Distributed Control Diffusion: Towards a Flexible Programming Paradigm for Modular Robots

Ulrik P. Schultz
Maersk Institute
University of Southern Denmark
ups@mmmi.sdu.dk

***Abstract*—A self-reconfigurable robot is a robotic device that can change its own shape. Self-reconfigurable robots are commonly built from multiple identical modules that can manipulate each other to change the shape of the robot. The robot can also perform tasks such as locomotion without changing shape. Programming a modular, self-reconfigurable robot is however a complicated task: the robot is essentially a real-time, distributed embedded system, where control and communication paths often are tightly coupled to the current physical configuration of the robot. To facilitate the task of programming modular, self-reconfigurable robots, we present the concept of *distributed control diffusion:* distributed queries are used to identify modules that play a specific role in the robot, and behaviors that implement specific control strategies are diffused throughout the robot based on these role assignments. This approach allows the programmer to dynamically distribute behaviors throughout a robot and moreover provides a partial abstraction over the concrete physical shape of the robot.**

We have implemented a prototype of a distributed control diffusion system for the ATRON modular, self-reconfigurable robot. The prototype relies on a simple virtual machine with a dedicated instruction set, allowing mobile programs to migrate between the modules that constitute a robot. Through a number of simulated experiments, we should how a single rule-based controller program implemented using distributed control diffusion can perform simple obstacle avoidance in a wide range of different car-like robots constructed using ATRON modules.

I. Introduction

A self-reconfigurable robot is a robot that can change its own shape. Self-reconfigurable robots are built from multiple identical modules that can manipulate each other to change the shape of the robot [4], [10], [12], [15], [18], [23]. The robot can also perform tasks such as locomotion without changing shape. Changing the physical shape of a robot allows it to adapt to its environment, for example by changing from a car configuration (best suited for flat terrain) to a snake configuration suitable for other kinds of terrain.

Programming self-reconfigurable robots is complicated by the need to (at least partially) distribute control across the modules that constitute the robot and furthermore to coordinate the actions of these modules. Algorithms for controlling the overall shape and locomotion of the robot have been investigated (e.g. [5], [21]), but the issue of providing a high-level programming platform for developing controllers remains largely unexplored. Moreover, constraints on the physical size and power consumption of each module limits the available processing power of each module. A typical consequence of the limited computational resources of each module is that the behavior of each module is programmed statically before deployment, resulting in a paradoxical scenario where the hardware can evolve dynamically through self-reconfiguration but the software cannot.

In this paper, we present a novel approach to programming modular, self-reconfigurable robots, based on the concept of *distributed control diffusion:* distributed queries are used to identify modules that play a specific role in the robot, and behaviors that implement specific control strategies are diffused throughout the robot based on these role assignments. In more detail, queries on the physical structure of the robot allows the program to identify specific parts of the robot such as "wheel", "leg", or "arm", independently of the configuration of the rest of the robot. The result of a query is to assign roles to selected modules. A role can directly activate preprogrammed behaviors in a module, similar to rule-based programming. Moreover, a role serves as an addressing mechanism when diffusing behaviors throughout the robot in the form of mobile programs. In effect, this allows new behaviors to be dynamically installed in a running robot, enabling dynamic software update scenarios and facilitating interactive experimentation with controllers. The concept of distributed control diffusion is based on the concept of directed data diffusion, as known from sensor networks [8], [9], albeit heavily adapted to take the control aspect of robots into account.

We have implemented a prototype of distributed control diffusion for the ATRON modular, self-reconfigurable robot [10], [11]. Our implementation is based on a simple virtual machine, which includes a bytecode interpreter as well as a simple network stack, memory manager, and task scheduler. The network stack provides compass and spatial coordinate information throughout the robot and implements broadcast communication for arbitrary module configurations without the assumption of having a unique identifier for each module. The virtual machine runs on the physical ATRON modules, the complete controller program (including hardware interface libraries and infrared communication stacks etc.) consumes less than 2K of RAM and less than 20K of program memory. Complete experiments with locomotion, obstacle avoidance, and basic self-reconfiguration have been performed in simulation.

ROBOCOMM 2007, 14th – 16th Oct 2007, Athens, Greece.

ISBN 978-963-9799-08-0
DOI 10.4108/ICST.ROBOCOMM2007.2197

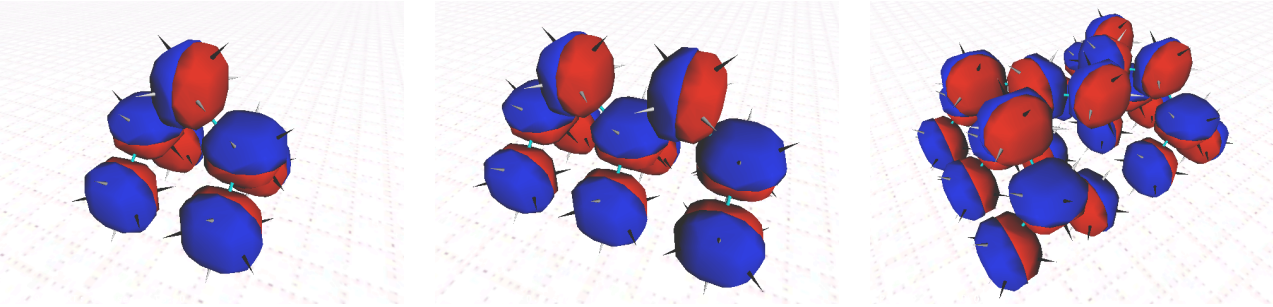

Fig. 2. Simulated car configurations: basic 4-wheels, long 6-wheels, and collaborating 12-wheels.

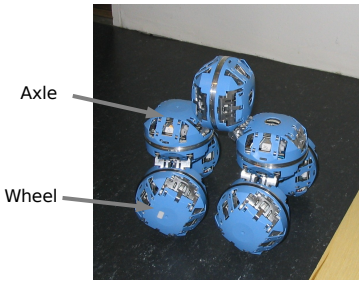

Fig. 1. Basic car configuration, physical modules

A. The ATRON Self-Reconfigurable Robot

Outline

The rest of this paper is organized as follows. First, Section II provides background information on self-reconfigurable robots. Next, Section III presents the concept of distributed control diffusion, Section IV describes various implementation issues, and Section V details our experiments. Last, Section VI presents related work, and Section VII concludes.

II. Background: Self-Reconfigurable Robots

The ATRON self-reconfigurable robot is a 3D lattice-type robot [10], [11]. An ATRON module is spherical, is composed of two hemispheres, and can actively rotate the two hemispheres relative to each other. A module may connect to neighbor modules using its four actuated male and four passive female connectors. The connectors are positioned at 90 degree intervals on each hemisphere. Eight infrared ports, one below each connector, are used by the modules to communicate with neighboring modules and sense distance to nearby obstacles or modules. A module weighs 0.850kg and has a diameter of 110mm. Currently 100 hardware prototypes of the ATRON modules exist. Motion constraints on the modules affect their ability to self-reconfigure. The single rotational degree of freedom of a module makes its ability to move very limited: in fact a module is unable to move by itself. The help of another module is always needed to achieve movement. All modules must also always stay connected to prevent modules from being disconnected from the robot. They must avoid collisions and respect their limited actuator strength: one module can lift two others against gravity. As a concrete example, consider the robots shown in Figures 1 and 2, which are different configurations for car-like robots, physical and simulated. These robots have a number of "wheels": modules that are in contact with the floor, that can turn freely, and that are aligned in the same direction. Turning the wheels allows a program to propel the robot in a given direction. The 12-wheeled "joined" configuration simulates two cars that have been linked by a bridge and must coordinate their movements.

Other examples of self-reconfigurable robots include the M-TRAN and the SuperBot self-reconfigurable robots [12], [18]. These robots are similar from a software point of view, but differ in mechanical design e.g. degrees of freedom per module, physical shape, and connector design. This means that algorithms controlling change of shape and locomotion often will be robot specific, however general software principles are more easily transferred.

A. Programming self-reconfigurable robots

General approaches to programming the self-reconfigurable ATRON robot include gradients, metamodules, and rule-based control [1], [5], [13], [14]; we return to some of these later in the context of related work. In the context of this article, we are more interested in role-based control, which is a generalization of rule-based control. Role-based control is an approach to behavior-based control for modular robots where the behavior of a module is derived from its context [20], [22]. The behavior of the robot at any given time is driven by a combination of sensor inputs and internally generated events. Roles allow modules to interpret sensors and events in a specific way, thus differentiating the behavior of the module according to the concrete needs of the robot.

All ATRON modules are equipped with numerous infrared sensors allowing them to sense nearby objects.[1] In the car example, all modules could collect information about their immediate surroundings, for example allowing them to detect presence and dimensions of obstacles. Aggregating this sensor information and using it to control the overall behavior of the robot is one of the key challenges in programming the ATRON system. Moreover, coordinating the movement of the

[1]Note that due to implementation issues, the infrared proximity sensors are currently only available in the simulator.

individual modules is a key challenge; the wheels and axles must for example work together to move the car.

III. Directed Control Diffusion

A. General principles

Directed control diffusion allows the programmer to abstract over the physical configuration of a modular robot. The key idea is to assign a behavioral role to each module depending on the properties of the module, including its physical position, current behavior, and connectivity to other modules. A role can activate selected behaviors stored natively in a module, or be used to identify the module as a target for behavior that is diffused throughout the robot using mobile code.

Modular robots often have a limited computational power, due to constraints on physical size, price, and battery power. For this reason, a general-purpose virtual machine such as the Java Virtual Machine cannot realistically be used for executing mobile code inside the robot. We propose to use an approach based on domain-specific languages, where the virtual machine is tailored to the problem domain. Specifically, the instruction set of the virtual machine is dedicated to the characteristics modular robot system, and can even be extended to support the specific application area being targeted by the programmer. Moreover, to facilitate role selection, the virtual machine maintains information about the local configuration of the robot and what roles are currently active.

Directed control diffusion does not explicitly target the issue of self-reconfiguration. Self-reconfiguration can be done using behaviors distributed to modules that play a specific role, but this is orthogonal to the principle of directed control diffusion. Nevertheless, self-reconfiguration can change the properties of a module, such as its physical position or connectivity. Thus, self-reconfiguration should in some cases trigger a new assignment of roles throughout the robot, potentially changing the role of all the modules that constitute the robot; we currently require the programmer to trigger this role reassignment manually based on knowledge of when the robot is in a stable configuration where role assignment is meaningful.

B. Directed control diffusion on ATRON

We have implemented directed control diffusion for the ATRON modular, self-reconfigurable robot. Each module is programmed with identical controller programs that provide a virtual machine with a network stack and a bytecode interpreter. Throughout this paper, we refer to our virtual machine implementation as the ATRON DCD-VM. The overall functionality of the virtual machine is illustrated in Figure 3.

The network stack supports messages containing notifications of role changes, control commands, mobile programs, and notification of self-reconfiguration. Notifications of neighboring role changes allows each module to maintain information about the local configuration. A control command activates a specific functionality in a module, either a generic functionality such as moving an actuator, a functionality implemented by a bytecode program loaded on the module, or a domain-specific functionality written in native code and stored in the module with the interpreter. All messages carry a *context*, a three-dimensional coordinate and compass direction relative to the origin of the message; coordinates and compass direction are automatically updated before transmission to another module. Mobile programs maintain the origin information when moving from one module to another, and pass this origin information when spawning new mobile programs or sending out commands. Thus, if a single module initiates a set of behaviors in the robot by distributing mobile code throughout the robot, all these behaviors will share a common coordinate system and compass direction. Control commands and mobile programs are not executed directly when they are received, but are enqueued as tasks that are executed in turn. Last, notifications of self-reconfiguration are simply used to reset the context of the module, to facilitate reprogramming the module for a new usage context.

The bytecode interpreter is used to execute bytecode programs; mobile programs are executed when they move onto a module, but a bytecode program can also store a new control command for a specific role, install an event handler for a sensor, or schedule itself for re-execution (thus implementing a recurring behavior). The interpreter has instructions for querying the module for its compass direction and position in the coordinate system, the connectivity to other modules, the role of the module and the modules that it is directly connected to, as well as reading the sensors of the module such as the accelerometer and the joint position. Moreover, instructions allow the program to assign a new role to the module and issue control commands.

C. Query and command language

Modules are programmed using a bytecode language; the development of one or more high-level languages is considered future work. The semantics of the bytecode language are as follows. A bytecode program always executes in a context that defines the compass direction and coordinates relative to its origin (the module that first evaluated the program). The context also defines a fixed-size stack for storing intermediate results, e.g., a conditional branch will pop the value at the top of the stack and jump to the specified instruction number if the value is non-zero.

Instructions can be divided into five categories: general-purpose instructions, queries, commands, and program control. General-purpose instructions include conditional branching, logical operators, an equality operator, and pushing a constant onto the stack. Query instructions inspect the context and the physical state of the module, as described earlier. Command instructions allow the program to control the actuators of the module, for example by turning the main joint or opening a connector. Program control instructions can terminate a program or migrate a program to other modules. Moreover, program control instructions can also install bytecode sequences carried inside the program as new control commands for a specific role or as handlers for specific events, such as proximity detection on a given connector.

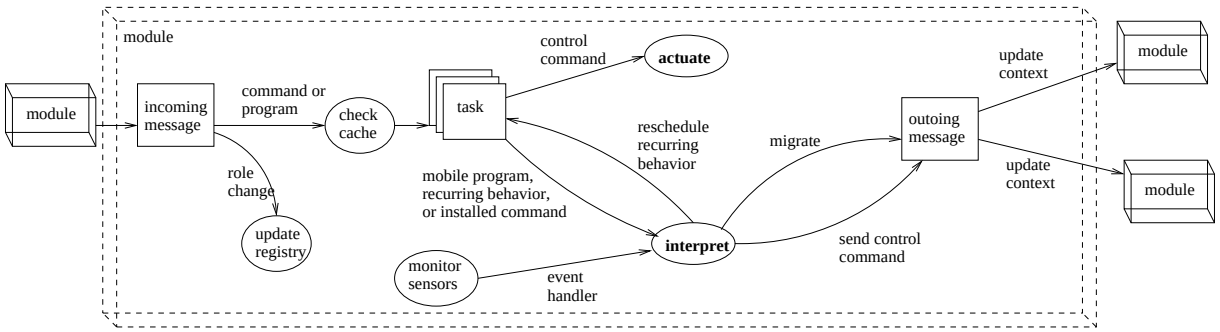

Fig. 3. The ATRON DCD-VM: messages are received, enqueued as tasks, and processed either by actuating the module or by the interpreter

The values manipulated by the instructions are always 8-bit bytes, either signed or unsigned. Signed bytes are used to represent integers, whereas unsigned bytes are used to represent connector sets: there are exactly 8 connectors on an ATRON module, so an 8-bit bit-vector can be used to e.g. represent the set of connectors that are connected to another module.

A common complication when working with the ATRON robot is to know what connector to use for actuation or communication. To facilitate programming controllers, we use the concept of a *virtual connector:* using the spatial coordinate and rotational information stored in the context, the physical connectors can be assigned virtual connector numbers based on how the module is physically rotated. For example, for a module with a north-south rotation axis, virtual connector number 2 is always upwards and facing forwards.

D. Communication

Messages containing mobile programs and control commands are dispersed throughout the robot using broadcast communication. Broadcast communication is however non-trivial to implement for arbitrary structures of ATRON modules. Messages that are received on one connector must be retransmitted on those connectors where modules are present, but structures may be cyclic and a given message must only be delivered once to each module. Deciding if a message has already been seen is critical for avoiding endless loops of message retransmissions, but is non-trivial given that the number of modules may be arbitrarily large and that messages may arrive out-of-order. Moreover, it is advantageous if message routing does not rely on each module having a unique identifier, since this improves scalability and facilitates combining arbitrary structures to form new robots.

As messages containing mobile programs or control commands are transmitted throughout the robot, the network stack maintains the context of the message: the spatial coordinate and compass direction relative to the origin of the message. The spatial coordinate supports querying the robot configuration, but also enables the network stack to discard messages that it has already seen, since the origin of the message is identical no matter what route the message has been transmitted through the robot. In more detail, on the sender side, each module maintains a counter that is incremented when a new message is generated, and the value of this counter is included in the message. On the receiver side, each module maintains a cache of messages that it has seen. The cache is indexed by the spatial coordinate stored in the message (and hence implicitly by the origin of the message), and uses a sliding bit-vector to keep track of what message counters have been received from each origin. When full, the cache uses a first-in first-out discipline to discard entries.

The size of the message cache is critical for avoiding repeated messages. If the cache size is greater than or equal to the number of modules in the robot, repeated delivery of messages can in most cases be avoided. The sliding bit-vector allows each module to keep track of some number of messages from each of the other modules, but since messages containing a mobile program only are retransmitted when they explicitly execute the instruction `MIGRATE_CONTINUE`, the delay between receiving two copies of a message can be arbitrarily long. If the message counter in an incoming message is outside the sliding bit-vector, a heuristic is used to determine if the message is old, and should be discarded, or is new, and should cause the bit-vector to slide to accommodate the new message counter. If the heuristic fails, messages may be lost in the former case or duplicated in the latter case. In our experiments, a message cache of 16 entries taking up 144 bytes in total size was sufficient, but depending on the expected size of the robot and the complexity of the communication, the parameters of the cache may need to be optimized to avoid communication problems.

E. Example

We now demonstrate how distributed control diffusion can be used to write a single, general controller program that can be dynamically deployed to robots that have wheels (see Figure 2 earlier in the paper). For readability, we use a C-like syntax to represent the bytecode programs, even though no compiler from this syntax to bytecode has been implemented. As an example, we will implement a controller that causes any car robot (see definition below) to move forwards until it encounters an obstacle, at which time the robot should stop. More elaborate examples can be found in Section V.

```
void find_wheels() {
 if(center_position()==EAST_WEST
  && size(connected())==1
  && size(connected_dir(UP))==1) {
  if(size(connected_dir(EAST))==1)
   set_role_notify(RIGHT_WHEEL);
  else
   set_role_notify(LEFT_WHEEL);
 } else
   migrate_continue();
}
```

```
void install_obstacle_avoidance() {
 if(instanceof(role(),RIGHT_WHEEL)
  && get_module_y()>0)
  install_handler(PROXIMITY_5, {
   send_command(WHEEL,ROTATE_STOP,0);
   clear_handler(PROXIMITY_5);
  });
 else
  migrate_continue();
}
```

Fig. 4. Program fragments for finding wheels (left) and for installing obstacle avoidance handlers (right).

The controller is implemented using a combination of mobile programs and control commands. Two of the mobile programs are shown in Figure 4. First, a query mobile program (shown in Figure 4, left) is used to identify the wheels in the robot. For simplicity, any module with a rotational axis perpendicular to the direction we wish to go in can be considered a wheel if it only has a single, upwards connection. (For the ATRON, a single upwards connection means that the other hemisphere is free to rotate and hence can act as a wheel.) Note that we assume that the robot has been configured with car-motion as a purpose: we do not detect any orthogonally aligned modules that may cause friction when moving forward, and free-hanging modules that cannot reach the surface are still considered wheels. The mobile program queries the position and connectivity properties of the module, and sets the role to either left or right wheel, as appropriate. When setting the role, any neighboring modules are notified of the role change, facilitating queries that include the role of neighboring modules. (For example, an axle has a wheel as a neighbor.)

Once the wheels have been identified, appropriate control commands turning the main actuator in either direction can be sent to the left and right wheels, respectively. Moreover, event handlers for detecting obstacles using the proximity sensors can be installed in the front modules of the robot. Installation of event handlers is done by another mobile program (shown in Figure 4, right). Note that for simplicity, we assume that the origin of the program is located at the center of the robot, and moreover we install proximity sensor handlers in all modules in the front part of the robot, relative to the origin module. Installing event handlers in modules inside the robot is appropriate for the simulator but would most likely be problematic in practice. The handler is installed for the sensor attached to a particular connector, corresponding to the front side of the module. The action to perform when the event is triggered is included in the program, and is stored in the module after the program terminates. When the event is triggered, the module that detected the obstacle sends out a stop command to all wheels in the robot. Thus, the controller has effectively been distributed to the relevant modules of the robot.

F. Discussion

Directed control diffusion provides a flexible means to programming a modular robot, but also has a number of liabilities in terms of scalability. Scalability of communication is an issue since we employ broadcast communication to distribute code and invoke commands using the role-based addressing scheme. A convenient solution to this issue is to use the role information contained in a message to route the message only to those modules that are in the target role, similarly to how gradients are formed in sensor networks and used to efficiently route data to a central sink [8], [9]. The implementation of such a role-based routing scheme is however considered future work.

Scalability in terms of behavior is another issue since the proposed bytecode language by design is intended to be simple and thus most likely will be inappropriate for expressing controllers for realistic scenarios involving physical hardware. As a solution we propose a generative approach where new functionality is prototyped using bytecodes but can subsequently be compiled to C code that becomes available as a new bytecode instruction. This way, prototype functionality can be dynamically deployed to the robot (facilitating experimentation), whereas stable code is eventually implemented in C, thus allowing a general-purpose language to be used to enrich the initial implementation with richer functionality. The link between bytecodes and more stable code in C is that the C code becomes available as a new bytecode instruction; configuration is currently done manually but should be supported by appropriate compilation tools.

IV. Implementation

A. Software architecture

The ATRON DCD-VM is structured in terms of a number of logical subsystems: an interpreter, a scheduler, a communication stack, and a memory manager. The interpreter is implemented as a simple switch-case interpreter. Safety checks ensure that stack overflow/underflow is not possible, and that it is not possible to set the program counter outside the program's instruction range. Essentially, we see the interpreted language as a scripting language that only performs simple operations, for which reason execution speed is not currently a significant issue.

All incoming communication is processed by an interrupt handler in the communications stack. To allow communication to start long-running operations such as interpreting a program, we use a scheduler to execute tasks sequentially outside the interrupt handler. The scheduler simply uses a fixed-size first-in first-out queue of statically allocated task structures. A task can currently either be a program to interpret or a control command to execute. A task can reschedule itself, meaning that it will be re-executed again later, but support for priorities and timers is considered future work.

Incoming mobile programs are processed as follows. First, the program is received as a data packet by the communications stack, which inspects the header of the packet to determine that it is a mobile program. The packet includes both the context (spatial coordinates etc.) and the bytecode program. The bytecode information is then stored in a program slot by the memory manager; the memory manager simply provides a fixed number of fixed-size program slots for storing incoming programs. Program slots are ordinarily freed when a program has finished execution; this is however not the case e.g. for event handlers. If no free program slots are available, an error is signaled and the incoming program is discarded. After storing the program, a task is created that includes the context and the index of the program slot. When the task is scheduled, the interpreter extracts the information and executes the program.

B. Instruction set

To reduce the size of bytecode programs, all commonly used instructions have been specialized for their arguments. For example, to query for a module where the number of modules connected to the upwards connectors is equals to 2, the following instruction sequence can be used:

```
01: CONNECTED UP          /* 2 bytes */
03: SIZEOF                /* 1 byte */
04: EQUALS 2              /* 2 bytes */
```

The first instruction pushes the bit-set of upwards connected modules on the stack, the second instruction computes the size of the set, and the last instruction tests whether this value is equal to 2. This sequence takes up 5 bytes, but can also be represented by the following specialized instructions:

```
01: CONNECTED_UP_SIZEOF /* 1 byte */
02: EQUALS_2            /* 1 byte */
```

This sequence only takes up 2 bytes which saves space and can increase the execution speed. The amount of space that can be saved using specialized instructions is typically smaller than what was the case in this minimal example; in the experiment reported in Section V, the savings are around 10%.

C. Status of the implementation

We have implemented the ATRON DCD-VM using a prototype simulator for modular robots currently under development here at the Maersk Institute. The simulator is written in Java but supports controllers written both in Java and in C, the latter option being supported using JNI (the Java Native Interface). The DCD controller has been developed in C code with the hardware constraints of the ATRON modules in mind. An ATRON module has 4K of RAM, 4K of self-programmable EEPROM, and 128K of flash memory for storing the program. Currently, we do not make use of the EEPROM, but we speculate that it could be used for storing mobile programs. The total size of the DCD controller, including the complete ATRON library containing the infrared communication stack and functions for interfacing to actuators, is less than 20K of program memory. The statically used RAM size is less than 2K when using a 16-entry communication cache, 10-entry stack queue, and 5-entry program store; these values were appropriate for all experiments reported in this paper, but may need to be adjusted for more complex applications. The ATRON DCD-VM runs on the physical modules but is currently not stable enough on the physical hardware for large-scale experiments.

Program	Target role	Size (bytes)	Uncompressed size (bytes)
Find wheels	Any	24	31
Install proximity handler	L/R-Wheel	21	24
Find axles	Any	13	15
Install stop command	L/R-Wheel	34	35
Axle behavior	Axle	48	60

TABLE I
MOBILE PROGRAMS FOR CAR OBSTACLE AVOIDANCE

V. EXPERIMENTS

A. Methodology

All distributed experiments described in this paper currently are simulated. Moreover, dynamic installation of code from outside the robot is not currently supported either in the simulator or on the physical hardware. For this reason, we store the bytecode on a single module that then diffuses the behavior to the rest of the robot.[2] The orientation of this module thus decides the coordinate system and compass directions for the other modules.

B. Obstacle avoidance

As a more complete example of distributed control diffusion, we have implemented a simple obstacle avoidance controller for wheeled ATRON robots. This controller is a generalization of the example from Section III-E that works for all the simulated ATRON cars shown in Figure 2, as follows. The car moves forward until it encounters an obstacle, at which time it stops and starts to reverse while turning. Once the wheels have rotated three times, it stops again and resumes the initial behavior. This behavior is illustrated in Figure 5. The 12-wheeled car built by joining two 6-wheeled cars is difficult to turn, but does turn [in simulation] by making those

[2]Storing the bytecode in a single module is equivalent to reprogramming a single module, which provides a significant advantage compared to requiring the user to manually reprogram all modules in the robot.

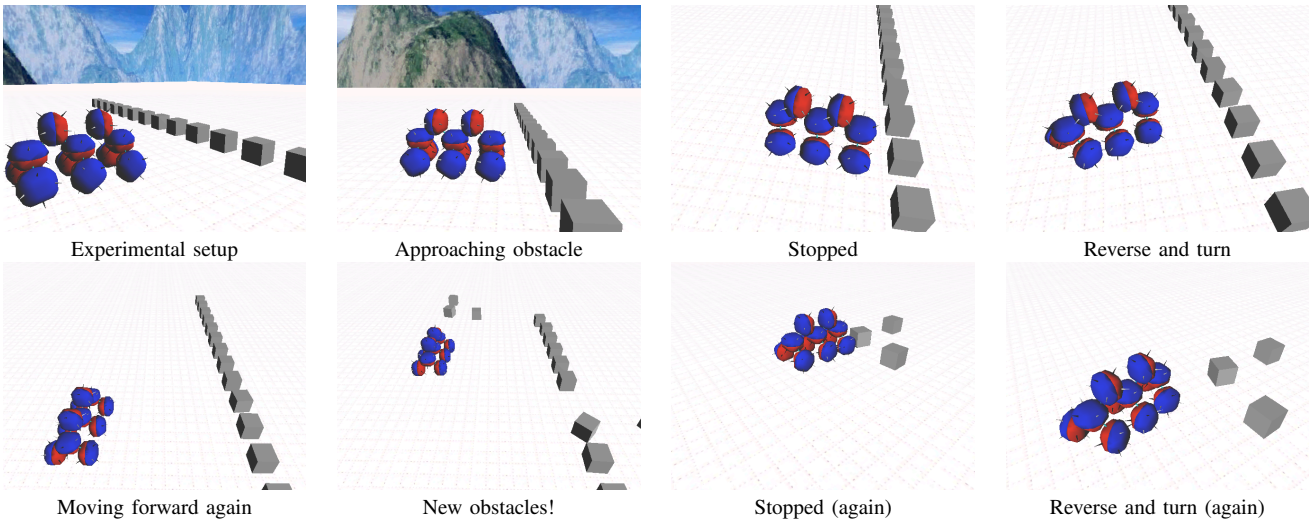

Fig. 5. Obstacle avoidance using generic distributed controller diffusion program. Note that some obstacles are moved manually during simulation.

wheels that are innermost in the turn more sharply than the outermost wheels.

The obstacle avoidance controller is implemented using seven different mobile programs that are diffused throughout the robot when the simulation starts. The programs are summarized in Table I along with the role they are targeted at and their sizes in bytes, using specialized and generic instructions respectively. The "find wheels" and "install proximity handler" programs are as in the example of Section III-E, except that the event handler sends out a user-defined command "stop" to all wheels rather than directly stopping the wheels. The "find axles" program locates axle modules by finding modules that are connected downwards to a module having any of the wheel roles (left wheel or right wheel). The "install stop command" program installs the "stop" command in the wheels. This command causes a wheel to assume a sub-role "reversing" and then to start to move in the reverse direction. Last, the axle behavior is to turn the axles so that the car moves straight when the wheels are in their standard roles, but to turn to an appropriate angle when the wheels assume a "reversing" role. The axle behavior uses the spatial coordinate of the module to differentiate its behavior, turning the front and back wheels symmetrically and turning the innermost wheels more sharply for wide car configurations such as the "collaborating cars" in Figure 2.

C. Self-Reconfiguration

To test directed control diffusion with self-reconfiguration, we have performed two simple experiments with the DCD-VM, as follows. The first experiment is to dynamically disrupt a structure after behaviors have been diffused, but without recomputing roles for each module. Concretely, we disrupt the 12-wheeled car by removing the bridge that connects the two 6-wheeled cars that make out the larger car. Removing the bridge is done simply by introducing a timer that disconnects the bridge modules after a fixed time period. The effect is that behavior is diffused in the 12-wheeled car but then has to keep functioning in each of the individual 6-wheel cars. Nonetheless, since the key principle of the DCD-VM is that behavior is distributed to all modules, each individual car still performs obstacle avoidance after the breakup, without requiring reprogramming. One car does however turn more sharply than the other, since the turning behavior was differentiated based on the spatial position of the modules.

The second experiment uses a 10-wheeled car built by attaching the 4-wheeled car to the end of the 6-wheeled car. When the 10-wheeled car meets an obstacle it will split into a 6-wheeled car and a 4-wheeled car; the 4-wheeled car is then reprogrammed be executing a dedicated "split" bytecode instruction. (Implementing complex behavior in dedicated bytecode instructions is central to our approach, as described in Section III-F.) In more detail, the controller used in the previous experiments is extended with a new behavior that causes the axle module that connects the 4-wheeled car to the 6-wheeled car to disconnect (the spatial position is used to identify this module). The disconnection happens when the wheels take on the "reversing" role, which happens when an obstacle is encountered. The module then executes the split bytecode which sends out a "reset" message to all modules in the 4-wheeled car (a standard broadcast is not used since it will also reach the 6-wheeled car so long as it is within communication distance). The split bytecode subsequently sends out the "find wheels" program from Table I and last sends appropriate rotation commands to the left and right wheels. The overall effect is that the 6-wheeled car continues performing the obstacle avoidance behavior whereas the 4-wheeled car takes on a new (albeit very simple) behavior of simply driving in a straight line. Note that this experiment requires the bytecode programs to be stored on one of the modules from the 4-wheeled car.

D. Assessment

The obstacle avoidance experiment demonstrates how distributed control diffusion is implemented on the DCD-VM.

Queries and the selection of roles is implemented using mobile programs that implicitly maintain coordinates, compass direction, and information about the local configuration, for example allowing the controller to identify those modules that can serve as wheels and axles. The axle turning behavior is implemented as a mobile program that always reschedules itself as its last instruction, thus providing a simple approach to programming behavior-based controllers [2]. Roles both serve to identify message receivers and to enable the module to respond to new commands. All programming is done dynamically while the robot is running, for example allowing the left sub-car in a 12-wheeled car configuration to activate roles and install behaviors on the modules in the right sub-car.

The simple reconfiguration experiments demonstrate some of the advantages of distributed control diffusion in self-reconfigurable robots. The first reconfiguration experiment shows that distributed control diffusion is robust to accidental reconfiguration such as module failure since it does not rely on any precomputed knowledge of the topology of the robot. The second reconfiguration experiment shows how a module structure can be reprogrammed after self-reconfiguration to obtain a new behavior.

Last, we note that all programs are smaller than 50 bytes, which facilitates transmitting them on the physical ATRON modules (which currently are sensitive to transmitting large amounts of data). The size of the largest programs could even be significantly reduced by replacing complex functionality implemented in bytecodes with single bytecodes implemented in C code.

VI. Related Work

A sensor network is a distributed, embedded system dedicated to gathering sensor information and conveying this information to a central unit. Network connections are typically wireless and ad-hoc, and price, size, and power constraints impose severe resource limitations on the individual sensor nodes and their network communication. For many applications of sensor networks, the exact identity of each sensor node and the routing of messages over specific ad-hoc connections is unimportant for the end goal of gathering specific data about a geographical location and assembling this data at a specific node (typically a server). This observation is the basis for the directed diffusion communication paradigm proposed by Chalermek et al [8], [9] (for clarity, we will refer to their technique as directed *data* diffusion). Using directed data diffusion, the programmer expresses an interest in a data source in terms of queries over required data properties, such as the physical location of the sensor. Queries form gradients in the network that implicitly connect data sources to data sinks, thus making it possible to route data packets based on the shortest path of the query from the sink to the source. Queries are broadcast throughout the network, and each node uses a cache to eliminate redundant messages. We have observed that many issues in programming self-reconfigurable robots are similar to the issues in programming sensor networks: a large number of low-end embedded systems connected using ad-hoc networking need to gather information using sensors and to convey this information to the appropriate party [16]. Moreover, dynamic software updates are in both cases critical to enable evolution of the system after it has been deployed. Distributed control diffusion has a strong resemblance to directed data diffusion from sensor networks. In both cases, a query mechanism is used as a means of identifying nodes (modules) for addressing. Nevertheless, distributed control diffusion is designed for controlling robots and thus has additional features for using actuators and allowing the robot to respond immediately to sensor inputs. In general, we expect that the synergy between modular robotics and sensor networks can benefit researchers from both fields.

Software architecture for modular robots has been investigated by Zhang et al by deploying a component infrastructure on the PolyBot self-reconfigurable robot [24], [25]. The component infrastructure is based on the *attribute/service model* where components acting either as attributes or services are distributed on a communication network that connects all the modules of the robot using a CAN bus. Here, attributes are distributed, thread-safe shared data repositories, whereas services are abstractions over hardware or software routines. Scalability to a large number of modules is achieved by using a dedicated CAN-bus protocol coupled with a generalized master/slave architecture where multiple masters control multiple slaves. Compared to the attribute/service model and multi-master/multi-slave model, distributed control diffusion is a lower level software layer providing flexible execution support. Indeed, the primary concern in our work is providing a means to identify what role a module should play and subsequently diffuse the required controller code to the all modules playing this role. As such, distributed control diffusion could probably provide a highly flexible platform for implementing an architecture similar to attribute/service and multi-master/multi-slave. Nevertheless, we note that since PolyBot modules are connected by a CAN bus, communication is significantly simplified compared to the strict module-to-module nature of communication in ATRON robots. Moreover, the available computational resources in the ATRON hardware are more constrained than those of PolyBot. For these reasons, the principles behind attribute/service and multi-master/multi-slave must be heavily adapted before they can be used on the ATRON hardware.

The experiments reported in this paper were performed using a combination of roles and rule-based control. The idea of role-based control is derived from Stoy et al where changes to behavior are driven by changes to the context, resulting in a robust and very flexible approach to controlling the CONRO reconfigurable, modular robot [20], [22]. Nevertheless, the only control examples investigated are cyclic, signal-driven behaviors for locomotion, whereas our work concerns runtime distribution of mobile code and a general-purpose concept of roles as a means for structuring arbitrary behaviors. The generality of our approach however comes at a price in terms of robustness, since there is no inherent tolerance to e.g. loss of communication or spurious reconfiguration. The use

of queries for selecting roles in the DCD-VM is similar to rule-based control [3], [7], [14], except that we use mobile code to install behaviors, potentially allowing arbitrary rules to be applied. The observation of Brandt and Ostergaard that more expressive means of writing rules leads to more compact and efficient rules [1] supports the DCD-VM approach of using a flexible bytecode instruction set to express conditions and actions. As an alternative to roles and rules, we believe control could be implemented using hormone-based control, as proposed by Shen et al [19]. Here, data packets referred to as *hormones* are broadcast through the robot structure, triggering various actions such as role assignment and actuation. Unlike distributed control diffusion, hormones do not contain program fragments, although this extension is listed as future work by Shen et al. We believe an approach to distributed control diffusion based on hormones would enhance robustness, for example reducing the consequences of packet loss.

VII. Conclusion and Future Work

In this paper, we have presented the principle of distributed control diffusion, where mobile code is used to query the physical characteristics of the robot structure to determine what role each part should play, and then subsequently install dedicated controller programs on each module depending on its role. We see the ATRON DCD-VM as a proof-of-concept implementation: virtual machines *can* be used on self-reconfigurable, modular robots to provide a higher level of abstraction and a higher degree of flexibility.

In terms of future work, we are interested in investigating high-level languages for programming the ATRON DCD-VM [6], [17], experimenting with hormone-based control as outlined in the previous section, and improving the efficiency of the communication stack, as described in Section III-F.

Acknowledgments

I would like to thank Nicolai Dvinge, David Christensen, David Brandt, Yves Demazeau, and Kasper Støy for inspirational discussions leading to the idea of distributed control diffusion. An additional special thanks goes to David Christensen for proofreading and last-minute helpful insights into behavior-based control.

References

[1] D. Brandt and E.H. Ostergaard. Behaviour subdivision and generalization of rules in rule based control of the ATRON self-reconfigurable robot. In *Proceeding of the International Symposium on Robotics and Automation (ISRA)*, pages 67–74, Queretaro, Mexico, September 2004.

[2] R. Brooks. A robust layered control system for a mobile robot. *IEEE Journal of Robotics and Automation*, 2:14–23, March 1986.

[3] Z. Butler, K. Kotay, D. Rus, and K. Tomita. Generic decentralized control for a class of self-reconfigurable robots. In *Proceedings, IEEE International Conference on Robotics and Automation (ICRA'02)*, pages 809–815, Washington, DC, USA, 2002. IEEE Press.

[4] A. Castano and P. Will. Autonomous and self-sufficient conro modules for reconfigurable robots. In *Proceedings of the 5th International Symposium on Distributed Autonomous Robotic Systems (DARS)*, pages 155–164, Knoxville, Texas, USA, 2000.

[5] D.J. Christensen and K. Støy. Selecting a meta-module to shape-change the ATRON self-reconfigurable robot. In *Proceedings of IEEE International Conference on Robotics and Automations (ICRA)*, pages 2532–2538, Orlando, USA, May 2006.

[6] Nicolai Dvinge, Ulrik P. Schultz, and David Christensen. Roles and self-reconfigurable robots, in *Proceedings of the^{2nd} Workshop on Roles and Relationships in Object Oriented Programming, Multiagent Systems, and Ontologies Workshop (co-located with ECOOP 2007 Berlin)*. Technical Report 2007-9, Technical University of Berlin, 2007.

[7] T. Fukuda and S. Nakagawa. Method of autonomous approach, docking and detaching between cells for dynamically reconfigurable robotic system CEBOT. *JSME International Journal*, 33(2):263–268, 1990.

[8] Chalermek Intanagonwiwat, Ramesh Govindan, and Deborah Estrin. Directed diffusion: a scalable and robust communication paradigm for sensor networks. In *MobiCom '00: Proceedings of the 6th annual international conference on Mobile computing and networking*, pages 56–67, New York, NY, USA, 2000. ACM Press.

[9] Chalermek Intanagonwiwat, Ramesh Govindan, Deborah Estrin, John Heidemann, and Fabio Silva. Directed diffusion for wireless sensor networking. *IEEE/ACM Trans. Netw.*, 11(1):2–16, 2003.

[10] M. W. Jorgensen, E. H. Ostergaard, and H. H. Lund. Modular ATRON: Modules for a self-reconfigurable robot. In *Proceedings of IEEE/RSJ International Conference on Robots and Systems (IROS)*, pages 2068–2073, Sendai, Japan, September 2004.

[11] H.H. Lund, R. Beck, and L. Dalgaard. Self-reconfigurable robots with ATRON modules. In *Proceedings of 3rd International Symposium on Autonomous Minirobots for Research and Edutainment (AMiRE 2005)*, Fukui, 2005. Springer-Verlag.

[12] S. Murata, E. Yoshida, K. Tomita, H. Kurokawa, A. Kamimura, and S. Kokaji. Hardware design of modular robotic system. In *Proceedings of the IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 2210–2217, Takamatsu, Japan, 2000.

[13] Esben H. Ostergaard. Efficient distributed "hormone" graph gradients. In *Proceedings of Nineteenth International Joint Conference on Artificial Intelligence (IJCAI)*, pages 1489–1495, Edinburgh, Scotland, July 2005.

[14] Esben H. Ostergaard and Henrik H. Lund. Distributed cluster walk for the atron self-reconfigurable robot. In *Proceedings of the The 8th Conference on Intelligent Autonomous Systems (IAS-8)*, pages 291–298, Amsterdam, Holland, March 2004.

[15] D. Rus and M. Vona. Crystalline robots: Self-reconfiguration with compressible unit modules. *Journal of Autonomous Robots*, 10(1):107–124, 2001.

[16] U. Schultz, K. Støy, N. Dvinge, and D. Christensen. Sensor networks and self-reconfigurable robots, October 2006. Position paper at the OOPSLA'06 Workshop on Building Software for Sensor Networks.

[17] U.P. Schultz, D. Christensen, and K. Støy. A domain-specific language for programming self-reconfigurable robots, October 2007. Accepted at the APGES'07 Workshop (co-located with GPCE 2007 Salzburg).

[18] W.-M. Shen, M. Krivokon, H. Chiu, J. Everist, M. Rubenstein, and J. Venkatesh. Multimode locomotion via superbot robots. In *Proceedings of the 2006 IEEE International Conference on Robotics and Automation*, pages 2552–2557, Orlando, FL, 2006.

[19] Wei-Min Shen, Yimin Lu, and Peter Will. Hormone-based control for self-reconfigurable robots. In *AGENTS '00: Proceedings of the fourth international conference on Autonomous agents*, pages 1–8, New York, NY, USA, 2000. ACM Press.

[20] Kasper Stoy, Wei-Min Shen, and Peter Will. Using role based control to produce locomotion in chain-type self-reconfigurable robots. *IEEE Transactions on Robotics and Automation, special issue on self-reconfigurable robots*, 2002.

[21] K. Støy. How to construct dense objects with self-reconfigurable robots. In *Proceedings of European Robotics Symposium (EUROS)*, pages 27–37, Palermo, Italy, May 2006.

[22] K. Støy, W.-M. Shen, and P. Will. Implementing configuration dependent gaits in a self-reconfigurable robot. In *Proceedings of the 2003 IEEE international conference on robotics and automation (ICRA'03)*, pages 3828–3833, Tai-Pei, Taiwan, September 2003.

[23] M. Yim, D. Duff, and K. Roufas. Polybot: A modular reconfigurable robot. In *Proceedings of the IEEE International Conference on Robotics and Automation (ICRA)*, pages 514–520, San Francisco, CA, USA, 2000.

[24] Y. Zhang, K. Roufas, and M. Yim. Software architecture for modular self-reconfigurable robots. In *IEEE/RSJ International Conference on Intelligent Robots and Systems*, Hawaii, 2001.

[25] Y. Zhang, M. Yim, K. Roufas, and C. Eldershaw. Attribute/service model: Design patterns for distributed coordination of sensors, actuators and tasks. In *Workshop on Embedded Systems Codesign*, 2002.