Architecture-Driven Self-Adaptation and Self-Management in Robotics Systems

George Edwards, Joshua Garcia, Hossein Tajalli, Daniel Popescu, Nenad Medvidovic, Gaurav Sukhatme

Computer Science Department
University of Southern California
Los Angeles, CA 90089 USA
{gedwards, joshuaga, tajalli, dpopescu, neno, gaurav}@usc.edu

Brad Petrus
Bosch Research & Tech. Center
Two NorthShore Center, Suite 320
Pittsburgh, PA 15212, U.S.A.
Brad.Petrus@us.bosch.com

Abstract

We describe an architecture-centric design and implementation approach for building self-adapting and self-managing robotics systems. The basis of our approach is the concept of meta-level components, which facilitate adaptation and management of application-level components. Our approach applies two key enhancements to the traditional usage of meta-level components: (1) we utilize three distinct, specialized meta-level components for the three fundamental activities of a robotics system: sensing, computation, and control, and (2) we allow meta-level components to themselves be monitored, managed and adapted by other (higher layer) meta-level components. In this way, our approach flexibly supports adaptive layered architectures of arbitrary depth, the specification of arbitrary system adaptation policies, and the provision of intelligent facilities for constructing adaptation plans on-the-fly. We showcase our approach using a team of autonomous mobile robots that engage in a leader-follower scenario and experience a wide variety of failures, activating distinct recovery mechanisms.

1 Introduction

A large and rapidly growing class of today’s software systems cannot be brought down for upgrades. Instead, they need to adapt themselves to software and hardware failures, changes in their computing and physical environments, and the arrival of new or upgraded services. Such systems increasingly tend to be long-lived, decentralized, heterogeneous, mobile, and ubiquitous. Their adaptation therefore cannot be effectively directed from and controlled by a central source; instead, these systems are expected to exhibit some degree of self-awareness and self-adaptivity. Mobile robotics systems are a class of system that benefits significantly from self-adaptation, and have thus generated growing research interest. Within the software engineering research community alone, several groups have begun studying this problem — Sykes et al. [12] and Georgas and Taylor [4] being two representative examples. The observation made by both of the above research groups is that software architecture can serve as the linchpin of robotics software design and adaptation. Both groups have proposed reference architectures for the domain of robotics software that support self-adaptation and different adaptation policies for the resulting systems.

These approaches demonstrate the benefits of affording software architecture a central role in constructing self-adaptive and self-managing robotics systems. Moreover, both of these proposed reference architectures, and, more generally, other architectures for self-adaptive robotics systems, are fundamentally based on the adaptive layered style. In this paper, we describe a design and implementation approach for adaptive layered systems.

In the adaptive layered style, system components are explicitly layered, but the relationship between the components of each layer is very different than the traditional relationship between components in a layered system. Normally, layering implies that components at a given layer invoke the services of components at the layer below (and are prohibited from invoking the services of components at the layers above). In adaptive layered systems, on the other hand, components at a given layer monitor, manage, and adapt components at the layer below. In this way, each successive layer may:

- ensure the correct operation of the layer below (fault tolerance),
- modify the functionality at the layer below (dynamic update),
add new capabilities to the layer below (*resource discovery*),

relocate components to improve quality-of-service characteristics (*redeployment*), or

implement other self-adaptive mechanisms.

Note that the use of adaptive layering does not preclude the use of traditional layering *within an adaptive layer*. That is, the components that comprise a layer in an adaptive layered system may themselves be organized into layers, such that each layer invokes the services of the layer below.

Our approach allows the capture of domain-specific architectures for adaptive robotics, including those proposed by Sykes and Georgas. We encapsulate facilities for monitoring, managing, and adapting application software components into *meta-level components*. Moreover, because meta-level components can themselves be seamlessly introduced, reconfigured, replaced, or removed during system run-time, they can hierarchically or recursively monitor, manage, and adapt other meta-level components. This allows:

- the creation of adaptive layered architectures of arbitrary depth,
- the specification of arbitrary system adaptation policies, and
- the provision of intelligent facilities for constructing adaptation plans on-the-fly.

Our approach is enabled by our architectural middleware infrastructure [10], which supports a close correspondence between a system’s architectural design and its implementation. An adaptive robotic system’s architecture is both designed and implemented in terms of (1) application components that control the robot and (2) meta-level components that implement fault-tolerance, dynamic update, resource discovery, redeployment, and other services. As we will explain, the crucial difference between application components and meta-level components is that meta-level components have direct access to the run-time instantiations of a system’s architectural design constructs. In turn, this allows for direct support of self-management and self-adaptation policies at the implementation level.

2 Application Scenario

We illustrate our approach’s support for self-awareness and self-adaptation using software architectural principles through the following robotics scenario, developed as part of a collaboration between USC and Bosch RTC. A set of software components are dynamically deployed to a platoon of mobile robots at startup. These components implement both (1) sensor processing and reactive control algorithms that execute basic behaviors, such as object following and obstacle avoidance, and (2) system monitoring, management, and adaptation services that execute goal-oriented plans, such as collaborative environment exploration. Once configured with scenario-specific behaviors and plans, these robots assemble autonomously and follow a leader robot along a given path (see Figure 1). The robots collect and process data from on-board sensors and stationary sensor nodes positioned throughout the environment. The robots encounter several base stations along their path that allow a robot to dock, recharge its battery, and download and upload data (see Figure 2). The robots collaborate by exchanging sensor data and reporting their status to each other.

![Figure 1. A platoon of three robots is following a leader and passing by a base station, shown in the top-left corner.](image1)

![Figure 2. The middle robot leaves the platoon to dock with the base station.](image2)
During the execution of the scenario, the robots may need to adapt to:

- changes in the physical environment, such as the loss of a power source,
- changes in the underlying hardware and network infrastructure, such as a device failure,
- changes in the available software resources, such as an update to a behavior control component, or
- changes in the scenario goals, such as a switch from object following to wall mapping.

These changes trigger the execution of adaptation policies. The robots run on-board analyses in order to track their own status. When necessary, the robots alter their behavior by reconfiguring their existing set of components, exchanging components with other robots, and retrieving software updates from base stations. For example, a robot whose battery is depleted may minimize its remote communication by suspending the execution of network-intensive components; a robot using a camera-based following mechanism may switch to infrared sensor-based following in the event of a camera failure; a robot with a buggy software component can apply an application-level patch to improve performance.

PDAs and laptops permit human intervention to modify robot behaviors and adaptation policies. The robots, sensors, PDAs, laptops and base stations represent a distributed, decentralized, and heterogeneous computing environment that must adapt to failures, new services, service updates, and changes in monitored system properties such as energy consumption. We will demonstrate how our approach deals with these self-aware and self-adaptive issues using a small, shared set of software architectural principles.

3 Approach

In this section, we describe our approach to supporting adaptive layered robotics systems. The basis of our approach is the concept of meta-level components. Meta-level components facilitate adaptation and management of application-level components. Numerous examples of successful use of meta-level components exist in the adaptive systems research literature [3, 5, 6]. Our approach applies two key enhancements to the traditional usage of meta-level components:

1. We utilize three distinct, specialized meta-level components for the three fundamental activities of a robotics system: sensing, computation, and control.

2. We allow meta-level components to themselves be monitored, managed and adapted by other (higher layer) meta-level components.

We begin in Section 3.1 with a summary of the key architectural concepts that are the basis of the approach described in this paper. We then define the three specialized types of meta-level components in Section 3.2. In Section 3.3, we explain how meta-level components can be layered to produce hierarchical self-management and self-adaptation. Then, in Section 4, we discuss how enhancement of an architectural middleware technology enabled us to faithfully implement the design concepts discussed here.

3.1 Architectural Foundations

We treat a robotics system as a collection of components that engage in event-based interaction via explicit communication facilities, i.e., connectors. The use of first-class architectural constructs facilitates software adaptation by providing target entities to which atomic adaptation operations, such as component lifecycle operations and link creation, can be applied. Components and connectors are self-contained, decoupled, and independently deployable, which allows them to migrate among hosts, be updated or replaced at run-time, or flexibly reconfigure their links with other system elements.

Software adaptation is triggered autonomically according to adaptation policies and executed autonomously according to adaptation plans. Adaptation policies specify when an adaptation is necessary (e.g., ComponentA fails to respond to a service request) and what the outcome of an adaptation should be (e.g., ComponentA is replaced with ComponentB). Adaptation plans specify the sequence of adaptation operations necessary to achieve that outcome (e.g., record ComponentA’s context, uninitialize ComponentA, download ComponentB’s implementation, instantiate ComponentB, set ComponentB’s context).

Adaptation and management services are implemented using meta-level components. Meta-level components instantiate, configure, monitor, and deploy application-level components according to application-specific policies and goals. The key difference between meta-level and application-level components is that meta-level components are architecturally aware. That is, meta-level components have access to an internal representation of the system’s current application architecture (e.g., the set of active components, the links among components and connectors, etc.) and are allowed to manipulate that architecture in specific ways. Therefore, architectural awareness allows meta-level components to implement advanced services by manipulating application-level components. For example, meta-level components implement fault tolerance by replicating appli-
cation components and dynamic update by replacing components.

### 3.2 Meta-Level Component Types

Most mobile robotics systems are reactive: they perform actions based on their perception of the environment. This reactive nature suggests three distinct activities: sensing the environment, computing response plans, and executing appropriate control actions. Similarly, adaptation mechanisms in software are reactive: they detect, for example, the availability of new services or the occurrence of a failure and subsequently plan and execute changes to the running system. Consequently, researchers have proposed approaches to adaptive software in general, and adaptive robotics software in particular, that embrace the sense-compute-control paradigm [2]. Our approach uses design constructs that correspond directly to the primary activities of reactive systems. We define the following types of meta-level components:

- **Collector** components gather data about the components at the layer below. This could include both behavioral data, such as the utilization of resources and the frequency of service invocations, and structural data, such as the topology of component instances and the assignment of components to hardware hosts. The nature of the data collected is application- and scenario-specific.

- **Analyzer** components interpret the data produced by Collectors to evaluate adaptation policies and create adaptation plans. Adaptation policies commonly specify high-level goals, such as the satisfaction of quality-of-service (QoS) requirements, such as continuous availability or predictable performance. Consequently, the construction of adaptation plans that achieve these goals can be quite complex, and Analyzer components may implement sophisticated algorithms to do so.

- **Admin** components control and manipulate components at the layer below according to the adaptation plans produced by Analyzers. This control is exercised via management interfaces that implement adaptation operations.

All three types of meta-level components are architecturally aware. This architectural awareness is achieved by providing each meta-level component with access to a runtime representation of the system’s architecture. Through this representation, meta-level components can access all the sub-elements (components, connectors, ports, etc.) of the architecture of the layer below. Collector components extract information about the Architecture and its sub-elements and package it in a form usable by an Analyzer. Analyzers check whether information about the architecture meets conditions that warrant adaptation. Analyzers are also responsible for transmitting component and connector implementations to other (possibly distributed) Analyzers if needed. If conditions for adaptation are met, an Analyzer produces a set of adaptation instructions. Admins then make modifications to the architecture and its sub-elements according to these instructions.

### 3.3 Meta-Level Component Layering

The ability of meta-level components to manage other meta-level components allows for a layered or hierarchical management structure of arbitrary depth. The recursive nature of our meta-level component management model plays an important role in robotics systems, which are frequently based on multi-tiered control and decision making architectures.

Figure 3 shows a conceptual depiction of a system implemented using our approach that demonstrates the power of multilayer adaptive capabilities. At the robotics layer, application components implement a basic robot behavior: object following. The Camera Driver takes raw streaming video from a camera hardware device. The Object Follower interprets the video to locate the relative position of the object being followed. The Motor Actuator directs the robot motor to move in the required direction.

At the meta layer, meta-level components implement a basic adaptive behavior: reactive fault recovery. Reactive fault recovery detects when faults occur and then takes a mitigating action. The Fault Detector monitors the camera and reports failures to a Replacement Selector. The Replacement Selector determines the best replacement component for the camera based on adaptation policies. The Replacement Selector notifies the Replacement Deployer of the new component that is needed, and the Replacement De-

![Figure 3. Notional depiction of an example adaptive layered architecture.](image-url)
ployer instantiates the component.

At the meta2 layer, meta-level components implement an advanced adaptive behavior: fault tolerance strategy selection. Fault tolerance strategy selection permits the use of different fault tolerance mechanisms for different circumstances. For example, if reactive fault recovery is resulting in unacceptable down-time during replacement selection and deployment, the active replication fault tolerance strategy can be used instead. Active replication ensures that multiple synchronized copies of a component are always available in case one fails. The Fault Recovery Monitor records the amount of system down-time after each fault. The Fault Recovery Evaluator determines whether the system availability is acceptable. If reactive fault recovery is not meeting availability requirements, the Fault Tolerance Deployer instantiates a new set of meta-layer components that implement active replication.

4 Implementation Support

To implement the design approach described in the previous section, we extended an architectural middleware platform. Architectural middleware provides implementation-level constructs for architectural design abstractions, such as components, connectors, interfaces, and styles, which helps to ensure architectural conformance. Specifically, we utilized the Prism-MW architectural middleware platform [10] to implement our specialized meta-layer components and enforce the constraints of the adaptive layered style. In this section, we summarize the key features of Prism-MW and show how we leveraged, and where necessary, adapted and enhanced those features.

4.1 Architectural Constructs

Prism-MW is a lightweight middleware platform implemented in Java and C++, which we have used to develop numerous mobile and embedded systems [10, 11]. Figure 4 shows the high-level design of Prism-MW. Each Architecture object serves as a run-time container for a topology of Component and Connector objects. Components implement application services, while Connectors implement interaction-oriented services. Components and Connectors communicate by exchanging Events via Ports. The other constructs shown in Figure 4 do not play a critical role in the approach described here, but a description of them may be found in [10].

Architectures can enforce stylistic constraints, such as interaction patterns and deployment requirements, to produce desirable system qualities. Moreover, integration of multiple styles into a single robotics system is readily supported by Prism-MW. This is an advantage because complex applications commonly employ several architectural styles [13].

Prism-MW’s architectural constructs are the core of a complete layered middleware solution [11], depicted in Figure 5, that (1) provides the necessary low-level abstractions for interfacing with the underlying operating system, network, and hardware; (2) incorporates different robotics libraries as appropriate; (3) implements software systems in terms of design-level constructs, and (4) enables an extensible collection of advanced services, such as resource discovery and fault tolerance.

We previously implemented advanced services in Prism-MW (shown at the top layer of Figure 5) using meta-level components. While these services provided support for
generalized monitoring, management, and adaptation, they did not support the following key capabilities:

1. **Multi-adaptive layered systems.** The meta-level components that implement Prism-MW’s advanced services cannot themselves be managed and adapted.

2. **Domain-specific adaptation mechanisms for robotics systems.** While the components that implement Prism-MW’s advanced services are customizable to some extent, robotics architectures frequently demand domain-specific algorithms, protocols, constraints, and patterns that can only be provided by application developers.

Our proposed technique generalizes the notion of meta-level components to allow implementation of multi-adaptive layered systems and robotics-specific meta-level components.

### 4.2 Implementation Support for Meta-Level Component Types

Recall that meta-level components gain architectural awareness through access to an internal representation of the system’s current architecture. This representation exists natively in Prism-MW applications in the form of Architecture objects. Because Collectors are only permitted to gather information, they hold a read-only reference to the Architecture, which prevents them from modifying the architecture. Analyzers also hold a read-only reference to the Architecture, which allows them to retrieve component implementations and configurations for transmission to other distributed Analyzers. Admins are the only meta-level components that hold a writable reference to the Architecture.

Collectors gather all information about the Architecture necessary to evaluate adaptation policies and create adaptation plans. Application developers utilize Collectors by implementing a set of Monitors, each of which inherits from the AbstractMonitor base class. Each application-specific Monitor is associated with a specific Brick instance (Brick is the abstract base class of all Prism-MW architectural constructs; recall Figure 4). The Monitor defines methods that capture data of interest and transmit it back to a Collector. Collectors receive, store, aggregate, filter, and package the data for transmission to Analyzers.

Analyzers evaluate adaptation policies based on monitoring data and construct adaptation plans. Application developers implement domain-specific algorithms to evaluate policies and construct plans. In a distributed system, multiple Analyzers will likely need to exchange information about a system’s state and collaboratively determine system-wide adaptation plans. Analyzers are responsible for requesting and retrieving the binary implementation of a software component when an adaptation plan requires a component to be transmitted across hosts.

Figure 6 shows an application-specific energy policy of a generic Analyzer implemented in Java. This Analyzer requires each application-specific policy to implement the policy interface depicted at the top of Figure 6. After receiving data from a Collector component, the Analyzer checks whether a policy was triggered and accordingly initiates the computation of an adaptation plan. For example, the EnergyPolicy depicted in Figure 6 evaluates the remaining energy of the robot after receiving the remaining energy measurement in a data object of type energy type. If the remaining energy is below a user-specified threshold, the policy evaluates to true. As a consequence, the Analyzer requests the adaptation plan from the EnergyPolicy, which initiates the calculation of an optimal deployment using the technique developed in our previous work [9].

```
public interface Policy {
    public boolean isTriggered(CollectorData data);
    public Plan computeAdaptationPlan(IArchitecture arch);
}

public class EnergyPolicy implements Policy {
    private int THRESHOLD = 500;
    public boolean isTriggered(CollectorData data) {
        if (data.getType().equals(AnalyzerBrick.getEnergyType()) &&
            currentEnergy = (Integer) data.getValue());
            return currentEnergy < THRESHOLD;
        } else {
            return false;
        }
    }

    public Plan computeAdaptationPlan(IArchitecture arch) {
        return DeploymentCalculator.computeOptimizedDeployment(
            OptimizationGoal.SAVENERGY, arch);
    }
}
```

**Figure 6. Policy of an example Analyzer implementation**

Admins execute the adaptation plans created by Analyzers and utilize the IArchitecture, IComponent, IConnector, and IPort interfaces to manipulate the corresponding Prism-MW architectural constructs. For example, the IArchitecture interface implements methods to add and remove components, while the IComponent interface implements methods to add and remove ports. To destroy a component, the Admin component disconnects all ports of the component and removes the component from the architecture.

Figure 7 shows an excerpt of a Prism-MW Admin component in Java that can execute adaptation plans. As one of its internal methods, this Admin component contains a method to remove a component from the architecture of the layer below. In Prism-MW, a component can communicate with other components and connectors only if one of its ports is **welded** to a port of another component or connector. Therefore, before the Admin component can remove
a component, it needs to *unweld* all ports of the component. After unwelding all ports, it removes the component from the architecture as depicted in Figure 7.

```java
public class Admin extends AbstractImplementation {
    private IArchitecture arch;

    private void removeComponent(Component comp) {
        List<IPort> ports = comp.getPorts();
        for (IPort p : ports) {
            arch.unweld(p, p.getMutualPort());
        }
        arch.remove(comp);
    }

    public void execute(Plan adaptationPlan) {
        // ...
    }
}
```

Figure 7. Excerpt from an `Admin` implementation

### 4.3 Implementation Support for Meta-Level Component Layering

In distributed applications developed using Prism-MW, each host in the system executes one or more `Architecture` objects, each of which encapsulates a set of components. In adaptive layered applications, the components on each host are separated into distinct Prism-MW `Architectures` corresponding to each layer. The application-layer `Architecture` contains application components, while the meta-level `Architectures` of a robot contain `Collectors`, `Analyzers`, and `Admins`. Each meta-level component possesses a reference to the `Architecture` that implements the layer below.

Separating layers into distinct `Architectures` provides the following benefits:

1. Components can be easily prevented from interacting with components in other layers, except through prescribed mechanisms, thus preserving architectural integrity.

2. Meta-level components are only given access to the monitoring and control interfaces of components at the layer directly below, helping to ensure that system adaptation only occurs in predefined, controlled ways.

3. The components within each layer are insulated from failures and unaware of adaptations taking place in other layers.

4. Different architectural styles can easily be used within each layer. For example, components within the application layer might use a traditional layered architecture, while components in the meta layer interact via peer-to-peer mechanisms.

Interactions among components in an adaptive layered system are not restricted by physical host boundaries. First, a meta-level component can communicate with other meta-level components on other hosts that reside at the same logical layer. Interactions among meta-level components within a layer are analogous to the interactions in the sense-compute-control paradigm. For example, a `Collector` on one host can report monitoring data to an `Analyzer` on another host, provided both are contained in the same layer. Second, a meta-level component can adapt components on other hosts that are contained in the layer below the meta-level component’s layer. For example, a meta2-layer component can adapt meta-layer components residing on other hosts. We demonstrate examples of both these interaction types in the application scenario presented in Section 5.

### 5 Self-Adaptation and Self-Management in the Application Scenario

In Section 2, we introduced a mobile robotics application scenario in which a group of robots assemble and follow a leader along a path. While the robots are moving, various autonomous adaptations take place due to changes in the execution environment, scenario goals, and other factors. In this section, we describe how we implemented this scenario using the approach described above. In particular, we highlight the various types of adaptations performed by the robots and illustrate how those adaptations are achieved by collaborations of meta-level components.

A simplified depiction of the architectures of two robots and a base station is shown in Figure 8. Each robot executes a set of application components that control robot behavior (shown at the robotics layer in Figure 8). The set of application components deployed to each robot depends on the robot’s role. A `Robot Leader` executes a `Line Follower` component that detects a guiding line on the ground with a set of infrared sensors and moves the robot accordingly. `Robot Followers` use cameras or infrared sensors to locate the robot directly ahead in the convoy and follow it.

Each robot also executes meta-level components (shown at the meta layer in Figure 8). The set of meta-level components also depends (to some extent) on the robot’s position in the convoy. The `Robot Follower` in Figure 8 executes a `Failure Collector` which monitors a `Camera Driver` for hardware failures. The `Robot Leader` executes a `Status Collector`, which monitors the operation of a `Line Follower`. In addition, the `Robot Followers` utilize a distributed meta2 layer to dynamically discover new versions of meta-level components. A `Version Collector` and `Updater` are deployed on the `Robot Followers`, while a `Version Analyzer` resides on the base station.

The robots execute the following adaptations autonomously:
Replacing failed components.

- Updating components if new versions are dynamically discovered at a base station.
- Reconfiguring components if a robot fails.
- Redeploying components if battery power is being drained more rapidly than expected.

Due to space limitations, we only describe the first two adaptations from the above list in detail.

Replacing failed components. The Failure Collector installs specialized Failure Monitors on application components that control robot sensors. Each Failure Monitor is an implementation of Prism-MW’s Abstract Monitor that detects any exception thrown by the monitored component. If a sensor fails, the software component controlling the sensor throws an exception which is detected by the Failure Monitor and transmitted to the Failure Collector. The Failure Collector reports failures to the Substitution Analyzer, which determines whether the component must be replaced based on the type and severity of the exception thrown. If replacement is required, the Substitution Analyzer selects a suitable replacement from a repository of component implementations. For example, if the robot camera fails, the Camera Driver throws an exception. Substitution Analyzer creates an adaptation plan to instantiate a new Infrared Driver. The robot uses the new Infrared Driver to follow the robot ahead of it using an infrared sensor.

Updating components at a base station. Each component is tagged with a version number to facilitate component updates. The meta2-level Version Collector on each Robot Follower tracks the versions of meta-level components and sends the data to a distribution connector depicted in Figure 8. The distribution connector buffers the version data while waiting for a potential connection to a base station. At the same time, a distribution connector on the base station scans the wireless network environment for bypassing robots. If a connection can be established, the versions of the robot’s components are sent to the base station’s Version Analyzer. In our scenario, the Version Analyzer discovers that a new version of the Failure Collector is available and consequently sends the new implementation of the Failure Collector to the connected robot’s Updater. After the new component is sent to the Updater, it replaces the old Failure Collector with the newly received Failure Collector.

6 Support for Adaptive Robotics Reference Architectures

In this section, we demonstrate how the approach can be mapped to two adaptation reference architectures: Georgas and Taylor’s policy-based approach to architectural adaptation management (PBAAM) [4] and Sykes et al.’s three-layer reference model [12].

6.1 Support for the PBAAM

Georgas and Taylor [4] present the policy-based approach to architectural adaptation management (PBAAM), which includes an architectural style, reference architecture, and implementation infrastructure for adaptive robotics systems. PBAAM systems include both robotics components, which implement robot behaviors, and infrastructure components, which implement adaptation and management.

The PBAAM infrastructure comprises three components that are responsible for architectural adaptation: the Architectural Model Manager (AAM), the Architectural Adaptation Manager (AAM), and the Architecture Runtime Manager (ARM). The AAM manages and evaluates adaptation policies based on collected event notifications, which may result in adaptation requests being sent to the AMM. As a result of adaptation requests from the AAM, the AMM applies changes to an architectural model of the system and
fewer types of data or does so at a lower frequency.

If a meta2-layer was consuming energy too rapidly, the meta2-layer could be used to monitor the energy consumption of the AAM since the ARM modifies the architecture of the robotics layer.

In the PBAAM approach, there is no inherent support for monitoring or adapting the AAM, AMM or the ARM; however, quality of service issues may warrant self-adaptation of these components. For example, one goal of a robotic system may be to minimize energy consumption (an issue that emerged repeatedly in the development of our scenario). A meta2-layer Collector could be used to monitor the energy consumption of the AAM Collector depicted in Figure 9. If a meta2-layer Analyzer decided that the AAM Collector was consuming energy too rapidly, the meta2-layer Analyzer could request that a meta2-layer Admin replace the current AAM Collector with one that monitors fewer types of data or does so at a lower frequency.

6.2 Support for the Three-Layer Reference Model

Like the PBAAM style, Sykes’s three-layer reference model is a layered style that supports self-adaptation. The bottom-most layer of this model is the component control layer, where components that implement robotics behavior reside. At the middle change management layer, a plan interpreter adapts components in the component control layer based on plans. At the topmost goal management layer, a model-based planner creates the plans based on high-level goals and provides those plans to the change management layer. Figure 10 shows how the three-layer reference model maps to the adaptive layered style. The component control layer would be implemented using application components, while the change management and goal management layers would be implemented using meta-level components.

The change management layer could be implemented using one of each type of meta-level component. A Collector component could detect failure events at the component control layer. The plan interpreter would be composed of an Analyzer and Admin. The plan interpreter’s Analyzer would receive data from the Collector and would use this data along with a plan to determine appropriate component configurations of the component control layer. Once an appropriate configuration is computed, the plan interpreter’s Admin can execute the plan. A single Analyzer could implement the goal management layer because its only responsibilities are to create plans that achieve high-level goals. In particular, the Analyzer would be used to implement the model-based planner, which contains the robotic system’s goals and transforms them into plans that are passed to the plan interpreter. It should be noted that the goal management and change management layers belong to the same meta layer in our approach because the goal management layer does not monitor, analyze, or adapt components in the change management layer; it only provides plans to the change management layer. Furthermore, the change management layer is aware of the goal management layer because the plan interpreter explicitly requests new plans from the model-based planner.

Sykes’s three-layer reference model could benefit from a meta2 layer that ensures a system’s quality-of-service by monitoring, analyzing and adapting the goal management and change management layers. For example, the decision to use reactive plans in Sykes’s model may adversely affect performance. A reactive plan contains a set of actions leading to a desired goal state for every possible state from
which the goal can be reached. Consequently, the plan remains valid even if a robot reaches an unexpected state after performing an action. Depending on the model of the environment, determining these plans could become intractable, resulting in excessive computation times. To avoid computational intractability, an additional meta-layer Collector could monitor the computation time of the model-based planner and the size of the resulting plans. In cases where the model-based planner consumes more time than desired, a meta-layer Analyzer could instruct a meta-layer Admin to replace the model-based planner with a more efficient planner, such as a planner that computes a set of actions for a subset of possible current states that lead to the goal state, rather than for every possible state that leads to a goal state.

7 Related Work

Besides the work in [12, 4], other approaches have utilized meta-level components and an explicit software architectural focus to achieve self-adaptation. The SHAGE (Self-Healing, Adaptive, and Growing Software) framework [8] consists of two parts that enable self-management. The first part consists of seven meta-level components, while the second part contains repository servers. The components in these two parts work together to monitor a system and reconfigure its architecture. The Rainbow framework [1] comprises an architecture layer that contains meta-level components and a system layer that contains meta-level and application components. This framework utilizes reusable descriptions of architectural styles to allow reuse of adaptation knowledge across different systems. Unlike our approach, neither SHAGE nor Rainbow use meta-layering or rely on architectural middleware to ensure the consistency between a system’s architecture and its implementation. As part of its autonomic computing initiative, IBM proposed autonomic managers that manage other components using the MAPE (monitor, analyze, plan and execute) control loop [7]. Our meta-level components realize similar functionality as the MAPE control loop. However, we are able to achieve higher decoupling and composability of meta-level components by utilizing event-based communication, architectural abstractions and architectural constraints.

8 Conclusion

We described an architectural approach to building self-aware and self-adaptive robotic systems. The core of our approach includes two key enhancements to the previous usage of meta-level components — domain-specific meta-level component types and higher-order meta-layering. Our approach allows roboticists to realize adaptive layered robotic systems that conform to their individual style or reference architecture. On the other hand, meta-level layering helps to ensure qualities of service in a flexible and adaptable manner. We explained how these specific benefits can be achieved when our approach is used in conjunction with the PBAAM approach or Sykes’s three-layer reference model. A specific instance of our approach was described in the context of a mobile robotics application scenario that requires self-awareness and self-adaptation.

9 Acknowledgments

This material is based upon work sponsored by Bosch. The work was also sponsored by the National Science Foundation under Grant numbers ITR-0312780, SRS-0820170, CCR-0120778, and IIS-00755534. The authors wish to express their gratitude to John Lewis for his contributions to the project.

References