Policy-Based Self-Adaptive Architectures: A Feasibility Study in the Robotics Domain

John C. Georgas and Richard N. Taylor
Institute for Software Research
University of California, Irvine
Irvine, CA 92697-3425 USA
{jgeorgas, taylor}@ics.uci.edu

ABSTRACT

Robotics is a challenging domain which sometimes exhibits a clear need for self-adaptive capabilities, as such functionality offers the potential for robots to account for their unstable and unpredictable deployment domains. This paper focuses on a feasibility study in applying a policy- and architecture-based approach to the development of self-adaptive robotic systems. We describe 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 a discussion of the architectural issues we encountered, and further evaluates our general-purpose approach.

Categories and Subject Descriptors

D.2.11 [**Software Engineering**]: Software Architectures—*languages, domain-specific architectures*

General Terms

Design, Experimentation

1. INTRODUCTION

One of the 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. The variety of goals that this capability can be applied toward has given rise to a number of sub-types within this class of systems: self-healing or self-optimizing, for example. These are self-adaptive systems where the adaptive behavior is targeted toward a specific goal to the exclusion of others. In this paper, we will use the more general term *self-adaptive* to inclusively refer to the entire class of systems no matter what the goal of the adaptation may be.

In addition to self-adaptive software, we are also interested in robotic systems. These systems are amalgams of software and hardware which are highly resource constrained, commonly deployed in environments out of reach of human operators, and commonly required to perform functions to which there can be little interruption. Robotic systems tend to be highly reactive in nature and depend on a great deal of interaction with their environment.

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. A motivating example can be found in the failure of the star tracker in the Deep Space 1 (DS1) mission [13]. Launched by NASA in 1998, DS1 was an experimental mission intended to evaluate a number of high-risk technologies. The most severe of difficulties the mission faced was the complete failure of its star tracker which was critical to its navigation. In response, the DS1 team launched into a 4 month long effort to use alternate instruments in place of the lost star tracker: the effort was successful, but culminated in the replacement of almost the entire craft's flight software. Despite this remarkable achievement, the fact that so much of the flight software had to be replaced is telling: this unforeseen but necessary adaptation was simply not well supported.

This example illustrates that 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 design and construction. Only then can adaptive behavior be integrated into the system. For our own research efforts, the fact that the construction of the system must support adaptation implies that modularity is one of the fundamental qualities which allows adaptation to take place in a fine-grained manner, rather than adapting a system 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 robotic self-adaptive 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 [5]. Our goals with the work described here in this paper are to:

- establish the feasibility of an integration between our research into policy- and architecture-based self-adaptive systems and the 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 toward these goals and our experiences in striving to meet them. We describe the two case studies we performed in developing self-adaptive robotic systems: we begin with our work in the ROBOCODE system – a robotic combat simulator and development framework – and continue by discussing the construction of an autonomous MINDSTORMS NXT robot (neither of these domains previously considered – much less supported – self-adaptive capabilities).

The key contributions of our work in the 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-adaptation capabilities in our case-study domains;
- uncovering an important architectural mismatch between architecture-based adaptation and current practice in the robotics domain; and
- demonstrating that our policy language, though simple, is adequate for expressing robotic adaptations.

2. BACKGROUND AND RELATED WORK

This section begins with a discussion of representative robotic architectures with particular emphasis on their support for runtime change, and also discusses 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) [11]. In SPA, robot control is accomplished through the *sense* component which gathers information from sensors, the *plan* component which maintains an internal world model used to decide on the robot's actions, and the *act* component which is responsible for executing actions.

As robotics systems grew, however, it became obvious that SPA architectures scaled poorly: SUBSUMPTION [1] 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 [3]. These hybrid architectures separate robotic systems into three 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) [6]. 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.

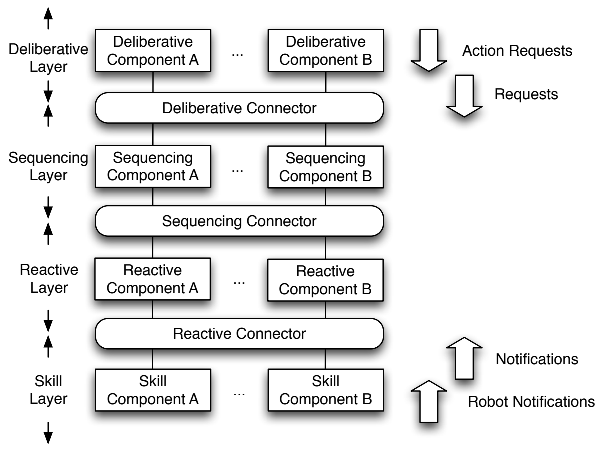

Figure 1: An illustration of the RAS architectural style, showing the style's layers and event types.

This lack of support for the architectural qualities which promote ease of runtime change is the motivation for the development of the RAS architectural style, also described in [6]. The style combines insights from event-based architectural styles such as C2 [14] and the SUBSUMPTION and 3L robotic architectures and is aimed at supporting the development of robotic architectures which are modular and incrementally evolvable while fostering component reuse. Figure 1 outlines the style, which is:

- component-based, with no shared memory;
- explicitly-layered into *skill*, *reactive*, *sequencing*, and *deliberative* layers[1] with components belonging to layers based on their complexity and state maintenance;
- 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.

The robotic systems we build in our case studies are built in 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

Other researchers in the software architecture community are investigating the development of self-adaptive systems using architectural models as the core abstraction, as our approach does. There is great variety, however, in these approaches, mainly with respect to the goals adaptations are

[1]While the RAS style uses similar layer names as 3L architectures, there are differences between the two; the reader is referred to [6] for further details.

intended to achieve and the methods through which adaptive behavior is specified.

Some work takes a formal approach to the specification of architectures and the artifacts governing adaptation. The work based on COMMUNITY [15], for example, models architectural models as abstract graphs while the approach based on the DARWIN [7] 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 [4] 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 [12], 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 robotic platforms, the SHAGE framework [9] supports the definition of adaptive strategies managed by a controlling infrastructure and focuses on adaptations which 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 [10]. The approach described here trades formal specifications of system behaviors for 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 relaying our robotic case-study experiences, this section presents in high-level terms the approach we used in developing these systems. As this paper is primarily focused on the application of our approach to robotic architectures, however, we keep this discussion minimal (earlier descriptions of the ideas presented here have appeared in [5]).

The core of our policy-based approach to architectural adaptation management (PBAAM)[2] appears in Figure 2 and shows the most important software components and documents involved (rectangles represent tools, rounded rectangles indicate documents, and numbered arrows show information flow and activity ordering).

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 parts are managed at runtime by elements of the PBAAM infrastructure: the *Architectural Model Manager* (AMM), the *Architectural Adaptation Manager* (AAM), and the *Architecture Runtime Manager* (ARM) respectively. Other elements not discussed here include support for the recording and visualization of adaptations as they take place and for the specification and resolution of constraints intended to preserve core system capabilities.

[2]In earlier publications, we referred to our work as knowledge-based architectural adaptation management (KBAAM). While we still use some technologies from the knowledge-base community, we feel the term policy-based is more appropriately descriptive.

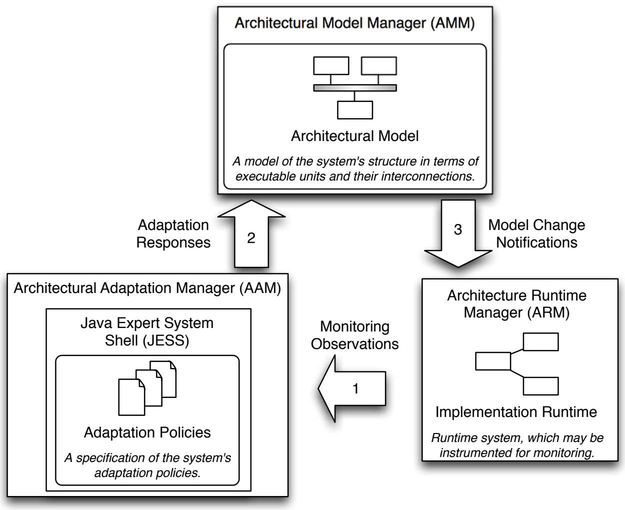
Architectural Model Manager (AMM)
Architectural Model
A model of the system's structure in terms of executable units and their interconnections.
Adaptation Responses
2
3
Model Change Notifications
Architectural Adaptation Manager (AAM)
Java Expert System Shell (JESS)
Adaptation Policies
A specification of the system's adaptation policies.
Monitoring Observations
1
Architecture Runtime Manager (ARM)
Implementation Runtime
Runtime system, which may be instrumented for monitoring.

Figure 2: A high-level view of our approach to architecture-based self-adaptive systems.

3.1 Adaptation Policy Specification

One of the fundamental abstractions in our approach is the set of adaptation policies: these policies are encapsulations of the system's reactive *adaptive behavior* and indicate what actions 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 software architectures and, in essence, reduce to additions and removals of architectural elements.

We specify the structure of adaptation policies using a XADL 2.0 schema [2] which extends the core schemas of the ADL and lays out policy structure. It is important to note that – in core XADL fashion – this policy schema is extensible and can be customized to fit the needs of specific projects. One of the extensions currently under development defines a set of *constraints*: restrictions on actions that are not allowed to take place. This class of policies will act as warden of the architectural model and prevent undesirable modifications from taking place.

3.2 Architectural Adaptation Management

The AAM is the element responsible for the runtime management of the adaptation policies specified using our schema. 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 is responsible for updating the set of active policies to reflect these changes.

We chose to implement the AAM by adopting an expert system approach to the runtime management of policies. In a very straightforward manner, policies can be translated to executable *condition-action* rules and then managed using an expert system shell. More specifically, we adopt the Java Expert System Shell (JESS) [8] 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 framework supporting runtime evolution predating 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.

3.3 Activity Flow

Referring to the activity flows indicated in Figure 2, the adaptation process begins when observations about the running system are collected and transmitted to the AAM (the 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 communicated to the AMM which maintains the system's architectural model (indicated by the activity flow labeled 2). Finally, the ARM is notified of any changes enacted on the architectural model (activity flow labeled as 3) and ensures that these changes are reflected on the executing system. This cyclic flow of information – continually executed – provides a reactive loop for self-adaptive behavior.

4. CASE STUDIES

This section of the paper presents our main focus: specific details about the two feasibility case studies we performed in integrating the self-adaptive capabilities described in the previous section in robotic systems. For each of these, we will discuss the systems we developed and our experiences with them as well as call out some of the difficulties we encountered and lessons we learned.

4.1 Robocode

Our first study was performed using the Robocode[3] 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. The system is supported by an active community of both developers and users, and supports a number of associated tournaments and competitions.

4.1.1 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 can move, use its radar to detect other robots, and use its gun to fire at opponents. The constraint of most importance for each robot is the amount of energy it has remaining (all robots begin a battle with the same level of energy): Energy is lost by being hit by bullets or colliding with other robots or walls. Energy is also 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

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

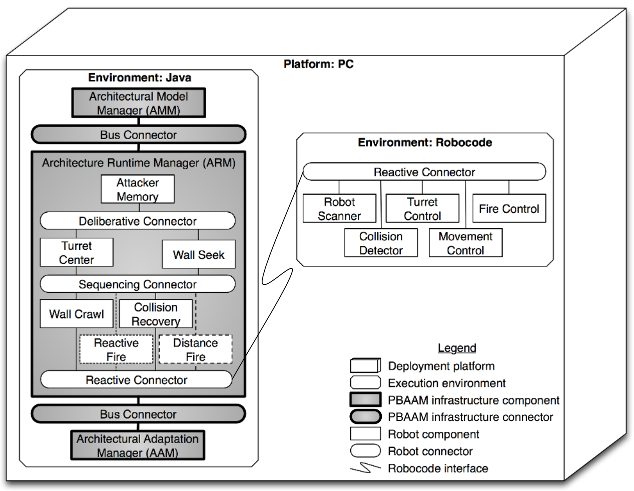

Figure 3: The architecture of the ArchWall robot for the Robocode simulator.

to preserve its own energy by both wisely firing 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 using these fundamental capabilities 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 normal development for the simulator, a robot is programmed and compiled as a single static unit which is then executed by the battle simulator without support or consideration for runtime adaptation.

4.1.2 Self-Adaptive Robocode

To begin exploring self-adaptation in this context, we first developed an integration between the Robocode framework and the PBAAM infrastructure. We constructed a special-purpose interface that conformed to the Robocode API (in addition to a number of modifications to the Robocode framework itself mainly aimed at loosening class management restrictions); this interface acts as a bridge between the two environments. PBAAM requests for robot actions such as movement or firing, are translated into Robocode API calls, while notifications of battlefield events from the simulator, such as collisions or bullet fire, are translated into architectural events and transmitted throughout the system's components. Implementing this integration allowed us to develop Robocode robots which, instead of being single units of executable code, are component-based architectures expressed in the xADL ADL and managed by the PBAAM framework.

A specific example is the ArchWall robot, the architecture of which is illustrated in Figure 3. The robot is built in the RAS style (the robot's architecture appears in unshaded elements, while the shaded elements are the PBAAM managing infrastructure; the curved connection indicates the bridge between PBAAM and Robocode events). Building on the fundamental capabilities of the simulator, ArchWall initially moves by seeking a wall and following it, targeting the center of the battlefield, and firing at any opponent robot it detects. Each behavior is captured by an independent component. This initial set of behaviors is sufficient for the ArchWall robot to compete in battles: in our testing experiences, the robot tends to rank between positions four and six in a field with ten opponents selected from the set of sample Robocode robots which are distributed with the simulator.

The robot is additionally composed of the AMM, ARM, and AAM components which allow us to augment the behavior of ArchWall with a number of adaptation policies which modify the robot's behavior as the conditions of the battlefield change. One policy, for example, states (in an abridged form for brevity):

```
<AdaptationPolicy id="ReplaceFiring">
  <StringObservation>
    (energy_report {energy < 60})
  </StringObservation>
  <RemoveComponentResponse>
    Reactive Fire
  </RemoveComponentResponse>
  <AddComponentResponse>
    <ComponentIdentifier>
      Distance Fire
    </ComponentIdentifier>
    ...
  </AddComponentResponse>
  </AdaptationPolicy>
```

This policy replaces the firing strategy used by ArchWall when the energy of the robot drops below the indicated level 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* – Figure 3 illustrates this change with the component and links being added indicated by dotted lines and the component being removed by dashed lines. 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.

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 the 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 robot would: 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 with sequencing of behaviors achieved by explicitly ordering instructions in the source code of a robot. 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 – which is not necessary in other applications we've applied our approach to – that maintains information about the state of the interactions it is engaged in with components to which it has dependencies.

4.2 Mindstorms NXT

We performed our second case study using the LEGO Mindstorms NXT development kit[4]. Released in the summer of 2006, this is another in LEGO's line of kits to support easily accessible and affordable robotics development.

4.2.1 Mindstorms NXT Background

Each Mindstorms kit is comprised of Technic pieces which are used to build the structure of robots, servo motors with built-in rotation sensors, and a variety of sensors. The basic commercial Mindstorms kit includes an ultrasonic sensor, a light sensor, a sound sensor, and a touch sensor; color, accelerometer, and compass sensors are also available. Computer control for the sensors and motors is provided by a NXT brick: each brick supports enough ports to accommodate up to three motor and four sensor connections and also supports a USB port and a Bluetooth wireless connection. These kits are extremely affordable 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 programming environment. The NXT-G environment supports a visual language where programs are composed by "bricks" which support basic programming constructs. While basic, this programming environment is sufficient for most casual users. 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 runtime change.

4.2.2 Self-Adaptive Mindstorms

Building on the work described in the previous section, we continued by developing an integration of our tools with the Mindstorms platform. The initial software development challenge was – once more – the integration between information sources and information consumers. The additional complication, however, was the lack of processing power of the NXT brick and its inability to support our existing self-adaptive architecture toolset. To address these challenges, we adopted a tele-operation design: the bulk of the processing is performed on a PC running the PBAAM infrastructure, while the NXT brick is only

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

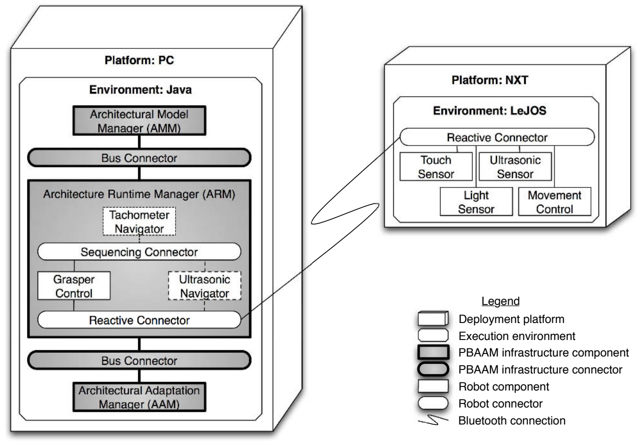

Figure 4: The architecture of the Archie robot for the Mindstorms platform.

responsible for executing commands sent to its actuators and gathering sensor data over the Bluetooth connection; this deployment can be seen in Figure 4. 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)[5].

The robotic platform we constructed – dubbed Archie – is a modification of a basic three-wheeled Mindstorms design: a picture of the robot in our lab can be seen in Figure 5. 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: by using opposite directions of rotation in each of the motors, the robot is capable of turning in place, therefore simplifying navigation. 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 reflectivity of the surface the robot is on, an ultrasonic sensor providing motion detection as well as range-finding in the frontal arc of the robot, and a compass sensor.

Archie's architecture is built in the RAS style and seen in Figure 4: the robot's architecture appears in unshaded elements while the infrastructure architecture is shaded, and the curved connection represents the Bluetooth interface between the PC and NXT brick. The robot 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 Navigation* component, which keeps track of the robot's location based on tachometer information from its motors. This component, however, tends to fail often – mostly because the high volume of Bluetooth communication that Archie currently implements tends to overflow the communication buffers, therefore losing positioning data. When this failure is detected (in our current implementation, the component

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

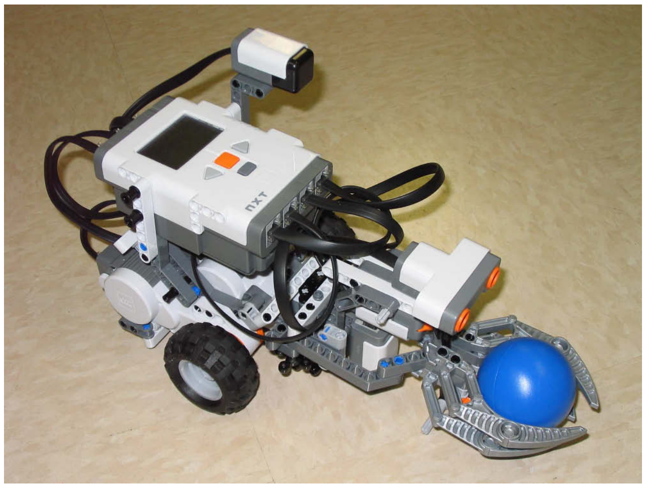

Figure 5: A picture of the basic Archie Mindstorms robot: a three-wheeled design with a grasping arm and a number of mounted sensors along with the NXT processor brick.

self-diagnoses by determining whether data conforms to a reasonable envelope), the *ReplaceNavigation* adaptation policy removes the *Tachometer Navigation* component (an operation indicated in Figure 4 by having the component and its links appear in dotted lines) and replaces it with the *Ultrasonic Navigator* component (an operation indicated in dashed lines in Figure 4). The policy is of the same type discussed in the previous section – the omitted parts of the policy deal with the specification of which connectors the newly added component is connected to, and are elided for brevity:

```
<AdaptationPolicy id="ReplaceNavigation">
  <StringObservation>
      (navigation_report {failure == true})
  </StringObservation>
  <RemoveComponentResponse>
    Tachometer Navigator
  </RemoveComponentResponse>
  <AddComponentResponse>
    <ComponentIdentifier>
      Ultrasonic Navigator
    </ComponentIdentifier>
    ...
  </AddComponentResponse>
</AdaptationPolicy>
```

The new *Ultrasonic Navigator* component maintains no state information – which is the reason it belongs to a lower 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 which it detects using the light sensor to measure the reflectivity of the floor (the starting location is adjacent to a wall and is more reflective than the remainder of the lab's floor). Archie's components were also designed in the state-based way used for the Robocode platform to account for synchronicity assumptions in the underlying development frameworks.

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 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 both real-world robotic systems and the types of MINDSTORMS robots – which are real, mobile, unreliable and resource constrained platforms – we are developing face in common, and why we feel justified in a feasibility case-study using this platform.

5. DISCUSSION

This section explores and offers our insights on some of the interesting issues and trade-offs we encountered in planning and performing the previously described work.

5.1 Architectural Mismatch

One of the difficulties we faced in our feasibility 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 communicate through the exchange of asynchronous events and have no assumptions of shared state. These design principles are critical in enabling the high level 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 and do not return control until they have completed execution. 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 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 need 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 (such as use of the RAS style) and 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[6].

5.2 Platform Selection

Our goal of examining the feasibility of developing self-adaptive robotic architectures – rather than the development of industrial-strength robots (work best left to roboticists) – 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 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 time 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 constraint policies and use them to ensure that our robots always maintain a core set of functions which errant adaptations may possibly corrupt.

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

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

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 feasibility of applying architecture-based techniques to the robotics domain to support the development of self-adaptive robotic systems. One of the critical challenges is supporting the dynamic modification of adaptive behaviors during system 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 regards. Most importantly, our case studies confirm the feasibility of applying an architecture-based approach to self-adaptation in the robotics domain. Furthermore, our work also contributes an initial understanding of the difficulties involved in transitioning our particular approach – and others 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 despite its simplicity.

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 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.

8. ACKNOWLEDGMENTS

The authors would like to thank Eric Dashofy for his work developing 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.

9. REFERENCES

[1] R. A. Brooks. A Robust Layered Control System for a Mobile Robot. *IEEE Journal of Robotics and Automation*, 2(1):14–23, 1986.

[2] E. M. Dashofy, A. v. d. Hoek, and R. N. Taylor. A Comprehensive Approach for the Development of Modular Software Architecture Description Languages. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 14(2):199–245, April 2005.

[3] R. J. Firby. *Adaptive Execution in Complex Dynamic Worlds*. PhD thesis, Yale University, 1990.

[4] D. Garlan, S.-W. Cheng, A.-C. Huang, B. Schmerl, and P. Steenkiste. Rainbow: Architecture-Based Self Adaptation with Reusable Infrastructure. *IEEE Computer*, 37(10), 2004.

[5] J. C. Georgas and R. N. Taylor. 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.

[6] J. C. Georgas and R. N. Taylor. 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.

[7] I. Georgiadis, J. Magee, and J. Kramer. Self-Organising Software Architectures for Distributed Systems. In *WOSS '02: Proceedings of the First Workshop on Self-Healing Systems*, pages 33–38, New York, NY, USA, 2002. ACM Press.

[8] E. F. Hill. *Jess in Action: Java Rule-Based Systems*. Manning Publications Co., Greenwich, CT, USA, 2003.

[9] D. Kim, S. Park, Y. Jin, H. Chang, Y.-S. Park, I.-Y. Ko, K. Lee, J. Lee, Y.-C. Park, and S. Lee. SHAGE: a Framework for Self-Managed Robot Software. In *SEAMS '06: Proceedings of the 2006 International Workshop on Self-Adaptation and Self-Managing Systems*, pages 79–85, 2006.

[10] J. Kramer and J. Magee. Self-Managed Systems: An Architectural Challenge. In *Future of Software Engineering (FOSE '07)*, pages 259–268, 2007.

[11] N. J. Nilsson. *Principles of Artificial Intelligence*. Tioga Publishing Company, 1980.

[12] P. Oreizy, M. M. Gorlick, R. N. Taylor, D. Heimbigner, G. Johnson, N. Medvidovic, A. Quilici, D. S. Rosenblum, and A. L. Wolf. An Architecture-based Approach to Self-Adaptive Software. *IEEE Intelligent Systems*, 14(3):54–62, May-June 1999.

[13] M. D. Rayman and P. Varghese. The Deep Space 1 Extended Mission. *Acta Astronautica*, 5(12):693–705, 2001.

[14] 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*, 22(6):390–406, 1996.

[15] M. Wermelinger, A. Lopes, and J. L. Fiadeiro. 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*, pages 21–32, New York, NY, USA, 2001. ACM Press.