.

Mindstorms NXT Background. Each Mindstorms kit is comprised of Technic pieces which are used to build the structure of robots, servo motors, and a variety of sensors (the commercial kit includes ultrasonic, light, sound, and touch sensors). Computer control for the sensors and motors is provided by an NXT brick: each brick supports enough ports to accommodate a maximum

[3] http://mindstorms.lego.com/Overview/

of three motor and four sensor connections, and also supports a USB port and a Bluetooth wireless connection. These kits are extremely affordable (at the time of this writing, the kits cost about $250 US) but resource constrained: processing is provided by a 32-bit ARM7TDMI microprocessor with 64KB of RAM available to it and 256KB of flash memory for non-volatile program storage.

From a software perspective, the basic platform supports development in two ways: the NXT processing brick firmware can execute user-written programs, and the development and compilation of these programs is supported through the NXT-G visual programming environment. More advanced users may leverage a large variety of third-party libraries and firmware replacements for a variety of programing languages. In the context of our discussion on self-adaptive systems, it is important to once more note that Mindstorms robots are developed as single units of code with no pre-existing support for or consideration of runtime change.

Architecting Archie**.** Building on the work described in the previous section, we continued by developing the Archie Mindstorms robot; a picture of the robot in our lab can be seen in Fig. 5.

The physical platform of the robot is a modification of a basic three-wheeled Mindstorms design. Movement is provided by two motors, each controlling one of the side wheels while the third wheel is unpowered and can freely rotate to any movement direction. A third motor opens and closes the grasping arm of the robot. The robot is equipped with the following sensors: A touch sensor which is mounted in place to detect when an object is within grasping range, a light sensor which detects the light reflectivity of the surface the robot is on, and an

Fig. 5. A picture of the Archie Mindstorms robot: a three-wheeled design with a grasping arm and a number of mounted sensors along with the NXT processor brick

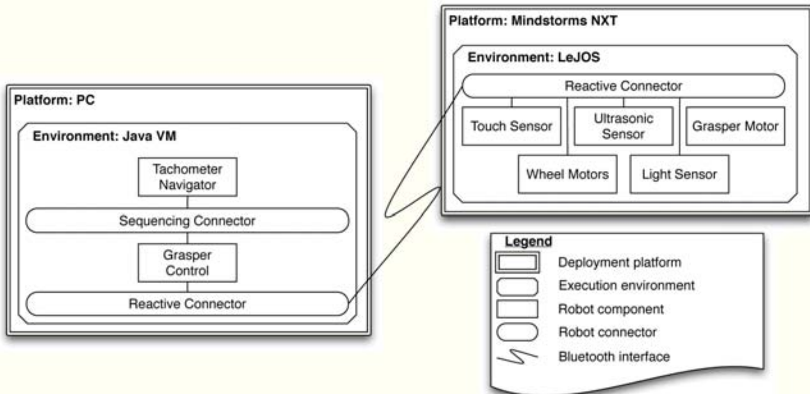

Fig. 6. The architecture ofthe ARCHIE robot, showing the robot's layered architecture distributed among two platforms and environments

ultrasonic sensor providing motion detection as well as range-finding in the front arc of the robot.

The software architecture of ARCHIE is built in the RAS style, and appears in Fig. 6. Once more, we abandon the traditional methods of building robot architectures in this domain, and construct ARCHIE in a component-based manner. The additional complication to integrating information providers and information consumers, in this domain, is the lack of the NXT brick's on-board processing power. As a result, ARCHIE adopts a tele-operation design: the bulk of the processing is performed on a PC running the PBAAM infrastructure, while the MINDSTORMS NXT brick is responsible for executing commands sent to its actuators and transmitting sensor data over the robot's Bluetooth connection. The deployment can be seen in Fig. 6, where the Bluetooth connection is indicated by the curved connection connecting the multi-platform architecture. This divide between the adaptive system and the facilities governing adaptation is not an essential challenge to developing robots using this approach, but was a solution driven by the limited resources of the NXT brick, which prevented us from deploying the PBAAM toolset on-board.

While limiting the range of the robot to that of the Bluetooth connection (roughly 10 meters), the solution was more than adequate for us to demonstrate self-adaptive behavior in our lab. We adopted the LEJOS ICOMMAND API (version 0.6) – a JAVA implementation of an NXT Bluetooth interface – and we replaced the default firmware of the MINDSTORMS platform with the LEJOS NXJ firmware update (version 0.4)[4].

With this architectural configuration, ARCHIE travels to a pre-defined location in our lab and grasps objects (in this case, small balls) if they are there. If it finds the object at the indicated location, it delivers it to its starting location. Navigation is implemented by the *Tachometer Navigator* component in the

[4] http://lejos.sourceforge.net/

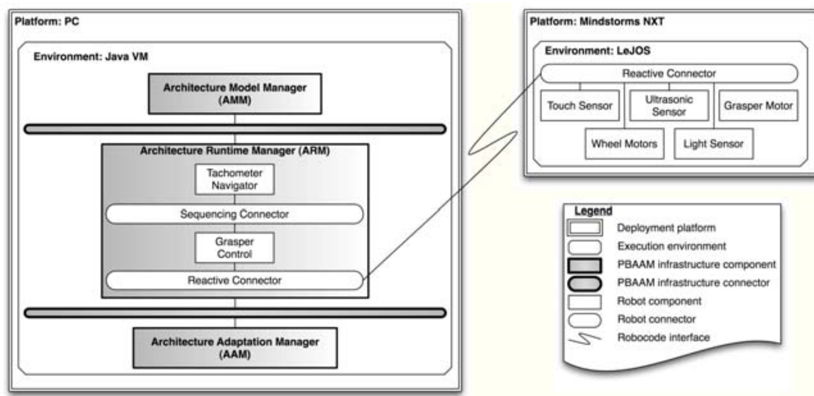

Fig. 7. The PBAAM-managed architecture of the ARCHIE robot, showing how the basic architecture is managed by the PBAAM tools in order to enable self-adaptive behavior

sequencing layer, which keeps track of the robot's location based on tachometer information from its motors. The *Grasper Control* component of the reactive layer provides control for the robot's grasping arm, and basic robot functionality resides in the skill layer.

Self-Adaptive MINDSTORMS**.** The need for adaptation in the ARCHIE robot is naturally motivated by the nature of the architecture itself. Due to driver incompatibilities between the Bluetooth libraries used by the NXT and the PC we used for development (running Mac OS X 10.4), the connection is unreliable and exhibits intermitted data loss and errors. Since the *Tachometer Navigator* component relies on reports from the robot's motors in order to calculate positioning data, corrupted information means that position information can be grossly mistaken, which makes navigation impossible. With this failure (currently self-diagnosed by *Tachometer Navigator* through a determination of whether data conforms to a reasonable envelope), the robot can no longer correctly navigate nor find its way back to the starting position.

One way to address this failure is through architectural self-adaptation. To this end, we developed a self-adaptive version of ARCHIE by managing the architecture with the PBAAM toolset: this augmented architecture appears in Fig. 7. We use PBAAM's adaptation management tools to deploy an adaptation policy that restores navigational support to the robot when *Tachometer Navigator* fails. The policy definition, elided for brevity, follows:

```
<AdaptationPolicy id="ReplaceNavigation">
  <ObservationList>
    <StringObservation>
      (navigation_report {failure ==true})
    </StringObservation>
```

```
  </ObservationList>
  <ResponseList>
    <RemoveComponentResponse>
      <RemoveComponent>
        Tachometer Navigator
      </RemoveComponent>
    </RemoveComponentResponse>
    <AddComponentResponse>
      <AddComponentIdentifier>
        Ultrasonic Navigator
      </AddComponentIdentifier>
      ...
    </AddComponentResponse>
  </ResponseList>
</AdaptationPolicy>
```

The new *Ultrasonic Navigator* component maintains no state information – which is the reason it is inserted into a lower RAS layer – but simply locates a wall using the robot's ultrasonic sensor and continues to follow the lab's walls until it locates the starting location that it detects using the light sensor to measure the reflectivity of the floor (the starting location is wall-adjacent and is more reflective than the remainder of the lab's floor). ARCHIE's components are also designed in the state-based way used for the ROBOCODE platform to account for synchronicity assumptions in the underlying development framework.

As with the ROBOCODE case study, our goal was to establish the feasibility of applying architecture-based self-adaptation techniques to a domain in which they had not previously been demonstrated, namely autonomous mobile robotic systems. So, while the architecture and behavior of ARCHIE are simple, they nevertheless demonstrate a practical self-adaptive solution to a practical problem as well as a successful application of our tools and techniques in this domain. And, despite the simplicity of the platform, we are confident our feasibility claim is valid due to the number of difficulties and challenges our MINDSTORMS robot shares with more complex robotic systems, such as the:

- necessity of integrating data from multiple sensors (sensor fusion) to determine courses of action;
- demands on timely actions in response to sensor information so that the robot's actions are current and actions are not performed too late, and;
- unreliability of sensor information and communication channels that the robot must account for in its control system.

These are just some examples of the kinds of difficulties shared by both real-world robotic systems and the types of MINDSTORMS robots we are developing. These systems are real, mobile, unreliable and resource constrained platforms, which is why we feel justified in the validity of a case-study using this platform.

5 Discussion

This section explores some of the interesting questions and trade-offs we encountered and offers some of our insights into the cross-section of self-adaptive architectures and robotic systems.

5.1 Architectural Mismatch

One of the difficulties we faced in our case studies was the architectural mismatch between the robotic frameworks we were working with and the assumptions of an architecture- and component-based approach. The PBAAM development framework was designed and built to support the development of component-based systems that use asynchronous events for communication and have no assumptions of shared state between them. These high-level design principles are critical in enabling the degree of modularity and decoupling that our work in developing self-adaptive systems relies on.

Robotic systems, on the other hand, tend to be constructed with architectural assumptions about synchronicity and strict temporal ordering of operations in mind. Many robotic libraries, for example, define interfaces to actuators and sensors that operate in a blocking manner: control is not returned until they have completed execution, which is particularly problematic in the case of long-term actions such as movement. This allows systems to be designed with great ease as long as they're constructed in a monolithic manner: it is quite easy to develop behaviors by chaining together a sequence of operations when these operations are invoked sequentially from a single unit of code. It is significantly more challenging, however, to compose these behaviors when fine-grained actions are distributed among many independent and potentially distributed components.

This mismatch between the fundamental design decisions of these domains was challenging to overcome, and required care in component design to account for the lack of synchronicity and the lack of guarantees about event ordering. The benefit, of course, from investing in this effort is the higher degree of modularity and enablement of runtime adaptation that this approach to building systems supports.

The conclusion from this necessity for more effort, of course, is not to dismiss the integration of architecture-based adaptation with robotics, but to recognize when the trade-off becomes worthwhile. Unless there is a driving need for self-adaptive behavior – and therefore the necessity to support a great deal of modularity and runtime evolvability – the effort to build robotic systems in a component-based manner and according to the RAS style while having to bridge this mismatch is not worth the benefits. It is interesting to note that a great deal of effort is currently being invested by the robotics community toward supporting the development of component-based robots and integrating software engineering technologies in their construction, as exemplified by the IEEE RAS TC-SOFT[5].

[5] http://robotics.unibg.it/tcsoft/

5.2 Platform Selection

Our goal of examining the integration of architecture-based adaptation with robotics – rather than the development of industrial-strength robots that is work best left to domain experts – guided our selection of the Mindstorms system for experimentation: The platform compares favorably with other commercially available platforms and development tool sets such as the iRobot or K-Team platforms in terms of flexibility as well as cost. While there is a clear loss of sturdiness compared to these pre-manufactured robots, the Mindstorms platform provides a degree of flexibility that other packages can't match: a new type of physical structure is only a matter of a few hours of (fun) re-assembly. Most important is the degree of broad availability and appeal of the LEGO kits: the kit supports both Windows and Macintosh platforms in a variety of programming languages, is supported by a large and dedicated community, has sold millions of units throughout the years, and seems to quickly capture the imagination and attention of those exposed to it. As we are interested in both disseminating our research tools and techniques as well as teaching self-adaptive architectures to university students, the Mindstorms platform is an ideal and affordable choice for such needs.

6 Future Work

There are a number of directions we plan on pursuing in continuing our work in the intersection of robotic systems and self-adaptive architectures. In the long term, the most important aspects of robotic construction which must be addressed involve examining, understanding, and improving the scalability and reliability of these architectures. This is both an issue of learning more about the implications of using architecture-based techniques in the robotics domain as well as a matter of refining our notations and tools: we expect, for example, to refine our conceptualization of constraints and express them in terms of behavior as opposed to structure.

We also plan on further strengthening our feasibility claims by continuing to build more sophisticated and complex robotic architectures. One such avenue of development involves the construction of mixed deployment, distributed architectures in which the more computationally intensive elements of a robotic architecture remain on a PC platform and communicate through the wireless connection, but where more efficient elements are deployed on the robot platform itself. We hope that these hybrid architectures will ease the difficulty of bridging the architectural mismatch we discussed in the previous section, and significantly improve the performance and stability of our robots.

7 Conclusion

The work presented in this paper is an exploration into the application of architecture-based techniques to the robotics domain in order to support the

development of self-adaptive robotic systems. One of the critical challenges of this effort is supporting the dynamic modification of adaptive behaviors during runtime.

Our previous work in architecture-based self-adaptive systems has been focused on supporting exactly this capability through the use of adaptation policies that are decoupled from the architectures they relate to; these policies are independently managed and the tools we have developed support their runtime modification. We therefore applied this work to the robotics domain by performing two case studies: The first focused on developing a self-adaptive robot for the Robocode robotic combat simulator, while the second involved the development of a self-adaptive Mindstorms NXT robot.

We believe that our efforts were successful in multiple ways. Most importantly, our case studies establish the feasibility of applying an architecture-based approach to self-adaptive robotic software by demonstrating practical solutions to practical robotic problems. Furthermore, our work also contributes an initial understanding of the difficulties involved in transitioning our particular approach – and others component-based approaches like it – to the robotics domain due to the architectural mismatch between the assumptions of architecture-based approaches and the actual practice of robotic system development. Finally, we continue to realize that the policy language of our approach is adequate for specifying adaptive behavior in a variety of settings.

The field of robotics is a rich application domain for software engineering research which provides dividends for both communities. Robotic software development greatly benefits from the application of software engineering research to the challenges of the domain, and software engineering researchers gain rigorous test settings and realistic scenarios for their work in a domain that stresses issues such as safety, reliability, and adaptation completeness which are more relaxed in other settings. We plan to continue to examine the intersection of robotic control systems and self-adaptive architectures, particularly with the Mindstorms platform as an easily accessible and inexpensive experimental testbed.

Acknowledgments. The authors would like to thank Eric Dashofy for his work on ArchStudio and André van der Hoek for his contributions to the conceptualization of our approach. This work sponsored in part by NSF Grants CCF-0430066 and IIS-0205724.

References

1. Georgas, J.C., Taylor, R.N.: Towards a Knowledge-Based Approach to Architectural Adaptation Management. In: Proceedings of ACM SIGSOFT Workshop on Self-Managed Systems (WOSS 2004), Newport Beach, CA (October 2004)
2. Nilsson, N.J.: Principles of Artificial Intelligence. Tioga Publishing Company (1980)
3. Brooks, R.A.: A Robust Layered Control System for a Mobile Robot. IEEE Journal of Robotics and Automation 2(1), 14–23 (1986)

4. Firby, R.J.: Adaptive Execution in Complex Dynamic Worlds. PhD thesis, Yale University (1990)
5. Georgas, J.C., Taylor, R.N.: An Architectural Style Perspective on Dynamic Robotic Architectures. In: Proceedings of the IEEE Second International Workshop on Software Development and Integration in Robotics (SDIR 2007), Rome, Italy (April 2007)
6. Taylor, R.N., Medvidovic, N., Anderson, K.M., James, E., Whitehead, J., Robbins, J.E., Nies, K.A., Oreizy, P., Dubrow, D.L.: A Component- and Message-Based Architectural Style for GUI Software. IEEE Transactions on Software Engineering 22(6), 390–406 (1996)
7. Wermelinger, M., Lopes, A., Fiadeiro, J.L.: A Graph Based Architectural (Re)configuration Language. In: ESEC/FSE-9: Proceedings of the 8th European Software Engineering Conference held jointly with 9th ACM SIGSOFT International Symposium on Foundations of Software Engineering, pp. 21–32. ACM Press, New York (2001)
8. Georgiadis, I., Magee, J., Kramer, J.: Self-Organising Software Architectures for Distributed Systems. In: WOSS 2002: Proceedings of the First Workshop on Self-Healing Systems, pp. 33–38. ACM Press, New York (2002)
9. Garlan, D., Cheng, S.W., Huang, A.C., Schmerl, B., Steenkiste, P.: Rainbow: Architecture-Based Self Adaptation with Reusable Infrastructure. IEEE Computer 37(10) (2004)
10. Oreizy, P., Gorlick, M.M., Taylor, R.N., Heimbigner, D., Johnson, G., Medvidovic, N., Quilici, A., Rosenblum, D.S., Wolf, A.L.: An Architecture-based Approach to Self-Adaptive Software. IEEE Intelligent Systems 14(3), 54–62 (1999)
11. Kim, D., Park, S., Jin, Y., Chang, H., Park, Y.S., Ko, I.Y., Lee, K., Lee, J., Park, Y.C., Lee, S.: SHAGE: a Framework for Self-Managed Robot Software. In: SEAMS 2006: Proceedings of the 2006 International Workshop on Self-Adaptation and Self-Managing Systems, pp. 79–85 (2006)
12. Kramer, J., Magee, J.: Self-Managed Systems: An Architectural Challenge. In: Future of Software Engineering (FOSE 2007), pp. 259–268 (2007)
13. Georgas, J.C.: Supporting Architecture- and Policy-Based Self-Adaptive Software Systems. PhD thesis, University of California, Irvine (2008)
14. Dashofy, E.M., Hoek, A.v.d., Taylor, R.N.: A Comprehensive Approach for the Development of Modular Software Architecture Description Languages. ACM Transactions on Software Engineering and Methodology (TOSEM) 14(2), 199–245 (2005)
15. Harel, D.: Statecharts: A visual formalism for complex systems. Science of Computer Programming 8(3), 231–274 (1987)
16. Hill, E.F.: Jess in Action: Java Rule-Based Systems. Manning Publications Co., Greenwich (2003)
17. Georgas, J.C., van der Hoek, A., Taylor, R.N.: Architectural runtime configuration management in support of dependanble self-adaptive software. In: Proceedings of ACM SIGSOFT Workshop on Architecting Dependable Systems (WADS 2005), St. Louis, MO (May 2005)