Towards Performance Guarantees For Emergent Behavior

Damian M. Lyons[1]
[1]*Dept. of Computer & Information Science*
Fordham University
Bronx NY 10458
dlyons@fordham.edu

Ronald C. Arkin[2]
[2]*College of Computing*
Georgia Institute of Technology
Atlanta GA 30332

***Abstract* – It is important to be able to guarantee the *safety and effectiveness* of robot behavior in applications where robots must operate alongside people or in hazardous situations. A modeling framework based on port automata and asynchronous communication is introduced inthis paper. By looking at the internal transitions between port communications, an analysis approach is developed that removes the combinatoric issues of looking at an asynchronous combination of robot and environment. An example application of the approach to wheel slippage in a mobile robot is presented.**

I. INTRODUCTION

Robot systems are starting to play an important role in military and in other government applications such as disaster recovery, and search & rescue. They are also appearing in the consumer area, e.g., Sony AIBO, Roomba, etc. The issue of being able to guarantee the *safety and effectiveness* of robot behavior is therefore coming increasingly to the forefront, especially when robots must operate near people or in hazardous situations. There is an analogy with the history of civil engineering: bridges and other major structures were constructed for thousands of years before the necessary mathematical tools were developed to guarantee their performance. In the 19th and 20th centuries, as such projects became more ambitious, some spectacular failures ensued due to the absence of effective performance guarantees.

Although formal modeling methods have found success in many computer science applications [5], they have been less prevalent in behavior-based robotics. The *behavior-based* robot programming paradigm [1] has achieved wide popularity and success in addressing the construction of robust robot systems that can operate in unstructured environments. However, the behavior-based approach uses assumptions quite different from those used generally in designing software, and this complicates the formal analysis of these systems. One key assumption is the reliance on *emergent behavior*. The resulting open-ended nature of the list of potential robot-environment interactions is a serious issue for formal analysis.

In this paper, we introduce a model-checking approach to the analysis of a behavior-based robot in an unstructured environment, with the objective of determining performance guarantees for the overall system. The principle research contribution in this paper is the development of an efficient approach to handle the open-endedness of dynamic robot-environment interplay. The ultimate objective of the work is to develop a software tool that can be used within *Missionlab* [13] to build mobile robot controllers that operate in unstructured and dynamic environments with formally guaranteed performance.

II. EXISTING WORK

The AI community has developed a number of approaches to reasoning about actions and their effects [16]. These approaches focus strongly on the representation of the behavioral schemas, and consider the robot to be the primary generator of events. The role of the environment is simply to respond. The situated automata approach [6] acknowledges the role of environment – but the environment model is represented using a modal logic of knowledge. The theoretical limitations of this approach are still unclear, since it requires automatically producing a 'program' from a modal logic description – a very difficult problem [3].

The landmark work by Ramadge & Wohnam [14] introduced a formalism and methodology for using a finite-state automaton (FSA) to control a discrete event plant – discrete-event control (DEC). There have been a number of extensions to this concept, including the addition of concurrent models as well as addressing the problems of integrating continuous and discrete control. There are also a number of successful automaton-based methods for representing robot programs. These include [9] as well as Georgia Tech's *MissionLab* [13]. [7] integrates a process description vocabulary with the FSA theory of Ramadge & Wonham. It associates an FSA-semantics with the process operators of [10] allowing an elegant integration with DEC results.

To model the emergent behavior of a behavior-based system, it is necessary to have models of salient aspects of the environment. It is reasonable to expect that most of the processes at work in the robot's environment are current and asynchronous with respect to the robot. However, FSA semantics for concurrent processes has the fundamental issue of *combinatorial state explosion*: combining FSAs into a single FSA requires computing a Cartesian product of states, a process whose complexity is exponential. This is a practical limitation on its usefulness.

The Port Automata (PA) model [17] exploits message passing between concurrent automata to simplify the analysis of concurrency. Each process can be analyzed separately up to a message passing event – avoiding the need to compute a Cartesian product. We exploit this approach to combat the computational complexity of our problem.

III. UNSTRUCTURED ENVIRONMENT

An unstructured environment contains a large number of phenomena with which a behavior-based system can interact, and the richnes of the resulting emergent behavior is strongly related to the richness of the environment. Thus, to make performance guarantees about the behavior, we have to model all the interactions between the environment and the robot. Consider a very simple robot controller and a very simple environment modeled by the two 3-state automata shown in Figure 1. For example, the controller FSA might control wheel velocity, and the environment FSA might model the interaction of the wheel with the ground.

The Controller and Environment will share events (the arrows in the diagram), and this specifies their potential interactions [14, 12]. In the wheel and ground example, this interaction would be the physical interface between wheel and ground. It is reasonable to assume that the number of shared events is much less than the total number of events. In the case where there are no shared events, that is, Controller and Environment are completely asynchronous, then the combination of the two automata, the *shuffle product automaton*, has 9 states. If one more 3-state automaton is added to the environment model then there are 27 states, illustrating the combinatorial nature of the shuffle product complexity. Shared events reduce the total number of states; however, there will be few shared events.

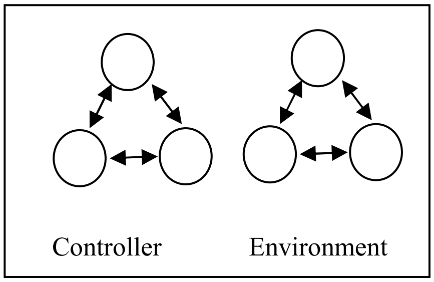

Figure 2: Controller & Environment Automata Models

This also illustrates that interaction complexity and emergent behavior is a function of environment as well as robot, and even a 3 DOF mobile robot will encounter substantial complexity in a sufficiently rich and realistic environment.

IV. PORT AUTOMATA

A port automaton (PA) is a finite-state automaton equipped with a set of synchronous communication ports. The focus in this work will be restricted to deterministic PAs, since [17] show that any non-deterministic PA can be built from a network of deteterministic PAs. Networks of deterministic processes, e.g, Kahn networks, have appeared in robotics before [18, 9, 13].

Formally we can write a port automaton P as:

$P = (Q, L, X\,\delta, \beta, \tau)$ where (1)

Q is the set of states

L is the set of ports

$X = (X_i \mid i \in L)$ is the event set for each port

Let $XL = \{ (i, X_i) \mid i \in L \}$ i.e., a disjoint union

$\delta : Q \times XL \to 2^Q$ is the transition function

$\beta = (\beta_i \mid i \in L)$ $\beta_i : Q \to X_i$ output map for port i

$\tau \in 2^Q$ is the set of start states

For example, $\delta(q,(1,a))= \{p\}$, $\beta_2(p)=b$, states that in state q, if there is an input a on port 1 then the automaton transitions to state p, and writes b to port 2. All communication involves a "swap" of values between the sender and receiver. One or more of these values could be "#", the trivial or blank value, in which case the transfer of information is in one direction; an input operation only or an output operation only.

Let $A(q)$ be the set of ports that are able to communicate in state q (the *active* ports):

$$A(q) = \{ i \in L \mid \exists (i,x) \in XL, \delta(q,(i,x)) \neq \varnothing \}$$

Note that this includes output as well as input activation.

Interaction is modeled *explicitly* via port communication. For example, if the wheel and ground example of Fig. 2 is modeled using port automata, there would be a pair of channels over which the wheel and ground automata communicate. The wheel controller automaton might transmit wheel torque and surface contact information to the ground model over one channel. The second channel could be used for the ground to transmit back reaction data for sensors modeled within the wheel controller. Note that the internal processing of the ground automaton on receiving its input until it produces a result value can be analyzed separately from the processing in the wheel controller automaton.

Consider two port automata, *P1* and *P2* (Fig. 3). Let some of the ports on *P1* and *P2* be connected as described by a one-to-one mapping *c*. These connected ports are the only channels over which the two automata can communicate. An expression will be developed for the internal processing of P2 from when it first communicates on one of its connected ports to when it communicates again.

Assume that each PA has a subset of its ports that are self-connected. A write to one of these ports is the equivalent of storing to an internal variable, which could be later retrieved by a read from that port. Let S be the set of self-connected ports, and let E be the set of ports available for external connections, where $L = E \cup S$ and $E \cap S = \varnothing$. In that case, the port map between P1 and P2 is $c: E1 \to E2$.

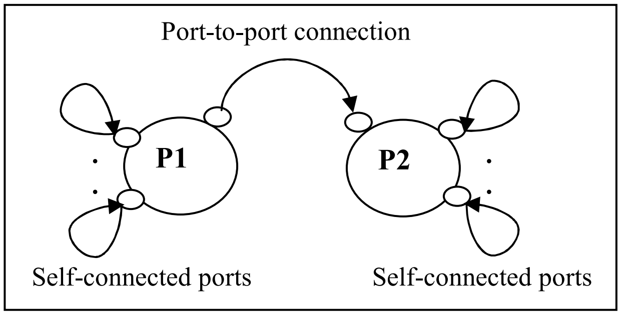

Figure 2: Two connected Port Automata

Consider a single communication event from P1 to P2, occurring in state $q \in Q2$ on the port $i \in E2$. The states reached by a single transition from q may include some states in which only self-connected ports are active, and some in which external ports are active. The latter indicates that the automaton is ready to communicate again. The former indicates that internal processing is still in progress. Each additional transition from such a state again brings the alternatives of internal processing or a potential communication. This reachability can be captured by an *internal reachability function* $\hat{\delta}(q,(i,x))$ that maps the state and communication pair to the set of states eventually reached after internal communication, and in which the next external communication event could occur.

Let $I(q)$ and $K(q)$ be defined as the sets:

$$I(q) \; = \{ \, q_k \mid q_k \in \delta(q,(i,x)) \; \& \; A(q_k) \cap E = \varnothing \, \}$$

$$K(q) \; = \; \{ \, q_k \mid q_k \in \delta(q,(i,x)) \; \& \; A(q_k) \cap E \neq \varnothing \}$$

The internal reachability function is then:

$$\hat{\delta}(q,(i,x)) \; = K(q) \cup \bigcup_{p \in I(q)} \; \bigcup_{j \in A(p) \cap E} \hat{\delta}(p,(j,v)) \qquad (2)$$

where $x \in X_i$ and $v \in X_j$.

A *port connection automaton PC* consists of two port automata *P1* and *P2* with their ports connected by a one-to-one mapping c and is written $P1 \mid_c P2$. Whereas analysis of the shuffle product of P1 and P2 involves generating and handling Q1×Q2, analysis of the port connection automaton involves handling Q1 and Q2 *separately* using the internal reachability function, until they communicate, after which they can be handled separately again. A computation of P1 and P2 can be modeled as a sequence of communication points, each separated by an application of $\hat{\delta}$ for P1 and for P2.

V. AUTOMATA NETWORKS

To specify networks of PA, we employ a notation of processes and process composition operations similar to the well-known CSP algebraic process model of [4, 15]. However, unlike CSP, but like [7, 10], the notation can be seen as simply a shortcut for specifying automata; A process is a port automaton, and a process composition operator is a port connection automaton.

Processes can take some initial parameter values to guide their computation and may produce some values at the end of their computation (if and when they terminate) that can be passed on to other processes. By $\mathbf{P}_u<v>$ is meant a process **P** that takes initial values u and produces results v. The semantics of **P** is a port automaton P constructed by considering the initial and result values to be conveyed over ports. A *basic process* [4] is atomic, and corresponds to a port automaton defined directly by transition function. These are processes with simple and easily characterized behavior: e.g., $\mathbf{Delay}_t$ is a process that terminates t seconds after it has been started; $\mathbf{Ran}_R<v>$ is a basic process that returns a random vector v from a set R.. We also introduce $\mathbf{In}_c<x>$ and $\mathbf{Out}_{c,x}$ as basic processes to perform input and output, respectively, on port c.

Processes are combined together using composition operators. $\mathbf{In}_{c1}<x>;\mathbf{Out}_{c2,x}$ is process that inputs a value on port *c1* and then outputs it on port *c2*. This is a port connection automaton of two automata, corresponding to the basic input and output processes, in which the first automaton executes to a termination state, at which point the second one starts. A port communication needs to occur to let the second process know when to start, and to transfer any values from the first to the second process. The semicolon denotes sequential composition.

In concurrent composition, both automata execute at the same time. For example $(\mathbf{Out}_{c2,x} \mid \mathbf{In}_{c2}<x>)$ is a port connection of two automata, one that outputs a value on its port *c2* and one that inputs a value from its port *c2;* we establish the convention that similarly named ports are connected to each other. The vertical bar denotes concurrent composition.

To analyze a network of processes, it is necessary to understand how that network *changes* as time progresses and processes terminate and/or are created. This is the process-level equivalent of the PA transition function, combined with the axioms that define port-to-port communication. We adapt[1] CSP notation and use the "→" symbol to denote this *process transition function,* e.g. $\mathbf{P};\mathbf{Q} \rightarrow \mathbf{Q}$ when **P** terminates, by the definition of synchronous composition (similar to the *evolves* operator of [10,11]). The following process expression follows directly from the definition of PA port communication[2]:

$$(\, \mathbf{In}_c<v> \, ; \, \mathbf{P}_v \mid \mathbf{Out}_{c,v} \, ; \, \mathbf{Q} \,) \rightarrow (\, \mathbf{P}_v \mid \mathbf{Q} \,)$$

This expression brings out the fact that the basic PA communication paradigm is a variation on *synchronous* communication. To analyze emergent behavior it is necessary also to support asynchronous communication. We introduce a process composition operation that allows us to explicitly model the timing of asynchronous communication.

A *disabling* composition of two processes is written (**P#Q**) and denotes a port connection automaton of **P** and **Q** connected so that whenever **P** terminates, it causes **Q** to terminate, and vice-versa. The connection entails: transmitting a message on a designated port on termination; rendering that port active in every transition; and, the reception of the value on that port taking the automaton to a termination state.

We define *asynchronous* communication between processes **P** and **Q** in a network **T** as follows:

$$\mathbf{P} = (\, \mathbf{In}_c<v> \,\#\, \mathbf{Delay}_{t1} \,) \, ; \, \mathbf{Delay}_{t2} \, ; \, \mathbf{P}$$

$$\mathbf{Q} = (\, \mathbf{Out}_{c,v} \,\#\, \mathbf{Delay}_{t3} \,) \, ; \, \mathbf{Delay}_{t4} \, ; \, \mathbf{Q}$$

$$\mathbf{T} = (\, \mathbf{P} \mid \mathbf{Q} \,)$$

P repeatedly offers to accept a message for some time *t1* and then does some internal processing represented by a delay of time *t2*. **Q** similarly repeatedly offers to send a message for time *t3* and then does internal processing represented by *t4.* As long as $t1+t2 \neq t3+t4$ then (**P|Q**) will repeatedly result in a network of concurrent **In** and **Out** processes that will communicate on *c.*

VI. ROBOT CONTROL NETWORK

A simple example of a random wander behavior design pattern can be defined as:

$$\mathbf{Wander} = \mathbf{Ran}_R<v> \;\; ; \; (\, \mathbf{Move}_v \,\#\, \mathbf{Delay}_t \,) \, ; \, \mathbf{Wander}$$

where $\mathbf{Move}_v$ is defined as a process that causes the robot to move with positive velocity *v.* Informally the behavior of **Wander** is to select a random velocity, then cause the robot to move in that direction for a fixed time *t*, and then repeat the

[1] In CSP, an event follows the → operator.

[2] The precedence order is: sequential, disabling, concurrent.

process forever. Consider how **Wander** behaves under the process transition function:

$$\mathbf{Wander} = \mathbf{Ran}_R{<}v{>}\ ;\ (\mathbf{Move}_v \# \mathbf{Delay}_t)\ ;\ \mathbf{Wander}$$
$$\rightarrow (\mathbf{Move}_v \# \mathbf{Delay}_t)\ ;\ \mathbf{Wander} \qquad \textit{for some } v \in R$$
$$\rightarrow \mathbf{Wander} \qquad \textit{after time } t$$

A *trace* of a CSP process is the finite sequence $tr(\mathbf{P}) = [a.b.c\ldots]$ of events the process has engaged in up to some moment in time. An event here is simply *the termination of a process,* and the label for the event will simply be the name of the process. The set of traces of **Wander** are of the form:

$$tr(\mathbf{Wander}) =$$
$$pref\ \{ s \mid s = [\ (\mathbf{Ran}_R{<}vi{>}\ .\ \mathbf{Delay}_t\ .\ \mathbf{Move}_{vi})^i\ ],\ \ i \geq 1\ \}$$

where *pref S* extends the set S to be prefix-closed. We define the set of traces associated with a transition operation, $tr(\mathbf{P} \rightarrow \mathbf{Q})$, to be the smallest set of traces generated in transitioning from **P** to **Q**. For example,

$$tr(\mathbf{Wander} \rightarrow \mathbf{Wander}) =$$
$$pref\ \{ s \mid\ s = [\ \mathbf{Ran}_R{<}v{>}\ .\ \mathbf{Delay}_t\ .\ \mathbf{Move}_v\ ],\ v \in R\ \}$$

This allows us to capture the "periodicity" in $tr(\mathbf{Wander})$ above: $tr(\mathbf{Wander}) = tr(\mathbf{Wander} \rightarrow \mathbf{Wander})^i,\ i \geq 1$.

These transition and trace definitions give a formal way to calculate what was informally stated in the previous section. However, if we want to establish performance guarantees, what is missing here is a model of the environment in which **Wander** operates.

VII. INTERACTION WITH ENVIRONMENT

Consider modeling the physical robot base as it moves around: For now, the environment will consist solely of the state of the base, including position and velocity. The actuator model is specified by the processes **Move** and **Base**:

$$\mathbf{Move}_v = \mathbf{Out}_{cv,v}\ ;\ \mathbf{Move}_v$$

This is a simple interface between the controller and the physical robot base: a velocity command is written to a port *cv*. The model of the physical robot base is defined by

$$\mathbf{Base}_{p,v} = (\mathbf{In}_{cv}{<}u{>}\ \#\ \mathbf{Moving}_{p,v}{<}q{>})\ ;\ \mathbf{Base}_{q,u}$$

The **In** process receives the velocity control input *u* and applies it to the robot base starting at the position *q* when the input was received. The base continues **Moving** until it receives the next control input.

$$\mathbf{Moving}_{p,v}{<}q{>} = (\mathbf{At}_p\ \#\ \mathbf{Delay}_t)\ ;\ \mathbf{Moving}_{p+vt,v}{<}p+vt{>}$$

The process $\mathbf{At}_p$ represents the current position of the base. In this network, the base remains at a position *p* for some small time *t* and then *asynchronously and instantaneously transitions* to the position *p+vt.* Thus, the values of the position state variable are restricted to lie on a grid, though the grid can be made as fine as desired by making *t* as small as necessary. Note that the *same process terminology* has been used to describe the physical world as was used to describe the controller.

We can now analyze how **Wander** behaves in this environment (a *model-checking* analysis) by looking at the concurrent composition **S1**:

$$\mathbf{S1}_{p,v} = (\ \mathbf{Wander} \mid \mathbf{Base}_{p,v}\ )$$

Recall: each process is a port automaton, and the networks built using composition operators are port connection automata. That means, for **S1** we can employ the internal transition function to analyze **Base** and **Wander** separately until they communicate and thus avoiding, as mentioned, the combinatorial issues associated with their shuffle. However, the special control port we introduced for disabling composition is active in every process on every transition. For this reason, when applying the internal transition function, we *omit any special ports introduced for composition operators*. Instead, we will use the process transition operator and traces to capture & analyze sequencing, and we retain the computational advantage of using the internal transition function to go from external communication to the next external communication in each automaton. We can rephrase the internal transition function in terms of the process transition operator:

$$\hat{\delta}(\mathbf{P}) = \{\ \mathbf{Q} \mid \mathbf{P} \rightarrow \mathbf{Q{=}f(Q1,\ldots,Qn)},$$
$$\exists i \in 1,\ldots,n,\ \mathbf{Qi{=}In} \vee \mathbf{Qi{=}Out}\ \}$$

That is, we look for the first network that we can reach from **P** via the process transition operator and which contains an input or output operation. When **P** is analyzed *separately* from other processes, then it can only transition from such a **Q** due to disabling composition with a process like **Delay** that disables the pending communication. If we use the asynchronous communication pattern described in the previous section, (and used in **Base** above) then the effect is that **P** repeatedly transitions to a network **Q** that is ready for communication. This periodic nature means that **Q** is a *fixpoint* of the process transition operator, defined here as $\mathbf{Q} \in \hat{\delta}(\mathbf{Q})$. We associate a fixpoint with a process network as follows: **Q** is a fixpoint of **P**, written $Y(\mathbf{P}){=}\mathbf{Q}$, iff $\hat{\delta}(\mathbf{P}){=}\mathbf{Q}$ and $\mathbf{Q} \in \hat{\delta}(\mathbf{Q})$. The trace $tr(\mathbf{Q} \rightarrow \mathbf{Q})$ is the trace of the fixpoint, we define $ftr(\mathbf{Q}) = tr(\mathbf{Q} \rightarrow \mathbf{Q})$.

The fixpoints for each component of $\mathbf{S1}_{p0,0}$ (**S1** with start position p_0 and start velocity *0)* are obtained as follows:

$$Y(\mathbf{Wander}) = (\mathbf{Out}_{cv,v}\ ;\mathbf{Move}_v \#\ \mathbf{Delay}_t)\ ;\ \mathbf{Wander}$$
$$\textit{for some } v \in R$$
$$Y(\mathbf{Base}_{p0,0}) =\ (\ \mathbf{Moving}_{p,v}{<}q{>}\ \#\ \mathbf{In}_{cv}{<}v{>}\ )\ ;\ \mathbf{Base}_{q,v}$$

and the behavior of **S1** up to this first communication point is captured by the set of traces $T1{=}ftr(\mathbf{S1}_{p0,0})$, and $tr(\mathbf{S1}_{p0,0}) = T1^{i,}$ $i \geq 1$. However, *T1* contains information about all the processes in the controller and the sensory and motor interface, as well as the state of the environment. We need to restrict the trace information to just the state of the robot and environment. In this example, this is simply the position of the robot (the $\mathbf{At}_p$ process), and the time (the $\mathbf{Delay}_t$ process).

The restriction of a trace *tr* to a set *S* is written $tr \downarrow S$ and is defined as the transformation of *tr* by eliminating all events relating to processes in a given set *S*, but preserving the order of remaining events. Let us define *State={***At**, **Delay***}:*

$$T1 \downarrow State = pref\ \{ s \mid s = [\ \mathbf{Delay}_t\ .\ \mathbf{At}_{p0}\ ] \}$$

There will eventually be a transfer of information across the port *cv*, described by the transition:

$$\mathbf{S1}_{p0,0} \rightarrow (\ \mathbf{Wander} \mid \mathbf{Base}_{p0,v}) = \mathbf{S1}_{p0,v}$$

The analysis of fixed points of each component can be repeated. $Y($**Wander**) remains the same, but

$$Y(\mathbf{Base}_{p0,v}) = (\ \mathbf{Moving}_{p0,v}{<}q{>} \ \#\ \mathbf{In}_{cv}{<}v{>}\ ) \ ;\ \mathbf{Base}_{q,v}$$

and the trace of this fixpoint is

$$ftr(\mathbf{S1}_{p0,v}) \downarrow State = pref\{\ s \mid s = [\ \mathbf{Delay}_{\mathrm{t}}\ .\ \mathbf{At}_{p0+vt}\ ]\ \}$$

and the traces of $\mathbf{S1}_{p0,v}$ before the next communication are

$$tr(\mathbf{S1}_{p0,v}) \downarrow State = pref\{\ s \mid$$
$$s = (\ [\ \mathbf{Delay}_{\mathrm{t}}\ .\ \mathbf{At}_{p0+ivt}\ ]\ )^{i}\ i \geq 1\ \}$$

where $(A_i)^i$ is the sequence $A_0.A_1.A_2...A_{i-1}$. There is an intuitive ordering on the sets in $ftr(\mathbf{S1}_{p0,v}) \downarrow State$ based on the length of the trace, with the longer traces representing a greater movement of the base from the start point. We will denote the longest trace in a set of traces tr as $\wedge tr$.

We can iteratively generate fixpoints for **S1**. Let us denote these as **F1**, **F2**, …,**Fn**, where $\mathbf{F1}=Y(\mathbf{S1}_{p0,0})$, $\mathbf{F2}=Y(\mathbf{S1}_{p0,v0})$, $\mathbf{F3}=Y(\mathbf{S1}_{p0+iv0t,v1})$, etc. The traces for **F1** become prefixes for the traces in **F2**, which in turn are prefixes for those in **F3**, etc. The last position for the largest trace of **F2** (p_0+iv_0t) becomes the first position of every trace in **F3**, etc. The final position of the base can be thus captured by the concatenated trace of the maximum of each trace set:

$$\wedge tr(\mathbf{F1}).\ \wedge tr(\mathbf{F2}).\ \wedge tr(\mathbf{F3}).,\dots,.\ \wedge tr(\mathbf{Fn})$$

The function $last(tr)$ returns the last element of a trace.

$$last(\wedge tr(\mathbf{F1})).last(\wedge tr(\mathbf{F2})).,\dots,.\ last(\wedge tr(\mathbf{Fn}))$$
$$= \mathbf{At}_{p0+n.v0.t}\ .\ \mathbf{At}_{p1+n1.v1.t}\ .\ \ \dots\ \ \mathbf{At}_{pm+nm.vm.t}$$

This is the definition of a random walk, $p_n = p_{n-1} + nv_{n-1}t$. A random walk on a 2D grid has a probability of unity of eventually visiting every point on that lattice.

Let us consider the computational effort in working out the answer. If we map the basic processes in **S1** to states, then we arrive at 5 states in the controller **Wander** and 6 states in the environment **Base**. This mapping of processes to states may be off by a constant factor, but it suffices to show the reduction in computational complexity. The shuffle produced analysis approach involves generating and exploring 30 states therefore, most of which involve no interaction between controller and environment. Our analysis involved checking only the 5 states in **Wander** and then the 6 in **Base**. However, we had to treat **Base** twice: once for the initial conditions (before any communications), and once after the first communication has happened. This gives us a total of 17 states explored, and improvement of 43%. The initial conditions just verify that no motion occurs before the first velocity command. By restricting our attention to the first communication events, we could gain an improvement of 63%.

VIII. TERRAIN FACTORS

A very common problem with wheeled bases, outdoors or indoors, is slippage between the wheel and the terrain [2]. A process network that allows modeling of a number of different slippage conditions for a base with two drive wheels will now be introduced.

The **Base** process is redefined as follows:

$$\mathbf{Base}_{(wl,wr)} = (\ \mathbf{Moving}_{(wl,wr)}\ \#\ \mathbf{In}_{cv}{<}(ul,ur){>}\ )\ ;\ \mathbf{Base}_{(ul,ur)}$$
$$\mathbf{Moving(}_{(wl,wr)} = \mathbf{Wheels}_{(wl,wr)}\ ;\ \ \mathbf{Moving}_{(wl,wr)}$$
$$\mathbf{Wheels}_{(wl,wr)} = (\mathbf{Lwheel}_{wl} \mid \mathbf{Rwheel}_{wr})\ \#\ \mathbf{Delay}_{t}$$

Where for convenience $w=(wl,wr)$ is defined to be the rotational velocity command for the drive wheels. **Moving** sends the velocity information to two processes representing the wheels. Each wheel process translates rotational velocity to translational velocity subject to interference by slippage.

$$\mathbf{Lwheel}_{w} = (\ \mathbf{Slip}_{w}{<}u{>}\ ;\ \mathbf{Out}_{pl,w.u}\ )\ ;\ \mathbf{Lwheel}_{w}$$
$$\mathbf{Rwheel}_{w} = (\ \mathbf{Slip}_{w}{<}u{>}\ ;\ \mathbf{Out}_{pr,w.u}\ )\ ;\ \mathbf{Rwheel}_{w}$$

The **At** state process is redefined to have the location p, the orientation θ, and wheel velocities of the base:

$$\mathbf{At}_{p,\theta,(vl,vr)} = ((\mathbf{In}_{pl}{<}vl{>} \mid \mathbf{In}_{pr}{<}vr{>})\ \#\ \mathbf{Delay}_{ta})\ ;\ \mathbf{At}_{p+p1,\theta+\theta 1,(vl,vr)}$$

where $p1 = 0.5(vl+vr)rt_a\vec{\theta}$ and $\theta 1=(vl-vr)rt_a/d$, d is the distance between the wheels, and r is the wheel radius.

Consider the following model of the slippage process (where **Ran** is the basic process mentioned in Section 5):

$$\mathbf{Slip3}_{w} = (\ \mathbf{Ran}_{R}{<}\ i\ {>}\ ;\ (\ \mathbf{LST}_{i,k}{<}0{>} \mid \mathbf{GTE}_{i,k}{<}w{>}\ )\ )$$
$$for\ constant\ k \in R$$

This model represents a highly nonlinear stop/start or jerky slippage typical of real bases. Let us consider moving the system with a fixed rotational velocity $v=(v,v)$ on each wheel with the objective of determining the spatial locus of endpoints in which the system can be, assuming this jerk/slip model. The full system is now:

$$\mathbf{S3}_{p,\theta,(v,v)} = (\ \mathbf{Move}_{(v,v)} \mid \mathbf{Base}_{(0,0)} \mid \mathbf{At}_{p,\theta,(0,0)}\ )$$

We start with $\mathbf{S3}_{\boldsymbol{p0,\theta 0,v}}$ which assumes that the base is initially at rest at position $p0$ and orientation $\theta 0$. We can look at the fixpoints of each of the three processes separately. Considering each basic process as a state, we have 2 states in **Move**, 20 states in **Base**, and 4 states in **At.** The shuffle product produces 160 states, the vast majority of which do not represent useful interactions. We need only look at 26 states (16%) to generate the fixpoints:

$$\mathrm{Y}(\mathbf{Move}_{(vl,vr)}) = \mathbf{Out}_{cv,(vl,vr)}\ ;\mathbf{Move}_{(vl,vr)}$$
$$\mathrm{Y}(\mathbf{Base}_{(vl,vr)}) = (\mathbf{Moving}_{(vl,vr)}\ \#\ \mathbf{In}_{cv}{<}(ul,ur){>})\ ;\ \mathbf{Base}_{(ul,ur)}$$

and

$$\mathrm{Y}(\mathbf{Base}_{(vl,vr)}) =$$
$$(\mathbf{Out}_{pl,\ r.vl}\ ;\ \mathbf{Lwheel}_{r,vl} \mid \mathbf{Out}_{pr,r.vr}\ \ ;\ \mathbf{Rwheel}_{r,vr})\ \ ;$$
$$\mathbf{Moving}_{(vl,vr)}\ \#\ \mathbf{In}_{cv}{<}(ul,ur){>}\ )\ ;$$
$$\mathbf{Base}_{(ul,ur)}$$
$$\mathrm{Y}(\mathbf{At}_{p,\theta,(vl,vr)}) = ((\mathbf{In}_{pl}{<}vl{>} \mid \mathbf{In}_{pr}{<}vr{>})\ \#\ \mathbf{Delay}_{ta})\ ;$$
$$\mathbf{At}_{p+p1,\ \theta+\theta 1,(vl,vr)}$$

The first two capture the transfer of the commanded velocity. The second two generate the motion of the base under slip with that velocity. We look at these further, assuming that the first velocity command $v=(v,v)$ has been sent. Restricting our attention to the interaction of the **Base** and **At** processes:

$$ftr(\ \mathbf{Base}_{(v,v)} \mid \mathbf{At}_{p,\theta,(v,v)}) \downarrow State =$$
$$pref\{\ s \mid s = [\mathbf{Delay}_{ta}\ .\ \mathbf{At}_{p+p1,\ \theta+\theta 1,(v,v)}\ ]\ \}$$
$$for\ p1 = 0.5(u_1v+u_2v)rt_a\vec{\theta}$$
$$\theta 1=(u_1v-u_2v)\ rt_a/d$$

$$u_1, u_2 \in \{0, 1\}$$

The values for u_1 and u_2 are generated randomly according to the **Slip3** process. Using the results from the previous section, we can see that

$$\begin{aligned} &tr(\mathbf{Base}_{(v,v)} \mid \mathbf{At}_{p,\theta,(v,v)}) \downarrow State = \\ &\qquad pref\, \{\, s \mid s = [\mathbf{Delay}_{ta} \,.\, \mathbf{At}_{p+pi,\ \theta+\theta i,(v,v)}]^i\ i \geq 1 \,\} \end{aligned}$$

We can produce the spatial envelope of travel by taking the last **At** process in the longest traces.

$$SpEnv = last(\wedge tr(\mathbf{Base}_{(v,v)} \mid \mathbf{At}_{p,\theta,(v,v)}) \downarrow State)$$

Figure 3 below shows results from a discrete Monte-Carlo simulation to generate *SpEnv*.

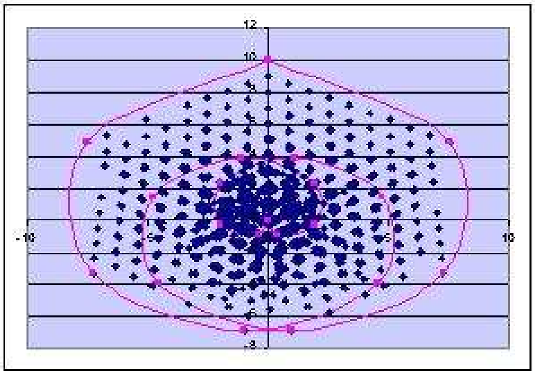

Figure 3: Simulation of Slip Envelope

Each point on Figure 3 is the position of one **At** in *SpEnv*. (The 'clumps' are due to the discrete nature of the simulation). If the furthest distance traveled is of interest then a little geometry indicates it is not necessary to generate all of *SpEnv* - only the traces in which slip occurs in the initial **At** processes up to a slip resulting in one half a revolution of the base. Additionally, only one of slip on the left and the right wheel needs to be generated and the other can be obtained by symmetry. This subset of *SpEnv* is shown as the outer contour in Fig. 3. Note that one only needs to generate enough that θ completes one full rotation. The contour in Fig.4 shows the distance for all initial slips up to the full length of the trace.

IX. DISCUSSION

This paper has addressed the difficult problem of obtaining performance guarantees for behavior-based robot system working in unstructured environments. The key issue in this problem is the difficulty of capturing the open-ended list of potential robot-environment interactions. We have presented a first step in solving the problem, an efficient way to model the asynchronous interactions of an environment and robot system, and we have demonstrated the application of the approach for wheel slippage in a mobile robot.

The examples in this paper used relatively simple environment and controller models as the objective was to demonstrate the efficiency of the proposed approach. Behavior-based methods work best in rich environments, in which there are many more active objects than just the robot base. The next step in this work is to apply our approach to this class of environment.

A second thrust of future research is to develop the automated form of the fixpoint and trace analysis used in this paper. A graph theoretic approach would appear to lend itself to this. Our ultimate objective is the construction of a software tool for *MissionLab* [13] and which allows the construction of robot controllers with specific performance guarantees.

REFERENCES

1. R. C. Arkin, Behavior-Based Robotics, MIT Press, Cambridge, MA, 1998.

2. Borenstein J., Wehe, D., *Internal Correction of Odometry Errors with the Omnimate.* In: proc. 7th Topical Meeting on Robotics & Remote Sys, April 1997, pp.323-329.

3. Emerson, E. A. (1990). Temporal and modal logic. In van Leeuwen, J., editor, *Handbook of Theoretical Computer Science*, pages 996-1072. Elsevier Science Publishers B.V.: Amsterdam, The Netherlands.

4. *Communicating Sequential Processes*, C.A.R. Hoare. Prentice Hall International Series in Computer Science, 1985.

5. *High-Integrity System Specification and Design*, M.G. Hinchey and J.P. Bowen. Springer-Verlag, London, 1999.

6. L. P. Kaelbling, *A Situated-Automata Approach to the Design of Embedded Agents*. SIGART Bull. 2(4): 85-88 1991.

7. Kosecka, J. (1996). *A Framework for Modeling and Verifying Visually Guided Agents*, *Analysis and Experiments*, Ph. D. dissertation, Dept of Computer and Information Science, Univ of Pennsylvania.

8. J. Kosek'a, H. Christensen, and R. Bajcsy, *Discrete event modeling of visually guided behaviors*, Int. Journal of Computer Vision, vol. 14, no. 2, pp. 179--191, March 1995.

9. Lyons D. M., and Arbib, M. A. *A Formal Model of Computation for Sensory-Based Robotics* IEEE Trans. Rob. Aut., vol. 5, no. 3 (June 1989), pp.280-293.

10. D. M. Lyons. *Representing and analyzing action plans as networks of concurrent processes*. IEEE Transactions on Robotics and Automation, V9 N3 June 1993 pp.241-256.

11. D. Lyons and A. Hendriks, *Exploiting patterns of interaction to achieve reactive behavior*, Artificial Intelligence 73, pp.117--148, 1995.

12. D.Lyons, *Discrete-Event Modeling of Misrecognition for PTZ Tracking.* IEEE Int. Conf. Advanced Video & Signal-based Surveillance, Miami FL 2003.

13. MacKenzie, D., Arkin, R.C., and Cameron, R., *Multiagent Mission Specification and Execution*, Autonomous Robots, Vol. 4, No. 1, Jan. 1997, pp. 29-52.

14. R. J. Ramadge and W. M. Wonham, 1987. *Supervisory control of a class of discrete event processes*. SIAM J. Control and Optimization, 25(1), pp. 206-230.

15. Concurrent and Real-time Systems: The CSP Approach, S. Schneider. Wiley, 1999.

16. Shapiro, Stuart C. (ed), "Encyclopedia of Artificial Intelligence", 2nd Ed., John Wiley & Sons, New York, 1992..

17. Martha Steenstrup, Michael A. Arbib, Ernest G. Manes, *Port Automata and the Algebra of Concurrent Processes*. JCSS 27(1): 29-50 (1983).

18. D.B. Stewart, R.A. Volpe, and P.K. Khosla, "Design of dynamically reconfigurable real-time software using port-based objects," *IEEE Trans. on Soft. Eng.*, vol.23, no.12, Dec. 1997.