University of Colorado, Boulder
CU Scholar

Electrical, Computer & Energy Engineering Graduate Theses & Dissertations | Electrical, Computer & Energy Engineering

Spring 1-1-2011

Self Assembly of Modular Robots with Finite Number of Modules Using Graph Grammar

Vijeth Rai
University of Colorado at Boulder, vijethrai.1987@gmail.com

Follow this and additional works at: https://scholar.colorado.edu/ecen_gradetds

Part of the Robotics Commons

Recommended Citation

Rai, Vijeth, "Self Assembly of Modular Robots with Finite Number of Modules Using Graph Grammar" (2011). *Electrical, Computer & Energy Engineering Graduate Theses & Dissertations*. 36.
https://scholar.colorado.edu/ecen_gradetds/36

This Thesis is brought to you for free and open access by Electrical, Computer & Energy Engineering at CU Scholar. It has been accepted for inclusion in Electrical, Computer & Energy Engineering Graduate Theses & Dissertations by an authorized administrator of CU Scholar. For more information, please contact cuscholaradmin@colorado.edu.

Self Assembly of Modular Robots with Finite Number of Modules Using Graph Grammar

by

Vijeth Rai

B.E., VTU, 2008

A thesis submitted to the

Faculty of the Graduate School of the

University of Colorado in partial fulfillment

of the requirements for the degree of

Master of Science

Department of Electrical Computer and Energy Engineering

2011

This thesis entitled:
Self Assembly of Modular Robots with Finite Number of Modules Using Graph Grammar
written by Vijeth Rai
has been approved for the Department of Electrical Computer and Energy Engineering

---

Nikolaus Correll

Prof. Sam Siewert

Dustin Reishus

Date ____________

The final copy of this thesis has been examined by the signatories, and we find that both the content and the form meet acceptable presentation standards of scholarly work in the above mentioned discipline.

Rai, Vijeth (Electrical Engineering)

Self Assembly of Modular Robots with Finite Number of Modules Using Graph Grammar

Thesis directed by Prof. Nikolaus Correll

We wish to design decentralized algorithms for self-assembly of robotic modules that have 100% yield even if the number of available building blocks is limited, and specifically when the number of available building blocks is identical to the number of blocks required by the structure. In contrast to self-assembly at the nano and micro scales where abundant building blocks are available, modular robotic systems need to self-assemble from a limited number of modules. In particular, when self-assembly is used for reconfiguration, it is desirable that the new conformation includes all of the available modules. We propose a suite of algorithms that (1) generate a reversible graph grammar, i.e., generates rules for a desired structure that allow the structure not only to assemble, but also to disassemble, and (2) have a set of structures that are growing in parallel converge to a single structure using broadcast communication. We show that by omitting a reversal rule for the last attached module, self-assembly eventually completes, and that communication can drastically speed up this process. We verify our results by running simulations on Matlab and Player/Stage 2D simulator

Acknowledgements

I would like to thank Anne van Rossum for his constant support and assistance. I would also like to thank my advisor Prof. Nikolaus Correll for his guidance and patience.

The research is supported by the European Project REPLICATOR within the work program "Cognitive Systems, Interaction, Robotics" under grant agreement no. 216240.

Contents

Chapter

1 Introduction 1

1.1 Replicator Platform and Algorithm Assumptions 2

1.2 Graph-Grammars 3

2 Algorithms 5

2.1 Reversible Single Growth Point Algorithm 5

2.2 Accelerated Self-Assembly using Wireless Communication 6

3 Matlab Simulation 11

3.1 Simulations 11

3.1.1 Reversible Single Growth Point Algorithm 12

3.1.2 Graph-Grammar using Wireless Communication 12

4 Player/Stage 2D simulation 13

4.1 Simulation: 13

4.1.1 Assumptions/Conditions: 13

5 Results and Discussion 16

5.1 Results 16

5.1.1 Reversible Single Growth Point Algorithm 16

5.1.2 Accelerated Self-Assembly Algorithm 16

5.2 Discussion 24

6 Conclusion 25

Bibliography 26

7 Player/Stage 28

7.1 Player/Stage Background 28

7.1.1 Player: 28

7.1.2 Stage 29

7.2 How Player Works 31

7.3 Building the Environment: 32

7.3.1 World file and Robot Specs 32

7.3.2 Configuration File 34

7.4 Client Code: 36

7.4.1 Player Client and Proxies 36

7.4.2 Reading data 39

7.4.3 Simulating Multiple Robots: 39

Tables

Table

Figures

Figure

1.1 Replicator Prototype 2

5.1 Desired Hexapod Assembly 17

5.2 Average Assembly Size vs Time for RSGPA 17

5.3 Average Time for varying Forward Reaction Rate 18

5.4 Number of Seeds vs Time for ASA 20

5.5 Average Assembly Time with varying MSI 20

5.6 Yield Distribution with MSI of 50 frames/message 21

5.7 Yield Distribution with MSI of 500 frames/message 21

5.8 Player/Stage results for varying MSI 22

5.9 Number of Seeds for varying MSI 22

5.10 Player/Stage Simulation with Multiple Seeds 23

7.1 Player Server Client model with actual Robot 30

7.2 Player Server model with Stage 30

7.3 Simulation window with robots and map initialized 37

Chapter 1

Introduction

We are interested in self-assembly algorithms for assembling arbitrary structures from strictly identical robotic modules. Combined with self-disassembly, such systems could actively change their shape in response to environmental cues. For example, the platform shown in Figure 1.1 could self-assemble into a hexapod, walk at high-speed and re-configure into a snake when encountering a narrow passage.The problem of self-assembly of robot modules into larger structures has been approached from different angles: Stochastic assembly of passive robots [24], using discrete rules implemented by cellular automata [10], gene regulatory networks in simulation [23], artificial neural networks [1, 22] and graph grammars [8], among others. Issue of regeneration and repair provide an additional perspective on the self-assembly problem [10, 20]. Whereas centralized planning for assembly and reconfiguration can achieve results theoretically in deterministic time [3, 11], self-assembly using stochastic rules converges only eventually. Accelerating assembly by controlling the reaction rate for assembly and disassembly has been proposed in [17, 14]. An additional problem in guaranteeing successful assembly arises when the number of modules is limited. In this case, parallel assembly of multiple competing structures — which is characteristic for a decentralized, stochastic self-assembly process — can lead to the depletion of building blocks, preventing a single structure from completing. Although disassembly can contribute to freeing up the required modules, completion is still probabilistic, which is undesirable in a modular robotic self-assembly and reconfiguration scenario. Nevertheless, stochasticity is attractive with respect to: scalability [25], robustness, both with respect to non-deterministic environments and noise in assembly/disassembly

Figure 1.1: **Left:** REPLICATOR robot prototype, courtesy of Replicator and Symbrion consortia. The robots have autonomous screw drives and four connectors to connect to other modules. There are also four cameras mounted on each side to detect the state of other robots. **Right:** Motivating example. 23 REPLICATOR modules can assemble to a hexapod on the plane. The structure can then stand up and walk using internal hinge joints. This thesis focus on graph grammar-based, stochastic algorithms to enable self-assembly of such a structure from limited number of blocks.

events (local minima can be escaped by random movements [21]), and no need for a dedicated seed robot [2] (contrary to [7]).

In this thesis, we are studying the particular case of self-assembly from limited building blocks and propose algorithms that let the system converge to a single structure by using a combination of forward and reversal graph grammar rules and broadcast communication. Convergence of the proposed algorithm as a function of the interval with which broadcast communication is sent and the likelihood with which the message is transmitted correctly are explored in simulation.

1.1 Replicator Platform and Algorithm Assumptions

In the field of modular robotics, self-assembly and reconfiguration allow invidual robots to grow into large robotic structures and morph from one structure to another. Possible algorithms are tightly related to the sensing, actuation, computation and communication capabilities of the actual platform. For example, modular robots such as the compressible Crystalline [3] or the hinge-based M-TRAN [11] achieve similar goals, but drastically differ in hardware capabilities and requiring different classes of motion planning algorithms. This thesis is motivated by the *REPLICATOR*

platform (Fig. 1.1), left). This is a modular robot that can autonomously move via a differential wheel drive, and connect with other modules via four of its sides. The REPLICATOR platform contains an additional hinge, which allows the robot to assemble to, for instance, a walking hexapod, Figure 1.1, right. Modules communicate via sensing, where the state of the other module, and its position, can be perceived via (flashing) LEDs and CCD cameras mounted at each face. Using the differential wheel drive, modules can actively explore the environment, detect other modules, align with them and lock. Graph grammars in this thesis are limited to binary rules to enforce locality and minimal awareness. Only *two* modules are allowed to communicate before a connection or disconnection event takes place. No third module is allowed to participate in a communication event.

We present algorithmic solutions for assembly of acyclic structures, such as hexapod shown in Figure 1.1. These structures allow modules to access other modules, to attach and detach easily on a single plane.

1.2 Graph-Grammars

Graph grammars, or graph rewriting systems, are rule sets that transform one graph into another. In a self-assembly context, a desired assembly can be represented as a graph. The assembly process becomes a sequence of (labeled graph) transformations of an edgeless graph into the target graph. In this thesis, we use the Graph Rewriting Systems on Induced Subgraphs [12] as shown in [8]. This graph rewriting systems takes one subgraph as input and has another subgraph as output (in the process removing or adding edges and changing the labels).

In our case, the nodes represent the Replicator modules, the rewriting represents reconfiguration of these modules, and the labels represent module states. The use of binary rules (see motivation above) limits our graph rewriting rules to handling subgraphs of maximum size two. These binary graph grammar rules are of the form:

$$\phi_{fi} : X \quad A \Rightarrow Y - Z \tag{1.1}$$

For executing a rule the labels of the two modules constituting the subgraph are compared to the LHS (left-hand side) of the graph grammar rule ϕ_{fi}. If these subgraphs match, they are replaced by the subgraph shown on the RHS of the ϕ_{fi}. This replacement indicates that the states of the original modules X and A are replaced by Y and Z respectively. The actual physical connection between modules is indicated by the existence or absence of edges '$-$' between the nodes of the subgraph on either side of the rule ϕ_i. We therefore refer to ϕ_{fi} as a **construction rule**. In order to break up a connection, we can define a **reversal rule**

$$\phi_{ri} : Y - Z \Rightarrow X \quad A \tag{1.2}$$

Thus, connections can be made or broken between modules, depending on whether the LHS of the rule is satisfied, which itself might depend on the existing connection between modules represented by the subgraph. Reversal rules, can be executed by the environment as in [16] or by active decisions of the modules themselves, which then need to exchange their states and initiate a disconnect sequence when either one detects a valid LHS. In this thesis, we use the convention that atomic modules that are not part of a structure are in state A.

Finally, we can use graph grammars to communicate state throughout the structure using a **communication rule**:

$$\phi_{ci} : U - W \Rightarrow P - Q \tag{1.3}$$

Notice that communication rules do not affect connectivity, but simply change the states of both the modules involved.

We also use the concept of **degree** in both our algorithms similar to the graph definition of degree. The degree of a module is the number of other modules attached to its docking ports.

Chapter 2

Algorithms

We will first present an algorithm that generates forward and reversal rules to assemble any desired structure that can be represented as a graph and leads to complete assembly of a structure even if the number of available modules is limited to the number of modules required to assemble the structure. We will then present an algorithm that lets an ensemble of structures, that assemble in parallel, converge to a single structure using broadcast communication.

2.1 Reversible Single Growth Point Algorithm

Given a graph $G = (V, E)$ (or more specifically, a tree) with a set of vertices V and edges E, we present the Reversible Single Growth Point (RSGP) algorithm to construct a rule-set that allows for decentralized growth of this given graph G. It is subdivided into several sub-algorithms. Algorithm 1 initializes the search by starting from a leaf edge and creating construction rules till a node is encountered with degree greater than 2. For each of such high-degree nodes, algorithm 2 performs a Depth First Search (DFS) with backtracking. The DFS adds all constructions rules necessary to create the rest of the structure. Backtracking is implemented by algorithm 4 by generating rules that only involve state changes (and no construction or reversal). We refer to these rules as communication rules. When algorithm 2 returns, algorithm 1 collects all the found rules, and reverses each individual rule, except for the last one. This is to ensure that there exists one irreversible transition after which the structure cannot retrace its steps and decompose. At any other stage, the structure is allowed to completely decompose into atomic modules.

If forward and reversal rules would be executed deterministically, the structure could not grow as reversal rules would immediately cancel out every forward rule. We therefore define the **Forward Reaction Rate** (FRR) to be the ratio of the probabilities of forward rule execution to reversible rule execution. Assuming that assembly modules randomly pick a rule from a list of applicable ones, an FRR=m can be achieved by creating m copies of the forward rule ϕ_{fi}, while maintaining only a single reversal rule ϕ_{ri}.

The rule set ϕ_{full} generated by the Reversible Single Growth Point Algorithm to assemble a graph $G = (V, E)$ will eventually complete if the number of available modules $N \geq \|V\|$, i.e., the number of vertices in the desired graph, and if the rules are executed probabilistically by the assembly modules.

Proof. By construction, for every forward and communication rule $\phi_{fi} \in \phi_{full}$, there exists a reversal rule $\phi_{ri} \in \phi_{full}$ **except** for the last construction rule. This is achieved in sub-routine **CreateForwardRule**, which terminates when the desired graph G is a sub-set of G_α, i.e., the graph describing the connectivity of the robotic modules. Therefore, the probability that the final rule needed to finish the structure gets reversed is zero, while all partially connected structures will potentially disassemble with probably larger than zero. □

Note that this theorem is a variant of the Poincaré recurrence theorem [6]. Large FRRs will favor construction (in acyclic structures) over disassembly. Even for $\lim_{FRR \to \infty}$ reversible rule still exist, and so does a Poincaré cycle. This can also be shown by reducing the assembly rules to a chemical reaction network, where the reaction rates are given by the ratio of forward and reversal rules. See also [4] for an analysis of aggregation dynamics with forward and reversal dynamics.

2.2 Accelerated Self-Assembly using Wireless Communication

Although the RSGP algorithm generates rules that lead to complete assembly of a structure even if the number of available modules is limited, its performance is probabilistic and heavily dependent on the choice of the FRR. This is not desirable in a modular robotics context and can

Algorithm 1 *Reversible Single Growth Point Algorithm*

Input: The target graph $G = (V, E)$
Output: Rule-set ϕ_{full}, reversible rule-set for assembly of G

- Let $\mu : V \to L$ be the function assigning labels to all modules, then set $\mu(v) = A, \forall v \in V$
- Let $G_\alpha = (V_\alpha, E_\alpha)$ be the null-graph
- Construct first branch:
  - ∗ Choose an edge $xy \in E$ such that x is a leaf node
  - ∗ $V_\alpha = V_\alpha \bigcup \{x\}$
  - ∗ While $deg(y) = 2$:
    - – $\phi_\beta = \phi_\beta \bigcup$ *CreateConstructionRule(xy)*
    - – Set $x \leftarrow y$ and $y \leftarrow N_G(x)$ (neighbor in G)
- If deg(y) ≥ 3, set root $= y$
  - ∗ Let $\phi_\gamma =$ *CreateForwardRule*(root)
  - ∗ Then $\phi_\delta = \phi_\delta \bigcup \phi_\gamma$
  - ∗ For each rule $\phi_{\delta i}$ (except the last rule)
    - – If $(\phi_{\delta i} : X_i \quad Y_i \Rightarrow R_i - S_i)$, then
      $\phi_{ri} : R_i - S_i \Rightarrow X_i \quad Y_i$
    - – If $(\phi_{\delta i} : X_i - Y_i \Rightarrow R_i \quad S_i)$, then
      $\phi_{ri} : R_i \quad S_i \Rightarrow X_i - Y_i$
- Final rule-set: $\phi_{full} = \phi_\delta \bigcup \phi_r$

Algorithm 2 *CreateForwardRule(a)*

Input: Node $a \in V$ such that $\deg(a) \geq 3$
Output: Rule-set ϕ_γ, construction and communication rules

- For all $y \in N_G(a)$ with $y \nexists V_\alpha$
  - ∗ Let x = a
  - ∗ While $deg(y) = 2$:
    - – $\phi_{\gamma 1} = \phi_{\gamma 1} \bigcup$ *CreateConstructionRule(xy)*
    - – Set $x \leftarrow y$ and $y \leftarrow N_G(x)$
  - ∗ $\phi_{\gamma 1} = \phi_{\gamma 1} \bigcup$ *CreateConstructionRule(xy)*
  - ∗ If $G \subseteq G_\alpha$
    - – $\phi_\gamma = \phi_{\gamma 1} \bigcup \phi_{\gamma 2} \bigcup \phi_{\gamma 3}$
    - – **exit**
  - ∗ Else if $deg(y) \geq 3$ then
    - – $\phi_{\gamma 2} = CreateForwardRule(y)$ recursively
  - ∗ Else (now $deg(y) = 1$) then
    - – $\phi_{\gamma 3} = CreateCommunicationRule(y, a)$

Algorithm 3 *CreateConstructionRule(xy)*

Input: Edge $xy \in E$ with $x \in V_\alpha$ and $\mu(y) = A$
Output: Construction rule ϕ_γ for edge xy and updated G_α

- $X \leftarrow \mu(x)$
- $\phi_\gamma : X \quad A \Rightarrow R - S$, with R and S new labels
- $V_\alpha = V_\alpha \bigcup \{y\}$ and $E_\alpha = E_\alpha \bigcup \{xy\}$

Algorithm 4 *CreateCommunicationRule (x,z)*

Input: Node x and y
Output: Communication rules from node x towards z

- While $x \neq z$
  - ∗ Choose edge $xy \in E_\alpha$, such that $y \in V_\alpha$ is closer to z than x
  - ∗ Let, $U = \mu(x)$ and $W = \mu(y)$
  - ∗ Create $\phi_\epsilon : U - W \Rightarrow P - Q$, where P and Q are new labels
  - ∗ $x \leftarrow y$

be improved upon when the robots have the ability to broadcast communication.

For that reason we introduce the Accelerated Self-Assembly (ASA) algorithm. Let ϕ_G be a rule-set that assembles graph G. This can be a ruleset generated by the RSGP algorithm or a forward-rule only algorithm from the literature [8]. We denote the first construction rule in this rule-set, that connects two free atoms, as the "Seed Forming Rule" and choose one of the modules from the resulting structure to be the "seed". Note that distributed execution of ϕ_G will lead to many structures assembling in parallel. Following this seed assignment, the seed is allowed to grow using the construction rules from the associated rule-set.

Each seed, m_s, will then calculate its structure's size at regular intervals and broadcast it at a constant **Message Sending Interval** (MSI). Evaluating its size can either be achieved by having modules broadcast messages throughout the entire structure in a multi-hop fashion or by relying on the communication rules generated by the RSGP. Upon reception of a broadcast message, each of the other seeds, m_r, will compare its own size with that of the sender. If the recipient's size,$\pi(m_r)$, is smaller than that of the sender $\pi(m_s)$, the recipient sends out a local message to all its connected modules to initiate decomposition starting from the leaf nodes. If, however, the size of the recipient is equal to that of the sender, the recipient would send out the local "decomposition" message with a probability P_{split} to decompose and return all its modules to state A. This can be achieved by appropriate communication and reversal rules (not shown in this thesis). If the receiving seed is part of a structure that is larger than that of the sender, it ignores the message. Using this algorithm, competition between parallel forming structures of similar size arises. We predict that P_{split} increases the variability in organism sizes and – depending on FRR – favors the formation of a complete structure. The process is repeated after another MSI, till the one complete assembly is formed. no favoring of a single dominate seed over the course of time.

Algorithm 5 Accelerated Self-Assembly using Wireless Communication

Input: Target graph $G = (V, E)$,Rule-set ϕ_G to assemble G
Output:Single assembly of G

- Let set $\mathbf{S} = \{m_i \in \mathbf{S} : m_i \text{ is a seed}\}$
- Let $G_\alpha = (V_\alpha, E_\alpha)$ be the null-graph
- Let π: be the function such that $\pi(m), \forall m \in \mathbf{S}$, returns size of structure associated with seed m
- While $G_\alpha \neq G$
    * for all $m_s \in \mathbf{S}$, let m_s be the sender of broad-cast message containing size $\pi(m_s)$ every MSI unit of time
        - let $m_r \in \mathbf{R}$ s.t $\mathbf{R} = \mathbf{S} - \{m_s\}$
        - for all $m_r \in \mathbf{R}$, let m_r be the receiver
            · If $\pi(m_s)$ ¿ $\pi(m_r)$
            · $\rightarrow m_r$ decomposes via local messages
            · If $\pi(m_s) = \pi(m_r)$
            · $\rightarrow m_r$ decomposes with probability of P_{split}¿ 0
            · If $\pi(m_s)$ ¡ $\pi(m_r)$
            · $\rightarrow m_r$ ignores message
    * Let G_α be the graph associated with seed of greatest size.

Chapter 3

Matlab Simulation

3.1 Simulations

The algorithms of chapter 2 are evaluated focusing on the hexapod self-assembly problem shown in Figure 1, right. Each robot module contains 1.) an entire copy of the rule set, 2.) the ability to sense its state and that of its neighbors, and 3.) a program to apply graph-grammar rules upon a label match. To simulate this robotic scenario, we use the following method:

(1) Randomly pick a module m from the environment and read its state $\mu(m)$.

(2) List all the rules which contains this state in the LHS.

(3) Randomly choose a rule from the above list and read the state of the adjacent module in the LHS of that rule.

(4) If this is a construction rule then randomly pick another module from the environment and compare its state with the desired state needed for the execution of the chosen rule.

(5) If the states match then apply the rule and "connect" the modules, else repeat the process from step 1

(6) If, however, the randomly chosen rule is a communication or a reversal rule, pick a connected neighbor module.

(7) If the states match, then apply the rule and alter the states and/or remove the connection.

This algorithm has been implemented in MATLAB using the GraphViz toolbox.

3.1.1 Reversible Single Growth Point Algorithm

In the previous section we showed how to generate forward and reversal rules for a given graph (more specific, for a given tree). Forward rules are rules which propel the assembly process towards completion for an *individual* structure. However, in our case we have a limited number of modules available. Hence, parallel growth of multiple structures at the same time exhausts all modules in the environment before any of these assemblies have reached completion. The addition of reversal rules allow structures to decompose and free atomic modules, that become again available in the environment. In our simulations, we test different ratios between the probability of construction versus reversal defined by the Forward Reaction Rate.

3.1.2 Graph-Grammar using Wireless Communication

In order to demonstrate the viability and robustness of our algorithm we introduce an additional parameter that complements the MSI:

Message Propagation Probability (MPP): This is the probability with which the broadcasted messages are received by other seeds. A MPP of 1.0 implies that all messages will be received, and MPP of 0.5 implies that half the messages will be lost.

Completeness of the ASA algorithm follows directly from completeness of the RSGP algorithm. In particular, even if $MSI \to \infty$ and/or messages fail, all properties of the RSGP algorithm are maintained (provided the same rule-set is used).

Results for the simulation are shown in Chapter 5

Chapter 4

Player/Stage 2D simulation

Whereas the Matlab simulations show theoretical proof of our algorithms, it does ignore aspects such as collision avoidance, field of view etc. In order to further test the veracity of our algorithms, we decided to utilize a more realistic robot environment simulator. We hence decided to move on to Player/Stage.

Background about Player/Stage can be found in Chapter 7.1.1 of this thesis.

4.1 Simulation:

We ran our simulations based on the design discussed above. The simulations were performed to show assembly rate for assembling a 5 robot snake. Player/Stage is inherently different from Matlab, hence a few of our assumptions and conditions were modified accordingly.

4.1.1 Assumptions/Conditions:

- In order to manage connections, we have introduced new sub-states to our free moving robots. Hence we now have free unconnected robots transitioning between free stationary robots (state **'X'**) and free wandering robots (state **'W'**) every **t** seconds, where **t** is randomly decided for each robot.

- Since the Player/Stage platform has no support for connectors, our robots are not going to be physically connected to each other but just placed in close proximity to the robot they are "connected" to in terms of state.

With these assumptions we ran the following procedure:

- All robots start in either state 'X' or 'W' and transitioning between the two states. The connection is always occurring between a stationary robot(referred to as target robot) and a wandering robot (state 'W') that detected the target robot.
- Wandering robots use the camera to detect stationary robots and move closer to them.
- Once they are at close proximity to the target robot, they determine the color and hence, the state of the target robot using a lookup table.
- The wandering robot then sends a broadcast message to all robots stating its GPS location and the state of the target robot it detected.
- All robots receiving the message would then check their own GPS position and if the wandering robot is within a certain radius (1.2 meters in our case) and also checks if its own state matches the state mentioned in the message. This method utilizing the GPS is only implemented in Player/Stage simulations since the "connector" model is not supported. In the actual case, the robots would come in physical contact to initiate connection.
- If both the conditions are satisfied, the recipient replies to the wandering robot with its ID. Now the wandering robot is aware of its target's ID and hence can initiate connection.
- The wandering robot sends another message with the rule number to be executed. This is necessary as there could be multiple rules applicable to the two states concerned and hence there has to be a mutual consensus.
- The target robot, upon receiving the rule number, looks up rule list and determines the new state corresponding to the rule number and switches its state. It also determines the ideal location for the wandering robot to be placed and sends this information back via message. This location is calculated based on the its current orientation and the orientation of port to which the wandering robot has to connect to.

- The wandering robot directs itself to the location indicated in the message and switches its state to indicate connected status.

- The process repeats for all other connections.

All the robots were executing the same code and only maintained a copy of the Graph Grammar Ruleset. We ran several simulations for varying values of Message Sending Interval and P_{split} and the results are shown in the section 5.1.2.2 in the Results chapter.

Chapter 5

Results and Discussion

5.1 Results

5.1.1 Reversible Single Growth Point Algorithm

We generated assembly rules using the RSGP algorithm to self-assemble a hexapod with 23 nodes as shown in Fig. 5.1. This hexapod structure forms 6 leaf nodes and 3 nodes with $deg(n) > 3$.

Figure 5.2 shows the progress (in %) of a sample run with FRR 6, i.e., forward rules are 6 times as likely to be executed than reversal rules. One can see that although progress heavily fluctuates, the structure eventually completes and remains stable.

We also performed simulations to study the impact of the FRR for the hexapod structure. Figure 5.3 shows that small FRRs decrease the likelihood of one of the structures to complete (for FRR = 3 and lower there is no complete structure in the given timespan). Large FRRs seem to favor variability in resulting structure size leading to shorter times in finding a complete structure. We conjecture that the optimal value for the FRR is a function of the number of available modules and the structure size. To show this analytically is subject to future work.

5.1.2 Accelerated Self-Assembly Algorithm

5.1.2.1 Matlab Simulation

The dynamics from a typical run of the ASA algorithm is shown in Fig. 5.4. This figure shows the number of seeds in the environment over time. The message sending interval (MSI)

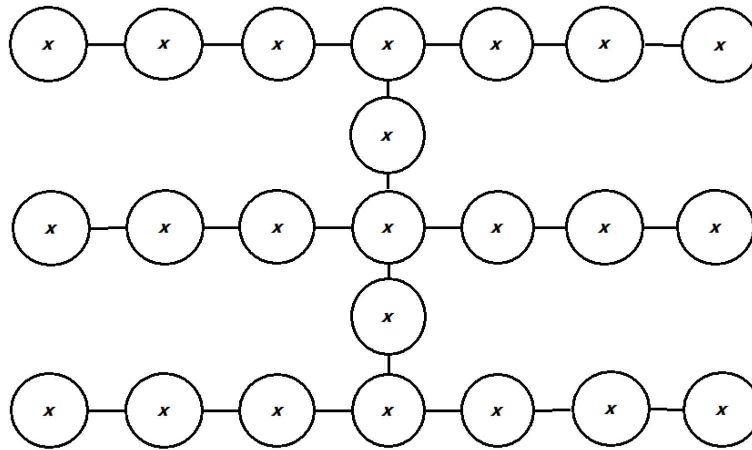

Figure 5.1: Desired structure shown as graph. X label implies that final label of the structure is immaterial to our purpose.

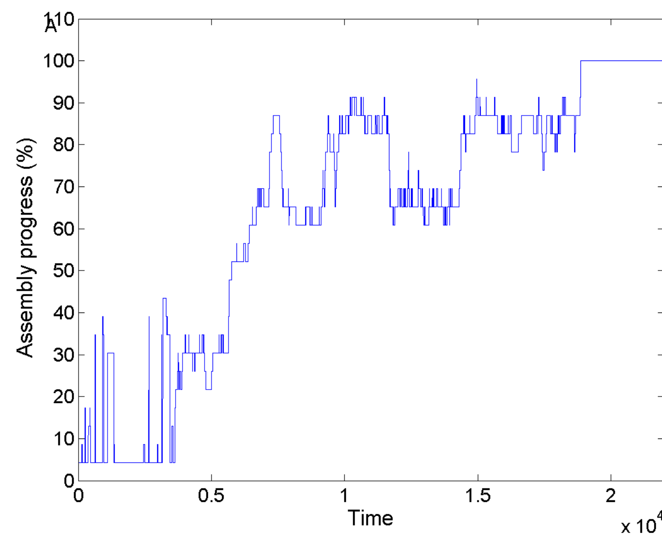

Figure 5.2: Assembly progress (percentage of target size) during the assembly of the 23-module hexapod (sample run). Progress fluctuates due to probabilistic execution of forward and reversal rules and eventually reaches 100%

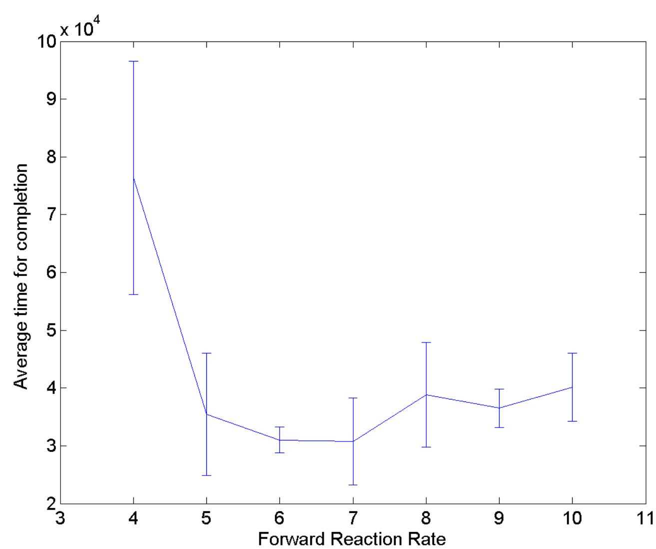

Figure 5.3: Average time needed for one complete assembly of hexapod shown in Figure 5.1, versus forward reaction rate with standard deviation (30 trials each). Desired structures are formed faster with increase in Forward Reaction Rate up to a certain point, after which an increase in FRR tends to slow overall structure completion.

was set to 50 (simulation frames between each message), and the message propagation probability (MPP) set to one, i.e., each message is successfully broadcast. After some initial parallel growth, communication between seeds lets a single structure prevail.

The ASA algorithm's performance is not only a function of the MSI, but also the MPP, which is smaller than one in most real systems. Figure 5.5 shows that there is a linear relationship between the length of the MSI and the assembly time. The figure also shows how decreasing MPP maintains 100% yield but takes longer for completion. Thus, the ASA algorithm maintains its efficiency even with failing message delivery.

Finally, Figures 5.6 and 5.7 show the completion time histograms for MSIs (message sending intervals) of 50 and 500, respectively (100 trials each), with MPP=1.0. A low MSI results in a target structure formed faster than with a high MSI. This is an expected result regarding the fact that feedback from larger formed structures towards smaller structures becomes more frequent.

5.1.2.2 Player/Stage Simulation

Player/Stage gave us the opportunity to test our ASA algorithm in a more realistic scenario. The number of robots in the environment had to be limited to 8 in order to maintain a fairly normal speed of simulation. The structure being assembled was a simple linear 5 moduled robot snake. Figure 5.8 shows variation of assembly rate with change in MSI. Low MSI prevents build up of multiple parallel structures more efficiently than higher MSI. Although both converge to complete assemblies eventually, low MSI allows more free robots to be in the environment for a longer time and thus faster assembly.

Figure 5.9 shows clearly that the number of seeds and thus the number of parallel growing structures is checked more often with low MSI. This prevents consumption of free robots at multiple sites. However, with higher MSI, the number of seeds may grow for quite a while before the next broadcasted message is sent that destroys some of these seeds, as shown in Figure 5.10. This delay of messages slows down assembly time.

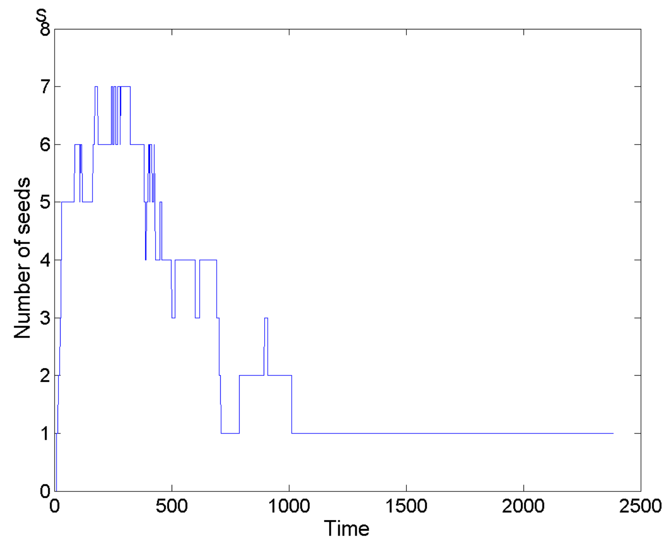

Figure 5.4: Sample run showing the number of seeds versus time for ASA. After some initial parallel growth of multiple seeds, the number of seeds (and structures) reduces to just one due to communication and decomposition of smaller structures.

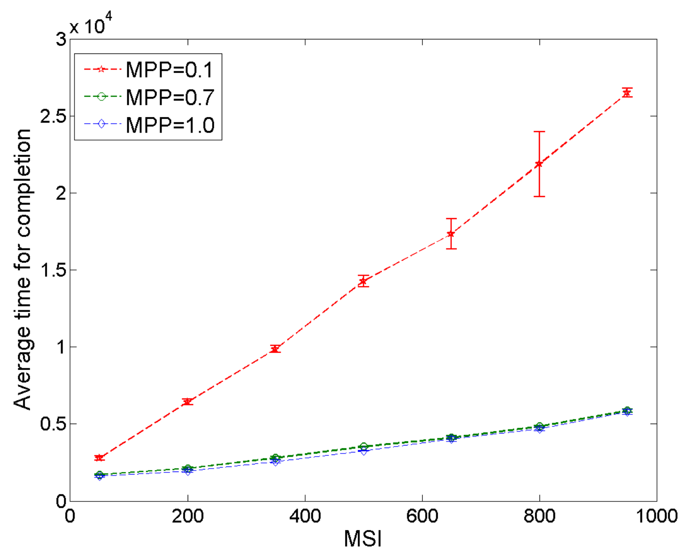

Figure 5.5: Average time needed for assembling the hexapod structure, as shown in Figure 5.1, versus Message Sending Interval (MSI) for varying Message Propagation Probability (MPP). Structures take less time for complete assembly with low MSI. Average assembly time increases with increasing time interval between broadcasted messages, irrespective of the MPP.

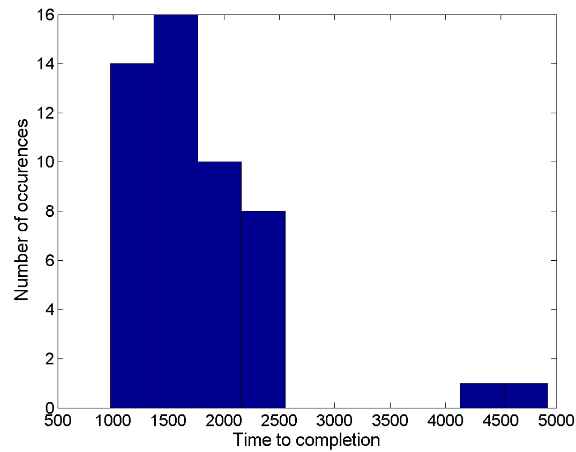

Figure 5.6: Yield distribution with MSI of 50 frames/message vs simulation time in frames. This figure shows maximum hexapods being formed in the range 1000 to 1750 time units

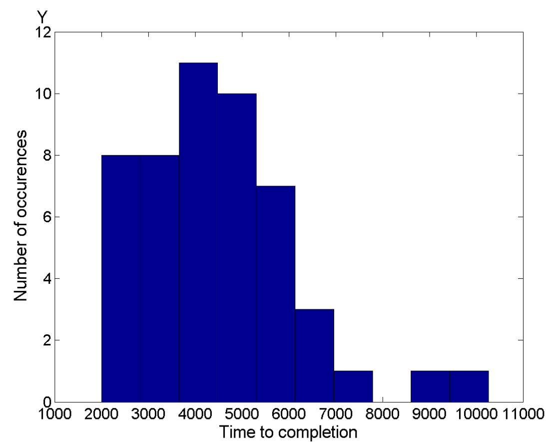

Figure 5.7: Yield distribution with MSI of 500 frames/message vs simulation time in frames. This figure shows maximum yield to be in the range of 4000 to 5500 time units

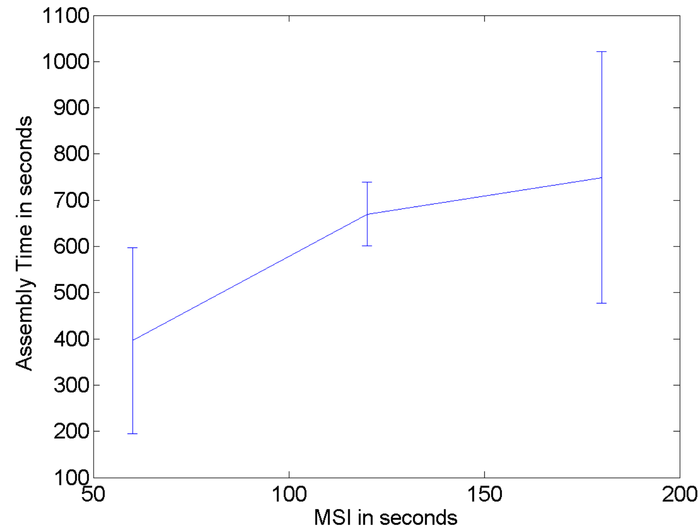

Figure 5.8: Player/Stage results showing Assembly Time vs change in Message Sending Interval

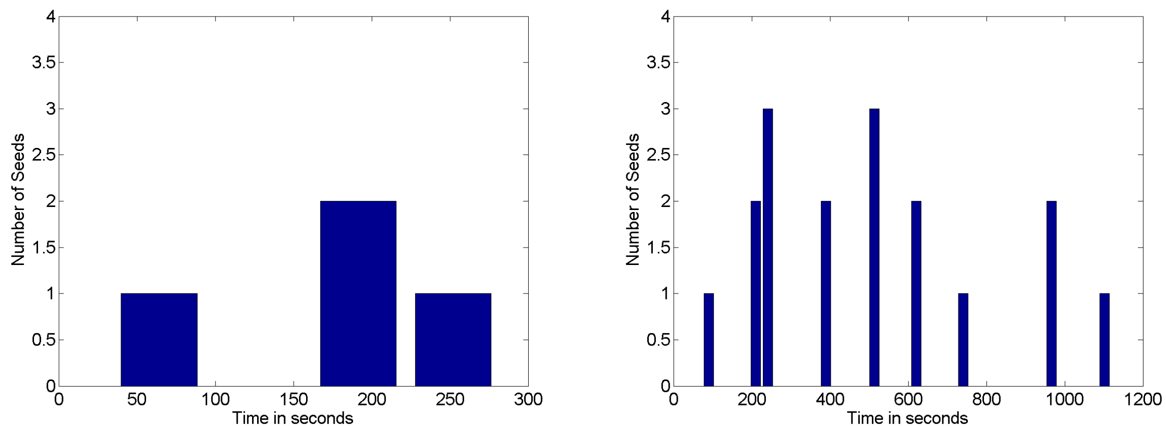

Figure 5.9: **LEFT:**Number of Seeds varying with time for MSI of 60. **RIGHT:**Number of Seeds varying with time for MSI of 180. It clearly shows that with a higher MSI, the number of seeds may grow unchecked to almost three before the broadcasted message is sent. With a MSI of 60, the seeds are regularly broken down to release more free robots at a faster rate than with an MSI of 180. This aids faster assembly time

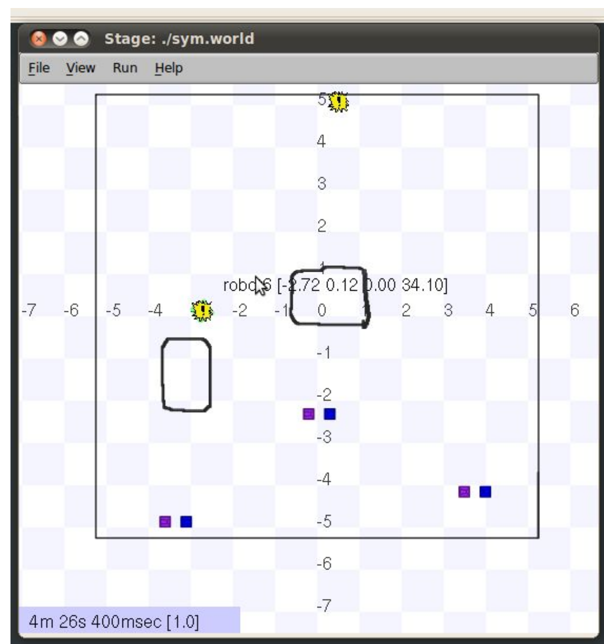

Figure 5.10: Player/Stage simulation with 3 parallel (purple and blue) structures being assembled in the environment for MSI of 180 seconds.

5.2 Discussion

We have shown that reversible stochastic assembly can lead to probabilistic completeness of self-assembly even when the number of building blocks are limited. We also showed that this process can be sped up using broadcast communication. Yet, we note that the intrinsic stochasticity still leaves room for outliers, i.e., cases where assembly takes longer than expected (see Figures 5.6 and 5.7). Having the capability to communicate, however, would also enable deliberative algorithms such as [10, 3, 11] that have deterministic performance. Nevertheless, we expect the proposed stochastic algorithms to be more robust and possibly even faster in robots that localize using dead-reckoning and landmarks and can only determine their position with certainty when they physically connect with each other. In fact, even if the REPLICATOR modules would be equipped with a global localization system, this system would be subject to measurement noise and make the system probabilistic, requiring additional error handling routines in the planning system to recover from such errors. We encountered such issues with our Player/Stage simulation. We plan to further investigate these trade-offs between robustness vs. completion time experimentally using the Robot3D simulator and eventually a team of REPLICATOR robots in future work.

Chapter 6

Conclusion

We designed a decentralized algorithm for self-assembly of robotic modules with 100% yield in the case of a limited number of building blocks in the environment. We proved that the proposed Reversible Single Growth Point (RSGP) algorithm will eventually lead to a complete structure given enough time. The Accelerated Self-Assembly (ASA) algorithm subsequently improves on the RSGP algorithm by accelerated disassembly of partially formed structures by utilizing wireless communication from larger structures. The results are tested on different environments. For Matlab simulations, we use a complex hexapod structure for 1.) different ratios of forward and reversal rules, 2.) message sending intervals, and 3.) message propagation probabilities. For Player/Stage simulations, we use a simple snake structure for 1.) different message sending intervals 2.) different P_{split} probability.

Our results show that it is indeed possible, under a range of these parameters, to self-assemble successfully into the given complex structure even if the number of building blocks in the environment are severely limited. We also show that the assembly rate can be improved with communication.

Bibliography

[1] Christos Ampatzis, Elio Tuci, Vito Trianni, Anders Lyhne Christensen, and Marco Dorigo. Evolving self-assembly in autonomous homogeneous robots: Experiments with two physical robots. Artif. Life, 15:465–484, October 2009.

[2] N. Bhalla, P. Bentley, and C. Jacob. Evolving physical self-assembling systems in two-dimensions. Evolvable Systems: From Biology to Hardware, pages 381–392, 2010.

[3] Z. Butler and D. Rus. Distributed planning and control for modular robots with unit-compressible modules. The International Journal of Robotics Research, 22(9):699–715, 2003.

[4] N. Correll and A. Martinoli. Modeling self-organized aggregation in a swarm of miniature robots. The International Journal of Robotics Research. Special Issue on Stochasticity in Robotics and Biological Systems. To appear., 2011.

[5] Brian P. Gerkey, Richard T. Vaughan, and Andrew Howard. The player/stage project: Tools for multi-robot and distributed sensor systems. In In Proceedings of the 11th International Conference on Advanced Robotics, pages 317–323, 2003.

[6] Y.M. Kao and P.G. Luan. Poincaré cycle of a multibox ehrenfest urn model with directed transport. Physical Review E, 67(3):031101, 2003.

[7] J. Kelly and H. Zhang. Combinatorial optimization of sensing for rule-based planar distributed assembly. In Intelligent Robots and Systems, 2006 IEEE/RSJ International Conference on, pages 3728–3734. IEEE, 2006.

[8] E. Klavins, R. Ghrist, and D. Lipsky. A grammatical approach to self-organizing robotic systems. Automatic Control, IEEE Transactions on, 51(6):949–962, 2006.

[9] Eric Klavins. Automatic synthesis of controllers for distributed assembly and formation forming. In ICRA, pages 3296–3302, 2002.

[10] K. Kotay and D. Rus. Generic distributed assembly and repair algorithms for self-reconfiguring robots. In Intelligent Robots and Systems, 2004.(IROS 2004). Proceedings. 2004 IEEE/RSJ International Conference on, volume 3, pages 2362–2369. IEEE, 2004.

[11] H. Kurokawa, K. Tomita, A. Kamimura, S. Kokaji, T. Hasuo, and S. Murata. The international journal of robotics research. Distributed Self-reconfiguration of M-TRAN III Modular Robotic System, 27(3–4):373–386, 2008.

[12] Igor Litovsky, Yves Métivier, and Wieslaw Zielonka. The power and the limitations of local computations on graphs. In WG, pages 333–345, 1992.

[13] M. Mastrangeli, G. Mermoud, and A. Martinoli. Modeling self-assembly across scales: The unifying perspective of smart minimal particles. Micromachines, 2(2):82–115, 2011.

[14] L. Matthey, S. Berman, and V. Kumar. Stochastic strategies for a swarm robotic assembly system. In Proc. of the IEEE Int. Conf. on Robotics and Automation (ICRA), pages 1953–1958, Kobe, Japan, 2009.

[15] R. Nagpal. Programmable self-assembly using biologically-inspired multiagent control. In Proceedings of the first international joint conference on Autonomous agents and multiagent systems: part 1, pages 418–425. ACM, 2002.

[16] N. Napp, S. Burden, and E. Klavins. The statistical dynamics of programmed self-assembly. In Robotics and Automation, 2006. ICRA 2006. Proceedings 2006 IEEE International Conference on, pages 1469–1476. IEEE, 2006.

[17] N. Napp, S. Burden, and E. Klavins. Setpoint regulation for stochastically interacting robots. In Proceedings of Robotics: Science and Systems, Seattle, USA, June 2009.

[18] Jennifer Owen. How to Use Player/Stage 2nd Edition, 2010.

[19] Vijeth Rai, Anne van Rossum, and Nikolaus Correll. Self-assembly of modular robots from finite number of modules using graph grammars. In Intelligent Robots and Systems (IROS), 2011 IEEE/RSJ International Conference on, pages 4783 –4789, sept. 2011.

[20] M. Rubenstein, Y. Sai, C. Chuong, and W. Shen. Regenerative patterning in swarm robots. International Journal of Developmental Biology, 53(5–6):869–881, 2009.

[21] M. Rubenstein and W.M. Shen. Scalable self-assembly and self-repair in a collective of robots. In Intelligent Robots and Systems, 2009. IROS 2009. IEEE/RSJ International Conference on, pages 1484–1489. IEEE, 2009.

[22] E. Tuci, R. Gross, V. Trianni, F. Mondada, Michael Bonani, and Marco Dorigo. Cooperation through self-assembly in multi-robot systems. ACM Transactions on Autonomous and Adaptive Systems, 1(2):115–150, 2006.

[23] A.C. van Rossum and H.J. van den Herik. Designing robot metamorphosis. In 22nd Benelux Conference on Artificial Intelligence (BNAIC 2010), Luxembourg, Luxembourg, 2010.

[24] P. White, V. Zykov, J. Bongard, and H. Lipson. Three dimensional stochastic reconfiguration of modular robots. In Proceedings of robotics science and systems, pages 161–168. Citeseer, 2005.

[25] PJ White, K. Kopanski, and H. Lipson. Stochastic self-reconfigurable cellular robotics. In Robotics and Automation, 2004. Proceedings. ICRA'04. 2004 IEEE International Conference on, volume 3, pages 2888–2893. IEEE, 2004.

Appendix A

Player/Stage

A.1 Player/Stage Background

A.1.1 Player:

Player is a network server for robot control providing a hardware abstraction layer for a clean and simple interface to the robot's sensors and actuators over the IP network. We use Client programs to communicate with Player over a TCP socket, reading data from sensors, writing commands to actuators, and configuring devices on the fly.

There are 3 key concepts in Player:

(1) **interface** : A specification of how to interact with a certain class of robotic sensors, actuators, or algorithm. The interface defines the syntax and semantics of all messages that can be exchanged with entities in the same class. This interface would be same for all devices of the same class, irrespective of their individual make.

(2) **driver** : A piece of software that talks to a robotic sensor, actuator, or algorithm, and translates its inputs and outputs to conform to one or more interfaces. The driver's job is to hide the specifics of any given entity by making it appear to be the same as any other entity in its class.

(3) **device** : A driver bound to an interface, and given a fully-qualified address. Messaging in Player occurs among devices, via interfaces. The drivers, while doing most of the work, are never accessed directly.

Standard Player drivers are kept as plug-in modules. We specify the drivers related to our robot's devices in our configuration file which loads the specific modules with some parameters. Player communicates with specific devices using device drivers, but provides to its clients a standard device interface. This allows clients to be portable to other robots.

A.1.2 Stage

Stage[1] is also a plugin to Player which provides Player with a simulated environment in absence of an actual robot hardware. Stage simulates sensor data and sends this to Player which in turn makes the sensor data available to client code. Player uses Stage in the same way it uses a robot; it thinks that it is a hardware driver and communicates with it as such.

Player uses a Server/Client structure[18] in order to pass data and instructions between the client code and the robot's hardware. The configuration file associated with robot(or the simulation) takes care of telling the Player server which devices are attached to it. So when we run the player server with this configuration file, the Player server connects all the necessary hardware devices to the server. A hardware device on the robot is subscribed as a client to the server via a "proxy". Once subscribed, we can use these proxies to control individual devices of the robot. Figure 7.1 shows a basic block diagram of the structure of Player when implemented on a robot. Note that there could be several proxies, each controlling and retrieving information from sensors and other devices.

Player server could also be connected to Stage environment instead of actual hardware in which case the design structure would be as shown in Figure 7.2

Thus our simulation is composed of three parts:

- Client Program: This communicates to Player server to control the robot.
- Player: This takes the client code and sends instructions to a robot. From the robot it gets sensor data and sends it to the client code.

[1] Stage and Player are freely available under the GNU General Public License from http://playerstage.sourceforge.net.

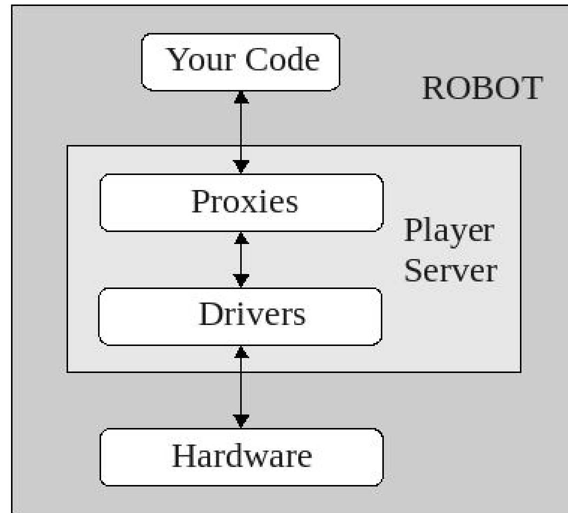

Figure A.1: Player Server Client model applicable when used with an actual robot.

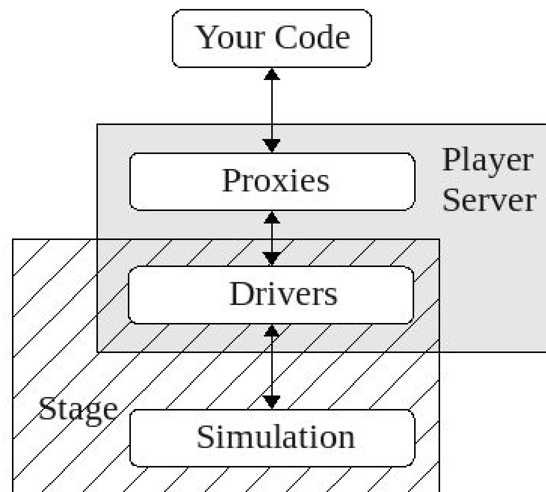

Figure A.2: Player Server model with Stage simulator instead of actual hardware.

- Stage: Stage interfaces with Player in the same way as a robot's hardware would. It receives instructions from Player and moves a simulated robot in a simulated world; it gets sensor data from the robot in the simulation and sends this to Player.

Player Features

The features that make Player/Stage an appropriate choice are:

- Player is designed to be language and platform independent: The client program can run on any machine that has a network connection to your robot, and it can be written in any language that supports TCP sockets.

- Player allows multiple devices to present the same interface: Thus the same control code could drive two different kinds of robot with same set of devices even if the the actual type/brand is different. This feature often allows control programs written for Stage's simulated robots to work unchanged on real hardware.

- Player is designed to support virtually any number of clients.

A.2 How Player Works

User programs are usually linked with client libraries, the client libraries manage the communication with the player server and provides an interface to the programs. The same programs can be used for the simulators and for the real robot. The configuration file of the Player server defines how the server talks with the robot hardware (in which ports are the devices attached, etc.) or the simulator. The simulator configuration file defines how to simulate the robot, how the simulated world map looks like, etc. Since Player provides standard interfaces to the client, the client operates the same no matter what driver is loaded into Player, so long as those drivers provide the required device interfaces. Player defines a set of standard interfaces, each of which is a specification of the ways that you can interact with some class of devices. For example the **position2d** interface covers ground-based mobile robots, allowing them to accept commands to make them move and to

report their state (current velocity and position). Player also provides transport mechanisms that allow data to be exchanged among drivers and control programs that are executing on different machines.The most common transport in use now is a client/server TCP socket-based transport.

A.3 Building the Environment:

There are 3 files needed for Player/Stage to initiate and build the simulation environment.

(1) **.world file**: This specifies the simulation world defining the layout, terrain, map and starting positions of the robots and other items.

(2) **Robot .inc file**: This is similar to the world file but it can be **included**. Hence it is used to specify objects that would be duplicated in the simulated world. We use this to define the robot shape and the devices attached to it with their relative positions

(3) **.cfg file**: This is the main Player configuration file defining the drivers to be used for the robot specified in the .world file. Client code uses these drivers to communicate with the robots and other items in the world. In the case where stage is used, the drivers communicate with the simulated devices and sensors.

A.3.1 World file and Robot Specs

A.3.1.1 Models

A world file is basically just a list of models that describes all the components in the simulation. This includes the basic environment, robots and other objects. The basic type of model is called “model”, and is defined as:

```
define model\_name model
(
\# parameters
)
```

This syntax can be used to define the floor plan of the simulated environment and the parameters such as size, boundary etc are specified in the parameters section. This syntax is also used for defining robot sensors as described in section below.

A.3.1.2 Building a Robot

In Player/Stage a robot is just a slightly advanced kind of model, with specific parameters.

Sensors and Devices

There are six built-in kinds of model that help with building a robot and are used to define the sensors and actuators that the robot has. These are associated with a set of model parameters which define aspects such as which sensors the model can be detected by, their color, their size etc. Each of these built in models acts as an interface between the simulation and Player.For the purpose of our simulation we use only a few of these sensors.

Blobfinder

This simulates a camera attached to the robot with color detection software. The blobfinder can only find a model if its blob_return parameter is true. The parameters for the blobfinder are:

- _ colors_count $\langle int \rangle$: the number of different colors the blobfinder can detect.
- _ colors [ ]: the names of the colors it can detect. This is given to the blobfinder definition in the form [“black” “blue” “cyan”]. These color names are from the built in X11 color database rgb.txt built into Linux.
- _ image [x y]: the size of the image from the camera, in pixels.
- _ range $\langle float \rangle$: The maximum range that the camera can detect, in meters.
- _ fov $\langle float \rangle$: field of view of the blob_nder in RADIANS.

Ranger

This simulates all variety of obstacle detection device. These can locate models whose ranger_return is true. Using a ranger model you can define any number of ranger sensors and

apply them all to a single robot. The parameters for the ranger model and their inputs are:

- _ scount ⟨*int*⟩ : The number of ranger sensors in this ranger model
- _ spose[ranger_number] [x y yaw]: Tells the simulator where the rangers are placed around the robot.
- _ ssize [x y]: how big the sensors are.
- _ sview [min max fov]: defines the maximum and minimum distances that can be sensed and also the field of view (fov) in DEGREES.

Position

The position model simulates the robot's odometry. This robot model is the most important of all because it allows the robot model to be embodied in the world, meaning it can collide with anything which has its obstacle_return parameter set to true. The position model uses the position2d interface, which is essential for Player because it tells Player where the robot actually is in the world. The most useful parameters of the position model are:

_ drive: Tells the odometry how the robot is driven. This is usually "diff " which means the robot is controlled by changing the speeds of the left and right wheels independently. Other possible values are "car" which means the robot uses a velocity and a steering angle, or "omni" which means it can control how it moves along the x and y axes of the simulation.

_ localization: tells the model how it should record the odometry "odom" if the robot calculates it as it moves along or "gps" for the robot to have perfect knowledge about where it is in the simulation.

_ odom_error [x y angle]: The amount of error that the robot will make in the odometry recordings.

A.3.2 Configuration File

The configuration file is needed in order to tell the Player server which drivers to use and which interfaces the drivers will be using. For each model in the simulation (or device on the robot)

that we wish to interact with, we will need to specify a driver. The driver specification is in the form:

driver (

name "driver_name"

provides [device_address]

other parameters...)

- The "name" specifies to the Player which standard driver to be used for the device. It must be the name of one of Player's inbuilt drivers that have been written for Player to interact with a robot device. When using Stage the name would be "stage" as there are no actual devices for the drivers to bind to.
- The "provides" parameter tells what kind of information driver would be providing. It is here that you tell Player what interface to use in order to interpret information given out by the driver which is most often the sensor information from a robot, or the information the "stage" driver is simulating. The input to the provides parameter is a "device address", which specifies which TCP port an interface to a robot device can be found. This uses the key:host:robot:interface:index form separated by white space.
- The "model" parameter is only used if "stage" is the driver. It tells Player which particular model in the worldfile is providing the interfaces for this particular driver. A different driver is needed for each model used. Models that aren't required to do anything (such as a map) don't need to have a driver written for them.

Device Addresses -key:host:robot:interface:index

A device address would inform the Player server about where the driver will present or receive information. It also tells which interface to use to read this information. This is a string in the form key:host:robot:interface:index where each field is separated by a colon.

_ host: This is the address of the host computer where the device is located. With a robot

it could be the IP address of the robot. The default host is “localhost” which means the computer on which Player is running.

_ robot: this is the TCP port through which Player should expect to receive data from the interface usually a single robot and all its necessary interfaces are assigned to one port. The default port used is 6665.

_ interface: The interface to use in order to interact with the data. There is no default value for this option because it is a mandatory field.

_ index: If a robot has multiple devices of the same type, for instance it has 2 cameras to give the robot depth perception, each device uses the same interface but would use different index to differentiate between the data provided.

Hence we would finally have something like this in our .cfg file:

driver (name “stage” provides [“6665:position2d:0” “6665:ranger:0” “6665:blobfinder:0”] model “robot1”)

Using the world file and the configuration file, we would have the simulation window as show in Figure 7.3

A.4 Client Code:

A.4.1 Player Client and Proxies

We use the code **PlayerClient robot_Client(hostname, port)** to declare a new object which is a PlayerClient called robot_client which connects to the Player server at the given address. Once we have established a PlayerClient we connect our code to the device proxies to exchange information with the devices. Proxies take the name of the interface which the drivers use to talk to Player. In our configuration file, we use the position2d, ranger, blobfinder and opaque interfaces. In our code, we hence connect to the position2d, ranger blobfinder and opaque proxies.

Each proxy is specialized towards controlling the device it connects to, with different commands to control and retrieve data from the device.

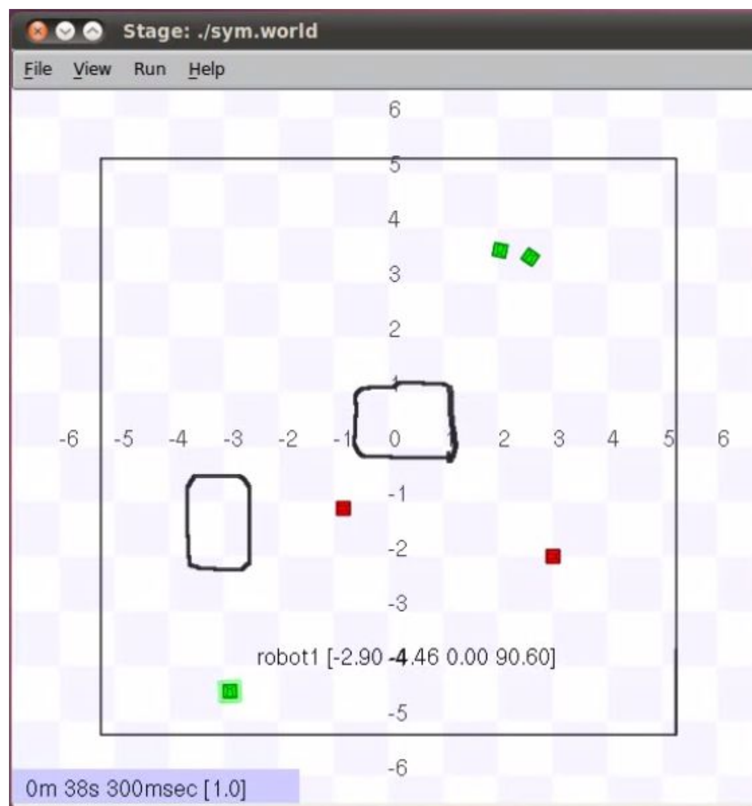

Figure A.3: Simulation window with robots and map initialized

A.4.1.1 Position2dProxy

This controls the robot's motors and keeps track of the robot's odometry. It has several in built functions but the most useful ones are:

SetSpeed: The SetSpeed command is used to tell the robot's motors how fast to turn.

Get Pos: This function is used to retrieve the GPS position of the robot in x,y and yaw terms.

A.4.1.2 RangerProxy

The ranger proxy can be used to receive the distance from the ranger to an obstacle in the environment, in meters. As mentioned before, it only detects the objects whose ranger return is set to true. To retrieve the distance to the obstacle in front of the ranger, we use:

rangerProxy_name[ranger_number] : where ranger number would be the index number of the particular ranger sensor in ranger array.

A.4.1.3 BlobfinderProxy

The blobfinder module analyses a camera image for areas of a desired color and returns an array of the structure playerc_blobfinder_blob_t. This structure is used to store the data about the colored object in the camera field of view.

A.4.1.4 OpaqueProxy

The Opaque proxy is one of the proxies useful for implementing communication in Player server. Since our algorithms need broadcasting, we decided to use Opaque Proxy. Opaque driver publishes any message posted, to all the clients subscribed to it. As opposed to other proxies, where each robot has its own set of devices, Opaque proxy is connected to just one single device for the whole environment. In our client code, we have to get our robot clients to subscribe to this single Opaque device. Thus, whenever any of the robots posts a message to this Opaque proxy, all the robots would receive the message.

A.4.1.5 SimulationProxy

The simulation proxy allows our code to interact with and change aspects of the simulation, such as an item's pose or its color. This is used in our implementation to change color of our robots to reflect change in their state.

A.4.2 Reading data

To make the proxies update with new sensor data, we need to tell the player server to update and we can do this using the PlayerClient object which we used to connect to the server. All we have to do is run the command robot_Client.Read() every time the data needs updating. Until this command is run, the proxies and any sensor information from them will be empty.

A.4.3 Simulating Multiple Robots:

There are several ways to simulate multiple robots.

- To use different TCP port address for each robot. For eg 6665 for one robot and 6666 for the next and so on.
- To use same TCP port address for all robots but use different **index** for the devices. This would work only when all the robots are similar. The robots use only one index for each of its devices. For e.g,
  - ∗ driver(name "stage" provides ["6665:position2d:1" "6665:ranger:1" "6665:blobfinder:1"] model "robot1")
  - ∗ driver(name "stage" provides ["6665:position2d:2" "6665:ranger:2" "6665:blobfinder:2"] model "robot2")

The second method allows us to update all the sensors of all the robots using a single read command to the client 6665. We then connect to particular robot's proxy using its index. For eg

Position2dProxy positionProxy_name(&robot_Client,index);