Chapter 16

Autonomous Discovery and Functional Response to Topology Change in Self-Reconfigurable Robots

Behnam Salemi, Peter Will and Wei-Min Shen
Information Sciences Institue
Department of Computer Science
University of Southern California
{salemi, will, shen}@isi.edu

1.1. Introduction

A self-reconfigurable system is a special type of complex systems that can autonomously rearrange its software and hardware components and adapt its configuration, such as shape, size, formation, structure, or organization, to accomplish difficult missions in dynamic, uncertain, and unanticipated environments. A self-reconfigurable system is typically made from a network of homogeneous or heterogeneous *reconfigurable modules* (or *agents*) that can autonomously change their physical or logical connections and rearrange their configurations.

Self-reconfigurable robots [Yim 2002, Rus 2002, Shen 2002a, Shen 2002b] are examples of such self-reconfigurable systems that consist of many autonomous modules that have sensors, actuators, and computational resources. These modules are physically connected to each other in the form of a configuration network. Since the topology of the configuration network may change from time to time, to accomplish a given task, the controller of the Self-reconfigurable robot must be distributed and decentralized to avoid single-point of failures, and communication bottleneck among modules.

These modules must have some essential capabilities in order to accomplish complex tasks in dynamic and uncertain environments. The capabilities that we addressed in our previous work were: (1) distributed task negotiation [Salemi 2003] – allowing modules to agree on a global task which is to be accomplished, (2) distributed

behavior collaboration [Salemi 2004] – allowing modules to "translate" a global task into local behaviors of modules; (3) synchronization – allowing modules to perform local behaviors in a coordinated and timely fashion; In these previous works we assumed the network of modules can have any initial topology but it remains unchanged during the process of accomplishing a selected task.

Here we relax this assumption and allow the topology of the network of modules to change at any time including the duration of accomplishment of the task. Our proposed solution for this problem is a distributed approach inspired by the concept of hormones [Salemi 2001] and is based on 1) giving the ability of detecting local changes in the topology of the network to the modules and 2) Allowing them to select and coordinate new behaviors when the topology of the network changes such that the selected global task can be accomplished.

The related approaches for solving similar problems include Role-based Control [Stoy 2002] and stochastic approaches for self-repair such as [Murata 1998]. The first approach is based on detecting the changes in local relationships of the immediate neighboring modules and local reaction to these changes. This approach does not require a lot of computational power. However, it is an open-loop approach and might not be suitable for accomplishing complex tasks. The second approach has been applied to lattice-based self-reconfigurable robots which their configuration space is much smaller than that of the chain-type self-reconfigurable robots such as CONRO.

This chapter is organized as follows: Section 2 defines the problem of autonomous discovery and functional response to topology changes and introduces CONRO self-reconfigurable robots as an illustrative example; Section 3 presents the idea of probing for solving the problem of autonomous discovery and functional response to topology changes; Section 4 presents the FEATURE algorithm and reviews its sub-algorithms; Section 5 gives examples of applying the FEATURE algorithm to real CONRO modules, and Section 6 concludes the chapter with future research directions.

1.2. Autonomous Discovery and Functional Response to Topology Changes

The problem of autonomous discovery and functional response to topology changes can be defined as follows: Given a global task and a set of self-reconfigurable modules, coordinating global responses to local changes in the topology of the network of modules in order to produce the desired global effects. Local changes include adding or deleting new modules or communication links to/from the network of modules.

This problem is very challenging due to several reasons: relationships among modules may change anytime; changes in configuration is locally detectable but a coordinated global response is required; the number of modules in the robot is not known; modules have no unique global identifiers or addresses; modules do not know the global configuration in advance, and can only communicate with their immediate neighbors.

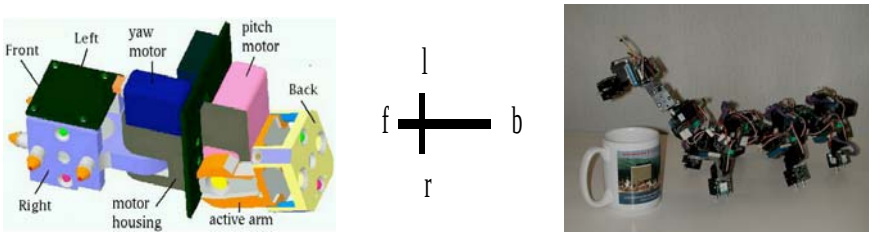

Figure. 1. A CONRO module, the schematic view of one module, and a hexapod configuration with 11 modules.

Generally, accomplishing a given global task is dependent on the topology of the network of modules [Salemi 2004]. Modules can accomplish a global task by selecting correct *behaviors* in coordination with other modules and performing them synchronously. As a result, changes in the topology will directly influence the behaviors that should be selected and the time they should be performed.

Formally, an autonomous discovery and functional response to topology changes problem is a tuple [$G(P, C)$, Q, T, B], where G is the *configuration graph* of the network of modules consisting of P, a list of nodes, p_i and C, a list of labeled physical or logical links, c_j, such that $j \in \{locally\ unique\ labels\}$; Q is the list of the internal states, q_i, associated with each node p_i, such that $i \in \{1,\ldots, N\}$; T is the global task shared by all nodes; and B is a set of behaviors, b_i, available to the nodes. In this situation, an autonomous discovery and functional response to topology changes problem is solved if and only if there is a function, $f(G,Q) \rightarrow B$, that it is a mapping from the current topology of the configuration graph, and the internal state of a node to a sequence of behaviors that can accomplish task T. Here, the nodes and links represent the modules and the communication links between them, respectively. Note that the size of the network is dynamic and unknown to the individual nodes; also the index numbers are only used for defining the problem and not used in the solution.

Under these circumstances, a satisfactory solution to this problem must be distributed. Modules must detect local changes in the configuration graph and inform the rest of the modules in order to let them select the correct behavior.

To illustrate this problem, we use the CONRO self-reconfigurable robot as an example. CONRO is a chain-type self-reconfigurable robot developed at USC/ISI (http://www.isi.edu/robots). Figure 1 shows the schematic views of CONRO module and a six-legged CONRO robot. Each CONRO module is autonomous and contains two batteries, one STAMP II-SX micro-controller, two servo-motors, and four docking connectors for connecting with other modules. Each connector has a pair of infrared transmitters/receivers, called *outgoing-Links* and *incoming-Links*, to support communication as well as docking guidance.

Each module has a set of open I/O ports so that various sensors for tilt, touch, acceleration, and miniature vision, can be installed dynamically. Each module has two Degrees Of Freedom: DOF1 for pitch (about 0-130° up and down) and DOF2 for yaw (about 0-130° left and right). The range of yaw and pitch of a module is divided to

255 steps. The internal state of each module includes the current values of the yaw, pitch of a module, and the number of the sent and received messages. The modules' *actions* consist of moving the two degrees of freedom to one of the 255 positions, attaching to or detaching from other modules, or sending messages to the communication links through the IR senders.
Modules can be connected together by their docking connectors. Connected docking connectors are called *active* connectors. Docking connectors, located at either end of each module. At one end, labeled *back* (*b* for short), there is a female connector, consisting of two holes for accepting another module's docking pins. At the other end, three male connectors of two pins each are located on three sides of the module, labeled *left* (*l*), *right* (*r*) and *front* (*f*).

1.3. Probing and Communication

The first step for modules in responding to the network topology change consists of detecting local changes. Instances of local changes are: 1) When a new module connects to one of the existing modules in the network, 2) When an existing module disconnects from all other modules in the network, 3) When an existing module establishes a new connection with another module in the network, and 4) When a module disconnects some of its connectors from other modules in the network. In situations 1 and 2 the number of nodes and links and in situations 3 and 4 the number of links in the configuration network changes.
Modules can detect local changes in the topology of the network by periodically monitoring their active docking connectors for disconnections and inactive docking connectors for new connections. This action is called *probing*. In order to detect all the above-mentioned cases of topology change efficiently, we will use two types of probing: 1) Probing when modules are communicating and 2) Probing using *probing signals*.

1.3.1. Probing by Communication

The communication protocols that use handshaking signals when sending and/or receiving messages between modules can be used for probing the active connection links between modules. A successful communication action over an active connector shows that the connection is still active. Similarly, an unsuccessful communication action shows the disconnection of an already active connector.
Figure 2 shows an asynchronous communication protocol that was implemented in CONRO modules. What follows is a brief description of the handshaking sequence of this protocol: Agent_1 is the sender and agent_2 is the receiver.

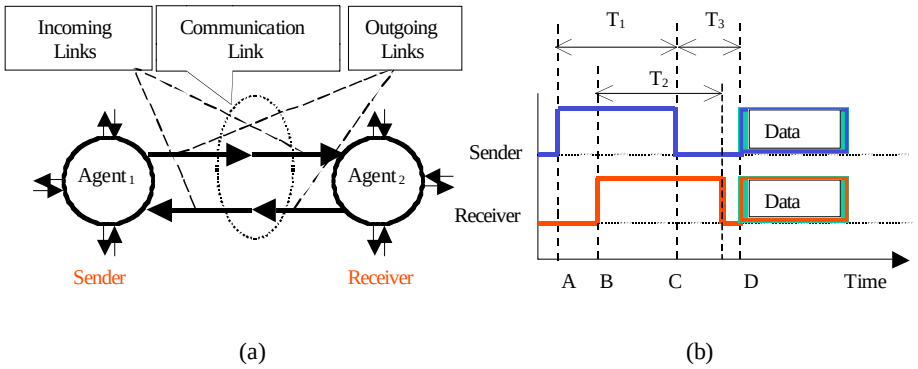

Figure. 2. (a) The communication link between two agents. (b) The asynchronous communication protocol between two agents

1. The sender requests to send a message by making its outgoing link 'High', point A in figure 2b and then continues checking its incoming link for receiving a 'High' signal.

2. The receiver responds by making its outgoing link 'High', point B, and waits.

3. After receiving the 'High', sender makes its outgoing link 'Low', point C, and starts sending the message (Data) after a short delay, point D. This short delay is called *preparation time* (T_3), which allows the receiver to prepare for receiving Data. Data is communicated using RS232 asynchronous communication protocol. In order to avoid dead-lock, timeouts are added to the sender and receiver to limit their waiting time. T_1 and T_2 are sender's and receiver's timeouts, respectively.

This simple handshaking protocol successfully completes if and only if both modules actively participate. Therefore a successful communication confirms for both modules an active link between the two. In oppositely, an unsuccessful communication confirms that the receiving module is not present and the link is inactive.

This method of probing, however, is not efficient for probing the inactive docking connectors unless one attempts to send a message to an inactive docking connector and waits for the timeout, which could be a quite long time. Also, in situations where two modules do not communicate for longer than a pre-specified duration called 'monitoring period', the communication based approach will produce a wrong conclusion. In such situations we will use a different type of probing method based on sending *probing signals*.

1.3.2. Probing Signals

Probing signal are narrow pulses that are periodically sent to inactive connections or active connections if no communication occurs on them for a long time. The width of probing signals is much narrower than the communication protocol signals such that

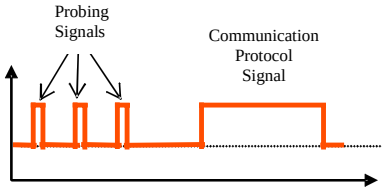

Figure 3: Probing and Communication protocol signals

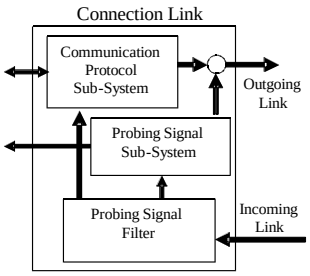

Figure 4: Merging and separating the probing and communication signals

they can be distinguished and filtered from the communication protocol signals. Figure 3 compares the width of the probing and communication protocol signals.
Figure 4 shows the block diagram of a module's connection link. The 'Probing Signal Filter' on the incoming link separates the probing signals from the communication signals. On the outgoing link, the communication and probing signals are merged to a single output line.

1.3.3. Probing Algorithm

Figure 5 describes the probing algorithm for detecting local changes in the topology of the network. This algorithm consists of two procedures. The first procedure, **GenerateProbe**, is called for generating probing signals on inactive connectors or the active connectors that have not communicated for longer than the duration of the 'monitoring period'.
The second procedure, **CheckTopology**, is called for detecting changes in local topology of the network based on the received probing signals or the recent communicated messages. This procedure returns a true value if the topology has been changed.

1.4. Functional Response to Topology Change Using Probing

Our solution for the functional response to topology change problem in self-reconfigurable robots relies on our previous work on distributed control for self-reconfigurable robots. Specifically, the 'distributed task negotiation' and 'distributed behavior collaboration' problems. In this section we will describe these problems and our proposed solutions. Then we will present the FEATURE algorithm that utilizes probing for solving the functional response to the topology change problem.

```
when GenerateProbe () do
  for each C ∈ Connectors do
    if (C = Inactive) or (NoComm (C, Period) = true) do
       send ProbingSignal to C;
    end do;
  end do;
end do;

when CheckTopology () do
  TempLocalTopology = CurrentLocalTopology;
  TopologyChanged = false;
      for each C ∈ Connectors do        //reset
       CurrentLocalTopology (C) = Inactive; end do;
  for each C ∈ Connectors do
     if (CommOccurred (C) = true) or
        (Probe Signal Received (C) = true) do
         CurrentLocalTopology (C) = active;
     end do; end do;
  if (TempLocalTopology ≠ CurrentLocalTopology) do
     TopologyChanged = true;
  end do;
  return TopologyChanged;
end do;
```

Figure 5: The probing Algorithm

1.4.1. Distributed Task Negotiation

Distributed Task Negotiation is a process by which modules in a self-reconfigurable robot can negotiate and select a single coherent task among many different and even conflicting choices. This is a very challenging problem due to several reasons: the relationships among modules are not static, but change with configurations; modules have no unique global identifiers or addresses; modules do not know the global configuration in advance, and can only communicate with their immediate neighbors.
In [Salemi 2003] we presented the DISTINCT algorithm as a solution for the distributed task negotiation problem. The main idea is that all modules work together to build global spanning trees and each tree is associated with a task. Initially, all modules that have their own competing tasks start building their own trees, but as they exchange messages for tree building, most modules will give up their selected tasks and "root" status and participate in building trees for other tasks. In this process, modules report their status to their parent module in the tree to which they belong, and the module that does not have parent but received reports from all its children is the root for the entire network of modules. When this happens, this root module can

conclude that the negotiation process has succeeded and all modules in the tree have agreed on the same task. An embedded synchronization algorithm detects the termination of the negotiation process.

Formally, a distributed task negotiation problem consists of a tuple (*P, L, T, S*), where *P* is a list of nodes, p_i, such that $i \in \{1,\ldots, N\}$; *L* is a list of communication links, l_{jk}, such that $j,k \in \{1,\ldots, N\}$; *T* is a list of tasks, t_m, such that $1 \leq m \leq N$, and *S* is a set of task selection functions, $S_i: (T') \rightarrow t_i$, such that $i \in \{1,\ldots,N\}$ and $T' \subset T$. Each node has a task selection function that can select a single task from a set of given tasks. A distributed task negotiation problem is solved when all nodes have selected the same task from *T*, called t^*, and have been notified that the negotiation process is terminated. Note that the index numbers assigned to *P* are only used for defining the problem and not used in the negotiation process. In addition, the size of the network is unknown to the individual nodes.

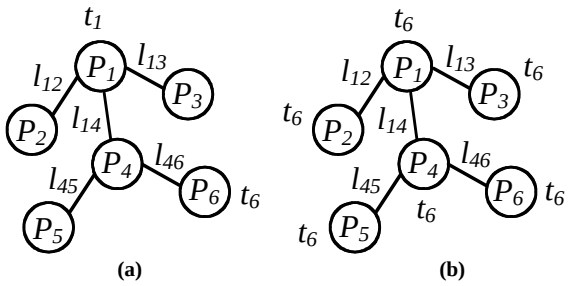

Figure 6: An example of a distributed task negotiation problem. **a**) Initially p_1 and p_6 initiated two tasks (t_1, t_6). **b**) A solution, when all agents have selected $t^* = t_6$.

To illustrate the above definition, consider the example in Figure 6a, where $P = \{p_1, p_2, p_3, p_4, p_5, p_6\}$, $L = \{l_{12}, l_{14}, l_{13}, l_{45}, l_{46}\}$, $T = \{t_1, t_6\}$, and *S* is a selection function that prefers tasks with greater indexes and shared by all nodes. Initially, node p_1 and p_6 have initiated two tasks, t_1 and t_6, respectively, and the rest of the nodes are waiting to receive tasks. Figure 6b depicts a solution for the given problem where all nodes agreed on task $t^* = t_6$.

Important characteristics of this solution are: 1) modules do not require having unique Ids; 2) ensures that all nodes will select the same task coherently; regardless of the number of competing tasks initiated in the network; and more importantly 3) it is not dependent on the topology of the network of modules.

1.4.1.1. Negotiation by Creating Spanning Trees

The most obvious solution for the problem is to assign priorities to the competing tasks and force nodes to select tasks that have higher priorities. However, since the importance of tasks cannot be determined statically, it is extremely hard to determine the correct priorities for an arbitrary set of competing tasks.

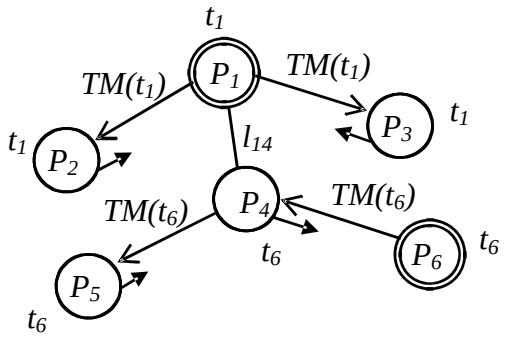

Figure 7: Task message propagation. Arrows on the links indicate messages in transit and arrows parallel to links indicate the "child-of" relationship. Double circles indicate the roots of partial TSTs.

In our solution, nodes propagate their tasks to their neighbors and generate a Task Spanning Tree (TST) for each propagated task. As a result, when more than one task is initiated, a forest of partial TSTs is created. These partial TSTs negotiate with each other and gradually merge into one and only one TST. This final TST represents the task that has been selected by all nodes in the network.
During the tree building process, all nodes report their status to their parent nodes. The negotiation process terminates when a node that has no parent has received reports from all of its children. This node is the root of the final TST, and it then notifies all nodes in the tree with an "end of task negotiation" message and all nodes will select the task associated with the final TST.

1.4.1.2. Distributed Task Selection

For nodes that have competing tasks to select a single task, the goal is to create a single TST. Each node must decide on two issues: 1) what task to select and propagate, and 2) how to be a part of a TST.
Initially, nodes that have competing tasks propagate their tasks by sending a *task message* (*TM*) to their neighbors and designating themselves as the root of a partial TST. Assuming that the recipient of a *TM* has no tasks for itself and receives only one *TM*, then it will adopt the received task and create a "child-of" relationship toward the sender of the *TM*. The recipient will in turn propagate the received task by sending a new *TM* to the rest of its neighbors. To illustrate this idea, Figure 7 shows an example in which nodes P_1 and P_6 are the initiators of tasks t_1 and t_6 respectively and the rest of the nodes are non-initiator nodes. Node P_2 and P_3 are the recipients of $TM(t_1)$ sent by P_1, and therefore have selected task t_1. Similarly, P_4 and P_5 are the recipients of $TM(t_6)$ sent by P_6, and therefore have selected task t_6. In this situation, parallel arrows show the "child-of" relationships that the nodes have created.

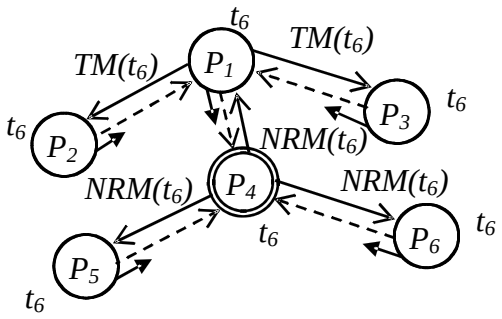

Figure 8: Merging partial TSTs from Figure 2. P_4 is the new root of the merged TST. The dashed arrows indicate the ack messages.

Based on the above assumption, no message has been sent through the link l_{14}. As a result two TSTs have been formed; one rooted at P_1 and the other rooted at P_6. In each TST, all nodes have selected the same task.

At this point, if we relax the above assumption, two cases might occur: **1**) either a root node receives a *TM*, or **2**) a non-root node receives a *TM* from a node that is not its parent. An example of the first case is shown in Figure 8 where P_1, a root node, receives a *TM* from P_4. An example of the second case happens when P_4, a non-root node in the TST rooted at P_6, receives a *TM* from P_1, which belongs to another partial TST.

In the first case, the recipient, which is a root node, drops being a root, adopts the received task, establishes a "child-of" relationship with the sender of the *TM* and propagates new *TM* to the rest of its neighbors, which are its children. In this situation, these nodes adopt the new received task and propagate it to the rest of their neighbors.

In the second case, the received *TM* is a conflicting message since it was received from a non-parent node. To resolve the conflict, the recipient node deletes all of its previous "child-of" relationships, makes a choice between its previous task and the received task (using its task selection function), propagates a *newRoot message* (*NRM)* containing the newly selected task to all of its neighbors, and then promotes itself as a new root for the selected task.

The role of *NRM* is to merge partial TSTs and create a new root for the resulting TST. Therefore, the recipient of a *NRM* adopts the received task, creates a new "child-of" relationship towards the sender of the *NRM*, becomes a non-root node (if previously a root), and propagates a new *NRM* containing the received task to the rest of its children.

Figure 8 shows the result of merging the two partial TSTs in Figure 7 for the situation where P_4 has been the node that has received a conflicting *TM* from P_1. As a result, P_4 chooses a task between t_6 and t_1 (say t_6 is chosen), promotes itself to be the root of the

new TST, and propagates $NRM(t_6)$ to P_1, P_5 and P_6, which turns P_1 and P_6 into non-root nodes. Consequently, P_1 will adopt t_6 as its new task and propagate a new *TM* to P_2 and P_3 for the task switch.
As shown in Figure 8, the final result of the task negotiation process is a single TST with a specified root node and a selected task. However, at this point the nodes do not know that the task negotiation process has been terminated. Unless a mechanism for detecting the termination of negation is in place, the nodes would wait indefinitely.

1.4.1.3. Distributed Termination Detection

In order to detect the termination of the task negotiation process, we use an approach similar to the "termination detection algorithm for diffusing computation" by Dijkstra and Scholten [Dijkstra 1980]. For each received *TM* and *NRM*, each node must reply with an acknowledge message (*AM*), after the node receives acknowledges from all its children. For a leaf node, this means that it will acknowledge immediately for every received message. For a non-leaf node, it will send an acknowledge message to its parent after it receives *AM* from all of its children. If a non-leaf node receives all *AM* from all its children and it has no parent, then this node is the root for the final TST and it can conclude that the task negotiation process has succeeded.
In Figure 8, dashed arrows indicate the *AM* messages. The root node, P_4, expects to receive *AM*s from each of the P_1, P_5, and P_6 nodes. Since P_5 and P_6 do not have any child nodes, they send their *AM* as soon as they receive $NRM(t_6)$ messages from P_4. However, P_1 sends its *AM* to P_4 only after it receives *AM*s from P_2 and P_3. When P_4 receives all of its expected *AM*s, it detects the termination of the negotiation process and propagates a *taskSelected* message to all of its children. This message will be propagated to all the nodes in the tree and the task negotiation process successfully terminates.

1.4.2. Distributed Behavior Collaboration

The problem of distributed behavior collaboration can be defined as follows: Given a global task and a group behavior, to select a correct set of local behaviors at each module and coordinate the selected behaviors to produce the desired global effects.
The problem is very challenging due to several reasons: relationships among modules are not static but change with configurations; the number of modules in the robot is not known; modules have no unique global identifiers or addresses; modules do not know the global configuration in advance, and can only communicate with immediate neighbors. Under these circumstances, a satisfactory solution to distributed behavior collaboration must be distributed. Modules must select behaviors through local communication, and the execution of the selected behaviors must be synchronized.
Formally, the problem of distributed behavior collaboration is a tuple (P, Q, C, A, B, t, GB), where P is a list of nodes, p_i; Q is the list of the internal state, q_i, associated with each node p_i, such that $i \in \{1,\ldots, N\}$; C is a list of labeled physical or logical links, c_j, such that $j \in \{\text{locally unique labels}\}$; A is a set of actions a_s a node can execute, such that $s \in \{1,\ldots, S\}$; B is a set of behaviors in the form of $b_m = (a_x, a_y\, a_z, \ldots)$, such that $m \in \{1,\ldots, M\}$; t is the global task given to all nodes, and GB is the desired group

behavior in the form of behavior selection rules. These rules are mappings from nodes internal states to behaviors, $Q \rightarrow B$. The *configuration graph* of the network of modules is a graph consists of P nodes and C edges. A distributed behavior selection problem is solved if and only if the ordered sequence of the selected behaviors of all nodes over time is equal to the desired group behavior. If β represents the ordered sequence of the selected behaviors or $\beta = \beta + b_{pi}$ where, $i \in \{1,\ldots, N\}$, $\beta = GB$, should hold. Note that the size of the network is dynamic and unknown to the individual nodes; also the index numbers are only used for defining the problem and not used in the solution.

1.4.2.1. Extended Neighborhood Topology

When modules in a self-reconfigurable robot have negotiated and decided on a global task, they must then generate a group behavior to accomplish the task. A group behavior is the result of the coordinated performance of local behaviors of individual modules, while the local behaviors are selected based on the location of the modules relative to other modules. In [Salemi 2001], we represented the module's location in a configuration as the *type* of the module. Table 1 lists 32 local types of CONRO module, which reflects the ways that a module can connect to its immediate neighbors. Figure 9 shows some example types in various CONRO configurations.
The type information in Table 1 could provide modules with the necessary information to uniquely determine their location in most cases and select the appropriate local behaviors for the global task accordingly (see details in [Salemi 2001]). However, these types are not enough to guarantee determining modules' location in any complex configuration. For example, consider the T-shape and the snake configurations in Figure 9. Modules A, and A' are both of type T2, yet they must behave differently in the two different configuration. The module A' must perform a sinusoidal behavior in the snake configuration, while the A module must

Connected to other modules	This Module					This Module				
	b	*f*	*r*	*l*	Type	*b*	*f*	*r*	*l*	Type
					T0	*f*	*b*			T16
	f				T1	*f*		*b*		T17
		b			T2	*f*			*b*	T18
			b		T3		*b*	*b*	*b*	T19
				b	T4	*f*	*b*	*b*		T20
	l				T5	*f*		*b*	*b*	T21
	r				T6	*f*	*b*		*b*	T22
		b	*b*		T7	*l*	*b*	*b*		T23
			b	*b*	T8	*l*		*b*	*b*	T24
		b		*b*	T9	*l*	*b*		*b*	T25
	l	*b*			T10	*r*	*b*	*b*		T26
	l		*b*		T11	*r*		*b*	*b*	T27
	l			*b*	T12	*r*	*b*		*b*	T28
	r	*b*			T13	*f*	*b*	*b*	*b*	T29
	r		*b*		T14	*l*	*b*	*b*	*b*	T30
	r			*b*	T15	*r*	*b*	*b*	*b*	T31

Table 1: 32 local topological types of CONRO module

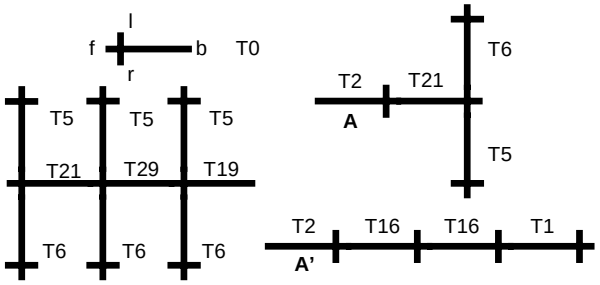

Figure 9: Immediate neighborhood topological types of CONRO modules in different configurations: a single module, a hexapod, a T-shape, and a snake.

keep still in the T-shape "butter-fly" locomotion (i.e., the leg modules move in a cycle of up, left, down, and right, while the body modules keep still).

To solve this problem, we extend module's type from the immediate neighborhood to neighbors that are *n* modules away. We call this extended type, type(*n*), and define it as how the active connection links of a given module are connected to the connection links of the modules of distance *n*. For example, in Figure 10, type(0) for module A is [(*bf*)] because module B is the only one of distance zero from A, and the *b* connector of B is connected to the *f* connector of A. However, type(1) of module A is [(*br*,*bf*),(*bl*,*bf*)] because this is the way A is connected to module C and D, which are one module away (*n* = 1). Similarly, the type(0) of module A' is [(*bf*)] and its type(1) is [(*bf*,*bf*)]. Note that module B has only immediate neighbors (distance = 0) and therefore it only has type(0) information.

As we can see, although type(0) values of module A in the T-configuration and

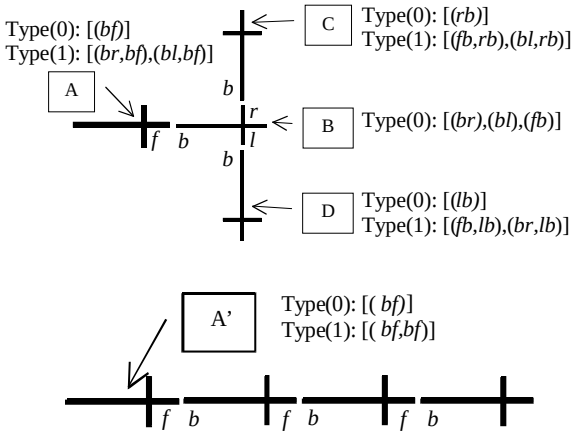

Figure 10: Examples of the extendable type(*n*).

module A' in the snake configuration are the same, but they have different type(1) value. It can be proven that using the extended types, modules can always uniquely identify themselves in a configuration as long as the labels of connection links of modules are locally unique. The proof is based on having unique path between any two nodes in a tree. Later we will show that modules can use the extended type information to select the appropriate behavior based on the given task.

It can be shown that the *type* definitions in [Stoy 2002, Salemi 2001] are special cases of the extended type and equivalent to type(0). In addition, global representation of the entire network for each agent is equivalent to [type(0),type(1), . . ., type(*d-1*)], where *d* is the diameter of the network.

It is possible for the modules to autonomously dynamically discover their extended types and. The solution is based on the characteristics of hormone-inspired messages described in [Shen 2002a]. Each hormone message contains a path field that records a list of connector-pairs (ex. *bf*) through which the message has been propagated. When a module receives a hormone message with |*path*|=*m*, it will insert the path into its extended type(*m-1*) values. As more and more messages are received, the extended type information will be built up. Since messages are propagated through the network, each module will eventually build up the correct type values for itself. See [Salemi 2004] for more details.

1.4.2.2. Selecting Local Behaviors Via Type(n) Values

A straightforward approach for behavior collaboration in a modular system is the centralized 'gait control table' [Yim 1994], in which a designated module, called the central controller, is given the information about behaviors of other modules in the form of a table. Each column of this table contains the sequence of actions that a module, identified by its identifier, has to perform over time based on its location in the configuration (equivalent to the behavior of the module). The central controller job is to send each row of the table specifying the actions that all modules should perform at a time.

The 'gait control table' approach, however, is not an ideal approach for controlling the self-reconfigurable system for the following reasons: first, requiring the central controller to send actions to the rest of the modules in the configuration creates a communication bottleneck. In addition, if the central controller becomes faulty the entire system will be disabled. More importantly, when the network of modules restructures themselves, the pre-specified behaviors of the modules in the table might not be valid anymore. The source of this difficulty is that modules do not know how their behaviors are chosen so they cannot select new behaviors as they re-locate in the configuration.

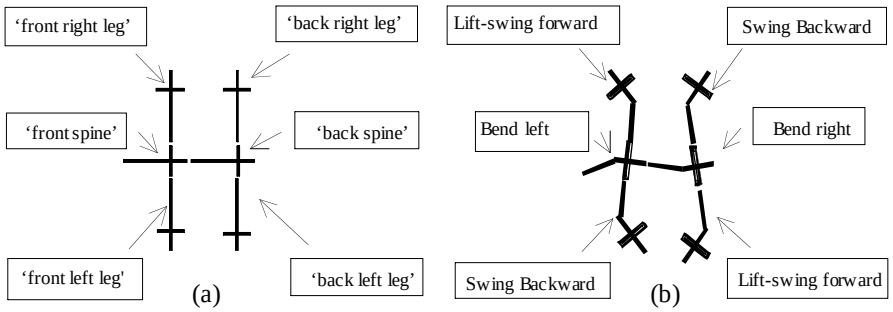

Figure 11: (a) The module types in a four-legged self-reconfigurable robot; (b) The selected local behaviors of each module for "move forward".

Our approach for solving this problem is based on using the extended types to uniquely identify the location of modules in a configuration, and use them to select the correct local behaviors by the modules for the given global task. For example, as shown in Figure 11, for a quadruped to accomplish the 'Move forward' task, the 'front left leg' module and 'back right leg' will select 'Swing Backward' behavior, while the 'front right leg' module and the 'back left leg' module will select the 'Lift and Swing Forward' behavior and modules of types 'front spine' and 'back spine' will select 'bend left' and 'bend right', respectively. Figure 11b shows the robot after the modules have performed their selected behaviors. In general, the group behavior to accomplish "Move forward" consists of the following behaviors: the 'front left leg' and the 'back right leg' perform the 'Lift and Swing Backward' behavior, while the 'front right leg' and the 'back left leg' is performing the 'Swing Forward' behavior, and then the two groups switch their behaviors. In the next section we present a flexible algorithm called Distributed BEhavior SelecTion (D-BEST) for behavior selection, which is based on the extended neighboring types.

1.4.2.3. The D-Best Algorithm

Using the extended types as the condition for selecting behaviors will provide the modules with the information they require to autonomously collaborate to select new behaviors. Figure 12 illustrates the basic idea of the behavior collaboration based on the extended types. Initially, modules communicate their currently selected behaviors to their neighbors by sending hormone messages and wait for receiving new hormone message. This initial behavior could be a **Null** behavior. The communicated hormone message content consists of the type of the message, in this case of type **<Behavior-selection>**, and an initially empty path field, represented by **<(path)>**.

When a new hormone message is received, the module updates the path field of the message, updates its extended type based on the received path and propagates the message to its neighbors. Then it uses the current extended type to select a behavior

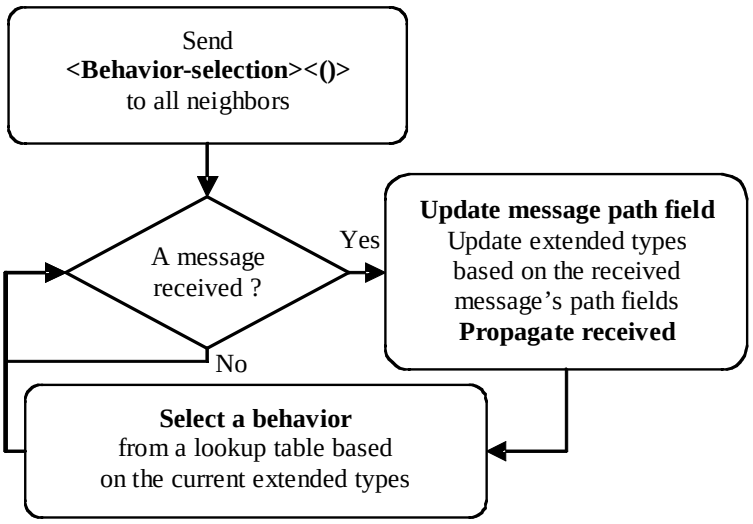

Figure 12: Basic idea of behavior selection based on the extended types

from a lookup table representing the desired group behavior. This process will continue until all modules receive and propagate the initiated hormone messages.

Although this approach can dynamically adapt to the changes in the topology of the network, it has two problems. First, an initiated message from a module will be propagated to all other modules in the configuration. This means that the total number of communicated messages will be $O(N^2)$, where N is the number of modules. The second problem is that when the diameter of the configuration is large, the size of the path field, and therefore the size of the message, will be large. This will considerably slow down the communication when the bandwidth is narrow.

To solve these two problems, we limit the maximum length of the path field in the messages. For example, if the maximum length of path is set to k, a message will stop being propagated after k hops. In this situation, if there are N agents in the network, and each agent has the average number of a active connectors, the number of communicated messages will be $O(N)$ (since at most $a*N$ messages will be initiated and each message will be communicated k times therefore $k*a*N$ messages). The tradeoff of this solution is that the created extended type will be partial, and therefore might not be enough for some modules to select the correct local behaviors.

This problem can be solved by including the modules' selected behaviors in the communicated hormone messages and representing the group behavior as a set of *decision rules* based on the partial paths and received behaviors. In this situation, the content of the hormone messages and decision rules are shown in Figures 13a and 13b, respectively.

The basic idea of this solution is that receiving the selected behavior of an extended neighbor gives an overview about the configuration of the module around that module,

<**Behavior-selection**><(**path**)><Selected behavior> (**a**)

if (received *path* == X) **and** (received *behavior* == Y) (**b**)
then (select local behavior Z)

Figure 13: a) the format of the hormone messages. b) the format of the decision rules

which combined with the received partial path can be used for selecting the correct behavior.

This control algorithm has some unique features that are different from previous approaches. Unlike the approaches based on the pre-assigned behaviors, this controller can adapt with the dynamic self-reconfiguration of the network, prevent communication bottleneck and is robust to individual modules failure. In addition, D-BEST is more flexible and powerful than the approaches for behavior selection based on immediate neighboring connection patterns as D-BEST uses both immediate and extended neighboring modules connection pattern for behavior selection.

It can be seen that the shorter the maximum size of the path results the smaller number of communicated message. This feature can be utilized at the design time of the decision rules for a desired group behavior in the following way. Starting from the smallest maximum length ($k = 0$) the designer of the group behavior will write the rules that can uniquely select the correct behaviors for the modules. If there is ambiguity in selecting behaviors for the possible configurations, the k will be increased to provide the modules with more information such that the ambiguity is resolved. This characteristic of the D-BEST algorithm allows the number of the communicated message to be a function of the complexity of the desired group behavior and/or possible configurations.

Figure 14 shows an example of the behavior collaboration for generating a Butterfly locomotion in a T-shape CONRO robot. In this example Butterfly_Spine, Move_East, Move_West and CAT0 are different behaviors and four decision rules are used. The maximum path length is chosen to be zero, $k = 0$, specifying that the messages that immediate neighboring modules communicate will not be propagated to other modules.

According to the rule 1, if a module receives a message from one of its left or right connectors, it will be a spine module otherwise it can be a spine or a module in a snake configuration. Based on this rule, module B can select the correct behavior, Butterfly_Spine. However, module A does not know if it is a spine module or in the snake configuration. Rule 4 can resolves this issue by determining the module cannot be part of a snake configuration if the neighboring module B has selected Butterfly_Spine. Rules 2 and 3 will be used by the side legs to select the correct direction for their movements. If module A applies rule 1, it will consider the possibility of being part of a snake by selecting the CAT0 (sinusoidal motion starting

1) **If** path = ((*bl*) **or** path = (*br*))
then select Butterfly_Spine
else select CAT_0

2) **If** path = (*rb*)
and behavior = Butterfly_Spine
then select Move_West

3) **If** path = (*lb*)
and behavior = Butterfly_Spine
then select Move_East

4) **If** (path = (*fb*) **or** path = (*bf*))
and behavior = Butterfly_Spine
then select Butterfly_Spine

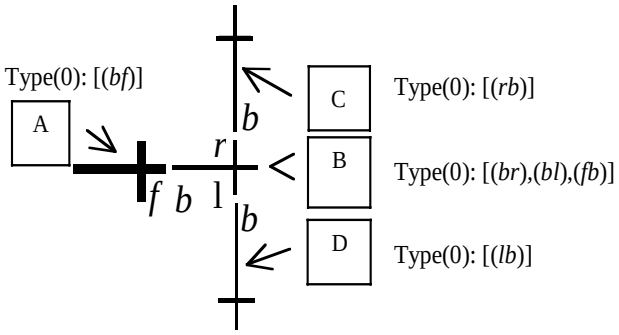

Figure 14: Decision rule for Butterfly locomotion.

from angle zero for the caterpillar move). This selection will be corrected if at some point rule 4 is applicable.

1.4.3. The FEATURE Algorithm

In this section, we will describe the FEATURE algorithm that brings all the above-mentioned pieces together and solves the problem of functional response to topology change in self-reconfigurable robots. This will be the algorithm that will run on all modules of the self-reconfigurable robot to ensure the homogeneity of all modules. Figure 15 depicts this algorithm.
Initially all modules will wait to receive a new task. The new task can be initiated by an outside controller or one of the sensors on a module. Receiving a new task initiates a distributed negotiation process among all modules in the robot. This is necessary to ensure that 1) all modules know what task they accomplishing and 2) in cases where multiple modules have initiated more than one task, all modules will agree on accomplishing the same task that has the highest priority. The above-mentioned process is controlled by the DISTINCT, task negotiation algorithm shown on top part of the figure 15.

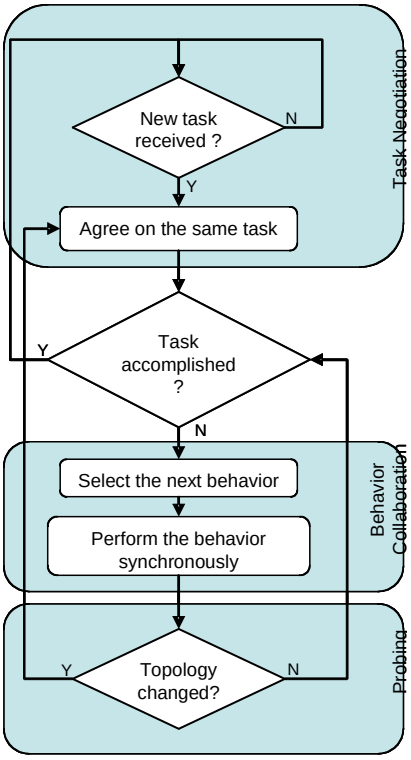

Figure 15: The FEATURE algorithm

If the selected task is not already accomplished, modules will generate a set of relevant behaviors. The relevant behaviors are represented by a set of decision rules that have been downloaded in all modules. The execution of the selected behaviors will be coordinated by an embedded distributed synchronization mechanism. The above-mentioned process is controlled by the D-BEST, behavior collaboration algorithm shown in the middle section of the figure 15.
If the topology of the network of module changes while modules are performing their behaviors, the modules that detected the local changes, will initiate the DISTINCT algorithm using the current selected task in order to dynamically create a new spanning tree for synchronizing and initiating new sets of behaviors based on the current topology of the network. The topology detection process will be controlled by the probing algorithm shown on bottom of the figure 15.

1.5. Examples

We have implemented and tested the FEATURE algorithm and all of it sub–algorithms on the CONRO self-reconfigurable robots. All modules are running as autonomous systems without any off-line computational resources and are loaded

with the same control program and decision rules. For economic reasons, the power of the modules is supplied independently through cables from an off-board power supply.

In our experiment we gave a 'Move' task to a quadruped CONRO robot. The robot initiated a 'Four-Legged Walking' gait. While performing the gait, we detached the two spine modules. The resulting configuration was two separate T-shape robots. In this situation, each T-shape robot continued the locomotion by executing the 'Butterfly Stroke' gait. Later, two T-shape robots were re-connected and the resulting four-legged robot re-initiated the 'Four-Legged Walking' gait.

For the snake configuration, we have experimented with caterpillar movement with different lengths ranging from 1 module to 10 modules. With no modification of programs, all these configurations can move and snakes with more than 3 modules can move properly as caterpillar. The average speed of the caterpillar movements is approximately 30cm/minute. In this experiment, we have dynamically "cut" a 10-module running snake into three segments with lengths of 4, 4, and 2. All these segments adapt to the new configuration and continue to move as independent caterpillars. We also dynamically connected two or three independent running caterpillars with various lengths into a single and longer caterpillar. The new caterpillar adapted to the new configuration and continued to move in the caterpillar gait. These experiments show that the described approach is robust to changes in the length of the snake configuration.

To test this approach for self-reconfiguration from a Snake to T-shape, a self-reconfiguration task was manually given to one the middle modules of a snake-shape robot consisting of seven modules. After completion of the self-reconfiguration task the new topology of the robot was detected and a butterfly gait for the T-shape robot was generated. The videos of these experiments are available at http://www.isi.edu/robots.

1.6. Conclusion and Future work

This chapter presented the FEATURE algorithm that combines a set of distributed algorithms for accomplishing global tasks in a chain-type self-reconfigurable robots consisting of multiple modules with dynamic topology. These combined algorithms were DISTINCT for distributed task negotiation, and D-BEST for distributed behavior collaboration. The FEATURE algorithm used probing for detecting the local topology changes and this information was used for global coordinated selection of the new behaviors in the modules. The FEATURE algorithm was implemented on the real CONRO self-reconfigurable robot modules and the experimental results were presented.

As the future work, we will study the conditions for performing successful self-reconfiguration and locomotion tasks based on the received messages and develop a complete set of decision rules for performing all possible self-reconfiguration and locomotion tasks.

References

Dijkstra, E.W., C.S. Scholten, *Termination Detection for Diffusing Computations.* Information Processing Letters, 1980. **11**.

Murata, S., H. Kurokawa, E. Toshida, K. Tomita, and S. Kokaji, , *A 3-D self-reconfigurable structure* in ICRA,1998.

Rus, D., Z. Butler, K. Kotay, M. Vona, *Self-Reconfiguring Robots.* ACM Communication, 2002

Salemi, Behnam, WM. Shen and P. Will, *Hormone Controlled Metamorphic Robots*. in *ICRA* 2001.

Salemi Behnam, Peter Will, and Wei-Min Shen. "*Distributed Task Negotiation in Modular Robots*". Robotics Society of Japan, Special Issue on "Modular Robots", 2003.

Salemi Behnam, Wei-Min Shen *Distributed Behavior Collaboration for Self-Reconfigurable Robots.* International Conference on Robotics and Automation, New Orleans, LA, USA, 2004.

Shen, W.-M., B. Salemi, and P. Will., *Hormone-Inspired Adaptive Communication and Distributed Control for CONRO Self-Reconfigurable Robots*. IEEE Transaction on Robotics and Automation,. 18(5): p. 700-712, 2002a.

Shen, W.-M. and M. Yim (editors), *Special Issue on Self-Reconfigurable Modular Robots*, IEEE Transactions on Mechatronics, 7(4), 2002b.

Stoy, K., Shen, WM., Will, P., *Using Role-Based Control to Produce Locomotion in Chain-Type Self-Reconfigurable Robots*. IEEE/ASME Transactions on Mechatronics,. **7**(4): p. 410.M. Young, The Technical Writer's Handbook. Mill Valley, CA: University Science, 2002.

Yim, M., Y. Zhang, D. Duff, *Modular Robots*. IEEE Spectrum, 2002.

Yim, M., *Locomotion with a unit-modular reconfigurable robot* (Ph.D. Thesis), in Department of Mechanical Engineering. 1994, Stanford University.