



10-1-2012

Designing Autonomous Robot Missions with Performance Guarantees

Damian M. Lyons

Fordham University, dlyons@fordham.edu

Ronald Arkin

Georgia Institute of Technology

Prem Nirmal

Fordham University

Shu Jiang

Georgia Institute of Technology

Follow this and additional works at: http://fordham.bepress.com/frcv_facultypubs

 Part of the [Robotics Commons](#)

Recommended Citation

Lyons, Damian M.; Arkin, Ronald; Nirmal, Prem; and Jiang, Shu, "Designing Autonomous Robot Missions with Performance Guarantees" (2012). *Faculty Publications*. Paper 3.

http://fordham.bepress.com/frcv_facultypubs/3

This Conference Proceeding is brought to you for free and open access by the Robotics and Computer Vision Laboratory at DigitalResearch@Fordham. It has been accepted for inclusion in Faculty Publications by an authorized administrator of DigitalResearch@Fordham. For more information, please contact considine@fordham.edu.

Designing Autonomous Robot Missions with Performance Guarantees*

D.M. Lyons, *Member, IEEE*, R.C. Arkin, *Fellow, IEEE*,
P. Nirmal, *Student Member, IEEE*, and S. Jiang *Student Member, IEEE*

Abstract— This paper describes the need and methods required to construct an integrated software verification and mission specification system for use in robotic missions intended for counter-weapons of mass destruction (c-WMD) operations, as part of a 3-year effort for the Defense Threat Reduction Agency. The overall system architecture is described. The principal tool for verification is a process algebra, PARS, based on port automata theory. PARS is introduced, emphasizing its ability to represent probabilistic programs and uncertain and dynamic environments, followed by the analysis of mission properties for an example robotic mission.

Keywords; mobile robots, performance guarantees, formal properties, verification, robot programming.

I. INTRODUCTION

In an ongoing project for the Defense Threat Reduction Agency, we are developing methods to provide explicit performance guarantees for critical missions for autonomous and semi-autonomous robots. These specifically focus on counter-weapons of mass destruction operations, such as might be encountered in search, containment, and/or neutralization of chemical, biological or nuclear weapons, typically in urban indoor environments. Performance guarantees are essential for these missions as there may be only one opportunity to engage in the operation: failure may not be an option.

Toward that end, the project's goals include the design of a robotic software architecture that includes pre-mission performance analysis tools and methods for clearly presenting and confirming operator intention and acceptance of the mission, based on the level of success predicted and presented by said analysis. The system architecture also will allow for iterative refinement prior to deployment to maximize the likelihood of success derived for feedback from the verification methods, and the opportunity for the operator to make a go-no go decision based on these results. This research builds on our previously developed and usability-tested mission specification software system, *MissionLab*¹ [22][24] and earlier research on performance guarantees for similar systems [20]. The overall intent is to provide highly reliable performance bounds for autonomous robots operating in uncertain environments, so that the robot can *get it right*

* This research is supported by the Defense Threat Reduction Agency, Basic Research Award #HDTRA1-11-1-0038, to Georgia Tech. with subcontract to Fordham University.

D.M. Lyons and P. Nirmal are with the Dept. of Computer & Information Science, Fordham University, Bronx NY 10458, USA (Ph: 718-817-4485, Fx: 718-817-4488, Em: dlyons@cis.fordham.edu).

R.C. Arkin and S. Jiang are with the Mobile Robotics Laboratory, Georgia Institute of Technology, GA 30332, USA (Em: arkin@cc.gatech.edu).

¹ *MissionLab* is freely available for research and educational purposes at: <http://www.cc.gatech.edu/ai/robot-lab/research/MissionLab/>.

the first time.

In Section II we review relevant literature as a basis for presenting our System Architecture in Section III and our Verification Model in Section IV. Verifying probabilistic robot programs has unique challenges, including handling real-valued variables representing durations, positions and velocities, and an environment replete with dynamic and concurrent activities as well as uncertainty. Arguing from the unique characteristics of robot computation, we propose a concept, the system period, to analyze this computationally complex problem. We introduce a process algebra to analyze the structure of the system of robot program and environment model to identify the system period. In Section V, we introduce our principal verification algorithms using a series of examples and conclude the paper with a summary and discussion in Section VI.

II. LITERATURE REVIEW

Many robot software development frameworks exist today that can be potentially extended to include a verification component for the resulting robot program. Player/Stage is a robot programming environment developed at USC Robotics Research Lab [15]. Player was designed to be a robot device server that provides interface to a robot's sensors and actuators via TCP sockets. Pyro, or Python Robotics, is a Python-based programming framework that allows users to write robot-independent programs [7]. The authors of Pyro intended to use it as a tool for teaching robotics at a higher level without the students worrying about low-level control. URBI, a Universal Robotic Body Interface, is a programming environment based on the client/server architecture [6]. Microsoft Robotics Studios (MSRS) is a programming environment for robot control based on Windows OS [17]. MSRS also has a powerful 3D physics simulator for robot controllers. MSRS includes a visual programming language (VPL) that is translated into C# code for compilation. ROS, or the Robotic Operating System, is a software development framework that provides operating system like functionality for robot devices [25]. The Common Control Language (CCL) provides a mission programming environment tailored for multiple autonomous underwater vehicles (AUVs) [11]. CCL addressed the main issues of communication and coordination among AUVs. We chose *MissionLab* [24][22] as the software infrastructure to build our mission verification tool upon because (1) it has a usability-tested [21][14] graphical programming interface where a user can create her program as visual, finite state automata (FSAs), and (2) the high-level FSA is translated to a dataflow language [23] that is compatible with the proposed verification algorithm.

The automated verification problem for robot programs differs from the general automated verification problem addressed by the field of model checking [10][18] in several very important aspects. The behavior of a robot controlled

by a specific program will be different in different physical environments. Thus, we need to include a model of the environment as part of the verification problem. We represent the robot program as a process that communicates through its sensor and actuator processes with an environment process forming a single network of communicating processes as shown in Figure 1. In fact, for a realistic example, each of the processes shown in Figure 1 will consist of hierarchically nested process networks.

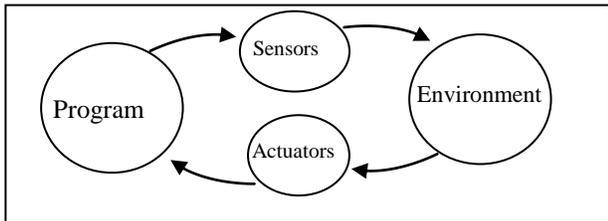


Figure 1: Program and Environment Network.

Discrete-Event Control (DEC) considers the analysis of a controller coupled to a plant model [26]. DEC approaches have been combined with model-checking software for the automated generation of higher-level robot controllers in [12][19]. While many of the techniques developed there can be applied to parts of our problem, we have to deal with the combined issues of representing real-valued variables concurrent activities and uncertainty. The variables may represent robot positions or velocities, or the duration or occurrence time of events, or the probability of a sensed feature or landmark. Concurrency is important to realistically represent the way the environment changes while the robot program is executing. The inclusion of uncertainty is important to handle the characterization of realistic environments as well as to probabilistic programs.

Software verification captures the effect of computation as a trajectory in the state-space that is the Cartesian product of the value sets for all the variables in the program. The reachability of states in this space can be investigated, within the limits of computability and the inherent exponential nature of the space², to determine whether a program will fail or succeed [10]. While tremendous progress has been made in this field [18], the additional variable, concurrency and uncertainty aspects that we add introduce combinatorial increases in the size of the combined state space of the program and environment system, rendering it prohibitively large to use reachability as a verification paradigm. A standard approach in the field is to search for regularities that can be leveraged to handle the state explosion, for example the *assume-guarantee* approach to modularizing a system, or the use of *fairness conditions* to eliminate undesirable states from analysis [18]. In Section IV we will introduce a regularity appropriate to robotics, the system period, leveraging the complexity reduction this allows.

Our main tool will be process algebra [5][16]: a formal model of concurrent computation in which processes can be built from other processes using composition operators. Our operators are similar to CSP operators but chosen for their

² E.g., run time is polynomial in the number of reachable locations but the number of reachable locations is exponential in the number of variables of each procedure during execution.

use in robot programs in particular. It does not include a built-in concept of optimization as does, e.g., the λ -calculus algebra [13]. The algebraic properties of the composition operators allow process descriptions to be transformed to investigate issues such as process equivalence as well as issues of liveness, safety, and deadlock.

In Section IV, we define a process algebra for robot programs that has an automaton semantics based heavily on our preliminary work in [20]. Certain reasoning is made simpler by this approach such as reasoning related to the periodic program structure and to the flow of variable values along communication channels.

III. SYSTEM ARCHITECTURE

The robotic software architecture is being built upon the *MissionLab* [22] robot programming environment. The goal of including a software verification component into the programming environment is to provide pre-mission performance analysis of the designed robot controller. The system architecture, Figure 2, supports three phases of the robot program development: design, verification, and execution. The design and verification phases plus the user form a robot program design loop that iterates until the mission-specific performance guarantee is satisfied or deemed unattainable. At this point, the operator can proceed to the execution phase with the confidence that the robot will meet the requisite mission requirements.

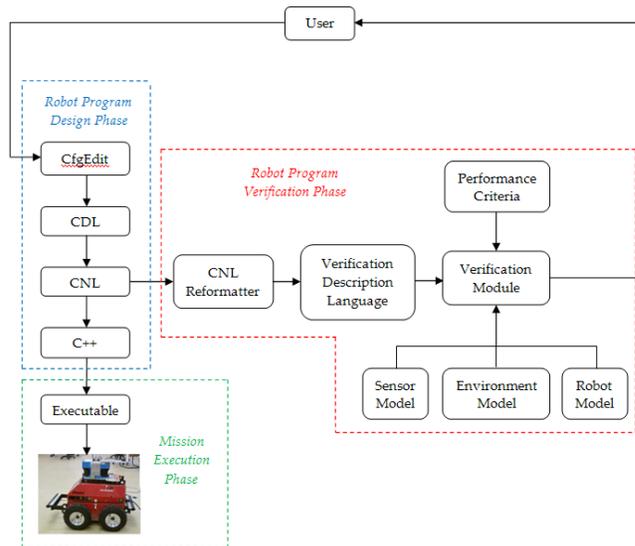


Figure 2: Robotic Software System Architecture

To illustrate the process of designing robot controllers, we step through the three development phases with a simple example: a controller for a robot to repeatedly go back and forth between two locations (A and B). This task, though simple, captures the main concept of a waypoint controller, and we will build on this in future work. The finite state automaton (FSA) of the *back_and_forth* robot is shown in Figure 3. While this example only uses one robot, our system design also supports programming and verification for multi-robot systems.

A. The Design Phase

During the design phase, the operator designs a controller

for a robot to complete a given mission using *CfgEdit* (Configuration Editor), Figure 4. *CfgEdit* is the graphical visual programming interface frontend of *MissionLab* where the configuration of the robot program is constructed as an FSA. The configuration of the robot program is based on the Configuration Description Language (CDL), a specification language that supports recursive compositions of robot behaviors that are physical robot-independent [24][22].

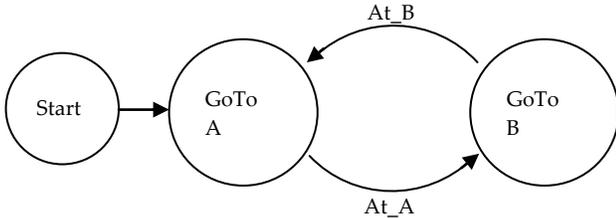


Figure 3: FSA for *back_and_forth* robot

The FSA in Figure 4 implements the *back_and_forth* robot in Figure 3. The *back_and_forth* behavior consists of two primitive *GoTo* behaviors, which are provided within a library of primitive behaviors in *MissionLab*. More complex programs can be constructed from this library of primitive behaviors for more complicated missions (e.g., biohazard search within a building).

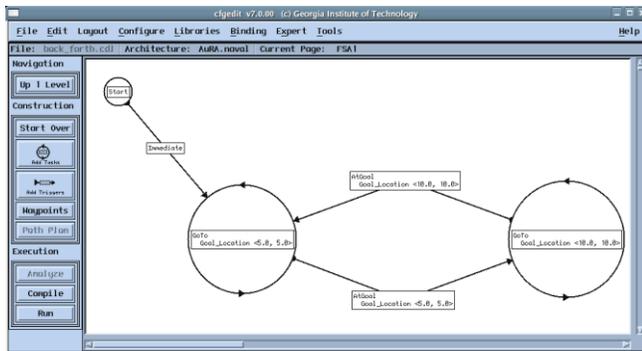


Figure 4: CfgEdit example for *back_and_forth* robot

To generate the C++ source code for the *back_and_forth* robot that can be compiled to control simulated or real robots, the CDL is first compiled into the Configuration Network Language (CNL) specification [23]. CNL specifies a robot program as a directed graph, where nodes represent primitive behaviors and edges represent dataflow links between behaviors. The CNL code is then compiled into C++ code. Finally, the C++ source code is compiled to generate robot executables for either simulation or mission execution using real robots.

The operator can simulate the controller design in *MissionLab* to verify her design intent before running it on a real robot. However, to obtain any guarantee that the robot will successfully complete the required mission in the real world target environment, a formal verification step is necessary before robot deployment. Even for the *back_and_forth* mission, the slippage between the robot's wheels and the floor in the real environment can cause the robot to fail the mission. One of the goals of the software verification system is to take into account the model of the environment (e.g., slippage) as well as noise and failure

models of the sensors and actuators, and provide pre-mission performance analysis of the controller design.

B. The Verification Phase

One crucial aspect of verification of robot programs that is different from traditional software verification is that robots have to interact with real environments both from a sensing and an actuation perspective. Uncertainty about the world makes it difficult to predict the exact response of the robot, which makes formal verification of robot software a unique challenge. Nonetheless, formal verification is necessary for critical missions (e.g., c-WMD missions such as finding, containing, and neutralizing Chemical-Biological-Nuclear (CBN) weapons) where failure is not an option.

The verification phase of the system design starts by translating the CNL specification of the robot program into the language required by the verification module, (see Fig. 2). The verification language, called *Process Algebra for Robot Schemas* (PARS), is presented in the next section. We use CNL as the input to the verification module because the CNL specification of the robot program is similar to the port automata based *Robot Schemas* (RS) framework [20].

For pre-mission performance analysis of a robot, it is necessary for the verification module to take into account the mission performance criteria and models of both the robot hardware and the operating environment. Our system includes three pre-constructed model libraries for verification: the robot model library, sensor model library, and environment model library (Fig. 5-7). The operator can select from these libraries for different combinations of robot platforms, sensors, and environments to match the mission's conditions. Similarly, for performance criteria, the operator is offered a selection of customizable criteria in terms of verification conditions and constraints.

Based on the choices of verification constraints and robot, sensor, environment models, the verification module tests the combination of the robot software with the chosen constraints and models for specific properties of safeness, liveness, and/or efficiency. At the end of verification, the verifier provides the operator with performance guarantee information regarding the robot carrying out the required mission under the specified conditions. If the result is unsatisfactory (e.g., some verification constraints are violated), the operator can use the feedback from the verifier to iteratively refine the robot program. In other words, besides simply telling the operator "yes/no" that the robot program satisfies the specified performance criteria, the verifier also identifies potential causes of failure in the program and provides the operator with this useful information, to assist in mission completion enhancement.

For the *back_and_forth* example, the operator can verify her robot controller by choosing a Pioneer robot with wheel encoders and noisy sonar sensors operating in an empty room with flat tiled floor and no obstacles. The operator can also specify a performance criterion such that the robot needs to be within 0.1 meters radius of each goal's spatial location for that leg to be considered successful. Other mission constraints such as time (e.g., maximum round trip time = 10 seconds) or minimum battery level can also be

specified. The verifier then conducts a pre-mission analysis and generate a performance guarantee of whether the robot will successfully accomplish the mission with specified performance criteria. If the verifier found that the robot cannot complete the mission successfully because the maximum round trip time constraint is violated, it reports failure along with the violated constraint(s). With the this information, the operator can make new design choices: e.g., increase the robot's speed or maximum round trip time.

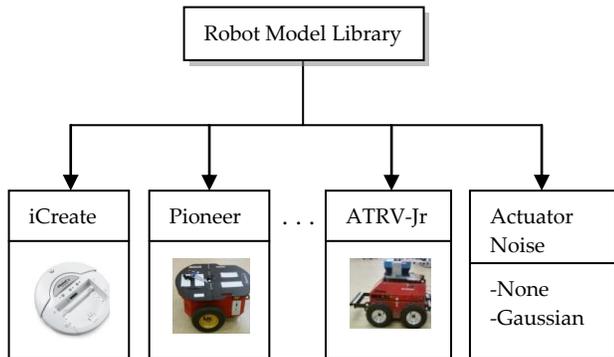


Figure 5: Notional example of robot model library

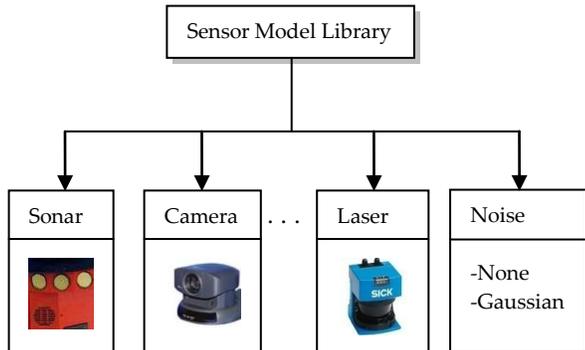


Figure 6: Notional example of sensor model library

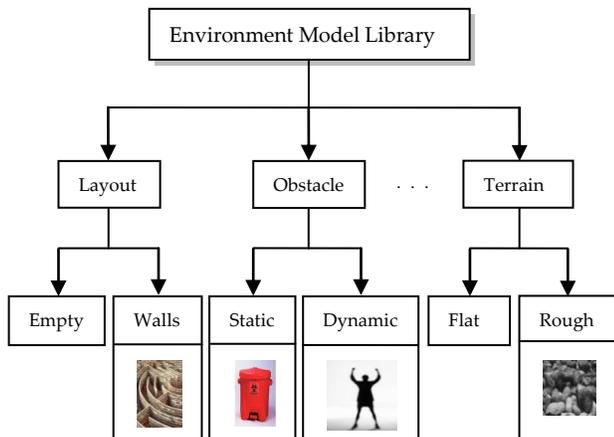


Figure 7: Notional example of environment model library

C. The Execution Phase

The operator makes a go-no go decision based on the verification result. If a satisfactory performance guarantee is provided, she proceeds to the execution phase with confidence that the robot will successfully complete the mission. Executing the mission with *MissionLab* is straightforward [22][24]. The robot program executes on the

robot directly or remotely from a base-station. For more details on robot, environment, and sensor libraries, see [4].

IV. VERIFICATION MODEL

This section (and the next) addresses the verification processing in Figure 2 in more detail. It begins by introducing a regularity that can be used to combat the exponential nature of the combined system state-space explained at the end of Section II. The process-algebra tool, PARS, we have developed to represent programs and environment is introduced. Examples of robot programs and environment models which illustrate use of real-value variables, concurrency and uncertainty are presented. We then show how tail-recursion in PARS can be used to represent and identify periodic behavior.

A. The System Period

To address the intractability of the combined system state-space, we identify a program regularity unique to behavior-based robotics, and we leverage this to modularize the verification problem. A behavior-based robot interacting with its environment [2] will respond to a specific set of environmental affordances as programmed by its behaviors. In this situation, which we refer to as a behavioral state, because the robot continually responds to the fixed set of affordances, a periodic regularity is induced in the combined state-space. Once an affordance is responded to, the robot may return to this behavioral state or move to another that handles a different set of affordances. However, the essence of the behavioral state is this potential to repeatedly handle a specific set of affordances. We will analyze the combined program and environmental models to identify the periodic structure imposed by behavioral states. First, we introduce the algebraic framework in which we situate this problem.

B. Verification language – PARS

The *process* is our basic unit of program and environment model structure. The *port automaton* [27] (PA) model, extended slightly, will provide the semantics for a process. We formalize processes as automata, and communication connections between processes as ports. The PA model is compositional: a composite *port connection automaton* can be constructed for a set of communicating PA. We formalize the ways in which the automata can be composed to a port connection automaton as process algebra composition operations. Our structural analysis will be performed primarily at the more abstract, process algebra level and not at the more detailed, PA level.

We will write a process P with initial parameter values $u1, u2, \dots$ and which produces final result values $v1, v2, \dots$ as:

$$P_{u1, u2, \dots} \langle v1, v2, \dots \rangle$$

It is understood that this process refers to a timed PA: a PA augmented with a duration map and a partitioned set of end states, which is written as:

$$P = (Q, L, X, \delta, \beta, d, T, \tau, \omega) \text{ where} \quad (1)$$

Q is the set of states
 L is the set of ports
 $X = (X_i | i \in L)$ is the event set for each port
 $\delta: Q \times XL \rightarrow 2^Q$ is the port transition function,

where $XL = \{(i, X_i) \mid i \in L\}$
 $d: Q \rightarrow Q \times T$ is the timed transition function,
where $dom(d) \cap proj_1 dom(\delta) = \emptyset$
 $\beta = (\beta_i \mid i \in L)$ $\beta_i: Q \rightarrow X_i$ output map for port i
 $\tau \in 2^Q$ is the set of start states
 $\omega \in 2^Q$ is the set of end states

and where there are two mappings

$$\begin{aligned} i(u1, u2, \dots) &= \tau0 \subseteq \tau \\ e(v1, v2, \dots) &= \omega0 \subseteq \omega \end{aligned}$$

that relate the initial parameter values to the starting states, and the final result values to the end states of the PA. We partition the set of end states ω into a set of stop end states ω_+ and a set of abort end states ω_- where a process that terminates in ω_+ is said to *stop* and a process that terminates in ω_- is said to *abort*.

A *basic process* corresponds to a PA defined using (1). All other processes are defined in terms of compositions of these processes. Examples of basic processes include:

- **Delay_t** is a process that stops a duration t after it has been started;
- **Ran _{ϕ} (v)** is a process that stops and returns a random sample v from a distribution ϕ .
- **In_c(y)** and **Out_{c,x}** are processes that perform input and output, respectively, on port c and then stop.
- **Eq_{a,b}**, **Neq_{a,b}**, **Gtr_{a,b}**, etc., are processes that *stop* when $a=b$, $a \neq b$, $a > b$, etc., respectively and *abort* otherwise.

Non-basic processes are defined using composition operators, e.g.:

$$\mathbf{T} = \mathbf{In}_{c1}(\hat{x}) ; \mathbf{Out}_{c2,x}$$

is process that inputs a value on port $c1$ and then outputs it on port $c2$. In our definition of sequential composition, if the first PA aborts (terminates in ω_-) rather than stops, the port connection automaton aborts without proceeding to the second PA. Mapping this to a standard programming language such as C, this one operation implements both the sequence (‘;’) and the conditional (‘if’) operations.

In concurrent composition, both PA execute at the same time. For example

$$\mathbf{T} = \mathbf{Out}_{c2,x} \mid \mathbf{In}_{c1}(\hat{x})$$

is a port connection of two PA, one that outputs a value on its port $c2$ and one that inputs a value from its port $c1$; we use here the simplification that similarly named input and output ports are connected to each other.

A *disabling* composition of two processes is written

$$\mathbf{T} = \mathbf{P} \# \mathbf{Q}$$

and denotes a port connection automaton of \mathbf{P} and \mathbf{Q} connected so that whenever \mathbf{P} terminates, it causes \mathbf{Q} to terminate, and vice-versa.

Having developed the PARS notation sufficiently far, we now present some examples of its use for robot programs.

C. Example Programs and Environments

We continue the running example program $\mathbf{BF}_{a,b}$ that moves the robot back and forth between two locations a and b specified in a global coordinates frame.

$$\mathbf{BF}_{a,b} = \mathbf{MoveTo}_a ; \mathbf{MoveTo}_b ; \mathbf{BF}_{a,b}$$

The **MoveTo** process moves the robot towards a location. Let the position of the robot be available on a port p and let the port v accept a velocity and then move the robot according to that velocity. We can write a very simple controller as:

$$\mathbf{MoveTo}_g = \mathbf{In}_p(\hat{r}) ; \mathbf{Neq}_{r,g} ; \mathbf{Out}_{v,s(g-r)} ; \mathbf{MoveTo}_g$$

where $s(d)$ maps a position difference to an appropriate velocity vector, pulling the robot towards the goal using the potential field [2] approach until the goal is reached.

In a simple model of the environment, the robot accepts the velocity on port v and instantly moves at that commanded velocity. The exact position of the robot at any time is available on port p , and the robot operates on flat, obstacle-free terrain:

$$\begin{aligned} \mathbf{DEnv}_r &= (\mathbf{Delay}_t \# \mathbf{DOdo}_r \# \mathbf{At}_r) ; \mathbf{In}_v(\hat{u}) ; \mathbf{DEnv}_{r+ut} \quad (2) \\ \mathbf{DOdo}_r &= \mathbf{Out}_{p,r} ; \mathbf{DOdo}_r \end{aligned}$$

This **DEnv** model implements a discrete integral of velocity (with time granularity t) to generate position. The actual position of the robot, at any time, is represented by \mathbf{At}_r .

In a more realistic model, there will be noise associated with the sensors and actuators:

$$\begin{aligned} \mathbf{NEnv}_{r,q} &= (\mathbf{Delay}_t \# \mathbf{NOdo}_q \# \mathbf{At}_r) ; \quad (3) \\ &(\mathbf{Ran}_{N(0,s1)}(\hat{e}) \mid \mathbf{In}_v(\hat{u})) ; \mathbf{NEnv}_{r+(u+\hat{e})t, q+ut} \\ \mathbf{NOdo}_q &= \mathbf{Ran}_{N(0,s2)}(\hat{e}) ; \mathbf{Out}_{p,q+\hat{e}} ; \mathbf{NOdo}_q \end{aligned}$$

Here, the velocity that the robot acts on is the command velocity u contaminated with a zero-mean normal error \hat{e} . The position that the robot reads from its odometry on port p is the actual position contaminated with a zero mean normal error. See [20] for other kinds of noise models, including terrain slip, in a similar process algebra framework.

We can also model static obstacles by modifying the environment dynamics so that if the robot tries to move through an obstacle, it collides with the obstacle and remains stuck.

$$\begin{aligned} \mathbf{Obs}_q &= \mathbf{Out}_{b,q} ; \mathbf{Obs}_q \quad (4) \\ \mathbf{OEnv}_r &= (\mathbf{Delay}_t \# \mathbf{Odo}_r \# \mathbf{At}_r) ; (\mathbf{In}_v(\hat{u}) \mid \mathbf{In}_b(\hat{q})) ; \\ &(\mathbf{Neq}_{q,r+ut} ; \mathbf{Set}_{r+ut}(\hat{nq}) \mid \\ &\mathbf{Eq}_{q,r+ut} ; \mathbf{Set}_{q+\hat{e}}(\hat{nq})) ; \mathbf{OEnv}_{nq} \end{aligned}$$

Here, $\mathbf{Set}_v(\hat{q})$ is a basic process that just sets output $q=r$.

The semantics of PARS allows for a rich description of process delays and probabilistic functionality. Stochastic properties that can be modeled include probabilistically delayed enabling/disabling, and synchronization as well as random variable values. The dynamic arrival of an object is captured by **ObsGen** (for $\phi \sim \text{Exp}(\lambda)$):

$$\mathbf{ObsGen} = \mathbf{Ran}_{\phi}(\hat{x}) ; \mathbf{Delay}_x ; \mathbf{Obj}$$

Probabilistically delayed arrival and termination, for example the existence and duration of difficult terrain patches in an uncertain environment, can be captured as:

$$\mathbf{TerGen} = (\mathbf{Ran}_{\phi}(\hat{t}) \mid \mathbf{Ran}_{\psi}(\hat{d})) ; \mathbf{Delay}_t ; (\mathbf{Delay}_d \# \mathbf{Rough}) .$$

Where $\phi \sim \text{Exp}(\lambda)$ and $\psi \sim N(\mu, \sigma)$ (a combination referred to as a *severity* [28]). This framework can represent previously unseen or unknown objects and events, and the adaptive/learning algorithms to handle them, as long as the potential for these events is written in the form above.

We will now look at how periodic behavior, as described in subsection A, can be represented and identified in PARS.

D. Recursive Processes

The tail-recursive (TR) expression:

$$\mathbf{T} = \mathbf{P}; \mathbf{T}$$

describes a process that repeats \mathbf{P} continually until it aborts. The semantics of a TR process is a port connection automaton that implements such a loop; an efficient ‘implementation’ of recursion. Mapping to a standard programming language such as C, this implements a ‘while’ loop. Boem and Jacopini [8] established that any language that implements sequence, condition and loop constructs is, in fact, sufficient to represent *any* program.

For convenience, the following notation is used for sequential compositions:

$$\mathbf{P}^n = \mathbf{P}^{n-1}; \mathbf{P} \text{ and } \mathbf{P}^1 = \mathbf{P}$$

This allows us to ‘unroll’ a TR process:

$$\mathbf{T} = \mathbf{P}; \mathbf{T} = \mathbf{P}^n; \mathbf{T} \quad n > 0 \quad (5)$$

We can interpret \mathbf{T} as a process that goes through multiple instances of the same computation, \mathbf{P} . That is, \mathbf{P} is the periodic aspect of \mathbf{T} . As it stands however, this iteration is history-less. We can include history if we add parameters:

$$\mathbf{T}_u = \mathbf{P}_u; \mathbf{T}_{f(u)} = \mathbf{P}_{f^{n-1}(u)}^n; \mathbf{T}_{f^n(u)} \quad n > 0, f^0(u) = u \quad (6)$$

\mathbf{T} offers a repeated interaction with its environment, but now the effect of previous events after the *i*th iteration is captured by the sequence of values of $f^i(u)$. In the case that \mathbf{P} can terminate with *abort* as well as *stop*, because of the conditional nature of the sequential composition, we get

$$\mathbf{T}_u = \mathbf{P}_u; \mathbf{T}_{f(u)} = \mathbf{P}_{f^{n-1}(u)}^n \quad n > 0 \quad (7)$$

We will refer to f as a *parameter-flow function*: a mapping $f: \mathbb{D}^m \rightarrow \mathbb{D}^m$ that relates the values of m parameters in the *n*th and (*n*+1)th iterations of the period \mathbf{P} . The use of flow functions allows us to handle the issue of real-valued variables ($\mathbb{D} = \mathbb{R}$) transformed by processes, and this feature is one of the principal motivations in using tail recursion as a loop construct.

Verification Approach: The period and parameter flow function associated with a TR process are a concise implicit representation of the entire state-space of the process and offer an alternative to explicit state enumeration.

E. Identifying the System Period

When we analyze a robot program operating in a given environment, we analyze a concurrent, communicating composition of the program and the environment (Fig. 1). If we have a set of TR process equations $\mathbf{P1}, \mathbf{P2}, \dots, \mathbf{Pm}$ that form a system \mathbf{Sys} through concurrent composition:

$$\mathbf{Sys} = \mathbf{P1} | \mathbf{P2} | \dots | \mathbf{Pm}$$

then an important question is, can this \mathbf{Sys} be rewritten in a TR form? That is, under what conditions can we develop an expression for the *period of the system* in terms of the *periods of its component processes*?

For a TR definition of process \mathbf{P} , the section of the process definition between the equal sign and the tail-recursion will be called the period \mathbf{P}'

$$\mathbf{P} = \mathbf{P}'; \mathbf{P}$$

If the component periods $\mathbf{P1}', \mathbf{P2}', \dots, \mathbf{Pm}'$ contain port communication, then those interactions sequence the component periods with respect to one another, forcing a partial sequence of computations. In the case that all ports’ communications can be matched, input to output, across the component periods, then a system period can be identified:

$$\mathbf{Sys} = \mathbf{Sys}'; \mathbf{Sys}$$

$$\mathbf{Sys}' = \mathbf{F}(\mathbf{P1}', \mathbf{P2}', \dots, \mathbf{Pm}')$$

Where $\mathbf{F}(\cdot)$ is a process composition obtained by matching the port communications in component periods. It may be necessary to unroll shorter component periods one or more times using results (5) & (6) to provide sufficient port communications for a longer component period (longer and shorter in this context refer to the number of port communication operations).

F. Implications and Computational Complexity

What is the practical implication of assuming that a system period exists for the combination of robot program and environment model? A system period will not exist in the cases that:

- the processes don’t communicate with each other;
- port communications are unmatched;
- an infinite sequence of unrollings is needed.

With respect to the first case, a robot program that does not actually interact with its environment can never achieve any level of performance! This is a major error that should be flagged. In the second case, unmatched communications means that some part of the robot program will block forever, a potential deadlock case that also needs to be flagged to the attention of the designer.

Finally, an infinite sequence of unrollings is evidence of a lack of periodic behavior. However, *MissionLab*’s FSA program model is based on the kind of behavioral states mentioned in subsection A.

The computational complexity of calculating the system period relies on the complexity of the matching process. If no explicit fan-in or fan-out is allowed, then the computational complexity is $O(n^2)$ where n is the number of input processes in the (final, unrolled) system.

V. VERIFICATION ALGORITHMS

Here we present by example our approach to the analysis of the system period to determine if a robot program achieves a performance guarantee in an environment.

A. Task Completion

A designer may wish to know if a specified system completes a task (*liveness*). In our approach, both the system and property to be verified are specified in PARS. The two process networks are compared and analyzed, and if the specified system can be shown *equivalent* to the property, we report success (cf. [1],[9]). For example, the designer may wish to know if the robot arrives and stays in position *a* after time *t1*. We can specify this verification property using PARS as:

$$\mathbf{Goal} = \mathbf{Delay}_{t1}; (\mathbf{Delay}_{t2} \# \mathbf{At}_a); \mathbf{Delay}_{t3}; \mathbf{Goal}$$

This states that the robot will be at position *a* after time *t1* and remain there at least a subsequent time *t2*. Notice that *t1*,

$t2$ and $t3$ are variables not constants. A property specification process network differs from a process network in that it is actually a *process network constraint expression*, a specification of a set of possible networks.

Consider the deterministic environment model **DEnv** from (2) for the **BF** program. The system in this case is

$$\mathbf{Sys} = \mathbf{DEnv}_{p0} \mid \mathbf{BF}_{a,b}$$

The first step in our verification approach is to find the system period from the component periods:

$$\begin{aligned} \mathbf{BF}'_{a,b} &= \mathbf{MoveTo}_a; \mathbf{MoveTo}_b \\ \mathbf{MoveTo}'_g &= \mathbf{In}_{p\langle r \rangle}; \mathbf{Neq}_{r,g}; \mathbf{Out}_{v, s(g-r)} \\ \mathbf{DEnv}'_r &= (\mathbf{Delay}_t \# \mathbf{Odo}_r \# \mathbf{At}_r); \mathbf{In}_{v\langle u \rangle} \end{aligned}$$

The parameter flow functions (that link variables values in the n^{th} iteration to those in the $n+1^{\text{th}}$ iteration) are:

$$\begin{aligned} \mathbf{BF}'_{a,b} \quad f_{\mathbf{BF}}(a,b) &= (a,b) \\ \mathbf{MoveTo}'_g \quad f_{\mathbf{MoveTo}}(g) &= g \\ \mathbf{DEnv}'_r \quad f_{\mathbf{DEnv}}(r) &= r+ut \end{aligned}$$

The only port constraints between the environment and the program are input on port p and the output on port v . The period of **DEnv** must be unrolled n times to match the first part of **BF**. To match the second part of **BF** we need to unroll **DEnv** again, yielding:

$$\begin{aligned} \mathbf{Sys} &= \mathbf{Sys}'_p \langle r \rangle; \mathbf{Sys}_r \\ \mathbf{Sys}'_p &= (\mathbf{Sys1}_{f^n(p)})^n; (\mathbf{Sys1}_{f^m \circ f^n(p)})^m \\ \mathbf{Sys1}_p &= \mathbf{DEnv}'_p \mid \mathbf{MoveTo}_a \end{aligned}$$

To generate the parameter flow function f for **Sys1**, we need to propagate values across the port connections between **DEnv'** and **MoveTo'**. We can associate a set of recurrence relations with the system flow functions:

Flow Function	Recurrence
$f_{\mathbf{Sys1}}(r) = r + s(g - r)t$	$r_{n+1} = r_n + s(g_n - r_n)t$
$f_{\mathbf{MoveTo}}(g) = g$	$g_{n+1} = g_n$

If we project (\downarrow) the network onto the processes in the property specification network, and if we reasonably assert that $\mathbf{Delay}^n = \mathbf{Delay}_{nt}$ then we have:

$$\begin{aligned} \mathbf{Sys}' \downarrow \{ \mathbf{Delay}, \mathbf{At} \} &= \\ \mathbf{Delay}_{(n-1)t}; (\mathbf{Delay}_t \# \mathbf{At}_{f^n(p)}) &; \\ \mathbf{Delay}_{(m-1)t}; (\mathbf{Delay}_t \# \mathbf{At}_{f^m \circ f^n(p)}) & \end{aligned}$$

We need to find a structural mapping between the **Sys'** and **Goal'** networks. In this case, structural mapping is trivial:

$$\begin{aligned} \mathbf{Delay}_{(n-1)t}; (\mathbf{Delay}_t \# \mathbf{At}_{f^n(p)}) &; \\ \mathbf{Delay}_{(m-1)t}; (\mathbf{Delay}_t \# \mathbf{At}_{f^m \circ f^n(p)}) &= \\ \mathbf{Delay}_{t1}; (\mathbf{Delay}_{t2} \# \mathbf{At}_a); \mathbf{Delay}_{t3} & \end{aligned}$$

To complete the mapping, the flow recurrences need to be solved for n and m for which $f^n(p0)=a$ in which case $t1=(n-1)t$ and $t2=mt$.

The automatic construction of system flow functions requires tracing a variable through a system period. The complexity is linear in the length of the period. It also requires substituting values communicated via port connections. This is linear in the number of concurrent processes in the period. We use existing tools (e.g. PURRS³) to solve flow functions.

B. Task Safety

A safety property states that ‘‘bad things’’ won't happen. We analyze the system period to make sure that the safety property is something that *is always addressed by the program*. Once again, we will use algebraic equivalence relations to re-structure the network, and map this re-structured network to the property to be verified.

Consider the **BF** program for the environment model (4) that includes obstacles, **OEnv**. A safety property in this example would be obstacle avoidance: *The robot is never within 1m of an obstacle*. As a process network, this is:

$$\mathbf{G} = (\mathbf{Odo}_p \mid \mathbf{Obs}_{p+q}); \mathbf{G} \text{ where } q > 1$$

We analyze the network consisting of an obstacle at location d with **BF** _{a,b} :

$$\mathbf{Sys} = \mathbf{Obs}_d \mid \mathbf{OEnv}_{p,0} \mid \mathbf{BF}_{a,b}$$

As before, we obtain the periods of each component process and construct the system period.

$$\begin{aligned} \mathbf{BF}'_{a,b} &= \mathbf{MoveTo}_a; \mathbf{MoveTo}_b \\ \mathbf{MoveTo}'_g &= \mathbf{In}_{p\langle r \rangle}; \mathbf{Neq}_{r,g}; \mathbf{Out}_{v, s(g-r)} \\ \mathbf{Obs}'_q &= \mathbf{Out}_{b,q} \\ \mathbf{OEnv}'_r &= (\mathbf{Delay}_t \# \mathbf{Odo}_r \# \mathbf{At}_r); (\mathbf{In}_{v\langle u \rangle} \mid \mathbf{In}_{b\langle q \rangle}); \\ &(\mathbf{Neq}_{q,r+ut}; \mathbf{Set}_{r+ut\langle nq \rangle} \mid \mathbf{Eq}_{q,r+ut}; \mathbf{Set}_{q+e\langle nq \rangle}) \end{aligned}$$

Since the system period *must always satisfy* the safety property, the first step is a connectivity check. Starting at **Odo** and at **Obs** (the processes in the safety property) we follow the port and variable value connections, dividing the system period network into equivalence classes of processes. Unless **Odo** and at **Obs** fall into the same class, there is no way to guarantee the safety property. In this case, since there is no connection from the obstacle process to any part of the **BF** network, the safety property fails.

To address the safety property, we need to introduce a sensor that can report on the object location:

$$\mathbf{Sensor}_c = (\mathbf{In}_{b\langle q \rangle} \mid \mathbf{In}_{p\langle r \rangle}); \mathbf{Gtr}_{|r-q|,c}; \mathbf{Out}_{sp\langle q \rangle}; \mathbf{Sensor}_c$$

This sensor accurately reports on the location of the obstacle as long as it is less than c from the robot. Modifying **BF**:

$$\begin{aligned} \mathbf{MoveTo}_g &= (\mathbf{In}_{p\langle r \rangle} \mid \mathbf{In}_{sp\langle q \rangle}); \mathbf{Neq}_{r,g}; \\ &(\mathbf{Gtr}_{|r-q|,h}; \mathbf{Out}_{v, s(g-r)} \mid \\ &\mathbf{Lte}_{|r-q|,h}; \mathbf{Out}_{v, s(r-q)}); \mathbf{MoveTo}_g \end{aligned}$$

If the obstacle is within a distance $h > c$ the robot will stop moving towards the goal (velocity $s(g-r)$) and instead move away from the obstacle. Because of the additional connectivity to the **Sensor** _{c} process, on examination of the period of the system, we see that **Odo** and **Obs** fall into the same equivalence class. We have to verify that the parameters to **Odo** and **Obs** satisfy the safety property. We construct the system period flow function as before, projecting the system period onto the goal process network, yielding:

$$f_{\mathbf{Sys1}}(r) = r + \begin{cases} |r-d| > h & s(g-r) \\ |r-d| \leq h & s(r-d) \end{cases}$$

We can match the safety property network if $h > 1$, and hence verifying the safety property.

C. Stochastic properties

Now we consider a verification example that includes uncertain environment. The specification goal will be: *The*

³ <http://www.cs.unipr.it/purrs>.

robot arrives to position a within $10t$ time units at least 90% of the time. This can be specified in PARS as:

$$\mathbf{Goal} = \mathbf{Ran}_{\Phi}(x); \mathbf{Delay}_t; \mathbf{At}_a,$$

where Φ is a distribution such that $P(x \leq 10t) \geq 90\%$.

Consider the non-deterministic environment model **NEnv** in (3). When we analyze the system (**NEnv** | **MoveTo**), we now get the recurrence relation:

$$r_{n+1} = r_n + (u_n + e_n)t,$$

where $e_i \sim N(0, \sigma^2)$ is a normally distributed random variable. To handle uncertainty, we need to move from the case of variables with a single value to variables with a distribution of values. The domain \mathbb{D}^m of the flow function now becomes distributions and the recurrence becomes

$$r_{n+1} = r_n * (u_n + \Psi)t$$

Where $\Psi = N(0, \sigma^2)$ and $*$ denotes convolution. If we simplify our environment models to capture uncertainty and noise with just normal distributions, or mixtures of normal distributions, this expression can be solved efficiently. The sum of normal distributions is also normal $N(0, n\sigma^2)$.

If we include Poisson processes in our environment, to model arrival times, then we will need to also allow for exponential-normal interaction. A very important example of this is the severity terrain model that we presented at the end of section IV.C. In terms of the recurrence relation above, this would mean that in addition to $e_i \sim N(0, \sigma^2)$ the solution $n \sim \text{Exp}(\lambda)$. A form for the severity distribution is developed in [28] by Yang and Nadarajah.

VI. DISCUSSION

This paper addresses the pressing need to be able to verify mission-critical robotic software in the context of its environment and specific physical hardware (sensors and actuators) with the goal of providing performance guarantees to an operator prior to their sending a robot into a potentially hazardous situation. To do so we have developed a software architecture that provides visual programming capabilities and an internal language that is amenable to such analysis.

To verify mission critical robot programs, we need to analyze a combination of program and environment model, each of which contains real valued variables, concurrent activities and uncertainty. A unique regularity of behavior-based robot computation, the system period, is introduced to render this complex problem more tractable. A process algebra, PARS, is introduced to represent and analyze programs and their environments. The expressiveness of PARS for both deterministic and stochastic processes, as well as the proposed verification approach, is illustrated on a simple back and forth robotic mission.

We are currently in the first year of a 3-year program to develop and integrate the existing software and incorporate the new ideas described herein. We anticipate the final system to be able to operate in a range of environments, using multiple types of hardware platforms, specified over teams of robots.

References

[1] Alur, R., Theory of Timed Automata. *Theoretical Computer Science*, v. 126 Issue 2, pp. 183-235, 1994.

[2] Arkin, R.C., Motor Schema-Based Mobile Robot Navigation. *Int. Journal of Robotics Research*, Vol. 8, No. 4, Aug. 1989, pp. 92-112.

[3] Arkin, R.C., Diaz, J., Line of Sight Constrained Exploration for Reactive Multiagent Robotic teams. *AMC02*, July 2002, pp. 455-461.

[4] Arkin, R.C., Lyons, D.M., Jiang, S., Nirmal, P., Zafar, M., Getting it Right the First Time: Predicted Performance Guarantees from the Analysis of Emergent Behavior in Autonomous and Semi-autonomous Systems, *SPIE Defense, Security and Sensing: Unmanned Systems Technology XIV*, Baltimore MD, 2012.

[5] Baeten, J., A Brief History of Process Algebra. *Elsevier J. Theoretical Comp. Sci. – Process Algebra*, 335(2-3), 23 May 2005

[6] Baillie, Jean-Christophe, URBI: Towards a Universal Robotic Low-Level Programming Language. *Proc. IROS*, 2005.

[7] Blank, D.S., Kumar, D., Meeden, L., and Yanco, H., Pyro: A Python-based Versatile Programming Environment for Teaching Robotics. *J. Educational Resources in Computing (JERIC) 2003*.

[8] Boem, C. and Jacopini, G., “Flow diagrams, Turing machines and languages with only two formation rules, *CACM* 9(5) 1966.

[9] Buchholz, P., Equivalence Relations for Stochastic Automata Networks. *Proc. 2nd Int. Meeting on the Numerical Solution of Markov Chains*, 1995

[10] Clark, E., et al., *Model Checking*. MIT Press 1999.

[11] Cuarte, C., et al, “A Common Control Language to support multiple cooperating AUVs” *Proc. 14th Int. Symp. Unmanned Untethered Submersible Vehicles Technology*, 2005

[12] Ding, X., Kloetzer, M., Chen, Y., and Belta, C., Automatic Deployment of Robotic Teams. *Robotics & Automation Magazine* 18(3) Sept 2011.

[13] Eberbach, E., “ \mathcal{S} -calculus of bounded rational agents: flexible optimization as search under bounded resources in interactive systems.” *Fundamenta Informatica*, V68 p47-102, 2005.

[14] Endo, Y., MacKenzie, D., and Arkin, R.C., Usability Evaluation of High-level User Assistance for Robot Mission Specification. *IEEE Trans. SMC* Vol. 34, No. 2, pp.168-180, May 2004.

[15] Gerkey, B., Vaughan, R., Howard, A., The Player/Stage Project: Tools for Multi-Robot and Distributed Sensor Systems. *11th Int. Conf. Adv. Robotics*, pp. 317-323, Coimbra Portugal, June 2003.

[16] Hennessy, M. *Algebraic Theory of Processes*. The MIT Press, 1988.

[17] Jackson, J., Microsoft Robot Studio: A Technical Introduction. *IEEE Rob. & Aut. Magazine*, Dec. 2007.

[18] Jhala, R., Majumdar, R., Software Model Checking. *ACM Computing Surveys*, V41 N4, Oct. 2009.

[19] Kress-Gazit, H., Wongpiromsarn, T., Corrective, Reactive, High-level Control. *Robotics & Automation Magazine* 18(3) Sept 2011.

[20] Lyons, D., Arkin, R., Towards Performance Guarantees for Emergent Behavior. *IEEE Int. Conf. on Robotics and Aut.*, 2004.

[21] MacKenzie, D., Arkin, R., Evaluating the Usability of Robot Programming Toolsets. *Int. Journal of Robotics Research*, Vol. 4, No. 7, April 1998, pp. 381-401.

[22] MacKenzie, D., Arkin, R.C., Cameron, R., Multiagent Mission Specification and Execution. *Aut. Robots*, 4(1), pp. 29-52, 1997.

[23] MacKenzie, D.C., *Configuration Network Language (CNL) User Manual*. College of Computing, Georgia Tech., V 1.5, June 1996.

[24] MissionLab v7.0 User Manual, available at http://www.cc.gatech.edu/aimosaic/robotlab/research/MissionLab/mlab_manual-7.0.pdf

[25] Quigley, M., Gerkey, B., Conley, K., Faust, J., Foote, T., Leibs, J., Berger, E., Wheeler, R., and Ng, A., ROS: An open-source robot operating system. *Proc. Open-Source Software Workshop, Int. Conf. Robotics and Aut.*, 2009.

[26] Ramadge, R., Wonham, W., Supervisory control of a class of discrete event processes. *SIAM J. Cont. & Opt.* 25(1) 1987.

[27] Steenstrup, M., Arbib, M.A., Manes, E.G., *Port Automata and the Algebra of Concurrent Processes*. *JCSS* 27(1): 29-50 (1983).

[28] Yang, D.W., Nadarajah, S., Drought modeling and products of random variables with exponential kernels *Stoch. Env. Research and Risk Ass.* V21 N2 2006.