Towards a Robust Spatial Computing Language for Modular Robots (Position Paper)

Ulrik Pagh Schultz
Modular Robotics Lab, University of Southern Denmark
Email: ups@mmmi.sdu.dk

***Abstract*—Self-reconfigurable, modular robots are distributed mechatronic devices that can autonomously change their physical shape. Self-reconfiguration from one shape to another is typically achieved through a specific sequence of actuation operations distributed across the modules of the robot. More generally, control of self-reconfigurable robots requires individual modules to act in specific ways in response to sensor input, and these actions need to be coordinated across the modules of the robot. Robust sequential control and role-based control of individual modules has been experimentally demonstrated using the DynaRole language. DynaRole however only allows simple sequences of distributed operations to be executed, which is suitable for self-reconfiguration sequences but lacks the generality needed to implement more complex behaviors.**

In this position paper we present initial ideas on generalizing the DynaRole language to support a wider range of modular robot control scenarios, while retaining robustness, scalability, and the ability to declaratively address issues pertaining to the spatial composition of the robot.

I. Introduction

Modular robotics is an approach to the design, construction and operation of robotic devices aiming to achieve flexibility and reliability by using a reconfigurable assembly of simple subsystems [1]. Robots built from modular components can potentially overcome the limitations of traditional fixed-morphology systems because they are able to rearrange modules automatically on a need basis, a process known as self-reconfiguration, and are able to replace unserviceable modules without disrupting the system's operations significantly. 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 and the difficulty of handling partial hardware failures in a robust manner. These challenges bring to mind the use of spatial programming techniques to provide robust and scalable control coupled with the physical shape of the robot.

Control of self-reconfigurable robots can broadly be divided into centralized and distributed approaches. The distributed approaches are considered superior compared to centralized approaches due to their robustness and inherent parallelism, but are on the other hand often intractable in terms of controller design. Centralized approaches are more tractable, but have limited robustness due to having a single point of failure. In earlier work, we have investigated the distributed execution of a pre-specified self-reconfiguration sequence in a modular robot [2]. A sequence is specified using a simple, centralized scripting language, which either could be the outcome of a planner or be hand-coded. The distributed controller generated from this language allows for parallel self-reconfiguration steps and is highly robust to communication errors and loss of local state due to software failures. Furthermore, the self-reconfiguration sequence can automatically be reversed if desired. The scripting language is based on the DynaRole role-based language for modular robots [3], but the distributed scripting facility is only superficially integrated with the role-based control principle, which prompts the development of an improved language which integrates roles and robust, distributed execution.

This position paper reviews the existing work on control of modular robots in the context of spatial computing, with a focus on language-based approaches. Based on this review we propose a generalization of the DynaRole language, named RoCoRo (for Robust Collaobrative Roles). This language incorporates a state sharing feature heavily inspired by the MIT Proto language [4], a notion of distributed scopes for delimiting a modular robot into distinct ensembles of closely collaborating modules, and a generalized approach to robust distributed execution.

II. Spatial Computing and Modular Robots

The term spatial computing denotes collections of local computational devices distributed through a physical space, in which: (1) the difficulty of moving information between any two devices is strongly dependent on the distance between them, and (2) the "functional goals" of the system are generally defined in terms of the system's spatial structure [5]. Modular robots are obviously spatial computing systems: computation and actuation is local to the individual module, communication is in general module-to-module (global communication such as radio could be used, but would hamper scalability), and the typical modular robot application has to do with controlling the physical spatial structure of the system. Modular robots are an interesting application area for spatial computing techniques: space and time are critical given the robotic nature of the system, numerous variations of concrete hardware is available for experimenting with programming, and specifying a global behavior that is compiled into local and robust control is considered a key issue.

Modular robotics has a significant inspiration from biological systems, as is also the case for spatial computing. The individual module is here seen as a cell which is part of a larger multicellular organism. In *homogeneous* systems the modules are physically identical but will typically differentiate their behavior depending on their physical position in the structure, whereas in *heterogeneous* systems the modules also have different physical characteristics [6]. Chemical and biological concepts such as gradients, hormones and central pattern generators have been used for robust, scalable control of modular robotic systems, although typically in an ad-hoc fashion with an application-specific implementation in C.

A. *Modular robot hardware*

There are numerous different kinds of modular robots [1]. From the point of view of spatial computing, we can make an overall categorization into macroscale modules, that must be carefully controlled due to motion constraints, and microscale modules, that are typically controlled in a probabilistic way that ignores most if not all physical constraints (such modules so far only exist in simulation). In this paper, we focus on macroscale modules, and we are concerned with the problem of global-to-local compilation of programs for physical modular robots with significant motion constraints, limited processing capacity, and unreliable neighbor-to-neighbor communcation. Microscale modules would typically be more directly amenable to principles of self-organization and mathemetical modelling in general, whereas macroscale modules face many significant implementation issues that must be resolved before these principles become relevant to consider in practice.

As an example of a macroscale module, consider the ATRON self-reconfigurable modular robot (Figure 1), which is our primary experimental platform. The ATRON is a 3D lattice-type system [7]. Each unit is composed of two hemispheres, which rotate relative to each other, giving the module one degree of freedom. Connection to neighboring modules is performed by using its four actuated male and four passive female connectors, each positioned at 90 degree intervals on each hemisphere. The likewise positioned eight infrared ports are used to communicate among neighboring modules and to sense distance to nearby objects. The ATRON exists in two hardware generations: one with an Atmel ATMega128 micro-controller and 4K of RAM per hemisphere, and one with a 1.2MGate FPGA and 64Mb of RAM per hemisphere, in both cases linked by a serial connection. The first generation ATRON is typical of most modular robotic systems: the processing units are severely constrained in order to keep the system simple, realistic to reduce to small size, and potentially cost-effective by mass production. The second generation ATRON is designed as an experimental platform enabling experiments with standard operating systems and programming languages [8].

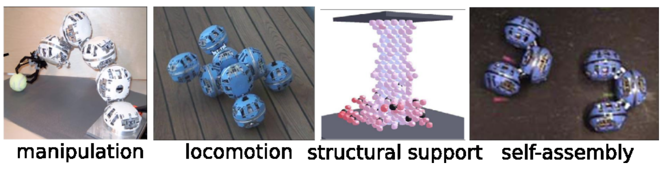

Fig. 1. The ATRON modular robot used for various applications

B. *Self-reconfiguration*

Self-reconfiguration concerns the spatial transformation of the robot morphology from one shape to another. It is typically viewed as a sequence of operations performed by the robot; in some cases self-reconfiguration could be the only operation performed e.g. if performing locomotion based on self-reconfiguration by shifting the modules towards a specific direction in a caterpillar-like motion.

Off-line planning of self-reconfiguration sequences has been studied for a large number of different robotic systems [6], but is largely complementary to the concerns addressed in this paper: we are interested in providing runtime execution support for control of modular robots, including self-reconfiguration. An off-line planner could use the language proposed in this paper as target, and would thus benefit from its features when performing self-reconfiguration.

On-line, distributed self-reconfiguration algorithms address the execution issue that a number of independent modules must coordinate their actions to perform the correct sequence of actions required for self-reconfiguration [9], [10], [11], [12], [13], [14]. Unlike systems which require an off-line plan to be computed first, these algorithms allow self-reconfiguration to be done automatically given a target shape. However, a limitation of the existing, purely on-line distributed algorithms is that neighbor-to-neighbor communication is essential to determine the position of modules relative to each other, and thus a broken communication link, even if it is only one way, is problematic. For example, in the Proteo system by Yim et al., two-way neighbor to neighbor communication is required for coordination between neighboring modules and propagation of heat values in the heat-based method [14].

For modules with motion constraints, scaling self-reconfiguration to large-scale scenarios is often done by the use of *metamodules:* small, flexible *ensembles* (groups) of closely collaborating modules that can move as a unit through the structure of the robot to shape-change the system [15], [16]. Metamodules emerge from the larger robot configuration, move on the surface of other modules, and stop at a new position. The flow of metamodules, from one place to another on the structure of modules, realizes the desired self-reconfiguration. For the ATRON robot, 3 modules are typically combined into a metamodule; a central module plays the role of a "leg" whereas the two others are attached as "feet". Programming metamodules has so far been done in a low-level manner using a combination of local actions executed by specific metamodules and global information propagated throughout the structure, such as a gradient serving as attractor for the flow of metamodules [17].

C. *Biologically inspired locomotion*

Whereas self-reconfiguration typically serves the purpose of transforming the robot between configurations, locomotion is typically performed by actuating modules in a fixed configuration, for example using gait tables [6]. We here review two examples of biologically inspired locomotion that relate both to spatial computing and to the RoCoRo language proposed in this paper. In both cases, locomotion is achieved by propagating timed communication signals through the spatial structure of the robot.

Shen et al. investigates a hormone-inspired approach to communication and control in the CONRO self-reconfigurable robot, where a set of communication signals triggers different behaviors in modules [18]. Hormone signals are packets that are diffused throughout the structure of the robot, possibly causing operations to activate or new hormones to be created when they arrive. This is similar to chemical diffusion, which has also been used as the basis for spatial computing systems [19], [20], [21], and has been shown to be an effective basis for decentralized communication and execution of programs in spatial computing systems. Shen et al. demonstrate experimentally how hormones can be used to control locomotion and self-reconfiguration of physical modular robots in a highly dynamic fashion that automatically adapts to the current topology. Self-reconfiguration is performed using a "cascade of actions" that in execution is similar to the distributed sequences of DynaRole.

In the work of Stoy et al, a lizard-like structure with four legs is programmed using a primitive form of role-based control where modules respond differently to a time-pulse stimuli that propagates through the structure [22]. The behavior of each module and its response to communication is given by its position in the robot, such as "head", "leg", or "spine". Concretely, roles are used to express how modules interpret sensors and events, and the behavior of each module of the robot at any given time is driven by a combination of its role and timed signals propagated through the structure. In this work, the sole focus is on performing cyclic behavior for locomotion, there is no support for coordination or for performing sequences of actions in response to events.

D. Language-based approaches

The self-reconfiguration and locomotion techniques presented thus far all follow a high-level pattern, but are to the author's knowledge in every case implemented using complicated low-level code that is difficult to reuse in a different scenario. Recently, language-based approaches have however been used in the attempt of creating succinct and reusable software for controlling modular robots.

Locally Distributed Predicates [23], [24] and Meld [25] are two declarative programming languages specifically developed to support the operation of large-scale modular robots composed of spherical microrobots [26] that form a self-reconfigurable spatial computing system. The declarative style of these languages enables complex behaviors of subsets of modules to be derived from concise specifications of spatial constraints. The feasibility of executing these languages on resource-constrainted modular robots has however not been addressed. Moreover, from a language design point of view, the declarative style is perhaps not ideal for specifying complex sequences of operations, as the actual operations to be performed are the result of constraint resolutions as opposed to programmer-specified behavior. This is an open issue that we return to later in this paper. We note that while the context and purpose are similar to the work presented in this paper, a significant difference is the number of modules that robots are anticipated to comprise: The spherical microrobots are assumed to exist in numbers several order of magnitudes higher than macroscale modular robots such as the ATRON. Million-module structures are an ideal match for the typical spatial computing scenario, and can afford to overlook reliability issues that we are intereted in addressing: in the typical macroscale scenario, a single failing module can disrupt locomotion or a whole self-reconfiguration sequence, and must thus be taken into account, while in a highly-redundant context, the same occurrence is often not as significant and can in many cases be ignored due to physical redundancy.

The DynaRole language is designed for role-based control of macroscale modular robots [3]. The DynaRole language is a role-oriented language that allows the programmer to use roles to declaratively specify how behaviors are deployed and activated in the modular robot as a function of its spatial layout and, similarly to the idea of role-based control, how each module responds to sensor inputs and communication. DynaRole programs run on a virtual machine that enables fast and incremental on-line updates of programs, allowing the programmer to interactively experiment with the physical robots. The use of roles allows behaviors to be organized into modules that again are organized into an inheritance hierarchy, providing both reuse and behavioral specialization. Nevertheless, the language provides no support for specifying behaviors at a global level, the underlying virtual machine assumes reliable communication, and in general there is no robustness towards partial failures. Role selection is based on declarative spatial specifications e.g. identifying wheel modules as "modules that have a horizontal rotation axis, only have a single connection, and are at y coordinate 0". The declarative selection primitives and 3D coordinate computations are specific to the robot kinematics, but are in fact automatically generated based on the geometrical description of a single module in the M3L kinematics language [27], which when combined with spatial labels [28] enables morphology-independent programming of modular robots [29].

To enable DynaRole to be used for self-reconfiguration, we extended the language to support robust execution of distributed sequences of operations [2]. Specifically, self-reconfiguration sequences are compiled to a robust and efficient implementation based on a distributed state machine that continuously shares the current execution state between the modules of the robot. Dependencies between operations are explicitly stated to allow independent operations to be performed in parallel while enforcing sequential ordering between actions that are physically dependent on each other. The language is *reversible* meaning that for any self-reconfiguration sequence the reverse one is automatically generated: due to the sequential nature of the programs, any self-reconfiguration process described in the language is reversible by simply performing the corresponding inverse operations in reverse order. Reversibility is subject to physical constraints such as gravity, changes in the environment, and hardware failures. The continuous diffusion of the state of each module to its neighboring modules provides a high degree of robustness towards partial failures: one-way communication links still serve to propagate state throughout the structure, and modules that are reset (e.g., due to hardware issues or by a watchdog-based timer) are automatically restored from the neighboring modules. Nevertheless, the distributed sequences are extremely simple, there are no conditionals, loops, or propagation of any state except how far the sequence has executed.

III. Analysis

The extension of DynaRole to support execution of distributed sequences provided a significant increase in robustness, which was demonstrated both with (relatively) long-running, reversible self-reconfiguration experiments using physical ATRON modules, and a comprehensive set of self-reconfiguration experiments using simulated ATRON modules (and simulated M-TRAN [9] modules) [2]. One of the primary challenges in programming the ATRON is ensuring robustness towards partial hardware failures in communication, for example two-way communication links that only provide one-way connectivity due to misaligned infrared transceivers. Due to the continuous state diffusion, execution in theory works as long as for any two modules there exists a communication path between them in the robot. The path needs neither to be reliable nor to be static. On the other hand, as mentioned earlier, the sequences cannot react to changes in the environment and are not really integrated with the role-based behavior specification language.

DynaRole sequences could be made more general by adding support for shared program state and conditionals. Shared program state could be diffused similarly to how the distributed sequence progression is shared. In the specific case of sequence progression, each module is reponsible for merging the global state received from neighboring modules with the local state — for arbitrary program state this would have to be handled by the programmer. Such a state sharing approach is inspired by and bears many similarities to MIT Proto [4]: there is not necessarily a single, consistent global state, rather each module continously computes its own view of the shared state. Given that changes to state and progression of execution are propagated together, different parts of the robot may have different views on the state of the sequence, but each of those views will be consistent and will ultimately converge if conditionals are guaranteed to always take the same branch. Indeed, for conditionals the primary challenge is to handle the case where different modules executing parts of the same sequence would take different branches due to sensor inputs or local copies of a shared state having different values. More generally, there is also the question of when to start the execution of a distributed sequence: since the sequence typically involves operations that modify the physical state of the robot, running more than one sequence at a time is usually not relevant. A solution to both of these issues is to delegate the responsibility of triggering sequences and testing conditionals to a single module in the structure. This provides a simple semantics perfectly suitable for e.g. local creation and control of metamodules, but at the obvious cost of limitations in scalability and robustness.

The Meld and LDP language have been designed for controlling subsets of modules within the larger structure. Declarations are used to identify subsets of modules that perform specified operations over time. This approach is obviously required for scalability to larger scenarios, and is essential for supporting the concept of metamodules, which is a proven way of controlling larger-scale ATRON structures. In these scenarios, module groups must be created and dissolved at runtime. A similar scenario is that of self-assembly of modular robots [30]. Here, a larger ensemble is built from smaller groups of modules that become dynamically connected, but the reverse operation splits up the ensemble into smaller groups, each forming their own ensemble. In all these cases, the module subsets can be seen as a dynamic scope delimiter for execution and state propagation. This scope identifies modules that are sharing state and optionally are participating in the execution of a distributed sequence of actions. (This notion of a distributed scope has many similarities to logical neighborhoods [31].)

The design of domain-specific languages often exhibits a tension between declarative and imperative styles of programming. Unlike Meld and LDP which are purely declarative, DynaRole favors a mixed style where declarations are used to control the selection of behaviors in response to the spatial layout of the robot, whereas the behaviors themselves are in an imperative style, similarly the growing point language [19]. We believe the mixed style to be most well-suited to the task of programming modular robots, but this remains an open issue in the design of programming languages for modular robots in particular and for spatial computing in general.

IV. Towards The RoCoRo Language

A. Introducing RoCoRo

We propose the RoCoRo (Robust Collaborative Roles) language as a generalization of the DynaRole language, intended for robust, general-purpose control of modular robots. The language has two primary abstractions: ensembles and roles. An *ensemble* is a dynamic, distributed scope the covers a number of modules and introduces shared state and distributed behaviors into these modules. A *role* applies to a single module, and introduces local state and local behaviors into the module. Roles are further divided into *primary roles* of which only one can be active on a given module at a given point in time, and *mixin roles* of which any number can be active on a given module at a given point in time. A module can be a member of any number of ensembles at a given point in time. Ensembles and roles together are referred to as *entities*, and the set of entities active on a given module is called its *activation*. Declarative rules are used to control the activation of entities based on spatial constraints, the entity activations on neighboring modules, local state from roles, and shared state from ensembles. Entities can be specialized with a semantics

```
abstract ensemble GradiField {
  int g = @MAX_INT;
  g.update {
    min = @MAX_INT;
    for(ng: GradiField.g) min = Math_min(min,ng);
    if(g<@MAX_INT) g = min+1;
  }
}
mixin role GradiSource within GradiField { g.update { g = 0; } }
```

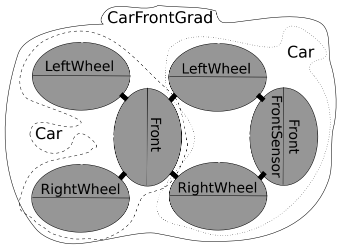

Fig. 2. Gradient field in RoCoRo applied to a configuration of two docked cars using concrete instantiations `CarFrontGrad` and `FrontSensor` which specialize (instantiate) `GradiField` and `GradiSource` respectively. The docked car configuration moreover contains two `Car` ensembles which each consists of three modules playing car-specific roles.

resembling standard object-oriented inheritance: members can be added and existing members can be overridden.

B. *RoCoRo by example: gradients*

As a classical spatial computing example also relevant in modular robotics, consider the implementation of a simple gradient field in RoCoRo, shown in Figure 2. The ensemble `GradiField` is used to describe the scope of the gradient field, it introduces a shared variable `g` and provides an update rule for `g` which continuously updates the value of `g`. The update rule accesses the available (cached) values of `g` from the neighboring modules, and uses these to compute the local gradient value (the `@` sign indicates an external constant). The mixin role `GradiSource` can be activated on some module that already is part of the `GradiField` ensemble, and overrides the update rule from the ensemble to always make the local gradient value be zero, making the module a source in the gradient field. Member overriding depends on the order at which the entities were activated at runtime, in this case since the role is created within a pre-existing ensemble, the update rule from the ensemble would necessarily be overridden by the update rule from the role. Both entities are however declared abstract, meaning they cannot be activated without first making a specialization for a concrete scenario.

As an example of a specialization of the gradient entities, consider a gradient field for an arbitrary car vehicle; this gradient field is illustrated for a configuration of two docked cars in Figure 2. Assume that the modules in each vehicle are subroles of `Front` or `Wheel`, and that the gradient source should be modules playing the role of `Front` and have no forwards (“north”) connections. In this case, the following specialization will activate the gradient field:

```
ensemble CarFrontGrad extends GradiField {
 require subrole(Front) || subrole(Wheel);
}
mixin role FrontSensor extends GradiSource {
 require subrole(Front) && connected(NORTH)==0;
}
```

The ensemble specialization adds a “require” declaration which specifies under what conditions the ensemble can be activated on a given module, and similarly for the mixin role specializing the gradient source.

```
enum Obstacle { None, Left, Right, Center }

ensemble Car {
  // State shared between all modules
  Obstacle obstacle = Obstacle.None;

  // Distributed control behavior
  behavior Front.move() {
    Front.if(Car.obstacle==Obstacle.None) {
      Wheel.drive(); Wheel!evade();
    }
    else {
      Wheel.evade(); Wheel!drive();
    }
  }
}

role Front within Car {
  // Needs 2 neighbors
  require connected(@COMPASS_ANY)==2;
  // Continuously monitor proximity
  behavior checkProximity() {
    if(isProximity(@FRONT_LEFT) &&
      isProximity(@FRONT_RIGHT)) {
      obstacle = Obstacle.Center;
    } else { ... }
  }
}

abstract role Wheel within Car {
  abstract constant Obstacle MY_SIDE;
  abstract constant Compass CONNECTED_SIDE;
  // Require 1 connection + break symmetry
  require connected(CONNECTED_SIDE==1)
      && connected(@COMPASS_ANY)==1);
  // Activated as behaviors by Car.move
  void drive() {
    self.rotateContinuous(100,1);
  }
  void evade() {
    if((obstacle==MY_SIDE)) {
      self.rotateContinuous(50,0);
    } else {
      self.rotateContinuous(100,0);
    }
  }
}

role LeftWheel extends Wheel { ... }
role RigthWheel extends Wheel { ... }
```

Fig. 3. Two-wheeler ATRON car obstacle evasion in RoCoRo

C. RoCoRo by example: obstacle avoidance

One of the primary design goals of RoCoRo is to allow modular robots to be controlled in a robust manner based on a global description of the behavior. As an example of this, consider obstacle avoidance for the small ATRON cars from Figure 1 (rightmost picture), depicted schematically in a docked configuration in Figure 2: a "Front" module in the middle and two "Wheel" modules on the sides. An obstacle evasion program for such a two-wheeler car robot is shown in Figure 3. The ensemble `Car` is used to define the scope of the car; the ensemble is non-abstract and defines no requirements, and so is automatically activated on every module of the robot. This ensemble defines a shared variable `obstacle` (of an enum type, similarly to e.g. C or Java) and a shared behavior `move` which can only be initiated by modules playing the role `Front`.[1] Shared behaviors execute as distributed sequences of operations across the modules of the robot. In this example, the `move` behavior is a global description that expresses the coordination between the various modules of the ensemble; since it only has two steps it could however also have been implemented as a behavior in the role `Front`, but the chosen design arguably makes the overall behavior of the robot more clear. In general, an ensemble behavior can consist of several steps which execute as a robust sequence.

Behaviors are always initiated continuously and atomically to the entity in which they are declared (e.g., for a given role or ensemble, only a single behavior runs at a given point in time). Behavior initiation is decided by a scheduler on the individual module, whether defined on an ensemble or on a role. This is also the case for the behavior `move` which starts with a test on the shared variable `obstacle`. Depending on the value of `obstacle`, the sequence either activates the method `drive` and deactivates the `evade` method or alternatively the inverse, and it does this on all modules of the robot playing the role `Wheel` or a subrole. Activating a method means that it acts like a behavior, that is, it is continuously activated on the respective modules. Deactivating a method means that it stops acting like a behavior. For a given module, this method activation is subject to the state of the distributed sequence being propagated to this module.

The role `Front` defines the requirements for role activation (connected to two modules, the wheels) and its behavior which is to continuously monitor the proximity sensors and update the shared variable `obstacle` correspondingly. Note that no update rules are defined for the shared variable, this means that the default update rule is used, which simply overwrites the local value with the most recent value received from the neighbors, unless the variable was assigned locally in which case it no longer updates automatically but will start to propagate to its neighbors. (If the variable is assigned multiple places in the robot, modules that have not assigned a value to the variable will receive different values at different times through state propagation.) The abstract role `Wheel` defines the conditions under which a wheel is activated as well as how it behaves when the methods `drive` or `evade` are activated. The exact requirement and behavior depends on whether it is a left or a right wheel, which is described by the abstract constants that are defined by the concrete subroles for left and right wheels (not shown). The method `evade` uses a slower rotation speed if the obstacle is on the same side as the wheel, which causes the car to turn away from the obstacle.

[1]If there were multiple modules playing this role, the behavior could be initiated in multiple places at once, dealing with this issue is considered future work.

D. RoCoRo runtime behavior

The RoCoRo language is built on the idea of continuous state propagation by diffusion to neighboring modules. Roles react to changes in the environment that they receive through diffusion, both in terms of what methods are activated and in terms of what role should be active on the module. The shared state from ensembles is also propagated using diffusion, and the local update rules (explicit or implicit) define how the state is merged.

The shared behaviors execute similarly to the distributed sequences from DynaRole: once initiated, each operation in the behavior explicitly denotes the module where it should execute. A program counter denoting the set of parallel executing operations is shared between all modules executing the sequence, and is advanced when modules begin and complete operations. Here, the activation of a method is instantaneous and does not wait for the method to run; rather, the method is activated and will start to run the next time state propagation is performed, and will continue to do so until deactivated.

State propagation is not assumed to be reliable, on the contrary the language is designed for operation of modular robots with unreliable communication links, such as the ATRON. Changes to the module activation, updates to shared variables, activation and deactivation of methods, and progress in the execution of a distributed sequence propagates asynchronously throughout the module structure, and only when the underlying communication system has succeeded in propagating information through a communication link. For consistency, complete information about the state of a module is transmitted in a single packet, but this is problematic on a system like the ATRON where the older generation modules cannot reliably transmit more than roughly 100 bytes of information per packet. We leave this issue to future work.

E. Implementation status

A complete RoCoRo frontend and code generator for Java source code is currently being implemented using the xtext eclipse framework. The generated code assumes a high-level object-oriented runtime system, which is being implemented on top of the USSR generic simulation framework for modular robots [32]. Unlike earlier work [3], this implementation does not address the issue of code distribution in any way, this is considered future work. Moreover, there currently is no underlying spatial information framework, meaning that only very simple predicates can be used to query the physical structure of the robot. Our plan is however to integrate the

M3L language [27] with the simulator to enable automatic generation of new robot implementations from M3L declarations. Such generated robot implementations would be automatically equipped with the ability to compute precise spatial information based on the M3L generation of forward kinematics.

V. Discussion

There are numerous issues that must be resolved before RoCoRo can be used for large-scale scenarios like self-assembly and metamodules. In particular, the semantics of ensembles needs to be defined such that multiple ensembles of the same type can exist in the same robot. We expect that the existing work on logical neighborhoods [31] will be a useful source of inspiration. The RoCoRo language has been pragmatically designed for robust control of modular robots, and we expect that it can be applied as a generally useful programming language for modular robots. The question of how well RoCoRo is suited to other spatial computing tasks, such as programming of sensor networks or swarm robotics, is in an interesting one that will be explored in future work.

Acknowledgment: I would like to thank the anonymous reviewers for their insightful and constructive comments.

References

[1] M. Yim, W.-M. Shen, B. Salemi, D. Rus, M. Moll, H. Lipson, E. Klavins, and G. S. Chirikjian, "Modular Self-Reconfigurable Robot Systems [Grand Challenges of Robotics]," *IEEE Robot. Automat. Mag.*, March 2007.

[2] U. P. Schultz, M. Bordignon, and K. Stoy, "Robust and reversible execution of self-reconfiguration sequences," *Robotica*, vol. 29, pp. 35–57, 2011.

[3] M. Bordignon, K. Stoy, and U. P. Schultz, "A Virtual Machine-based Approach for Fast and Flexible Reprogramming of Modular Robots," in *Proc. IEEE Int. Conf. on Robotics and Automation (ICRA'09)*, Kobe, Japan, May 12-17 2009, pp. 4273–4280.

[4] "MIT proto," retrieved March 8, 2012. Software available at http://proto.bbn.com/.

[5] J. Beal, S. Dulman, J.-L. Giavitto, and A. Spicher, "Spatial computing workshop 2012 call for papers," 2012, http://www.spatial-computing.org/scw12:start, downloaded April 15th 2012.

[6] K. Stoy, D. Brandt, and D. J. Christensen, *An Introduction to Self-Reconfigurable Robots*. MIT Press, 2010.

[7] E. Østergaard, K. Kassow, R. Beck, and H. Lund, "Design of the ATRON lattice-based self-reconfigurable robot," *Autonomous Robots*, vol. 21, no. 2, pp. 165–183, 2006.

[8] M. Moghadam, D. Christensen, D. Brandt, and U. Schultz, "Towards Python-based DSL languages for self-reconfigurable modular robotics research," in *2nd Int. Workshop on Domain-Specific Languages and models for ROBotic systems (DSLRob'11)*, 2011.

[9] E. Yoshida, S. Murata, H. Kurokawa, K. Tomita, and S. Kokaji, "A distributed method for reconfiguration of a three-dimensional homogeneous structure," *Advanced Robotics*, no. 13, pp. 363–379, 1999.

[10] C. Ünsal, H. Kiliccöte, and P. K. Khosla, "A modular self-reconfigurable bipartite robotic system: Implementation and motion planning," *Autonomous Robots*, no. 10, pp. 23–40, 2001.

[11] Z. Butler and D. Rus, "Distributed planning and control for modular robots with unit-compressible modules," *The International Journal of Robotics Research*, no. 22, pp. 699–715, 2003.

[12] M. D. Rosa, S. Goldstein, P. Lee, J. Campbell, and P. Pillai, "Scalable shape sculpting via hole motion: Motion planning in lattice-constrained modular robots," in *Proc. of the 2006 IEEE Int. Conf. on Robotics and Automation (ICRA'06)*, 2006.

[13] S. Murata, H. Kurokawa, and S. Kokaji, "Self-assembling machine," in *Proc. of the 1994 IEEE Int. Conf. on Robotics and Automation*, 1994, pp. 441–448.

[14] M. Yim, Y. Zhang, J. Lamping, and E. Mao, "Distributed control for 3d metamorphosis," *Auton. Robots*, vol. 10, no. 1, pp. 41–56, 2001.

[15] K. C. Prevas, C. Unsal, M. O. Efe, and P. K. Khosla, "A hierarchical motion planning strategy for a uniform self-reconfigurable modular robotic system," in *Proc. of the IEEE Int. Conf. on Robotics and Automation (ICRA)*, May 2002.

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

[17] D. J. Christensen, "Experiments on fault-tolerant self-reconfiguration and emergent self-repair," in *Proc. of Symposium on Artificial Life part of the IEEE Symposium Series on Computational Intelligence*, Honolulu, Hawaii, Apr. 2007.

[18] W.-M. Shen, B. Salemi, and P. Will, "Hormone-inspired adaptive communication and distributed control for conro self-reconfigurable robots," *IEEE Transactions on Robotics and Automation*, vol. 18, pp. 700–712, 2002.

[19] D. Coore, "Botanical computing: A developmental approach to generating interconnect topologies on an amorphous computer," Ph.D. dissertation, MIT, 1999.

[20] R. Nagpal, "Programmable self-assembly: Constructing global shape using biologically-inspired local interactions and origami mathematics," Ph.D. dissertation, MIT, 2001.

[21] M. Viroli, M. Casadei, S. Montagna, and F. Zambonelli, "Spatial coordination of pervasive services through chemical-inspired tuple spaces," *ACM Trans. Auton. Adapt. Syst.*, vol. 6, no. 2, pp. 14:1–14:24, June 2011. [Online]. Available: http://doi.acm.org/10.1145/1968513.1968517

[22] K. Stoy, W.-M. Shen, and P. Will, "Implementing configuration dependent gaits in a self-reconfigurable robot," in *Proc. of the 2003 IEEE Int. Conf. on Robotics and Automation (ICRA'03)*, Tai-Pei, Taiwan, Sept. 2003, pp. 3828–3833.

[23] 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 2008 IEEE International Conference on Robotics and Automation (ICRA'08)*, Pasadena, CA, USA, May 19-23 2008, pp. 3156–3162.

[24] M. De Rosa, S. C. Goldstein, P. Lee, J. Campbell, and P. S. Pillai, "Detecting locally distributed predicates," *ACM Trans. Auton. Adapt. Syst.*, vol. 6, no. 2, pp. 13:1–13:14, June 2011. [Online]. Available: http://doi.acm.org/10.1145/1968513.1968516

[25] M. P. Ashley-Rollman, S. C. Goldstein, P. Lee, T. C. Mowry, and P. Pillai, "Meld: A Declarative Approach to Programming Ensembles," in *Proceedings of the 2007 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS'07)*, San Diego, CA, USA, October 29 - November 2 2007, pp. 2794–2800.

[26] S. C. Goldstein, J. D. Campbell, and T. C. Mowry, "Programmable Matter," *IEEE Computer*, vol. 38, no. 6, pp. 99–101, June 2005.

[27] M. Bordignon, U. P. Schultz, and K. Stoy, "Model-based Kinematics Generation for Modular Mechatronic Toolkits," in *Proc. 9th ACM SIGPLAN/SIGSOFT Int. Conf. on Generative Programming and Component Engineering (GPCE'10)*, Eindhoven, The Netherlands, October 10-13 2010.

[28] U. P. Schultz, M. Bordignon, D. J. Christensen, and K. Stoy, "Spatial Computing with Labels," in *Proc. SASO'08 Spatial Computing Workshop (SCW'08)*, Venice, Italy, October 20 2008.

[29] M. Bordignon, K. Stoy, and U. Schultz, "Generalized programming of modular robots through kinematic configurations," in *Proc. of the 2011 IEEE/RSJ Int. Conf. on Intelligent Robots and Systems (IROS)*, 2011, pp. 3659–3666.

[30] K. Stoy, D. J. Christensen, D. Brandt, M. Bordignon, and U. P. Schultz, "Exploit morphology to simplify docking of self-reconfigurable robots," in *Proc. Int. Symp. on Distributed Autonomous Robotic Systems (DARS'08)*, Tsukuba, Japan, 2008, pp. 441–452.

[31] L. Mottola and G. Picco, "Logical neighborhoods: A programming abstraction for wireless sensor networks," in *Distributed Computing in Sensor Systems*, 2006, pp. 150–168.

[32] D. J. Christensen, D. Brandt, K. Stoy, and U. P. Schultz, "A Unified Simulator for Self-Reconfigurable Robots," in *Proc. IEEE/RSJ Int. Conf. on Intelligent Robots and Systems (IROS'08)*, France, 2008, pp. 870–876.