# Spatial Computing with Labels

U.P. Schultz, M. Bordignon, D. Christensen, K. Stoy

***Abstract*— A reconfigurable robot is a robot that can change shape. Programming reconfigurable robots is complicated by the need to adapt the behavior of each of the individual module to the overall physical shape of the robot. In this position paper, we investigate a simple approach to allow the programmer to abstract over the concrete shape of a robot using the notion of a *label* as a simple means of addressing various parts of the structure of a robot. Labels provide the programming language designer with a means of stratifying two main components of a spatial programming language for modular robots, namely specifying the physical structure of a robot and specifying its behavior. Based on previous experience with the ATRON robot, we find that labels are a useful concept for programming modular robots.**

## I. Introduction

A reconfigurable robot is a robot that can change shape. Reconfigurable robots are typically built from multiple modules that, in the case of self-reconfiguration, can manipulate each other to change the shape of the robot [1], [2], [3], [4], [5], [6], [7], [8], [9]. Changing the physical shape of a robot allows it to be adapted to various tasks, for example by changing from a car configuration (best suited for flat terrain) to a snake configuration suitable for other kinds of terrain. Programming reconfigurable robots is however complicated by the need to adapt the behavior of each of the individual modules to the overall physical shape of the robot. Concretely, we would like to enable changing the spatial relations between the modules to change the behavior of the robot without needing to rewrite our controller programs to take irrelevant details of the current spatial layout into account.

In this position paper, we investigate a simple approach to allowing the programmer to abstract over the concrete shape of a robot using the notion of a *label* as a basic means of addressing various parts of the structure of a robot. Informally, labels are to modular robots what variable names are to structured programming. Labels provide the programming language designer with a means of stratifying two main components of a spatial programming language for modular robots, namely specifying the physical structure of a robot and specifying its behavior. We have in earlier work implicitly used the notion of labels in two languages for role-oriented programming [10], [11], [12], but using labels allows us to more clearly describe the features of these languages. As such, we find that labels are a useful concept for describing recurring issues in programming modular, reconfigurable robots. For this reason, we believe that programming languages designed with modular robots in mind need to incorporate the notion of a label.

The authors are with The Modular Robotics Lab, Maersk Mc-Kinney Moller Institute, Faculty of Engineering, University of Southern Denmark, Denmark.{ups,mirko,david,kaspers}@mmmi.sdu.dk

This work was supported by The Danish Council for Technology and Production.

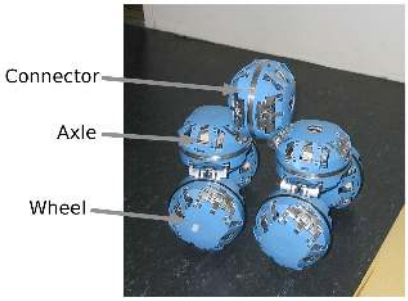



Fig. 1. The ATRON self-reconfigurable robot. Seven modules are connected in a car-like structure.

The rest of this paper is organized as follows. First, Section II presents our experimental platform, the ATRON and Odin robots, as well as general approaches to programming these robots and a virtual machine for the ATRON designed to facilitate structural abstraction. Then, Section III defines the abstract notion of a label as a programming language concept, after which Section IV describes a number of concrete approaches to using labels in programming languages, based on experiments we have performed using the ATRON robot. Last, Section V discusses our ideas and how they can be applied to other robots, including the Odin robot.

## II. ATRON and Odin: Hardware and Software

### A. Hardware

The ATRON self-reconfigurable modular robot is a 3D lattice-type system [3]. Figure 1 shows an example ATRON car robot built from 7 modules. An ATRON module has one degree of freedom, is spherical and is composed of two hemispheres which can be rotated relatively to each other. A module may connect to neighboring 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 objects. A module weighs 0.850kg and has a diameter of 110mm. Currently 100 hardware prototypes of the ATRON modules exist. The hardware is controlled by two Atmel ATMega128 microcontrollers, one on each hemisphere, communicating between them through an RS-485 serial link. Each CPU

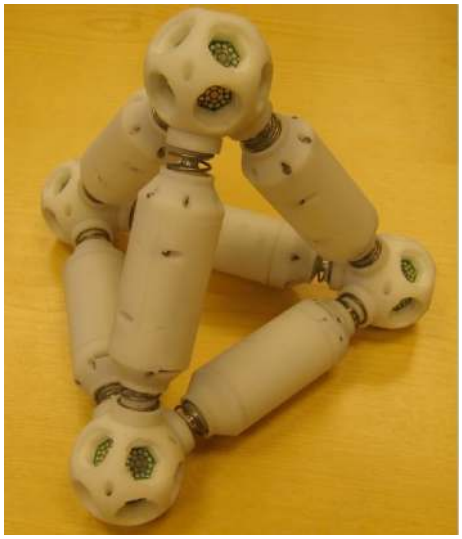


Fig. 2. The Odin reconfigurable robot in a pyramid walker configuration.

has 128 Kbytes of flash memory for storing programs and 4 Kbytes of RAM for use during program execution.

The Odin modular robot is a heterogeneous system composed of link and joint modules [9]. A link connects to a joint at each end of its cylindrical body, and the arrangement of the connections on the joints determines the lattice structure of the robot. Joints are passive elements of the mechanical structure, allowing also power sharing and communication over RS-485 among different link modules. Different functionalities are implemented by different kinds of link modules [13], like passive structural support, sensing, actuation and power storage. The electronics of the link modules consists of a general board common to all the different links and a specific one related to the different incorporated functionalities. The control software runs on the CPU of the general board, an Atmel AT91SAM7S microcontroller with 256Kbytes of program memory and 16 Kbytes of RAM.

### B. Software

Programming the ATRON and Odin robots is complicated by the distributed, real-time nature of the system coupled with limited computational resources (although more so for the ATRON) and the difficulty of abstracting over the concrete physical configuration when writing controller programs. General approaches to programming the ATRON robot include metamodules [14], motion planning, and rule-based programming [15]. Our initial, general approaches to programming the Odin robot include simulated experiments with various learning strategies. Moreover, as will be described later, we are currently experimenting with an interactive approach to programming both robot systems based on deriving an autonomous controller from a set of gait tables [16].

To facilitate dynamically updating the behavior of 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 the those modules where a specific behavior is needed. The virtual machine, named DCD-VM, runs on each module, has an instruction set that is dedicated to the ATRON hardware, and includes operations that are typically required in ATRON controllers. For example, on each module the virtual machine maintains an awareness of the compass direction of each module and the roles of its neighbors, and specific instructions allow this local context information to be queried. The DCD-VM supports a basic notion of roles to indicate the state of a module to its neighbors 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 implemented by the programmer or provided by a higher-level programming language.

In a first attempt to provide a high-level language for programming the ATRON robot, we have developed a number of domain-specific languages for the DCD-VM based on the idea of 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 [10]. 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. Our role-based languages allow the programmer to declaratively specify a hierarchy of roles and associated behaviors, using either a role-centric [11] approach inspired by object-oriented programming or a behavior-centric [12] approach inspired by functional programming. In the former approach behaviors are specified per-module using roles whereas in the latter approach behaviors are specified across multiple modules and are differentiated using roles. Nevertheless, in both cases the conditions under which a role is active are specified using logical predicates on the context of a module. Moreover, in both cases the link between role conditions and behaviors is made using role names, making the role condition declarations largely independent of the behavior specifications, and vice versa. To simplify the language design and enable experimenting with different approaches to specifying role conditions and behaviors, we propose to separate the two using the concept of a *label*, as will be explained in the next section.

## III. The Role of Labels

We now describe the concept of a label and how it can be used as a fundamental concept in the design of programming languages for modular robots. Concrete examples of how labels can be used as a programming language concept are provided in the next section.

### A. Labels

We propose the notion of *labels* as a means of separating knowledge of the spatial structure of a robot from the behaviors of the individual parts of the robot. The word "label" is used because we conceptually label various physical parts of the robot. A label provides a symbolic means of addressing one or more parts of the physical structure of the robot, at different scales ranging from groups of modules down to single modules or even individual components in a module. The labels that are defined for a given robot provide a vocabulary for expressing predicates that define other labels. Moreover, labels are our means for expressing conditions under which certain behaviors must be active, allowing us to relate form to behavior in a systematic way.

### B. Defining labels

A label is defined in a programming language according to a spatial model. The programming language provides a means of expressing the conditions under which a given label can be assigned to some part of the robot. The spatial model defines the physical scale at which labels can be defined and the spatial properties that can be used as conditions for when the label is attached to part of the structure. For a concrete robot, a label may occur zero, one, or many places in the structure; specifying the multiplicity of a label can be useful, for example to enforce that only a single module is assigned a specific label that identifies the leader of a group of modules.

How a label is defined depends on the programming language being used to express the labelling of the robot. A relatively low-level, operational language like C could for example use an API that would allow the programmer to explicitly assign labels to specific parts of the robot. At the other extreme, a declarative domain-specific language like LDP could allow labels to be assigned to modules based on conditions [18]. We observe that in the presentation of LDP by De Rosa et al, module-local variables are used ad-hoc on each module to indicate labels such as the seed of a tree or the role played in a metamodule. In the concrete example of the next section, we show how the language RDCD allows labels, in the form of roles, to be declaratively specified using predicates on the spatial model.

The physical scale includes individual components such as individual sensors or actuators with a specific orientation, modules, and hierarchical groupings of modules. For example, a label might be used to identify the forwards proximity sensors of a car robot or the modules that serve as wheels. A label can also be used to identify a local group of modules, such as the modules that constitute a metamodule or a hierarchical structure such as a leg built from a number of smaller groups forming joints.

The spatial properties that can be used as conditions for defining labels depend on the morphological properties of the robot and the runtime services it provides for determining shape information. For example, when the ATRON robot is confined to 90-degree rotations of the main actuator, all modules can be assigned relative 3D coordinates and a relative compass direction for the main rotational axis. Conversely, computing coordinates and compass directions for the Odin robot is difficult due to the flexible design of the joints, for which reason counting the number of connections and the minimum distance between modules might be a more appropriate model. Even more extreme, languages such as LDP and Meld [19] have been designed for robots where the only property that can be determined is the distance between modules (including whether they are neighboring).

### C. Programming with labels

Given a set of labels that are defined on the structure of a modular robot, behaviors can be defined for the robot that abstract over the concrete structure. In principle the use of labels does not impose any constraints on the design of the language used to specify behaviors. Again, labels could for example be addressed using a C API or be integrated into the language as has been done for RDCD.

Using labels provides a useful language design stratification where the structure- and behavior-definition sublanguages can vary independently. In some cases the behavioral language might not be used at all if the structure declarations alone can specify the shape of a robot and thereby implicitly the way it should change shape, as is the case for Meld [19]. In any case, labels are a convenient means of naming entities in both the structure declaration and behavior specification parts of a program. If the structure- and behavior-definition languages are tightly integrated, the labels are useful as a concept for relating structure and behavior.

## IV. Concrete Example: ATRON

We now provide a number of concrete examples of defining and using labels on the ATRON robot; we have not yet made any concrete experiments with defining labels on the Odin robot, but we discuss the issue of generalization in Section V.

### A. Defining labels

As a first example, consider the role-based language RDCD which we have implemented for the DCD-VM. This languages uses a simple, declarative model for specifying the conditions under which a module should be assigned a given label. The BNF for the part of RDCD that concerns definition of labels is shown in Figure 3. Currently, RDCD only supports labels in the form of roles that are assigned at the level of granularity of a module. A role defines a number of invariants on a module and its immediate neighbors and activates the associated behaviors when the invariants are satisfied [11]. Roles are organized into a hierarchy using the keyword `extends`, similarly to Java classes; the most specific role in the hierarchy is automatically selected at runtime based on the invariants. In labelling terminology, the role label is automatically assigned to the module when the invariants are satisfied. For example, to define the parts of the structure that should labelled as a wheel (for a car robot such as the one in Figure 1), one could specify the following constraints:

```
center == EAST_WEST;
```

```
PROGRAM   ::=  ROLE* ...
ROLE      ::=  ... role NAME extends NAME { MEMBER* }| role NAME modifies NAME { MEMBER* }
MEMBER    ::=  INVARIANT | ...
INVARIANT ::=  EXP ;
EXP       ::=  VAR | NAME ( EXP* ) | EXP BINOP EXP | UNARYOP EXP | ...
VAR       ::=  NAME
VALUE     ::=  NUMBER | PREDEFINED
```

Fig. 3. BNF for the part of RDCD that concerns label definition. Note that for simplicity, commas between function arguments are omitted.

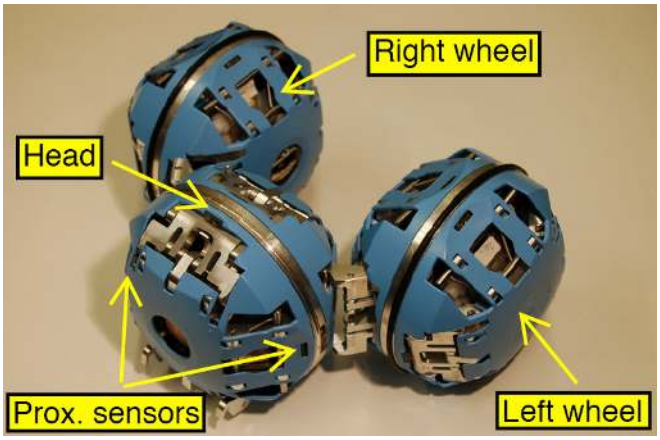
```
#connected(*) == 1;
#connected(UP) == 1;
```

These constraints indicate that the compass direction of the rotational axis of a module should be "east-west" (perpendicular to the direction in which the robot should move) and should have a specific number of connections in total and upwards. We here rely on the DCD-VM to provide relative compass directions for each module, allowing predicates to reason about the relative orientation of each module. To differentiate between the left and right wheels (which must rotate in different directions), more specific predicates can be used that count connections in specific compass directions, as shown in Figure 4 which defines roles for an arbitrary car-like shape. Hierarchical classification is used to define small hierarchies of wheels (left and right) and axles (front and rear). This example also makes used of the keyword `modifies` which designates a so-called "mixin role" that can *modify* another role, similarly to a mixin [20] as known from object-oriented programming: when the invariants in the mixin role and the role that it modifies are satisfied, the associated behaviors are also activated, in effect allowing a module to play multiple roles at the same time. In labelling terminology, additional labels are associated with the same module so long as the additional invariants are satisfied.

The spatial orientation service provided by the DCD-VM relies on the user to designate a single module that is considered the origin of the coordinate system; this module is currently also the module from which code is diffused to the remaining modules of the robot. As an alternative to requiring the user to designate an origin module, such a module can also be found by using the accelerometer when initially assigning labels. This approach works by assigning labels bottom-up using an iterative process that initially relies on e.g. connector counting and then later uses compass directions to refine the label assignments. This use of the accelerometer has however not been integrated into the DCD-VM.

### B. Programming with labels

As a second example, consider defining behaviors in a distributed fashion using labels to address individual module parts inside a group of tightly coordinated modules. We refer to the set of tightly coordinated modules that we program as a *behavior group*. Each statement in the distributed scripting language is prefixed which the label of the components that are going to perform the statement. The same script-based program is interpreted on all modules in the behavior group, but only statements involving components of the module on which the code is running are actually executed. The scripting language is not currently integrated with a label definition language, the labels definitions are in fact programmed manually for the specific modules that are used in the experiment. Nevertheless, we believe any of the label definition approaches described in the previous subsection could be used.




```
LeftWheel, CMD_ROTATE_CLOCKWISE,
RightWheel, CMD_ROTATE_COUNTER_CLOCKWISE,

Car, CMD_REPEAT, FOREVER,
  Car, CMD_IF_OBJ_PRESENT,
       ProxFrontLeft | ProxFrontRight,
  Car, CMD_IF_OBJ_PRESENT_BEGIN,
    LeftWheel, CMD_ROTATE_COUNTER_CLOCKWISE,
    Car, CMD_DELAY, 20, 0,
    LeftWheel, CMD_ROTATE_CLOCKWISE,
  Car, CMD_IF_OBJ_PRESENT_END,
  Car, CMD_DELAY, 2, 0,
Car, CMD_REPEAT_END
```

Fig. 5. A small piece of script code that allows the three-module ATRON car shown at the top to reverse if its front sensors are activated. The components the labels refer to are shown in the figure above except for `Car`, which refers to all modules in the behavior-group.

In order to test this approach we have implemented a group-behavior that allows two ATRON car robots consisting of three modules to dock with each other [21]. Figure 5 shows a small program fragment from this experiment. This program allows a car, also shown in the figure, to turn on the spot if one of its two frontal proximity sensors are activated. In this example the components have been labelled by hand as indicated in the figure, but we plan to have them labelled automatically as described below. In this program the wheels are initially told to rotate to move the car forward. In the

```
role Wheel { center==EAST_WEST && #connected(*)==1 && #connected(UP)==1; ... }
role LeftWheel extends Wheel { #connected(EAST)==1; ... }
role RightWheel extends Wheel { #connected(WEST)==1; ... }
role Axle { #connected(Wheel)>1; ... }
role FrontAxle extends Axle { #connected(NORTH)==0; ... }
role RearAxle extends Axle { #connected(SOUTH)==0; ... }
role FrontWheel modifies Wheel { #connected(FrontAxle)==1; ... }
```

Fig. 4. RDCD program fragment defining role invariants for arbitrary car, role names emphasized

```
role LeftWheel { #connected(WEST)==0; ... }
role RightWheel { #connected(EAST)==0; ... }
role Head {
 #connected(NORTH)==0;
 label ProxFrontLeft=proximity(NORTH&WEST);
 label ProxFrontRight=proximity(NORTH&EAST);
}
```

Fig. 6. Hypothetical RDCD program used to define labels for the program of Figure 5

following if-statement all modules wait for the distribution of the proximity sensor states and depending on the result either make the left wheel reverse for two seconds or iterate one more time. Labels are used to refer not only to the left and right wheels but also to the car as a whole and to specific proximity sensors that are used for the docking behavior.

Labels are currently assigned manually since the DCD-VM currently does not support distributed interpretation and hence cannot run the program shown in Figure 5. Nevertheless, labelling the modules automatically using an RDCD-like approach could be done using the hypothical code fragment shown in Figure 6. Here, the left and right wheels are defined using a single invariant each, since this is sufficient for unambiguously assigning the role to the correct module. The role for the head module also defines two labels for the proximity sensors, which could then be accessed from any other role definitions. Communication in the distributed interpreter is currently done by broadcast, but we plan to implement a simple routing approach based on gradients: a dependence on a label would form a request that is broadcast throughout the module structure until it reaches a module providing that label, at which time the return path can be used to form a routing table; we however leave the details to future work.

### C. Assessment

The role selection and distributed interpretation examples described in this section are obviously signficantly more compact and easier to implement than a similar program written in C or a similar general-purpose language. The specific contribution of labels is to allow the behaviors to be programmed independently of specific module IDs and orientations. This independence is highly useful in practice, since modules fail and need to be replaced, or are temporarily used for other experiments and are then put back together in a different way. Moreover, the readability is obviously improved through the use of symbolic names. In the example of this section we use a declarative language (the role definition component of RDCD) to define labels, but even if labels are defined manually in a configuration file (as is currently the case for the distributed interpreter), the decoupling of structure and behavior provided by labels obviously provides a signficant advantage compared to embedding the concrete spatial structure directly into the behaviors. Moreover, the location independence provided by the use of labels significantly simplifies programming the modules as a distributed system; the sensor values read on one module are for example implicitly available on the other modules.

## V. Discussion

We have observed that two main concerns in spatial programming languages for modular robots are structure and behavior, and we use the notion of a label as a means of relating the structure and the behavior. In essence, we are concerned with a basic notion programming languages, that of being able to assign symbolic names to entities in the domain of the programming language. Labels are already implicitly used in various approaches based on roles, since a role basically can be described as the combination of a structure-based label definition and behavior definition. Nevertheless, the two can in many cases vary independently, for which reason labels can serve as a useful concept by itself. For example, it would be very interesting to use locally distributed predicates [18] as a means of defining labels on the DCD-VM (the mobile program fragments provided by the DCD-VM might be prove to be useful for this task). The ability to name resources in a distributed system is central to many middleware systems, and allows the programmer to transparently use services independently of their location. We can see such naming as a special case of using labels; a spatial language supporting the definition of labels would be useful for querying the available services in a system, for example supporting the notion of “front” or “rear” proximity sensors.

To generalize the use of labels, we are interested in providing a general-purpose framework for defining labels using spatial properties, with different spatial properties available depending on the concrete robot. A number of properties are useful for most kinds of modular robots, for example the number of connections, the labels associated with neighboring modules, or perhaps the distance to some other modules with a specific label. A heterogeneous robot such as the Odin might also provide more detailed spatial information at a local scale, for example a sensor module

might be equipped with an accelerometer which would provide precise orientation information for the module itself but only partial orientation information for neighboring modules due to the flexible joints of the Odin robot. In addition to the Odin robot, we expect to be able to experiment with other kinds of robots in simulation, using the USSR simulator for self-reconfigurable robots [22], [23].

A general-purpose framework for using labels to program modular, self-reconfigurable robots should however take into account a number of quality-of-service issues: for a given label it is for example useful to know the latency and the degree of connectivity. Latency identifies the amount of time from when a command is sent to a label to when it is received by the corresponding module, which is critical for real-time control (similarly for sensors). Having latency information can allow the controller to adapt to the concrete physical shape of the robot. Similarly, in a self-reconfigurable robot connectivity to a label may go through a module that changes connections often and hence does not provide a very reliable neighbor-to-neighbor communication.

We are currently developing a general-purpose label programming framework in the context of a programming platform supporting interactive and incremental development of autonomous controllers for the ATRON and Odin modular robots. We envision interactive programming of distributed controllers being done by incrementally transitioning from simple remote control of a specific robot configuration to autonomous control of a whole class of similar robot configurations. This incremental transition relies on three key elements: (1) using labels names to globally identify what components (modules, specific sensors, actuators, etc.) to control, (2) describing behaviors using a distributed scripting language such as the one described in Section IV-B, and (3) dynamically updating code in the running system using a virtual-machine approach similar to the DCD-VM described in Section II-B.

## VI. Acknowledgements

This research is funded by the Danish Council for Technology and Production through the project "Morphing Production Lines". The authors also want to thank the people who have contributed with ideas and suggestions: David Brandt, Danish Shaikh, Lars Kristian Lindegaard Mikkelsen, and Jørgen C. Larsen.

## References

[1] 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)*, Knoxville, Texas, USA, 2000, pp. 155–164.

[2] S. Goldstein and T. Mowry, "Claytronics: A scalable basis for future robots," *Robosphere*, Nov. 2004.

[3] 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)*, Sendai, Japan, Sep. 2004, pp. 2068–2073.

[4] 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)*, Takamatsu, Japan, 2000, pp. 2210–2217.

[5] D. Rus and M. Vona, "Crystalline robots: Self-reconfiguration with compressible unit modules," *Journal of Autonomous Robots*, vol. 10, no. 1, pp. 107–124, 2001.

[6] 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*, Orlando, FL, 2006, pp. 2552–2557.

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

[8] M. Yim, "A reconfigurable modular robot with many modes of locomotion," in *Proceedings of the JSME international conference on advanced mechatronics*, Tokyo, Japan, 1993, pp. 283–288.

[9] K. Stoy, A. Lyder, R. Garcia, and D. Christensen, "Hierarchical robots," in *Proceedings of the IROS Workshop on Self-Reconfigurable Robots/Systems and Applications*, 2007.

[10] K. Stoy, 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)*, Tai-Pei, Taiwan, Sep. 2003, pp. 3828–3833.

[11] U. Schultz, D. Christensen, and K. Stoy, "A domain-specific language for programming self-reconfigurable robots," in *APGES 2007 — Automatic Program Generation for Embedded Systems — Workshop Proceedings*, Oct. 2007, pp. 28–36.

[12] U. Schultz, M. Bordignon, D. Christensen, and K. Stoy, "A functional language for programming self-reconfigurable robots," in *Draft Proceedings of The Ninth Symposium on Trends in Functional Programming (TFP)*, May 2008.

[13] A. Lyder, R. F. Mendoza Garcia, and K. Stoy, "Mechanical Design of Odin, an Extendable Heterogeneous Deformable Modular Robot," in *Proc., The IEEE/RSJ International Conference on Intelligent Robots and Systems*, Nice, France, 2008, (to appear).

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

[15] D. Brandt and E. 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)*, Queretaro, Mexico, Sep. 2004, pp. 67–74.

[16] M. Bordignon, K. Stoy, D. Christensen, and U. Schultz, "Towards interactive programming of modular robots," in *Workshop on Self-Reconfigurable Robots/Systems and Applications at IROS'08*, Sep. 2008, (submitted for review).

[17] U. Schultz, "Distributed control diffusion: Towards a flexible programming paradigm for modular robots," in *Proceedings of the First International Conference on Robot Communication and Coordination (ROBOCOMM2007)*. ACM, Oct. 2007.

[18] M. De Rosa, S. C. Goldstein, P. Lee, J. D. Campbell, and P. Pillai, "Programming modular robots with locally distributed predicates," in *Proceedings of the IEEE International Conference on Robotics and Automation ICRA '08*, 2008.

[19] M. Ashley-Rollman, S. Goldstein, P. Lee, T. Mowry, and P. Pillai, "Meld: A declarative approach to programming ensembles," in *Proceedings of the 2007 IEEE/RSJ International Conference on Intelligents Robots and Systems (IROS) 2007*, San Diego, CA, USA, Oct. 2007.

[20] G. Bracha and W. Cook, "Mixin-based inheritance," in *OOPSLA/ECOOP '90 Proceedings*, N. Meyrowitz, Ed. ACM SIGPLAN, 1990, pp. 303–311.

[21] K. Stoy, D. Christensen, D. Brandt, and U. Schultz, "Exploit morphology to simplify docking of self-reconfigurable robots," in *Proc., The 9th Int. Symposium on Distributed Autonomous Robotic Systems*, Tsukuba, Japan, 2008, (submitted for review).

[22] D. Christensen, U. Schultz, D. Brandt, and K. Stoy, "A unified simulator for self-reconfigurable robots," 2008, accepted for publication at IROS'08.

[23] U. Schultz, "Unified Simulator for Self-reconfigurable Robots (USSR)," http://modular.mmmi.sdu.dk/wiki/USSR.