A Domain-Specific Language for Programming Self-Reconfigurable Robots

Schultz, Ulrik Pagh; Christensen, David Johan; Støy, Kasper

Published in:
APGES 2007 - Automatic Program Generation for Embedded Systems - Workshop Proceedings

Publication date:
2007

Document Version
Publisher's PDF, also known as Version of record

A Domain-Specific Language for Programming Self-Reconfigurable Robots

Ulrik P. Schultz, David Christensen, Kasper Støy
University of Southern Denmark

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 have developed a declarative, role-based language that allows the programmer to define roles and behavior independently of the concrete physical structure of the robot. Roles are compiled to mobile code fragments that distribute themselves over the physical structure of the robot using a dedicated virtual machine implemented on the ATRON self-reconfigurable robot.

1. 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, 9, 11, 14, 16, 18, 24, 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 however 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.

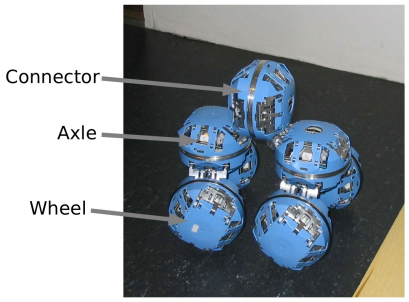


Figure 1: The ATRON self-reconfigurable robot. Seven modules are connected in a car-like structure.

In this paper, we present a role-based approach to programming a controller for a distributed robot system independently of the concrete physical structure of the robot. A role defines a specific set of behaviors for a physical module that are activated when the structural invariants associated with the role are fulfilled. Using the principle of distributed control diffusion [17], the roles are compiled into code fragments that are dynamically diffused throughout the physical structure of the robot and activated where applicable. Our programming language targets the distributed control diffusion virtual machine (DCD-VM) running on the ATRON modular, self-reconfigurable robot [11, 13]. Although the compiler implementation is still preliminary, it is capable of generating code for a complex example involving multiple roles.

The rest of this paper is organized as follows. First, Section 2 presents the ATRON hardware, discusses issues in programming the ATRON robot, and describes the DCD-VM. Then, Section 3 presents the main contribution of this paper, a high-level role-based programming language for the DCD-VM. Last, Section 4 presents related work and Section 5 concludes and outlines directions for future work.

2. THE ATRON SELF-RECONFIGURABLE ROBOT

2.1 Hardware

The ATRON self-reconfigurable robot is a 3D lattice-type robot [11, 13]. Figure 1 shows an example ATRON car robot built from 7 modules. Two sets of wheels (ATRON mod-

ules with rubber rings providing traction) are mounted on ATRON modules playing the role of an axle; the two axles are joined by a single module playing the role of "connector." As a concrete example of self-reconfiguration, this car robot can change its shape to become a snake (a long string of modules); such a reconfiguration can for example allow the robot to traverse obstacles such as crevices that cannot be traversed using a car shape.

An ATRON module has one degree of freedom, 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. 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. A module has 128K of flash memory for storing programs and 4K of RAM for use during execution of the program.

Other examples of self-reconfigurable robots include the M-TRAN and the SuperBot self-reconfigurable robots [14, 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.

2.2 Software

Programming the ATRON robot is complicated by the distributed, real-time nature of the system coupled with limited computational resources and the difficulty of abstracting over the concrete physical configuration when writing controller programs. General approaches to programming the self-reconfigurable ATRON robot include meta-modules [5], motion planning and rule-based programming. In the context of this article, we are however interested in role-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 [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.

2.3 Distributed control diffusion

To enable dynamic deployment of programs on the ATRON robot, we have developed a virtual machine that enables small bytecode programs to move throughout a structure of ATRON modules [17]. The virtual machine supports a concept we refer to as *distributed control diffusion:* controller code is dynamically deployed to those modules where a specific behavior is needed. The virtual machine, named DCD-VM, has an instruction set that is dedicated to the ATRON hardware and includes operations that are typically required in ATRON controllers. For example, the virtual machine maintains an awareness of the compass direction of each module and the roles of its neighbors, and specific instructions allow this information to be queried. Moreover, the virtual machine provides a lightweight and highly scalable broadcast protocol for distributing code throughout the structure of ATRON modules, making the task of programming controllers that adapt to their immediate surroundings significantly easier.

The DCD-VM supports a basic notion of roles to indicate the state of a module and to provide polymorphic dispatching of remote commands between modules, but at a very low level of abstraction. There is no explicit association between roles and behaviors, this currently has to be manually implemented by the programmer. Moreover, initial experiments with the virtual machine were performed by writing bytecode programs by hand, since no higher-level language (not even an assembly language) was available. To improve the situation, a high-level language for programming the DCD-VM has been developed, which is the subject of this paper.

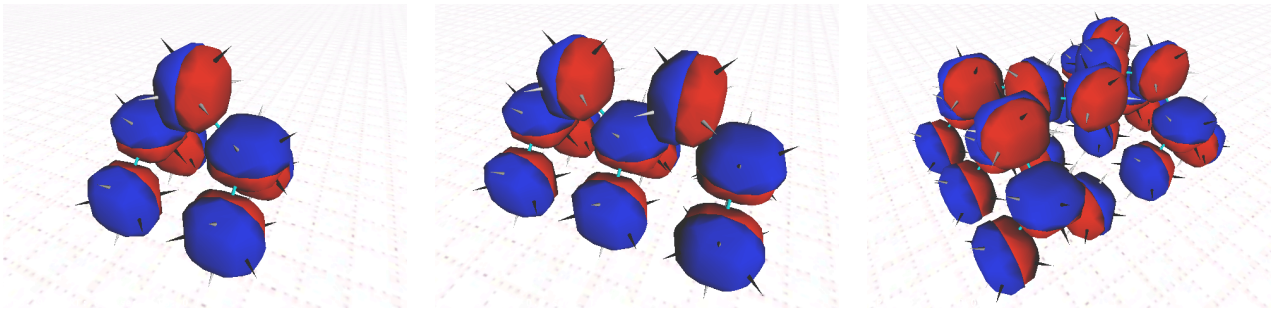
3. A HIGH-LEVEL ATRON PROGRAMMING LANGUAGE

3.1 Motivating example: obstacle avoidance

As a motivating example, consider a simple obstacle avoidance scenario where a car (such as the one shown in Figure 1 in the introduction) is moving forwards until it detects an obstacle using the forward proximity sensors of the front-most modules. In this case it reverses while turning, and then continues moving forwards. There are however many ways of making a car from ATRON modules, as shown in Figure 2: the car can be made longer (although more than 6 wheels makes turning impractical) and we can imagine two ATRON cars joining up in the field to create a more powerful vehicle. A controller that was programmed independently of the concrete physical configuration of the robot would solve many of these issues, and we describe how that can be done using distributed control diffusion, as follows.

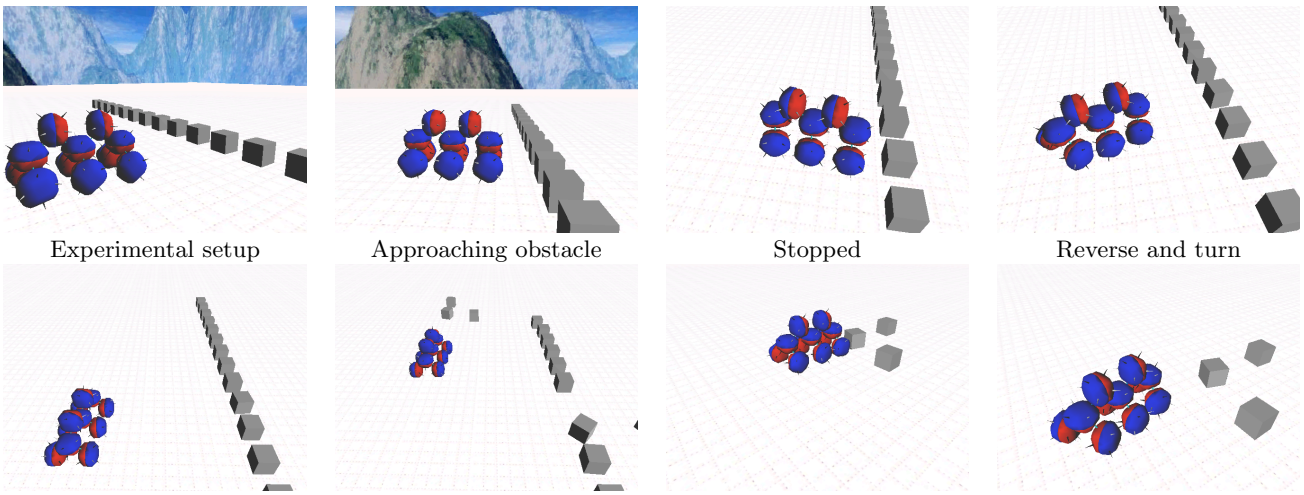
First, a query mobile program is used to identify the wheels in the robot. For simplicity, any module with a rotational axis perpendicular to the direction we wish to go in can be considered a wheel when it only has a single, upwards connection. (For the ATRON, a single upwards connection means that 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 wheel" 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,



Basic car | Long car | Collaborating cars

Figure 2: Different car configurations (simulated)



Experimental setup | Approaching obstacle | Stopped | Reverse and turn

Moving forward again | New obstacles! | Stopped (again) | Reverse and turn (again)

Figure 3: Obstacle avoidance using generic distributed controller diffusion program.

event handlers for detecting obstacles using the proximity sensors are installed in the front wheels of the robot using another mobile program. When the event is triggered, the module that detected the obstacle sends out a "reverse" command to all wheels in the robot. This way, the controller has effectively been distributed to the relevant modules of the robot. Before the wheels start reversing the role is changed to a "reversing wheel," which is observed by the axle behavior. The axles then turn an appropriate number of degrees to make the car change orientation while reversing. Once the wheels have finished reversing, they return to their respective forwards-moving roles, and the axles react accordingly by returning to the original position.

3.2 The RDCD language

The Role-based Distribution Control Diffusion (RDCD) language provides roles as a fundamental abstraction to structure the set of behaviors that are to be diffused into the module structure. Diffusion of code is however implicit: RDCD is a declarative language that allows roles to be assigned to specific modules in the structure based on invariants; behaviors are implicitly distributed to modules based on their association with roles. RDCD is a domain-specific language in the sense that it is targeted to the ATRON robots and moreover has very limited support for general-purpose computation. RDCD provides primitives for simple decision-making, but all complex computations must be performed in external code.

Our compiler currently does not have a parser and must therefore be given an abstract syntax tree constructed manually. Nevertheless, to present the RDCD language, we show the proposed BNF for RDCD in Figure 4 (non-terminals are written using italicized capitals, concrete syntax in courier font). An RDCD program declares a number of roles. A role normally extends a super-role meaning that it inherits all the members of the super-role; the common super-role `Module` defines the capabilities of all modules. A role can be concrete or abstract, with the usual semantics: all abstract members must be overridden by concrete members for a role be usable at runtime. A role declares a number of members in the form of constants, invariants, and methods. There currently is no explicit notion of state, so state must be represented using external code and accessed using functions. A constant can be concrete or abstract, and always defines an 8-bit signed value. For a module to play a given role, all invariants declared by the role must be true (in case of conflicts between roles, the choice of role is undefined). An invariant is simply a boolean expression over constants and functions.

Methods are used to define behavior that is active when

```
PROGRAM     ::= ROLE* DEPLOYMENT
ROLE        ::= abstract? role NAME extends NAME { MEMBER* }
              | role NAME modifies NAME { MEMBER* }
MEMBER      ::= CONSTANT | INVARIANT | METHOD
CONSTANT    ::= abstract NAME | NAME := VALUE
INVARIANT   ::= EXP ;
METHOD      ::= MODIFIER* NAME () BLOCK
BLOCK       ::= { STATEMENT* }
STATEMENT   ::= ϵ | FUNCTION ; | if( EXP ) { STATEMENT* } else { STATEMENT* }
EXP         ::= VAR | FUNCTION | EXP BINOP EXP | BLOCK
FUNCTION    ::= self. NAME ( EXP* ) | NAME ( EXP* ) | NAME . NAME ( EXP* )
MODIFIER    ::= abstract | behavior | startup | command
VAR         ::= NAME
VALUE       ::= NUMBER | PREDEFINED
DEPLOYMENT  ::= deployment { NAME* }
```

Figure 4: Proposed BNF for RDCD. Note that for simplicity, commas between function arguments are omitted in the BNF.

a module plays a given role or any of the super-roles. A method is simply a sequence of statements that either are function invocations or conditionals. For simplicity methods currently always take zero arguments, but we expect this limitation to change in the future. Function invocations are either local commands, functions, or global commands. Local commands access the physical state of the module (sensors, actuators, external code) and are prefixed with the term "`self.`" to indicate that it is a local operation. Functions are basically used to represent stateless operations such as computing the size of (i.e., number of bits in) a bit set.[1] Global commands are of the form "`Role.command`" and causes the command to be asynchronously invoked on all modules currently playing that role or any of its sub-roles. Arguments to functions are expressions, either constants, compound expressions, or code blocks; a code block allows code to be stored for later use (e.g., an event handler) or to be executed in a special context (see example below). Note that since the code is stateless no closure representation is required. The function invocation syntax for primitive functionality from the role `Module` (such as turning the main actuator) is the same as that of user-defined functions.

A method declaration can be prefixed by a modifier, as follows. The method modifier "`abstract`" works in the usual way (forces the enclosing role to be declared abstract). The method modifier "`behavior`" causes the method to execute repeatedly so long as the role is active, whereas the method modifier "`startup`" causes the method to execute once when the role is activated. Last, the method modifier "`command`" causes the method to become exported for invocation as a global command.

To supplement the basic "extends" approach to creating a hierarchy of roles, a role can also "modify" another role meaning that it is a *mixin role* that can be applied to the designated role or any of its sub-roles. This approach allows smaller units of behavior to be encapsulated into well-structured roles that can be activated throughout specific parts of the role hierarchy. Mixin roles currently cannot be activated automatically using invariants but must be explicitly selected using a special self function `assumeRole` that takes a mixin role and a code block as arguments and causes the module to temporarily change to the given role while the code block is executed.

[1]Bit sets are used in the DCD-VM to represent sets of connectors, which conveniently can be done using a single byte since there are only 8 connectors.

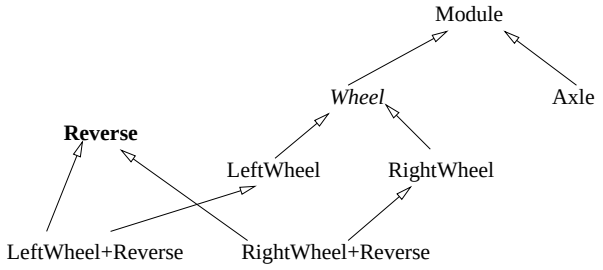


Figure 6: Hierarchy of sub-role relations for the RDCD program of Figure 5. Arrows represent the sub-role relationship, roles in italics are abstract, roles in bold are mixins.

To facilitate the implementation we currently require the programmer to explicitly specify in what order roles discovery is performed in the structure. For example, in a car an axle is a module that is attached to wheels, so wheels must be identified before axles. Such dependencies can be detected automatically by an analysis on the invariants or even made redundant by having role discovery run for a while until it stabilizes; such extensions are however considered future work.

3.3 Example resolved

The complete RDCD program for implementing obstacle avoidance in an arbitrary car-like structure of ATRON modules is shown in Figure 5. The role structure is illustrated in Figure 6: a generic wheel role is used as a basis for defining concrete roles for left and right wheels. The difference between a left wheel and a right wheel is what direction to turn the main actuator to advance and on what connector to monitor for obstacles. Moreover, a mixin role is used to indicate a reversing wheel since its behavior is different which should be observable to the rest of the structure.

In more detail, the abstract role `Wheel` abstracts over constants defining on what side the wheel should be connected, what event handler vector should be monitored for proximity

```
abstract role Wheel {

  abstract constant connected_direction, event_handler, turn_direction;

  self.center_position == EAST_WEST;
  sizeof(self.total_connected()) == 1;
  sizeof(self.connected(UP)) == 1;
  sizeof(self.connected(connected_direction)) == 1;

  startup move() {
    if(self.y()>0) self.handleEvent(event_handler, { self.disableEvent(event_handler); Wheel.stop(); });
    self.turnContinuously(turn_direction);
  }

  command stop() {
    self.assumeRole(Reverse,{
      self.turnContinuously(-turn_direction);
      self.sleepWhileTurning(3);
      self.turnContinuously(turn_direction);
      self.enableEvent(event_handler);
    });
  }
}

role RightWheel extends Wheel {
  constant connected_direction := EAST;
  constant turn_direction := 1;
  constant event_handler := EVENT_PROXIMITY_5;
}

role LeftWheel extends Wheel {
  constant connected_direction := WEST;
  constant turn_direction := -1;
  constant event_handler := EVENT_PROXIMITY_1;
}

role Reverse modifies Wheel { }

role Axle {
  sizeof(connected_role(DOWN,Wheel)) > 0;
  behavior steer() {
    if(connected_role(DOWN,Reverse) > 0) {
      if(self.y>0)
        self.turnTowards(30);
      else
        self.turnTowards(-30);
      }
    }
    else
      self.turnTowards(0);
  }
}

deployment { RightWheel, LeftWheel, Axle }
```

Figure 5: RDCD program implementing obstacle avoidance (manually pretty-printed to improve readability)

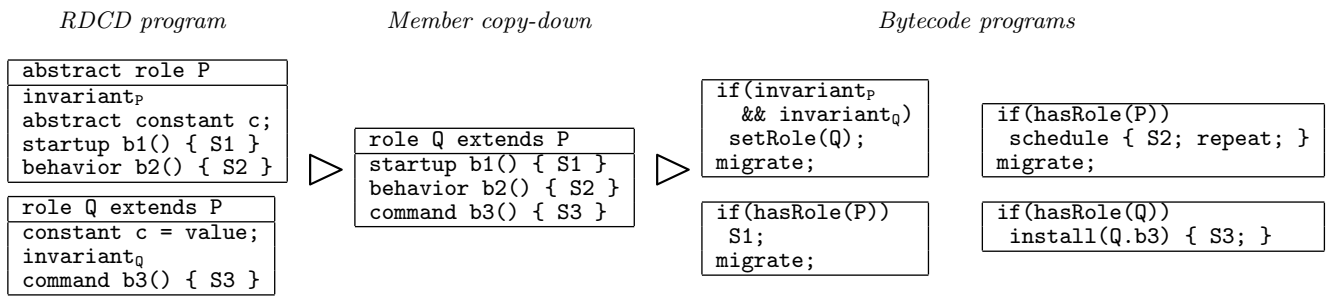


Figure 7: Basic RDCD compilation process from roles to mobile bytecode programs

detection, and in what direction the main actuator should turn. Then follows a number of invariants for defining a wheel: rotational axis perpendicular to the direction the vehicle should be moving, only connected to a single module etc. The initial behavior of a wheel is to install an event handler if the y coordinate is positive[2] and then to start turning continuously to make the car move forward. The event handler starts by disabling itself (to avoid triggering multiple events) and then invokes the method `stop` on all wheels. The stop command temporarily assumes the mixin role `Reverse`, reverses, and then restores the wheel to its previous state. The left and right wheels simply concretize the abstract wheel class by defining the abstract constants.

The mixin role `Reverse` is empty, it is in fact used as a "marker role": the role `Axle` reacts to its adjacent wheel modules assuming the reverse role, and will in this case turn the axle as appropriate. Note that turning the axle depends on the global y coordinate which for example causes the front and back wheel on the 6-wheeled car to turn in different directions. This steering behavior in the axle is represented using a behavior method that continuously monitors the role of the connected module. (The DCD-VM maintains a locally stored awareness of the roles of the adjacent modules, meaning that distributed communication only is used when the wheel module changes roles, not every time the steering behavior is run.)

3.4 Compiling RDCD to the DCD-VM

We now describe the compilation of RDCD into stateless, mobile programs for the DCD-VM. A critical constraint is the size of the compiled programs, since the DCD-VM currently transmits mobile programs using the standard ATRON communication primitives which can become unstable when the buffer size exceeds 50 bytes. For this reason, we prefer multiple smaller mobile programs that move concurrently throughout the structure as opposed to a single, larger program that is harder to transmit correctly on the physical hardware. Apart from a few peephole optimizations the compiler does not do any analysis and optimization, but there are numerous opportunities for optimizations, as will be discussed later. Unless otherwise noted, all mobile programs generated use migration instructions to disperse throughout the module structure.

The compilation process form roles to mobile bytecode programs is illustrated in Figure 7. The first step of the compilation process is to copy down members from super-roles to sub-roles; as will be explained later, mixin roles can currently be ignored at this point. For each role a mobile program is then generated that checks the associated invariants and sets the role accordingly if all the invariants are satisfied. Next, for each startup method a mobile program is generated that first checks the role and evaluates the method body if the role matches. Similarly for behavior methods, except that compiled behaviors use a special "repeat" instruction that causes the method to be rescheduled for later execution from the start. Last, commands are installed on all modules that implement the appropriate roles.

Since mixin roles currently only can be activated explicitly using the function `assumeRole`, they can simply be represented by generating code sequences for changing to a different role and back again. (Method overriding is non-trivial to update when the role changes, but this can be done since mobile programs essentially can modify the "remote invocation vtable" of each module.)

The copy-down approach allows role-specific constants to be inlined into every program, and moreover simplifies the implementation of features such as startup methods since they simply can check for an exact role match instead of having to take method overriding in sub-roles into account. The downside is that deep role hierarchies will generate numerous mobile programs, many of which may be redundant. We believe a useful optimization would be to reduce the number of mobile programs by combining them without exceeding the optimal message size for the physical modules.

3.5 Experiments

The RDCD compiler has been implemented in roughly 2000 lines of Java code, but currently does not include a parser. Nevertheless, our running example is the program of Figure 5 which has been fed to the compiler by manually building the AST.[3] The output of the compiler is a C program that initializes a collection of arrays with DCD-VM bytecodes. This approach is currently required to run programs on the DCD-VM since it does not support downloading code from a non-module source. In effect, all the code is loaded in a single module (the "connector" module in the car) and diffused throughout the module structure from this module.[4] The generated code is equivalent in func-

[2]The DCD-VM maintains compass directions and a 3D coordinate system of the entire structure relative to the module where the program was injected into the structure. This module thus determines the directionality for the entire structure.

[3]We are currently investigating different options for the concrete syntax based on feedback from researchers in the robotics community, and plan on implementing a complete parser when this study has been completed.

[4]This approach corresponds to reprogramming a single mod-

tionality to the hand-written bytecode initially used for the obstacle avoidance, which can perform the obstacle avoidance described at the beginning of this section in simulation (the DCD-VM is not currently working on the physical hardware due to various low-level implementation issues). The generated code is however less efficient: generated code fragments are typically 50% larger and twice the number of code fragments are generated by the compiler than was present in the manually written code. As mentioned earlier, we believe that improved peephole optimizations and sharing of code between related classes will close the gap between the automatically generated code and the manually written code.

3.6 Assessment

The use of a high-level language to program the ATRON modules using the DCD-VM provides a significantly higher level of abstraction to the programmer which we expect will result in a massive increase in productivity. A larger set of experiments are however required to determine if this is the case. Moreover, we are also interested by how useful the individual features of RDCD are when writing programs. The required experiments are however out of the scope of this paper due to the preliminary state of the compiler. Nevertheless, we can conclude that the use of inheritance between roles combined with abstract constants allows the compiler to generate mobile code fragments that are small and have minimal resource requirements, which is a perfect fit for the DCD-VM. Moreover, the use of explicitly activated mixin roles provides language support for behaviors to temporarily modify the role that a module is playing without requiring numerous redundant declarations at the source level.

The RDCD compiler currently does not implement a type checking, but the language is by design statically typed in the sense that it is possible to check statically that local invocations of behaviors always succeed. (The lack of threading on a single module combined with the simple lexical nesting of the argument to `self.assumeRole` facilitates type checking.) Due to the distributed nature of the ATRONs, remote invocation of behaviors cannot be guaranteed to succeed. For example, a module may change role just after a remote command has been delivered to the module, but before it has been scheduled for execution (such a command is ignored in the current implementation). In general, we believe that a statically typed approach is likely to be too brittle for a dynamically evolving distributed system, but large-scale experiments are needed to determine what is the most useful approach.

The RDCD language currently does not support state, which significantly simplifies the role change mechanism. Programmers thus have to resort to defining their own operations implemented in C code for manipulating state, which is obviously not a satisfactory solution. Moreover, there is currently no support for migrating state with mobile programs, which complicates e.g. writing a mobile program that finds those potential wheel modules that are at the bottom of the structure. Resolving these issues is considered future work, but we envision allowing the programmer to declare state both globally (persistent across role changes) and locally to roles (transient across role changes), since a preliminary study of existing programs for the ATRON seems to indicate the need for both kinds of state.

ule which in practice is much easier than reprogramming all modules in a robot; memory is not a practical issue since the constant arrays holding the generated C code are stored in the 128K flash program memory which is much larger than the 4K of RAM.

4. RELATED WORK

As an alternative to the DCD-VM, we have developed the RAPL system that statically compiles role declarations written in a simple XML-based language to conventional C programs [7, 8]. Each role declaration is explicitly tied to the physical structure of the robot, making it easy to deploy and experiment with in practice, but less flexible in terms of what robot structures a given program can support. The commands declared for each role can simply be called remotely by the neighboring module. We see this system as a simple precursor to RDCD, since it only supports a small subset of its features, namely the basic concept of roles. Nevertheless, this system is more complete in the sense that it from an XML specification generates code that works on the physical modules.

Autonomous robots are often programmed using behavior-based control [3]; behaviors are typically sensor-driven, reactive, and goal-oriented controllers. Certain behaviors may inhibit other behaviors, allowing the set of active behaviors to vary. Modular robots often use the concept of a role albeit in an ad-hoc fashion: complex overall behaviors can be derived from a robot where different modules react differently to the same stimuli, in effect allowing each module to play a different role (e.g., [1, 5, 19]). Recently, Støy et al have explicitly used the concept of a role to obtain a very robust and composable behavior [20, 22]. Compared to RDCD, the implementation of roles is ad-hoc and the only control examples investigated are cyclic, signal-driven behaviors for locomotion.

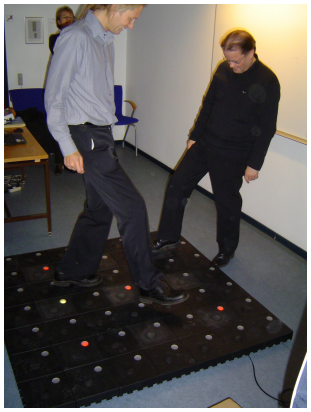
Apart from RDCD and RAPL, the only high-level programming language for modular robots that the authors are aware of is the Phase Automata Robot Scripting Language (PARSL) [10, 25]. Here, XML-based declarations are used to describe the behavior of each module in the PolyBot self-reconfigurable robot [24]. Compared to RDCD and RAPL, the tool support is much more complete and the language has many advanced features for controlling locomotion using behavior-based control. Nevertheless, PARSL completely lacks the concept of a role for structuring the code: each behavior is assigned to a specific module as an atomic unit. Moreover, PARSL has no support for dynamically distributing code in the robot.

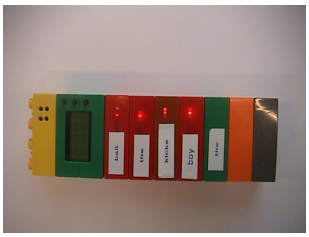
Outside the field of robotics, roles and mixins have been investigated in numerous cases, which forms the basis of our language design. Regarding static typing, our role-change mechanism resembles that of Fickle [6], but since RDCD roles have no state and role change is always local to a single behavior, our approach is much more restricted but also easier to both implement and type check statically (although the latter property has not been investigated in practice). Mixin roles are a particularly simple use of the more general concept of a mixin [2] that we expect to explore more generally in future work.

5. CONCLUSION AND FUTURE WORK

In the paper we have presented the design of the RDCD language for programming ATRON modules using role-based programming coupled with distributed control diffusion. The


Playware: initial prototype


Playware: new prototype


iBlocks: initial prototype


iBlocks: new prototype

Figure 8: Examples of physically interlocked systems. The Playware modules are interactive playgrounds with pressure sensors and color LEDs, adjacent modules communicate using a wired connection that is established when the modules are combined. The iBlocks are physical artifacts that allow children to interact with computing devices, they are equipped with connectors with infrared communication (the newest version uses magnetic connectors), tilt sensors and LEDs. In both cases, when users physically reconfigure the system the behavior of the system as a whole should evolve accordingly.

design is supported by a preliminary implementation of a compiler that can generate code for the non-trivial obstacle avoidance scenario. Ongoing improvements to the compiler includes completing the front-end parser and improving the back-end optimizations.

In terms of future work, there are numerous improvements that could be made to RDCD and the DCD-VM. In the shorter term, a major issue is enabling the programmer to express more precisely the relations and collaborations between modules, as opposed to describing the individual behaviors that give rise to the collaboration. For example, the obstacle avoidance program in Figure 5 is not obviously an obstacle avoidance algorithm; we believe that a programming language that has a greater focus on the collaborations would facilitate expressing such an algorithm clearly and succinctly. As a long-term perspective, we are however interested in generalizing the application domain, not only to other types of modular self-reconfigurable systems, but also to a more general class of embedded devices that could be referred to as *physically interlocked systems:* networked embedded systems with physical connectors, where the way the systems are connected affects their behavior. Modular robots are an example of such a system, but the authors are currently investigating other systems that share the same characteristics, such as the Playware Tiles and the iBlocks, both shown in Figure 8 [12, 15]. We believe parts of the DCD-VM and the RDCD language also would be applicable to such systems, which can lead to the development of a family of language platforms for physically interlocked systems.

6. REFERENCES

[1] H. Bojinov, A. Casal, and T. Hogg. Multiagent control of self-reconfigurable robots. In *Proceedings of Fourth International Conference on MultiAgent Systems*, pages 143–150, 2000.

[2] G. Bracha and W Cook. Mixin-based inheritance. In N. Meyrowitz, editor, *OOPSLA/ECOOP '90 Proceedings*, pages 303–311. ACM SIGPLAN, 1990.

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

[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] Sophia Drossopoulou, Ferruccio Damiani, Mariangiola Dezani-Ciancaglini, and Paola Giannini. More dynamic object reclassification: Fickle∥. *ACM TOPLAS*, 24(2):153–191, 2002.

[7] Nicolai Dvinge. A programming language for ATRON modules. Master's thesis, University of Southern Denmark, 2007.

[8] Nicolai Dvinge, Ulrik P. Schultz, and David Christensen. Roles and self-reconfigurable robots. In *Proceedings of the ECOOP'07 Workshop Roles'07 — Roles and Relationships in OO Programming, Multiagent systems and Ontologies*, 2007. To appear.

[9] S.C. Goldstein and T. Mowry. Claytronics: A scalable basis for future robots. *Robosphere*, November 2004.

[10] Alex Golovinsky, Mark Yim, Ying Zhang, Craig Eldershaw, and Dave Duff. Polybot and PolyKinetic system: A modular robotic platform for education. In *IEEE International Conference on Robots and Automation (ICRA)*, 2004.

[11] 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.

[12] H. H. Lund, T. Klitbo, and C. Jessen. Playware

technology for physically activating play. *Artificial Life and Robotics Journal*, 9:165–174, 2005.

[13] 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.

[14] 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.

[15] J. Nielsen and H. H. Lund. Modular robotics as a tool for education and entertainment. In *Proceedings of IADIS International Conference on Cognition and Exploratory Learning in Digital Age (CELDA 2005)*, 2005.

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

[17] Ulrik P. Schultz. Distributed control diffusion: Towards a flexible programming paradigm for modular robots. Submitted for publication, preliminary version available at `http://www.mmmi.sdu.dk/~ups/apges07/dcd.pdf`.

[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. A reconfigurable modular robot with many modes of locomotion. In *Proceedings of the JSME international conference on advanced mechatronics*, pages 283–288, Tokyo, Japan, 1993.

[24] 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.

[25] Ying Zhang, Alex Golovinsky, Mark Yim, and Craig Eldershaw. An XML-based scripting language for chain-type modular robotic systems. In *Proceedings of the 8th Conference on Intelligent Autonomous Systems (IAS)*, 2004.