A Robot Team for Surveillance Tasks: Design and Architecture

Sascha A. Stoeter*, Paul E. Rybski, Kristen N. Stubbs, Colin P. McMillen,
Maria Gini, Dean F. Hougen, Nikolaos Papanikolopoulos
Department of Computer Science and Engineering, University of Minnesota, U.S.A.

Reduced cost of robotic hardware enables the use of teams of robots instead of a single device. Multi-robot approaches promise faster results and more robust systems as each individual robot becomes dispensable. Given higher numbers of robots, writing dependable control software becomes more complex and thus more expensive. Consequently, a software architecture that is readily applied to new missions becomes essential. In the following, an architecture for distributed control of a team of heterogeneous mobile robots is introduced. Design as well as implementation details are presented. A distinguishing feature of the architecture is its versatility in handling resources. An example application for a surveillance task is discussed.

Keywords: multiple robots; software architecture; surveillance applications; resource allocation

1. Introduction

With unit costs dwindling, multi-robot teams are being used more widely. As a considerable number of problems lend themselves to divide-and-conquer approaches, teams of robots are an obvious choice to increase performance. For instance, an area could be subdivided into multiple parts that are assigned to different robots. Robots would then operate in their portions in parallel, accomplishing the task faster. Multiple robots also provide for redundancy that might save a mission from complete failure. If a robot malfunctions, its task could be taken over by another nearby robot. This is especially true for homogeneous robot teams. Heterogeneous teams could also benefit from a graceful degradation of performance instead of failing outright.

Writing control software for mobile robots is a non-trivial task due to the need of operating in noisy and cluttered environments. Additional complexity is introduced by the need for robots to communicate to each other over a wireless network. This impose constraints on the bandwidth, maximum distance between robots, and requires error correction. Additional planning must be introduced to effectively control robots with heterogeneous capabilities. As discussed by Coste-Manière and Simmons [7], a good architecture is vital to the construction of working robotic systems.

We propose an architecture for multi-robot control that treats a distributed network of robots as a set of dynamic resources. Robot control is accomplished through a hierarchical behavior tree, which operates across a location-transparent wireless network. The entire system is designed to be extremely modular, allowing for rapid addition of resources and behaviors to create new missions.

The remainder of this paper is organized as follows. An overview of related work is given in the next section. Section 3 describes our heterogeneous robotic team. The software architecture is presented in detail in Sections 4 and 5. The paper concludes with a list of future research directions.

2. Related Work

Researchers have taken numerous approaches to collaborative robot control. Cao *et al.* deserve credit for attempting the categorization of different architectures in the cooperative, multi-robot domain [6].

There has been a decade and a half of research aimed at providing robots with a behavioral or task-based decomposition of the control system, since Brooks introduced it as an alternative to the traditional functional approach [4]. The application of behavior-based robotics to groups of robots has been explored extensively in the latter half of the 1990s (e.g., [15]) and architectures for cooperative control have recently been introduced (e.g., [20]). Much work is focused on how different behaviors have to be combined to achieve a desired effect that solves a given problem. Proposed solutions include hard-coded or learned weights [17]. Attempts have also been made to analytically determine the minimum information requirements for solving a task [9] and at automatic generation of robot teams [18]. However, most of these systems have not tackled the problems of distributed collaborative behaviors and distribution of resources across robots.

The idea of splitting behaviors into sensing, computing, and acting across multiple physical robots has been approached by treating a whole collection of robots as a single *generalized vehicle* [22]. This allows appropriate sensors, computational resources, and actuators to be chosen for each task without requiring them to all come from the same physical robot.

KAMARA [14] is a true collaborative behavior architecture which has been implemented on the Karlsruhe Autonomous Robot, which consists of two manipulator arms with several sensing components all mounted on a mobile base. The distributed and decentralized modes of operation are adequate for independent operations of the two manipulator arms and mobile base, but lack the coordination and communication needed for a larger group of robots.

The MARTHA project by Alami *et al.* deals with coordinating a large number of mobile robots in a structured environment for shipment missions [1]. The robots can communicate with each other and with a central control station that issues jobs. Both a topological and a geometrical map have to exist. Efforts are made to resolve deadlocks locally, but for a large fleet that operates in a small area, the system becomes centralized. The cooperation is limited to avoiding the other robots. Similarly, Brumitt *et al.* present a method for multi-robot path-planning executed on a distributed system [5]. This work requires accurate pose estimation. Wang *et al.* propose adding extra space to the environment for resolving deadlocks thus circumventing the need for a central planner altogether [25].

Blum proposes a hierarchical control architecture named OSCAR that utilizes modules as reusable software components [3]. Modules encapsulate resources and processing entities. They can be grouped into multi core modules that can be executed on different computers. A control module implemented as a state machine activates required modules. The architecture has been used to control a single mobile robot in an exploration mission.

Our architecture has some similarities with CAMPOUT [19], a distributed hybrid-architecture based on behaviors. The major difference is that we focus on resource allocation and dynamic scheduling, while CAMPOUT is mostly designed for behavior fusion. We rely on CORBA [16] for distributed processing, while in CAMPOUT each robot runs an instance of the architecture and uses sockets for communication with other robots.

Inter-robot communication is primarily achieved with wireless networks as exemplified by the majority of the presented architectures. It is assumed that direct communication between any pair of communication endpoints is available. Asama *et al.* take a different approach. They use *intelligent data carriers* which are distributed in the environment by the robots and serve as information kiosks [11,13]. Typically, local information about the immediate neighborhood is saved onto them. Robots passing by can read and update the stored information only within the device's vicinity.

Resource allocation and dynamic scheduling are essential to ensure robust execution. Our work focuses on dynamic allocation of resources at execution time, as opposed to analyzing resource requests off-line, as in [2,10], and modifying the plans when requests cannot be satisfied. Our approach is specially suited to unpredictable environments, where resources have to be allocated in a dynamic way that cannot be predicted in advance. We rely on the wide body of algorithms that exists in the area of real-time scheduling [23] and load balancing [8].

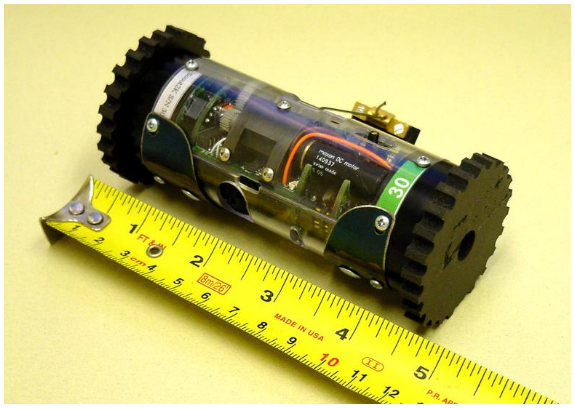
Fig. 1. Closeup of a Scout robot. The foot required for jumping is located on the robot's far side.

3. Hardware

Our heterogeneous robotic team consists of two types of robots. The first type is a larger, heavy-duty robotic platform called the *Ranger*. It is used to transport and deploy a number of small, mobile sensor platforms called *Scouts*, the second type of robot, into the environment. Together, the Scouts and Rangers form a hierarchical team capable of carrying out complex missions in a wide variety of environments. A team is created by pairing several Scouts with one or more Rangers. Because of the Scout radios' range limitations, a Ranger has to remain close by to act as a proxy. Proxy processing allows the Scouts to engage in visual servoing – an activity they would never be able to accomplish with their own limited computational resources.

3.1. Scouts

Scouts have a cylindrical shape, 40 mm in diameter and 110 mm in length (see Fig. 1). A Scout moves using a unique combination of locomotion types. It can roll using the wheels mounted on both ends of its body, and jump using a spring mechanism (also known as the *foot*). Rolling allows for efficient traversal of smooth surfaces, while jumping allows Scouts to operate in uneven terrain and pass over obstacles.

The Scouts act as the mobile eyes and ears of the team. Their electronics include transmitters and receivers, microcontrollers, magnetometers, and tiltometers. In addition, they have a modular sensor payload which can carry a miniature video camera with an optional pan-tilt unit and video transmitter, a microphone, a vibration sensor or a gas sensor. More details on the hardware can be found in Hougen *et al.* [12].

3.2. Rangers

The Rangers are based on the ATRV-Jr.™ platform from iRobot (see Fig. 2). A single Ranger can carry a payload of roughly 25 kg. With a battery life of 3 to 6 hours (depending on terrain and load), the maximum range is about 20 km. The Ranger is customized with a Scout launcher that includes a magazine, Scout communication hardware, and a video camera. The Rangers are equipped with on-board personal computers that have video capture cards allowing them to process images from their own camera as well as from the cameras mounted on the Scouts.

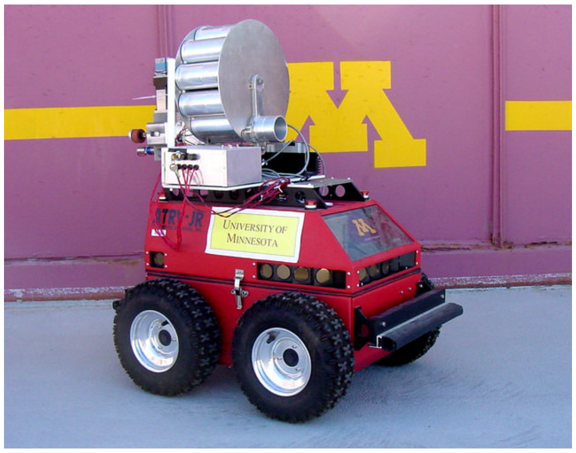

Fig. 2. The Ranger with a custom-built magazine for ten Scouts.

The Scout launcher allows the Ranger to shoot Scouts through windows or over obstacles that the Ranger could not itself surmount (see Fig. 3). The Ranger can deploy up to ten Scouts in any order from its launcher before needing to be reloaded. With control over the launch angle and propulsive force, a Scout can be launched up to 30 m.

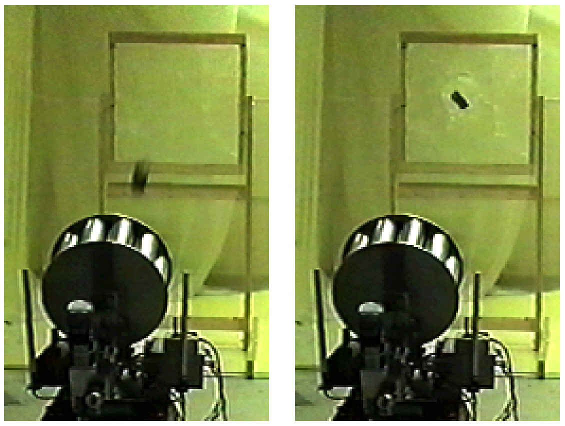

Fig. 3. Deployment of a Scout from a Ranger. The Scout is enclosed in a padded shell for protection from the impact of hitting the window and the floor.

4. Design Goals of the Software Architecture

The goal of the control software is to provide distributed, fault-tolerant management of all available resources to fulfill the mission objectives. Additionally, mission design time is to be minimized while resources should be utilized efficiently.

The software architecture is composed of four major subsystems as depicted in the UML diagram in Fig. 4. The Mission Control is the brain of the system. It consists of layers of increasingly simple behaviors that execute the desired mission. The Resource Pool provides the behaviors with priority-managed, shared access to physical hardware as well as software resources such as maps. Humans interact with the system through the User Interface. The Backbone binds the other subsystems together.

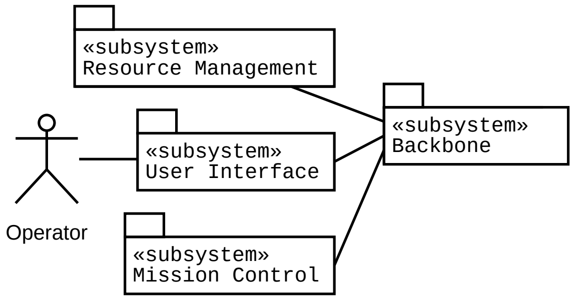

Fig. 4. The architecture's major subsystems and user interaction.

4.1. User Interface Subsystem

Humans access the system through the User Interface. In the off-line mission design phase, this subsystem provides tools to allow the user to construct missions. Throughout the entire distributed system, many components, i.e. resources and behaviors, are at the mission designer's disposal. Ideally a designer is offered prefabricated components that can be tied together easily to form a new mission. While the designer must certainly be aware of their functional characteristics, the system should hide lower-level details such as their execution location unless such information is requested by the designer. After all, information about the exact location of components is generally not available until run-time. Computational resources can vary from one run of mission to the next, and computers may become inoperable during mission execution.

During run-time, requirements of the User Interface are highly mission-dependent. For certain scenarios, such as rescue operations, the interface must be interactive. In other scenarios, such as planetary exploration, the interface will be used mostly for providing occasional status since real-time human response is not possible.

Applications such as search and rescue operations in urban terrain do not allow for cumbersome user interfaces. In a potentially messy, fast-paced situation, the users must be presented with an intuitive and foolproof user interface that takes as little of their attention away from the dangerous situation surrounding them as possible. A simple prototype of a user interface is shown in Fig. 5. It consists of a personal digital assistant (PDA) as the command center, a monitor to view the robots' video transmissions, and a backpack filled with an antenna, a tuner, and a power supply. The monitor and the PDA can be attached at any position on either sleeve of the operator with velcro, leaving the hands free to carry additional items.

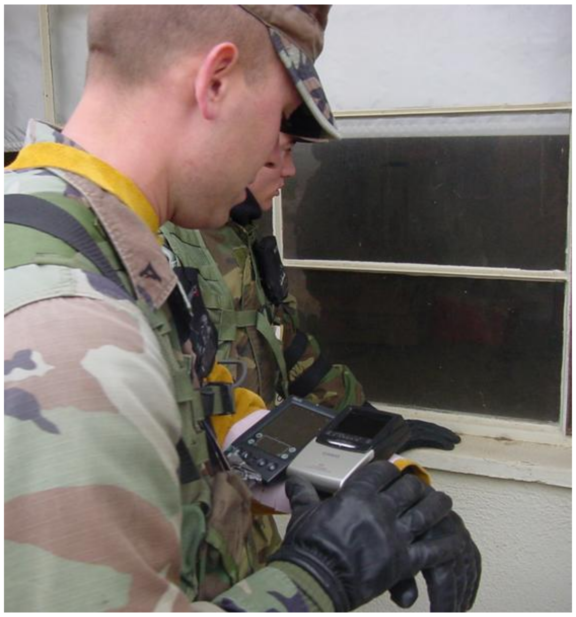
Fig. 5. Operator wearing prototype user interface. Commands can be issued via the PDA while the LCD monitor is used to view video transmitted from a Scout.

4.2. Mission Control Subsystem

The Mission Control hosts prioritized behaviors that make up a solution to the mission. A behavior is defined as a functional unit that works on achieving a well-defined task. A mission is modeled as such a behavioral component. A complex top-level task (e.g., Find Intruders in a compromised building) is recursively decomposed into simpler behaviors until elementary behaviors (e.g., Drive Forward) are reached. Behaviors can be executed sequentially or in parallel. Composite behaviors can be created using existing behaviors.

4.3. Resource Pool Subsystem

The Resource Pool maintains the interfaces to *behavior resources* which are directly requested for use by behaviors. Such behavior resources are usually specific pieces of hardware (e.g., framegrabbers), but radio frequencies or maps of the environment can also be used. This should not be confused with *system resources*, such as available CPU time or memory, to which behaviors have only implicit access. For notational simplification, the term *resources* denotes behavior resources unless otherwise specified.

To achieve efficient resource usage, behaviors are encouraged to request only that portion of a resource's capacity that they need to successfully execute their work. The remaining portions are then still available to other behaviors. Access to a resource by a behavior may also be limited by other constraints. A robot may have the constraint to not travel beyond a certain distance from a landmark, but may move freely within that boundary. The Resource Pool provides for

a. *resource subscription* to allow behaviors to request access to resources,

b. *access control* to coordinate the utilization of shared resources,

c. *preallocation* of resources to behaviors to ensure their availability when needed,

d. *constraints monitoring* to watch for resource control violations, and

e. *constraints enforcement* to take control over resources and resolve conflicts.

4.4. Backbone Subsystem

The Backbone ties the other subsystems together. It is the responsibility of the Backbone to enable the components of the system to work seamlessly over all allocated machines. It also tries to balance system resource usage to avoid bottlenecks. The Backbone provides for

a. *component placement* to start components on a specified platform,

b. *location transparency* of components over all platforms to ease addressing,

c. *communication* among the subsystems,

d. *location information* of both running and available components,

e. *hot-plugability* to allow for run-time addition and removal of components,

f. *system load monitoring* to report on the workload of the system resources,

g. *load balancing* to spread the workload evenly over the available system resources,

h. *redundancy* for fault-tolerance, and

i. *component migration* from one computer to another.

5. Implementation

At present, only a limited number of the design goals (a–b from the Resource Pool and a–g from the Backbone) are implemented with the remaining to follow after initial proof of concept.

The system relies extensively on CORBA [16] and XML [26] in order to achieve a high degree of flexibility while keeping the code portable. All entities in the system are registered with a CORBA name service and can therefore be easily addressed. XML documents all components (i.e. their name, parameters including types and default values, and a human readable textual description) as well as missions and static dependencies among the hardware.

5.1. The Startup Services

In order to launch a mission, a number of core system services must be in place. The first service is the Distributed Robotics Daemon, the master startup service, which runs on each of the computers that will take part in a mission. When contacted by a User Interface, it launches a CORBA environment to serve the behaviors. The second service, the Component Database, keeps track of all components available to the system storing names, locations, types, and multiplicity information, i.e. the number of instances of each component allowed to exist simultaneously. Component Creators, the third service, are started on all machines. On mission startup, each Component Creator is responsible for determining the available components on its host and passing that information along to the Component Database. While the mission is running, the Component Creator serves requests for launching components and registering them with the Name Service. A Load Balancer receives periodic status updates from Load Reporters instantiated on all hosts. This information is used to start new components on the least used machine with respect to some metric. The final service, the Component Placer, is the global service to start components and saves clients from having to know what hosts to run components on.

Fig. 6 shows a typical startup sequence for a component C. When the Component Placer (CP) receives a request from a client (CPC) such as a behavior, it queries the Component Database (CD) for a host that supports execution of the component. A specific host is picked by the Load Balancer (LB) which has knowledge about each participating machine's resource usage such as processor load and memory utilization. The host selection can be influenced through policies that determine the importance of the resources. The Component Placer then creates a unique CORBA name for the component and instructs the Component Creator (CC) on the chosen host to start it. The component initializes and registers itself first with the Name Service (NS) and then reports to the Component Placer. The Component Placer returns either the component's CORBA

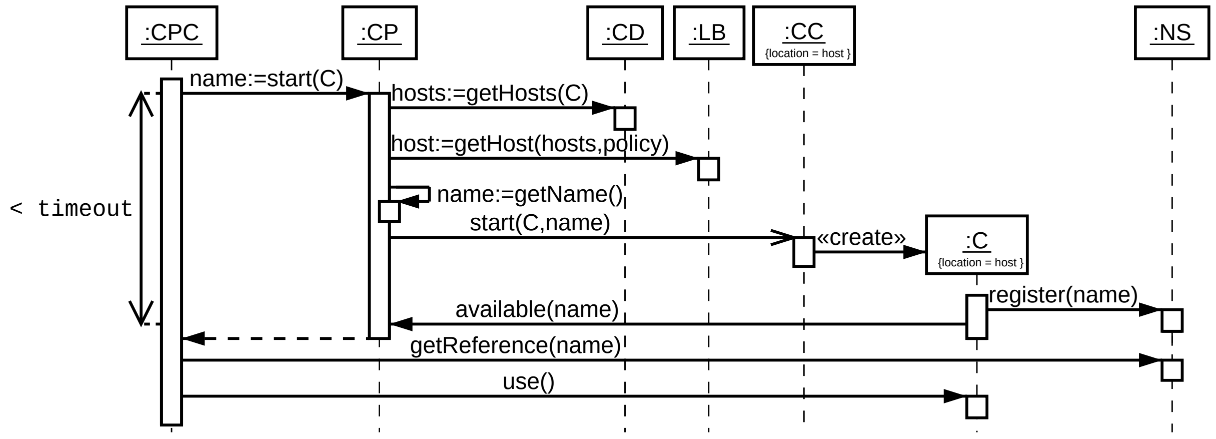

Fig. 6. Interaction diagram showing the component startup sequence.

name to the client or a failure condition in the case of a timeout. Timeouts can occur when a component could not be started or the network connection has been lost. At last, the client can access and use the new component.

5.2. Access to Resources

The Resource Controller Manager manages components called Resource Controllers (RCs) and Aggregate Resource Controllers (ARCs). RCs provide standard CORBA interfaces to single hardware or software resources. Because multiple RCs must often be used simultaneously when controlling something complex like a robot, ARCs are used as higher-level interfaces to groups of RCs (see Fig. 7). Behaviors cannot access RCs directly and are given access to ARCs by the Resource Controller Manager. An ARC serves only a single behavior.

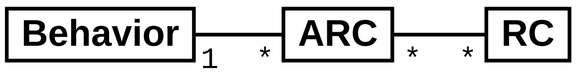

Fig. 7. Class diagram showing the association of Behaviors, Aggregate Resource Controllers, and Resource Controllers.

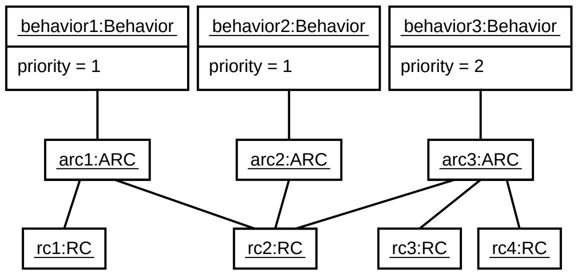

Fig. 8. Example of behaviors accessing resources.

Some RCs can be used simultaneously by multiple ARCs while others cannot. When an ARC requests an RC from the Resource Controller Manager, it specifies a time slice and, in the case of a sharable RC, designates a quantity. ARCs can share an RC unless the sum of their requested quantities exceeds a maximum allowed capacity. Conflicts arising from concurrent requests are resolved by granting access based on the ARCs' priorities. Each ARC inherits its controlling behavior's priority. In the example shown in Fig. 8, each of the three behaviors is trying to obtain access to the non-sharable RC rc2 through its ARC. The priority 1 behaviors will be able to share the RC in a round-robin fashion while the priority 2 behavior will block until the priority 1 behaviors release it.

Each behavior is programmed with the knowledge of what ARCs it needs and what kinds of RCs each ARC controls. This allows behaviors to instantiate a particular ARC by parameterizing it on a list of RCs. To specify this information, behaviors must be aware of dependencies among pieces of hardware. Since the users can change the hardware between runs, the Static Dependency Database stores the names, locations and startup options for the RCs available in the system and provide this information to the behaviors.

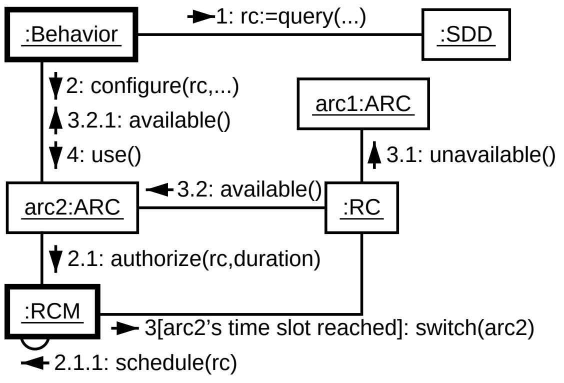

Fig. 9. Behavior requesting access to an ARC.

When a behavior is initialized, its first task is to access all the ARCs that it will need to run. Fig. 9 illustrates an example of this for a single RC. The component startup as shown previously in Fig. 6 is omitted for simplicity. The behavior first contacts the Static Dependency Database (SDD) to select the particular pieces of hardware from a set of available resources for the ARCs it needs. Then it instructs the Component Placer to instantiate the desired ARCs and configures them with the chosen RCs. Each ARC requests its RCs to be scheduled by the Resource Controller Manager (RCM) for a required minimum runtime. This duration reflects the smallest time interval necessary for the behavior to achieve a subgoal. Unless the requested RCs are already present in the system, the Resource Controller Manager has the Component Placer start them and a new schedule is created. At the ARC's scheduled time, the Resource Controller Manager instructs the RCs to serve the new ARC upon which they break their ties with the presently served ARC and signal their availability to the new one. The ARC, in turn, informs its controlling behavior that it is now ready to serve.

5.3. Mission Creation and Startup

The user can start a mission once all core services are in place. This is done by starting the top-level behavior of the mission. Behaviors are organized in a hierarchical structure and so the higher-level behaviors activate lower level behaviors as needed. Mission designers need only specify a partial order plan; linearization is accomplished by the system. Complex behaviors can be built from simpler, existing behaviors. Parent behaviors request the creation of their children through the Component Placer and set their parameters after successful instantiation. Some parameters are required, while others are optional. Named variables are used to obtain a common interface to allow for run-time addition of new behaviors. The parent behavior then activates the children and awaits their termination. It must also react to an asynchronous shutdown message from its parent. In that case, it recursively shuts down its children.

6. A Walk through an Example Mission

A scenario we have demonstrated requires the robotic team to quickly explore a building and set up a surveillance sensor network. Experimental details are reported in Rybski *et al.* [21]. In this scenario, a Ranger moves into a building and searches for rooms into which to deploy Scouts, using control software to navigate autonomously along a corridor and to find open doors (see Stoeter *et al.* [24] for details). The Ranger then launches a Scout into each room. The Scouts investigate their rooms independently

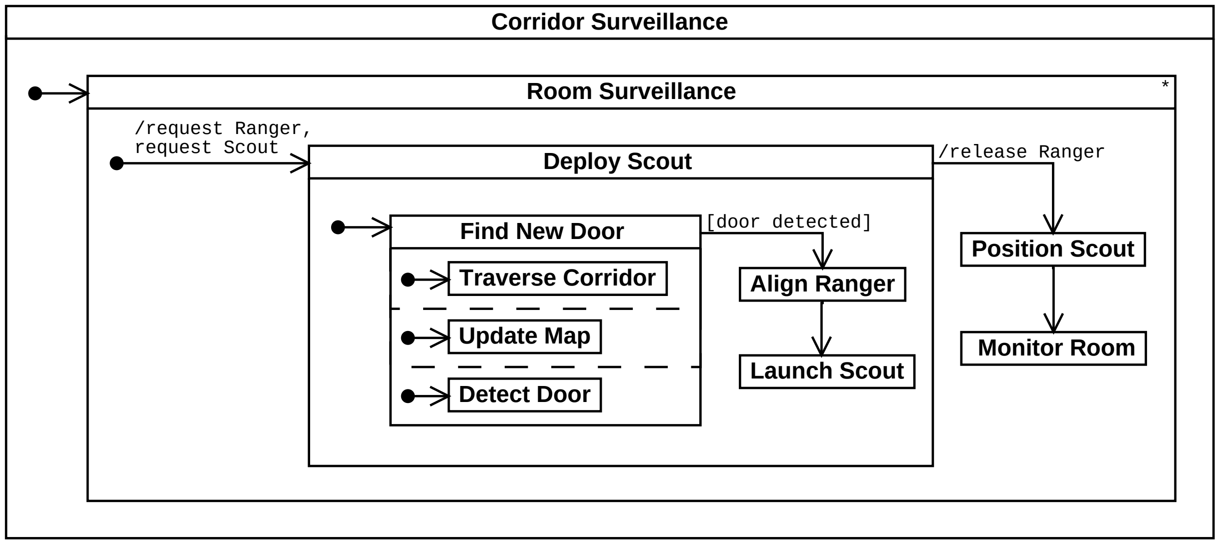

Fig. 10. Example corridor surveillance mission. The mission is decomposed into manageable behaviors.

(by transmitting images to the Rangers for proxy processing), move toward dark corners in which they can hide, turn to watch their area, and wait for people to move through the environment.

The mission is depicted in Fig. 10. A box corresponds to a behavior; arrows denote transitions between behaviors. For visual clarity, error transitions are excluded. The diagram shows the structural decomposition of the top level behavior Corridor Surveillance into simpler behaviors. The diagram also shows the behavioral aspects of the system, i.e. the flow of control from one behavior to the next. A transition is triggered when its condition becomes valid. This happens when a composite behavior is entered and its default transitions (as marked with dots at their origins) are activated, when the source behavior terminates, or when the transition condition (as shown next to the arrow in brackets) evaluates to true. When a transition is taken, first the source behavior is shut down, then the optional transition action (as denoted by a slash in the figure) is executed, and finally the destination behavior is activated.

Note that behaviors can execute in parallel as indicated by dashed lines for aggregated behaviors or by a multiplicity range next to a behavior's name. For instance, all three children behaviors of Find New Door run concurrently. Similarly, several Room Surveillance behaviors could be active at the same time.

At startup, Corridor Surveillance instantiates all its children, i.e. as many Room Surveillance behaviors as there are Scouts available for the mission. Each one tries to subscribe to a Ranger. If just a single Ranger resource is available in the system, only one Room Surveillance behavior can proceed while the remaining sleep until the Resource Controller Manager is able to fulfill their request. With multiple Rangers, the system would automatically grant access to more behaviors, and pieces of the mission could continue in parallel. This mechanism frees the mission designer from explicitly synchronizing resource access. Once in control of a Ranger, the behaviors use the Static Dependency Database to subscribe to a Scout that is stored in that Ranger's magazine.

When Find New Door is activated, it creates a map that it makes available to all three of its children. This is the only way for behaviors to share information, as they are unaware of each other's presence. When Detect Door terminates successfully, Find New Door shuts down the remaining two children and activation passes to Align Ranger to obtain a good launching position and then to Launch Scout. After the Scout has been deployed, Find New Door no longer requires a Ranger and thus frees the resource. At this point, the Resource Controller Manager hands the resource to another instance of Room Surveillance. The Scout is directed to a good location using visual servoing and subsequently begins monitoring the room for trespassers.

7. Future Work

Future work includes extensions of the system and more thorough testing. First, the remaining design goals will be tackled. The most interesting and challenging parts promise to be the component migration at run-time and the incorporation of dynamic constraints enforcement. The latter will free mission designers and component writers from a large amount of manual constraints-checking necessary at present, which is both expensive and error prone. Second, the system will be tested on a variety of different missions. We intend to develop solutions to a number of well studied problems and compare our system performance. We plan on exploring the unique opportunities offered by our robotic team.

Acknowledgement

This material is based upon work supported by the Defense Advanced Research Projects Agency, Microsystems Technology Office (Distributed Robotics), ARPA Order No. G155, Program Code No. 8H20, issued by DARPA/CMD under Contract #MDA972-98-C-0008.

[1] R. Alami, S. Fleury, M. Herrb, F. Ingrand, and F. Robert. Multi-robot cooperation in the MARTHA project. *IEEE Robotics and Automation Magazine*, pages 36–46, Mar. 1998.

[2] E. M. Atkins, T. F. Abdelzaher, K. G. Shin, and E. H. Durfee. Planning and resource allocation for hard real-time, fault-tolerant plan execution. *Autonomous Agents and Multi-Agent Systems*, 4(1/2), Mar. 2001.

[3] S. Blum. OSCAR – Eine Systemarchitektur für den autonomen, mobilen Roboter MARVIN. In R. Dillmann, H. Wörn, and M. von Ehr, editors, *Proc. of Autonome Mobile Systeme*, Informatik aktuell, pages 218–230, Karlsruhe, Germany, Nov. 2000. Gesellschaft für Informatik, Springer.

[4] R. A. Brooks. A robust layered control system for a mobile robot. *IEEE Journal of Robotics and Automation*, RA-2(1):14–23, Mar. 1986.

[5] B. L. Brumitt and A. Stentz. GRAMMPS: A generalized mission planner for multiple mobile robots in unstructured environments. In *Proc. of the IEEE Int'l Conference on Robotics and Automation*, pages 1564–1571, Leuven, Belgium, May 1998.

[6] Y. U. Cao, A. S. Fukunaga, and A. B. Khang. Cooperative mobile robotics: Antecedents and directions. *Autonomous Robots*, 4(1):7–27, Mar. 1997.

[7] E. Coste-Marière and R. Simmons. Architecture, the backbone of robotic systems. In *Proc. of the IEEE Int'l Conference on Robotics and Automation*, pages 67–72, San Francisco, CA, Apr. 2000.

[8] G. Cybenko. Dynamic load balancing for distributed memory multiprocessors. *Journal of Parallel Distributed Computing*, 7(2):279–301, 1989.

[9] B. R. Donald. On information invariants in robotics. *Artificial Intelligence*, 72(1–2):217–304, Jan. 1995.

[10] E. H. Durfee. Distributed continual planning for unmanned ground vehicle teams. *AI Magazine*, 20(4):55–61, 1999.

[11] T. Fujii, H. Asama, T. Fujita, Y. Asakawa, H. Kaetsu, A. Matsumoto, and I. Endo. Knowledge sharing among multiple autonomous mobile robots through indirect

communication using intelligent data carriers. In *Proc. of the IEEE Int'l Conf. on Intelligent Robots and Systems*, pages 1466–1471, 1996.

[12] D. F. Hougen, S. Benjaafar, J. C. Bonney, J. R. Budenske, M. Dvorak, M. Gini, D. G. Krantz, P. Y. Li, F. Malver, B. Nelson, N. Papanikolopoulos, P. E. Rybski, S. A. Stoeter, R. Voyles, and K. B. Yesin. A miniature robotic system for reconnaissance and surveillance. In *Proc. of the IEEE Int'l Conference on Robotics and Automation*, pages 501–507, San Francisco, CA, U.S.A., Apr. 2000.

[13] D. Kurabayashi and H. Asama. Knowledge sharing and cooperation of autonomous robots by intelligent data carrier system. In *Proc. of the IEEE Int'l Conference on Robotics and Automation*, pages 464–469, San Francisco, CA, U.S.A., Apr. 2000.

[14] T. C. Lueth and T. Laengle. Task description, decomposition, and allocation in a distributed autonomous multi-agent robot system. In *Proc. of the IEEE Int'l Conf. on Intelligent Robots and Systems*, volume 3, pages 1516–1523, Munich, Germany, Sept. 1994.

[15] M. J. Matarić. Issues and approaches in the design of collective autonomous agents. *Robotics and Autonomous Systems*, 16:321–331, Dec. 1995.

[16] Object Management Group. *The Common Object Request Broker: Architecture and Specification.* Object Management Group, Needham, MA, U.S.A., 1998.

[17] L. E. Parker. L-ALLIANCE: Task-oriented multi-robot learning in behavioral-based systems. *Advanced Robotics*, 11(4):305–322, 1997.

[18] L. E. Parker. Toward the automated synthesis of cooperative mobile robot teams. In *Proc. of SPIE Mobile Robots XIII*, volume 3525, pages 82–93, 1998.

[19] P. Pirjanian, T. Huntsberger, A. Trebi-Ollennu, H. Aghazarian, H. Das, S. Joshi, and P. Schenker. Campout: a control architecture for multirobot planetary outposts. In *Proc. SPIE Conf. Sensor Fusion and Decentralized Control in Robotic Systems III*, Nov. 2000.

[20] J. K. Rosenblatt. DAMN: A distributed architecture for mobile navigation. *Journal of Experimental and Theoretical Artificial Intelligence*, pages 339–360, 1997.

[21] P. E. Rybski, S. A. Stoeter, M. D. Erickson, M. Gini, D. F. Hougen, and N. Papanikolopoulos. A team of robotic agents for surveillance. In *Proc. of the Int'l Conf. on Autonomous Agents*, pages 9–16, Barcelona, Spain, June 2000.

[22] J. B. Sousa and F. L. Pereira. A general control architecture for multiple vehicles. In *Proc. of the IEEE Int'l Conference on Robotics and Automation*, pages 692–697, Minneapolis, MN, 1996.

[23] J. Stankovic, M. Spuri, K. Ramamritham, and G. Buttazzo. *Deadline Scheduling For Real-Time Systems: EDF and Related Algorithms.* Kluwer Academic Publishers, Boston, 1998.

[24] S. A. Stoeter, F. Le Mauff, and N. P. Papanikolopoulos. Real-time door detection in cluttered environments. In *IEEE Int'l Symposium on Intelligent Control*, pages 187–192, Rio, Greece, July 2000.

[25] J. Wang and S. Premvuti. Distributed traffic regulation and control for multiple autonomous mobile robots operating in discrete space. In *Proc. of the IEEE Int'l Conference on Robotics and Automation*, pages 1619–1624, 1995.

[26] World Wide Web Consortium. Extensible markup language (XML) 1.0 (second edition). http://www.w3.org/TR/2000/REC-xml-20001006, Oct. 2000.