A Dynamic ID Assignment Approach for Modular Robots

Joseph Assaker, Abdallah Makhoul, Julien Bourgeois, Benoît Piranda and Jacques Demerjian

Abstract In this paper, we present a distributed Id assignment algorithm for modular robots. Our proposed solution supports both the removal and the addition of particles in the system, while maintaining particular characteristics in the logical tree, allowing for fast and efficient inter-module communications. The key goal here is to maintain easily calculated routes between any two particles in the system, with the minimal overhead possible. The idea of "holes" or free IDs is introduced in the system by three main alterations to our previous unique id assignment algorithm [2]. The first being the modification of the unique ID assignment phase. The second being the handling of particles removal from the system. And the third being the handling and initiation of a newly added particle anywhere in the system.

1 INTRODUCTION

Modular robots are autonomous systems composed of a large number of modules/particles that change their morphology/shape according to the surrounding environment and the application requirements [12, 1]. Self-reconfiguration by rearranging connections between modules is the main task of modular robots. This task and others [11, 9] rely on message transmission between particles directly connected or connected through other particles. Therefore, it is important to have some kind of identification for each module that remains constant throughout all of the modi-

Abdallah Makhoul, Julien Bourgeois and Benoît Piranda
FEMTO-ST institute, Univ. Bourgogne Franche-Comté, CNRS, 1 cours Leprince-Ringuet, 25200, Montbéliard, France, e-mail: \{first\}.\{last\}@femto-st.fr

Joseph Assaker, Jacques Demerjian
LaRRIS, Faculty of Sciences, Lebanese University, Fanar, Lebanon, e-mail: jacques.demerjian@ul.edu.lb.
Acknowledgement: This work has been supported by the EIPHI Graduate School (contract ANR-17-EURE-0002).

fications the system may go through. To accomplish this goal one might suggest at the manufacturing level and whenever a module is manufactured to directly assign a global unique ID to it (similar to a MAC on most electronics). Another simple technique that comes to mind is a random ID assignment [10]. However, such approaches are highly inefficient and limiting, as it would impose very long IDs which is not suitable for small and energy constraint modules. If having a system composed of 100 modules for example, why would one use long globally unique IDs between all existing modules whereas IDs composed of 7 bits (total of 128 unique IDs) are enough. A similar related problem is ID Assignment in Sensor Networks. In this context, Several schemes have been proposed to assign locally unique identifiers for sensor nodes [13, 5, 7, 4]. However, theses approaches can not be applied to modular robots. Indeed, there are significant differences between the two systems, notably that the topology of modular robots will evolve continuously as the robot changes its morphology. The communications in modular robots are generally done only with the adjacent neighbour modules [3], and with the absence of a sink base station).

In our previous work [2], we presented a distributed algorithm that assigns unique IDs to modules. It is composed of three phases. The first one consists in discovering the whole system while building a logical tree. The second one finds the total size of particles in the system needed, and the third one is dedicated to the unique ID assignment. After the execution of this algorithm, the final system tree distributing the unique IDs could be compared to a B-Tree; where in order to reach the module with ID x, we need to take the route passing by the module with ID less or equal to x and its sibling with ID greater than x. This would render message passing between modules anywhere on the system very efficient. In this work, we assumed that all modules remain static and do not change their positions until the end of the algorithm. However, modular robots are by nature 'modular', meaning that the modules' positions will vary throughout the operation the system. This would result in the shuffling of IDs in the system, thus losing the B-Tree-like property.

In this paper, we propose an extension to our previous algorithm that relaxes this assumption to accommodate a dynamic topology where modules join and leave at any time during the execution of the algorithm or after its termination. This new version of the algorithm allows the system to handle various modules position changes with the goal of maintaining the order of the logical tree, by introducing holes or free IDs in the system. The goal here is to have available free spots for newly added modules to fit in, without breaking the order of the logical tree. We also aim at addressing this obstacle, that is of post-communication and routing between any two modules in the system after the termination of the ID assignment algorithm. Maintenance of the B-Tree-like property is one of the proposed solutions.

The remainder of this paper is organized as follows. In Section 2, we recall the main steps of our previous algorithm and present the main idea of the new version proposed in this paper. In Section 3, we develop the distributed dynamic ID assignment algorithm. Section 4 presents some discussions. In Section 5 we present several simulation results. Finally, Section 6 concludes with a brief description of future work.

2 Background on unique ID Assignement

In our previous work [2], we proposed a three phases distributed ID assignment algorithm. In the first phase, the goal is to discover the whole system of modules while along the way building a tree structure rooted at the leader module. The second phase is dedicated to collect the system size (i.e., number of modules in the system) with the final goal of reporting the total size from the whole system to the leader module. Having the system size in hand, the leader module can calculate the least amount of bits needed in order to code global unique IDs for every module in the system. In the third phase, and after building the tree structure that logically connects modules and calculating the least amount of bits needed, the final step of unique ID assignment is launched from the leader to the whole system. In this algorithm we used 5 types of message: 1 (Explore neighbours for potential children); 2 (Confirm that the explored node is a child); 3 (Decline that the explored node is a child); 4 (Report the node's sub-tree size to its parent); 5 (Distribute the global unique IDs to children) [2].

One of the most useful benefits derived from this ID distribution method, is that the unique IDs are allocated in such a way throughout the logical tree that they resemble a B-tree structure. In the sense that, given any node in the system with ID i, all of its children will have IDs x, with $i < x < j$, where j is the ID of the node's direct sibling (or its parent's sibling if it is the last child node) [2]. This is particularly useful in the context of inter-module communication. First, given the constructed logical tree, each module would reach the root node relatively quickly and efficiently. In the other hand, if the root node wanted to reach any other node in the system, and without having this B-tree-like characteristic in the IDs distribution, a broadcast would have been required in order to ensure that the message reach its intended destination. However, having this ordered characteristic allows the root module to find any other module in the system following the same, relatively short, path from the module in question to the root, by abiding straightforward binary decisions while searching for the module in question. This can be further extended to any communication between any two modules in the system. Short communication paths between any two modules can be easily determined. Albeit this being a powerful feature to add to the system, its practicality, as is, falls short due to the variable morphology of the modular robots systems. In other words, if a single module changes its position while maintaining the same unique ID, it will break the ordered structure rules and render the implementation of efficient inter-module communication unfeasible, unless we impose very strict rules on the system by preventing it from changing any module position without a re-execution of the algorithm. Thus, the necessity of maintaining this characteristic throughout various morphology changes arises. Here, we propose the extended version of the unique ID assignment algorithm, which supports both the removal and addition of modules in the system after the initial execution of the algorithm, while maintaining the characteristics of an ordered tree structure. In order to be able to achieve such a goal, the IDs distribution phase will have to be revamped by introducing a new concept during this phase: the intended addition of unused IDs, or holes, in the IDs logical tree of the system.

These holes will consist of ”free” IDs, added in various nodes in the logical tree of the system, which will serve as available IDs for whenever a new module is connected to the system, via an existing module. The idea here is to have available IDs ready to be assigned to new modules, without breaking the order of the logical tree structure, and with the minimal amount of alterations possible made in the system. For instance, by adding just 1 bit to the ID length, we can double the space of IDs in the system, and allocate all unused IDs as free IDs throughout the system. The following will describe the three main, altered or newly added, functionalities of the ID Assignment algorithm. Starting with the alteration of the IDs distribution phase by introducing the idea of free IDs, followed by the newly added functionalities of module addition and module removal handling, while maintaining the logical tree characteristics with the minimum cost possible.

The main new idea added to the algorithm, which will allow us to handle various modules position changes with the goal of maintaining the order of the logical tree, is the introduction of holes or free IDs in the system. The goal here is to have available free spots for newly added modules to fit in, without breaking the order of the logical tree. This concept will be incorporated in the addition of free IDs at various levels of the tree. For example, considering a module with ID 10 and a reserved free ID, 11, having children with IDs going from 12 till 20, with its direct sibling having the ID 21. If a newly added module connects to this module, it will be able to take ID 11, without breaking the order of the logical tree. i.e., all of its children would still have IDs x with 10 ¡ x ¡ 21. The initial distribution, and later various re-distributions, of free IDs will be handled by modifying or upgrading our IDs distribution phase from the original algorithm. Furthermore, the management of those free IDs will be handled by both the module removal and module addition sections of the algorithm.

3 Dynamic ID Assignment Algorithm

It is worth mentioning that whenever a module receives a type 5 message (cf Section 2), now containing both its unique ID and the number of free IDs available for its sub-tree, it is crucial for the algorithm to allocate the unique ID as the node’s ID, and the directly following IDs as free IDs. For example, let’s consider a type 5 message with the unique ID 10, and F free IDs. If the module determines that it should store 1 free ID at its level, this ID have to be ID 11. That is because, if the module gives ID 11 to its first child and stores for example ID 15 at its level, it will create an ambiguity in the system while searching for the module with ID 15. In other words, while searching for ID 15 and arriving at the module with ID 10, the algorithm would assume that all IDs in the sub-tree of its first child should be greater than 11. So the algorithm would not be able to determine whether to search for a potential ID 15 in the sub-tree of the first child, or the sub-tree of the second child, or if the module with ID 15 doesn’t exist. Whereas if the module reserved ID 11 as a free ID, while providing its first child with ID 12, the algorithm can directly

determine that the module with ID 11 is missing, as it is neither the module with ID 10, nor its first child with ID 12.

Keeping the first two phases of the original algorithm intact, and after the leader module calculates the least amount of bits needed to cover all IDs in the system, it adds 1 bit to this length, creating at least N free IDs for a system composed of N modules. The leader module would then proceed by assigning the ID 0 to itself, calculating the number of free IDs it should store at its level, and distribute unique IDs and free IDs to its children via type 5 messages. A given module determines the number of free IDs, M, to store following this formula: $M = \lfloor (message.n\_free\_ids / subtree\_size) \rfloor$, which assures the balanced distribution of free IDs throughout the system, with more weight being given to lower level modules in the logical tree. This basically states that, if all the modules in a module's sub-tree can store at least Y free IDs, the module itself will also store Y free IDs.

Next, each child's share of the available free IDs is calculated with a basic percentage model, where each child will receive free IDs relative to its sub-tree size. The distribution of IDs to a module's children should respect the following rues: (i) send the module's ID + the number of the module's free IDs + 1 to the first child, (ii) send the module's ID + its number of free IDs + 1 + $\sum_{n=1}^{i-1}(S_n + F_n)$ to the i^{th} child, where S_n is the sub-tree size of the nth child, and F_n the number of free IDs allocated to the n^{th} child. The algorithm's part concerning this modification of the unique ID assignment phase is presented in Algorithm 1. In this algorithm, the variable n_free_ids contains the total number of free IDs in the module's sub-tree. The variable all_ids contains the total size of the ID space allocated to its sub-tree size (i.e., the sum of both the number of allocated unique IDs and the number of available free IDs). Finally, the variable $free\_ids$ is a set which contains an ordered list of available free IDs specifically at this module. The reason for this variable to be a set, and not an integer for example, is because the set of available free IDs in a given module at a given time t could be a non-contiguous set of IDs.

An illustrative example of the execution of this modified phase is presented in Figure 1. In the example, we consider that the execution of phase 1 and 2 of the algorithm have finished (c.f. Section 2). Let's assume that the number of free IDs is equal to the number of unique IDs in the system (i.e., each module would store exactly 1 free ID). Supposing that the module to the left of the leader has a sub-tree size of 2, the leader would send the ID $0+1+1=2$ (i.e., its own ID + the number of free IDs it will store + 1) to it, while sending the ID $0+1+1+(2+2)=6$ to the module below it (following the formula defined above). This reserves all IDs from 2 till 5 to the sub-tree of the module to the left of the leader. The same principle applies supposing that the module below the leader has a sub-tree size of 7 and the module to the right of the leader has a sub-tree size of 87, the ID $0+1+1+[(2+2)+(7+7)+(87+87)]=194$ will be sent to the fourth and last child of the leader (the one above it). Numbers in the upper right corner of each module represent free IDs in this module.

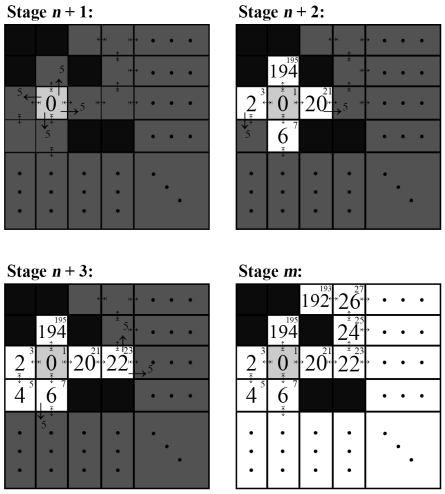


Fig. 1: Illustrative example of the first three stages of the unique IDs + free IDs assignment phase execution and the final stage, m, of the system.

3.1 Module Removal Handling

In this section we present how our approach deals with module removal. We distinguish two cases, the first one when the module leaves completely the system and the second one when it moves from a position to another one. Whenever a module decides to leave the system, whether to change its location in the system or to turn off completely, it must firstly notify its parent of that decision. This notification will be carried out via a type 6 message (cf. Table 1). Type 6 messages will contain extra information concerning the leaving module's free IDs. Given the scheme through which the IDs and free IDs are distributed (a module's set of free IDs contain IDs directly after the module's ID), a simple integer could be sent in order to denote the set of free IDs available in the leaving module. When a module receives a type 6 message, from one of its children, it will remove this child from its children list, update its sub-tree size and number of free ids and add the newly collected set of IDs, including both the child's ID and free IDs set, to its own set of free IDs. Doing so, the algorithm would have successfully "recycled" the leaving module's IDs, ready to be given away to any new module that comes back to this location in the system. One extra step required in this process is to send out type 7 messages (cf. Table 1), from parent to parent up to the root, in order to notify these modules about the modification that occurred in their sub-trees, updating their respective sub-tree sizes and number of free IDs. Algorithm 2 presents the steps concerning the handling of module removal and free ids recycling.

Algorithm 1: Dynamic ID Assignment Algorithm

```
switch message.type do
  case 5
    n_free_ids ← message.n_free_ids ; all_ids ← subtree_size + n_free_ids
    M ← floor(message.n_free_ids/subtree_size) ; free_ids ← ∅
    for K ← 1 to M do
      free_ids ← free_ids ∪ {message.id + k}
    unique_id ← message.id ; next_id ← message.id + M + 1
    for each child in children do
      percentage ← child.subtree_size/subtree_size ; child_n_free_ids ← in(free_ids *
percentage) ; send type 5 message to child
      next_id+ = child.subtree_size + child_free_ids
procedure CHECK( )
  if children.size = 0 OR received all children subtree_size then
    if leader = false then . . .
    else
      calculate least necessary bits ; calculate n_free_ids
      all_ids ← subtree_size + n_free_ids ; unique_id ← 0
      M ← floor(n_free_ids/subtree_size) ; free_ids ← ∅
      for K ← 1 to M do
        free_ids ← free_ids ∪ {k}
      next_id ← message.id + M + 1
      for each child in children do
        percentage ← child.subtree_size/subtree_size
        child_n_free_ids ← int(n_free_ids * percentage)
        send type 5 message to child
        next_id+ = child.subtree_size + child_n_free_ids
end procedure
```

Table 1: Messages' role description

Type	Role Description
6	Notifying parent that module is leaving
7	Notifying ancestors, up to the root, that a module left
8	Notifying neighbor that it is now parent to the module
9	Notifying ancestors, up to the root, that a module joined
10	Delegating ID re-assignment to parent module

3.2 Module Addition Handling

Many schemes could be adopted for adding one module to the system. Whenever a newly added module connects to an already existing module in the system, it is the role of the latter to provide the former with a unique ID, and either with or without a set of free IDs depending on the situation. When a new module is added to the system, it would send a type 8 message (cf. Table 1) to any of its neighbors, which it now considers as its parent (note that this is a situation where the child chooses its parent, and not the inverse). A module receiving a type 8 message would firstly

Algorithm 2: Module removal handling

```
n_free_ids ← 0
all_ids ← 0
free_ids ← ∅
if received message then
  switch message.type do
    ...
    case 6
      children ← children − {< message.origin, 1 >} ; subtree_size− = 1 ; n_free_ids+ = 1
      for K ← 0 to M do
        free_ids ← free_ids ∪ {message.id + k}
      free_ids ← SORT(free_ids)
      if leader = false then
        send type 7 message to parent
    case 7
      child ← find message.origin in children
      child.subtree_size− = 1
      subtree_size− = 1
      n_free_ids+ = 1
      if leader = false then
        send type 7 message to parent
procedure NODE_LEAVE( )
  send type 6 message to parent
end procedure
```

add the origin of this message to its set of children, with a sub-tree size of 1. Next, it would study its options as to how to instantiate this newly added module by providing it with an appropriate unique ID. Multiple scenarios could occur at this stage. If the module has a non-empty set of free IDs, it will directly assign to the newly added module a unique ID, with the possibility of also providing it with free IDs. If the set of free IDs available in the module is a contiguous set, the module would split this set in half, and provide the second half to the new module. This action will be carried out with a type 5 message, where the first element of this set would be treated as the unique ID of the new module, and all other IDs as free IDs in the new module. For instance, if the module's set of free IDs is equal to $\{10,11,12,13,14\}$, ID 12 will be assigned as the new module's unique ID and the set $\{13,14\}$ as its free IDs set. Whereas if the set of free IDs available in the module is a non-contiguous set, the module would provide the right-most contiguous subset of its free IDs set to the new module, with the same method as in the prior scenario (i.e., first element as the unique ID, and the rest as free IDs). For example, considering the free IDs set of $\{10,11,12,15,16\}$, ID 15 will be assigned as the new module's unique ID and the set [16] as its free IDs set. On the other hand, and if the module does not have any available free ID, a re-distribution of IDs will be carried out. Two cases present themselves: if $\backslash n\_free\_ids$" is 0 or greater than 0. If the number of free ids in the module's sub-tree is greater than 0, this means that a free ID is available somewhere in its sub-tree. A re-distribution of unique IDs and free IDs will be performed from this module, down to all modules in its sub-tree. This will be carried out by having

the module send out a type 5 message to itself. This means, it will reconsider its options of free IDs and how to re-distribute IDs to its children, given the new modification (i.e., addition of a new module) in its sub-tree. Finally, and if the number of free ids in the module's sub-tree is 0, the module would have to seek out free IDs from a higher level module in the logical tree of the system. Basically, the module's role now would be to search for an appropriate temporary root module from which a re-distribution phase will be instantiated from. Consequently, the module would send out a type 10 message (cf. Table 1) to its parent, which basically transfers the task of re-distribution to it. Whenever a module receives a type 10 message, and after updating its sub-tree size and its child's sub-tree size, it would check its options. If the module's "$\backslash n\_free\_ids$" is 0, it would send out to its turn a type 10 message to its parent. If "$\backslash n\_free\_ids$" is greater than 0 (i.e., there are holes in its sub-tree that can be utilized), a re-distribution will be performed from this module, down to all modules in its sub-tree. This also will be carried out by having the module send out a type 5 message to itself, which will lead to all modules in its sub-tree to accordingly receive type 5 messages and update their IDs and sets of free IDs. One extra step required in this process is to send out type 9 messages (cf. Table 1), from parent to parent up to the root, in order to notify these modules about the modification that occurred in their sub-trees, updating their respective sub-tree sizes and number of free IDs. The steps concerning the handling of modules addition to the system is presented in Algorithm 3. It should be noted that in the algorithm, four functions are assumed to be available. Function "*length*", which takes as input a given set and returns its length. Function "*sort*", which takes as input a given set and returns the sorted version of this set. Function "*is_contiguous*", which takes as input a given sorted set returns a Boolean whether this set is composed of a contiguous block of items or not. And function "*get_last_subset*", which takes as input a given sorted set and returns the right-most contiguous subset from this set.

4 Discussion

To better understand the benefits provided as a result of our approach, Figure 2 is presented, in order to showcase a small scale example of the execution of the various sections of this algorithm. We assume having an initial system size of 13 modules, with a total ID space of 26 IDs resulting in an initial 13 unique IDs and 13 free IDs. At stage k, the initial execution of the algorithm has already been terminated with success, assigning to each module in the system an appropriate unique ID and free IDs. In addition, a new module is connected to the system in the bottom right corner of the system, right next to the module with ID 8. This new module, and in order to instantiate itself in the system, sends out a message of type 8 to the module with ID 8 and now considers it, its parent. In stage k + 1, it is now up to the module with ID 8 to instantiate the new module added to the system. Given that this module has an available free ID, 9, it sends out this ID to the new module via a type 5 message, and the new module is now assigned the ID 9. Also, type 9 messages are forwarded

Algorithm 3: Module addition handling

```
switch message.type do
  case 8
    children ← children ∪ < message.origin, 1 > ; subtree_size+ = 1
    len ← LENGTH(free_ids)
    if len > 0 then
      contiguous ← IS_CONTIGUOUS(free_ids)
      if contiguous = true then
        id_index ← len/2 ; id ← free_id[id_index] ; child_n_free_ids ← len − id_index − 1
      else
        subset ← GET_LAST_SUBSET(free_ids) ; id ← subset[0]
        child_n_free_free_ids ← LENGTH(subset) − 1
      for K ← 0 to child_n_free_ids do
        free_ids ← free_ids − {id + k}
      n_free_ids− = 1
      if leader = false then
        send type 9 message to parent
      send type 5 message to message.origin
    else
      if n_free_ids > 0 then
        if leader = false then
          send type 9 message to parent
        send type 5 message to itself
      else
        send type 10 message to parent
  case 9
    child ← find message.origin in children ; child.subtree_size+ = 1
    subtree_size+ = 1 ; n_free_ids− = 1
    if leader = false then
      send type 9 message to parent
  case 10
    child ← find message.origin in children ; child.subtree_size+ = 1
    subtree_size+ = 1
    if n_free_ids then
      if leader = false then
        send type 9 message to parent
      send type 5 message to itself
    else
      send type 10 message to parent
procedure NODE_ADDED( )
  parent ← neighbor ; subtree_size ← 1 ; send type 8 message to neighbors
end procedure
```

to the modules with IDs 6, 4 and 0, in order to notify them about the modification which occurred in their sub-trees, however these messages are omitted in the illustration. Going to stage k + 2, two main activities can be observed in the system. Firstly, the module on the upper left of the system, with ID 22, left the system. Before its departure, it sends out a message of type 6 to its parent, notifying it about its departure and providing it with its own unique ID and set of free IDs to be stored

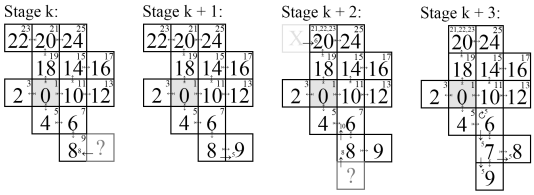


Fig. 2: example of the execution of various algorithm tasks related to the management of free IDs in the a given modular robots system.

in it. The parent, with ID 20, received this request, removes the module with ID 22 from its children set, and adds the IDs 21 and 22 to its own set of IDs, being now [21, 22, 23]. Also, type 7 messages are forwarded to the modules with IDs 18 and 0, in order to notify them about the modification which occurred in their sub-trees, however these messages are omitted in the illustration. Secondly, yet another new module is added to the system, below the module with ID 8. As in the previous case, the new module would send out a message of type 8 to the module with ID 8 and considers it now its parent. Finally, and in stage k + 3, it is now the responsibility of the module with ID 8 to instantiate the new module. However, the module with ID 8 does not have any free IDs left stored in it. Furthermore, "$\backslash n\_free\_ids$" of the module with ID 8 is equal to 0 (i.e., there are no free IDs within its own sub-tree). Thus, this module would have to delegate the task of ID re-assignment to its parent, via a message of type 10. Upon receiving the type 10 message, the module with ID 6 determines that free IDs are available within its sub-tree, and can be utilized in order to redistribute the IDs evenly to its children and descendants. It does so by sending a type 5 message to itself, launching a re-assignment task for its own sub-tree. This eventually results in an equal redistribution of IDs, with the new module being now assigned the unique ID 9. As showcased above, the overhead cost of maintaining the ordered logical tree of IDs is exceptionally low. This is a particularly attractive option when the alternatives are the re-execution of the whole algorithm, or the last phase of the algorithm, or even using broadcasting as a mean of message exchange between any two modules of the system. In the example above, in the case of the addition of the first new module, only a handful of messages were exchanged in order to assign to the new module a unique ID. In the second case of module addition, we can see that only 4 modules, out of 13, handled the assignment of a unique ID to the new module while maintaining the order of the IDs logical tree. The benefits of this algorithm only grow with larger system sizes. In a system composed of thousands of modules, no modules may need to be altered in order to accommodate to a new addition to the system, or just a dozen modules may be affected by this module addition, instead of the whole system. Furthermore, one can start understanding the wide range of possible configuration for this algorithm. For example, if many modules are expected to move on the leaf levels of the system, a free IDs distribution scheme giving more weight to lower level nodes could be implemented. If modules are predicted to move from the top area to the bottom area of a given system, more

free IDs could be reserved in the latter area. Moreover, the number of free IDs in the system could be easily altered by choosing how many bits to add to the ID length, basically defining the space of available IDs. One could think of this configuration as a trade-off between IDs length and overhead costs when moving modules around. A system with a high morphology variance and low number of messages transmitted may benefit greatly from a bigger IDs space. A system with few modules changing their position in the system and a high rate of message transmission may benefit more from a smaller IDs space.

5 Experimental results

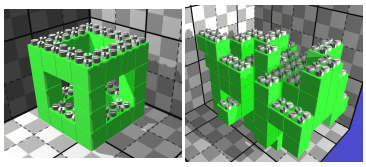
In order to evaluate our proposed algorithm, we conducted various simulations. We implemented the distributed approach in VisibleSim [8] which is a simulator supporting large-scales ensembles and different modular robot systems including Blinky Blocks used in our simulations. In fact, Blinky Blocks are centimeter-size blocks placed in a cubic lattice able to communicate and coordinate together through serial links on the block faces (neighbor-to-neighbor communication model).

To evaluate our proposed algorithm and test disposal of free IDs throughout the system, we carried out two different scenarios. The first scenario consists of adding modules to already existing shapes. The idea is to add modules on different shapes consisting of hundreds of modules. The second one is a reconfiguration scenario consisting of transforming a chain shaped system into an 'S' letter like shape.

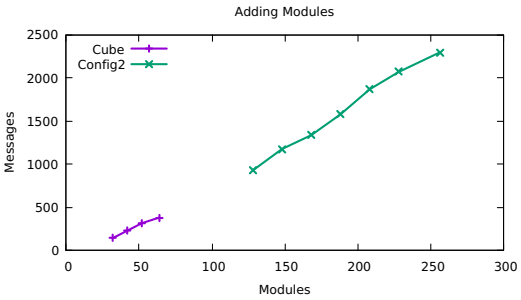
First Scenario: Adding new modules to existing shapes

In this scenario, we carried out several additions of modules on two different configurations. The first is a cube shaped one presenting very regular volumes with symmetries (cf. Figure 3a) and the second one representing a large set of randomly assembled blocks resulting in a variable number of neighbours disposed at each module (cf. Figure 3b). For the first configuration (cube shape), We added several modules to the system until we reached double the initial number of modules. We considered a system composed of 32 blocks and at each stage of the simulation, we added 10 new modules, until we reached the total number of 64 modules. The second example illustrated in Figure 3b shows a set of blocks assembled randomly with some irregularities. This kind of shapes is resulting in a variable number of neighbours. In this case, we considered a network of 128 modules and at each stage of the simulation, we added 20 new modules, until we reached the total number of 256 modules.

In the two cases and for the different scenarios, it has been clearly shown that our approach ensures the assignment of new IDs to all newly added modules. Furthermore, after the execution of our algorithm the tree has been updated and the leader knows all the routes to reach these new modules. However, the cost of this assignment is in the number of messages sent in the system. Therefore, we evaluated the number of messages exchanged while varying the number of added modules. In Figure 3c, we showed the results of the total number of messages exchanged during



(a) Cube (b) Random



(c) Number of exchanged messages

Fig. 3: Modular robots configurations and results

the execution of our approach. As aforementioned, we studied the number of messages exchanged while adding each time 10 or 20 blocks respectively to the cube and random configurations. The messages in the system are used to assign id for each block and distribute free spots for each one as explained in previous sections. As well, they are used for inter-module communication and notification purposes whenever a module joins/leaves the system. The results show that the number of messages linearly increases with the number of added modules. In order to be able to assign new ids, in the proposed approach the modular robot has to exchange an additional number of messages. On the other hand, it is more cost-effective than the case where the whole process has to be started all over again and from the beginning. Furthermore, our approach ensures easy access to specific identified nodes, which can be very useful for fault detection issue.

Second scenario: Self-Reconfiguration

In these series of simulations, we considered the free ids assignment in a self reconfiguration scenario where the number of modules is fixed and just the positions and neighbors of some modules change. We considered an initial chain shape of the modular robot to be reconfigured into the letter 'S' like shape. The self-reconfiguration algorithm proposed in [6] is used in this simulation. As a matter of fact, the idea is having modules moving from one position to another by disconnecting from the module's interface and joining another one having a free spot ready to fit in. The results showed that our approach can handle the reconfiguration from initial to final shape by involving both the removal and the addition of modules in the system after the initial execution of the algorithm. Furthermore, the order of the logical tree is maintained due to the free Ids disposal at each block resulting in a minimum number of modifications to be made in the system as discussed earlier in previous sections. We also evaluated the additional number of messages exchanged during this reconfiguration task. This increase is about 30% more messages compared to the messages sent just for the reconfiguration process.

6 Conclusion and Future Work

In this paper, we presented an extended version of our approach concerning the unique ID assignment of modular robots [2]. This new version was proposed, rendering the feasibility of fast and efficient inter-module communications in modular robots system much more likely. This was possible due to the implementation of a reconfiguration adapting algorithm, that handles the distribution and handling of holes or free IDs throughout the system. The three main tasks of this extended algorithm were thoroughly explained. The discussion and obtained results showed clearly the benefits of the extension of the algorithm, while providing insights as to how to define the IDs length in a particular modular robots system. In a future work, we plan on studying the effects of having multiple leaders distributed evenly throughout the system on the time and energy complexity of the algorithm.

References

1. Reem J Alattas, Sarosh Patel, and Tarek M Sobh. Evolutionary modular robotics: Survey and analysis. *Journal of Intelligent & Robotic Systems*, 95(3-4):815–828, 2019.
2. J. Assaker, A. Makhoul, J. Bourgeois, and J. Demerjian. A unique identifier assignment method for distributed modular robots. In *2020 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 3304–3311, 2020.
3. A. Brunete and al. Current trends in reconfigurable modular robots design. *Int. Jour. of Advanced Robotic Systems*, 14(3), 2017.
4. Md. Rakibul Haque, Mahmuda Naznin, and Rifat Shahriyar. Distributed low overhead id in a wireless sensor network. In *Proceedings of the 17th International Conference on Distributed Computing and Networking*, ICDCN '16, pages 12:1–12:4, 2016.
5. Jialiu Lin, Yunhuai Liu, and Lionel M Ni. Sida: self-organized id assignment in wireless sensor networks. In *IEEE Int. Conference on Mobile Adhoc and Sensor Systems*, pages 1–8, 2007.
6. Mohamad Moussa, Benoit Piranda, Abdallah Makhoul, and Julien Bourgeois. Cluster-based distributed self-reconfiguration algorithm for modular robots. In *35th International Conference on Advanced Information Networking and Applications (AINA 2021)*, may 2021.
7. Roberto Petroccia. A distributed id assignment and topology discovery protocol for underwater acoustic networks. In *Third Underwater Communications and Networking Conference*, pages 1–5, 2016.
8. Benoit Piranda, S Fekete, A Richa, K Römer, and C Scheideler. Visiblesim: Your simulator for programmable matter. In *Algorithmic Foundations of Programmable Matter*, 2016.
9. F. Pratissoli, A. Reina, Y Kaszubowski Lopes, L Sabattini, and Roderich Groß. A soft-bodied modular reconfigurable robotic system composed of interconnected kilobots. In *MRS*, 2019.
10. J. R Smith. Distributing identity symmetry breaking distributed access protocols. *IEEE Robotics & Automation Mag*, 6(1):49–56, 1999.
11. P. Thalamy, B. Piranda, and J. Bourgeois. Distributed self-reconfiguration using a deterministic autonomous scaffolding structure. In *AAMAS '19*, pages 140–148, 2019.
12. Meibao Yao, Christoph H Belke, Hutao Cui, and Jamie Paik. A reconfiguration strategy for modular robots using origami folding. *The International Journal of Robotics Research*, 38(1):73–89, 2019.
13. Hongbo Zhou, Matt W Mutka, and Lionel M Ni. Reactive id assignment for sensor networks. In *IEEE International Conference on Mobile Adhoc and Sensor Systems Conference, 2005.*, pages 6–pp. IEEE, 2005.