Robust by Composition: Programs for Multi-Robot Systems

Nils Napp, Eric Klavins
Electrical Engineering
University of Washington
Seattle, WA
{nnapp,klavins}@uw.edu

I. Introduction

***Abstract*— This paper describes how to specify the local reactive behavior of robots via *guarded command programs with rates*. These programs express concurrency and can be composed easily. Rates allow programs to be interpreted as Markov processes, which we use to define an appropriate notion of robustness and performance. We use composition to "robustify" programs with good performance, i.e. create a robust program with good performance from a program that has good performance but is not robust. We demonstrate this approach on a sub process of a reconfiguration program in a multi-robot system.**

A. Multi-robot Systems

Loosely coupled multi-robot systems are concurrent: The independent actions of different robots can modify the global state simultaneously. Concurrency is a result of the very features that make multi-robot systems attractive from an engineering perspective: modularity and autonomy of modules. Robots can be maintained, programmed, or replaced individually. As a result, tools for designing and reasoning about concurrent algorithms are of central importance when building and programming multi-robot systems. Especially, since such tools have been studied mostly in the context of software systems.

Here, we introduce abstract models of concurrent systems and use them to reason about programs for multi-robot systems. In addition to *performance*, we are primarily interested in the idea of *robustness* of such programs, for two reasons. First, algorithms that operate on physical systems need to be robust to some types of uncertainty in order to operate reliability. Second, multi-robot systems in particular have the potential for reliability through redundancy, but programs need to take advantage of this feature.

A robust multi-robot system should behave correctly and reject disturbances, such as the failure of an individual robot. While designing robust controllers is well understood in the context of other systems, such as linear time invariant ones [1], writing robust programs for multi-robot systems is an active area of research. In large part the difficulty in designing robust algorithms stems from the concurrency in such systems, due to the combinatorial explosion of possible combinations of local disturbances.

The main contribution of this paper is a way to "robustify" programs for a class of concurrent multi-robot systems by taking advantage of the natural program composition that concurrent systems provide – multiple programs can run at the same time. By interpreting programs as Markov processes the notion of robustness to module failure is preserved during composition. We exploit this feature and show how to compose a high performance program with a robust program to create a new program that is both robust and has good performance.

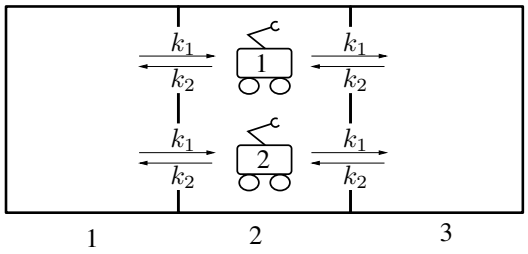

Fig. 1. Schematic representation of a very simple multi-robot system. The system consists of two robots and three rooms. Robots can only move between adjacent rooms. Each room change has an associate *rate*, k_i (see Sec. II-D).

B. Related Work

Some examples of loosely coupled multi-robot systems and their associated applications are: Formation control and estimation [2], self-assembly [3], [4], [5], and reconfiguration [6], [7]. This paper uses a model roughly similar to [5] where the system behaves according to stochastically applied local actions. However, we use a much more general framework called *guarded command programming* borrowed from the computer science literature [8].

Programs for the robotic system presented in Sec. IV are written in the Computation and Control Language (CCL), a particular implementation of a guarded command language that can easily interface to robotic hardware [9]. After adding probabilistic rates to the guards the result is similar to the specification language for the probabilistic model checker PRISM [10].

Framing the problem of programming multi-robot systems as a question of finding transition rates of Markov processes that result from local interactions is similar to [11][12]. We have addressed other control related questions about multi-robot systems such as performance optimization [3], and set-point regulation [13] in the context of such models. In [14] we give a more detailed description of the guarded command framework, with an emphasis on composition and

other operations on programs. Here we turn our attention to the idea of robustness for multi-robot systems that are well modeled as Markov processes.

C. Outline

The next section describes the connection between guarded command programs and Markov processes. Sec. II-A describes the syntax of guarded command programs with rates, Sec. II-D collects some results about Markov processes needed to define the semantics of guarded command programs with rates in Sec. II-E.

In Sec. III, III-A and III-B lay the groundwork for defining robustness and performance in the context of the concurrent multi-robot systems under discussion and III-C develops two theorems about program composition.

Sec. IV is an extended example about a distributed multi-robot reconfiguration testbed. Thm. 5 is applied to a subprogram in a reconfiguration task to show how one can create robust programs with good performance by composing a high performance program with a robust one.

II. Interpreting Programs as Markov Processes

This section describes the connection between programming and modeling concurrent multi-robot systems using guarded command programs with rates and Markov processes. Sec. IV uses the guarded command programing language CCL, but the concepts behind guarded command programming are abstract and by keeping the discussion general and using Markov processes to formulate Thm. 4 and 5 our approach is not tied to a particular testbed or language.

A. Syntax of Guarded Command Programs with Rates

Let S denote the state space of a concurrent system. For example, the positions, orientations, and internal states of the robots. A guarded command program with rates Ψ is a set of *rules*

$$r = (g, a, k),$$

each composed of a *guard* g, an *action* a , and a *rate* k. A guard is a predicate on the state space $g \subseteq S$, an action is a relation on the state space $a \subseteq S \times S$, and a rate is a positive real number, the significance of which is described in Sec. II-E. For a given rule $r \in \Psi$ its action a_r is subset of all the physically possible actions, denoted by $\mathcal{A} \subset S \times S$. The guards are used to program a system and restrict the set of all possible actions from a given state to a smaller set of desirable actions. This interpretation is similar to the way guarded commands work in [8], [9].

Example 1. In the system described in Fig. 1 the state space of the system is $S = \{1, 2, 3\} \times \{1, 2, 3\}$ where the first coordinate corresponds to the position of robot 1 and the second to the position of robot 2. For a given state $s \in S$ let s_1 denote first coordinate and s_2 the second. The set of physically possible action is given by

$$\mathcal{A} = \{(s, s') \in S \times S \mid \ |s_1 - s'_1| + |s_2 - s'_2| = 1\}.$$

Robots can only move one field at a time. The rule for robot 1 moving right is given by $r_1 = (g_1, a_1, k_1)$ where

$$\begin{aligned} g_1 &= \{s \in S \mid s_1 \neq 3\} \\ a_1 &= \{(s, s') \in S \times S \mid s_2 = s'_2,\ s_1 + 1 = s'_1\ \} \end{aligned}$$

and k_1 is some positive real number. The rule corresponding to the left moves $r_2 = (g_2, a_2, k_2)$ is given by

$$\begin{aligned} g_2 &= \{s \in S \mid s_1 \neq 1\} \\ a_2 &= \{(s, s') \in S \times S \mid s_2 = s'_2,\ s_1 = s'_1 + 1\} \end{aligned}$$

and k_2 is the a positive real number. The program for robot 1 is $\Psi_1 = \{r_1, r_2\}$. Let Ψ_2 denote the program for the second robot, which is the same except the indices are reversed. ■

B. Local Guards and Actions

Distributed programs take advantage of the concurrent nature of multi-robot systems. Individual robots should not have to know about the *global* state, but instead sense and interact with only their *local* environment. In the context of guarded command programs this means that $\mathcal{A}$ and the guards are restricted to actions and predicates that correspond to local interactions and locally checkable guards [15]. The precise definition of what local means depend intimately on the particular system and capability of the robots. In general, distributed programs are scalable since adding more robots does not change the computational load of each one.

When guards and actions are local it is often possible to write rules in a way that makes their local nature more transparent by borrowing notation from chemical reactions. Expressions to the left of a reaction arrow (reactants) correspond to the guard of a rule. Expressions to the right (products) correspond to the outcome of an action applied to the state on the left hand side. Parts of the state space that do not appear as products right remain unmodified.

Example 2. The right and left moves of robots in Ex. 1 could be written as

$$(s_1 < 3) \quad \xrightarrow{k_1} \quad (s_1 + 1) \tag{1}$$

$$(s_1 > 1) \quad \xrightarrow{k_2} \quad (s_1 - 1). \tag{2}$$

These reactions are interpreted as follows: When $s_i < 3$ at a rate of k_1, s_i is replaced with $s_i + 1$, which is the same as the first rule in Ψ_1. Similarly, (2) corresponds to the second rule of Ψ_1. ■

This notation highlights the local aspect of rules. If the left and right sides are checks and physical actions an individual robot can perform, then the resulting program is distributed by construction and takes advantage of concurrency. The reconfiguration program in Sec. IV is represented this way.

C. Composition of Programs

A convenient feature of guarded command programs is that two programs with the same state space S can easily be composed. The *composition* of two programs Ψ_1 and Ψ_2 is a new program

$$\Psi_3 = \Psi_1 \cup \Psi_2 \tag{3}$$

that contains all the rules of its component programs. For a guarded command program with rates Ψ a *scaled* version of Ψ is defined as

$$\gamma\Psi = \bigcup_{r\in\Psi} \{(g_r, a_r, \gamma k_r)\} \tag{4}$$

for $\gamma \in \mathbb{R}^+$, where g_r,a_r, and k_r denote the guard, action, and rate associated with rule r respectively. This corresponds to speeding up or slowing down a program, depending on whether γ is greater or less than one.

Example 3. If Ψ_2 is the program that describes how robot 2 moves, $\Psi_3 = 2\Psi_1 \cup \Psi_2$ describes the concurrent behavior of both robots moving, with robot 1 transitioning twice as frequently as robot 2. ■

D. Markov Processes

The randomness in a stochastic process can model a number of different aspects of a multi-robot system. For example, by considering distributions instead of individual states, stochastic dynamics can be used to reason about whole sets of paths instead of individual trajectories. As a result, stochastic models capture the different possible interleavings of actions in concurrent systems. Stochastic models can also incorporate failure statistics gathered from physical systems. By explicitly incorporating empirical knowledge of component failure rates one can reason about the likelihood of global system failures. Stochastic models are also frequently used to express uncertainty or to abstract away the details of poorly understood dynamics. This paper focuses on modeling concurrency and failures. Also, the discussion is restricted to *Markov processes* for computational reasons. They have a rich set of tools for analyzing and simulating them [16].

1) Definitions: Programmed multi-robot system can be considered as a finite state, continuous time Markov process X_t with state space S. This paper uses the following notation. To aid expression in terms of linear algebra, fix some arbitrary enumeration of states and let $i \in S$ denote the ith element of S. Given two states $i, j \in S$ the *transition rate* from i to j is denoted by $k_{i,j} \in \overline{\mathbb{R}^+}$. When there is no transition between states, the transition rate between them is defined as zero. This way all pairs have a transition rate associated with them. Denote probability distributions on S by a vector $\mathbf{p}(t)$ with $\mathbf{p}_i(t) = P(X_t = i)$ and let $\mathbf{Q}$ be the matrix given by

- $\mathbf{Q}_{ji} = k_{i,j}$ for $i \neq j$
- $\mathbf{Q}_{ii} = -\sum_{j\neq i} Q_{ji}$.

The differential equation

$$\dot{\mathbf{p}} = \mathbf{Q}\mathbf{p} \tag{5}$$

is called the *master equation* [17]. The $\mathbf{Q}$-matrix in (5) governs the evolution of probability distributions. By analyzing its structure one can infer properties of the stochastic process, for example, the steady state distribution(s), the convergence rate, or the hitting times (see next section).

2) Properties of Markov Processes: This section collects some of the key results about finite state Markov processes. For a more complete treatment see, for example [16], [17].

The *hitting time* for some set $h \subseteq S$ is defined as

$$\zeta_h = \inf\{t \mid X_t \in h\},$$

and the return time as

$$\sigma_h = \inf\{t \geq \zeta_{h^C} | X_t \in h\},$$

where h^C denotes the complement of h. When the process starts outside of h, ζ_h and σ_h are the same, but when $X_0 \in h$ then $\zeta_h = 0$ while σ_h is the first time that the process returns to h after leaving it. A state $i \in S$ is called *recurrent* if $P(\sigma_{\{i\}} < \infty | X_0 = i) = 1$ and *transient* if it is not recurrent.

These probabilistic quantities are closely related to the structure of possible transitions. Considering a Markov process as a graph $G(\mathbf{Q})$ with directed, labeled edges highlights this connection. Define $G(\mathbf{Q})$ as the triple $G(\mathbf{Q}) \equiv (V, E, L)$ where the vertex set $V = S$ is the state space of X, the edge set E is given by $E = \{(i, j) \in S \times S \mid k_{i,j} > 0\}$, and the labeling function for edges $L : E \to \overline{\mathbb{R}^+}$ is given by $L(i, j) = k_{i,j}$. A state $j \in S$ is said to be *reachable* from state $i \in S$ if there exists a path from i to j in $G(\mathbf{Q})$. Two states i, j are said to *communicate*, denoted by $i \leftrightarrow j$, when i is reachable from j and j is reachable from i. The relation $\leftrightarrow$ is an equivalence relation that partitions S into equivalence classes. If an equivalence class h has the property that there are no states outside h that are reachable from states in h, then all states in h are recurrent.

Example 4. In Fig. 2a each state is an equivalence class. In Fig. 2b has equivalence classes $\{1, 2, 4\}$ and $\{3\}$. State 3 is recurrent in all the Markov processes, in Fig. 2c state 1 is also recurrent. ■

For finite state process with a given $\mathbf{Q}$-matrix and initial distribution $\mathbf{p}(0)$ the limit

$$\pi_{\mathbf{Q}} \equiv \lim_{t\to\infty} \mathbf{p}(t)$$

always exists. When a process has only a single recurrent communicating class, this limit is unique and $\pi_{\mathbf{Q}}$ is called the *steady state* distribution. Only recurrent states have positive probability in $\pi_{\mathbf{Q}}$. The second largest eigenvalue $\lambda_2(\mathbf{Q})$ is the worst case convergence rate to $\pi_{\mathbf{Q}}$ from any initial distribution.

The remainder of this section looks at the *connectivity* of graphs and its consequences for Markov processes. For a more thorough treatment see, for example [18, III.2]. A graph G is said to be *connected* if any two vertices have a path between them. Here, paths are considered to be directed. A connected graph G is said to be *separated* by a set of vertices $h \subset V$ if the graph induced by removing h is no longer connected. A graph is said to be $k-$connected if there is no set of $k-1$ vertices that separates it. The largest value of k for which a graph G is $k-$connected is called the *connectivity*, denoted $\kappa(G)$. For two sets $h, h' \subset V$ we define the *restricted connectivity* $\kappa(h, h')$ to be the largest k

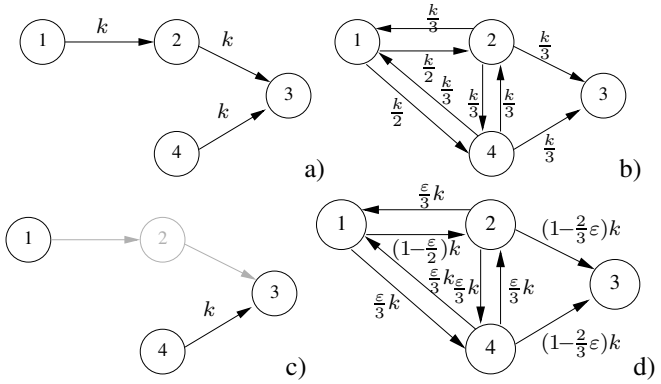

Fig. 2. Schematic representation of Markov processes as graphs. a) Markov process with a unique sink state and low connectivity, denote its $\mathbf{Q}$-matrix by A. b) Markov process with same sink state and high connectivity, denote its $\mathbf{Q}$-matrix by B. c) Markov process induced by removing state 2 from A. (d) Markov process formed by the composition $(1-\varepsilon)A + \varepsilon B$.

such that no set of $k-1$ vertices from h disconnects vertices in h from h', i.e. there is a path from every $s \in h$ to some vertex in h'. When comparing the connectivity for different graphs, a subscript to κ denotes the corresponding $\mathbf{Q}$-matrix.

The restricted connectivity $\kappa(h, h')$ measures the minimum number if independent paths to h' from states in h. Its utility in the context of multi-robot systems is that it can be used to quantify redundancy.

Example 5. In Fig. 2bd have $\kappa(\{1,2,4\},\{4\}) = 2$, while Fig. 2c shows the $\kappa(\{1,2,4\},\{3\})$ is smaller for a. The restricted connectivity from transient states to recurrent of these two processes obey the following relation

$$\kappa_B(\{1,2,4\},\{3\}) = 2 > \kappa_A(\{1,2,4\},\{3\}) = 1.$$

■

E. Semantics of Guarded Command Programs with Rates

The $\mathbf{Q}$-matrix and an initial probability distribution on S suffice to define a Markov process on S. Defining these two quantities for guarded command programs with rates thus allows us to treat them as Markov processes. For a guarded commands with rates Ψ the probability distributions are defined in the same way as for (5). To construct $\mathbf{Q}(\Psi)$ first define

$$R_{i,j} = \{r \in \Psi \mid s_i \in g_r, (s_i, s_j) \in a_r\}, \tag{6}$$

to be the set of rules make a transitions from state $s_i \in S$ to state $s_j \in S$. The entries of $\mathbf{Q}(\Psi)$ are defined as

- $\mathbf{Q}(\Psi)_{ji} = \sum_{r \in R_{i,j}} k_r$ for $i \neq j$
- $\mathbf{Q}(\Psi)_{ii} = -\sum_{j \neq i} \mathbf{Q}(\Psi)_{ji}$.

The dynamics of the system with program Ψ are given by the master equation (5).

Example 6. Continuing Ex. 1, with the enumeration $index(s) = (s_1) + 3(s_2 - 1)$ the associated $\mathbf{Q}$-matrix is the 9-by-9 matrix given by

$$\mathbf{Q}(\Psi_1) = \begin{pmatrix} \mathbf{Q}' & \mathbf{0} & \mathbf{0} \\ \mathbf{0} & \mathbf{Q}' & \mathbf{0} \\ \mathbf{0} & \mathbf{0} & \mathbf{Q}' \end{pmatrix}$$

where

$$\mathbf{Q}' = \begin{pmatrix} -k_1 & k_2 & 0 \\ k_1 & -(k_1+k_2) & k_2 \\ 0 & k_1 & -k_2 \end{pmatrix}.$$

Since Ψ_1 only contains actions affecting robot 1, there are three recurrence classes one for each position of robot 2, which is reflected in the block diagonal structure of $\mathbf{Q}$. ■

F. Composition of Markov Processes

Composition (3) and scaling (4) operations on guarded command programs with rates correspond directly to operations on their associated $\mathbf{Q}$-matrices.

Lemma 1. *Given any two guarded command programs with rates Ψ and Φ on the same state space, and two positive scalers $\alpha, \beta \in \mathbb{R}^+$, the following equation holds*

$$\mathbf{Q}(\alpha\Psi \cup \beta\Phi) = \alpha\mathbf{Q}(\Psi) + \beta\mathbf{Q}(\Phi). \tag{7}$$

Proof: Define the $R_{i,j}$ as in (6) for Ψ and $R'_{i,j}$ for Φ. For the off diagonal elements $i \neq j$ we have

$$\begin{aligned} & \alpha\mathbf{Q}(\Psi)_{ji} + \beta\mathbf{Q}(\Phi)_{ji} \\ = & \sum_{r \in R_{i,j}} \alpha k_r + \sum_{r' \in R'_{i,j}} \beta k_{r'} \\ = & \mathbf{Q}(\alpha\Psi \cup \beta\Phi). \end{aligned}$$

Since the diagonal elements are computed the off diagonal entries in each column (7) follows. □

This mapping of programs to $\mathbf{Q}$-matrices allows us to reason about new, composed programs by examining properties of the corresponding matrices.

After defining the syntax and semantics of guarded command programs with rates in this section, the next section gives some reasons for why this is a useful approach.

III. Robustness by Composition

Discussing robustness requires both a notion of *correct behavior* and a description of the disturbances a system is supposed to be robust to. If the system behaves correctly in the face of these disturbances it is said to be robust.

The idea of robustness pursued here is similar to the idea of fault tolerance in other engineered systems and provides a way to eliminate the probability of incorrect program executions. The idea is the make sure that even in the presence of disturbances actions that take the system to a desirable state are always enabled.

A. Correctness and Performance

The goal of our programs is to put a multi-robot system into a subset $h \subset S$ of the state space, called the *target*. A program Ψ is *correct* if all recurrent states of $\mathbf{Q}(\Psi)$ are in h and $\mathbf{Q}(\Psi)$ has only one recurrent equivalence class. To construct correct programs it is sufficient to make sure there are no transitions out of the target h that there exists some state in h that is reachable from every state $s \in S$.

The task of constructing correct program in this way is in general not trivial, since the state space of programs grows

exponentially with robots, also the set h can be difficult to express with local guards. We assume for the rest of this paper that we are able to create correct programs by some means.

Since correct behavior corresponds to the steady state $\pi_{\mathbf{Q}}$, *performance* can be interpreted as the rate of convergence to the steady state. From an arbitrary initial condition the rate of convergence is bounded by $\lambda_2(\mathbf{Q})$. All eigenvalues of a $\mathbf{Q}$-matrix are non-positive and when there is only a single recurrence class $\lambda = 0$ has multiplicity 1. In this case λ_2 is negative and the more negative it is the faster the process converges.

B. Disturbances

Two common types of disturbances considered in robust design are various types of noise and parameter uncertainty. In the context of guarded command programs with rates parameter uncertainty corresponds to uncertainty in the rate parameters associated with rules. However, as long as they are positive, changes in these parameters do not affect the structure of the equivalence classes of $\mathbf{Q}(\Psi)$. Uncertainty or noise in the rates therefore cannot break a correct program and are not a useful class of disturbances to consider in this context.

Instead, the disturbances considered in this paper are failures of individual robots. These failures can be modeled as removing states from S as well as transitions to and from from the states. A given fault is assumed to affect a subset of the states $h \subset S$. When a fault occurs the resulting Markov process with faults has a graph corresponding to the transition graph induced by removing the vertices h, i.e. all the edges to and from states in h as well as the vertices are removed. Removing vertices in this manner can change the recurrence structure, even if h does not intersect the target. The problem is that parts of the graph can become disconnected, and each of the resulting components can have one or more recurrence classes.

For a more thorough discussion of modeling errors and uncertainty with guarded command programs see [14].

C. Properties of Composition

This section gives two theorems about the composition of programs. Both are about the continuity of composition. Adding a small amount of an arbitrary program Ψ_2 to a nominal program Ψ_1 means that the composition will have a similar steady state and convergence rate as Ψ_1. The theorems are stated in terms of the $\mathbf{Q}$-matrices for generality. In Sec. IV we demonstrate how these theorems can be applied to programs for a particular multi-robot system.

Lemma 2. *Given two* $\mathbf{Q}$*-matrices* A *and* B *with the same dimension then* $\forall \delta > 0\ \exists\ \varepsilon > 0$ *such that*

$$|\lambda_2(A) - \lambda_2(A + \varepsilon B)| < \delta.$$

Proof: The eigenvalues of $A + \varepsilon B$ are the solutions of the characteristic polynomial, $\det(\lambda I - (A + \varepsilon B)) = 0$. The roots of a polynomial are continuous functions of the coefficients, which in turn are continuous functions of ε. By composition on continuous function the eigenvalues of $A + \varepsilon B$ depend continuously on ε. □

Lemma 3. *Given two* $\mathbf{Q}$*-matrices* A *and* B *with the same target* $h \subset S$*, then for* $C = A + B$

$$\kappa_C(h^C, h) \geq \max\{\kappa_A(h^C, h), \kappa_B(h^C, h)\}.$$

Proof: Adding two $\mathbf{Q}$-matrices can only increase the number of (vertex) independent paths from h^C to h , therefore $\kappa_C(h^C, h)$ is at least as large as $\kappa_A(h^C, h)$ and $\kappa_B(h^C, h)$. □

Theorem 4. *Given a* $\mathbf{Q}$*-matrix* A *with a single recurrent communicating class and some other* $\mathbf{Q}$*-matrix* B*, then* $C = (1-\varepsilon)A + \varepsilon B, \varepsilon \in [0, 1)$ *has a single recurrence class and the entries of* π^*_C *depend continuously on* ε*.*

Proof: In steady state the flux balance equations ($\mathbf{Qp} = \mathbf{0}$) and the probability vector constraint ($\mathbf{1}^T\mathbf{p} = 1$) can be written as a system of equations. We can multiply them and note that one of the roots of the resulting polynomial is the steady state probability distribution, the entries of which vary continuously with ε. Because of the assumptions A and C each have exactly one steady state distribution, so that varying ε cannot produce a bifurcation. □

Note that in general when $dim(Null(A)) > 1$ adding εB can make steady state solutions disappear, for example a transition in B could connect two different recurrent equivalence classes. In this case, the steady state from an given initial condition might not be continuous in ε. Including he condition on the number of recurrent classes in Thm. 4 is necessary.

Theorem 5. *Given two* $\mathbf{Q}$*-matrices* A *and* B *with the same, unique recurrent state* s^**, then for any* $\delta > 0\ \exists\ \varepsilon > 0$ *such that for* $C = A + \varepsilon B$

1) $|\lambda_2(A) - \lambda_2(C)| < \delta$,
2) $\kappa_C(\{s^*\}^C, s^*) \geq \kappa_B(\{s^*\}^C, s^*)$,
3) $\pi_C = \pi_A$.

Theorem 5 follows directly from applying Lem. 2 and 3.

Both Thm. 4 and 5 are about composing a nominal program Ψ with other programs. By Lem. 2 if Ψ is composed with some small about of an arbitrary program Φ then the convergence rate of $\Psi + \varepsilon\Phi$ is close to the convergence rate of Ψ. By Thm. 4 the steady state of Ψ and $\Psi + \varepsilon\Phi$ are also close element wise.

Thm. 5 is about composing programs that are correct (with respect to the same target), but have different performance and robustness. It states that one can add a robust (high relative connectivity) with a high performance ($|\lambda_2|$ large) program and obtain a new program that has both good performance and is robust.

IV. The Factory Floor Testbed

This section describes an extended example demonstrating how to apply Thm. 5 to the Factory Floor testbed, a multi-robot systems that can assemble, disassemble, and reconfigure structures [7]. The goal of this testbed is developing robust algorithms and hardware to autonomously build

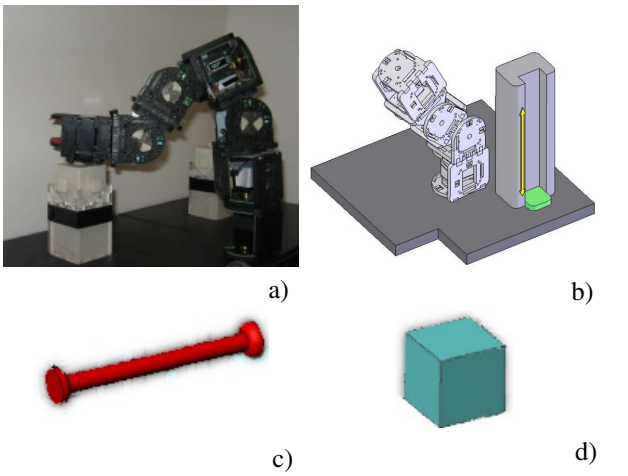

Fig. 3. Images of Factory Floor Module. a) Photo of Factory Floor module prototype. b) Rendering of digital prototype with labeled features. Rendering of both types of raw material, nodes c) and trusses (d).

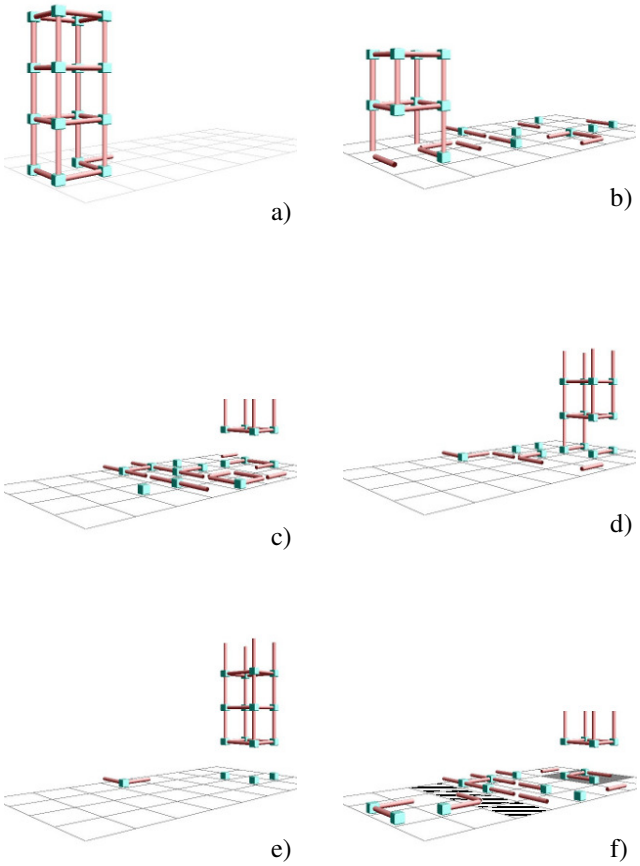

Fig. 4. Sequence of snap shots from a reconfiguration simulation. The sequence a)-e) shows the program at various stages of progress a) is the initial configuration and e) is the final goal configuration. Image f) shows the loading and target area of the routing program, see Fig. 5. To the left of the loading area a disassembly program takes a apart a structure and feeds to raw materials to the routing program.

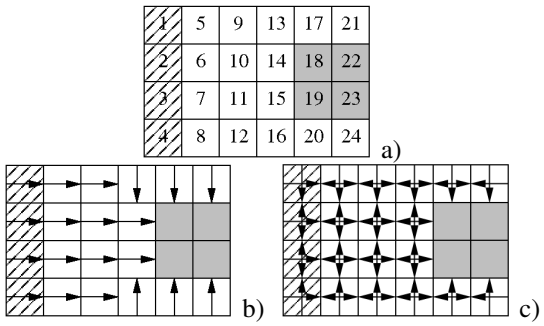

Fig. 5. Schematic representations of routing programs. The loading area L is denoted by hashed modules, the routing area R by white modules, and the target area T by grey modules.a) Layout of the factory floor testbed. The numbers in each module position identify the module when writing programs. b) Deterministic path program, only transitions that provide progress are enabled. c) Random Program, in each location all possible transitions are enabled. This program is slow since many of the transitions do not provide progress toward the target area.

structures in uncertain environments. For clarity, the program discussed in this section is comparatively simple. However, the the theorems and definitions carry over directly to more complicated programs such as the the one shown in Fig. 7.

A. Description of Testbed

The testbed consists of an array of identical robotic modules that build structures made from two different types *raw material* called *trusses* and *nodes*, see Fig. 3. Each module has a manipulator with an end effector that can grab and release nodes and trusses, a temporary storage place for nodes and trusses, and a lifting mechanism. Assembly and disassembly proceeds layer by layer. Modules manipulate raw materials into place and then coordinate lifting with other modules. The sequence of pictures in Fig. 4 shows a typical reconfiguration task. A tower is disassembled on one side of the testbed, the raw materials are routed across the testbed, and then a tower is assembled on the opposite side. Fig. 4 f) shows which modules in the testbed run the three different tasks.

B. Routing Programs

The remainder of this section describes programs for the *routing* portion of the reconfiguration task in more detail. This sub-task is both the task performed by the most modules and (partially because of it) the task that has the most redundancy, which is important to the idea of robustness to module failure. Programs can only be robust to individual robot failures if there are redundant robots that can take over the tasks of the failed robots. Fig. 5a gives the layout of the factory floor testbed for a configuration task. The hashed modules $\{1, 2, 3, 4\}$ corresponds to the loading area and grey modules to the target area $\{18, 19, 22, 23\}$. Each module either has a node or not, so the state of each module is in $\{true, false\}$. The state of a module is $true$ when the module has a node and $false$ when it does not. In the routing portion of a reconfiguration program what happens in the target area is not important. During routing they act as a sink, nodes disappear as soon as they reach the target.

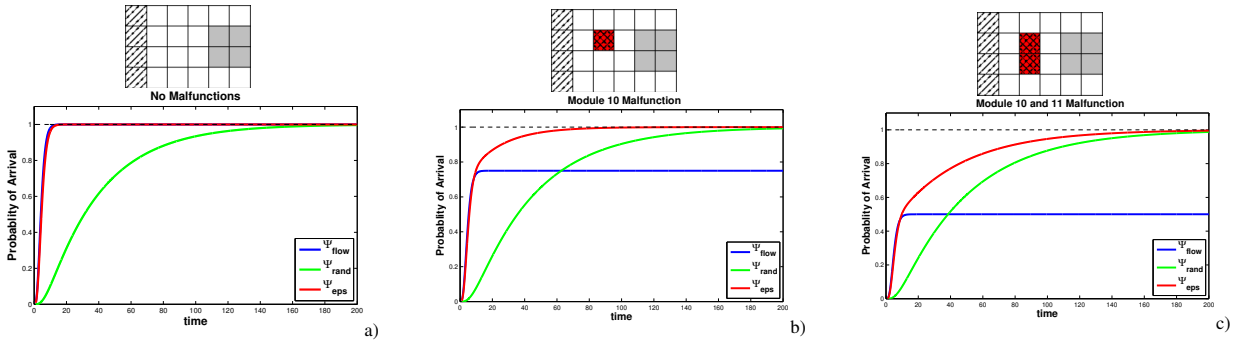

Fig. 6. Program performance with various failure scenarios. The plots how the probability of a node arriving at the target (given it was at the loading area at $t = 0$) as a function of time. The blue line corresponds to the program shown in Ψ_{flow} in Fig. 5b , the green line to the program Ψ_{random} shown in Fig. 5c, and the red line to the composed program $\Psi_\varepsilon = 0.9\Psi_{flow} + 0.1\Psi_{random}$. a) Programs with no module failures, Ψ_{flow} and Ψ have similar good performance and Ψ_ε has poor performance, all programs are correct. b) c) Programs with one and two module failures respectively. Only $\Psi_{epsilon}$ and Ψ_{random} are correct, they are robust to module failure.

Since there are 24 modules, the state space S for this example is $\{true, false\}^{24}$, the 24-fold cross product of the module state. In a state $s \in S$, the occupancy state of module i is denoted by s_i if the module is occupied and $\overline{s}_i$ if module i is empty. Fig. 5bc shows two different pictorial representations of routing programs. Each arrow corresponds to a guarded command, and each module runs a program containing all the rules corresponding to arrows in that location. Using a similar approach as in Sec. II-B programs for these reactive modules can be written like,

$$\begin{array}{ccc} s_3, \overline{s}_7 & \overset{k_{pass}}{\longrightarrow} & \overline{s}_3, s_7 \\ \overline{s}_3 & \overset{k_{load}}{\longrightarrow} & s_3 \end{array}$$

in the case of module 3 in program in Fig. 5b, for example.

The associated rates are such, the rates of all rules corresponding to actions performed by the same modules sum to a constant k_{pass}. This rate models the speed at which modules perform tasks, in this case how long it takes to pass raw materials. The average time to pass raw materials is $\frac{1}{k_{pass}}$. If there are multiple arrows, then each of the associated guarded commands has the same fraction of the total rate k_{pass}. Also, nodes appear randomly in the target area denoted by hashed modules, at a rate k_{load}. In this way the diagram Fig. 5bc can be turned into guarded command program.

To analyze the performance and robustness of programs we look at how well they route a single node. The reason for doing so is that this assumption drastically reduces the size of the state space, yet the connectivity properties that govern the robustness in the single node case carry over to the general case with multiple nodes. With this restriction the state space of the system is simply the position of the single node. When a module fails and stops routing nodes only a single state (but multiple transitions) become unavailable. The same approach also works when the state space is more complicated, but the relationship between the number of failures and the connectivity of the program is not as straight forward because a single failure might remove multiple states.

In this simplified problem a node randomly appears in the *loading area*, either from an external source or, as in this example, from a disassembly program. The target area is modeled as a single state that only accepts nodes. When all modules operate correctly it is easy to see that nodes will be routed to the single accepting sink state, the *target area*. The rate of convergence (λ_2) of the two programs Ψ_{flow} and Ψ_{random} shown in Fig. 5 is -1.0 and -0.029 respectively.

The program Ψ_{random} has better performance, but when any of routing modules fail the resulting **Q**-matrix has multiple recurrent equivalence classes with some recurrent states outside the target. In contrast program Ψ_{flow} has the opposite problem, it has low performance, due to the backward passes that do not provide progress, but it performs correctly when up to three modules fail. This is as well was one can hope to do since removing four modules can separate the routing area into two parts. Then there are no possible paths from the loading area to the target area.

By Thm. 5 we can combine both programs and obtain one that is both robust and has good performance. For example, choosing $\varepsilon = 0.1$ the program $\Psi_\varepsilon = (1-\varepsilon)\Psi_{flow} + \varepsilon\Psi_{random}$ has $\lambda_2 = -0.66$ and the same robustness as Ψ_{random}.

Exactly how these programs fair under the failure scenarios is shown in Fig. 6. The top of each subfigure shows which modules have failed. The plot on the bottom shows the probability of arriving as a function of time, given that a node showed up in the loading area at time 0. With no failures Fig. 6a the three different programs behave as described above, all programs are correct and the Ψ_{flow} and Ψ_ε have good performance while the Ψ_{random} has bad performance. Failure scenarios where one and two modules fail are shown in Fig. 6bc. With failures only Ψ_{random} and Ψ_{flow} behave correctly. They are robust to failure. Program Ψ_ε also has good performance. By this process one can combine the desirable properties of robust and high performance programs via composition and scaling.

V. Conclusion

We describe how to use guarded command programs with rates to program and model multi-robot systems. These programs can be easily composed, represent the inherent

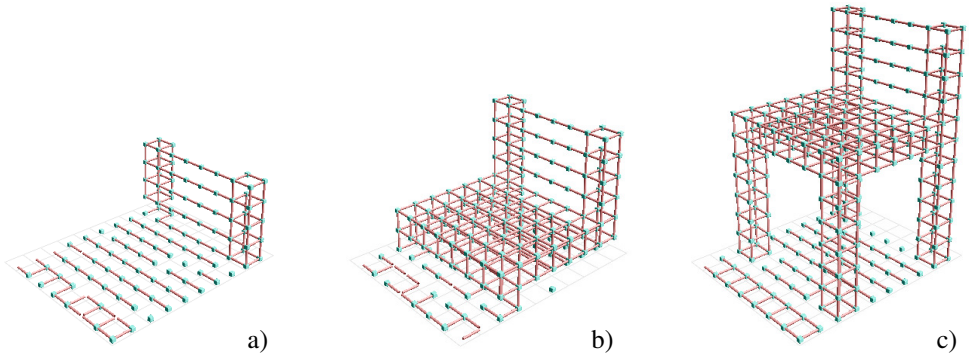

Fig. 7. Chair at various stages of completion. a) The back rest is emerging from the Factory Floor testbed. b) The seat has just finished. c) The program is done. There are no more applicable actions. The various components of the chair have their own subprograms, which run concurrently via composition. A simple broadcast protocol is used to communicate the current level.

concurrency of such system, and be interpreted as Markov processes. The Markov process description allows us to define notions of robustness and performance. Robustness is converging to the right state, even when some robots fail and performance is how fast the system converges. The contribution of this paper is showing how to create robust programs with good performance for a class concurrent multi-robots systems, namely those that are well modeled as Markov Processes.

In the future we would like to investigate alternative ways of writing programs for "robustification". The program Ψ_{random} applies all possible actions to provide high connectivity, however many of these actions move the node away from the target, which unnecessarily degrades performance. One question is how to construct programs that have high connectivity and good performance to begin with. Such programs should yield better results during robustification. Also, the naive approach of completely random exploration is likely to seriously degrade performance when the state space is very large.

Eventually, we want to apply robustification to more complicated programs such as the one shown in Fig. 7. While the theorems presented here certainly hold, applying them in effectively is no straight forward. For example, in Fig 7 there is little redundancy because almost all modules need to work to complete the seat of the chair. But in the routing for the back rest and legs the robustification we presented should work well. What are the implications of only robustifying subprograms? When is it a good idea?

We plan to implement these algorithms on the Factory Floor testbed. When more modules become available for experimentation we would like to measure and incorporate actual failure statistics into the robustification scheme.

REFERENCES

[1] G. E. Dullerud and F. Paganini, *A Course in Robust Control Theory: A Convex Approach*. Springer, 1999.

[2] J. McLurkin, "Dynamic task assignment in robot swarms," in *Robotics: Science and Systems I*, (Cambridge, MA), pp. 129–136, 2005.

[3] E. Klavins, S. Burden, and N. Napp, "Optimal rules for programmed stochastic self-assembly," in *Robotics: Science and Systems II*, (Philadelphia, PA), pp. 9–16, 2006.

[4] P. J. White, K. Kopanski, and H. Lipson, "Stochastic self-reconfigurable cellular robotics," *IEEE International Conference on Robotics and Automation (ICRA04)*, pp. 2888–2893, 2004.

[5] N. Napp, S. Burden, and E. Klavins, "The statistical dynamics of programmed self-assembly," in *Proc. IEEE International Conference on Robotics and Automation (ICRA) 2006*, pp. 1469–1476, 2006.

[6] D. Lobo, D. A. Hjelle, and H. Lips, "Reconfiguration algorithms for robotically manipulatable structures," in *Proc. of International Conference on Reconfigurable Mechanisms and Robots (ReMAR)*, ASME/IFToMM, June 2009.

[7] K. Galloway, R. Jois, and M. Yim, "Factory floor: A robotically reconfigurable construction platform," in *To Appear ICRA Proceedings*, 2010.

[8] E. W. Dijkstra, "Guarded commands, nondeterminacy and formal derivation of programs," *Commun. ACM*, vol. 18, no. 8, pp. 453–457, 1975.

[9] E. Klavins and R. M. Murray, "Distributed algorithms for cooperative control," *Pervasive Computing, IEEE*, vol. 3, no. 1, pp. 56– 65, 2004.

[10] M. Kwiatkowska, G. Norman, and D. Parker, "Probabilistic symbolic model checking with PRISM: A hybrid approach," *International Journal on Software Tools for Technology Transfer (STTT)*, vol. 6, no. 2, pp. 128–142, 2004.

[11] S. Berman, A. Halsz, M. A. Hsieh, and V. Kumar, "Optimized stochastic policies for task allocation in swarms of robots," *Robotics, IEEE Transactions on*, vol. 25, pp. 927–937, Aug. 2009.

[12] I. Chattopadhyay and A. Ray, "Supervised self-organization of large homogeneous swarms using ergodic projections of Markov chains," in *Proc. American Control Conference (ACC)*, pp. 2922–2927, 2009.

[13] N. Napp, S. Burden, and E. Klavins, "Setpoint regulation for stochastically interacting robots," in *Robotics: Science and Systems*, 2009.

[14] N. Napp and E. Klavins, "A compositional framework for programming stochastically interacting robots," *Submitted for Publication*, 2011.

[15] J. McNew, E. Klavins, and M. Egerstedt, "Solving coverage problems with embedded graph grammars," in *Hybrid Systems: Computation and Control*, pp. 413–427, Springer-Verlag, April 2007.

[16] D. W. Stroock, *An Introduction to Markov Processes*. Graduate Texts in mathematics, Springer, 1 ed., 2005.

[17] N. V. Kampen, *Stochastic Processes in Physics and Chemistry*. Elsevier, 3 ed., 2007.

[18] B. Bollobás, *Modern Graph Theory*. Springer, 1998.