Roles and Self-Reconfigurable Robots

Nicolai Dvinge, Ulrik P. Schultz, and David Christensen

Maersk Institute
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 associate roles and behavior to structural elements in a modular robot. Based on the role declarations, a dedicated middleware for high-level distributed communication is generated, significantly simplifying the task of programming self-reconfigurable robots. Our language fully supports programming the ATRON self-reconfigurable robot, and has been used to implement several controllers running both on the physical modules and in simulation.

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 [2, 6, 7, 9–11, 16, 15]. 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. [3, 13]), 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.

In this paper, we present a role-based approach to programming a controller for a distributed robot system. We have implemented a prototype role-based programming language, named “RAPL”, for the ATRON modular, self-reconfigurable robot [7, 8]. Our implementation is based on roles as the main

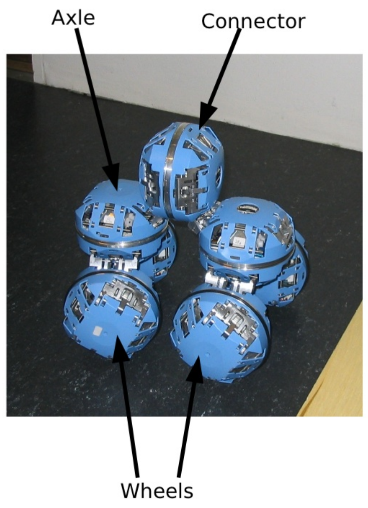

Fig. 1. The ATRON self-reconfigurable robot. Seven modules are connected in a car-like structure.

abstraction of behavior and implements a remote method invocation framework based on the roles and their structural interconnections. RAPL allows the programmer to construct a controller for a structure of ATRON modules in less time and with less knowledge of hardware than was the case before. Moreover, we argue that the controller constructed is less error-prone and is more intuitive to comprehend.

The contributions of our work are as follows: We use the concept of role-based programming to identify the central entities in a distributed robot controller. We increase the level of abstraction in the process of programming robot controllers by means of a domain specific language (DSL), which is built upon these concepts. The structural information from the DSL is used to provide a lightweight remote method invocation framework appropriate for the physical constraints of the ATRON modules. Moreover, our language allows the programmer to specify the behavior of the robot as a whole from the constituent parts. Our compiler generates an application framework either in embedded C code for execution on the physical modules or in well-structured Java code for execution in a virtual simulation environment.

2 The ATRON Self-Reconfigurable Robot

The ATRON self-reconfigurable robot is a 3D lattice-type robot [7, 8]. Figure 1 shows an example ATRON car robot built from 7 modules. Two sets of wheels (ATRON modules 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. 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.

Other examples of self-reconfigurable robots include the M-TRAN and the SuperBot self-reconfigurable robots [9, 11]. 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.

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 metamodules [3], 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 [14]. 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.

3 A Role-based Conceptual Model

The level of abstraction offered by focusing on the behavior of a specific module in a given context is somewhat similar to that of role-based programming [12]. We use this approach as a basis for our language by making roles the fundamental concept for expressing the desired behavior. A fundamental difference between previous work and our approach is however that previous role-based experiments have focused on performing cyclic behavior, e.g., locomotion, and not event routing and reactive behavior. Moreover, all implementations presented in earlier work have been constructed in an ad-hoc manner with little or no language support.

Our conceptual view of a role is that it defines the module structure and the active and reactive behavior of each module in a robot. In other words, it defines what the module *can* do and what it *will* do. There is a one-to-one mapping between a role and a module, but modules can change their roles (and thus their behavior) as a reaction to messages from other modules or internal events. The behavior of a role is thus determined by the physical state of the module (as reported by sensors), program state stored in the memory of the module, and messages received from other modules. The behavior is encapsulated in methods that are activated either through external messages or internal events.

An ATRON robot as a whole is implicitly assigned a role using the object-oriented concept of a whole-part structure (as known from the whole-part design pattern [1]). Behavior for the robot is declared for each individual role. For example, all modules in a car may be able to play the role of a "car" by receiving messages for the "car" role. A module may either process such a message or forward it to another module, as designated by the programmer. The functionality of the whole and the role that it can play is thus created in coordination between the individual modules, corresponding to how the control of a modular robot necessarily must be implemented in practice.

4 The RAPL Compiler

We have implemented a role specification language for the ATRON modules, named RAPL (Role-based ATRON Programming Language). RAPL can be compiled either to Java, for use with the ATRON simulator, or to C, for execution directly on the modules. The Java backend simply generates an implementation of the proxy and state design patterns [5]. The C backend compiles a role to a skeleton that invokes C functions written by the programmer and to a proxy represented as a collection of C functions that can be used to send messages to other modules implementing this role.

4.1 RMI based communication

RAPL implements a remote procedure calling functionality to facilitate distributed communication. A RAPL program declares a list of functions each belonging to a specific role. Functions can be tied to an event or provide a default behavior for a role; an event can be a message from a neighbor-module or an internal event signal (e.g. timer, tilt sensor, etc.). Having a list of functions for each role is sufficient to generate a stub/skeleton proxy framework that provides abstraction over communication which is critical to our approach. The issue of addressing the correct modules is resolved by exploiting the structural information from the roles.

4.2 Whole-part design architecture

The whole-part design pattern is integrated in the implementation of the RMI framework, providing a whole-part functionality for the entire structure of ATRON modules. Functions declared on a controller level expose the controllers main functionality to other controllers or external clients, which would not normally know, e.g., how to make the car drive or turn.

4.3 A domain specific language with simple logic

RAPL is a domain-specific language used to express roles and functions; allowing control structures and other imperative language constructs would hamper the intention of defining behavior at a higher abstraction level. Moreover, we believe that general-purpose languages such as Java and C are better suited for specifying more complex, internal behavior in controllers. To facilitate practical experiments, we do however enable RAPL to directly express primitive actuation operations and message forwarding with simple arithmetic processing of arguments. More advanced functionality will have to be included from externally linked code supplied directly by the programmer. To this end, we have defined a simple programmatic interface between RAPL and each of the platforms that it supports: RAPL methods can be implemented in the target language and code written in the target language can call RAPL methods. We currently use XML as the concrete syntax for writing RAPL declarations; in the future we envision providing one or more high-level syntaxes, as exemplified in Figure 2.

5 Example

We now outline a few simple examples of using RAPL to program an ATRON car. We refer to the first author's MS for more detailed examples [4].

5.1 A simple car program

As a concrete example, consider the ATRON car shown earlier in Figure 1. Simple reactive control of this robot can be implemented using the role declarations shown in Figure 2, left. A role has a name and declares its structural dependencies on other roles, and can moreover extend another role creating a hierarchy of roles (not shown). Structure is specified in terms of the roles of the neighboring modules and the physical communication port used to contact the module (auto-detection of neighboring modules, although possible, is currently slightly problematic on the physical modules due to hardware difficulties). Behaviors are simply declared as functions that are attached to roles.

In the program of Figure 2, when a `move` event is delivered to the "connector" it forwards it to the two axles, which again forwards it to the wheels. The action performed by the wheel is a primitive actuation of the main joint, implemented by all modules (similarly to the methods provided by `Object` in Java). Similarly for the `turn` event. To increase readability, we are currently investigating a more human-friendly syntax resembling Java declarations, as shown in Figure 2, right.

The roles thus provide a means of denoting the behavior of each module as it is used for a specific purpose in the robot. Moreover, the roles provide a simple and very light-weight way of (albeit manually) routing events through the topology of the robot. This is particularly interesting given the resource constraints of the ATRON module, which only has 4K of RAM available for communication buffers, operating system functionality, and program state. (Program size on the other hand is not so much of a problem since there is 128K of flash memory available for storing programs.)

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE Controller>
<Controller Name="Car" xsi="Xatron.xsd">
 <Role RoleName="Connector" ...>
  <Structure RoleToUse="Axle"
         StructureName="axleFront"
               Channel="2"/>
  <Structure RoleToUse="Axle"
         StructureName="axleRear"
               Channel="6"/>
 </Role>
 <Role RoleName="Axle" ...> ...</Role>
 <Role RoleName="Wheel" ...> ...</Role>
 <Function FunctionName="move" Target="Connector">
  <Action ActionName="move"
              Target="axleFront"
               Value="value"/>
  ...
 </Function>
 <Function FunctionName="turn" Target="Connector">
  <Action ActionName="rotate"
              Target="axleFront"
               Value="value/2"/>
  <Action ActionName="rotate"
              Target="axleRear"
               Value="value/2"/>
 </Function>
 ...
</Controller>
```

```
role Connector implements Car {
  Axle frontAxle = Axle(channel#2);
  Axle rearAxle = Axle(channel#6);

  move(int value) {
        frontAxle.move(value);
        rearAxle.move(value);
  }
  turn(int value) {
        frontAxle.rotate(value/2);
        rearAxle.rotate(-value/2);
  }
}

role Axle implements Car {
  Wheel leftWheel = Wheel(channel#0);
  Wheel rightWheel = Wheel(channel#2);

  move(int value) {
        leftWheel.move(value);
        rightWheel.move(-1*value);
  }
}

role Wheel implements Car {
  Axle axle = Axle(channel#5);
}
```

Fig. 2. A simple car controller implemented in RAPL (left) and in our proposed Java-like syntax (right)

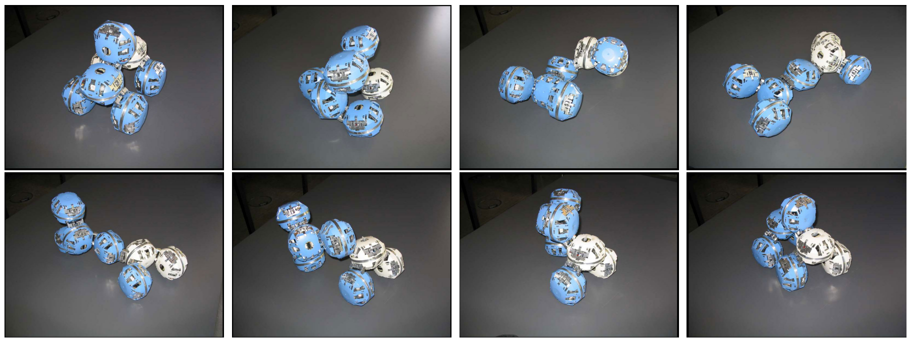

Fig. 3. The ATRON car rebuilding itself after a tilt

```
<Function FunctionName="Run" Target="Connector">
      <Action ActionName="extTiltTurnLogic" Target="EXT" Value="0"/>
      <Action ActionName="extTiltGetupLogic" Target="EXT" Value="0"/>
</Function>
<Function FunctionName="drive" Target="Car">
  <Action ActionName="move" Target="connector" Value="value"/>
</Function>
<Function FunctionName="drive" Target="Car">
  <Action ActionName="move" Target="connector" Value="value"/>
</Function>
<Function FunctionName="getup" Target="Car">
  <Action ActionName="getupimpl" Target="connector" Value="value"/>
</Function>

<Function FunctionName="getupimpl" Target="Connector">
  <Action ActionName="rotate" Target="Connector" Value="1"/>
  <Action ActionName="pause" Target="Connector" Value="2000"/>
  <Action ActionName="rotate" Target="Connector" Value="1"/>
  ...
  <Action ActionName="pause" Target="Connector" Value="2000"/>
  <Action ActionName="rotateDegrees" Target="axleFront" Value="-25"/>
</Function>
```

Fig. 4. A tilt-aware Car controller in RAPL

5.2 Adding proactive behavior

Our working example of a car is shown in action in Figure 3. Here, we have added custom functionality which fires events based on the internal tilt-sensor. Tilting makes the car turn towards higher grounds (the car will move towards the top of a hill) whereas an extreme tilt-level (indicating that the car has fallen over) triggers a series of reconfigurations which rises the car. The RAPL declarations for this more advanced controller extend those of Figure 2 and are shown in Figure 4. In this example we specify several functions and some custom code functionality to exemplify the diversity of the RAPL language.

The behavior of the controller for the tilting car is defined in the `Run` method on the connector. This tells the connector module to call the two external tilt-functions, which are shown in Figure 5. These two functions monitor the y-axis tilt sensor for a minor or severe change in tilt level. The first method `extTiltLogic` reacts upon tilt changes in steps of 10, and for each step it calls the `Connector` method `turn`, thus turning the axles. Should the car tilt over completely, we would normally be stuck with a car that cannot move, since its wheels are not touching the ground anymore. This is resolved in the controller by the `extTiltToGetup` method which senses an extreme tilt level (currently set to 70 degrees) and triggers a self-reconfiguration sequence of several rotations. The sequence of rotations can be seen in Figure 4 in the function `getupimpl`. The two functions in the custom code of Figure 5 display an obvious example

```
#include "rapl.h" /* header file generated by RAPL for the car */

char isGettingup = 0;
                                     void extTiltToGetup() {
void extTiltLogic() {                 signed char x = getTiltY();
 signed char x = getTiltY();          setNorthIOPort(abs(x));
 if(abs(x-last)>10) {
  if ((x - last) < 0)                 if(abs(x)>70) {
   Connector_turn_impl(-20);           if (isGettingup == 0) {
  else                                  isGettingup = 1;
   Connector_turn_impl(20);             Connector_getup_impl();
  last = x;                            }
 }                                    }
}                                    }
```

Fig. 5. The custom code for the tilt-logic

of functionality that would be laborious to provide in our RAPL compiler and thus should be supplied in native code.

6 Conclusion

In this paper, we have presented the RAPL system, a role-based approach to abstraction of hardware and communication in the ATRON system. Based on our experiments, we conclude that supporting role-based programming at the language level makes programming distributed robots less tedious and thereby more accessible and less error prone. We believe that providing a high-level programming interface is critical for improving the maturity of the ATRON robotic system and is an important first step in moving towards concrete applications of self-reconfigurable robots. In terms of language design, we are interested by the interplay between role-based programming and object-oriented design principles such as whole-part. Furthermore, we believe that the use of roles and object oriented principles like whole-part will appeal to people less minded for hardware. Or to put it to the point: never before has a constructor made as much sense as seeing a car being built! Reusability could be much higher in the future when programmers can concentrate on *what* is to be done instead of *how* to provide the infrastructure that supports it. Architecture-oriented programmers can leave the basic behavioral programming to people more focused on the modules and "orchestrate" a structure of ATRON modules by choosing a combination of control strategies.

References

1. F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal. *Pattern-Oriented Software Architecture: A System of Patterns.* Wiley, 1996.

2. 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.
3. 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.
4. Nicolai Dvinge. A programming language for ATRON modules. Master's thesis, University of Southern Denmark, 2007.
5. E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1994.
6. S.C. Goldstein and T. Mowry. Claytronics: A scalable basis for future robots. *Robosphere*, November 2004.
7. 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.
8. 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.
9. 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.
10. D. Rus and M. Vona. Crystalline robots: Self-reconfiguration with compressible unit modules. *Journal of Autonomous Robots*, 10(1):107–124, 2001.
11. 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.
12. 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.
13. 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.
14. 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.
15. 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.
16. 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.