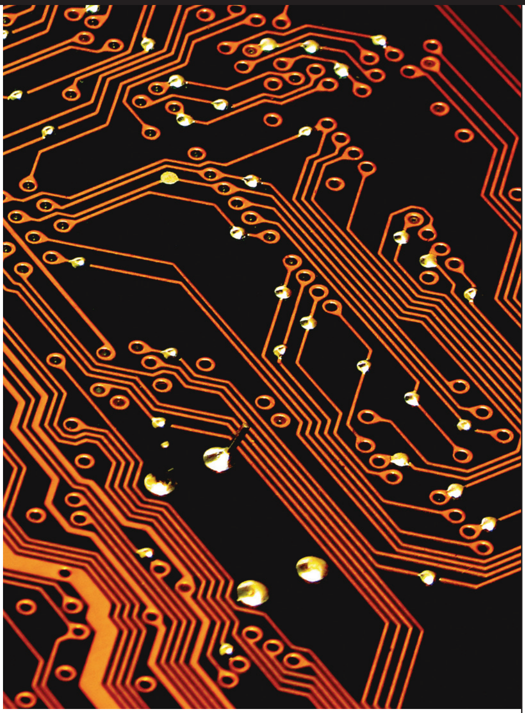
Engineering Heterogeneous Robotics Systems: A Software-Architecture-Based Approach



Nenad Medvidovic, Hossein Tajalli, Joshua Garcia, and Ivo Krka
University of Southern California
Yuriy Brun, *University of Washington*
George Edwards, *Blue Cell Software*

RoboPrism, a framework that supports software-architecture-based development of robotic systems, is accessible to nonexperts in robotics, deals effectively with heterogeneity in distributed and mobile robotics systems, and facilitates adaptation in complex, dynamic environments.

Robotics systems exhibit characteristics that argue for a software engineering focus, including

- a high degree of heterogeneity among constituent subsystems,
- strict operational requirements dictated by real-time interactions with the physical world, and
- system complexity that extends beyond a single engineer's ability to grasp.

In fact, developers have increasingly applied software engineering to robotics systems, as reflected in a recent special issue of *IEEE Robotics and Automation*[1] and in the formation of the *Journal of Software Engineering and Robotics*.[2]

Despite these developments, it is still common for an engineering team to develop the decision-making and control apparatus of a robotics system from scratch, only to discover that it is too difficult to separate this software from the rest of the system and reuse it. They are thus forced to develop a new system from scratch.

The use of common robotics software libraries, such as Player and CLARAty, only partially alleviates this problem. Although these libraries consist of robotics-specific middleware that provides a low-level robot framework and helps with specific advanced features such as distributed communication and code mobility, the existing solutions provide no guidance or support for faithfully preserving the design-time structure of robotics systems.

Furthermore, relying on a given library results in applications that aren't easily ported to robot platforms that do not already support the library. Likewise, engineers must devise solutions for dealing with requirements (such as the dynamic loading of components) that the chosen technology does not natively support.

Recent approaches have adopted an explicit software engineering perspective for building robotics systems, resulting in reusable design and implementation frameworks. However, these approaches tend to neglect critical software engineering issues, including

- exploration of the design space and of the effective software design solutions within that space, needed both for the initial system design and subsequent dynamic runtime adaptations;
- modeling the distributed software-intensive system that is deployed on a set of robot (and possibly traditional) platforms as opposed to modeling robotic algorithms;
- analysis of the system models for key properties before constructing and deploying the system and during dynamic adaptations;
- traceability of the design-time artifacts, such as components and connectors to implementation constructs; and
- support for heterogeneous development and deployment platforms.

Our approach aims to remedy these shortcomings. The basis of this work is software architecture,[3] a set of principal design decisions about a software-intensive system embodied in the system's components (operational entities that perform computation), connectors (entities that facilitate interaction and coordination among components), and configurations (assemblies of components and connectors into system-specific topologies). Our approach uses a robotics system's architectural basis to address the five problem areas.

ROBOTICS THROUGH THE PRISM OF SOFTWARE ARCHITECTURE

We propose a novel architectural style that supports guided exploration of design alternatives for a dynamically adaptive robotic system and uses a rigorous system modeling and analysis framework. It also uses implementation and deployment middleware with the explicit architecture traceability support that's necessary for heterogeneous settings. In the process, our work can make the development of robotics software more accessible to nonexperts in robotics, reduce the time and effort required to create and maintain robotics software, and improve the exchange of design solutions among robotics engineers.

Our approach to engineering robotics software adapts and applies three important software architecture concepts:[3]

- *architectural design abstractions*, enabling the creation of reusable, adaptive, and hierarchical components and systems;
- *architectural modeling and analysis*, allowing early, integrated, and continuous (re)evaluation of system behaviors and properties; and
- *architectural middleware*, permitting system implementation, deployment, monitoring, and runtime (self-)adaptation in highly dynamic, mobile, and heterogeneous environments.

The "Scenario for a Heterogeneous Robotics System" sidebar illustrates a typical multirobot application scenario that calls for a software engineering approach.

Design abstractions

A significant focus of software engineering research has been to codify design abstractions, which engineers use to represent and reason about complex systems at a high level. To this end, software architecture researchers have developed a canonical set of architectural design constructs: components, connectors, communication ports, interfaces (or services), events, and configurations. Furthermore, the uses of these constructs, prescribed via design heuristics or constraints, result in architectural styles (such as client-server or peer to peer) that are key design principles in software engineering. These constructs and principles have been highly useful in practice.

Implementation and deployment middleware provides the explicit architecture traceability support that's necessary for heterogeneous settings.

In traditional software, layering implies that components at a given layer invoke the services of components at the layer below. In contrast, components at a given layer in the *adaptive-layered* style monitor, manage, and adapt components at the layer below.[4]

The bottom layer in an adaptive-layered system is the application layer. Components in this layer implement functionality that achieves the application goals. An adaptive-layered architecture can have an arbitrary number of metalayers. Components in these layers—*collectors*, *analyzers*, and *admins*—are designed to handle operations that deal with monitoring, analysis, and adaptation. Collectors monitor lower-layer components, analyzers evaluate adaptation policies or plans based on monitored data, and admins perform adaptations. This approach ensures the separation of application-level from metalevel functionality, while allowing the system a high degree of autonomy.

We used an adaptive-layered style to realize different adaptive software systems.[4] In recent work, we leveraged this approach to design the Plan-Based Layered Architecture for Software Model-Driven Adaptation (PLASMA).[5] As Figure 1 shows, PLASMA employs three adaptive layers. Application-level components reside in the bottom layer. The middle layer—called the adaptation layer—monitors,

SCENARIO FOR A HETEROGENEOUS ROBOTICS SYSTEM

Consider the following scenario. A convoy of mobile robots must assemble autonomously and follow a leader robot along a prespecified path given as a series of waypoints, as Figure A shows. These robots collect and process data from onboard sensors and stationary sensor nodes deployed at various locations within the environment. As they traverse the path, the robots encounter several base stations, which can assess the robots' state, allow a robot to dock and recharge its battery, transfer data to and from the robot, and even release software updates to the robot.

Robots can collaborate by exchanging data as well as computational components (such as mobile code). They can also run onboard analyses to track their own health. For example, a robot with a depleted battery can minimize its remote communication or its onboard computation. Robots also need to adapt to changing environmental conditions, such as GPS signal loss or low visibility. Finally, the mission's goal might change at runtime from, for example, following the leader to mapping an unknown terrain. Overall, the robots, sensors, and base stations are a distributed, decentralized, and heterogeneous computing environment that must be capable of dynamic adaptation.

Such a scenario involves several technical challenges; some—such as developing effective algorithms to achieve the robot-following behavior—are clearly robotics specific. However, we argue that a majority of the remaining technical challenges fall within software engineering, and that software engineering provides the appropriate abstractions, methods, techniques, and tools to address such problems. This position has, in fact, been increasingly recognized by researchers who have tried to construct robotic systems using model-driven development and reusable domain-specific middleware platforms. In fact, several software engineering researchers have recently targeted their techniques toward dynamically adaptive robotics systems.[1-3]

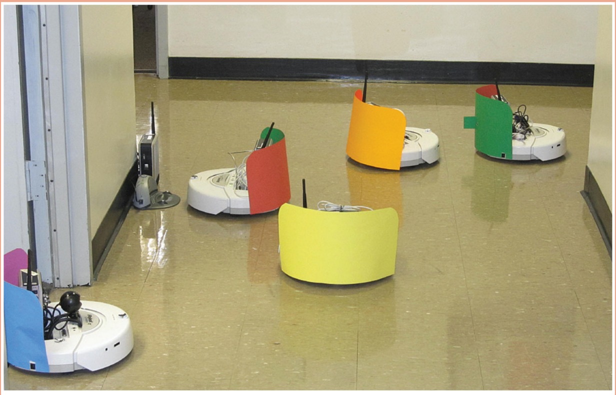


Figure A. Convoy of four robots following a leader. The red robot leaves the group to charge its battery at a base station.

References

1. J.C. Georgas and R.N. Taylor, "Policy-Based Self-Adaptive Architectures: A Feasibility Study in the Robotics Domain," *Proc. 2008 Int'l Workshop Software Eng. for Adaptive and Self-Managing Systems* (SEAMS 08), ACM Press, 2008, pp. 105-112.
2. D. Sykes et al., "From Goals to Components: A Combined Approach to Self-Management," *Proc. 2008 Int'l Workshop Software Eng. for Adaptive and Self-Managing Systems* (SEAMS 08), ACM Press, 2008, pp. 1-8.
3. H. Tajalli et al., "PLASMA: A Plan-Based Layered Architecture for Software Model-Driven Adaptation," *Proc. 25th IEEE/ACM Int'l Conf. Automated Software Eng.* (ASE 10), IEEE CS Press, 2010, pp. 467-476.

manages, and adapts components in the application layer. The top layer (planning) manages the adaptation layer and the generation of plans based on user-supplied goals and component specifications. The planning layer defines both the target architecture for the application layer (in the adaptation plan) and the actions for the application layer to carry out (in the application plan). The planning layer can respond to changing system requirements or operational environments by regenerating plans.

This three-layer architecture offers a high degree of autonomy and enforces a clear separation of concerns, whereby each layer provides a different form of adaptation capability. To use the adaptation capabilities, an architect must provide an architectural description of the system components and application goals. Alternatively, an architect can use only the application layer when developing a nonadaptive system.

Modeling and analysis

Our approach to engineering robotics software employs architectural models and analyses to inform and direct design decisions related to dynamic planning and adaptation.

First, we use architecture models specified in the Software Architecture Description and Evaluation Language (SADEL)[6] to automatically generate models needed for planning. A SADEL model specifying the functional interfaces of application components helps determine the actions available to the system and the effects of those actions on the environment. A SADEL model specifying the management interfaces of components (such as deploy, suspend, connect, and so on) helps determine how the adaptation layer can manipulate components to achieve a goal.

Second, we implemented tools that let engineers experiment with different

- system design decisions with respect to nonfunctional properties,
- policies for triggering dynamic replanning, and
- options for redeploying software components.

These tools are extensions to the Extensible Tool-Chain for Evaluation of Architectural Models (XTEAM) modeling and analysis toolset.[7] XTEAM provides an editing environment for specifying architecture models,

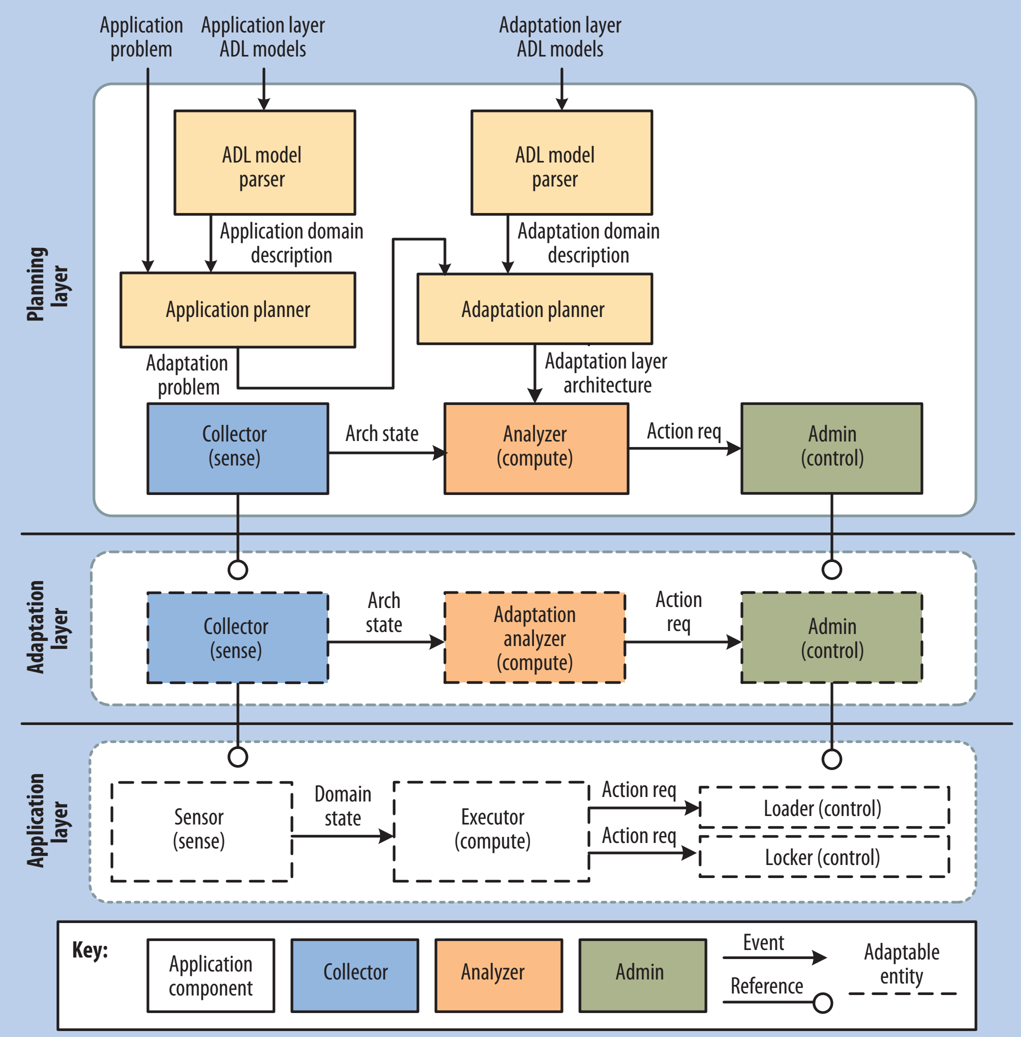


Figure 1. PLASMA adaptive-layered architecture.

a simulation generator for generating a discrete-event simulation of a system, and a code generator. Developers use the discrete-event simulation to observe the system's dynamic behavior under different operational conditions, assumptions, and constraints. XTEAM natively includes facilities for

- representing a software system architecture's structure and behavior in a formal model;
- attaching properties to model elements to capture parameters needed for various analyses; and
- analyzing simulations generated from models with respect to performance, reliability, and energy efficiency.

Engineers can use XTEAM to determine the impact of different replanning and redeployment strategies and to establish varying policies on system performance, reliability, and power efficiency.

Middleware

The existing robotics libraries and frameworks, although useful in many settings, are not always effective middleware platforms for developing robot-based software systems. This is particularly the case for systems distributed across multiple, heterogeneous platforms. Instead, we have developed and modified a layered middleware solution, RoboPrism, that alleviates these shortcomings by

- providing the necessary low-level abstractions for interfacing with the underlying operating system, network, and hardware;
- incorporating different robotics libraries, as appropriate;
- implementing software systems in terms of constructs (component, connector, event, port, style, and so on) that directly mirror architectural-design-level concepts;

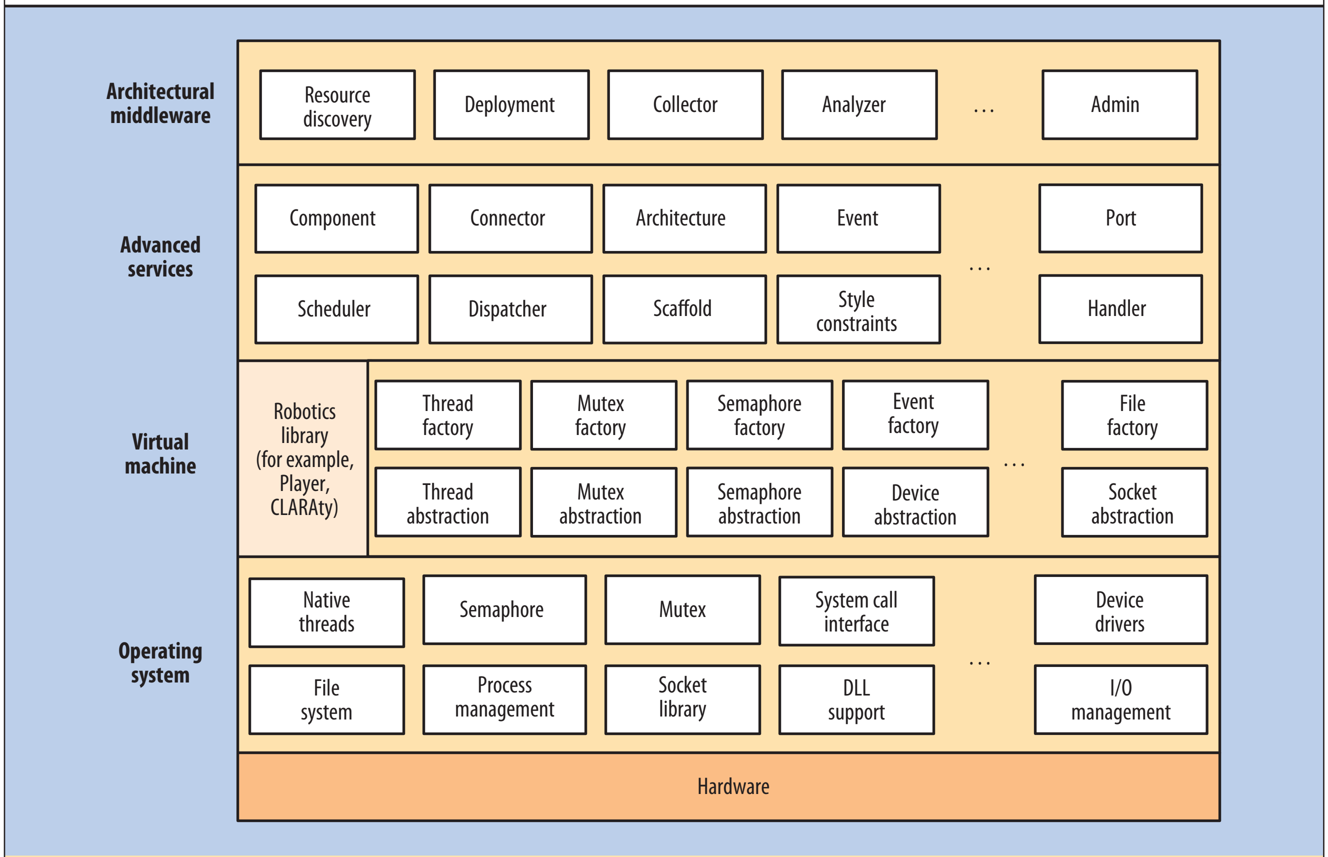


Figure 2. RoboPrism, a layered architectural middleware platform. Researchers have successfully integrated different robotics libraries within RoboPrism's virtual machine layer.

- providing an extensible collection of advanced, meta-level services, such as resource discovery or dynamic replanning and self-adaptation components;
- enabling the management and adaptation of the meta-level components and services to provide an adaptive layered system; and
- achieving the preceding without imposing unacceptable resource costs (in terms of memory, CPU, or network) on the resulting systems.

The resulting middleware is an adaptation of the Prism-MW middleware platform (http://sunset.usc.edu/~softarch/Prism) developed for embedded systems. Prism-MW focuses primarily on the architectural middleware layer in Figure 2. It relies on substrates (the virtual machine layer in Figure 2), such as the JVM for the Java version. Extensive measurements indicate that Prism-MW introduces less than 5 percent overhead for advanced services (deployment, mobility, disconnected operation, and monitoring),[8] which is acceptable for the architectural traceability that benefits analysis, maintenance, and reuse. Furthermore, providing these additional architectural abstractions does not impose a noticeable performance penalty.[8]

Using the RoboPrism platform yields several important benefits. First, systems designed according to RoboPrism insulate application software developers from reliance on the underlying robotics libraries, if any: the architectural middleware layer exports a single interface to application developers.

Second, RoboPrism allows the implementation of applications in multiple programming languages: the architecture construct bounds an address space, while specialized first-class connectors carry out interaction across address spaces.

Third, RoboPrism provides meta-architectures, which contain specialized metacomponents (admin, collector, and analyzer) that enable adaptive-layered applications. In such applications, components on each host are separated into distinct Prism-MW meta-architectures corresponding to each layer. Separating layers into distinct architectures enforces and guarantees the following architectural constraints:

- components in different layers only interact through prescribed mechanisms, and
- each meta-architecture only manages and adapts the architecture in the layer immediately below it.

Moreover, this separation insulates components in each layer from failures and adaptations in other layers, thus supporting a high degree of autonomy.

EXPERIENCE

We have investigated these concepts in the context of two scenarios using the iRobot Create platform. Our investigations also used the eBox3854 embedded PC running Linux, laptops running Windows XP and Vista, and Compaq iPAQ PDAs running Linux and Windows CE.

To dock iRobots and charge their batteries during scenario execution, we used the iRobot Home Bases. Creative Webcam and Logitech QuickCam cameras (controlled via the Java Media Framework, or JMF) provided visual information that enabled robot following, and SunSpot Java-based sensors provided the ability to manually control robot movement through accelerometers.

We relied on three options for controlling the iRobots: the Player and Create Open Interface libraries, both of which are in C, as well as our custom iRobot driver in Java. This, in turn, let us use two versions of RoboPrism: the Java version running on JamVM and the GNU C++ version running on a virtual machine developed by Bosch RTC. The 2.0.5 version of Player is compatible with JavaClient2, offering two options for interacting with iRobots for each version of RoboPrism. This highly heterogeneous environment has proven appropriate for validating the benefits of our approach. The "Hardware and Software Sources" sidebar provides links to websites providing more information about these tools.

HARDWARE AND SOFTWARE SOURCES

- iRobot Create, www.irobot.com/home/index.jsp
- eBox3854 embedded PC, www.microcomputersystems.com/eBox.htm
- iRobot Home Base, http://store.irobot.com/product/index.jsp?productId=2814855
- Java Media Framework (JMF), http://java.sun.com/javase/technologies/desktop/media/jmf
- SunSpot sensors, www.sunspotworld.com
- Create Open Interface library, http://code.google.com/p/libcreateoi

Environment exploration scenario

Our initial scenario involved exploring and mapping an unknown environment with randomly placed obstacles, as Figure 3 shows. We designed, modeled, and implemented this scenario using the Java version of RoboPrism. Five teams of two or three graduate students worked on this scenario during a 10-week, two-part project. Only one student had prior robotics or embedded-systems experience; four other students had previously been exposed to Prism-MW, the precursor to RoboPrism. The project was initiated before, but completed after, we obtained the iRobots. The project's objective was to investigate whether an explicit focus on software architecture and the use of architectural middleware could

- reduce the initial development effort and subsequent modification of a robotics system for non-experts in robotics,
- facilitate traceability (that is, preserve the designed architecture in the implementation),
- enhance exchange of design solutions, and
- alleviate heterogeneity challenges.

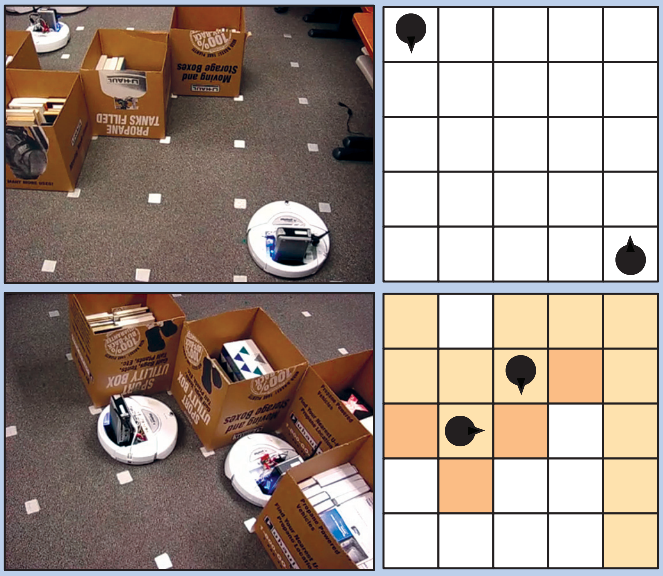


Figure 3. Two remote-controlled robots map out a 5 × 5 grid with unknown obstacles. The initial configuration, indicated by the blank map containing only the robots' positions and orientations, appears at the top. An intermediate configuration, with a majority of the grid traversed and four obstacles found, appears at the bottom.

The project's first part involved developing a simulated environment exploration system, in which the robots were "virtual"—simulated in a GUI. The virtual robots had to run on a host other than the host from which they were controlled. Like real robots, they had to move in the requested direction and report any obstacles found so that the students could construct a map of the environment.

The project's second part involved replacing the virtual robots with the iRobots. Students had to do so without altering the application's architecture: all changes to the components running on the (initially virtual and then real) robots had to be contained entirely inside the components.

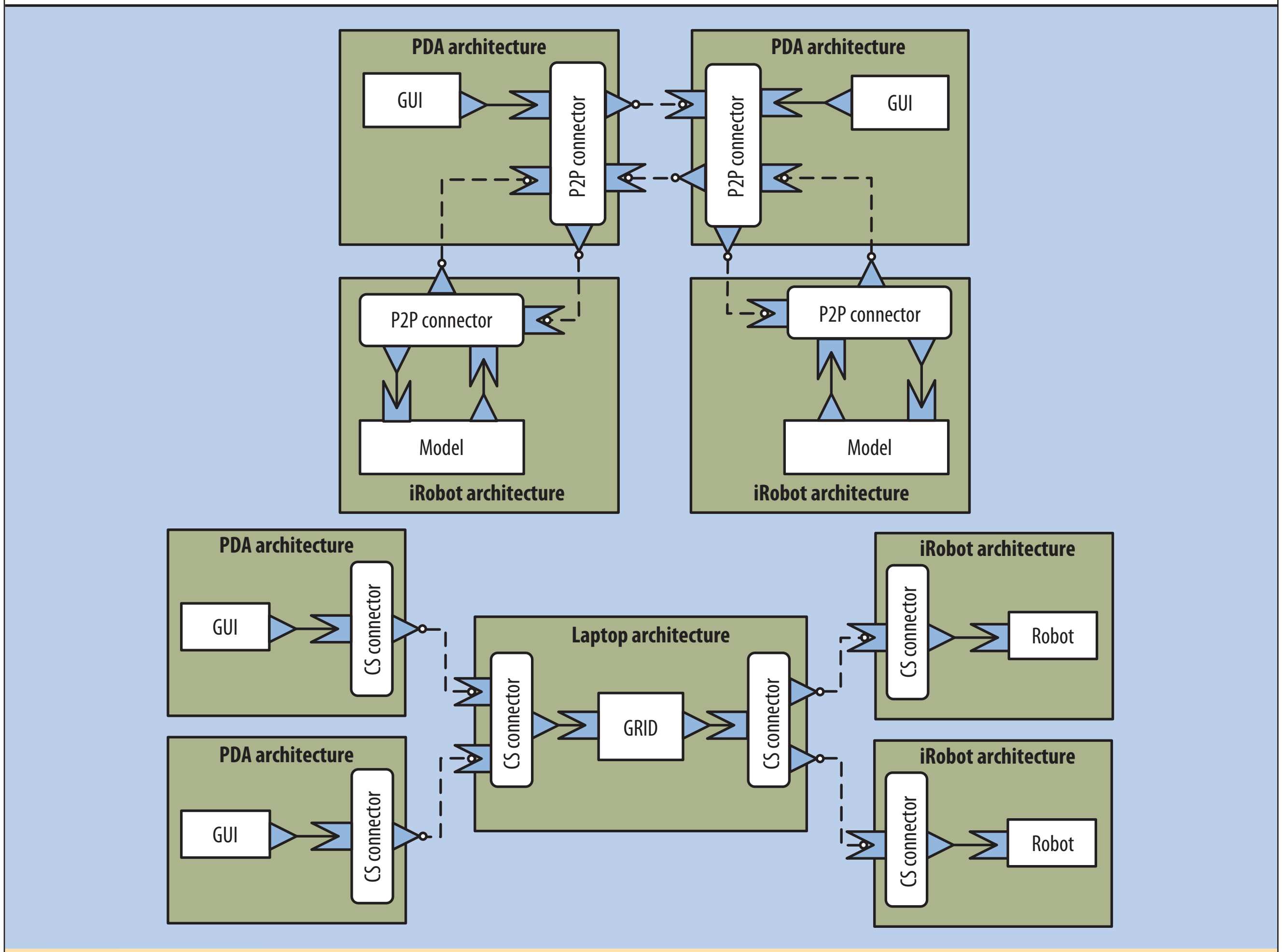


Figure 4. Two architectures for the environment exploration scenario, each relying on a different style: peer-to-peer (top) and client-server (bottom).

All five teams succeeded in preserving their architectures during the migration to iRobots. This success implies better maintainability of the resulting systems because the implementations preserve the designed architectures, avoiding architectural drift.

Two teams experienced difficulty controlling the iRobots' movement while trying to accurately map the unknown environment. The primary difficulties arose from their unfamiliarity with programming robots, the iRobot Create platform, and the Player library. The five resulting applications had similar functionalities with minor variations in numbers of PDAs used and the navigation algorithm's degree of automation. However, because our approach does not mandate a particular architecture for a system, engineers can explore and decide on the architecture that best fits their design decisions and objectives. As a result, the five architectures were substantially different in terms of the system decomposition into components and connectors, interfaces, interactions via events, and deployment onto the hardware nodes.

For example, Figure 4 shows two architectures that emerged from this project. The peer-to-peer solution in the top diagram will likely scale well and remain tolerant to host failures. On the other hand, it could experience data consistency problems if the events sent by peers are dropped or arrive and are processed in the incorrect order. The client-server solution in the bottom diagram has a central grid component that ensures a consistent global view of the system and avoids synchronization problems. At the same time, the grid component represents a single point of failure and might also become a performance bottleneck.

While this system's heterogeneity otherwise might have posed a serious problem in migrating from a Java GUI-based back end to the iRobots running Player, the application's use of RoboPrism greatly reduced such problems. In particular, the middleware allowed seamless communication among components regardless of the hardware platform that housed them. Demonstrating code portability and modularity, the students were able to easily wrap the robot control libraries and use them inside the components they had developed in the first part of the project.

The software design and implementation support let users with little domain expertise rapidly develop distributed, user-friendly robotics applications. The explicit focus on software architecture facilitated easy communication and exchange of high-level design solutions. Furthermore; we reused several modules from these systems in later research.

To evaluate the reduction of effort from using our approach, we measured the source lines of code (SLOC) and development effort estimates for the students' application code; Table 1 shows these results. We estimated effort using the Cocomo II software project cost estimation model,[9] and intend these numbers to indicate the complexity of the students' application code. Cocomo II suggests that two- to three-person teams with no personnel turnover would have required 3.8 to 8.8 months to produce the respective amounts of code. In contrast, the students in this project completed their work much more quickly; on average, they expended about four weeks of concerted programming effort. Although a more definitive conclusion would require further investigation, these numbers are suggestive of RoboPrism's effectiveness.

Table 1. Source lines of code and development effort estimates for the student projects.

Code base	No. of team members	SLOC	Development effort estimate (person-months)
1	3	1,600	3.8
2	2	2,700	6.7
3	2	2,900	7.5
4	2	1,700	4.2
5	2	3,400	8.8

Robot-following scenario

We designed and implemented several variations of the robot-following scenario. In the first set of scenarios, designed in tandem with an industrial collaborator, we manually designed adaptation policies and coded them in metalevel components, according to the adaptive-layered style. In the second set of scenarios, we leveraged PLASMA[5] to automatically design the adaptation plans as well as the application architecture.

Adaptive-layered implementations. In this scenario, the leader robot follows a line drawn on the floor using infrared sensors. Other robots use a camera to observe the color of and follow the robot in front of them. A robot also can follow an infrared signal emitted from the robot in front of it. A robot uses the infrared mechanism when it doesn't have a camera or its camera malfunctions. Along the way, robots encounter base stations and SunSpot sensors; they can choose to dock with the base stations to recharge their batteries, exchange data with SunSpots, or perform software updates. Robots dock and update software through autonomous control components.

Researchers also can use SunSpots as remote controllers to correct the orientation of an iRobot when it loses sight of the robot in front of it. When a robot leaves the convoy, it notifies the robot immediately behind it, and the remaining robots adjust their leader-follower roles to maintain the organization. A robot can rejoin the convoy when it sees the trailing robot's color. Researchers can issue commands from laptops and iPAQ, and they can receive feedback about the robots' progress and energy consumption.

To enable this functionality, we designed several components, including LineFollower, ColorFollower, IRFollower, SunSpotController, and SunspotReader. We also designed metalevel RoboPrism components to directly support runtime monitoring, analyses, and the system's dynamic adaptation. These components monitor and adapt the system's architecture in anticipated situations. For example, a monitor component detects camera failures and initiates an adaptation plan, which in turn replaces the ColorFollower component with an IRFollower component. These RoboPrism components organize the application and metalevel components into a two-layer adaptive-layered architecture.

We designed the adaptation policies captured within the metalevel components and refined them using XTEAM models. First, we used the rate of battery drain during different operational modes such as camera following, infrared following, and so on to determine appropriate thresholds at which to trigger recharging. Second, XTEAM analyses determined that we could not deploy all the follower components simultaneously due to the robots' limited available memory, necessitating component redeployment when hardware or software faults trigger adaptation policies.

This scenario demonstrates several benefits of our approach, including

- modeling and nonfunctional property analysis for adaptive systems,
- heterogeneity support;
- traceability, reuse, and modularity; and
- runtime architectural analysis and adaptability.

We first designed the entire system by exploring appropriate decompositions into components and connectors, as well as different candidate architectural styles. Then, we modeled the resulting design and analyzed it using XTEAM for completeness, consistency, and nonfunctional characteristics. We then transferred the model directly to the system implementation via RoboPrism's native support for architectural constructs. This allowed us to create a modular architecture that exhibited desired properties.

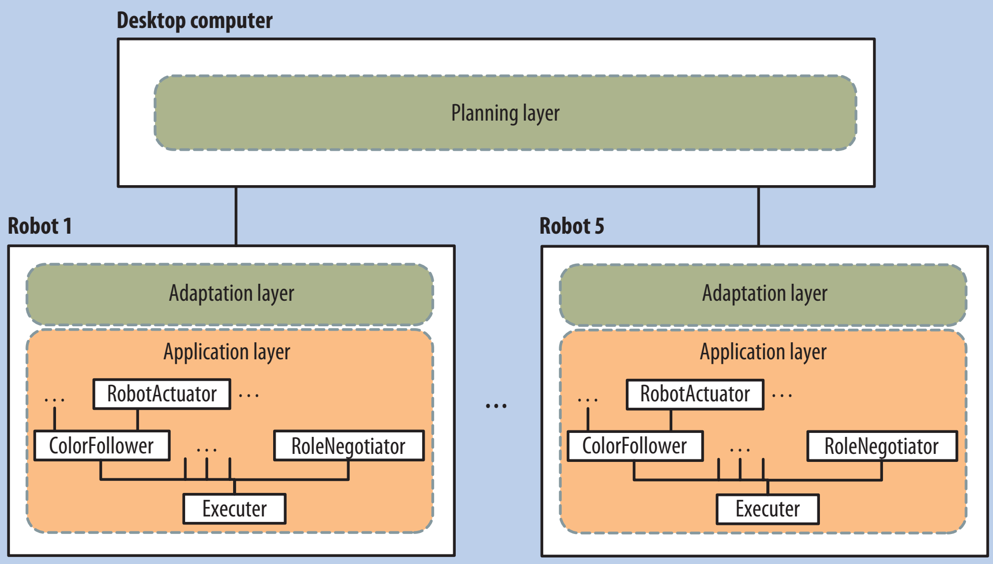


Figure 5. Deployment view of the system architecture for the robotics scenario.

PLASMA implementation. The three-layer PLASMA architecture follows the adaptive-layered style supported by RoboPrism, enabling a high degree of separation, modularity, and multilayer adaptation. To provide a high degree of autonomy, PLASMA relies on architect-generated SADEL models of the components in the application and adaptation layers. We transform the SADEL models into state transition models to use in adaptation planning.

PLASMA constructs separate plans for the application and adaptation layers. The application plans control the application behavior to achieve system goals. Similarly, the adaptation plans control the behavior of the adaptation layer—setting and adapting the application layer's architecture. To assess the benefits of PLASMA's dynamic adaptation support, we implemented a variant of the robotics scenario.

While transferring the robotics scenario to PLASMA, we successfully reused most of the application components from the scenario implementation; this further validated the reusability of our approach. In the PLASMA version, the leader robot follows a path defined by a series of spatial coordinates called waypoints. Initially, we provided PLASMA with the SADEL models of 15 application components. One component developed for the PLASMA scenario, RoleNegotiator, implements a distributed negotiation to assign a role (leader or follower) to all robots in the convoy. The negotiation protocol ensures that it assigns only one robot the leader role. Only the leader uses waypoint following; followers use other types of following.

PLASMA reduces the burden on the system architect by automatically generating adaptation plans, which the architect would otherwise specify manually. In PLASMA, the architect only provides the application's goal.

In our scenario, each robot's goal is to follow the robot in front of it and avoid obstacles. The PLASMA planning layer, deployed on a laptop, generates application and adaptation plans. The planning layer also automatically generates and compiles implementation code for the adaptation analyzer and executor components that perform the adaptation.

PLASMA then deploys compiled binaries of all required components (application components, adaptation analyzer, collectors, and so on) and instantiates an identical adaptation layer on each robot. The adaptation layer on each robot instantiates the application layer, and the Executor begins executing the application plan, in which the first step is role negotiation. Figure 5 shows an instance of this architecture's deployment.

Automatically generated application and adaptation plans support different types of system adaptations under different circumstances. As a result, the system architect need not predict and plan for all adaptations. The application plan automatically handles basic adaptations. For example, if a robot is using a camera for following and the area becomes dark, the Executor can use an application plan to automatically switch to GPS or infrared following. More powerful adaptations require dynamic replanning.

Consider the case in which robots must recharge their batteries using docking stations along the route. To satisfy this requirement, we specified new SADEL models for the BatteryMonitor and StationDocker components. We also specified a new application goal that defines the acceptable battery power threshold, and then initiated replanning. PLASMA computed new plans and regenerated and redeployed the Adaptation Analyzer and Executor, along with the other required application components.

Another adaptation occurs when a component fails and the application removes it from the set of available components. In this case, replanning adapts the application with a new application plan that does not use the removed component.

The automatically generated plans in these scenarios ranged from 790 to 4,390 state actions. Each state action specifies the behavior required in a specific state (for example, the invocation of a particular operation). Manually specifying policies of this size would be tedious, cumbersome, and error prone. By automating the process, PLASMA removes this burden, letting architects focus on their primary task—architectural description. For example, modifying the application goal in the case of battery recharging only requires specifying two additional SADEL models (55 lines of architectural description) and a new problem description (a single line change), along with the implementations of the two components.

Software architecture provides critical abstractions, techniques, and tools for designing and organizing software systems, and is particularly important in the case of complex heterogeneous systems that might need future extension or modification. To make it easier to use software architectural concepts in robotics, we have created three tools: XTEAM to automate system modeling and analysis; RoboPrism to give architectural abstractions first-class status in system implementations and allow dynamic analysis and redeployment of the system; and PLASMA to dynamically generate complex adaptation plans.

In our future research, we intend to expand the boundaries of using software engineering and software architecture concepts in the context of robotics systems. The recent improvements in the area of domain-specific modeling languages can facilitate flexible modeling of robotics applications in different domains, while preserving compatibility with existing analysis tools.[10] Further, we plan to enhance our adaptive framework with runtime reasoning about nonfunctional properties in an environment that has notable resource constraints. We believe that these enhancements will make robotics systems more accessible, reproducible, reusable, and adaptable to changes in their runtime environment. C

References

1. D. Brugali and E. Prassler, "Software Engineering for Robotics," *IEEE Robotics and Automation Magazine*, Mar. 2009, pp. 9, 15.
2. D. Brugali, "From the Editor-in-Chief: A New Research Community, a New Journal," *J. Software Eng. for Robotics*, Jan. 2010, pp. 1-2.
3. R.N. Taylor, N. Medvidovic, and E.M. Dashofy, *Software Architecture: Foundations, Theory, and Practice*, John Wiley & Sons, 2009.
4. G. Edwards et al., "Architecture-Driven Self-Adaptation and Self-Management in Robotics Systems," *Proc. Int'l Workshop Software Eng. for Adaptive and Self-Managing Systems* (SEASS 09), IEEE CS Press, 2009, pp. 142-151.
5. H. Tajalli et al., "PLASMA: A Plan-Based Layered Architecture for Software Model-Driven Adaptation," *Proc. 25th IEEE/ACM Int'l Conf. Automated Software Eng.* (ASE 10), IEEE CS Press, 2010, pp. 467-476.
6. N. Medvidovic, D.S. Rosenblum, and R.N. Taylor, "A Language and Environment for Architecture-Based Software Development and Evolution," *Proc. 21st Int'l Conf. Software Eng.* (ICSE 99), IEEE CS Press, 1999, pp. 44-53.
7. G. Edwards and N. Medvidovic, "A Methodology and Framework for Creating Domain-Specific Development Infrastructures," *Proc. 23rd IEEE/ACM Int'l Conf. Automated Software Eng.* (ASE 08), IEEE CS Press, 2008, pp. 168-177.
8. S. Malek, M.M. Rakic, and N. Medvidovic, "A Style-Aware Architectural Middleware for Resource Constrained, Distributed Systems," *IEEE Trans. Software Eng.*, Mar. 2005, pp. 256-272.
9. B. Boehm et al., "Cost Models for Future Software Life Cycle Processes: COCOMO 2.0," *Ann. Software Eng.*, Dec. 1995, pp. 57-94.
10. G. Edwards, "Automated Synthesis of Domain-Specific Model Interpreters," doctoral dissertation, Dept. Computer Science, Univ. of Southern California, 2010.

Nenad Medvidovic *is a professor in the Department of Computer Science at the University of Southern California and director of the USC Center for Systems and Software Engineering. Medvidovic received a PhD in information and computer science from the University of California, Irvine. Contact him at neno@usc.edu.*

Hossein Tajalli *is a PhD student in the Department of Computer Science at the University of Southern California, where he is a member of the Software Architecture Research Group in the Center for Systems and Software Engineering. Tajalli received an MS in electrical engineering from the University of Tehran, Iran. Contact him at tajalli@usc.edu.*

Joshua Garcia *is a PhD student in the Department of Computer Science at the University of Southern California. He received an MS in computer science from the University of Southern California. Contact him at joshuaga@usc.edu.*

Ivo Krka *is a PhD student in the Department of Computer Science at the University of Southern California, where he is a USC Provost's Fellow. He received an MS in computer science from USC and an MEng in computing from the University of Zagreb. Krka is a member of IEEE and ACM Sigsoft. Contact him at krka@usc.edu.*

Yuriy Brun *is an NSF CRA postdoctoral Computing Innovation Fellow at the University of Washington. He received a PhD in computer science from the University of Southern California. Brun is a member of the ACM and ACM Sigsoft. Contact him at brun@cs.washington.edu.*

George Edwards *is the chief scientist at Blue Cell Software in Los Angeles. He received a PhD in computer science from the University of Southern California. Contact him at george@bluecellsoftware.com.*