April 2000

A formalism for the composition of concurrent robot behaviors

Eric Klavins
University of Michigan

Daniel E. Koditschek
University of Pennsylvania, kod@seas.upenn.edu

Follow this and additional works at: http://repository.upenn.edu/ese_papers

Recommended Citation


This material is posted here with permission of the IEEE. Such permission of the IEEE does not in any way imply IEEE endorsement of any of the University of Pennsylvania's products or services. Internal or personal use of this material is permitted. However, permission to reprint/republish this material for advertising or promotional purposes or for creating new collective works for resale or redistribution must be obtained from the IEEE by writing to pubs-permissions@ieee.org. By choosing to view this document, you agree to all provisions of the copyright laws protecting it.

NOTE: At the time of publication, author Daniel Koditschek was affiliated with the University of Michigan. Currently, he is a faculty member in the Department of Electrical and Systems Engineering at the University of Pennsylvania.
A formalism for the composition of concurrent robot behaviors

Abstract
We introduce tools which help one to compose concurrent, hybrid control programs for a class of distributed robotic systems, assuming a palette of controllers for individual tasks is already constructed. These tools, which combine the backchaining of continuous robot behaviors with Petri nets, expand on successful work in sequential composition of robot behaviors. We apply these ideas to the design and verification of a robotic bucket brigade and to simple, distributed assembly tasks as found in automated factories.

Comments

This material is posted here with permission of the IEEE. Such permission of the IEEE does not in any way imply IEEE endorsement of any of the University of Pennsylvania's products or services. Internal or personal use of this material is permitted. However, permission to reprint/republish this material for advertising or promotional purposes or for creating new collective works for resale or redistribution must be obtained from the IEEE by writing to pubs-permissions@ieee.org. By choosing to view this document, you agree to all provisions of the copyright laws protecting it.

NOTE: At the time of publication, author Daniel Koditschek was affiliated with the University of Michigan. Currently, he is a faculty member in the Department of Electrical and Systems Engineering at the University of Pennsylvania.

This conference paper is available at ScholarlyCommons: http://repository.upenn.edu/ese_papers/361
A Formalism for the Composition of Concurrent Robot Behaviors

Eric Klavins

Department of Electrical Engineering and Computer Science
University of Michigan
Ann Arbor, MI 48109

Daniel E. Koditschek

Abstract

We introduce tools which help us to compose concurrent, hybrid control programs for a class of distributed robotic systems, assuming a palette of controllers for individual tasks is already constructed. These tools, which combine the backchaining of continuous robot behaviors with Petri Nets, expand on successful work in sequential composition of robot behaviors. We apply these ideas to the design and verification of a robotic bucket brigade and to simple, distributed assembly tasks as are found in automated factories.

1 Introduction

Large distributed networks of robots and computers form the basis of modern manufacturing systems. It is desired that these systems be rapidly reconfigurable and easily programmable. These goals, however, are seldom achieved in practice. A main cost of designing or reconfiguring these systems is in programming low level controllers and control logic, challenging because of the complexity that hundreds of interconnected, concurrently operating robots necessarily incurs. The programming process can be ad hoc and frequently results in a large fraction of the control code being “exception handler code”. Unless a principled design method for these systems is developed, this cost will be felt in terms of expensive programming projects, incompletely understood factory behavior, and delays in the introduction of new products.

Burridge [1] et al., introduced backchaining as a way to sequentially compose closed loop robot behaviors in a safe and formal fashion. We believe that for this method to be useful in distributed robotics applications, it should be generalized to concurrent situations. The generalization should allow for decentralized control, modularity, resistance against disturbances both physical and logical, and ease of design.

Our goal is to develop such generalizations and use them to produce distributed control programs automatically from high level descriptions of assembly tasks. Although we believe the formalism developed here will prove applicable to a broad range of automation settings, our notion of assembly is more immediately inspired by the high flexibility, low volume setting targeted by the “Minifactory” of Rizzi et al. [12]. There, decentralized general purpose robots accomplish all the factory’s parts transport and assembly operations in fluidly choreographed transactions. For example, a complex subassembly task requiring four or six coordinated degrees of freedom can only transpire in such a Minifactory when some subgroup of the decentralized robots “agrees” to collaborate in forming the specialized “machine” (the higher degree of freedom coordinated mechanism) suited to the specific task at hand. Of course, that alliance must be temporary, since each of the participating agents is required to play different roles in other machines, both prior and subsequent to the instantiation of the one in question. The burden of keeping track of and regulating these threads of machine-intermingled agents as they form and dissolve is rendered particularly problematic by the absence of any centralized processing capability. The formalism we develop here provides a means of specifying, verifying, and then situating in robotic hardware, a decentralized decision making program of this kind.

It must be stressed that this formalism presupposes an infra-structure of tunable and switchable feedback controllers for which the methods developed here merely serve as “glue.” Such a palette of controllers is relatively easy to build for environments well described by generalized damper dynamics [7], but becomes challenging when dynamical dexterity is required. For example, in [1], substantial “hand building” affords deployments of controllers whose domains of attraction explicitly include portions of the forward limit sets of their neighbors. Here, we simply assume that these “dynamical systems details"
have been worked out via parametrized families of regulators, and focus on the logical coordination and scheduling problems that follow.

We address the problems of concurrency and composition of behaviors by introducing a formalism which subsumes the work in sequential composition. We define a way in which simple Petri Nets can contain a form of concurrent backchaining robot behaviors. We call the resulting nets Threaded Petri Nets, or TPNs. We also describe a simple net composition method which lends itself well to the kinds of decentralized assembly tasks encountered in manufacturing systems. This method allows us to compose many single robot programs into decentralized, concurrent programs for groups of robots which are guaranteed not to deadlock. Finally, we give several examples of how this formalism may be used to automatically construct provably correct distributed robotic systems: a robot bucket brigade, a simple assembly arrangement, and a factory compiler.

2 Related Work and Specific Contributions

Preimage backchaining was introduced into the motion planning literature in [10] as a method of sequentially composing motion strategies. In [1] this method was extended to dynamically dexterous robot manipulators in work that serves as the basis of our current research. The idea is to start with a palette of controllers \(\Phi_1, \ldots, \Phi_n\) for a robot. Suppose \(\Phi_k\) has domain \(D_k\) and goal \(G_k\). Order the palette by setting \(\Phi_i \succeq \Phi_j\) (read \(\Phi_i\) prepares \(\Phi_j\)) whenever \(G_i \subseteq D_j\). If the palette is suitably designed, then a switching strategy may be obtained which drives the robot to a goal from any initial condition in \(\bigcup_{i=1}^n D_i\). In this paper we expand these ideas (see Def. 3.4) to include the notion of concurrent composition of behaviors for the case of several robots in a shared workspace.

The approach to assembly in [7], for simple situations, introduces an automatic method for constructing a control law which guides a single robot to assemble a product from its parts based on the notion of an artificial energy landscape wherein the configuration of least energy is the one in which the product is assembled. This method is impractical when applied to the problem controlling an entire factory. Thus, in this paper we take the view that the product assembly graph (PAG) of a product corresponds to a discrete and parallelized version of the energy landscape. The steps of the assembly - the nodes in the assembly graph - may be given by artificial potential field controllers, but the logic of the assembly is given by the PAG. This allows us to use multiple robots, as in a high volume factory setting, and to design in a modular fashion.

Hybrid systems combine a discrete state and a continuous state into the same model, as in [3] where the hybrid automaton is used to model a hybrid system. A hybrid automaton is a finite automaton with continuous dynamics at the nodes and conditions on the state of the dynamical system which provide a switching rule between nodes. In [8], when the node dynamics are simple, the authors use modal logic to prove properties about their hybrid models. In [13], controllers for a given hybrid model are synthesized by computing the avoidable set of states of the automaton. We take a somewhat different approach here. Instead of finite automata, we use Petri Nets. Our compositional method is similar to those found in work on bottom-up synthesis of Petri Nets, especially [9] where simple Petri Nets are combined along paths and invariants of the resulting net are obtained from the constituent nets. We are also inspired by the modular construction of control policies introduced in [8] in which finite state supervisory controllers are combined to form more complex controllers. In the present work, the dynamics for nodes are taken from a well-understood palette of controllers where we backchain the domains and goals of the controllers to get concurrent transitions. Most importantly, we intend to automatically synthesize systems instead of modeling and then controlling a given system.

In summary, the chief contributions of this work are exhibited in Definitions 3.2 and 3.4. The first definition (Threaded Petri Net or TPN), introduces a formalism for keeping track of how the degrees of freedom in a large factory couple and decouple over time and what role each particular degree of freedom plays in a well specified sequence of controllers defining its trajectory. The second definition (the firing rule) situates in explicit physical terms the otherwise traditional (abstract) Petri Net firing rules. The "symbols" used to write the preconditions and postconditions of these rules - the "goals" and "domains" - are precisely defined by the robots' sensory and actuation signal streams. Specifically, they represent subsets of the robot's state space whose interpretation via concretely sensed events is built into the known properties of the associated closed loop regulator and state estimator. Thus, assuming the palette of controllers as described above, these firing rules can be implemented by automatically generated predicates involving the robot's intrinsic sensory signals. In consequence of these definitions, we are able to formalize in Definition 3.6 a simple but effective composition technique for combining together single cycles of such Threaded Petri Nets and then, in Theorem 3.2, to give a formal proof of liveness of the complicated physical factories that result. A pair of exam-
3 Formal Ideas

In this section we introduce the formal ideas that support this line of research. Because space is limited, we state properties without proof. The reader is referred to [6] for the details.

We adopt the following definition of a Petri Net, also called a condition/event net, found in [4].

**Definition 3.1** A Petri Net is a pair \((T, P)\) where \(T\) is a finite set of elements called transitions and \(P \subseteq 2^T \times 2^T\) whose elements are called places.

In a graphical representation of a Petri Net, such as in Figure 2, places are represented by circles and transitions by squares. In our research, a place represents a controlled dynamical subsystem of the entire system in question. Transitions represent discrete changes in the dynamics of subsystems. If \(\{(a_1, ..., a_i), (b_1, ..., b_j)\} \in P\), we write \([a_1, ..., a_i, b_1, ..., b_j] \in P\). If \(p = [a_1, ..., a_i, b_1, ..., b_j]\) then \(\text{left}(p)\) is the set \(\{a_1, ..., a_i\}\) and \(\text{right}(p)\) is the set \(\{b_1, ..., b_j\}\). A marking of a net \((T, P)\) is a set \(m \subseteq P\). The flow relation \(F\) of a Petri Net \((T, P)\) is the relation where \((t, p) \in F\) if \(t \in \text{left}(p)\) and \((p, t) \in F\) if \(t \in \text{right}(p)\). The preset of an element \(x \in T \cup P\) is set \(\{y \mid y \in F \ x\}\) and is denoted \(\ast x\). The postset of \(x\) is the set \(\{y\mid x \in F \ y\}\) and is denoted \(x^\ast\). These and other Petri Net basics can be found in introductory texts, such as [1].

Suppose we have a collection of robots and parts whose continuous state can be given by \(x = (x_1, ..., x_n)\) and whose global dynamics is simply \(x = u\). The dynamics of components of \(x\) are almost independent of each other. For example, \(x_1\) might correspond to the position of a robot on a guideway, while \((x_2, x_3)\) might correspond to the position of a part on an altogether different location of the factory floor. However, robots and parts do interact for short periods of time, as during a parts mating operation, so that the dynamics of certain components of \(x\) may occasionally be tightly coupled.

To describe how couplings change and which dynamics are operating on which components of \(x\), we introduce the Threaded Petri Net, or TPN. It is essentially a Petri Net with information added to keep track of how these couplings change. Every place will correspond to a control mode which we will have chosen from a palette of such modes. Thus, for each place \(p\) there is an \(l_p\)-dimensional system given by \(y = F_p(y)\) where \(y\) is an \(l_p\)-dimensional vector and \(F_p\) is chosen from the palette of controllers that we assume is already constructed. The mode has domain of attraction \(D_p\) and goal set \(G_p\). Let \(N = \{1, ..., n\}\) be the index set for the components of \(x = (x_1, ..., x_n)\).

**Definition 3.2** A Threaded Petri Net (TPN) consists of

1. a set \(T\) of transitions;
2. a set \(P \subseteq 2^T \times 2^T\) of places;
3. for each \(p \in P\), dimension, dynamics, domain and goal \(l_p, F_p, D_p\) and \(G_p\);
4. for each \(e \in T\) a bijective function

\[d_e : \bigcup_{p \in \ast e} \{p\} \times \{1, ..., l_p\} \rightarrow \bigcup_{q \in \ast e} \{q\} \times \{1, ..., l_q\}\]

called the redistribution function of \(e\);

subject to the condition that for each \(e \in T\),

\[\sum_{p \in \ast e} l_p = \sum_{q \in \ast e} l_q\]

(so that it is possible for \(d_e\) to be bijective).

Note that the difference between a TPN and a condition/event net is not only the additional information associated with each place. We have also added the redistribution functions, \(d_e\) for each \(e \in E\), which define what happens to each degree of freedom as mode changes occur. Figure 1 illustrates the redistribution function for a transition \(e\) in a sample net.

![Figure 1: A redistribution function of a transition can be represented by lines from components of the preset to components of the postset. In this net fragment, \(d_e(p_2, 1) = (q_2, 2)\) for example.]

**Definition 3.3** A marking is a pair \((m, f_m)\) where \(m \subseteq P\) and

\[f_m : \bigcup_{p \in m} \{p\} \times \{1, ..., l_p\} \rightarrow N\]

which specifies which degrees of freedom of the system each mode is operating on. A legal marking is one where \(f_m\) is bijective. We will be concerned only with legal markings in what follows.
A legal marking \((m, f_m)\) of a TPN says, for each \(p \in m\), which components of \(x\) \(F_p\) is acting on and what the dynamics of each component of \(x\) are. Thus, we can say how the state of the system is changing given a particular marking \((m, f_m)\). Given \(j \in N\), suppose that \(f_m^j(p) = (p, i)\). That is, under the marking \((m, f_m)\) the \(j\)th component of \(x\) is changing according to the \(i\)th dimension of the mode dynamics of \(p\):

\[
\dot{x}_j = \pi_i \circ F_p(x_{f_m(p,1)}, \ldots, x_{f_m(p,|p|)})
\]

where \(\pi_i\) gives the \(i\)th projection of the \(|p|\)-dimensional vector function \(F_p\). This is valid until some mode changes, which leads us to a definition of how events are triggered.

**Definition 3.4** Let \((m, f_m)\) be a legal marking. \(e \in T\) is \(m\)-enabled with respect to \(x \in \mathbb{R}^n\) if

1. \(*e \subseteq m\) and \(*e \cap m = \emptyset*\);
2. for each \(p \in e\), \((x_{f_m(p,1)}, \ldots, x_{f_m(p,|p|)}) \in G_p\);
3. for each \(q \in e^*, (x_{f_m\circ d_j^{-1}(q_1)}, \ldots, x_{f_m\circ d_j^{-1}(q_{|q|})}) \in D_q\).

Notice that condition (1) is just the usual definition of \(m\)-enabled for condition/event nets. The second two conditions impose the restriction that the dynamic systems in the preset of the enabled event must be in goal states and the systems in the postset must all be prepared. These two conditions do not affect the logical dynamics of the underlying net – they simply require that the control modes be designed so that events can become enabled.

A set of events \(G\) is called detached if whenever \(e_1\) and \(e_2\) are distinct events in \(G\), \(*e_1 \cap *e_2 = *e_1^* \cap *e_2^* = \emptyset*\). Suppose we have a marking \((m, f_m)\). The follower marking \((m', f_{m'})\) with respect to \(G \subseteq E\) is calculated as follows. As with condition/event nets

\[
m' = (m - *G) \cup G^*\].

The function \(f_{m'}(p, j) = \{f_m(p, j) \text{ if } p \in m - *G\}
\[
f_m \circ d_j^{-1}(p, j) \text{ otherwise}
\]

where \(e\) is the single event in \(p^* \cap G\). We write \((m, f_m) \rightarrow_G (f_m', m')\) when \((f_m', m')\) is the follower marking of \((f, m)\) with respect to \(G\). Since legal markings \((m, f_m)\) are such that \(f_m\) is bijective, we can be sure that every component of \(x\) is accounted for when the system is in the set of modes given by \(m\). We would also like that only legal markings be reachable from given legal markings, so that once the distributed process is underway, there is no point at which some part of \(x\) is not acting under a mode of the net. The following easy lemma gives us this.

**Lemma 3.1** Say \((m, f_m)\) is a legal marking, that \(G\) is a detached set of events, and that \(G\) is \(m\)-enabled (with respect to some \(x\)). If \((m, f_m) \rightarrow_G (m', f_{m'})\), then \((m', f_{m'})\) is also a legal marking.

The most important property of TPNs is that any subprocess corresponding to a particular component of \(x\) is completely sequential. This can be stated formally as follows.

**Theorem 3.1** Let \((m_0, f_{m_0}) \rightarrow_G^1 \cdots \rightarrow_G^k (m_k, f_{m_k})\) be a run of a TPN \((T, P)\). Given \(i \in N\), let \(p_j \in P\) be the place in \(m_j\) such that \(f_{m_j}(p_j, i) = j\) for some \(l\). Then there is a \(t_j \in G_j + 1\) such that \(p_j \in *t_j\) and \(t_j \in *p_{j+1}\) for each \(j \in \{0, \ldots, k - 1\}\).

A proof of this theorem is beyond the scope of this paper so we refer the reader to [6]. The main consequence of Theorem 3.1 is that if we would like to prove that a particular part, corresponding to a particular component of \(x\), is moved from its parts feeder, all the way through the factory, and finally into an output buffer, we simply examine the sequential sub-process of the TPN corresponding to the dynamics of the part. We will illustrate this in Section 4.

As mentioned, we intend to compose TPNs into factories. We present a simple type of composition to complete this section. It is based on the idea of a cyclic subprocess, which we call a gear, and which we use as the basic building block of our nets. A gear represents the simplest thing a robot in a factory can do, besides remain idle: cycle repeatedly through some set of behaviors. A robot might, for example, (1) pick up a part at a parts feeder, (2) bring the part to a station to be glued to another part, (3) take the result to a manipulator to be added to some other subassembly, and then start the sequence again. Formally, we have:

**Definition 3.5** A \(k\)-gear is a net \((T, P)\) where \(T = \{t_0, \ldots, t_{k-1}\}\) and \(P = \{[t_i, t_{i+1}] \mid i \in \mathbb{Z}/k\}\). \(m \subseteq P\) is a legal marking for a \(k\)-gear if \(|m| = 1\).

(We ignore the dynamics and redistribution functions for now.) A gear for a robot models only what the single robot in question is doing while, in fact, the robot must coordinate with the controllers of other robots. Thus, the gear of the robot must be synchronized with the programs of other robots. Of course, the programs of other robots are also given by gears. What is needed is a way to compose gears, so that control modes are synchronized. Furthermore, it is important that, before entering the mode, each robot involved in a control mode wait for the other robots involved. With these constraints in mind, we are led to a definition of a gear net as a certain union of gears.
Definition 3.6 A gear net is defined recursively:

1. A gear is a gear net.

2. If \((T, P)\) is a gear net and \((S, Q)\) is a gear then \((T \cup S, P \cup Q)\) is a gear net as long as the following conditions hold:

   (a) let \((T_1, P_1), \ldots, (T_k, P_k)\) be the set of gears in \((T \cup S, P \cup Q)\) which intersect \((S, Q)\). Then \(\cap_{i=1}^k P_i = \{[a; b]\}\) and \(\cap_{i=1}^k T_i = \{a, b\}\) for some transitions \(a\) and \(b\).

   (b) there exists a transition \(c \in S - T\) such that \([c; a] \in Q\).

A legal marking for a gear net is one in which each gear in the net is marked exactly once.

Since all places in a gear net are of the form \([x; y]\), gear nets are a kind of marked graph, a class of nets which have been extensively studied. (See [2], for example.) Note that a legal marking gives the state of every gear in the gear net. This corresponds to the fact that each robot is in exactly one state in its program. Conditions (a) and (b) require that gears be added with a "standard interface".

We can show the following properties about gear nets. The first says that the gears we add in the composition are the only cycles in the net.

Proposition 3.1 If \((T, P)\) and \((S, Q)\) are as in Definition 3.6 and \((S', Q')\) is a gear contained in \((T \cup S, P \cup Q)\), then either \((S', Q') = (S, Q)\) or \((S', Q')\) is contained entirely within \((T, P)\).

The next property gives us that each gear can be marked exactly once.

Proposition 3.2 Every gear net has a legal marking.

The last property is that all gear nets are deadlock free. Thus we are assured that systems we build up from gear nets are live, logically conflict free processes.

Theorem 3.2 Gear nets are deadlock free under legal markings.

To combine gear nets with TPNs to get factories, we need parts feeders and output bins. Suppose we have a gear net \((T, P)\). We may add a special parts feeder place, \(p = [0, t]\) to the net to get \((T, P \cup \{p\})\) where \(t \in T\) is a transition occurring in exactly one gear of \((T, P)\). We assume that \(p\) is always marked except when \(t^*\) is marked (to eliminate contact). Thus, a parts feeder node is a variation on the idea of a source node, commonly used in the Petri Net literature.

Similarly we may add an output buffer place, \(q = [s, 0]\) to the net where \(s \in T\) is also unique to a single gear. We suppose that \(q\) absorbs parts so that it is never present in a marking. Thus, an output bin is a variation on the idea of a sink. These notions can be added to the dynamics of a Petri Net by adding special conditions, to the definition of follower marking, for parts feeders and output bins. In general, a factory arising from a gear net, with \(i\) parts feeders and \(j\) output bins, will have the form

\[(T, P \cup \{\tilde{p}_1, \ldots, \tilde{p}_i, \tilde{q}_1, \ldots, \tilde{q}_j\}).\]

The addition of sources and sinks to gear nets does not alter the fact that they are deadlock free.

When we add dynamics to a gear net with feeders and output buffers, we simply define the dimension of a parts feeder to be 1 and the dynamics to be \(F_p(y) = 0\) with \(D_p = G_p = \mathbb{R}\). Thus, whether the transition after a parts feeder node is enabled under a marking does not depend on the state of the dynamics of \(\tilde{p}\). A similar definition is made for output bins.

4 Examples

In this section we describe two examples that demonstrate the above formalism. The first, the most simple, nontrivial example, is what we call a bucket brigade. It includes: task level information, where we specify a palette of controllers, which correspond to
places in a TPN; task switching and concurrency; and a simple notion of product flow. Second, we describe a simple assembly process in terms of three robots. Simulations of these and other examples can be viewed at http://www.eecs.umich.edu/~klavins/mf/.

4.1 The Bucket Brigade

Bucket brigades correspond to individual lines in an assembly process. We expect their analysis to contribute to our understanding of more complicated factories. Figure 2 shows the TPN we will use to model a brigade.

Figure 3: A simple, three robot bucket brigade

A simplistic, three robot bucket brigade consists of three robots, R1, R2 and R3, a parts feeder and an output buffer arranged in a line as in Figure 3. The task is for R1 to pick up parts, one at a time, and transfer them to R2, which transfers them to R3. R3 deposits the parts in an output buffer. The robots have width 2r where 0 < r < 1. The workspace is defined to be the closed interval of the real line [0, 3]. The robots have continuous state variables, x1, x2 and x3, corresponding to their positions on this line. We assume that the range of each robot is restricted, so that x1 ∈ [i − 1, i] only and also that robots can not inhabit the same place at the same time. That is, the distance between any two robots must be greater than 2r. Each robot also has a discrete state, corresponding to whether or not it is carrying a part, denoted b1, b2, b3 ∈ {0, 1}. (b1 = 1 if R1 is carrying a part and b1 = 0 otherwise.) Finally, there is a supply of parts with state variables z1, ..., z3. The physics are not entirely realistic: the velocity of the robots is directly controllable (z = u); parts move with the robots they are near (so that when b1 = 1, z1 = x1 for some part with index k); and part transfers happen instantaneously as long as the robots involved are close together.

First we specify the palette of controllers. There are nine: Fpick, Fdrop, Fwait, and Fhold, for i ∈ {1, 2, 3}, Ftrans1,2, and Ftrans2,3. The use of these controllers is summarized in Table 1. Note that controller goals are given by a point x* in the table but we usually consider them to be a set, B1(x*) = (x* − ε, x* + ε). We will describe each of them qualitatively. In simple situations, we have implemented such controllers (see the web site mentioned above).

Fpick is a two-dimensional controller. One dimension corresponds to the position R1 and the other to the position of the 4th part, which we assume is stationary in the parts feeder (so z4 = 0). There is a single attracting equilibrium point at 0 where R1 is next to the parts feeder with a part. The parts feeder operates by proximity: if b1 = 0 and ||z1|| < ε then b1 will eventually become 1. In the brigade, when b1 = 0, R1 will run ˙z1 = Fpick(z1) until b1 = 1. Since the controller does not affect the part, z4 = 0.

Fdrop controls R3 to drop off a part at the output buffer. It is used when b3 = 1 and eventually results in b3 becoming 0.

Fwaiti, for each i, is a one-dimensional controller with attracting equilibrium point i − 1/2. It is used when Ri needs to wait for another robot, the parts feeder or the output buffer, essentially driving the robot to a safe place in the workspace and keeping it there. Fhold, is the same except it is two-dimensional and used when the robot is waiting and holding a part.

Ftrans1,2 is a three-dimensional controller for R1 to hand R2 a part, the position of which we assume to be given by z2. It has attracting point (z1, z2) = (1 − r, 1 + r) (implying that z2 is attracted to 1 − r) and thus drives R1, R2 and the part to a configuration where the robots are touching. When (b1, b2) = (1, 0) and ||z2 − x1|| < ε for some ε, the state simultaneously becomes (b1, b2) = (0, 1) and z2 = x2. Note that according to our notion of controller, Ftrans1,2 is being run asynchronously: half by R1 and half by R2. That is, R1 obtains an estimate of x1 of R via its sensors and an estimate x2 of R2 via the communication system and runs the feedback controller

\[ \dot{x}_1 = \pi_1 \circ Ftrans1,2(x_1, \dot{x}_2) \]

concurrently and asynchronously while R2 runs \( \pi_2 \circ Ftrans1,2 \). While the controller is running, the change in the part's position is the same as that of the robot that is holding it. Ftrans2,3 is similar.

With the controllers specified, we construct programs for each robot. The first robot repeatedly receives parts from the parts feeder and hands them to the second robot. Thus, R1 runs the program

```
While True
  If b1 = 0
    Wait for parts feeder
  Run ˙x1 = Fpick(x1) until b1 = 1
  Else
    Wait for R2 to be ready
  Run ˙x1 = π1 \circ Ftrans1,2(x1, x2) until b1 = 0
EndIf
EndWhile
```

The other two robots have similar programs. The second robot receives parts from the first and hands
them to the third. The last robot receives parts from the previous robot and deposits them in the output buffer. Notice that any line in a program that says "wait for ..." has the meaning "run \( F_{\text{wait}} \), until the robot (or feeder or buffer) is in its goal set" and presumes some simple communication system that we do not describe in any detail here.

The dynamics of the bucket brigade we have constructed can be modeled by the tools we presented in Section 3. Figure 2 shows the resulting TPN. There are three gears, one for each robot. Gear two corresponds to the program of \( R_2 \), for example, and consists of transitions \( \{ t_1, t_2, t_3, t_4, t_5 \} \) and places \( \text{wait}_2 = [t_5, t_1], \text{trans}_1,2 = [t_1, t_2], \text{hold}_2 = [t_2, t_4], \text{trans}_2,3 = [t_4, t_5] \).

As an example, we supply one redistribution function in detail, namely, \( d_{t_1} \). \( F_{\text{hold}_2} \) expects a robot position and then a part position, in that order. \( F_{\text{wait}_2} \) expects a robot position. \( F_{\text{trans}_1,2} \) expects the position of the sending robot, the receiving robot and the part in that order. Then we have:

\[
\begin{align*}
 d_{t_1}(\text{hold}_1, 1) &= (\text{trans}_1,2, 1) \\
 d_{t_1}(\text{hold}_1, 2) &= (\text{trans}_1,2, 3) \\
 d_{t_1}(\text{wait}_2, 1) &= (\text{trans}_2,3, 2)
\end{align*}
\]

The rest of the redistribution functions should now be apparent from Figure 2.

Let \( \tilde{p} = [0, t_3] \) and \( \tilde{q} = [t_7, 0] \). The initial marking, \( (m_0, f_{m_0}) \) is given by \( m_0 = \{ \text{wait}_1, \text{wait}_2, \text{wait}_3, \tilde{p} \} \) and

\[
\begin{align*}
 f_{m_0}(\text{wait}_1, 1) &= 1 \\
 f_{m_0}(\text{wait}_2, 1) &= 2 \\
 f_{m_0}(\text{wait}_3, 1) &= 3 \\
 f_{m_0}(\tilde{p}, 1) &= k
\end{align*}
\]

for some \( k \). Note that the addition of parts feeders results in our having to expand and contract the index set \( N \). \( N \) will always include \( \{1, 2, 3\} \) and, depending on what parts are present in the brigade and the parts feeder, may also include some set \( \{k_1, \ldots, k_j\} \) of part indices. In our "toy" factory compiler, these programs are constructed automatically.

Now we can show two things. First, the bucket brigade never deadlocks by Theorem 3.2. This follows from the fact that its underlying structure is a gear net and because the domains of each place include the goals of the places that precede it. As an example, consider \( \text{trans}_1,2 \). Its domain is \( D_{\text{trans}_1,2} = [r, 1] \times [1, 2 - r] \) according to Table 1 (we ignore the part position since it is the same as one of the robots). Now, \( \text{trans}_1,2 = \{ \text{hold}_1, \text{wait}_2 \} \) and

\[
G_{\text{hold}_2} \times G_{\text{wait}_2} = B_1(\frac{1}{2}) \times B_3(\frac{3}{2}) \subseteq D_{\text{trans}_1,2}.
\]

Second, we can deduce from Theorem 3.1 that the parts move from one end of the brigade to the other. Suppose that at some marking \( (m_j, f_{m_j}) \) that \( \text{pick} \in m_j \) and \( f_{m_j}(\text{pick}, 2) = k \). That is, suppose that \( R_1 \) has picked up part \( k \). Then we know there is a sequence of places that control part \( k \). In fact, the sequence is

\[
< \text{pick}, \text{hold}_1, \text{trans}_1,2, \text{hold}_2, \text{trans}_2,3, \text{hold}_3, \text{drop} > .
\]

Now if we trace the position of the robot carrying the part in the goals of these controllers from \( \text{pick} \) to \( \text{hold} \) (see Figure 1 again), we see that the initial state of the part is \( z_k = 0 \) and the final state is \( z_k = 3 \).

Notice that control is decentralized as we required in our statement of the problem. Communication is kept low: each robot need only communicate with at most one other robot at a time. And the dimension of the control laws is limited. We could, in principle, build bucket brigades with an arbitrary number of robots, yet we would not have to build anything that is fundamentally different from what we already have. Thus, for our simple situation, the method scales.

### 4.2 Assembly

We may also use our formalism to model simple assembly processes. In this example, three robots perform this task on a T-shaped guidepath. The first two pick up parts at their respective parts feeders and then, in a synchronized operation with the third robot, attach the two parts together and place the resulting assembly on the third robot. The third robot drops the assembly into a parts bin. Figure 4 shows how the robots are arranged.
There are wait, and hold, operations as before, two pick operations corresponding to the two different parts, and one drop operation. New to this example is the mate operation. It controls two robots, each carrying a part, and a third robot to meet at the intersection of the guidepaths. Once there, the assembly of the two parts occurs, and the result is place on the third robot. The TPN we use to model this small factory has three gears connected at a single interface, which represents the mate operation. Two parts lines originate at parts feeders, meet at the mate operation and end up at an output buffer. We can show that this system does not deadlock using the properties of gear nets. We can also show that the two types of parts progress through the factory using the properties of TPNs.

5 Toward a Factory Compiler

A main goal of our research is to use the TPN formalism and gear composition to generate distributed factory programs automatically from product assembly graphs. We have developed a simple compilation procedure which, given a PAG, can generate a toy factory [5]. To each operation in the PAG, we assume that a template controller is already built over an “ideal workspace”. We also suppose the existence of template controllers for picking up parts at parts feeders and dropping them off at output buffers. To compile, we annotate the PAG with Petri Net fragments (which represent the template controllers) and then connect them so as to create a gear net. We know that the resulting net is live because it is a gear net. To date we have experimented only with simple layout procedures which embed the union of the “ideal workspaces” into a single instantiated workspace.

We expect to be able to optimize the resulting net for robot reuse (reallocating tasks so that one robot alternates between tasks formerly assigned to two robots) and for parallelization of tasks. This leads to TPNs that are not based on gear nets but which do have a regular structure. We are also working on applying these ideas to a factory design tool for use with the Minifactory—a modular, rapid response manufacturing system currently in development [12].

Acknowledgments

Thanks to Bill Rounds and Al Rizzi for their advice and suggestions regarding this research. Eric Klavins is supported by the Charles DeVlieg Foundation Fellowship for Manufacturing and Dan Koditschek’s contribution to this work was supported in part by the NSF under Grant IRI-9510673.

References


