ÉCOLE POLYTECHNIQUE FÉDÉRALE DE LAUSANNE

BIOLOGICALLY INSPIRED ROBOTICS GROUP (BIRG)

Distributed Roombot Locomotion and Self-Reconfiguration

Philippe Laprade

January 15, 2010

Master Project

Supervised by Professor Auke Ijspeert and Alexander Sproewitz

Abstract

Modular robotics is a new and exciting research field offering many potentially useful applications. Their hidden power lies in their abilities to reconfigure in order to adapt to their environment, perform tasks with greater efficiency, repair themselves and replicate. Self-reconfiguration in modular robotics is however a hard problem to solve. We will introduce modular robots and share some interesting locomotion, self-reconfiguration and self-organization algorithms available in the literature. In this report, we will then study a new generation of modular robots designed by the Biologically Inspired Robotics Group named Roombots. We will describe a new decentralized framework that allows them to move and self-reconfigure with near real-time competitive performances.

Contents

Chapter 1

Modular Robotics

1.1 Introduction to the world of modular robots

Welcome to the world of modular robotics, a world full of potential and surprises. The field of modular robots is very young and we are still waiting for many major breakthroughs to happen. Most of us know robots as one big mechanical and electrical structure that was built to fulfill one particular purpose. We usually imagine a robot as a humanoid machine that mimics human behavior to some extend. Perhaps did you see robots that can walk, dance or even ride a bicycle. You may also have seen dog-like robots that can learn their names, execute tricks and mimic the behavior of living biological dogs.

All these robots share one thing in common. Although they may be extremely complex, they are designed as one single entity. Their shape, size and morphology have been studied so that the robot can fulfill it's purpose. They are unable to perform tasks for which they have not been designed for. If a robot's arm isn't barely long enough to reach for an object, it will not be able to transfer some of it's mass to the arms in order to extend them. They were just not designed to reach objects so far away. Once a normal robot has been built, the complexity of their design usually makes it very hard for engineers to change the original function of the robot. If a robot was designed to built car doors, it is very unlikely that it will be able to do anything very different from that.

(c) Atron modules [1]. Image source: http://www.mrt.dis.titech.ac.jp

(d) M-Tran modules [2]. Image source: http://www.aist.go.jp

(e) Roombot modules [3]. Image from BIRG [4]

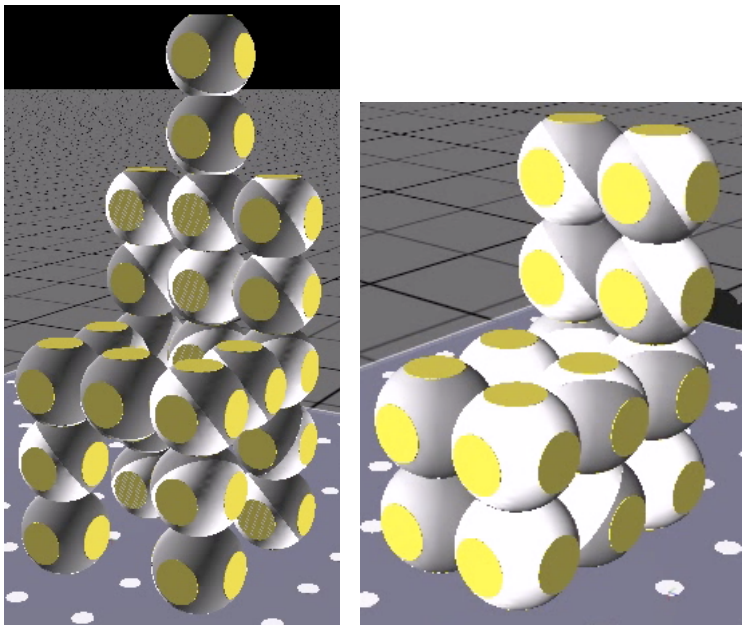
Figure 1.1: Different modular robot modules at work

Modular robots offers a very elegant solution to the structural rigidity of "normal" robots by allowing them to physically reconfigure their parts in order to adapt themselves to the task at hand. Unlike a normal robot that is made out of one big and complex structure, modular robots are made out of many simple and elegant identical pieces. Usually, modular robots reconfigure themselves by changing the way the pieces are connected to one another. This allows them to assume different forms which in turn can fulfill many different functions. Creativity is the only barrier preventing modular robots

from achieving what they need to do. The functions that a modular robot can fulfill are almost unlimited and are defined by the shape of the modular robots and not by the initial design and manufacturing decisions. It is still possible to find many new functions for modular robots even years after they have been built. Using some artificial intelligence, we can even let the modular robots find new functions by themselves that humans might never have thought of. The shear power of modular robots comes from their seemingly unlimited potential.

Figure 1.1 shows three different types of modular robots currently being studied, namely the Atron [1], the M-Tran [2] and the Roombots [3]. There have been dozens of different models that have been developed in the history of modular robotics and there will still be many more to come.

Modular robots allow for new paradigms to emerge that could never have been imagined in normal robotics. Because of their simplicity of design, a modular robot can self-repair very easily simply by replacing failed modules by new identical modules. As all the pieces are identical and much simpler than normal robots, costs of manufacturing them are usually very low. Replacing failed robot modules by other new modules is thus very simple and cost efficient. In some very harsh situations such as space exploration where the number of modules are limited, a modular robot may still be able to resume normal operations even if many of it's pieces have failed. The remaining functional pieces can choose to reconfigure themselves in such a way that they may still be able to perform well even if they are not as efficient as with the full set of pieces. If the leg of a normal robot fails, it is very likely that the complete motor functions of the robot will be compromised and that there will be no way for the robot to resume normal movement. On the other hand, if the leg of a modular robot fails or gets destroyed, the robot may decide the assume a new, smaller shape with smaller legs and resume normal operations at lower speeds.



(a) Stool made out of 6 Roombot meta-modules (24 cubes)

(b) Stool made out of 4 Roombot meta-modules (16 cubes)

Figure 1.2: Roombots reconfiguring themselves to maintain their function with fewer modules

The figure above shows how modular robot reconfiguration can be useful to maintain functionality even when modules are damaged or destroyed. The initial structure 1.2(a) is made out of 6 Roombot meta-module. Their goal is to provide a stool-like structure for people to sit on. If two of the meta-modules like the legs of the chair become damaged, the Roombots can decide to reconfigure themselves into a smaller stool 1.2(b) in order to maintain their functionality. This is just a simple example of what modular robots are capable of doing and the implications are great.

The design and development of modular robots has been greatly influenced by the close study of biological systems. We would like to mimic the incredible elegance and healing power of our own bodies, which are made of simple and elegant pieces we call cells. Our cells replicate so that newer cells can take over the operations of older failing cells. When we are hurt and cells are destroyed, we produce new cells that heal our wounds and take over the functions of the missing cells. In a way, we are an almost perfect self-repairing modular system made out of many cells.

Modular robots do have one big drawback compared to standard robots. Designing and controlling them is an incredibly challenging task for engineers. The more modules we add into our modular system, the harder it becomes to control them in a coherent and functional way. Many parameters need to be taken into account such as inter-module communication, collisions, cooperation and structural stability. The more module are integrated into a modular system, the greater the computational complexity of any meaningful action. Even on a purely engineering standpoint, we are still waiting for large-scale modular robot systems of thousands of modules or more to be demonstrated. Although the field seems very promising, we have still much to learn and much to develop before modular robots will one day serve the coffee and walk the dog.

1.2 Introduction to centralized and decentralized algorithms

If there is one thing that sets modular robots apart from their normal robot counterparts, it is definitively their ability to self-reconfigure. This skill allows them to adapt themselves to dynamic changing environments and to perform specific tasks with greater efficiency. On the other hand, self-reconfiguration is by far not a trivial problem to solve. Naive reconfiguration algorithms require exponential computation time at best in the number of modules in the system. Combinatorial explosion is inevitable if we seek optimal solutions for large amounts of modules.

As it is for all interesting problems in robotics and computer science, the self-reconfiguration problem is computationally hard to solve. This opens the door to many heuristic algorithms that solve the self-reconfiguration problem by trading off optimal solutions and computation time. Typically, these algorithms will find sub-optimal but still acceptable solutions in relatively acceptable computation time. Two categories of heuristic algorithms can be distinguished, namely centralized and decentralized self-reconfiguration algorithms. In centralized algorithms, all the actions of every individual module in the system are computed by one entity. Once a complete plan for every module has been generated, individual orders are sent to the modules and executed. Individual modules don't make decisions on their own and don't take any initiatives. They simply follow orders. In decentralized algorithms, there exists no supervisor ordering the modules what to do. The whole decision making process is implemented into the individual modules themselves. They will communicate with each other and work together in order to solve the reconfiguration problem.

Perhaps an example might help clarify the difference between these two concepts. Consider the real case scenario of airplanes flying over the Swiss airspace. Security is a very high priority in this situation and pilots are not authorized to take decisions by themselves without notifying an air traffic controller. It is usually the air traffic controller that orders the pilot to change altitude, speed and course to avoid collisions with other aircrafts in the air. This might be considered as a centralized system where the individual modules are the airplanes and the centralized decision maker is the air traffic controller. Now consider cars driving through the city of Geneva during peak traffic hours. Everyone driving a car knows the common traffic rules (hopefully). All the decisions are made by the individual drivers themselves by following a set of common rules known to all drivers. The traffic is completely self-regulated and all decisions are made by the drivers themselves. This is a good example of a decentralized system.

Centralized algorithms tend to me more efficient in finding optimal solutions for small sets of modules. They are usually very inefficient in computation time when the number of modules grows larger. We use them when we are dealing with small sets of modules and when optimality or close-to-optimal is an important criteria. Decentralized algorithms might produce solutions less optimal than with centralized algorithms but have the advantage of requiring reasonably less computation time. We use them when dealing with large amounts of modules and when optimality is not a critical feature of the system.

1.3 Project description

In this project, we will study decentralized locomotion and self-reconfiguration algorithms for the Roombot modules [3]. The Roombots are modular robots developed at the Biologically Inspired Robotics Group (BIRG, see [4]) and can be seen in Figure 1.3. They are building blocks for adaptive furniture such as tables, chairs and benches that can move and reconfigure themselves. Their six degrees of freedom and their ability to attach and detach themselves grant them a lot of flexibility and power. A lot of work has already been done by former BIRG students on these modules to understand their movement and reconfiguration capabilities. Inverse kinematic models [5] have been designed and centralized self-reconfiguration models using graph theory [6] have been studied. The goal of this project is to study these modules further in the direction of decentralized methods for locomotion and self-reconfiguration. Here is the list of tasks for this project

as well as the deliverables.

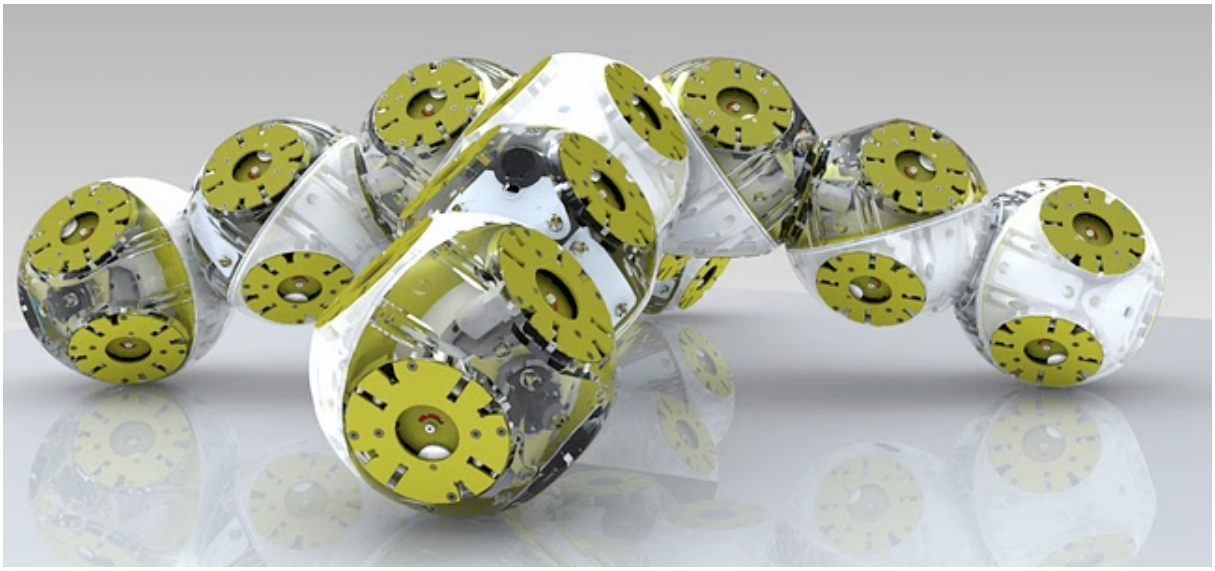


Figure 1.3: Roombot modules. Image from BIRG [4]

1.3.1 Task List

1. Get familiarized with modular robot literature and study previous work on Roombot modules.
2. Get familiarized with simulating Roombots on the Webots platform [7].
   - Understand and run Mikael's work on *Kinematics Considerations for Movement Optimizations* [5]
3. Study centralized self-reconfiguration methods.
   - Graph theory approaches ([8] and [9])
4. Study decentralized self-reconfiguration methods:
   - Potential fields [10]
   - Stochastic assembly approaches [11]
   - Hormone-inspired self-organization ([12] and [13])
   - Non-dense frame structures [14]
   - Cellular automata and gradient control [15]
5. Adapt, implement and simulate a decentralized method on the Roombots for a particular shape.
   - Study Roombot movement primitives, collision and deadlock detection.
   - Find a good set of distributed rules using local communication.
   - Simulate and visualize results through Webots.
   - Performance and quality evaluation.
6. Study the generalization of the decentralized method to arbitrary shapes
7. Record videos of self-reconfiguring Roombots.
8. Hand in final report.

1.3.2 Deliverables

1. A timetable including the above points as milestones, the mid-defence, the final submission date and the final presentation date.
2. A "cahier des charges".
3. Code, software with documentation and videos on a CD.
4. Final report
5. An entry on the BIRG student web-page summarizing the work on the project (see [16]).

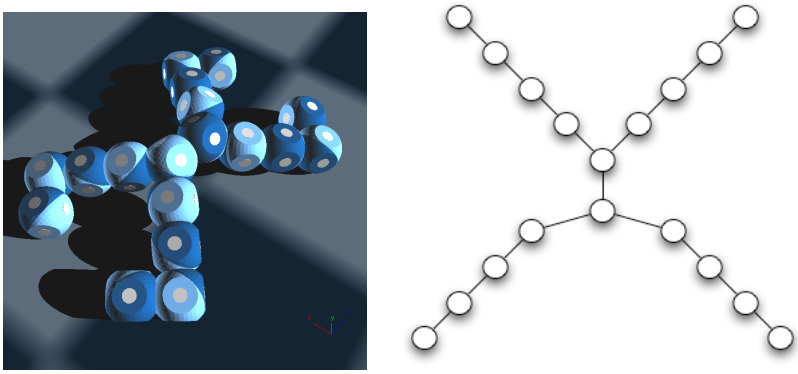
1.4 Theory of modular robot reconfiguration

Modular robots typically come in two different types : lattice-type and chain-type. Lattice type modular robots can only move through self-reconfiguration. They have no power joints that allow them to move in one particular configuration. They have to continuously reconfigure themselves in order to move. Chain-type modular robots can also reconfigure themselves but have the additional advantage of being able to move in one configuration without the need of reconfiguring. They usually don't use reconfiguration for movement, only to adapt themselves to new tasks or changing environments. We will study essentially chain-type robots as Roombots fall into that category. Their movements are more complex than lattice-type robots, the servo's might not be able to lift big chains of modules and their movements are typically subject to self-collision.

1.4.1 Graph algorithms

Most of the centralized self-reconfiguration algorithms for modular robots use graph theory. We will present the basic principles of solving graph-based centralized self-reconfiguration problems using a method proposed by Asadpour et al (see [8] and [9]). It is based on a heuristic search through robot graphs. Let us first introduce some fundamental notions.

When multiple robot modules are connected together, they form a configuration. Servo movements don't modify the configuration of the robots, only locking and unlocking and changing the connections between the modules. A configuration can be represented as a directed graph $G(V,E)$ where the vertices V are the robot modules and the edges E are the connections between the modules. The direction of an edge always goes from a male connector to a female connector. If the connectors happen to be genderless, the edge is undirected. Figure 1.4 is an example of a configuration made out of connected Roombots and one possible graph representation of that configuration.



(a) Roombot modules in a configuration. Picture from Simon Lepine [17]

(b) Possible graph representation of the configuration in Figure 1.4(a)

Figure 1.4: Roombot configuration and the corresponding graph representation

The edges of the graph are typically labeled according to which connectors of the modules are locked and their relative orientation. If every module has 6 connectors and there are 4 relative possible orientation for a connection than there are 144 ($6 \times 6 \times 4$) possible edge labeling. The main idea is to find a plan that will transform a start configuration to a desired end configuration using a graph search. The search is guided by two heuristics, namely a graph edit distance metric and a graph signature. The graph edit distance helps the search process to focus on the branches that are closer to the goal. Additionally, every time a graph is visited, it's graph signature is store in a database so that when the same graph is encountered again it will not be computed a second time.

The graph edit distance provides a graph search algorithm an indication where to search next to find a good or optional solution. It is usually defined as the shortest sequence of operations (delete and insert) on edges and vertices that transforms one graph into another one. The graph edit distance $\delta(I,F)$ has the following relationship to the MCS (Maximum Common edge-induced Subgraph) problem:

$$\delta(I,F) = 1 - \frac{|MCS(I,F)|}{max(|I|,|F|)}$$

As the *MCS* problem is NP-Complete and thus difficult to compute in general, we can use an upper bound of *MCS* to compute the graph edit distance as follows. C_l is defined as the number of edges in a graph that are labelled l.

$$\sigma_{UB}(I,F) = 1 - \frac{\sum_{l=0}^{l_{max}} min(C_l^1, C_l^2)}{max(|I|,|F|)}$$

This result comes from the observation that for a graph to be considered as a subgraph of another, the number of edges must be equal with respect to their labelling. This is a necessary but not sufficient condition, thus offering only an upper-bound to the *MCS* problem. The essence of centralized graph-based reconfiguration algorithms lies in finding good graph distance metrics such as this graph edit distance. Other heuristics and methods can also be used to refine the search process. The computation time of these methods are largely bound by the complexity and morphology of the corresponding modular robots.

1.4.2 Hormone-inspired locomotion and self-organization

Let us present the basic principles behind hormone-inspired locomotion and self-organization algorithms using the method from Shen et al. [13] called DHM (Distributed Hormone Model). This method is elegantly designed to solve specific problems in modular robot locomotion and self-organization with large amounts of modules. It is however not very well suited for precise controlled self-reconfiguration as the method is relatively stochastic in nature. The DHM model is a biologically inspired method based on how cells organize amongst themselves to form global patterns without any centralized control mechanism. Cells are known to communicate through hormone diffusion and do not have any global identifiers that differentiates them from other cells. The DHM model is based exactly on those properties.

The main concept behind the DHM model is that robots secrete activator and inhibitor hormones and respond to the concentration of activator and inhibitor hormones in their environment. They are typically attracted by activator hormones and repulsed by inhibitor hormones. The actions that a robot executes are probabilistic and depend on the received hormone concentration but also on connector information, sensor information and various local variables.

In a DHM model, the robots will usually form different patterns based on many parameters. One very important parameter in the DHM model is the ratio of activator/inhibitor hormones generated by the robots. One pattern that hormone-based methods can create are feathers. By studying the generation of feathers with varying parameters, we can understand the effects of activator and inhibitor hormones in the DHM model more accurately. For instance, for a fixed activator/inhibitor ratio, the size of the feathers are fixed regardless of the amount of initial robots or cells that are available in the system. If fewer robots are available for a fixed hormone ratio, the final feather pattern will simply be less dense. We also noticed that by varying the ratio of activator/inhibitor hormones, we effectively modify the size of the feathers. The more activator hormones, the bigger the feathers. The more inhibitor hormones, the smaller the feathers, regardless of the amount of initial robots or cells available.

The DHM model seems to be a method of choice in the following situations.

1. Searching for a target.
2. Spreading out inside an unknown environment.
3. Self-repair unexpected damage.
4. Surmount obstacles.

Figure 1.5 is a conceptual representation of how robots spread out in an unknown environment by creating inhibitor hormones. Figure 1.6 is another representation of how robots can self-repair a pattern if some of the robots are damaged.

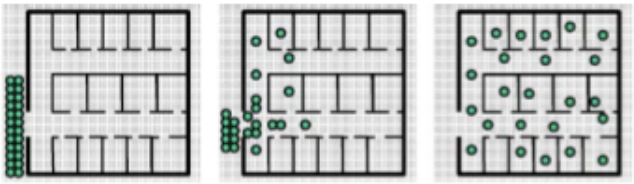


Figure 1.5: Robots exploring unknown space in the DHM model. Image from Shen et al. [13]

Searching for a target in unknown space can be implemented in the following way. All the robots initially start by creating inhibitor hormones that repulse the robots from one another. They start covering the whole space available.

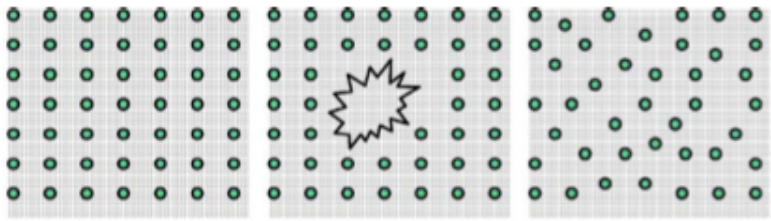


Figure 1.6: Robots repairing a pattern in the DHM model. Image from Shen et al. [13]

When one of the robots finds the desired target, it starts secreting activator hormones that attract other robots towards it. Similarly, spreading out robots inside an unknown environment is done by creating inhibitor hormones that forces the robots to spread out maximally, taking up all of the space available.

Self-repairing unexpected damage is also an interesting feature of the DHM model. Let us suppose a pattern has been formed and stabilized by a large amount of robots such as a feather. If we remove some robots in one location of the pattern or if they get destroyed, the pattern will be momentarily unstable. All the robots in the whole feather will move to fill the empty space and stabilize the new, less dense feather pattern. This result, as seen previously, comes from the fact that the global pattern is only controlled by the hormone diffusion and concentration, and not by the amount of robots available in the system.

Hormone inspired methods are very interesting and useful for solving specific locomotion and self-organization problems that don't require precise control over individual robots as they are mainly stochastic methods. However, they may not be useful in solving self-reconfiguration problems but may be handy in solving sub-problems of the self-reconfiguration problem such as collision avoidance.

1.4.3 Cellular automata and gradient fields

Stoy [15] presents an interesting self-reconfiguration method for modular robots based on seed-modules, cellular automata and gradients. The method assumes that we are working with simple cubic robots that have 6 connectors (one on each of their sides). The robots can slide on the surface formed by other robots and they can only communicate with direct neighbours. Even though these simplifications tend to make the problem easier, we will inspire ourselves from this method to implement a decentralized self-reconfiguration method for the Roombots that we will present in the next chapter. Here are the general steps involved in this self-reconfiguration method:

1. The user defines the three-dimensional model of the desired final configuration using a CAD tool.
2. The representation of the final configuration is transformed into a cellular automaton.
3. The cellular automaton is made porous to avoid local minimums during the self-reconfiguration process. Local minimums can happen if some unused modules get trapped inside other modules and can't get out.
4. Launch the self-reconfiguration process using a seed and gradients to guide the growth progress.

Figure 1.7 shows the self-reconfiguration process at work. Many simple cubic robots work together to create a chair-like structure.

The first part of the method is to transform a three-dimensional object into a porous cellular automata. Stoy [15] presents a systematic way of doing this. The structure of the CAD model is approximated by placing a cube inside the volume and adding additional cubes recursively until the whole volume of the model is filled. The volume is then made porous by enforcing a special substructure inside the whole volume. This structure ensures that the self-reconfiguration process will converge as the robots can never get stuck because there exists no local minimum. Figure 1.8 shows the form of that special substructure.

The self-reconfiguration process can now start. All the robots have a copy of the final configuration and are configured randomly. An arbitrary module is assigned to be the seed module for an arbitrary position in the final configuration. If there are neighbors missing around the seed module, the seed creates and propagates a gradient vector. The source of the gradient vector is the seed module which has a value of 0. The seed propagates the gradient to all neighbor modules by

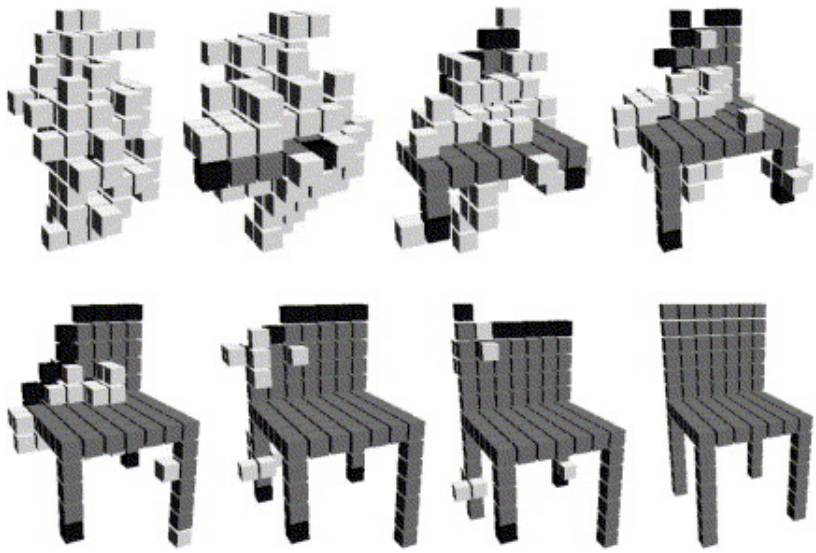


Figure 1.7: Robots reconfiguring themselves from an initial random configuration into a chair. Image from Stoy2006 [15].

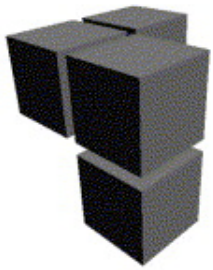


Figure 1.8: Substructure used to create complete structures that allow for the reconfiguration process to converge. Image from Stoy2006 [15].

adding 1 and sending the vector direction to the seed. All modules then continue propagating the gradient vector to their neighbors by adding 1 to the smallest received value. Free modules will then go down the gradient to fill the holes in the structure.

Once a module is in place, it becomes itself a seed module and creates gradients for it's missing neighbors until all neighbors are filled. All seed modules are called "finalized" because they have arrived at their final destination in the configuration and do not need to move anymore. Furthermore, in this algorithm, special care must be taken when moving a robot as we want the whole structure to stay connected. According to the author, the configuration stays connected only if a module moves when:

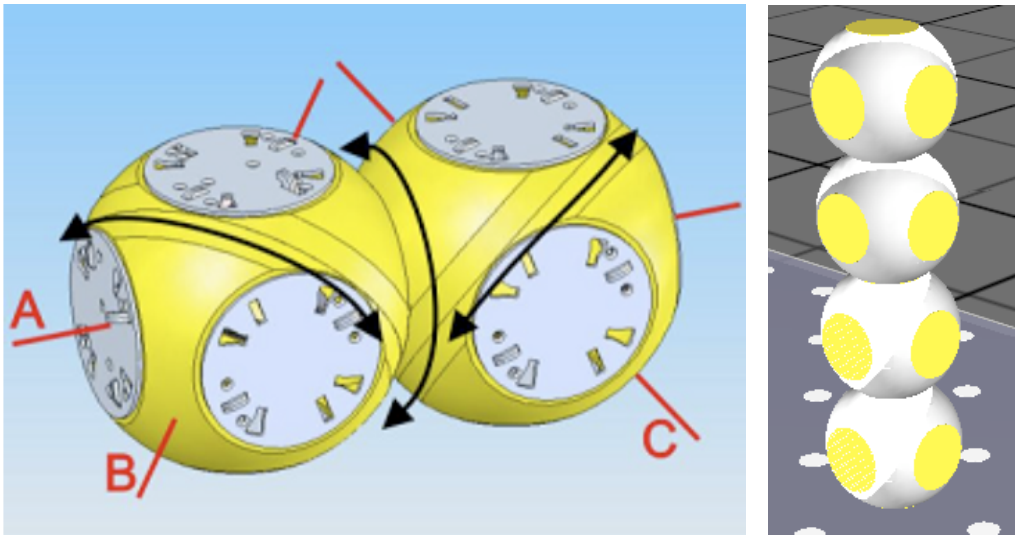
1. The connection gradient is initialized in the module and its neighbors.
2. Moving the modules does not change the gradient in neighboring modules.
3. It is not a source

By following these three rules and by assuming that the gradient in a module is un-initialized every time it moves, it is guaranteed that the configuration stays connected. These rules are interesting as they can easily be implemented on the modules in a distributed way. This method is relatively elegant and easy to implement. The only downfall is the simplifications that are made on the robot modules. They are perfect cubic robots that can freely slide along the structure. This assumption can usually not be generalized to all modular robots and the Roombots we will be studying certainly do not fall under this category. The reconfiguration process will have to take into account the more complex movement

conditions of the modular robots.

1.5 The Roombot modules

The Roombot modules are modular robots developed by the Biological Inspired Robotics Group [4] at EPFL. They are designed to build adaptive and reconfigurable furniture such as chairs, tables and benches, thus their name Roombots. One Roombot module can be seen in Figure 1.9(a). It is made out of two spherical cubes and has three degrees of freedom. The two cubes have a servo that allows them to rotate in the direction of their greatest diagonal (rotation axis B and C in Figure 1.9(a)). The two cubes are also connected together by a servo that allows the modules to rotate relatively to each other with respect to the rotation axis A. On every one of it's faces, a Roombot module has a connector which allows it to connect with other Roombot modules and the floor.



(a) Roombot module. Image from Mayer2009 [5]

(b) Roombot meta-module

Figure 1.9: Roombot module and meta-module

By connecting two Roombot modules together, we obtain a Roombot meta-module as seen in Figure 1.9(b). The meta-module is more powerful than a simple module as it has 6 degrees of freedom rather than 3 and can reach for more points in space. Two Roombot modules can be connected in four different ways to form a Roombot meta-module as seen in Figure 1.10. For more information on the Roombot modules, meta-modules and the kinematic chains of the Roombots, please read the report of Mayer2009 [5].

Let us define a few terms that we will be using in the rest of this report. When referring to a Roombot module, we are referring to the simplest Roombot structure possible made out of two cubes (Figure 1.9(a)). When speaking of a Roombot meta-module, we refer to two Roombot modules connected together (Figure 1.9(b)). We can individually rotate the six servos of a Roombot meta-module so that it can assume different configurations. We will use the term Roombot configuration to define a specific set of rotations on a Roombot meta-module. Figure 1.11 shows a Roombot meta-module in different possible configurations. Finally, when speaking of a Roombot structure, we refer to the structure that the Roombots can build by connecting themselves together. Figure 1.12 shows a picture of a Roombot structure made out of 6 Roombot meta-modules.

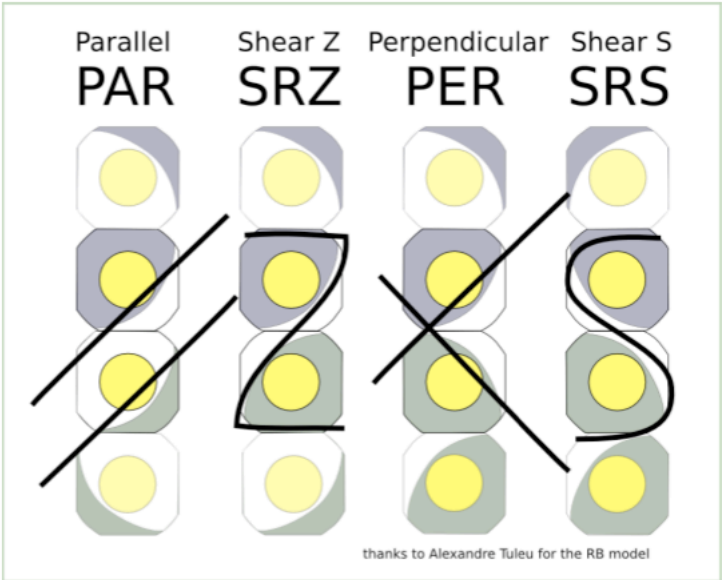
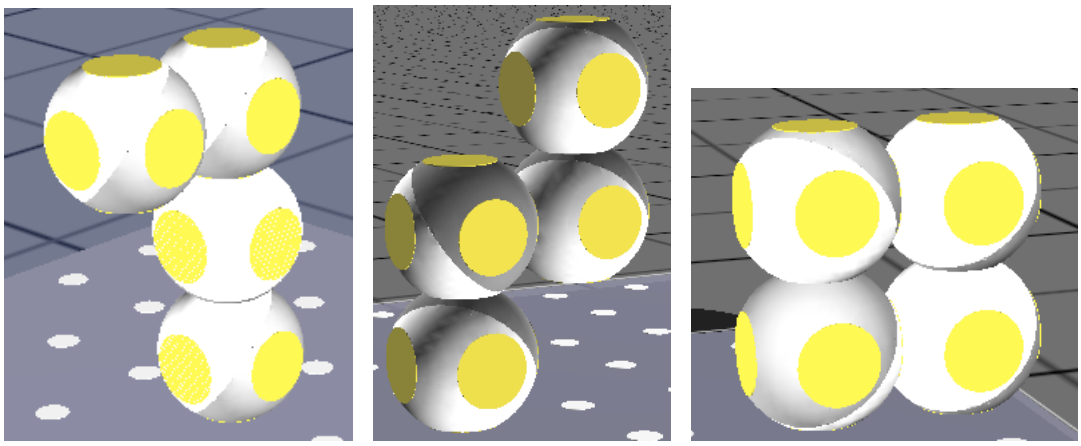


Figure 1.10: Roombot meta-module connection types. Image from Mayer2009 [5] and Tuleu2009 [18]



(a) Roombot meta-module configuration 1 (b) Roombot meta-module configuration 2 (c) Roombot meta-module configuration 3

Figure 1.11: Roombot meta-module in three different configurations

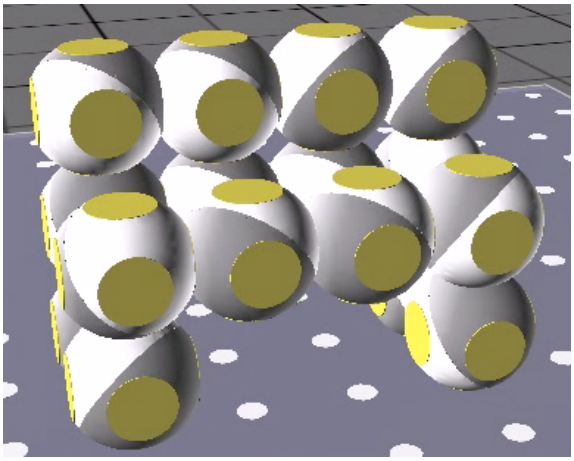


Figure 1.12: Roombot meta-module bench structure

In the next chapter, we will present the Roombot decentralized framework, an algorithm that allows Roombots to self-reconfigure into different structures. We will be working inside a three-dimensional grid world where a grid position corresponds to the space occupied by one cube of a Roombot module. Unless they are moving between two different configurations, the cubes of a Roombot will always reside completely inside the 3D grid. We will also use only two active connectors on the Roombot meta-modules during the whole reconfiguration process, one on each of the extremity cubes. The other connectors are passive, which means that only other active connectors can lock on them. The floor of the world in which the Roombots evolve is also made out of passive connectors on which the Roombots can lock on to (the white dots in Figure 1.12).

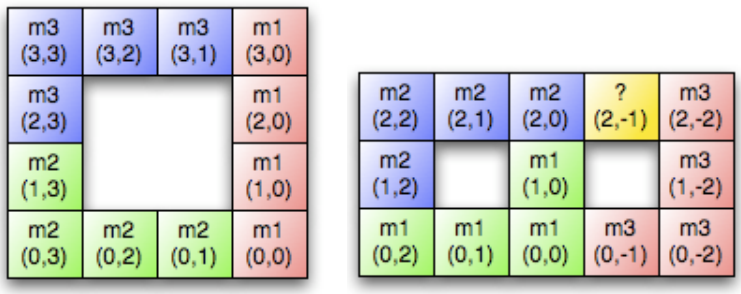
Chapter 2

Implementation

2.1 Our solution : seed modules and gradient control

To solve the problem of decentralized self-reconfiguration and locomotion for the Roombots, we have decided to implement a solution inspired by Stoy 2006 [15] and adapted it to the the needs of the Roombot meta-modules. In this decentralized approach, each Roombot meta-module moves down a gradient field to fill seed positions. Once the module has filled a seed position, it becomes itself a seed module that can advertise new seed positions. The idea is to build a Roombot structure such as a chair or a table by successively filling seed positions inside this structure. Let us go over the different steps required to get such a method to work.

First, we need to define a three-dimensional model of a structure we want to build such as a chair, a bench or a cup holder. This step can be done using different 3D CAD tools available on the market. We then have to fill this three dimensional structure with real Roombot meta-modules, effectively assigning meta-modules positions and configurations inside the structure. Sometimes, it might not be possible to completely fill the 3D structure with meta-modules. In those cases, we try to fill the three-dimensional structure in the best possible way by maximizing the number of filled structure positions without compromising structural integrity as in Figure 2.1(b). The yellow cube at position $(2,-1)$ could not be filled by a meta-module but will not compromise the structural integrity of the structure if it is missing. This step can be done either manually by assigning Roombot meta-module positions inside the structure by hand or automatically. Obviously, the automatic solution is more convenient for the end-user but is more complex to implement as structural integrity must be satisfied. In it's current state, the framework requires to manually assign the meta-modules inside the structure.



(a) Completely filled structure (b) Partially filled structure

Figure 2.1: Two structures filled with Roombot meta-modules (4 cubes)

Figure 2.1 contains an example of 2 Roombot structure (in 2D for simplification) that have been filled with Roombot meta-modules. A Roombot meta-module is 4 cubes long, is of the same color and has an identification number (m1, m2 and m3). The first structure 2.1(a) has been completely filled with meta-modules whereas the second structure 2.1(b) could only be partially filled. The yellow cube is part of the structure we would like to build but there is no Roombot meta-module that can occupy this position.

Once the Roombot meta-modules have been assigned inside the three-dimensional structure, we must specify an order of construction. Once launched, the Roombot meta-modules will try to fill the different positions inside the structure. To

ensure that the whole structure does not collapse during it's construction, we would prefer that the Roombot meta-modules build the structure in a bottom-up approach, starting by filling first the lower positions of the structure and climbing their way up as the structure is being built. Again, this process can be done either manually or automatically. First, we select a set of Roombot meta-module positions as starting seed positions that Roombots will try to fill first. Then, for every Roombot meta-module positions, we define a set of new positions inside the structure that will effectively be "unlocked" as soon as it is filled by a real meta-module. The idea is to create a build order that enforces a bottom-up approach, unlocking the upper meta-module positions when the lower ones have been filled. In it's current implementation, an easy framework is provided to build a Roombot structure by hand and force a build order on it.

For example, in the first structure 2.1(a), the Roombot meta-module m3 can not be filled until the meta-module m1 or m2 is in place. The first seed meta-module could be m1. Once a Roombot meta-module has filled this configuration, m2 is unlocked and then m3. This will enforce a bottom-up approach on the construction of the structure. In figure 2.1(b), we can start by advertising simultaneously meta-modules m1 and m3 as both are rooted on the floor and are independant from each other. Once m1 is filled, m2 will be unlocked.

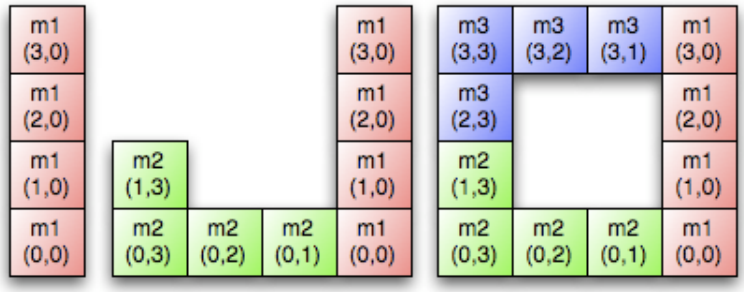


Figure 2.2: Building steps of the structure in Figure 2.1(a)

The next step to solving the self-reconfiguration problem is to allow the Roombot meta-modules to move and effectively fill seed positions inside the structure. This is where the gradient field comes in. Every Roombot meta-module is itself an independent controller with limited knowledge of the world. It makes local decisions based on the content of the world in it's immediate surroundings and the direction of the gradient field at it's positions. The gradient field is built in such a way that if a Roombot meta-module were to move in it's direction, it will inevitably find a seed position in the three-dimensional structure to fill. The Roombot meta-module will thus try to move in the direction of the gradient vector until it can see a seed position in it's immediate surroundings that it can fill. A Roombot can see three grid positions in every direction. Many problems arise in this steps such as avoiding collisions with the environment and with other modules, choosing an appropriate move and avoid getting stuck.

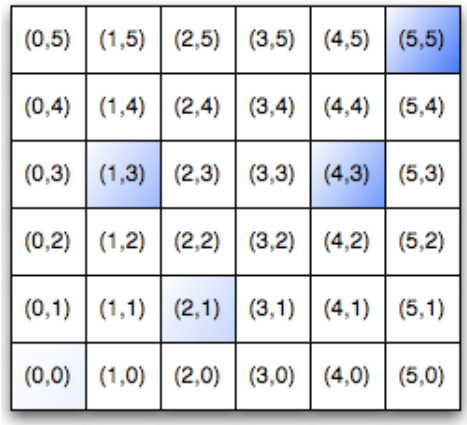


Figure 2.3: Example of a Roombot meta-module moving in a 2D grid (top view of the grid)

Figure 2.3 is an example of a Roombot meta-module moving inside a 2D grid. It starts at position $(0,0)$ and wants to

fill a seed at position $(5,5)$. In order to get there, it will pass through grid positions $(2,1)$, $(1,3)$, $(4,3)$ and $(5,5)$. Please visit my project website [16] to view the Webots Roombot video corresponding to this example.

2.2 Configuring Webots

The decentralized reconfiguration framework for the Roombots is written in $C++$ for the Webots [7] platform. Webots allows us to simulate the reconfiguration process of the Roombot modules by implementing robot controllers. The decentralized framework contains two controllers : the WorldController and the SeedController. Only one WorldController controller is loaded during any webots simulation. It is a centralized controller that provides all the other controllers with information on the content of the world and the direction of the gradient vector. The SeedController contains all the distributed logic of a Roombot meta-module. It needs to be loaded into every Roombot meta-module that wants to take part in the reconfiguration simulation.

You will need one supervisor robot in the Webots simulation with the WorldController loaded. It will require an input argument of the following form.

```
-s Chair.xml
```

By doing so, you ask the WorldController to load a specific Roombot structure contained in the provided xml file. It is possible to change the structure that the Roombots will build simply by changing this line in the arguments of the WorldController. Please refer to the section on the WorldController to understand how to build new structures with the correct xml format.

The SeedControllers should also be loaded into supervisor robots. Each SeedController will be in charge of controlling the movements and reconfiguration process of one meta-module (two Roombot modules). The input arguments required for the SeedController are of the following form.

```
-a 7 -b 8 -w 200 -s SRS
```

The *a* and *b* arguments correspond the the ID's of the two Roombot modules in the webots simulation that make up the Roombot meta-module. The *w* argument is the ID of the WorldController robot in the webots simulation. Finally, the *s* argument is the connection type between the two Roombot modules making up this Roombot meta-module. For the simulation to run properly, the individual Roombot modules in webots should be loaded with the kine_controller controller. It's main task is simply to listen for instructions coming from the SeedController and executing them locally on their respective Roombot modules.

If you wish to recompile the decentralized framework (WorldController and SeedController), it will be necessary to configure the respective makefiles for your own workspace and install the following libraries. The framework was built for the Webots 6 API and it may need to be ported in the future if there are some major changes in the API.

1. CMake (cross-platform build system used to build Eigen and KDL, [19])
2. Eigen2 (C++ template library for linear algebra [20])
3. KDL (Kinematics Dynamic Library, [21])
4. TinyXML (XML Parser Library [22])

2.3 Project Overview

The Roombot Decentralized Controller (RDC) is essentially a Webots controller implementing the decentralized self-reconfiguration solution explained in the previous section. It is made out of many modules as shown in Figure 2.4. The distributed self-reconfiguration controller is built on top of the Mayer 2009 [5] framework which can be seen as the orange blocks in Figure 2.4. Mayer designed a kinematic model of the Roombot meta-modules which will be used to compute end effector positions and rotations throughout our implementation. The meta-module supervisor controller is a webots controller designed to manipulate two Roombot meta-modules connected together in either one of the possible connection types : SRS, SRZ, PAR or PER. It essentially forwards movement and lock/unlock instructions to the single-module controllers under it's supervision. The module kinematic controller listens for messages coming from the meta-module supervisor controller and executes the corresponding servo rotations and lock/unlock instructions as requested. It works

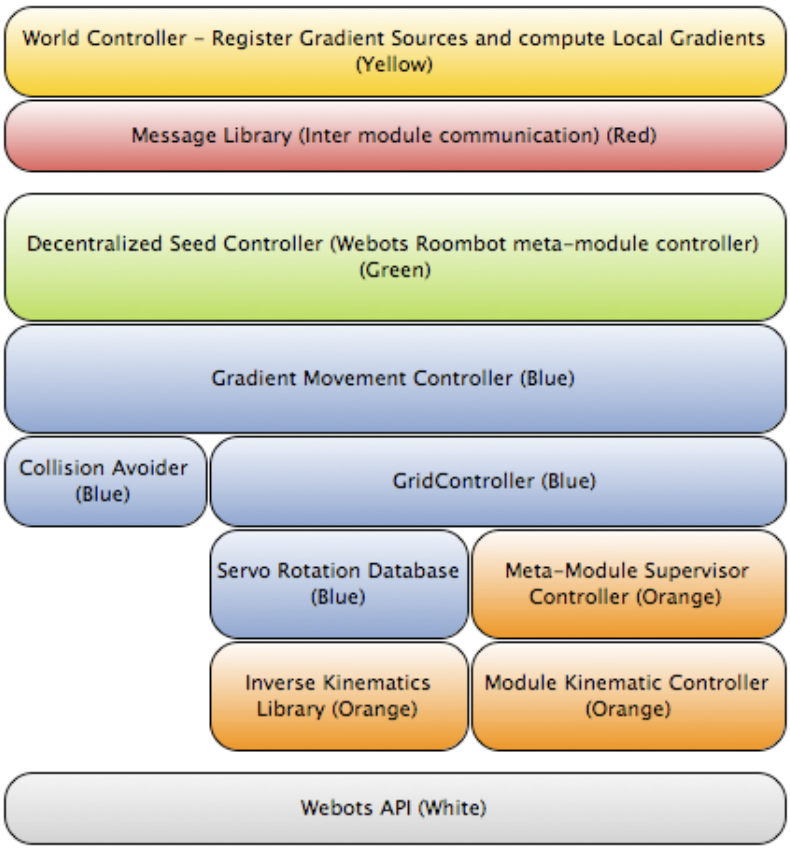


Figure 2.4: RDC Application Overview

closely with the webots API to achieve those goals.

The framework of Mayer 2009 [5] allows one to easily control a Roombot meta-module by providing the rotations of the 6 servo's of the meta-module. The rotations can be provided either by a user on a terminal or by a larger framework. On top of this framework lies the decentralized controller. The blue modules are essentially responsible for locomotion and moving the Roombot meta-module into a seed position. The green seed controller contains the seeding logic of the application and the red message library is a useful library built by Mayer 2009 [5] that allows multiple webots controllers to communicate with each other. Finally, the yellow world controller is a separate centralized controller with which all decentralized seed controllers communicate to obtain information on their immediate surroundings and the direction of the gradient vector. In a real running application, there will be multiple decentralized seed controllers running concurrently, communicating with each other and with one world controller which has global knowledge of the content of the world and the whole gradient field. The main purpose of the world controller is to provide information to the decentralized controllers that they could also obtain using local proximity sensors. Even though the world controller is a centralized piece of the application, it does not fundamentally break the decentralized spirit of the whole controller.

2.3.1 Meta-module supervisor controller

The Roombot Decentralized Controller will interface directly with the meta-module supervisor which is designed to handle one Roombot meta-modules (4 cubes). The meta-module supervisor is designed to accept and execute the following instruction formats:

1. **movefw** $s1_1$ $m1_1$ $s2_1$ $s1_2$ $m1_2$ $s2_2$
2. **movebw** $s1_1$ $m1_1$ $s2_1$ $s1_2$ $m1_2$ $s2_2$
3. **lock** moduleNumber connectorID
4. **unlock** moduleNumber connectorID

5. **step** time
6. **reachfw** D_x D_y D_z dir_x dir_y dir_z Orientation
7. **reachbw** D_x D_y D_z dir_x dir_y dir_z Orientation
8. **dropBoxFw**
9. **dropBoxBw**
10. **cmd**

The **move** instructions (1 and 2) are used to specify absolute servo rotations that the Roombot meta-module should perform. The **lock** and **unlock** instructions (3 and 4) are used to lock and unlock specific connectors of the Roombot roombot modules. The **step** instruction (5) essentially asks the meta-module to wait for a specified amount of time. The **reach** instructions (6 and 7) are used to reach for specific positions in space and the corresponding servo rotations are solved using the inverse kinematic model of the Roombot modules. The **dropBox** instructions (8 and 9) draws a box around the Roombot modules if the physics plugin is enable. Finally, the **cmd** instruction (10) allows a user to enter all the previous instructions manually from a terminal. Otherwise, the instructions are fetched from a file.

Upon every instruction execution, the meta-module supervisor will gather the necessary information from the parameters and instruct the two underlaying module controllers to execute the corresponding actions. Initially, this controller would look for instructions inside a file. Now, as it is part of a larger framework, it will receive instructions from the decentralized Roombot controller. For our implementation of a decentralized framework, we will essentially use the move and lock instructions from the meta-module supervisor.

2.3.2 Decentralized Seed Controller

The main goal of the decentralized seed controller is to supervise and control the building steps of a Roombot structure. It is made of a gradient movement controller, a collision avoider, a grid controller and a servo rotation database. It uses the meta-module supervisor controller from Mayer 2009 [5] to resolve end effector positions and rotations as well as to rotate the Roombot meta-modules. The gradient movement controller is in charge of moving the Roombot meta-module in the direction of the gradient vector. It will try, at each step, to move the Roombot meta-module as close as possible to a seed position avoiding any collisions with itself and the environment in the process. The servo rotation database is essentially a database containing the servo rotations for specific configurations that the Roombots might want to assume. It is also aware of the positions that a Roombot meta-module will go through in space during a move, providing thus valuable information used to detect collisions in higher level modules. Finally, the GridController is a module implemented somewhat as a facade design pattern that forwards high level controller actions to lower level controllers abstracting any information on the absolute rotation of the Roombot meta-module in the process.

In the next section, we will present the purpose and implementation of every module in the Decentralized Roombot Controller in greater detail. This section will be useful to understand the implementation of the controller if one wishes to understand, replace or extend some of it's parts. We will present all the modules in a bottom-up approach as displayed in Figure 2.4 as we believe this is the best way to describe and understand the complete application.

2.4 Servo Rotation Database

2.4.1 Introduction

The Servo Rotation DataBase (SRDB) is part of the distributed self-reconfiguration framework for the Roombot modules. It is one of the building blocks of the whole framework as it stores and provides Roombot servo rotations to the rest of the application. A Roombot controller inside the distributed framework is responsible of controlling a chain of two Roombot modules (4 cubes) which contains 6 servos. Additionally, this Roombot meta-module always moves on a 3D grid using only two active connectors at each of it's extremities (connector C0X on the first Roombot module and connector C3X on the second Roombot module).

While moving, one of the two connectors on a Roombot meta-module is always connected. The cube of this connector is called the root of the meta-module as it supports the whole Roombot chain. The cube on the other extremity of the chain is called the goal cube as it will reach for a 3D grid position goal. As we can see in Figure 2.5, the goal cube reaches for a grid position relative to the root cube. The latter will always be at grid position $(0,0,0)$ and the goal cube will reach

for a 3D grid position with respect to the root cube.

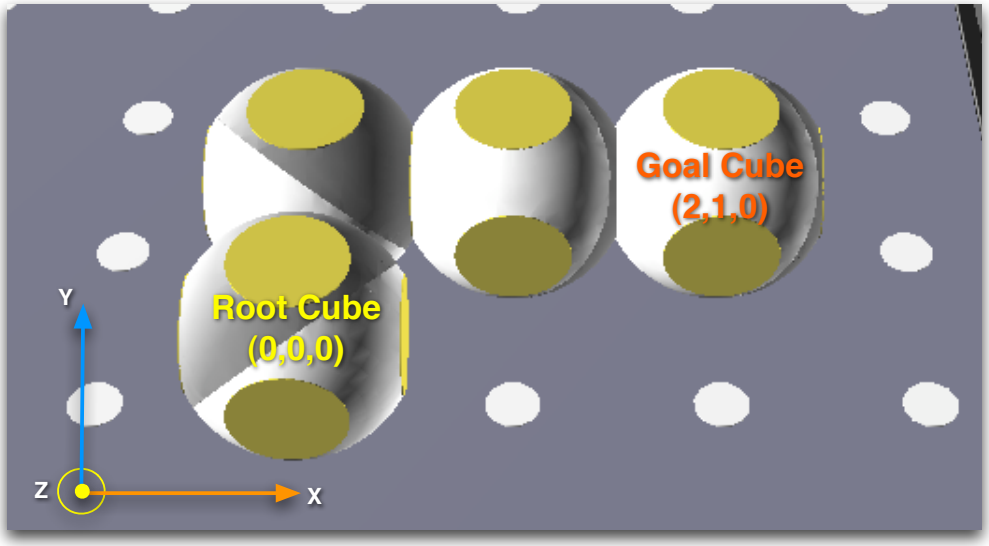


Figure 2.5: Roombot meta-module root and goal cubes

Considering the example above, a Roombot meta-module can move in the following way:

1. Make the goal cube reach a 3D grid position in the direction where we want to move
2. Lock the active connector of the goal cube on the ground or on a neighbor module
3. Unlock the active connector of the root cube
4. Switch roles between the goal cube and the root cube (root becomes goal and goal becomes root)
5. Go to step 1 and start over

It is important to notice that this methods works only if the goal cube connects correctly with another connector (either on the ground or on another fixed module) prior to disconnecting the root and switching roles. If the goal cube fails to connect properly, the whole meta-module will be completely lost as it will disconnect from the 3D grid. Another important thing to notice here is that only the first 4 servos of the chain are useful to bring the goal cube to an arbitrary grid position. Rotating the last two servos will not change the grid position of the goal cube. It will however change the position and orientation of the active connector on the goal cube. Thus, we will use the last two servos to orient the active connector of the goal cube so that it can connect correctly with a neighbor connector.

2.4.2 Motivation

Ideally, we would like to work exclusively with three dimensional grid position and not bother with the servo rotations. The SRDB provides exactly this service : it completely abstracts the servo rotations and allows higher level controllers to move the Roombot meta-module using only three-dimensional grid positions. The SRDB provides essentially a mapping between relative 3D grid positions and the corresponding servo rotations. It is much easier for higher level controllers to control a self-reconfiguration process in a three-dimensional grid position space than in a six-dimensional rotation space. It makes moving, short-range planning, collision detection and environment awareness much easier to implement.

The SRDB provides three distinct services:

1. Provide a mapping between relative 3D grid positions and corresponding servo rotations
2. Provide a mapping between a Roombot configuration and corresponding servo rotations
3. Provide a cloud (a set of 3D grid positions) resulting from the movement between two Roombot configurations.

Assuming a specific configuration in 3D space means that all the 4 cubes of a Roombot meta-modules need to be in a specific 3D grid position (instead of just the last goal cube). Assuming a specific configuration is important once a Roombot meta-module has reached the seed positions and has to position itself according to the plan of the structure it is helping to build. On the other hand, reaching for a relative 3D grid position is useful when the seed position is not yet reached and the Roombot meta-module is moving towards it. Finally, the cloud provides valuable information to any higher level controller that needs to know what 3D grid positions the Roombot meta-module will go through while moving between two configuration. This information can be used to detect collisions with the environment or with other Roombot meta-modules for instance. Figure 2.6 is an example of a grid position cloud. The initial configuration 1 is in blue and the final configuration 2 is in red. In order to move from configuration 1 to configuration 2, the Roombot will have to go through all the green grid positions. These green grid positions are what we call the cloud of grid points separating configuration 1 from configuration 2.

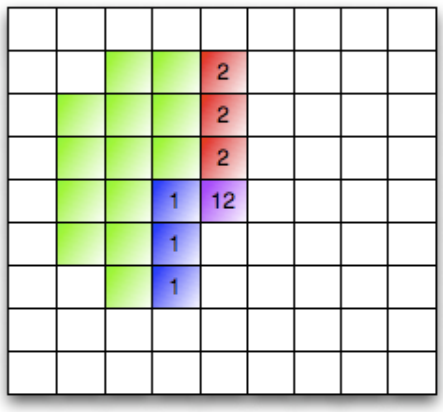


Figure 2.6: Cloud of grid positions separating two configurations. (Top view of the grid)

The mapping between grid positions and servo rotations is implemented as a set of C++ classes which are instantiated in the memory heap. The term database is merely used to describe the functionality of this set of classes. There is no conventional database implemented (such as oracle or mysql) as it would be an overkill considering the number of servo rotations to store and the desired response time.

Building the database requires going through all the possible servo rotations that would effectively put the 4 cubes of the Roombot meta-module on 3D grid positions. There are 3 possible rotations for servos 1,3,4 and 6 (0, $2\pi/3$, $4\pi/3$) and 4 possible rotations for servos 2 and 5 (0, $\pi/2$, π, $3\pi/2$). Additionally, the two modules making a Roombot meta-module can be connected in 4 different ways as seen in Figure 1.10 in the previous chapter.

An exhaustive search through all servo rotations and connections results in visiting 5184 configurations. Knowing the complexity of the task, a choice had to be made wether to run the complete search each time we wanted to reach a new grid position or to pre-compute all rotations and store them in special mappings. By running the search online each time we want to reach a new grid position, we sacrifice some computation time but spare some memory on the heap. By pre-computing the servo rotations, we eat up some memory on the heap but reduce the time to find a rotation as they are stored in special maps. The tradeoff is neglectable on a traditional computer but if the algorithm needs to be implemented on an embedded device, benchmarks would need to be performed in order to find the best possible implementation. We have decided to write the algorithm as a database with mappings as it provides a more elegant implementation and computation time is usually considered to be a more critical resource than memory on traditional computers. The cloud computation on the other hand was decided to be run online. See the last chapter for results on the performance of these various procedures.

2.4.3 Class Diagram

The UML class diagram implementation of the SRDB is presented on Figure 2.7. The information on the servo rotations is stored inside a RoombotConfiguration class. This class represents a configuration and contains the positions of the roombot cubes, the servo rotations and the orientation of the active connector on the goal cube. There are only 3 cube positions stored because the root cube position is always $(0,0,0)$ and thus doesn't need to be stored. The orientation of

the active connector specifies in which direction lies the active connector of the goal cube of this particular configuration. It could be in the direction *PLUSX* or *MINUSZ* for example. This information is useful when we want to connect the goal cube with a neighbor cube that lies in a specific direction.

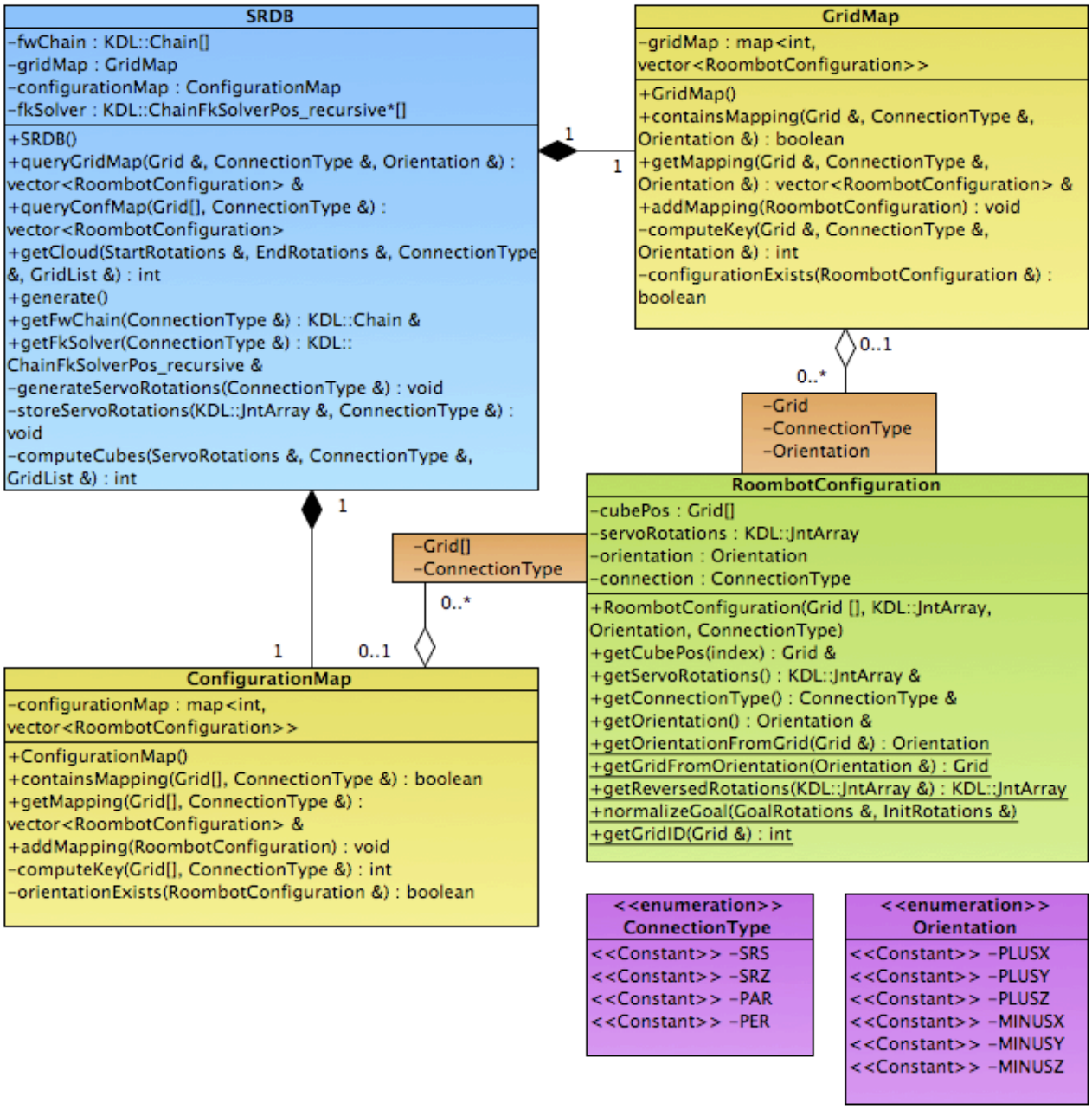


Figure 2.7: SRDB Class Diagram

While going through all the possible servo rotations and connection types, the SRDB will generate these RoombotConfiguration objects, store them on the heap and include references to them in two special mappings: a GridMap and a ConfigurationMap. The GridMap will return a list of servo rotations that correspond to a specific goal grid position, a connection type and an orientation for the active connector of the goal cube. This mapping is used by the GradientController when it moves the Roombot meta-module in the direction of the gradient vector. The GradientController will find a grid position in the direction of the gradient vector that has a neighbor grid that it can lock on. It will thus request the servo rotations corresponding to specific grid positions with the goal cube's active connector orientation in the direction of the neighbor connector.

The ConfigurationMap will be used by the GradientController when the Roombot will assume it's last configuration in the structure that is being built. When the Roombot meta-module needs to take a position inside the structure, it must not only reach for a specific grid position but all the cubes of the meta-module must be in a specific position (for example, when building the legs of a chair, the whole Roombot meta-module needs to be in an upright position). The GradientController can obtain the servo rotations corresponding to a specific configuration by querying the ConfigurationMap. Given the 3 positions of the Roombot's cubes and the connection type between the two Roombot modules, the ConfigurationMap will provide a list of RoombotConfigurations corresponding to the desired shape.

2.4.4 Procedure Implementation

The public interface of the SRDB is provided by the SRDB class itself and contains the following procedures:

1. generate( ) : void
2. queryGridMap(Grid&, ConnectionType&, Orientation&) : vector<RoombotConfiguration>&
3. queryConfMap(Grid[], ConnectionType&) : vector<RoombotConfiguration>&
4. getFwChain(ConnectionType&) : KDL::Chain&
5. getCloud(StartRotation&, EndRotation&, ConnectionType&, GridList&) : int

The generate method will fill the database and the mappings with RoombotConfigurations. The generation of the database is implemented there instead of the constructor to give the user better control over the time when the population of the database will be done as it might be time consuming. Depending on the platform where the code will run, this might be useful. The getFwChain returns the Roombot kinematic chain (in KDL [21] format) for a particular connection type between the two modules. It is used essentially by the GridController to compute the rotation of the Roombot meta-module. The queryGridMap and queryConfMap simply return lists of RoombotConfigurations corresponding to the specific requests as described previously. Finally, the getCloud method returns the cloud of 3D grid positions that a Roombot meta-module passes through when moving from a start to an end rotation. As mentioned previously, this information is used by the GradientController to detect collisions with the environment and other meta-modules before attempting to move.

Database Generation

The database is generated using the Kinematics Dynamic Library (KDL) [21]. For an initial position of the root cube, all the servo rotations are simulated and the position of all the cubes and the goal connector are stored inside a RoombotConfiguration object. Two RoombotConfiguration objects are considered identical (and thus are not duplicated in the mappings) if they have the same cube positions, the same connection type and the same goal connector orientation. It might happen that different servo rotations produce the same cube positions and goal connector orientations for a specific connection type. All these configurations are considered identical and only 1 per mapping will be stored. Additionally, the database is computed for an initial position of the root cube. In reality, the Roombot meta-module will be arbitrarily rotated and to obtain the good mappings from the database, the grid position queries should be rotated appropriately. This process will be described in greater detail in the GridController module. Figure 2.8 contains the activity diagram describing the generation process of the SRDB database.

Add mapping to the GridMap

The last two steps of Figure 2.8 need more explanations. During the Add Grid Mapping action, the SRDB will call the addMapping method on the GridMap object. It will store the RoombotConfiguration object inside the mapping if it doesn't exist already (i.e. another RoombotConfiguration object having the same 3 cube positions, the same connection type and the same connector orientation). The RoombotConfiguration object will be mapped according to it's key which is computed using the connection type, the connector orientation and the position of the goal cube. There will be multiple RoombotConfiguration objects that will map to the same key (they will differ by the position of the first 2 cubes) and will be stored inside a list. Figure 2.9 contains the activity diagram of the addMapping method of the GridMap class.

Even though the operations of the ConfigurationMap are not explained in detail here, they are very similar to the operations of the GridMap. The key in the ConfigurationMap is computed using the connection type between two Roombot modules and the position of the three cubes in the configuration.

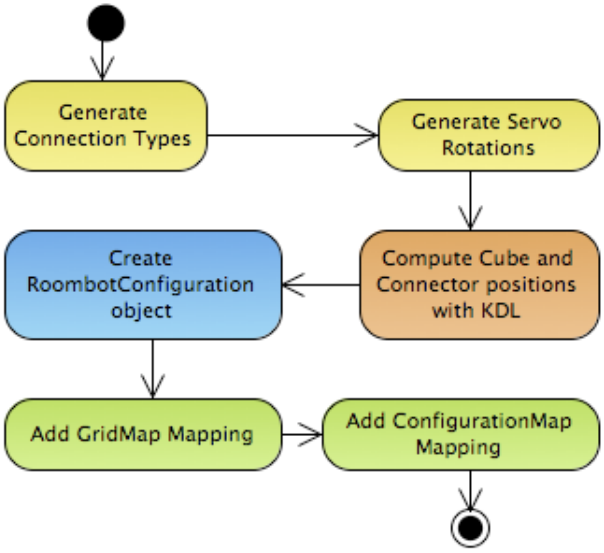


Figure 2.8: Database generation activity diagram

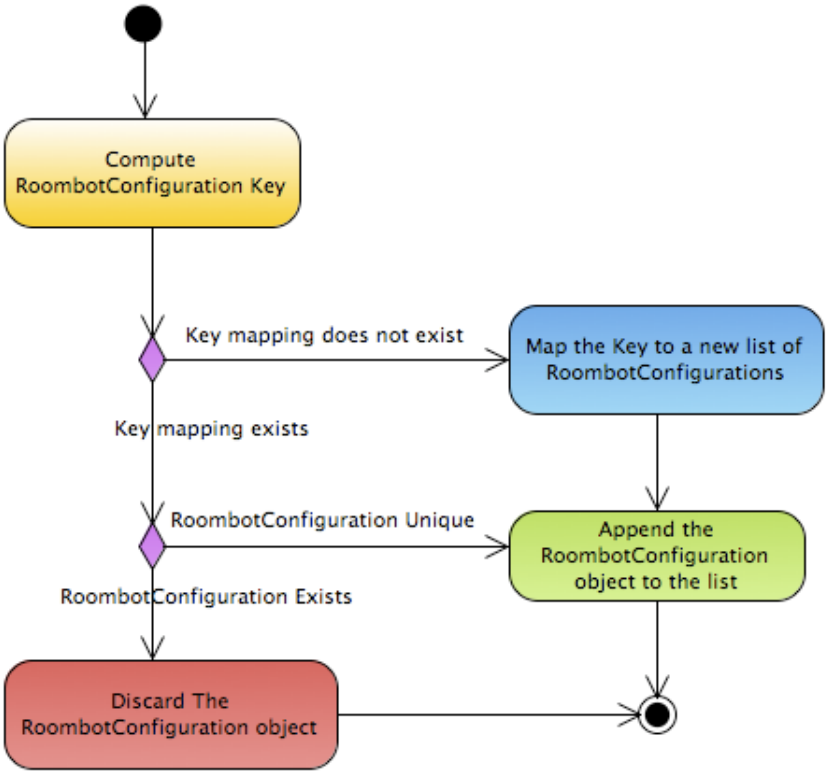


Figure 2.9: Adding a mapping to the GridMap

Grid Cloud between two configurations

The getCloud procedure allows one to obtain all the grid positions a Roombot meta-module has to go through when moving from a configuration to another. In order to do so, this procedure will simulate the motion by gradually incrementing the rotations of all six Roombot servos to go from the start configuration to the end configuration. At each rotation step, it computes the list of 3D grid positions that the Roombot meta-module occupies and adds them to the result list. The number of steps as well as the accuracy of the computation can be controlled through global variables available in the code. Figure 2.10 depicts the overall functions of the getCloud procedure.

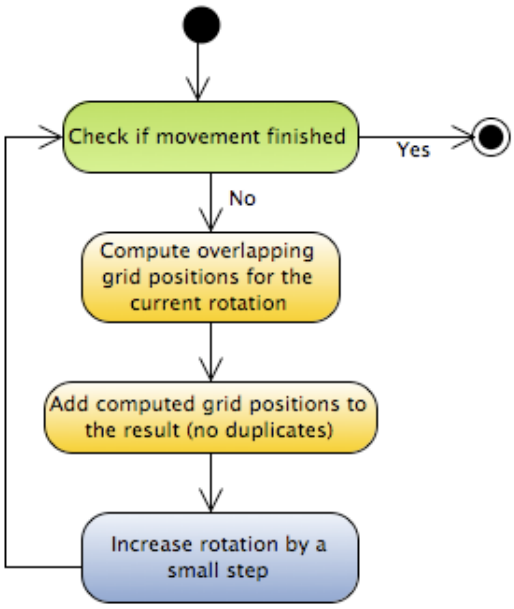


Figure 2.10: Computing the grid cloud between two configurations

2.4.5 Conclusion

The Servo Rotation Database is one of the key building blocks of the whole distributed Roombot self-reconfiguration framework. It assumes that we are working inside a 3D grid world and allows us to move Roombot meta-modules very efficiently inside this grid. Using this database, we can ask a Roombot meta-module to reach for any relative 3D grid position around it's root which is very helpful if we want to move in a particular direction. Additionally, we can also ask our Roombot meta-module to assume a specific configuration inside the 3D grid, effectively giving us complete control over the shape of the Roombot structure being built. The database is also aware of the grid positions a Roombot meta-module goes through when moving from a configuration to another. This information is very helpful if one wants to check for collisions with the environment or other Roombot meta-modules.

2.5 Grid Controller

2.5.1 Introduction

The SRDB database seen previously provides us with a mapping between three dimensional grid positions and servo rotations. This database is however computed by positioning a Roombot meta-module in an initial identity rotation. Thus, all relative grid positions have been computed relatively to this initial orientation. In simulation, the Roombot meta-module will however move and potentially be rotated by an arbitrary angle in an arbitrary direction. If we want our Roombot meta-module to reach for a relative Grid position in this case by querying the SRDB, we must take into consideration the relative rotation of the Roombot meta-module.

Figure 2.11 shows the problem of taking into account the relative rotation in greater detail. The identity rotation, the rotation with which the database made all it's computation, is represented as the blue configuration (Grid positions $(3,3),(3,4),(3,5),(3,6)$). The red configuration is the real position of the Roombot meta-module in space (Grid positions $(3,3),(2,3),(1,3),(0,3)$). It is rotated with respect to the identity rotation in blue. When querying the database, we will always obtain a position relative to the blue configuration. Suppose that our Roombot meta-module would like to reach the grid position $(2,1)$ and queries directly the database for that grid position. By looking where the grid position resides with respect to the blue configuration, we can safely assume that our Roombot meta-module will not end up where it wants to be but in grid position $(5,2)$. If we want our meta-module to end up in grid position $(2,1)$, we have to query for grid position $(1,4)$. Grid position $(1,4)$ is exactly at the same position relatively to the blue configuration than grid position $(2,1)$ relatively to the red configuration.

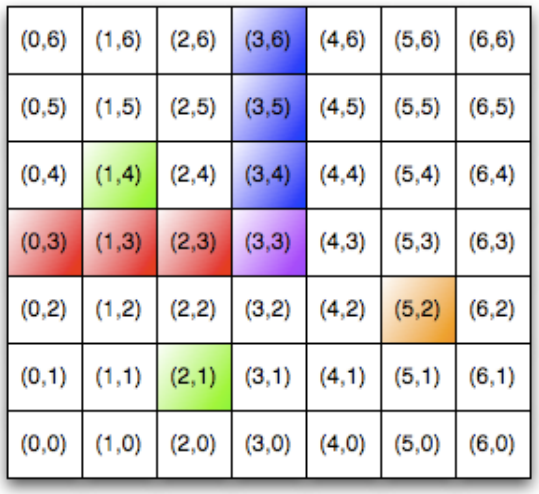


Figure 2.11: Rotating the query to match module rotation in world coordinates (Top view of the grid)

2.5.2 Motivation

Taking into consideration the rotation of the Roombot meta-module each time we query the SRDB and computing the necessary transformations can be quite cumbersome. We would like to completely mask the fact that Roombots can be rotated to higher level controllers. This is where the GridController comes in. The main goal of the GridController is to abstract all implementation details regarding the rotation of the meta-modules. Higher level controllers will query the GridController instead of directly querying the SRDB to get the required configurations. Higher level controllers will be able to move the Roombot meta-module only through 3D grid positions. The GridController is a stateful controller that will remember the rotation of the meta-module and transform the queries of higher level controllers correctly by taking into consideration this rotation.

2.5.3 Class Diagram

The GridController is implemented as a facade around the SRDB and essentially provides a rotation-abstraction service to higher level controllers. Figure 2.12 contains the class diagram of the GridController. The GridController also works closely together with the Kinematic Supervisor who directly controls the servo rotations and connector locks on the Roombot meta-module.

2.5.4 Procedure Implementation

The public interface of the GridController is essentially made of these methods:

1. queryGridMap(Grid &, Orientation &) : vector<RoombotConfiguration>
2. queryConfigurationMap(Grid[]) : vector<RoombotConfiguration>
3. assumeConfiguration(RoombotConfiguration &, InitRotation &) : int
4. getCloud(StartRotation &, EndRotation &, GridList &) : int
5. getCurrentRotations() : ServoRotations
6. connectorSwap() : void
7. connectAll() : void

The getCloud procedure allows a controller to get the cloud of 3D grid positions a Roombot meta-module passes through when moving from configuration A to configuration B. It essentially forwards the call to the SRDB and rotates the resulting cloud to match the rotation of the calling meta-module, abstracting any notion of meta-module rotation in the process. getCurrentRotations simply return all the six rotations of a Roombot meta-modules servos. connectorSwap effectively swaps the locks on the connectors of the root cube and the goal cube. This procedure is used when moving the

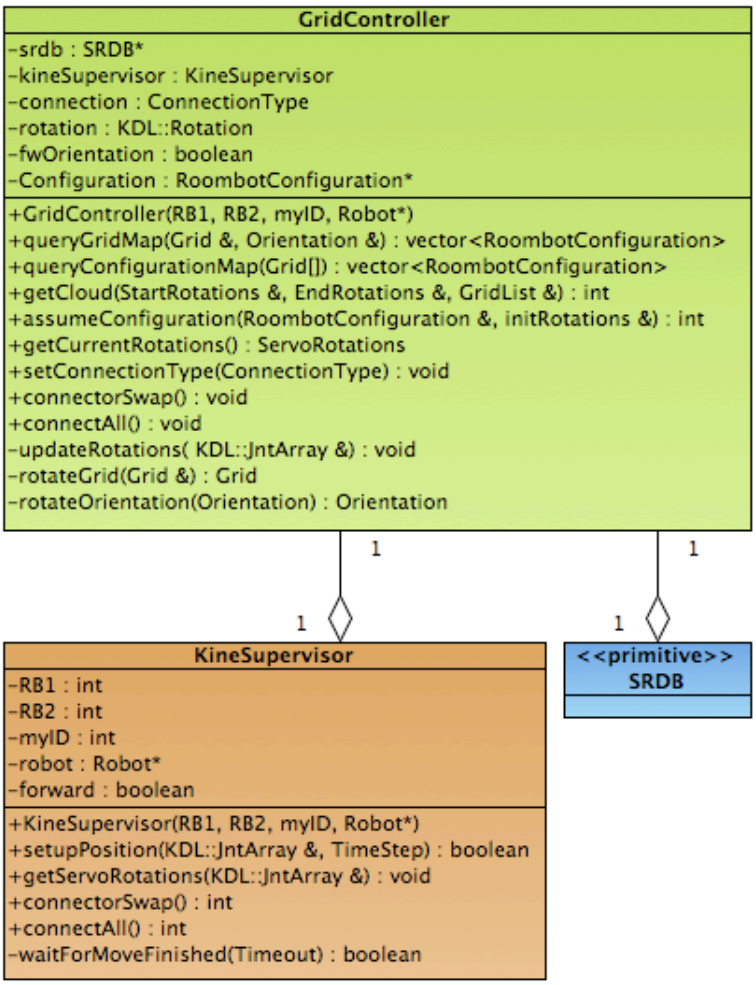


Figure 2.12: Grid Controller class diagram

Roombot meta-module. connectAll on the other hand makes all the connectors on a Roombot meta-module active and connected. This procedure is used once the Roombot meta-module enters a seeding position inside a Roombot structure and becomes a passive element on which other Roombot meta-modules can grip on.

Querying the Grid Map

The queryGridMap method allows a high level controller to get a list of RoombotConfigurations (containing servo rotations) for a given absolute 3D grid position to reach. Only absolute grid positions need to be given to this method. The high level controller calling this method does not need to know the absolute orientation of the Roombot meta-module. All the data contained in the RoombotConfiguration objects received (cube grid positions and connector orientation) will be transformed to match the rotation of the caller meta-module. Complete abstraction of the caller's rotation is thus achieved and all the data both in input and output is transformed to match absolute 3D grid positions. This makes the implementation of high level controllers easier.

The queryConfigurationMap allows a high level controller to get a list of RoombotConfiguration objects for a specific configuration to assume. Just as with the queryGridMap method, only absolute 3D cube positions need to be used for this method. The GridController takes care of transforming the data with respect to the absolute rotation of the Roombot meta-module. In Figure 2.13, you can see the activity diagram of the queryGridMap method. The queryConfigurationMap has a very similar diagram.

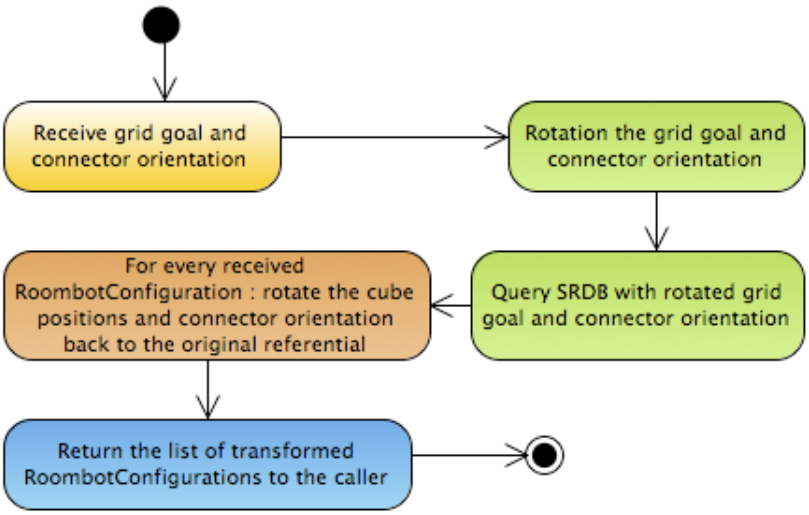


Figure 2.13: Grid Controller : querying the GridMap

Assuming a Configuration

The querying methods are used to receive lists of RoombotConfiguration objects. Once higher level controllers have chosen one of those configurations, it can call the assumeConfiguration procedure on the GridController to move the Roombot meta-modules into the appropriate shape. Once the configuration has been assumed successfully, the GridController will remember the new configuration so that it can update the rotation of the meta-module appropriately when a connector swap is ordered. If the Roombot meta-module fails to assume the configuration due to self-collision or collision with the environment, the procedure will return a non-zero integer flag indicating the nature of the error. Figure 2.14 contains the activity diagram of the assumeConfiguration procedure. It is interesting to note that a controller can call this procedure multiple times with different configurations before it decides to swap the connectors. This might be useful to avoid some obstacles in the immediate surroundings of the meta-module. The normalize goal rotation step noramlizes the goal rotation so that the angle between the initial rotation and the goal rotation is as small as possible.

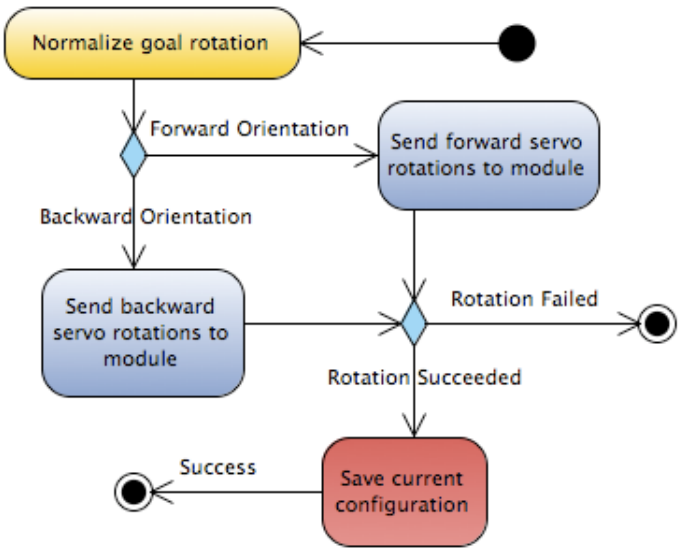


Figure 2.14: Grid Controller : assuming a configuration

2.5.5 Conclusion

The main purpose of the GridController is to provide more freedom to higher level controllers by abstracting the fact that the Roombot meta-modules can be rotated, upside down or laying on the side. The GridController allows high level controllers to move the Roombot meta-module on an absolute grid position relative to the root cube, no matter how the root cube is oriented. We effectively concentrate all the code concerning rotations in this controller such that it does not appear in the SRDB or in higher level controllers. The GridController is another big step making the self-reconfiguration process much easier for higher level controllers. At this point, with the SRDB, the Kinematic Supervisor and the Gradient Controller, we effectively have the means to easily move a Roombot meta-module simply by providing absolute grid positions.

2.6 Gradient Controller

2.6.1 Introduction

Now that we have the means to easily move a Roombot meta-module inside a three-dimensional grid world, we have to implement a set of distributed rules that will allow multiple meta-modules to effectively build a structure through cooperation. One of the challenges we have to face is moving a Roombot meta-module into a seed position while avoiding collisions with the environment and with other meta-modules. Essentially, this is exactly what the Gradient Controller does when launched. It will query the World Controller for information on the world in it's immediate surroundings and the direction of the gradient vector. It will then go down the gradient field until it reaches a seed position, avoiding other modules and collisions with the environment in the process. Once it has reached a seeding position, the Gradient Controller assumes the seed configuration requested to integrate itself in the structure and hands back control of the Roombot to the calling procedure. The Gradient Controller is undoubtedly the largest module at the heart of the distributed self-reconfiguration framework.

2.6.2 Motivation

As we are building a decentralized self-reconfiguration controller, we had to make multiple choices as to what the Roombot meta-module knows and what it doesn't know. First of all, a meta-module can always see the content of the world 3 grid positions ahead in every direction relatively to the root cube (the cube of the meta-module which is locked). This information can also be visualized as a 3D cube, 7 grid positions long centered around the root cube of the meta-module. The controller thus has to update and maintain 343 different grid positions (7^3). We have chosen 3 grid positions in every direction because it is the furthest away a Roombot meta-module can reach when it is fully extended. The Roombot can distinguish plain obstacles, connectable surfaces, seed positions and other meta-modules. Based on this local information of the world and the direction of the gradient vector, the GradientController will make some decision as to how it will move.

The gradient field can be imagined somewhat like an electric field where the seed positions correspond to sinks (positive charges) and the Roombot meta-modules to sources (negative charges). The Roombots do not have access to the gradient field by themselves as they have only a limited knowledge of the world. They ask the World Controller to provide them the value of the gradient field at their position in the 3D grid world. In a real Roombot implementation, we could imagine the Roombots obtaining this information using electronic compasses or through other similar sensing capabilities.

In order for the World Controller to have a complete and correct representation of the whole world, all the individual Roombot meta-modules will send their configurations to the World Controller every time they make a move. Additionally, the individual Roombot meta-modules do not have any knowledge as to what kind of structure they are actually building. All they need to do is fill in the empty seed positions. The World Controller will control the registration / unregistration of the seed positions as the structure grows to ensure the correct building sequence of the Roombot structure.

2.6.3 Class Diagram

The GradientController module is implemented as two classes : the GradientController class and the RoombotWorld class. The main application logic is written directly in the GradientController class. The RoombotWorld class contains the knowledge of a Roombot meta-module's immediate surrounding (a cube with a length of 7 grid positions) as well as many utility methods such as synchronizing the local world with the World Controller. Figure 2.15 contains the class

diagram of the GradientController module.

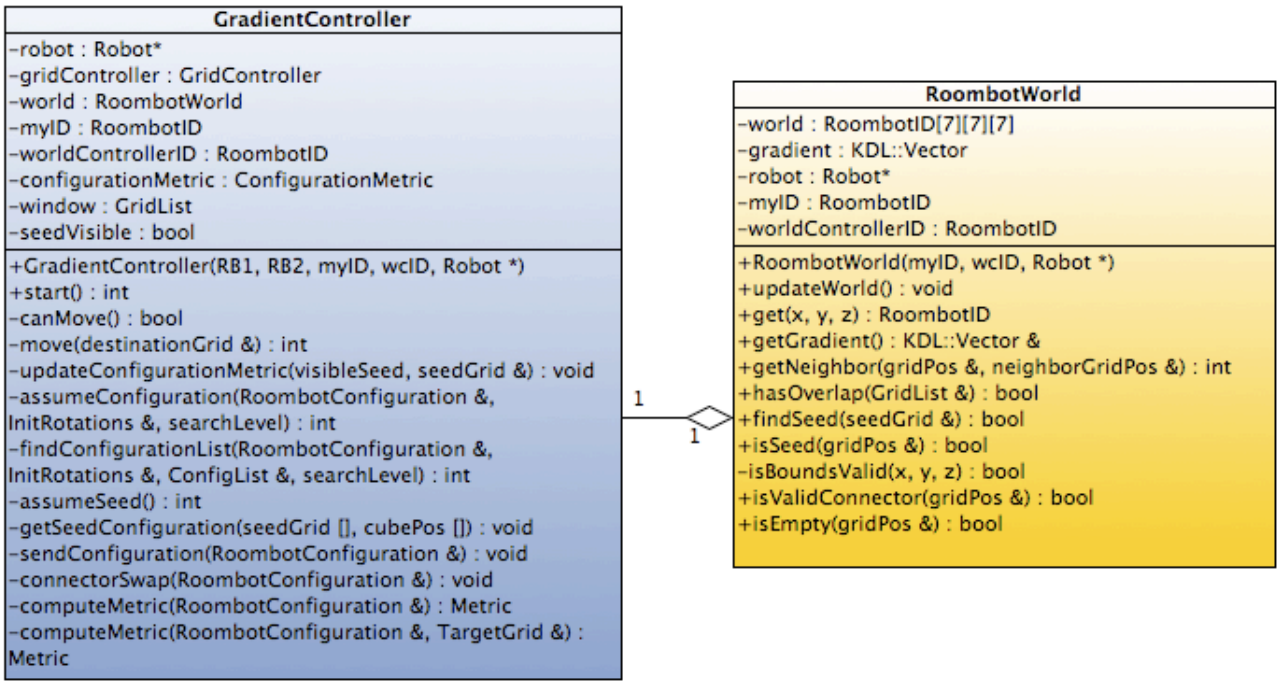


Figure 2.15: Gradient Controller class diagram

2.6.4 Procedure Implementation

Main Control Loop

The only procedure available as part of the public interface of the Gradient Controller is the start procedure. This procedure contains the main control loop of the Gradient Controller. Once it is called, it will take over the controls of a Roombot meta-module and make all necessary decisions and actions to bring that meta-module into a seed position. Once the seed position and configuration have been reached and the meta-module is part of the larger structure being built, the Gradient Controller releases control of the Roombot.

Figure 2.16 contains the activity diagram of the main control loop of the Gradient Controller. The Roombot meta-module starts by updating the information of the world in it's immediate surroundings by querying the World Controller. This will also update the value of the gradient vector at the Roombots position. Based on these new informations, it will make a certain number of decisions and moves. If the root cube of the meta-module is in a seed position, then the controller has achieved it's goal and it only needs to assume the final seed configuration before exiting.

If the meta-module is not in a seeding position, then it must try to reach one by moving in the direction of the gradient vector. First, it must check if it is allowed to move. This procedure will be explained in greater detail in it's own section later but the idea is to allow only one meta-module to move if they get too close to each other. As long as the meta-module is not allowed to move, it will wait a certain amount of time, update it's information on the world and try again. If it is allowed to move, it will try to move in the direction of the gradient vector by choosing a good 3D grid position. In order to do so, it will consider all the 3D grid positions that it can reach around itself and will sort them according to how close it will get the meta-module towards a seed position. If a seed position is visible in the immediate surroundings of the Roombot meta-module, then the 3D grid positions will be sorted according to how close they are to that seed. If no seed positions are visible in the immediate surroundings, then the 3D grid positions the furthest away in the direction of the gradient vector will be considered.

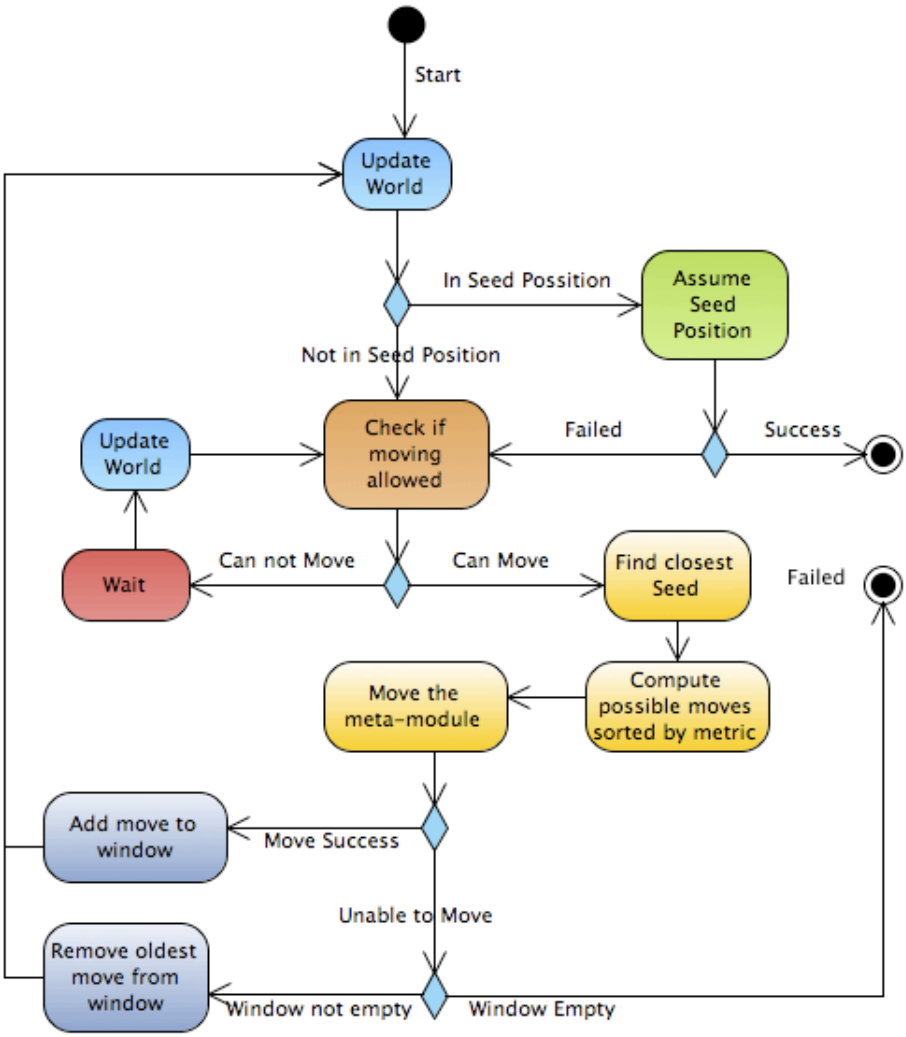


Figure 2.16: Main control loop of the Gradient Controller

Once a 3D grid position has been chosen according to these metrics, the meta-module controller will try to move the Roombot meta-module to that grid position. Recall from the SRDB section that the controller will try to move the goal cube of the meta-module into that grid position and lock on a neighbor grid. We will see in greater detail what decisions the controller makes during the move phase in it's own section later. Two outcomes are now possible : either the move succeeded or failed. Upon success, the 3D grid position newly reached is added into a history window of fixed length. The window remembers all the previously reached grid positions and prevents the Roombot from reaching those grid positions in the future. The idea is to prevent the Roombot from moving in circles if it reaches some kind of local minimum where the only way out is actually moving away from a seed position.

On the other hand, if the move fails, that means the meta-module tried every possible 3D grid position and could not reach any of them. In that case, the oldest grid position in the history window is removed, the world updated and the move is tried again. If the buffer window happens to be empty, then the Roombot meta-module is really stuck and there is nothing we can do about it. To controller stops and returns an error flag.

Roombot collision avoidance

This section handles the inter-Roombot collision avoidance. Recall from the previous section that every Roombot controller checks if it is allowed to move before doing any actions whatsoever. The goal of this procedure is to freeze the Roombots if they get clogged up together and allow only the closest Roombot to a seed position to actually move. Every Roombot makes this decision based on what it can see in it's immediate surroundings. Naturally, if a Roombot does not

detect another Roombot around itself, there can be no reason for it to be concerned with collisions and it will proceed to move. Figure 2.17 contains the activity diagram of this procedure. It is basically a decision tree that will indicate if a Roombot meta-module is allowed to move or not based on it's knowledge of the world.

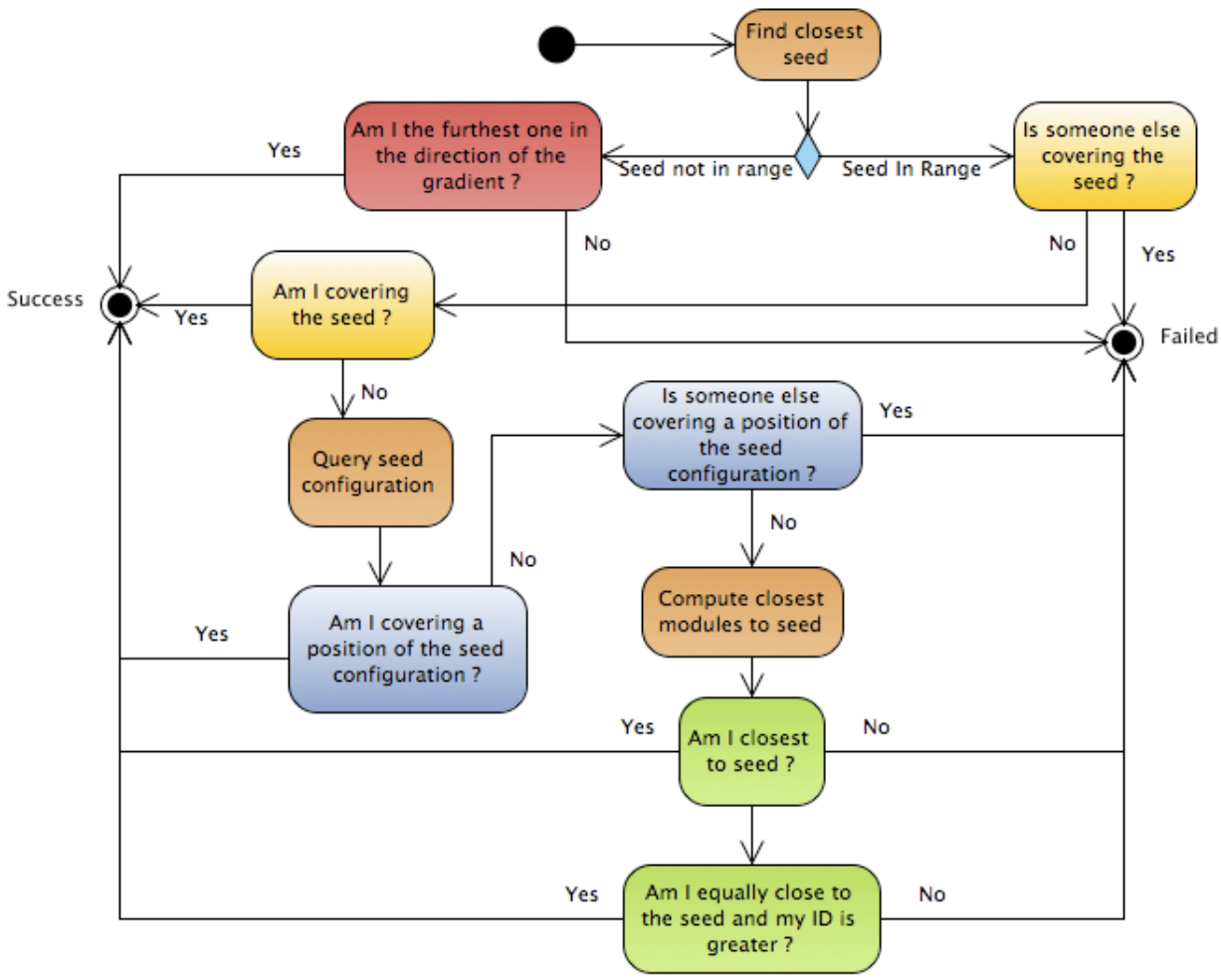


Figure 2.17: Is a Roombot meta-module allowed to move ?

If there is a seed position that yet needs to be filled in the immediate surroundings of the Roombot meta-modules, the closest Roombot to the seed will be allowed to move. If one of the Roombots is actually covering the seed or any grid position that is part of the final seed configuration, priority will be given to that Roombot as well because no other Roombot will successfully fulfill it's mission as long as someone covers the final seed configuration.

If no seed position is visible in the immediate surroundings of the Roombot meta-modules, priority will be given to the Roombot that is the furthest away in the direction of the gradient vector. Finally, if two or more modules happen to be equally close to the seed position or equally far away in the direction of the gradient vector, priority will be given to the module with the highest ID.

Updating the reachable grid position metrics

Once the Roombot meta-module has been cleared to move and before it actually moves, it needs to consider all the reachable grid positions and sort them according to a metric. Figure 2.18 contains the activity diagram of the procedure handling the update and sorting of the grid positions according to that metric. The metric is simply how close a 3D grid position is to a seed position if it is visible in the immediate surroundings of the Roombot meta-module or how far away it is in the direction of the gradient vector otherwise. The move procedure will try to move to the first grid position in this list and if that fails, it will try the second grid position and so on. From this list are removed all the grid positions that have

no neighbor grid position that can be connected on and all grid positions that are contained in the move history window.

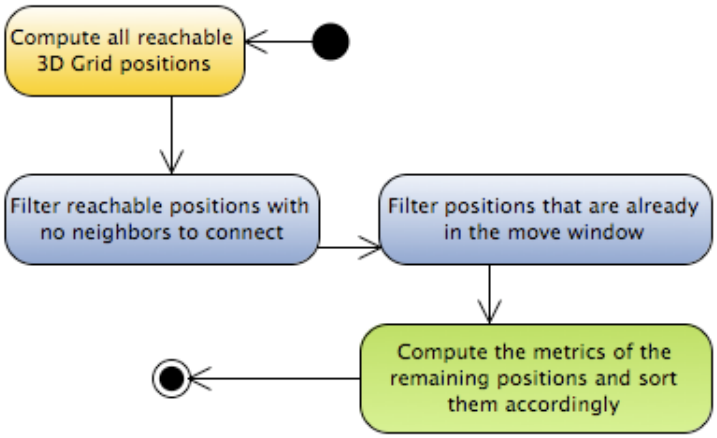


Figure 2.18: Updating the reachable grid position metrics

Moving the Roombot meta-module

Now the move procedure's purpose is to physically move the Roombot meta-module based on the sorted configuration list provided by the last step. Figure 2.19 contains the activity diagram of the move procedure. Basically, the procedure takes the first grid position in the sorted list, assumes the configuration that will get it's goal cube into that grid position and lock itself into that new position by swapping the connector locks on the root and goal cube. If the move fails, it will fetch the second grid position in the list and so on.

Before assuming the new configuration, the Roombot meta-module will be in a specific configuration, say configuration A. Let us name the final configuration that will bring the goal cube of the meta-module into the correct grid position configuration B. The meta-module in configuration A must somehow move to assume configuration B. Although moving directly from configuration A to B ($A \rightarrow B$) is sometimes possible, some obstacles may be in the way preventing this movement from executing successfully without collisions. In this case, we can try to find an intermediate configuration C such that the movement A to C to B ($A \rightarrow C \rightarrow B$) is possible without collisions. We call such a search for an intermediate configuration a level 2 search. We can also extend this search to a level 3 search by trying to find two intermediate configurations C and D such that the movement A to C to D to B ($A \rightarrow C \rightarrow D \rightarrow B$) is possible without collisions.

The level 2 search has a relatively small computation cost and can easily be afforded everywhere as it greatly increases the movement possibilities compared to a direct level 1 search. The level 3 seach however takes a few seconds to run and can only be afforded to run in very sensible situations. One of those situations is when the grid position where the Roombot meta-module tries to move is a seed position. In that case, it is worthwhile to try and reach the seed through a level 3 search.

Assume Configuration

Figure 2.20 contains the activity diagram of the assume configuration procedure. It's main task, as described previously, is to move the Roombot meta-module such that the goal cube of the meta-module is at the desired grid position. The first step of the procedure is to find a list of intermediate configurations that will allow the Roombot to move from the start configuration to the end configuration. If the movement from start to end is directly possible without collision, the list will be empty. If not, it will either contain one or two intermediate configurations depending on wether we asked for a level 2 or level 3 search. The controller then assumes all the intermediate configurations and the final configuration to reach the desired end position. Between each configuration change, the Roombot notifies the World Controller of it's new configuration, updates the content of the world and checks if it is still allowed to move. If not, the procedure returns the control of the Roombot to the caller who will handle this problem and proceed once it is allowed to move again.

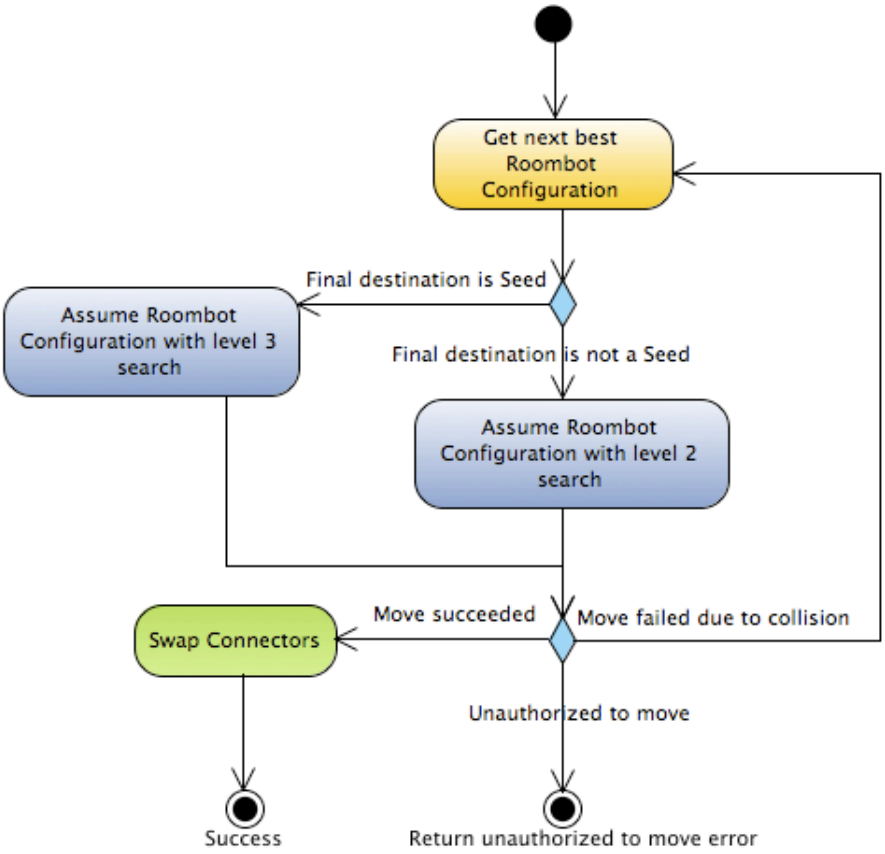


Figure 2.19: Moving the Roombot meta-module

Finding a list of good intermediate configurations

This procedure concerns the first bubble of Figure 2.20 and it's activity diagram is displayed in Figure 2.21. Given a start configuration A and a final configuration B, the goal of this procedure is to return an intermediate configuration C such that the movement $A \rightarrow C \rightarrow B$ is possible without collisions. Naturally, if the movement $A \rightarrow B$ is possible, the procedure will not return an intermediate configuration. Figure 2.21 shows a level 2 search for an intermediate configuration but a level 3 search is relatively similar.

First, the procedure checks wether the movement $A \rightarrow B$ is possible by computing the grid cloud between these two configurations (see SRDB section for more details on the grid cloud). If no collisions are detected between the grid cloud and the environment, the procedure returns without an intermediate configuration. Else, it will generate all possible intermediate configurations C and return one as soon as no collision is detected with the grid cloud from A to C and from C to B. If no intermediate configuration provided a collision free movement, the procedure returns an error flag.

Assuming a seed configuration

Once a Roombot meta-module journeyed it's way into a seed position, all it has to do before switching off is to assume the final seed configuration corresponding to it's position inside the Roombot structure. Figure 2.22 contains the activity diagram of this final step. First, the Roombot queries the World Controller for the configuration that it has to assume which depends on where it is inside the structure. It will then query the GridController for a RoombotConfiguration that matches the seed configuration received from the World Controller. If no RoombotConfiguration exists for that seed configuration, that means it is physically impossible for the Roombot, given it's current position and servo rotations, to assume the configuration. In that case it will return a failed error flag and the Roombot will go to another grid position and try again.

If a RoombotConfiguration object exists, that means that it is possible for the meta-module to assume that final seed

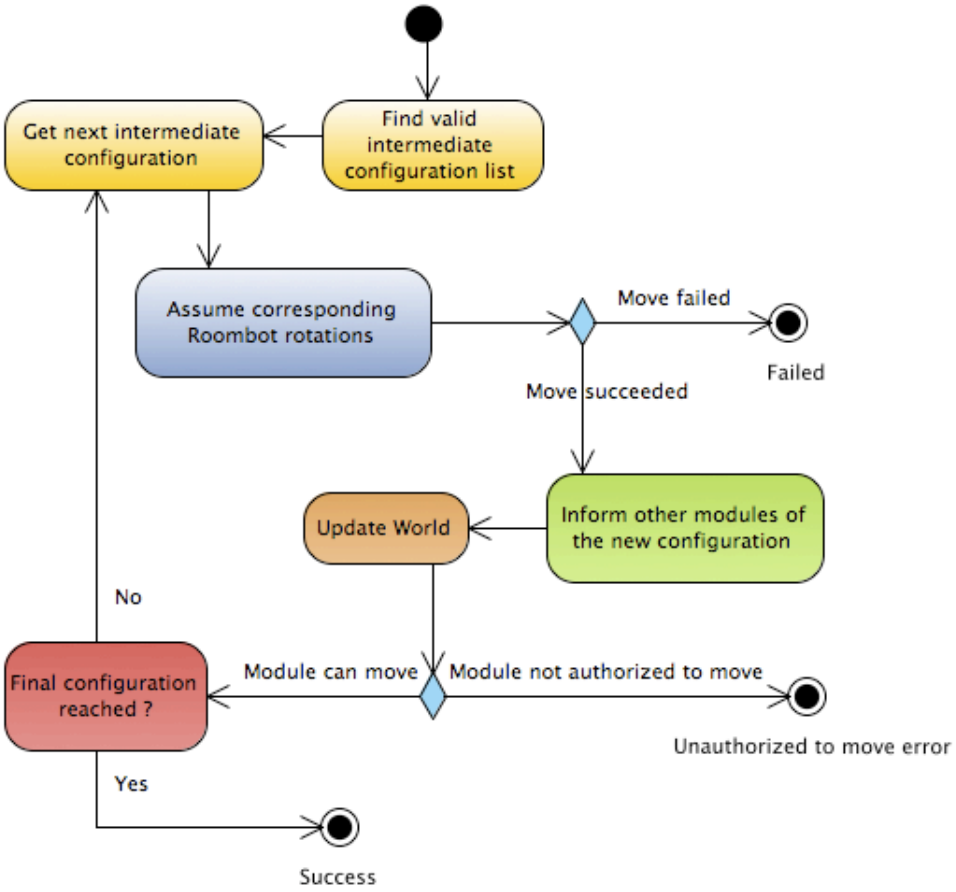


Figure 2.20: Assuming a configuration

configuration if there is no collision with the external environment. The controller will try to assume that final seed configuration by ordering a level 3 search for intermediate configurations to give the Roombot a good chance of avoiding any obstacles in it's way. Even though the level 3 search requires a few more seconds than the level 2 search, it is appropriate to use it here as it would effectively merge the Roombot inside the structure. If that step fails, the Roombot will have to try all over again by moving to some new grid position first.

If the move succeeded, the Roombot will ask the World Controller to unregister the seed position that it is now filling and do the appropriate changes to the gradient field. The Roombot will also enter a passive state and make all it's connectors active and available for other Roombot meta-modules to connect on. At this point, the GradientController module has achieved it's goal and returns the control of the Roombot meta-module to it's caller.

2.6.5 Conclusion

The GradientController module contains most of the logic and rules to ensure a successful distributed self-reconfiguration process. It is a key player in the success of the whole framework and undoubtedly one of the biggest parts of the puzzle. It makes all the movement decision that will bring the Roombot meta-module into a seed position, avoids collisions with the environment and with other Roombot meta-modules, assumes the final seed configuration and handles all the communications and updates with the World Controller. The activity diagrams of the key procedures of the Gradient Controller have been displayed and described in this section.

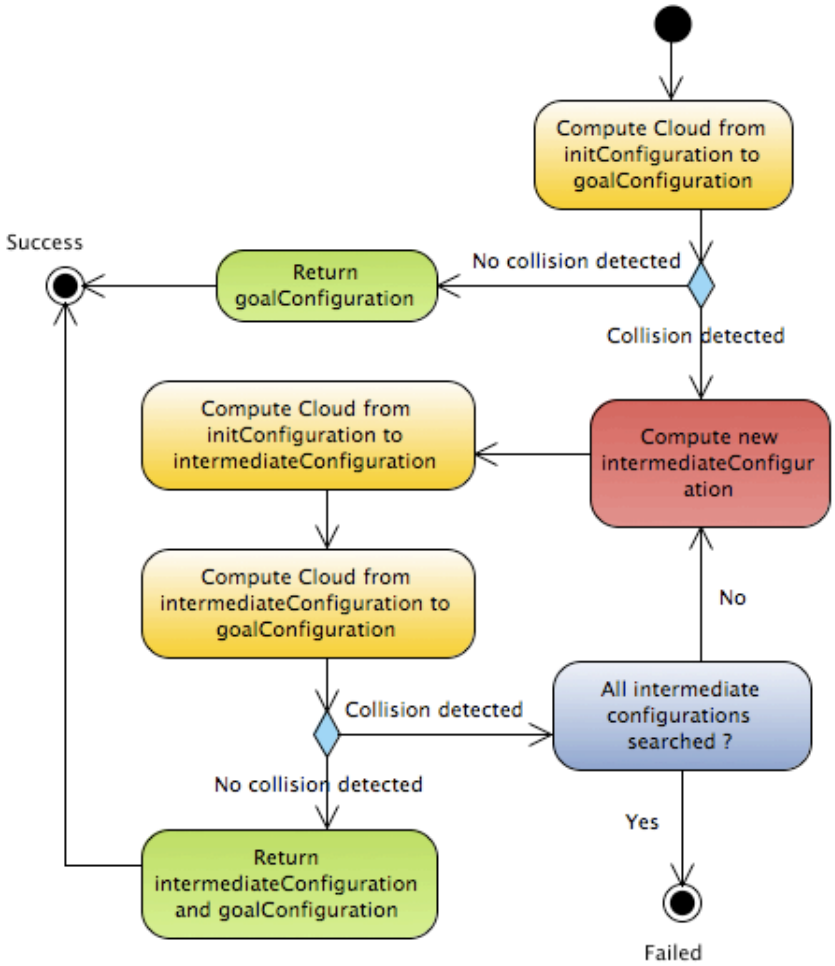


Figure 2.21: Finding a list of intermediate configurations through a level 2 search

2.7 World Controller

2.7.1 Introduction

The World Controller is the final piece of the whole distributed self-reconfiguration framework. It is a controller who's main task is to assist the individual decentralized Roombot controllers by providing them information on the content of the world around them and the gradient field. There is no reconfiguration logic implemented into this controller.

The World Controller keeps in memory the state of the complete world in which the Roombots evolve. It knows where those Roombots are, what structure the Roombots are building and where the seed positions reside. This controller was not designed to solve the self-reconfiguration problem even though it possesses a lot of information to do so as this would result in a centralized solution. The goal of this controller is simply to provide localized information to the individual Roombot meta-modules that could have otherwise been obtained through local sensing. The controller thus doesn't require a lot of computation power to operate.

2.7.2 Motivation

Individual Roombot controllers (Gradient Controllers) contain all the decentralized logic for moving and reconfiguring the Roombots. However they need local information on the world around them and the direction of the gradient vector to operate. If they had sensing capabilities, the Roombot modules could obtain that information but it is easier to simulate these sensing capabilities in Webots using a World Controller.

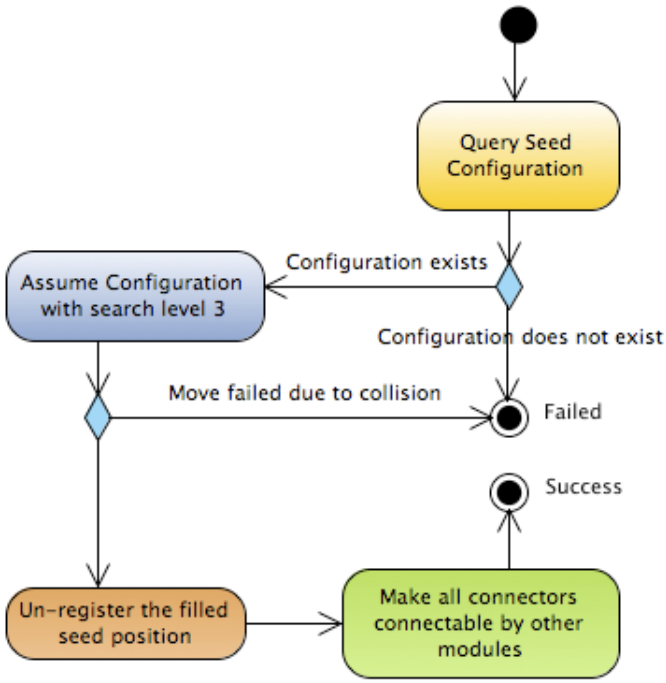


Figure 2.22: Assuming the seed configuration

The World Controller also needs to keep track of the Roombot structure that is being built. This part could eventually be designed and implemented directly into the Gradient Controller or another decentralized module running inside the Roombots but we decided to implement it into the World Controller for simplicity and testing purposes. For a Roombot structure to be built without collapsing, the build has to be performed in a bottom-up approach. Upper seed position can only be filled once the lower seed positions have been filled and are stable enough to supper other modules. A Roombot structure is thus made out of Roombot meta-modules which are labelled with a build level, a positive integer indicating when in the build process they are activated and can be filled. At the beginning of the building process, only level 0 meta-modules are unlocked as seed positions and once all level 0 seed positions are filled, level 1 meta-modules are unlocked as seed positions and so on.

The World Controller keeps track of these seed position configurations and makes the appropriate changes to ensure the correct building order of the structure. It also keeps an internal state of the world in which the Roombots evolve. This world is represented as a finite three-dimensional grid in which the content of every grid position is described by an integer value. Here are all the possible integer values that the Roombot world can be made of.

1. **Meta-module ID** = 1...n
2. **World Free** = 0
3. **World Connector** = -1
4. **World Wall** = -2
5. **World Seed** = -3
6. **Covered Seed** = -4
7. **Covered Seed Other** = -5
8. **World Undefined** = -6

All positive integers ≥ 1 are meta-module IDs and correspond to the position of those respective meta-modules. When the world contains 0's, then that part of the world is free of any meta-modules, obstacles, seeds or otherwise. A meta-module can freely move there without colliding. A World Connector (-1) means that this grid part of the world is filled by something that a module can lock on to. A World Wall (-2) is simply an obstacle that can not be locked on to. A World Seed (-3) means that this grid part of the world contains a seed position that can (and should) be filled by a Roombot meta-module. Covered Seed (-4) means there is a seed position at that specific grid position and the meta-module

who requested the information on the world is currently covering that seed position with one of it's four cubes. Covered Seed Other (-5) is the same as Covered Seed except that another meta-module than the one requesting the information is currently covering the seed position with one of it's cubes. Finally, World Undefined (-6) is simply an undefined part of the world, usually indicating that a Roombot meta-module can not venture there.

In order to ease the creation of new Roombot structures to build, an XML format has been designed for this controller to specify where every Roombot meta-module is placed inside the whole structure and what configuration it has. Here is an example of an XML file defining a box-like structure made out of 4 Roombots.

```
<structure>
    <module level="1">
        <root x="0" y="0" z="0" />
        <cube x="1" y="0" z="0" />
        <cube x="1" y="0" z="-1" />
        <cube x="1" y="0" z="-2" />
    </module>
    <module level="2">
        <root x="1" y="-2" z="0" />
        <cube x="0" y="0" z="-1" />
        <cube x="0" y="0" z="-2" />
        <cube x="0" y="1" z="-2" />
    </module>
    <module level="1">
        <root x="0" y="-2" z="0" />
        <cube x="-1" y="0" z="0" />
        <cube x="-1" y="0" z="-1" />
        <cube x="-1" y="0" z="-2" />
    </module>
    <module level="2">
        <root x="-1" y="0" z="0" />
        <cube x="0" y="0" z="-1" />
        <cube x="0" y="0" z="-2" />
        <cube x="0" y="-1" z="-2" />
    </module>
    <offset x="7" y="6" z="12" />
</structure>
```

The structure node defines a new Roombot structure made out of many module nodes and one offset node. Module nodes define the position and configuration of Roombot meta-modules (4 cubes) inside the Roombot structure. The root node defines where the root cube of the meta-module is positioned inside the Roombot structure. The other 3 cube nodes define the configuration of the meta-module by providing the position of the 3 other cubes of a meta-module relative to the root cube. There can be as many module nodes as necessary inside the structure node to build the Roombot structure. A module node is made of exactly one root node and 3 cube nodes. The level attribute of the module node defines the time when this module will be available to be filled as a seed position. The bigger the level, the longer it will take before the module will be activated as a seed position during the building phase because modules of level i are unlocked as seeds only once all seed positions of level $i-1$ are filled.

Finally, the offset node defines the position where this particular structure will be built in the world. More precisely, it tells the controller that the relative position $(0,0,0)$ of the Roombot structure should correspond to the absolute position $(7,6,12)$ in the Roombot world (as in the example above). This allows one to define a Roombot structure without concerning itself with it's placement in the absolute world. The offset node takes care of positioning the structure where it should be in the world.

2.7.3 Class Diagram

Figure 2.23 contains the class diagram of the world controller module. It is made of three classes : the main WorldController class, the RoombotStructure class and the Seed class. The WorldController class contains the main control loop of the World Controller module and uses the RoombotStructure class to store and organize the data of the Roombot Structure currently being built. The World Controller is implemented as a reactive controller using the messaging library of Mayer2009 [5] for inter-controller communication. It basically listens for incoming messages from individual Roombots, processes their requests and sends them back a message with the desired information.

The main goal of the RoombotStructure class is to store the and manage the Roombot structure that is currently being built. Before the reconfiguration process, it has to parse the correct XML file on disk and store it inside it's data structures. During the reconfiguration process, it keeps track of all the active and inactive seeds and makes all the appropriate changes when a new seed position is filled.

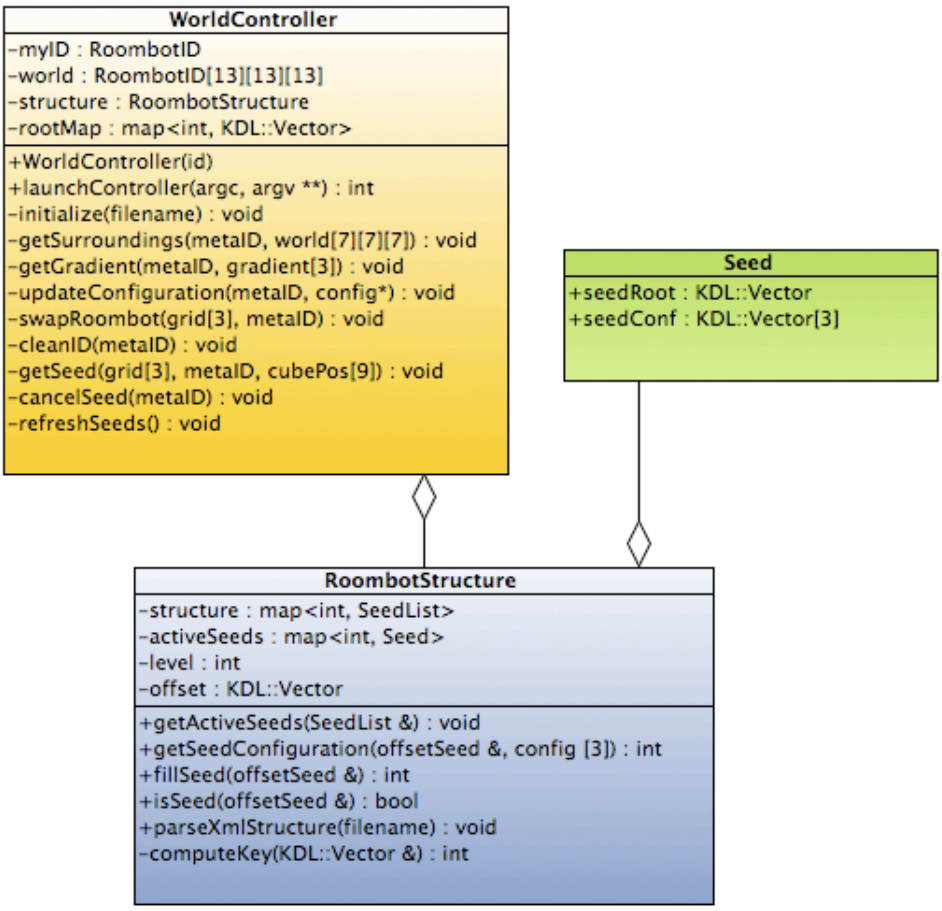


Figure 2.23: Class diagram of the World Controller module

2.7.4 Procedure Implementation

The main control loop of the World Controller is made to manage the following message types and information requests.

1. **Need World**
2. **Send Configuration**
3. **Need Seed**
4. **Cancel Seed**
5. **Send Swap**

The Need World message is a request from a Roombot asking for the content of the world in it's immediate surroundings. This will return both information of the world as well as information on the gradient vector. A Send Configuration message is a Roombot informing the World Controller of a change in it's configuration (position of all 4 cubes in the world). A Need Seed message is a request from a Roombot asking for the configuration of a particular seed in the Roombot structure. This will return the 4 cube positions (the configuration) of that particular seed. A Cancel Seed message is a Roombot announcing to the World Controller that it has filled a seed position and wants to unregister that seed. This will effectively avoid any other Roombot modules to try and fill that particular seed position. Finally, a Send Swap message is a Roombot informing the World Controller that is has just ordered a connector lock swap on it's meta-modules. The locks on the root and the goal cubes of the meta-modules have switched.

2.7.5 Conclusion

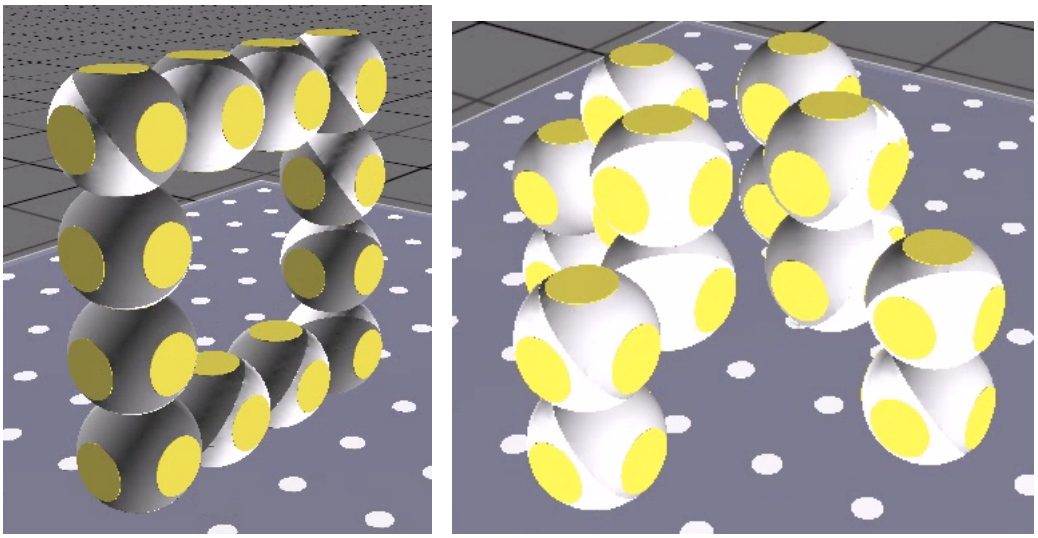
The World Controller plays an important part in the distributed self-reconfiguration framework, providing Roombots with crucial informations on the content of the world and the gradient field. Although this controller is implemented as a centralized controller (only one instance of the controller is running during any webots simulation), the decentralized spirit of the distributed framework is still maintained as the World Controller does not make any important decisions as to how the Roombots should behave, move, think or otherwise. It only provides the Roombots with localized information that they could also obtain through sensing their environment with cameras, sensors or other local scanning devices. In fact, if sensors are implemented on the Roombot modules, we could replace the centralized World Controller with a distributed Sensor Controller without changing the behavior of the Roombots.

Chapter 3

Results and Further improvements

3.1 Results analysis

The decentralized self-reconfiguration framework presented in the previous chapter allowed us to automatically create many different Roombot structures. We have tested the algorithm using 4 Roombot meta-modules as seen in Figures 3.1(b), 3.2(a) and 3.2(b) as well as with 6 meta-modules as seen in Figure 3.3. One can very easily create a new Roombot structure by defining the corresponding xml file without the need of recompiling the controller. It would certainly be possible to create Roombot structures with 8 and even 10 Roombots even though it would take more time to create them. As the controller is decentralized, the computation time scales very nicely with the number of modules. The simulation speed runs in near real-time speed which is a great success for the framework. Videos of different structure reconfigurations can be viewed on my project web page [16].

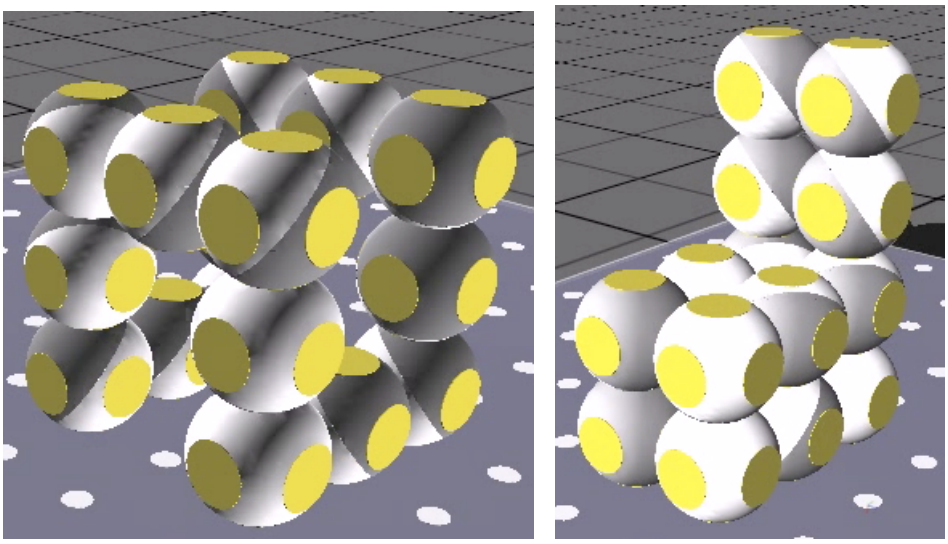


(a) Roombot structure 1 (b) Roombot structure 2

Figure 3.1: Roombot structures built with the decentralized reconfiguration framework

One of the main objectives of the decentralized framework was to obtain very good performances and allow for self-reconfiguration to be performed with large number of modules. With our simulations, we were able to obtain near real-time performances with Webots running on a dual-core 2.1Ghz standard MacBook. The main bottleneck in our application comes from the collision detection and avoidance module. This module has to compute all the grid positions that a meta-module passes through when moving between two configurations. To do so, it will simulate the movements by incrementing the servo rotations by a small value. In the code, it uses the *ROTATION_STEP* parameter to set this value. The smaller the *ROTATION_STEP*, the longer the cloud computation because it needs to go through more intermediate rotations to simulate the whole movement.

We had to find a good value for this parameter so that we obtain good performances without loosing too much precision in the simulation. We finally decided to use a *ROTATION_STEP* value of $\pi/8$. With this value, we can do a



(a) Roombot structure 3 (b) Roombot structure 4

Figure 3.2: Roombot structures built with the decentralized reconfiguration framework

level 1 configuration search in less than 10 millisecond, a level 2 configuration search under 30 to 40 milliseconds and a level 3 configuration search somewhere under 2 to 3 seconds. As we perform almost only level 2 configurations searches throughout the whole reconfiguration process, the speed is very reasonable. We notice slowdowns when level 3 configurations searches are performed but these only happen when Roombot meta-modules are entering seed positions, which happens about one time per meta-module in the simulation. If we had used a $ROTATION\_STEP$ of $\pi/360$ for instance, level 2 searches would have taken 3 to 5 seconds to run and level 3 searches between 300 to 440 seconds. These results are much slower and less desirable for a real-time application considering that the gain in precision is minimal.

The servo rotation database is another time-consuming procedure in the decentralized framework. We have to generate the database once per controller at the beginning of the simulation and then query it for specific configurations. On a dual-core 2.1Ghz standard MacBook from Apple, generating the database takes somewhere between 14 to 16 milliseconds and querying it less than 1 millisecond. These values are acceptable for the majority of chipsets and embedded devices out there. The major bottleneck might be the level 3 configuration searches which might need to be switched off or tweaked if they are going to be executed on low frequency embedded processors.

Additionally, some interesting results have been obtained through empirically studying results from many simulations. The meta-modules that have a connection type of *SRS* and *SRZ* can reconfigure inside the framework more easily than meta-modules with connection types *PER* and *PAR*. All four connection types can produce exactly 63 different kinds of configurations but it seems that the configurations induced by the *SRS* and *SRZ* meta-modules are more suitable for the reconfiguration process. They usually find solutions such as climbing over obstacles easier and tend to become less stuck. We have also noticed that only two active connectors are required on each of the extremity cubes to build all the structures in this project. By rotating the last two servos, we can orient that last connector in whatever direction we want to lock on neighbor modules. We might want to use more active connectors if we want a structure to be more stable but for moving and building the structures, only two active connectors are required. In our framework, we used the *C0X* connector of the first Roombot module and the *C3X* connector of the second Roombot module.

The framework relies heavily on collision detection to perform well and as a direct result, it performs more and more poorly the more the meta-module are clogged up together. When the meta-modules are close together, they have a lot less movement freedom and need more time before they can find suitable configurations without colliding into each other. For this reason, we used an initial meta-module configuration such as the one seen in Figure 3.4 because the meta-modules are nicely spaced out from one another. Another problem with the collision detection comes from the fact that the meta-modules can only see 3 grid positions in every direction. It is possible to construct a scenario where two meta-modules are separated by 4 grid positions. None of the two meta-modules can see the other one and they are close enough to run into each other. But even if they do collide, which happens sometimes but not that often, they are capable of resolving the conflicts and return to their own business.

In some very special cases, the collision detection algorithm, specifically the *canmove* method which tells a meta-

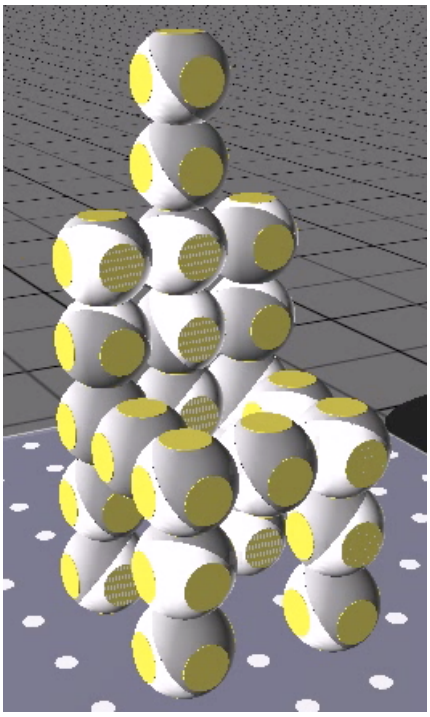


Figure 3.3: Roombot structure 5

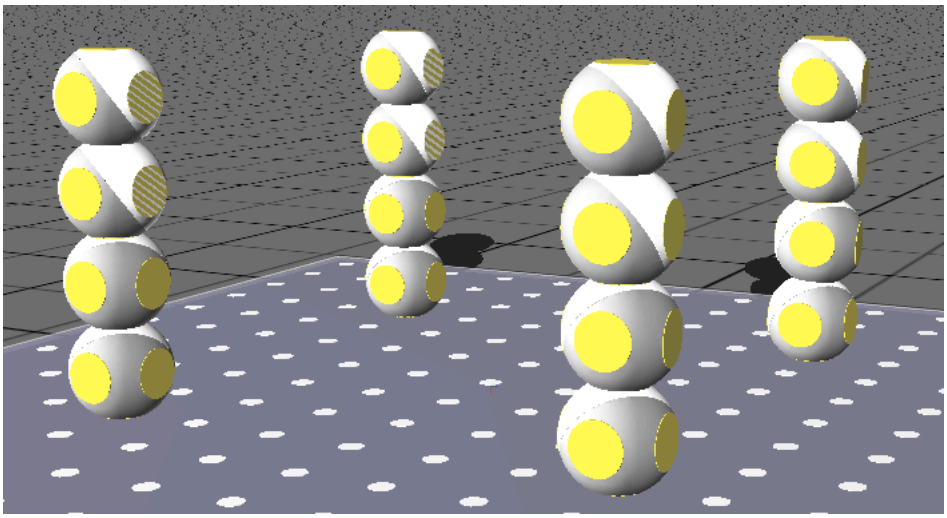
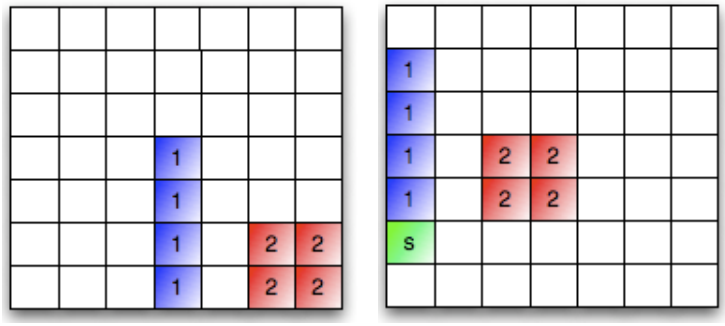


Figure 3.4: Initial Roombot structure

module if it is allowed to move or not, can result in a deadlock. At least 2 meta-modules need to be in special configurations and states for such a deadlock to happen, each one effectively waiting on the other one to move first. We can easily solve the problem by allowing only one module to move at any given time. Figure 3.1 shows the 2 worlds visible by each of the meta-module implicated in a deadlock. Meta-module 1 is in blue, meta-module 2 is in red and the seed is green. The first meta-module in Figure 3.5(a) does not see the seed as it is one grid position to far from it's surrounding environment. It will only consider the direction of the gradient vector which goes down and with respect to that gradient vector, the second module is closer to the seed than the first module. In the first module's perspective, it should not move because the second module is closer to the seed. The second module in Figure 3.5(b) however sees the seed in it's surrounding area and will not move as it clearly sees that the first module is closer to the seed than itself. The situation thus resolves in a deadlock.



(a) World of the blue module (b) World of the red module

Figure 3.5: Deadlock situation where each meta-module waits for the other meta-module to move

3.2 Further Improvements

Further improvements can be brought to the collision detection module which basically allows Roombot meta-modules to move or not. The most simple and trivial method one can implement is simply letting one meta-module move at any given time. In our decentralized framework, we have tried to let multiple Roombot meta-modules move as long as they do not interfere with each other but we noticed that collisions are still possible with our implementation and worst, deadlocks happen in some relatively rare cases. It would be nice to find a set of local rules that will prevent both collisions and deadlocks in any scenario. It is however hard to check wether a method works or not other than throughly testing it.

One of the problems of the framework in it's current state is the difficulty to reconfigure when the modules are very close to each other. Typically, when the Roombots have built a structure, they are very close to each other and reconfiguring into another structure in that case is relatively hard for them unless they are first separated. One nice add-on to the framework would be a module that effectively separates the meta-modules forming a structure before allowing them to reconfigure into another one. The structure should be taken apart following a top-down approach. We could use the information on the seed levels provided by the structure's xml file to unmount the structure without collapsing it and separating the modules from each other so that they look like Figure 3.4. It is then easy for them to assemble themselves into any structure they want.

Another nice add-on to the framework would be a module that effectively assigns meta-modules into a three dimensional structure that was designed using a 3D CAD tool. The software would need to assign meta-modules into the 3D structure and figure out a build order that would prevent the structure from collapsing at any point in it's construction. The result could be directly injected into the controller or it could generate an xml file in the same format as is actually recognized by the controller. This would allow one to simply design a shape in a 3D CAD tool and then watch the Roombots reconfigure themselves into that shape. Manually assigning Roombot meta-modules inside a structure isn't a problem with 6 or 8 modules but becomes impossible if we scale up to hundreds or thousands of modules.

3.3 Conclusion

The world of modular robotics is a fascinating world full of surprises and discoveries. It is a very young field of research and many progress is still awaited. One of the greatest powers of modular robots lies in their ability to reconfigure. They can adapt to their environment and change shapes to perform tasks with higher efficiency. Reconfiguration isn't however an easy problem to solve. During this project, we have studied the Roombots, a new generation of modular robots designed by the Biologically Inspired Robotics Group at EPFL. We have designed a decentralized framework allowing these Roombots to reconfigure themselves in various structures and shapes. Many challenges have been tackled such as collision detection with the environment, collision avoidance with other Roombots and computation time. An important result of the framework is it's capacity of reconfiguring with almost real-time performances which is a significant improvement over centralized algorithms that have been implemented for the Roombots in the past. The Roombot modules have a nice and bright future in front of them and we'll hope that many important breakthroughs in modular robotics will have been achieved with their help.

Bibliography

[1] D. J. Christensen and K. Stoy, "Selecting a meta-module to shape-change the atron self-reconfigurable robot," in *Proc. IEEE International Conference on Robotics and Automation ICRA 2006*, May 15–19, 2006, pp. 2532–2538.

[2] S. Murata, E. Yoshida, A. Kamimura, H. Kurokawa, K. Tomita, and S. Kokaji, "M-tran: self-reconfigurable modular robotic system," vol. 7, no. 4, pp. 431–441, dec 2002.

[3] A. Sproewitz, A. Billard, P. Dillenbourg, and A. J. Ijspeert, "Roombots-mechanical design of self-reconfiguring modular robots for adaptive furniture," in *Proc. IEEE International Conference on Robotics and Automation ICRA '09*, May 12–17, 2009, pp. 4259–4264.

[4] BIRG, "Biologically inspired robotics group, epfl, lausanne." [Online]. Available: http://birg.epfl.ch

[5] M. Mayer, "Roombot modules - kinematics considerations for moving optimizations," Biologically Inspired Robotic Group," Semester Project, 2009.

[6] J. Lotfi, "Self-reconfiguration for adaptive furniture," Master's thesis, Ecole Polytechnique Federale de Lausanne, 2009.

[7] Webots, Tech. Rep. [Online]. Available: http://www.cyberbotics.com/

[8] M. Asadpour, A. Sproewitz, A. Billard, P. Dillenbourg, and A. J. Ijspeert, "Graph signature for self-reconfiguration planning," in *Proc. IEEE/RSJ International Conference on Intelligent Robots and Systems IROS 2008*, 2008, pp. 863–869.

[9] M. Asadpour, M. H. Z. Ashtiani, A. Sproewitz, and A. Ijspeert, "Graph signature for self-reconfiguration planning of modules with symmetry," *IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, 2009.

[10] S. Murata, H. Kurokawa, E. Yoshida, K. Tomita, and S. Kokaji, "A 3-d self-reconfigurable structure," in *in Proc. of the IEEE Int. Conf. on Robotics and Automation (ICRA)*, 1998.

[11] Y. Terada and S. Murata, "Modular structure assembly using blackboard path planning system," *ISARC*, 2006.

[12] M. Krivokon, P. Will, and W.-M. Shen, "Hormone-inspired distributed control of self-reconfiguration," in *Proc. IEEE Networking, Sensing and Control*, Mar. 19–22, 2005, pp. 514–524.

[13] W.-M. Shen, P. Will, A. Galstyan, and C.-M. Chuong, "Hormone-inspired self-organization and distributed control of robotic swarms." *Autonomous Robots*, vol. 17(1), pp. 93–105, 2004. [Online]. Available: http://www.isi.edu/robots/prl/shen2004hormone-inspired-self-organization-and-distributed-control.pdf

[14] R. Fitch and Z. Butler, "Million module march: Scalable locomotion for large self-reconfiguring robots," *The International Journal of Robotics Research*, vol. 27, pp. 331–343, 2008. [Online]. Available: http://www.cs.rice.edu/ mmoll/RSS06/fitch.pdf

[15] K. Stoy, "Using cellular automata and gradients to control self-reconfiguration," *Robotics and Autonomous Systems*, vol. 54, pp. 135–141, 2006.

[16] P. Laprade, "Project website." [Online]. Available: http://birg.epfl.ch/page71118.html

[17] S. Lepine, "Locomotion in modular robots: Yamor host 3 and roombots," Biologically Inspired Robotic Group," Semester Project, 2008.

[18] A. Tuleu, "Roombots - central pattern generators, symmetries and online learning." Biologically Inspired Robotic Group" Semester Project, 2009.

[19] CMake, “Cross-platform build tool,” Tech. Rep. [Online]. Available: http://www.cmake.org

[20] Eigen2, “C++ template library for linear algebra,” Tech. Rep. [Online]. Available: http://eigen.tuxfamily.org

[21] KDL, “Kinematics dynamic library,” Tech. Rep. [Online]. Available: http://www.orocos.org/kdl

[22] TinyXML, “Xml parser library.” [Online]. Available: http://sourceforge.net/projects/tinyxml/