Performance Verification for Behavior-based Robot Missions

Damian M. Lyons, *Senior Member, IEEE*, Ronald C. Arkin, *Fellow, IEEE*, Shu Jiang, *Student Member, IEEE*, Tsung-Ming Liu, *Student Member, IEEE*, Paramesh Nirmal, *Student Member, IEEE*

***Abstract*—Certain robot missions need to perform predictably in a physical environment that may have significant uncertainty. One approach is to leverage automatic software verification techniques to establish a performance guarantee. The addition of an environment model and uncertainty in both program and environment, however, means the state-space of a model-checking solution to the problem can be prohibitively large. An approach based on behavior-based controllers in a process-algebra framework that avoids state-space combinatorics is presented here. In this approach, verification of the robot program in the uncertain environment is reduced to a filtering problem for a Bayesian Network. Validation results are presented for the verification of a multiple-waypoint and an autonomous exploration robot mission.**

***Index Terms*— Program Verification, Autonomous Agents, Behavior-based Systems, Control Architectures and Programming.**

I. Introduction

In research being conducted for the Defense Threat Reduction Agency (DTRA), we are concerned with robot missions that may only have a single opportunity for successful completion, with serious consequences if the mission is not completed properly. In particular the focus is on missions for Counter-Weapons of Mass Destruction (C-WMD) operations, which require discovering a WMD within a structure and then either neutralizing it or reporting its location and existence to the command authority. Typical scenarios consist of situations where the environment may have significant uncertainty, and have time-critical performance requirements. The goal is to provide reliable performance guarantees for whether or not the mission as specified may be successfully completed under these circumstances. Towards that end, a set of specialized software tools have been developed to provide guidance to an operator/commander prior to deployment of a robot tasked with such a mission. These tools can be highly valuable in other settings also – for example, in a manufacturing setting to verify performance or safety whenever anything is changed.

A. *Automatic Verification*

Automatic verification of software is a very desirable functionality in any application where software failure can incur heavy penalties [1]. While a general solution is ruled out by the undecidability of the halting problem, much research has been conducted on restricted instances of the problem. Model checking [2] [3] is a collection of techniques that conduct an exhaustive exploration of the state-space of a program to determine whether the program satisfies a temporal logic constraint on its behavior.

More recently, some researchers have effectively leveraged model-checking techniques to address the *correct-by-construction* robot control problem [4] [5]. A solution to the correct-by-construction problem takes as input a temporal logic description of the desired behavior of the robot controller and then fabricates a controller guaranteed to abide by this description.

The problem addressed by this paper differs from the correct-by-construction problem, and is similar to the general-purpose software verification problem, in that the input is mission software designed using the *MissionLab* toolkit [6], and the objective is to verify that this software abides by a performance constraint. It is similar to the correct-by-construction problem in that we require a model of the environment in which the software is to be carried out, something not typically explicit in general-purpose software verification [3].

However, the problem addressed by this paper differs from both in needing to efficiently process probabilistic software and environment models, continuous environment characteristics and asynchronous and concurrent environment dynamics. These problem aspects are troublesome for model-checking approaches: One of the biggest contributions to state-space explosion in model-checking is the translation from program to formal model. It is exponential in the number of program variables. Asynchronous concurrent modules are another formidable contributor to complexity, since the concurrent system state space grows as the Cartesian product of the component spaces.

B. *Process-Algebra Approach*

For all of these reasons, the approach to the problem presented in this paper focuses on avoiding an explicit state-space representation and especially one in which the number of program variables will introduce exponential complexity. Instead a process-algebra representation is leveraged to develop a solution in which the program is translated to a set of equations over the program variables (which can include random variables). The reason a process algebra is used is that it can formally capture the concept of a recursive process and

Submitted for review June 2013. **Resubmitted April 2014.** This research is supported by the Defense Threat Reduction Agency, Basic Research Award #HDTRA1-11-1-0038.

D. Lyons, T-M Liu and P. Nirmal are with the Department of Computer and Information Science, Fordham University NY 10458 (email: {dlyons, tliu17, pnirmal}@fordham.edu).

R. Arkin and S. Jiang are with the College of Computing, Georgia Institute of Technology GA 30332 (email: {arkin, sjiang}@cc.gatech.edu).

the way variables are modified by a process during recursion in a direct and concise manner.

In overview, the proposed approach will leverage four results.

- First, it is demonstrated that a process written in a *tail-recursive* (TR) form admits the extraction of a *flow function f*, a function that maps the values of the process variables in one recursion to that in the next. Any behavior of the process can be tested by inspecting f^n where $n \geq 0$ is a positive integer.
- Second, it is shown that under certain assumptions, any concurrent communicating network of TR processes can be rewritten as a single TR process.
- Third, it is shown that *MissionLab* missions can be mapped to a network of TR processes.
- Fourth, the set of flow functions from the network of TR processes can be tested for a performance constraint by mapping them to a Dynamic Bayesian Network and applying a filtering algorithm.

C. Performance Guarantee

Using process-algebra as the formal representation for the mission software means that there is the option to also use this, rather than a temporal logic, as the language for the performance guarantee and for the description of the environment models. When process-algebra is used for specification [7] [8], a major difficulty encountered is specifying proscription (e.g., the safety property that the robot does *not* collide). The performance guarantee used here separates constraints on process ordering from conditions on variable values, enabling proscription.

D. Environment Models

It is not proposed that *MissionLab* designers build, in detail, their own environment models (including robot and sensor models) against which to verify the mission. Instead, it is proposed that a set of standard environment models be constructed a-priori and provided as a library from which robot, sensor and environment features can be selected and composed automatically into an environment model.

The process-algebra used here employs communication ports and port-to-port connections [9] for concurrent modules. This facilitates specifying plug-and-play compatible environment models, since the formal model of the mission software just communicates over a set of ports with any selected environment model. The development of a standard set of environment models is not something we have pursued however beyond those we have developed and used in validation.

E. Validation

To demonstrate the accuracy of the verification results achievable by the method proposed here, predicted performance guarantees are validated by carrying out physical robot experimentation. Calibration data is collected on the robots and sensors used in missions, and suitable environment models constructed. Both of the example missions presented in the paper are verified and validated. Because the resulting robot/environment system is probabilistic, the verification answer is not a binary yes/no, but a probability landscape capturing the system's performance. Each mission is validated by carrying out multiple physical runs and collecting performance statistics on real robots. The validation and verification results are then compared to evaluate the quality of the verification prediction.

The remainder of this article is organized as follows. Section II presents a review of prior work. Section III introduces robot mission design using *MissionLab* and the two example missions which will later be verified and validated. Section IV introduces the process-algebra, PARS (Process Algebra for Robot Schemas), for representing *MissionLab* mission programs, robot, sensor and environment models, and performance guarantees. Section V builds the process algebra results on which VIPARS (Verification in PARS), the verification module within *MissionLab,* is based. Section VI maps these results to the filtering problem for Bayesian Networks. Section VII presents the verification of the two example missions and the experimental validation of those verification results. Section VIII summarizes the contributions of the paper and discusses the next key research challenges in extending this approach.

II. Prior Work

Model checking has been a very successful approach to the automatic verification of software [2]. A program is cast as a state-based transition system in which states are labeled with sets of logical propositions, a *Kripke* system. This labelling means that logical formulas may be satisfied by a state, and temporal logical formulas by sequences of states. The instructions in the program map values from one state to a successor state. If the program has n variables, and if each variable r_i can have values from a set $val(r_i)$, then the state space of the program is $\Pi_i\ val(r_i) = val(r_0) \times \ldots \times val(r_{n-1})$ [2]. The verification problem in model-checking is, at its heart, a test of the reachability of a state or set of states from the start state given the program instructions. The combinatorics involved in $\Pi_i\ val(r_i)$ have always been clear, and model-checking approaches are typically divided into enumerative methods that search this (perhaps huge) graph of states, and symbolic methods which instead explore (a smaller number of) sets of these states [3].

Automated verification of robot and multirobot software has several characteristics that distinguish it from general purpose software verification. The first is that the robot program does not execute based on static inputs, but rather interacts with an environment in an ongoing fashion. This is recognized in the related field of discrete-event control by considering the system as a parallel composition of the robot program (controller) and an environment (plant) model [10]. From a model-checking perspective, the system's state-space is now increased beyond the program state-space by the product of environment variables. A second characteristic is that there may be a necessary continuous nature to some aspects of the environment. Various hybrid continuous-discrete systems [11] have been introduced to handle this. Finally, significant uncertainty pertains to the result of robot sensing and motion; this cannot be ignored or the results are not realistic.

Uncertainty plays a major role in real-life robotic performance and needs to be included in any useful approach to robot verification. Napp and Klavins [12] introduce a guarded command language CCL for programming and reasoning about robot programs and environments. They address uncertainty by adding a concept of rates and exponential probability distributions to CCL, which allows them to reason about the robustness of programs. Johnson and Kress-Gazit [13] develop a model-checking algorithm that handles uncertainty based on Discrete Time Markov Chains; however, they comment on the intractability of their approach for large state spaces.

A state-based approach will experience significant combinatorial problems due to these characteristics. So rather than a state-based hybrid state/continuous approach, we have opted to avoid discussing state at all costs.

In [14] a process algebra approach for representing robot programs and environment models is introduced. Karaman et al. [8] also use a process algebra as a specification language for multiple UAV missions and develop a polynomial time algorithm that produces a plan to satisfy the specification. That work, and our previous work in process algebra for performance analysis of robot programs [15], leveraged the trace, or history of events, of a process. In this paper, however, the focus will be on how a process transforms its inputs to produce outputs without reference to states.

The proposed approach targets a specific kind of robot programming: behavior-based robot programming [16]. A behavior-based robot interacting with its environment will respond to a specific set of environmental percepts as programmed by its behaviors. Once a percept is responded to, the robot may return to this behavioral state or move to another that handles a different set of percepts.

III. Mission Specification

Dull, dirty, and dangerous missions are considered to be the natural niche for robots, and they have been a major driving force behind the advancement of robot technology. Over the past decades, we have seen an increasing number of robots being deployed to accomplish dangerous missions (e.g., disarming IEDs). Missions in the domains of urban search and rescue (USAR) and counter weapons of mass destruction (C-WMD) are not only dangerous, but their failures usually have dire consequences. It is highly desirable then to have the ability to verify the performance of a robot before it is deployed to carry out a mission. However, verification of robotic missions poses a unique and great challenge that is different from traditional software verification – the robot has to work in the real world, and the real world is inherently unpredictable. For example, robots were deployed during the World Trade Center rescue response, where the environment had become highly unstructured and filled with rubble piles [17]. In this article, we present our research [14] [18] on a mission design and verification framework for performance guarantees for critical missions where failure is not an option – the robot has to get it right the first time.

A. Mission Design Software Environment

The robot mission verification framework is built upon *MissionLab*, a behavior-based robot programming environment [6] [19]. *MissionLab* provides a graphical user interface *CfgEdit* where robot programs can be constructed as a finite state automaton (FSA) that sequences behaviors from a library of primitive behaviors. One of the many unique features of *MissionLab* is that it generates hardware-independent executables from user-constructed FSAs, which allows the desired robot platform to be chosen at run time. For critical missions where performance guarantees are desirable, a verification framework is developed in this paper for *MissionLab* where the mission can be verified before the software executable generation step.

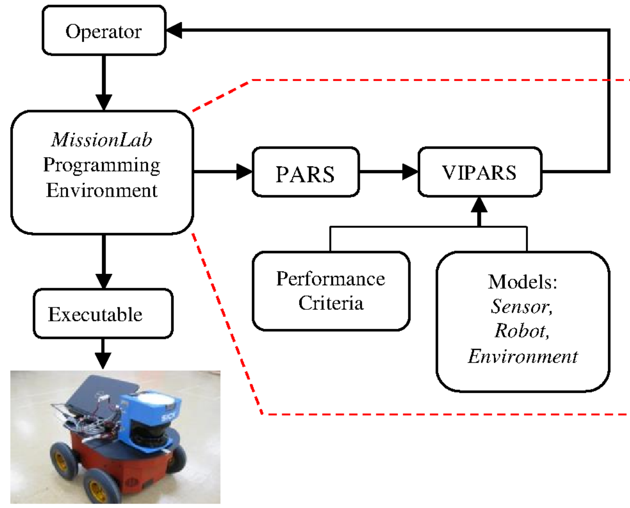

Figure 1. *MissionLab* robot mission specification toolset with VIPARS verification module.

The proposed verification framework is shown in Figure 1 as an extension to the *MissionLab* programming environment. To initiate the verification of a mission, the robot mission specification is compiled into PARS. The core of the framework is the process algebra based verification module, VIPARS. Two additional inputs are necessary for verification:

1) Robot, Sensor & Environment Models:

The robot operator specifies to *MissionLab* the models of robot, sensors and environment with which to conduct verification. It is *not* proposed that *MissionLab* designers build their own environment models against which to test the mission. Instead, just as *MissionLab* provides a set of robot drivers for simulation (Pioneer, Amigobot, ATRV-Jr etc) a set of standard robot, sensor and environment models will be constructed a-priori and provided as a library from which robot, sensor and environment features can be selected and composed automatically into an environment model. The development of a standard canonical set of environment models is not something we have pursued beyond those we have developed and used in validation. *MissionLab* also provides within its suite of tools data logging mechanisms for recording the performance of missions in terms of distance and time, and also has mechanisms for recording operator interaction [19].

2) Performance Criterion:

Performance criteria are mission constraints (e.g., safety and time constraints) that the robot system has to meet in order to assert "mission accomplished." The criterion consists of two parts: a probabilistic condition on a state variable of the robot, sensor and/or environment model and a time constraint on that condition. An example of a state variable is the position $p(t)$ of a robot at time t. An example of a performance criterion is that the robot have an 80% chance of arriving at a destination, $L1$ before a time limit, T.

$$P(\ p(t)=L1\ ) \geq 0.8 \text{ for some } t < T \quad (1)$$

Another example is that two robots at locations $p1(t)$ and $p2(t)$ are never closer to each other than a safety radius r:

$$P(\ |p1(t)-p2(t)| \ \leq\ r\ ) \geq 0.8 \text{ for all } t < T \quad (2)$$

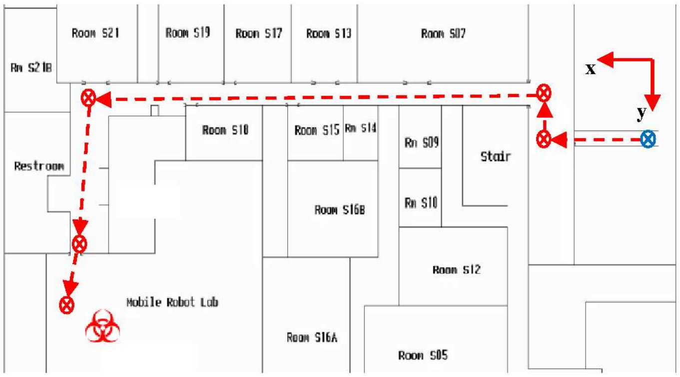

Figure 2. Building layout with mission waypoints labeled.

The output of VIPARS provided to the operator is a performance guarantee for the mission indicating whether the performance criteria were met. The verification module supports a feedback design loop, where the operator iteratively refines the robot program based on the performance information provided by VIPARS.

B. Mission Design

To illustrate the process of designing a mission with *MissionLab* and verifying it with VIPARS, two biohazard search scenarios are presented in which the robot needs to access a room inside the basement of a building where potential biological weapons might be located. This is representative of the types of C-WMD mission we are focusing on, i.e., where an approximate location of the weapon has been discovered and the building has been evacuated, thus no longer having any humans present in the setting. The robot's task is to confirm the location of the WMD and either remediate it itself via containment or manipulation (which we leave for future work) or provide the location to a well-protected human operator to subsequently enter and address the event. Any a-priori knowledge of the structure of the building if available can also be incorporated to guide the search.

In our example, the layout of the basement is shown in Figure 2, and the room the robot needs to access is shown with a biohazard symbol. Given a known layout of the environment, the simplest solution to accomplish the mission is to designate waypoints that the robot can follow to access the room containing the potential threat. The waypoints and the path of travel are shown in Figure 2. This is the first example mission.

It is more often the case, however, that there is not such strong knowledge of the operating environment. In these cases, autonomous exploration is necessary to find the biohazard. That scenario will be the second example mission.

1) Multiple Waypoint Mission

The design of the FSA for the multi-waypoint mission of Figure 2 is shown in Figure 3 and was created with *CfgEdit* in *MissionLab*. The FSA consists of a series of *GoToGuarded* and *Spin* behaviors with *AtGoal* and *HasTurned* triggers. The *GoToGuarded* behavior drives the robot to a specified goal location (i.e., waypoint) with a guarded radius of velocity dropoff around the goal location. The *AtGoal* trigger causes a transition to the next state when the robot reaches the goal location. The *Spin* behavior circulates the robot around an obstacle with a given velocity. The *HasTurned* behavior causes a state transition when the robot has turned a desired angle.

The performance criterion is a specification of the desired result for mission. The performance criterion for the waypoint mission is that the robot reach its final waypoint within the time limit, as in eq. (1).

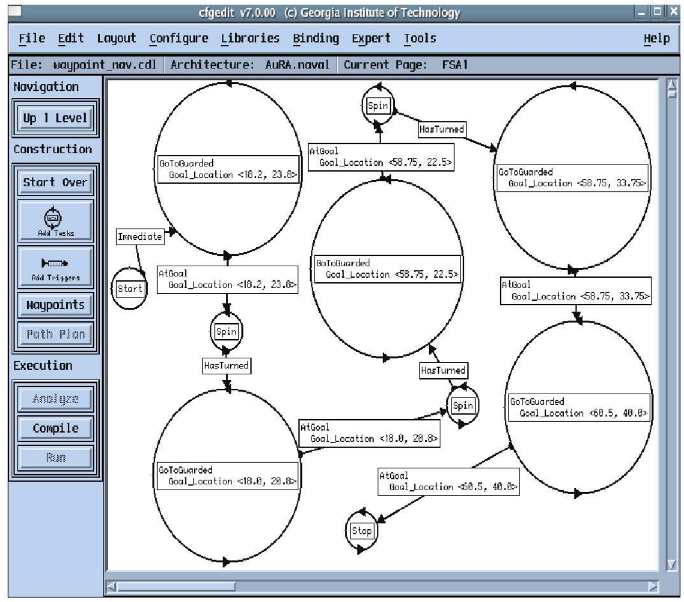

Figure 3. Mission design with *MissionLab*'s *CfgEdit.*

2) Autonomous Exploration Mission

The second example mission is a *Biohazard Search* mission: a robot is tasked to search an area for biohazard, Figure 4.

Figure 4. Indoor Biohazard Search Scenario

The control program for the mission, shown in Figure 5, is constructed in *MissionLab* as a behavioral assemblage in the form of an FSA. The FSA consists of three behaviors (*Wander, MoveToward,* and *Stop*) and three triggers (*Detect, NotDetected, and Near*). With this behavioral assemblage, the robot starts with random exploration of the environment. However, when *Detect* is triggered, the robot switches from random exploration to moving toward the detected biohazard. This mission is completed once the robot is within a certain distance of the biohazard.

The performance criterion in this case is that the robot find (i.e., approach) the biohazard within the time limit:

$$P(\ |p(t)-B| \leq \varepsilon\ ) \geq 0.8 \text{ for some } t < T \qquad (3)$$

where B is the location of the biohazard, $p(t)$ is the robot position, and ε is an approach distance constant.

C. *Verification of Performance Criterion*

Designs rarely work coming off the drawing board the first time. Final working products usually emerge only through numerous "going back to the drawing board" moments. The design of robot missions is no exception. However, for time-critical C-WMD and USAR missions where there might only be one opportunity to attempt the mission, it's necessary to have some guarantee that the designed robotic system will succeed before its deployment. To obtain a performance guarantee for the robot FSA in Figures 3 and 5, the operator needs to compile the robot program into PARS and provide VIPARS with the performance criteria and models of sensor, robot, and the environment (Figure 1).The details of the verification and validation of these two missions will be presented in Section VIII, after the theoretical basis of the approach has been introduced.

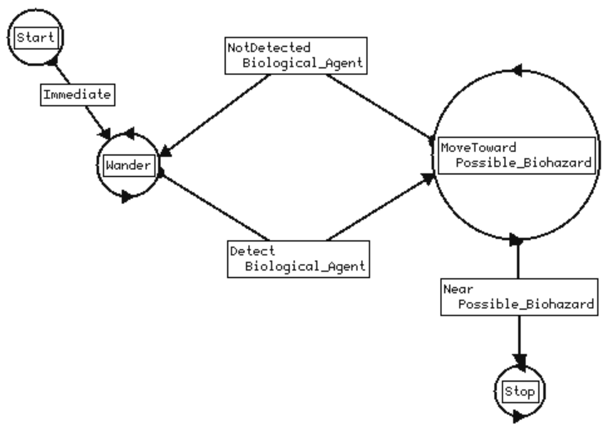

Figure 5. *CfgEdit* FSA for the Biohazard Search Mission

The robot used for both missions is a four-wheeled skid-steered mobile robot, the Pioneer 3-AT, shown in Figure 6. The robot is equipped with wheel encoders for localization, a gyro for heading correction, and a SICK laser for obstacle avoidance. For the exploration mission, the Pioneer 3-AT robot is equipped with a camera for biohazard. The principal source of uncertainty for these missions will be the sensor and actuator uncertainty and not uncertainty relating to the number and location of obstacles or terrain features.

Figure 6. Pioneer 3-AT Robot used in both Missions

VIPARS outputs 1) a Boolean answer to whether the mission will be successful as specified, and 2) a set of the variables in the performance criteria and their values at the time verification ended. When the performance criterion is probabilistic, the values returned are distributions.

If the predicted performance of the mission does not meet the necessary performance criteria, the operator can refine the robot program based on the feedback provided by VIPARS. This iterative process can continue until the operator is satisfied with the performance guarantee and sufficiently confident to deploy the robot.

IV. Representing Missions in PARS

PARS is a process algebra for representing robot programs and environments for automated verification. PARS provides a formal representation for capturing *tail-recursion* and it allows verification to be cast as the solution of a system of equations rather than as state enumeration and state checking and its associated combinatorics.

A. *Process Composition*

A process is written in PARS using a bolded capital letter, e.g., **P**, and using a common set of process composition operations (e.g., [20, 7]):

Definition 1 A process is defined as a composition of other processes as follows:

<processdef> ::= *<process>* *'='* *<processexpr>*
<processexpr> ::= *<processeq>* *'|'* *<processeq>*
<processeq> ::= *<processexpr>* *';'* *<processexpr>* |
'(' <processexpr> ')' |
<processname>

Where '|' denotes parallel composition and ';' denotes sequential composition, and where *<process>* and *<processname>* are a bolded capital letter or word.

Example 1. The process **P** is defined as the process **Q** followed by the process **R** and **S** in parallel, and when both terminate, followed by the process **T**:

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

The process description is modified to include the variables used by a process and a way to specify initial values for these variables and to return values as results. In general, process variables are written in lower case.

Definition 2. A process that takes initial variable values $u1, u2, \ldots, un$ and maps these to new values $v1, v2, \ldots, vm$ is written as:

$$\mathbf{P}\, \langle u1, u2, \ldots, un\rangle \langle v1, v2, \ldots, vm\rangle$$

A function $f_P(u1,u2,\ldots,un)=(v1,v2,\ldots,vm)$ is associated with **P** that maps the values $u1, u2, \ldots, un$ to $v1, v2, \ldots, vm$ in the same way. This is called the *flow function* associated with the process.

Composition operators can be used to funnel value calculations through a chain of processes.

Extending the syntax in Definition 1, a *<processname>* is now written as in Definition 2, and *<process>* is written as a bolded capital letter or word followed by a list of variable and result names between angle brackets.

Definition 3. The flow function of a composition is constructed from the flow functions of each member of the composition as the composition of flow functions on any common variable values.

Example 2. The flow function f_T of the process **T** defined as

$$\mathbf{T}\langle a\rangle\langle c\rangle = \mathbf{P}\langle a\rangle\langle b\rangle \,;\, \mathbf{Q}\langle b\rangle\langle c\rangle$$

is $f_T(a) = f_Q \circ f_P(a)$ since b is common to both processes. The flow function of the process **X** defined as

$$\mathbf{X}\langle a,c\rangle\langle b,d\rangle = \mathbf{Y}\langle a\rangle\langle b\rangle \mid \mathbf{Z}\langle c\rangle\langle d\rangle$$

is $f_X(a,c) = (f_Y(a), f_Z(c))$ since none of the variables are common to both process.

In practice there may be a mix of both of these cases.

B. Conditional Composition

PARS does not have a choice composition operator, used in many process algebras to implement conditional behavior. Instead, a sequence is conditional, as in LOTOS (Language of Temporal Ordering Specifications) [7]. A sequential chain is terminated immediately by a process ending in an abort condition.

Definition 4. Any process **P** can terminate in one of two conditions, a *stop* condition or an *abort* condition. The process **T** = **P** ; **Q** is defined to be the process **P** if **P** ends in an abort condition and the process **P** followed by the process **Q** if **P** ends in a stop condition.

Table 1. Examples of Finite Basic Processes

Process	Stop	Abort
Delay$\langle t \rangle$	After time t	Never
Ran$\langle \Phi \rangle \langle v \rangle$	returns a random sample v from a distribution Φ	Never
In$\langle p \rangle \langle x \rangle$, **Out**$\langle p,x \rangle$	perform input and output, respectively, on port p	Never
Eq$\langle a,b \rangle$, **Neq**$\langle a,b \rangle$, **Gtr**$\langle a,b \rangle$, etc.	$a=b$, $a \neq b$, $a>b$, etc.	Otherwise

Definition 5. A basic process **P** $\in$ *Basic* is a process whose behavior and flow function is defined a-priori, not by a composition of other processes.

Table 1 shows some basic processes that are used in this article. These are basic processes that *always* terminate, and are grouped in the set *Finite* $\subset$ *Basic*. Some that do not terminate will be introduced later. The last row shows several processes that calculate conditions, *basic condition processes*.

Definition 6. Each basic condition process **P** $\in$ *Cond,* where *Cond* $\subset$ *Finite* $\subset$ *Basic,* terminates in stop if its condition, denoted *cond*(**P**), is true, and in abort if the condition is false.

Example 3. A conditional statement that carries out **P** if $a=b$ and **Q** otherwise is written as follows:

$$\mathbf{T} = (\ \mathbf{Eq}\langle a,b \rangle \ ; \mathbf{P} \mid \mathbf{Neq}\langle a,b \rangle \ ; \mathbf{Q}\ )$$

The sequential chain **Eq**$\langle a,b \rangle$; **P** only continues to **P** if **Eq** *stops*, that is, if $a=b$. Similarly **Neq**$\langle a,b \rangle$; **Q** only continues if $a \neq b$.

Definition 7. The mapping $\Omega($ **P**$\langle u1,u2,\ldots,un \rangle$) maps a process **P** to a well formed *logical condition expression* over *conditions* with names of the process variables, *u1, u2,…,un* and condition operations ($=,\neq,>,\geq,<,\leq$), and *logical operations* between condition expressions ($\wedge,\vee,\neg$) that specifies the condition under which the process stops. For convenience, the *abort condition,* $\mho$(**P**) is defined as $\mho(\mathbf{P}) = \neg\Omega(\mathbf{P})$.

The mapping Ω is defined a-priori for basic condition processes **P**$\in$*Cond* by *cond*(**P**). The mapping must be calculated for compositions of processes.

Definition 8: Ω(**P**) is defined by:

If **P** $\in$ *Basic,*

$\Omega(\mathbf{P}) = cond(\mathbf{P})$ if $\mathbf{P} \in Cond,$

$\Omega(\mathbf{P}) = (\mathbf{P} \in Finite)$, else.

If **P** $\notin$ *Basic,*

$\Omega(\mathbf{P} \mid \mathbf{Q}) = (\Omega(\mathbf{P}) \wedge \Omega(\mathbf{Q}))$

$\Omega(\mathbf{P} \ ; \mathbf{Q}) = (\Omega(\mathbf{P}) \vee \Omega(\mathbf{Q}))$

Example 4. The Ω mapping for the basic condition process **Eq** is defined as follows: $\Omega($ **Eq**$\langle a,b \rangle$) = *cond*(**Eq**$\langle a,b \rangle$) = *"a=b"*.

A flow function can now be defined for a chain of processes as in Example 3.

Definition 9. The flow function f_T of the process **T** defined as

$$\mathbf{T} = \mathbf{P1};\ \mathbf{P2}$$

is $f_T = f_{P2} \circ f_{P1}$ if Ω(**P1**) evaluates to true and $f_T = f_{P1}$ otherwise.

Example 5. The flow function for **T** = (**Eq**$\langle a,b \rangle$; **P** | **Neq**$\langle a,b \rangle$; **Q**) is $f_T = f_P$ *if* $(a=b)$, and $f_T = f_Q$ *if* $(a \neq b)$.

C. Tail Recursive Processes

Definition 10. A tail-recursive (TR) process **T** is a process defined as a sequential composition of a non-recursive process expression (the *body* of the TR process) followed by a recursive reference to **T.**

Example 6. The process **T** = **P** ; **T** is a TR process if **P** is not recursive; **P** is the body of the TR process.

Definition 11. The process **T**$\langle a \rangle \langle b \rangle$ = **P**$\langle a \rangle \langle b \rangle$; **T**$\langle b \rangle$ is a TR process if **P** is not recursive.

- The flow function associated with **T** will be of the form $f_T = f_P{}^n\ (a)$ for $n \geq 0$. Furthermore,
- The value of n is the smallest n such that $\mho($ **P**$\langle\ f_P{}^{n-1}(a)\ \rangle$) evaluates to True.

Note 1. Recursion and iteration are equally expressive, and there is a method to transform general recursion to tail-recursion [21]. Hence tail recursion does not limit expressive power.

Note 2. Any language that implements sequence, condition and repetitive constructs is sufficient to represent any program [22]; thus, we can be confident that PARS can represent any program.

Note 3. Any computation of the TR process **T** can be examined as $f_T = f_P{}^n\ (a)$ for some $n>0$.

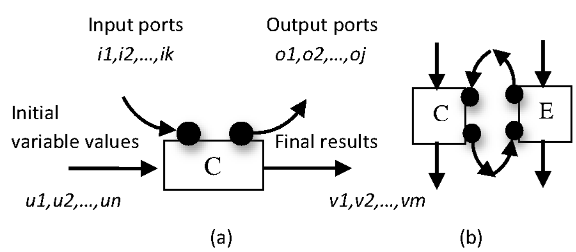

Figure 7. PARS Process Model (a) and Process Network Model (b)

D. Communicating Processes

The process algebra is now extended with a mechanism for parallel processes to exchange messages.

Definition 12. A process **C** with initial variable values *u1,u2,…,un* input port connections *i1,i2,…,ik* output port connections $o1,o2,\ldots,oj$ and final result values $v1,v2,\ldots,vm$ (see Figure 7(a)) is written as:

$$\mathtt{C}\langle u1,u2,\ldots,un\rangle\ (i1,i2,\ldots,ik)\ (o1,o2,\ldots,oj)\ \langle v1,v2,\ldots,vm\rangle$$

The input and output ports can be used by **C** for communicating with other, parallel processes *while* it is calculating its final values. A collection of connected, parallel processes will be referred to as a *network*. For example, in the network in Figure 7(b) the results calculated by **C** can be influenced by the process **E** to which it is connected. If **C** is a behavior-based robot control program, and **E** a model of its environment, then the results of that program may thus depend on the environment **E** in which the program is executed. Table 1, 3rd row, lists the basic processes for port communication.

Definition 13. $\mathbf{In}\langle p\rangle\langle x\rangle$ is a process that takes the name of a port p as an initial variable value, carries out a read operation on the port, and produces the value read, x, as a result; and $\mathbf{Out}\langle p, x\rangle$ is a process that takes the name of an output port p and a value to send on that port, x, as initial variable values and writes the value to the port.

Example 7. A process **C** that inputs a value on input port *pos* and then always outputs a value on port *vel* is defined as a sequential composition as follows:

$$\mathtt{C}\ (pos)(vel) = \mathbf{In}\langle pos\rangle\langle x\rangle\ ;\ \mathbf{Out}\langle vel,k*(g\text{-}x)\rangle$$

The value $k*(g\text{-}x)$ is the difference between a constant g and the value x read from the *pos* port, times a gain constant k. Initial variable values and results can be specified using standard arithmetic expressions and functions.

Example 8. The parallel composition:

$$\mathtt{S} = \mathtt{C}(c1)(c2)\ \ |\ \mathtt{E}(c2)(c1)$$

specifies two parallel processes **C** and **E** as shown in Figure 7(b), with the input and output ports connected correspondingly. The labels *c1* and *c2* are called *port connection labels* and their only purpose is to specify the connection map between the ports of the parallel processes.

Note 4. The addition of port communication complicates the relatively simple definition of flow functions! The flow function associated with a process no longer just depends on the variables of that process, but could depend on variables and computations of other parallel processes. This issue will be addressed by the addition of some structural constraints to the class of network to be analyzed in Theorem 1 of Section V.

Note 5. Port connections labels are a general way to describe port-to-port connections, and this is what the PARS/VIPARS implementation uses. However, they result in longer, more verbose process expressions. For many examples in this article, this is simplified by giving connected ports on parallel processes the same names (however cf. CSP or Promela channels [2]).

One more extension is made to PARS for the purpose of easily representing behavior-based programs. While a parallel-max composition '|' terminates when *both* processes terminate, a parallel-min composition called *disabling* composition is also introduced.

Definition 14. A disabling composition, written '#', is a parallel composition operation that terminates when *any* one of its arguments terminates (cf. LOTOS disabling [7]). The syntactic binding order is ';', '#' and '|'.

Definition 15. For the composition $\mathbf{T} = \mathbf{P}\ \#\ \mathbf{Q}$:

- $\Omega(\mathbf{P\#Q}) = \Omega(\mathbf{P}) \vee \Omega(\mathbf{Q})$
- $f_T = f_P$ *if* $\Omega(\mathbf{P})$ holds, f_Q *if* $\Omega(\mathbf{Q})$ holds, nondeterministic if both hold.

Definition 16. The basic process $\mathbf{Delay}\langle t\rangle$ terminates after a time t has elapsed. Its effect is similar to a condition process in that it indicates when a process ends rather than computes a value. *elapsed(t)* is added to the list of conditions in Definition 7 indicating the condition that time t has elapsed.

Example 9. The process $\mathbf{T1} = (\mathbf{Delay}\langle t1\rangle\ \#\ \mathbf{Delay}\langle t2\rangle)$ has $\Omega(\mathbf{T1}) = elapsed(min(t1,t2))$.

The process $\mathbf{T2} = (\mathbf{Delay}\langle t1\rangle\ |\ \mathbf{Delay}\langle t2\rangle)$ has $\Omega(\mathbf{T2}) = elapsed(max(t1,t2))$.

The process $\mathbf{T3} = (\mathbf{Delay}\langle t1\rangle\ ;\ \mathbf{Delay}\langle t2\rangle)$ has $\Omega(\mathbf{T3}) = elapsed(t1+t2)$.

E. PARS Controllers

In *MissionLab* a designer specifies the robot mission as a Finite State Automaton (FSA) (examples in Figure 3 and 5). Each state in the *CfgEdit* FSA involves the execution of a behavior which may result in many sensing and motor actions and interactions with the environment. Hence verification must occur at a greater level of the detail than that provided by simply using a model-checking approach with the states of the FSA. Prior work has investigated simply using a more detailed FSA for the problem [1] [2] [3], but this incurs the state-explosion issues discussed in Section II, and hence we do not take that approach.

1) TR Behavior Library

The states in the FSA (i.e., the circles in the *CfgEdit* diagram in Figure 3) correspond to behaviors from a library of robot behaviors in MissionLab. MissionLab behaviors are specified in the *Configuration Network Language (*CNL) [23]. To support the translation of CNL to PARS, a corresponding library of PARS behaviors was built. Only behaviors used in the kind of missions described in this article have been implemented to date. Each such behavior has been constructed as a TR process. The initial variable values for these processes can be used to parameterize behaviors, for example to provide a goal location to a *GotoGuarded* behavior. These processes use port communications to transfer information to other behaviors, or to the robot model, and acquire information from sensor models.

Example 10. An internal process in the PARS implementation of the *GoToGuarded* behavior, the **MoveTo** TR process, is defined as:

$$\mathbf{MoveTo}\langle g\rangle\ = \mathbf{In}\langle p\rangle\langle rp\rangle\ ;\ \mathbf{Gtr}\langle |rp\text{-}g|,\varepsilon\rangle\ ;\ \mathbf{Out}\langle v,\ d(g\text{-}rp)\rangle\ ;\ \mathbf{MoveTo}\langle g\rangle$$

The process inputs a value on the position input port p, checks to see if the position reported is close to the goal g, and if not (i.e., if the condition process **Gtr** stops) then it outputs an appropriate velocity $d(g-rp)$ to reach the goal on the port v. The function $d()$ is an arithmetic function that generates an appropriate velocity based on the distance from the goal, e.g., $k*(g-rp)$. As per Note 4, nothing can be stated about the flow function for this process yet. However, the Ʊ mapping for this process is:

$$\mho\ (\textbf{MoveTo}) = |rp\text{-}g| \leq \varepsilon$$

2) Triggers

The transitions between the states of the FSA are mediated by triggers (the rectangles in the *CfgEdit* diagram in Figure 3). These continually monitor some sensor condition and initiate the transition to a new behavior when the condition is satisfied. It is quite easy to see how a trigger can be phrased as a TR process, since it is principally a repeated condition process. A library of trigger processes was also constructed, sufficient to support the kind of missions described here.

Example 11. As an example, the **AtGoal** TR process is shown:

$$\textbf{AtGoal}\langle g\rangle = \textbf{In}\langle p\rangle\langle rp\rangle\ ;\ \textbf{Gtr}\langle|rp\text{-}g|,\varepsilon\rangle\ ;\ \textbf{AtGoal}\langle g\rangle$$

The position of the robot is read on port p, and if the position is close to g, then the tail recursion stops as in Example 10.

3) Mission and System Processes

The *CfgEdit* FSA is translated (through *CNL* [23]) to a PARS process called **Mission** by:

1. Identifying the library process associated with the trigger or behavior state.
2. Identifying the values of the parameters associated with the trigger or state and providing them as initial variable values for the process.
3. Composing the trigger and behavior processes based on their relationship within the FSA.

Example 12. This conversion is illustrated using the last two *GoToGuarded* states and intervening trigger in Figure 3. These can be composed as

$$\begin{aligned}\textbf{Mission} = \ &\textbf{GoToGuarded}\langle loc3\rangle\ |\\ &\textbf{AtGoal}\langle loc3\rangle\ ;\ \textbf{GoToGuarded}\langle loc4\rangle\end{aligned}$$

The values *loc3* and *loc4* are the corresponding locations from the *CfgEdit* FSA, and **GoToGuarded** and **AtGoal** are the TR processes from the behavior library.

F. PARS Environments

An environment model in PARS is a causal model of the environment in which the robot program is carried out.

Example 13. A robot model, **Robot,** which includes both position and heading uncertainty is shown below as a TR process:

$$\begin{aligned}\textbf{Robot}\langle r,a,s\rangle &= (\textbf{Delay}\langle\tau\rangle\ \#\ \textbf{Odo}\langle r\rangle\#\ \textbf{At}\langle \mathrm{r1}, r\rangle)\ ;\\ &\quad(\textbf{In}\langle h\rangle\langle a\rangle\ \#\ \textbf{In}\langle v\rangle\langle s\rangle)\ ;\\ &\quad(\textbf{Ran}\langle\Theta_h\rangle\langle z\rangle\ |\ \textbf{Ran}\langle\Theta_v\rangle\langle w\rangle)\ ;\\ &\quad\textbf{Robot}\langle r+u(a+z)*(s+w)*\tau,\ a,\ s\rangle\\ \textbf{Odo}\langle r\rangle &= \textbf{Ran}\langle\Phi\rangle\langle e\rangle\ ;\ \textbf{Out}\langle p, r+e\rangle\ ;\ \textbf{Odo}\langle r\rangle\end{aligned}$$

The environment model accepts a heading input on port h or a speed in the direction of the heading on port v. The process **At**$\langle r1,r\rangle$, an infinite basic process, represents robot $r1$ at location r. The process **Odo** (short for Odometry sensor) makes position information (with noise $e \sim \Phi$) available in a loop on port p until terminated by **Delay**. The new position of the robot is calculated as the old position r incremented by a noisy speed command $(s+w)$ in the direction of the noisy heading $(a+z)$ (where $u()$ returns the unit vector). The actuator and odometer noise (the variables z, w, and e) are characterized by the distributions for speed, heading and sensor noise, e.g., $\Theta_h = N(\mu_h,\sigma_h)$, $\Theta_v = N(\mu_v,\sigma_v)$, and $\Phi = N(\mu_m,\sigma_m)$.

G. PARS Goals

One of the strengths of model-checking is the established relationship between the state-based transition model for the program and a temporal logic specification of the property to be verified [2]. However, a process algebra can be just as intuitive a language to specify event orderings [7] as a temporal logic language. Of course, in a logic it is possible to concisely proscribe, i.e., to state what should *not* hold, whereas a process algebra is prescriptive. It will be shown that this is not a limitation for the proposed approach.

First, consider how a property is related to a program. *Implementation* is distinguished from *specification:* A property to be verified, a *specification*, is a more abstract process network, while an *implementation* is a process network with all the details filled in [7] [24].

Definition 17. **P**≈**Q** denotes process **P** and process **Q** are observationally equivalent by a specification implementation bisimulation. The details of **P**≈**Q** are postponed to Section V.

A performance criterion network is written using process composition operations and basic processes along with constraints on the values of the variables to the processes.

Example 14. A simple liveness criterion is that robot $r1$ reach a specific location (cf. Eq. (*1*)) ; this is written as follows:

$$\begin{aligned}\textbf{Q} = \textbf{Delay}\langle t1\rangle\ ;\ (\textbf{Delay}\langle t2\rangle\ \#\ \textbf{At}\langle r1,p\rangle)\\ \textit{for some } t1\leq T,\ |p\text{-}L| < \varepsilon\end{aligned}$$

The basic process **At**$\langle r,p\rangle$ represents robot r at position p. This process never terminates unless run in a disabling composition with a process that will terminate (such as **Delay**). This specification states that $r1$ arrives at location p by time T at the latest and the final position be within ε of location L. Notice there is no constraint on $t2$ (how long the robot is at the final location p).

Example 15. In this two robot example, robot $r1$ must arrive at its location before $r2$ arrives at its location. Process variables in the proposed framework may be *random variables*. An example constraint on a random variable can be its probability of meeting some condition. In this case, the criterion specifies that the location of robot $r1$ must have at least an *80%* probability of being within a distance R of location $L1$ by time $t1$.

$$\begin{aligned}\textbf{Q} = \textbf{Delay}\langle t1\rangle\ ;\ (\textbf{Delay}\langle t2\rangle\ \#\ \textbf{At}\langle r1,p\rangle)\ |\\ \textbf{Delay}\langle t3\rangle\ ;\ (\textbf{Delay}\langle t4\rangle\ \#\ \textbf{At}\langle r2,q\rangle)\\ \textit{for some } t1,t3\leq T,\ t1<t3,\\ P\,(|p\text{-}L1|<R)>0.8,\ P\,(|q\text{-}L2|<R)>0.8\end{aligned}$$

Example 16. Consider a safety criterion: that two robots never approach too closely (cf. Eq. (*2*)). Notice this involves proscription – that a specific process composition never happens – difficult with process algebra. It is handled here by moving the proscription to a constraint on the variable values. The probability of the robots being at least a distance ε *away from each other* must be at least 80% for all times:

$$\mathsf{Q} = \mathsf{Delay}\langle t1\rangle \;;\; (\mathsf{Delay}\langle t2\rangle \;\#\; \mathsf{At}\langle r1,p\rangle) \;|\; \mathsf{Delay}\langle t3\rangle \;;\; (\mathsf{Delay}\langle t4\rangle \;\#\; \mathsf{At}\langle r2,q\rangle)$$

$$\textit{for all } t1,t3>0,\ P\,(|p-q|>\varepsilon)>0.8$$

A *performance criterion* is defined as follows:

Definition 18. $G(\mathsf{Q},C)$ is a performance criterion, where

- Q is the process network associated with the criterion,
- C is a boolean function on the process variables in Q that is true if the constraints on the variables hold.

If u are the variables in Q, then $C(u)$ holds *iff* the constraints in C hold on the variable values of u. In the performance criterion in Example 16, $C(t1,t2,t3,t4,p,q)$ is true *iff* *for all* $t1,t3>0$, $P\,(|p-q|>\varepsilon)>0.8$.

As usual, a safety criterion, will be handled during verification as a negated liveness condition. A performance criterion is negated by negating the constraint on variable values: $\neg G(\mathsf{Q},C) = G(\mathsf{Q},\neg C)$.

V. Verification in PARS

Given a parallel communicating composition of a controller **Mission** and environment model **Robot**:

$$\mathsf{Sys} = \mathsf{Mission}\langle u1,\ldots\rangle(a1,\ldots)(b1,\ldots) \;|\; \mathsf{Robot}\langle e1,\ldots\rangle(b1,\ldots)(a1,\ldots) \tag{4}$$

our objective is to automatically verify that **Sys** will achieve some desired performance criterion $G(\mathsf{Q},C)$. Section IV has introduced a lot of the methodology needed to support this but with one significant gap: While it is possible to express the flow-function of a TR process (Definition 11, Note 3), eq. (4) is *not* a TR process, so there is no way to express its computation as a flow-function (Note 4).

This section will present an approach to determining the flow-function for a parallel, communicating composition of processes *with the assumption* that the component processes are themselves TR processes. This is a reasonable assumption; in the previous section it was shown that *MissionLab* behavior-based programs can be translated to TR processes, and that the environment model can also be constructed as a TR process. Furthermore, tail-recursion does not restrict what can be represented in either.

Determining this flow-function will require two steps:

1. Translate the network of TR processes into a single TR process (*SysGen* algorithm, presented in subsection A).
2. Combine the flow-functions for the individual TR processes with the port connection mappings for the network to produce the flow-function for the single TR process constructed in step 1 (*FloGen* algorithm, presented in subsection B).

With a procedure to determine the flow-function for a network such as eq. (4), it is only necessary to then describe how to use this to verify whether the network satisfies the performance criterion $G(\mathsf{Q},C)$ (presented in subsection C).

A. SysGen Interleaving Theorem and Algorithm

Consider a parallel composition of TR processes $\mathbf{P}_1, \mathbf{P}_2, \ldots, \mathbf{P}_m$ as follows:

$$\mathbf{Sys} = \mathbf{P}_1 \,|\, \mathbf{P}_2 \,|\, \ldots \,|\, \mathbf{P}_m$$

An *interleaving theorem* in process algebra relates sequential and parallel operations.

Theorem 1. A parallel, communicating composition of TR processes can be written as a single TR process:

$$\mathbf{Sys} = \mathbf{P}_1 \,|\, \mathbf{P}_2 \,|\, \ldots \,|\, \mathbf{P}_m = \mathrm{S}(\mathbf{P}_1, \mathbf{P}_2, \ldots, \mathbf{P}_m) \;;\; \mathbf{Sys}$$

iff a *system period* process, $\mathrm{S}(\mathbf{P}_1, \mathbf{P}_2, \ldots, \mathbf{P}_m)$ can be constructed.

Informally, **Sys** is a collection of TR processes, each of which is in the form $\mathbf{P}_i = \hat{\mathbf{P}}_i \,;\, \mathbf{P}_i$ where $\hat{\mathbf{P}}_i$ is the non-recursive *body* of the i^{th} TR process. The port communications between any process i and any other process j are what synchronize the execution of the bodies of these two processes, $\hat{\boldsymbol{P}}_i$ and $\hat{\boldsymbol{P}}_j$. For example, if the bodies were synchronized by communication to all start *at the same time*, then we could say that the system period process $\mathrm{S}(\mathbf{P}_1, \mathbf{P}_2, \ldots, \mathbf{P}_m)$ was just equal to the parallel composition of all the body processes $\hat{\boldsymbol{P}}_1|\hat{\boldsymbol{P}}_2|\ldots|\hat{\boldsymbol{P}}_m$. However, if there are more than just a single such synchronizing communication between processes, then some $\hat{\boldsymbol{P}}_i$ may have to repeat several times more or less than other $\hat{\boldsymbol{P}}_j$. The system period process is similar to the concept of the *hyper-period* (LCM of all the task periods in a scheduling problem) [25].

The proof of Theorem 1 is by construction of the system period process (and we call the resulting algorithm *SysGen).*

SysGen starts with the non-recursive *body* $\hat{\boldsymbol{P}}_i$ of the i^{th} TR process. The body is then restricted to just the port communication processes, hiding other processes as internal operations. The *abs* operation is introduced to formalize this hiding here (and in later sections).

Definition 19. **P** *abs S* is the process in which any processes in **P** not named in the set S are hidden by the internal action **i**. This can be defined recursively:

- $(\mathbf{P} \;;\, \mathbf{Q})\ abs\ S = (\mathbf{P} \;;\, \mathbf{i}) = \mathbf{P}$ if $\mathbf{Q} \notin S$ *or* $= (\mathbf{i} \;;\, \mathbf{Q}) = \mathbf{Q}$ if $\mathbf{P} \notin S$
- $(\mathbf{P} \,|\, \mathbf{Q})\ abs\ S = (\mathbf{P} \,|\, \mathbf{i}) = \mathbf{P}$ if $\mathbf{Q} \notin S$
- $(\mathbf{P} \,\#\, \mathbf{Q})\ abs\ S = (\mathbf{P} \,\#\, \mathbf{i}) = \mathbf{P}$ if $\mathbf{Q} \notin S$

For convenience, **P** *abs* **Q** is written to mean the process where any processes not named in **Q** are hidden. Compositions of internal actions can be grouped:

$$\mathbf{i} \,;\, \mathbf{i} = \mathbf{i}, \quad \mathbf{i} \,|\, \mathbf{i} = \mathbf{i} \quad \text{and} \quad \mathbf{i} \,\#\, \mathbf{i} = \mathbf{i}$$

Restriction to just port operations will allow the matching of input and output port operations between the parallel $\hat{\mathbf{P}}_i$.

$$\mathbf{IO}_i = \mathbf{P}_i\ abs\ \{\ \mathbf{In}, \mathbf{Out}\ \} \tag{5}$$

From the port connection labels in a parallel composition, a communication map *cm* can be built which specifies how ports on one process connect to ports on another in **Sys**. Call $\mathbf{IO}_i(j)$ the *jth* port operation in $\mathbf{IO}_i$ and let $p(\mathbf{IO}_i(j))$ be the portname in that operation. Then *SysGen* starts with $\mathbf{IO}_i(j_i = 0)$ for each process $\mathbf{P}_i$ and checks for the following:

$$cm(\,p(\mathbf{IO}_i(j_i)), \quad p(\mathbf{IO}_k(j_k))\,) \tag{6}$$

For sequential and disabling compositions in $\mathbf{IO}_i$, as soon as (6) produces a match with some other process $\mathbf{IO}_k$, j_i and j_k can be incremented to the next operation, since only one communication happens for the composition to terminate. For parallel composition however, all the communications in the composition need to be matched before j_i is incremented.

If at any point, two processes cannot be found for which (6) holds, but one or more processes have previously matched all their operations, then those processes can be *unwound*. For example, if $\mathbf{IO}_i$ has been completely matched already, then we can replace it with $\mathbf{IO}_i = \mathbf{IO}_i\ ; \mathbf{IO}_i$ which is just an unwinding (e.g., **T** = **P** ; **T** = **P** ; **P** ; **T**) of a tail recursion and in this way extend the body of the process and the opportunity to continue to match port connections. Informally: this unwinding in general means that this component process needs to loop twice (or more) to handle multiple communications from a single loop of another component process.

When all $\mathbf{IO}_i$ are completely matched (including any unwound processes), the system period process has been built. However, if (6) fails and no processes can be unwound, then no system period process exists. If it exists, the system period process will be given by:

$$\mathbf{S}(\mathbf{P_1}, \mathbf{P_2}, \ldots, \mathbf{P_m}) = \widehat{\mathbf{P}}_1^{\,k1} \,\big|\, \widehat{\mathbf{P}}_2^{\,k2} \,\big|\, \ldots \,\big|\, \widehat{\mathbf{P}}_m^{\,km} \tag{7}$$

for some *unwinding* constants $k1, \ldots, km$. For a discussion of the computation complexity of *SysGen*, see Appendix I.

B. *FloGen Algorithm*

SysGen recasts the analysis of **Sys** into the analysis of a single period process S(**Sys**)=S(**P**$_1$,…,**P**$_m$). S(**Sys**) transforms the values of the variables in **P**$_1$,…,**P**$_m$ at start of repetition k of the system period process to those at the start of repetition $k+1$. Within S(**Sys**) variables may have their values transformed in two ways:

- By operations within the processes **P**$_1$,…,**P**$_m$: This information is obtained directly from each process flow function.
- By message passing: Variable values may be sent via port communications to be included in other processes.

When a process receives a value from another process, the value arrives as a result variable of the **In** process. For example the TR process

$$\mathbf{T}\langle y\rangle = \mathbf{In}\langle p\rangle\langle x\rangle\ ; \mathbf{T}\langle x+y\rangle$$

repeatedly accepts a value on port p and then adds it to its variable y. It has a flow-function

$$f_T(y) = x+y,$$

which includes a value x that is *not* one of the process variables, and which can only be disambiguated by following the connection for port p and determining which process and which variable in that process produces x. Such variables will be referred to *message variables*.

Definition 20. Let R be set of variables of the processes **P**$_1$,…,**P**$_m$ and let $val(r_i)$ be the value set of the variable $r_i \in R$. The system period process flow function maps all variables in the k^{th} iteration of the system period process to their values in the $(k+1)^{\text{th}}$ iteration:

$$f_{sys}(r_1, \ldots, r_n):\quad val(r_1) \times \ldots \times val(r_n) \rightarrow val(r_1) \times \ldots \times val(r_n) \tag{8}$$

A variable's transformation is traced through the processes and port communications in S(**Sys**) to generate a single flow function f_{sys} defined in terms of flow functions for each variables in the system process $f_{sys,ri}$ as follows:

$$f_{sys}(r_1, \ldots, r_n) = (f_{sys,r1}(r_1, \ldots, r_n), f_{sys,r2}(r_1, \ldots, r_n), \ldots, f_{sys,rn}(r_1, \ldots, r_n)) \tag{9}$$

The *FloGen* algorithm (Figure 8) implements the following of connections in S(**Sys**) to replace all the message variables with process variables and hence generate the system period process flow function, eq. (9).

FloGen($FS = \{f_{P1}, \ldots f_{Pm}\}$, cm): // remove message variables from all FS

For each $f_{Pi} \in FS$
  For each v_j not in R for P$_i$ *//i.e. from an* ***In***
    $a \leftarrow$port variable of **In** that generated v_j
    Replace v_j with **FloFollow**(a, FS, cm)

FloFollow(a, FS, cm): // replace a single message variable

Find port b on P$_k$ connected to a on P$_i$ using cm
$expr \leftarrow$ variable expression in **Out** on port b in P$_k$
For each u_j in $expr$ not in R for P$_k$
  $a \leftarrow$ port in p$_k$ that generated u_j
  Replace u_j in $expr$ with **FloFollow**(a, FS, cm)
Replace v_j with $expr$

Figure 8. Flow Function Generation Algorithm, *FloGen* and recursive helper function *FloFollow*.

For a discussion of the computational complexity of *FloGen*, see Appendix II.

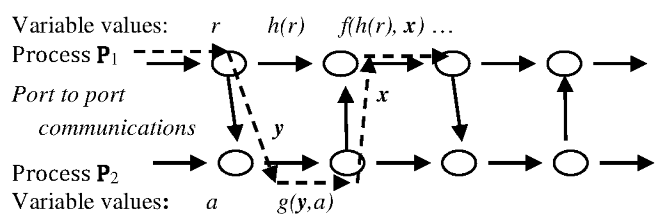

Figure 9. Example of variable value transformation (dotted lines) for variables r and a in a single system period process composed of two processes **P**$_1$ and **P**$_2$. The message variables $\boldsymbol{x}$ and $\boldsymbol{y}$ are resolved by *FloGen* as $\boldsymbol{y}=r$ and $\boldsymbol{x}=g(\boldsymbol{y},a)=g(r,a)$.

Figure 9 shows an illustrative example of this kind of variable value transformation between two processes. Recall that there are two ways a variable can have its value modified: by operations within the process, and/or by message passing with another process. In Figure 9, **P**$_1$ transmits its variable r to **P**$_2$, which then modifies its variable a by a transformation $g(r,a)$ and transmits this back to **P**$_1$. Process **P**$_1$ transforms its variable first to $h(r)$ and then, on receiving $g(r,a)$, to $f(h(r),g(r,a)$. The *FloFollow* algorithm follows the chain of transformations indicated by the dotted line in Figure 9 to resolve $f(h(r),g(r,a))$. The *FloGen* algorithm uses *FloFollow* to replace all the message variables in every flow function.

Example 17. Consider the system process defined as

$$\textsf{Sys}\langle g, r, a, s\rangle = \textsf{MoveTo}\,\langle g\rangle \mid \textsf{Robot}\langle r, a, s\rangle$$

where the component processes have been defined in Examples 10 and 13 respectively, and where the port communication is simplified to ports with the same name on different processes being connected (Note 5). **Sys** has a system period process that is simply the body processes of each of the component processes since communication on port p acts to synchronize the start of both body processes.

$$\begin{aligned}\mathrm{S}(\textsf{Sys}) = (\ & \textsf{In}\langle p\rangle\langle rp\rangle \,;\, \textsf{Gtr}\langle|rp\text{-}g|,\varepsilon\rangle \,;\, \textsf{Out}\langle v, d(g\text{-}rp\,)\rangle) \quad | \\ & ((\,\textsf{Delay}\langle\tau\rangle \,\#\, \textsf{Odo}\langle r\rangle\#\, \textsf{At}\langle \mathrm{r1}, r\rangle\,)\,; \\ & \quad (\,\textsf{In}\langle h\rangle\langle a\rangle \,\#\, \textsf{In}\langle v\rangle\langle s\rangle\,)\,; \\ & \quad (\textsf{Ran}\langle\Theta_v\rangle\langle w\rangle \mid \textsf{Ran}\langle\Theta_h\rangle\langle z\rangle\ ))\end{aligned}$$

The flow function for the system is arrived at as follows. The flow function $f_{sys,r2}$ ($r2$ is the *'r'* variable) with its original message variable s (from **Robot** in Example 13) is:

$$f_{sys,r2}(g,r,a,s)=r+u(a+z)*(s+w)*\tau$$

FloGen identifies s as a message variable (it's not one of the variables in the definition of **Robot**) and calls *FloFollow* which follows the port v from which s came to port v on **MoveTo**. It replaces s with the expression from **MoveTo** $d(g\text{-}rp);$ but rp is a message variable in **MoveTo** generated on port p. *FloFollow* follows the p port back to **Robot**, replacing rp with r.

The system flow function is therefore calculated as:

$$\begin{aligned}&f_{sys,r1}(g,r,a,s)=g\\ &f_{sys,r2}(g,r,a,s)=r+u(a+z)*(d(g\text{-}r)+w)*\tau,\ z\sim\Theta_h\,, w\sim\Theta_v\\ &f_{sys,r3}(g,r,a,s)=a\\ &f_{sys,r4}(g,r,a,s)=s\\ &f_{sys}(g,r,a,s)=(\,f_{sys,r1}(g,r,a,s), f_{sys,r2}(g,r,a,s), f_{sys,r3}(g,r,a,s),\\ &\qquad f_{sys,r4}(g,r,a,s)\ )\end{aligned}$$

The metric time behavior of a network is characterized completely by the **Delay**$\langle\tau\rangle$ processes in the network. Time behavior is restricted to discrete-time in this paper. To determine the time behavior of a network, the network is restricted to a network of just delay processes, and *elapsed* is calculated by combining the values of the variable for each process using the three results in Example 9.

The network obtained from **P** by hiding all process except the delay process is written **P** *abs* **Delay**.

Definition 21. The time behavior of **P** is defined to be: Ω (**P** *abs* **Delay**).

The time behavior of the system period process S(**Sys**) in Example 17 is given by Ω (**S** *abs* **Delay**) = Ω (**Delay**$\langle\tau\rangle$) = τ.

C. Satisfying Performance Guarantees

Given that a flow-function can be built for a network of interest, it now remains to be seen how this can be used in verifying whether or not the network satisfies a performance criterion. Recall that a performance criterion $G(\mathbf{Q},C)$ consists (Definition 18) of a process network **Q** and constraints C on the variables of **Q**.

Definition 22. Let **Sys** be a TR process with system period process **S**=S(**Sys**), system flow function f and initial variable values u_o and let $G(\mathbf{Q},C)$ be a performance criterion, then

Sys *satisfies* $G(\mathbf{Q},C)$ *iff*, for some $n\geq 0$, $\mathbf{S}^n \approx \mathbf{Q}$, and $C(\,f^n(u_o))$

Where $\mathbf{P}^1$=**P**, $\mathbf{P}^k=\mathbf{P}^{k\text{-}1}$; **P** is repeated sequential composition.

To use *satisfies*, it is necessary to define how it can be established that $\mathbf{P}\approx\mathbf{Q}$. This is accomplished in two steps. First **P** is restricted to contain just the processes in **Q**, by evaluating **P** *abs* **Q**. Secondly, the networks, now containing the same processes are compared structurally (**P** *abs* **Q**) $\approx_s$ **Q**. If this is successful, a mapping can be made between processes in (**P** *abs* **Q**) and processes in **Q** and the constraint function C can be applied to the variables in (**P** *abs* **Q**).

Definition 23. $\mathbf{P}\approx_s\mathbf{Q}$ holds iff

$$\mathbf{P}=\mathbf{Q} \text{ or } \mathbf{P}=\mathbf{P1}\ \alpha\ \mathbf{P2},\ \mathbf{Q}=\mathbf{Q1}\ \beta\ \mathbf{Q2} \text{ where,}$$
$$\mathbf{P1}\approx_s\mathbf{Q1},\ \mathbf{P2}\approx_s\mathbf{Q2},\ \alpha=\beta\in\{';', '|', '\#'\}$$

Structural equivalence is more restrictive ($\mathbf{P}\approx_s\mathbf{Q}\Rightarrow\mathbf{P}\approx\mathbf{Q}$ but not vice-versa) but is fast.

Example 17 (Cont). Consider verifying **Sys**$\langle L1, L0, H0, V0\rangle$ where these initial values are those for the system in Example 17, i.e., the goal location L1, the initial position L0, heading H0 and velocity V0. We verify against the performance constraint **Q** in Example 14, that the robot reaches its goal L1 by time T. Applying the two step process, using filtering and structural equivalence, we have that:

1. ($\mathbf{S}^n$ *abs* **Q**)=
   Delay$\langle(n\text{-}1)\ \tau\rangle$; (**Delay**$\langle\tau\rangle$ #**At**$\langle \mathrm{r1}, f_{r2}{}^n(L1,L0,H0)\rangle$)
2. ($\mathbf{S}^n$ *abs* **Q**) $\approx_s$ **Q** $\Rightarrow t1=(n\text{-}1)\tau,\ t2=\tau,\ L1=f_{r2}{}^n(L1,L0,H0)$

Establishing $\mathbf{P}\approx_s\mathbf{Q}$ established a mapping between the initial variable values in **P** and those in **Q**, shown as the RHS of the implication in step 2. The constraint function C is now applied to the mapped variables in **P.** The final step in verification is to determine whether the constraints C will be met given the system flow function. This final verification step is implemented using a Bayesian Network approach, and described in Section VI. An important extension to Definition 22 is addressed first. However, those readers eager to finish the story could skip the next section.

D. Non-TR Missions

A behavior-based program continually and concurrently responds to affordances in its environment. *MissionLab CfgEdit* missions represent this with triggers and behaviors, and these are mapped to parallel compositions of sequences of processes in Section IV.E. Such a sequence is not a TR process, and any **Mission** process thus generated will be *non-TR*. The definition of a system satisfying a performance criterion must be extended to cover a system with sequences of TR processes. This is accomplished in two steps.

Let $\mathbf{P}=\mathbf{P}_1\,;\,\mathbf{P}_2\,;\,\mathbf{P}_3 \ldots \mathbf{P}_m$ be a *single* non-TR process in the system **Sys** where each $\mathbf{P}_i$ is a TR process. Let $\mathbf{Sys}_i$ be **Sys,** but where **P** is replaced by $\mathbf{P}_i$. Let $\mathbf{S}_i$ be the system period process for $\mathbf{Sys}_i$ and let f_i be the $\mathbf{Sys}_i$ flow function.. In that case:

Definition 24. Sys *satisfies* $G(\mathbf{Q},C)$ iff for $n1,n2,..,nm\geq 0$

- $\mathbf{S}_1{}^{n1};\ \mathbf{S}_2{}^{n2}\ ;\ \ldots\ ;\ \mathbf{S}_m{}^{nm}\approx\mathbf{Q}$, and
- $C(f_1{}^{n1}(u_o), f_2{}^{n2}(u_1), f_m{}^{nm}(u_{m\text{-}1}))$,

Note 6. If $u_1 = f_1^{n1}(u_o)$ and so forth for u_1 through u_{m-1}, then each subproblem *depends* on the previous. This can be very useful for example to propagate position uncertainty through the system of subproblems. However, it is also useful to be able to isolate each subproblem by re-initializing between subproblems. This allows each subproblem, e.g., each waypoint in a multiple waypoint mission, to be treated *independently.*

This definition breaks the verification problem into the verifications of a sequence of TR subsystems, matching the sequence to the performance criterion network. This can be used to enforce the constraint that a robot visit locations in sequence, or just reach the last location.

However, a difficulty of multirobot systems is each agent could have one (or more) non-TR mission processes. Thus for k agents, this definition of satisfy would require k^m different sequences to be tested! To avoid these combinatorics, an approach is proposed below which only needs to test one sequence.

A TR process terminates when its body aborts (Definition 11). Let **Sys** be composed of k non-TR processes, each of which is m TR processes in sequence. $\mathbf{P}_{ij}$ is written for non-TR component $i \in \{1...k\}$ and sequential component $j \in \{1...m\}$. Let **Sys1** include all $\mathbf{P}_{i1}$ for $i \in \{1...k\}$, and any other TR processes, $\mathbf{R}_1,\ldots,\mathbf{R}_h$ that are in the system. (The TR processes might include for example, the TR robot process as in eq. (4).) To determine what should be in **Sys2** we proceed as in Definition 24 except that $\bigvee_i \mho(\mathbf{P}_{i1})$ is evaluated to determine which trigger process achieves its termination conditions first; If this condition contains random variables, then its satisfaction gives the *most likely* process to terminate first

Sys2 is constructed from **Sys1** with any $\mathbf{P}_{i1}$ for which $\mho(\mathbf{P}_{i1})$ evaluates to True replaced by $\mathbf{P}_{i2}$. A single sequence of systems is thus constructed, each representing the most likely next system given the previous systems.

Definition 25. Let **Sys** be composed of k non-TR processes, **Sys** *satisfies* $G(\mathbf{Q},C)$ *iff* for $n1,n2,..,nm \geq 0$

$\mathbf{S}_1^{n1}; \mathbf{S}_2^{n2}; .. ; \mathbf{S}_m^{nm} \approx \mathbf{Q}$, where

- $\mathbf{Sys_1} = (\mathbf{P}_{11} \mid \mathbf{P}_{21} \mid ... \mid \mathbf{P}_{k1}) \mid (\mathbf{R}_1 \mid ... \mid \mathbf{R}_h)$ and
- each subsequent system **Sys+** is the previous system **Sys-** with each $\mathbf{P}_{ij}$ for which $\mho(\mathbf{P}_{ij})$ evaluates to True replaced by $\mathbf{P}_{il}$, $l=j+1$, and
- $C(f_1^{n1}(u_o), f_2^{n2}(u_1), f_m^{nm}(u_{m-1}))$.

VI. Verification by Filtering

Definition 22, and its extensions, Definitions 24 and 25, specify how to automatically verify that a parallel communicating network will satisfy a performance criterion. However they require the solution of a Boolean expression $C(f^n(u_o))$. This section will introduce an approach to solving this expression based on using a *Dynamic Bayesian Network*: A Dynamic Bayesian Network (DBN) is a Bayesian Network which relates variables to each other within a time step and also between time-steps [26]. Solving $C(f^n(u_o))$ will be mapped onto the filtering problem for a DBN in subsection A. Several short examples of the results of applying this verification approach are presented in subsection B.

A. Filtering Algorithm

Recall the system flow function $f_{sys,ri}$ relates the value of $r_i \in$ R at time step k to its value in time step $k+1$. Since all the flow functions in the following relate to a single **Sys**, the flow function $f_{sys,ri}$ will be written below as f_{ri} for better readability. Let the set of variable values at time k be $R_k = \{ (r_i, r_{i,k}) \mid r_{i,k} \in \mathrm{val}(r_i) \}$. Not all variables in R_k may be needed to calculate each $r_{i,k+1}$. Any particular variable r_i may only depend on some of the variables in R_k as given by the structure of the processes and process communications; this subset is written $R_{i,k}$.

Definition 26. The posterior probability $r_{i,k+1}$ is defined by

$$P(r_{i,k+1} \mid R_{i,k}) = f_{ri}(R_{i,k}),\ R_{i,k} \subseteq R_k$$

The resulting structure can be drawn as a Bayesian network as shown in Figure 10. As long as flow functions can include the effect of program conditionals [27] it can be assumed $R_{i,k}$ is independent of k and hence that the evolution of variable values is a stationary process that can be captured as the *Dynamic Bayesian Network* shown in Figure 11.

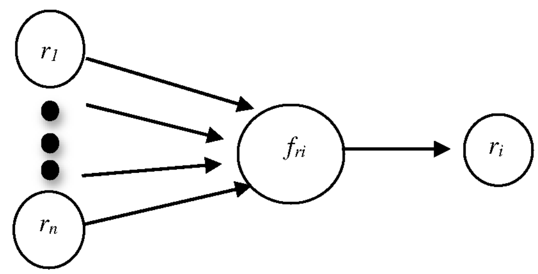

Figure 10. Flow function $f_{ri}(r_1...r_n)=r_i$ evaluation shown as a Bayesian Network

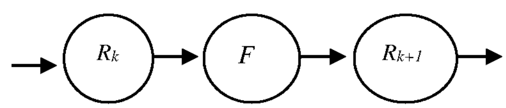

Figure 11. Dynamic Bayesian Network

Definition 27. The transition model of the DBN constructed for flow functions $f_{r1}, f_{r2}, ...$ is F, where

$$F(R_k) = (\ f_{r1}(R_{1,k}),\ f_{r2}(R_{2,k}), ...\ )$$

Sys *satisfies* G(**Q**,C) identifies a subset of the variables $V \subseteq R$, where V contains those variables whose values are constrained by the performance criterion G(**Q**,C).

Definition 28. A performance criterion G(**Q**, C) is *verified* if for some k

$$P(C(V_k) \mid R_{1:k}) > P_{min}$$

where P_{min} is either specified in C (as the probabilities were in the performance criterion Examples 14-16,) or is a default value, and where $R_{1:k}$ means the sequence of variable values from the first step to step k and where V_k are the values of the variables constrained by C at step k.

An observation model $GF(R_k)$ is introduced to implement this evaluation:

$$GF(R_k) = P(C(V_k) \mid R_k) \quad (10)$$

The verification conditions may be achieved on any filtering step, so the probability of achieving this is the disjunction of the probabilities on each step.

$$P(C(V_k)|\, R_{1:k}) = P(C(V_1)|R_1) + P(C(V_2)|R_2)P(R_2|R_1) + .. + P(C(V_{k-1})|R_{k-1})P(R_{k-1}|R_{1:k-2}). \quad (11)$$

This is written more compactly as:

$$P(C(V_k) \mid R_{1:k}) = \sum_{j=1}^{k} P(C(V_j) \mid R_j) P(R_j \mid R_{1:j-1}) \quad (12)$$

Since each R_j is linked to the one before in the DBN by the transition model $R_{j+1} = F(R_j)$, and verification condition satisfaction is related to R_j by the observation model $GF(R_j)$:

$$\begin{aligned} P(C(V_k) \mid R_{1:k}) &= \sum_{j=1}^{k} P(C(V_j) \mid R_j) \prod_{l=1}^{i} P(R_l \mid R_{l-1}) \quad (13) \\ &= \sum_{j=1}^{k} P(C(V_j) \mid R_j) F^j(R_1) \\ &= \sum_{j=1}^{k} GF(F^j(R_1)) \end{aligned}$$

While P_{min} gives a way to determine a successful verification, it does not allow the determination of a non-successful verification. One solution is to bound k as follows.

Definition 29. A performance criterion G(**Q**, C) is verified for **Sys** iff:

$$P(C(V_k)|\, R_{1:k}) > P_{min} \text{ and } k<K_{max}$$

where GF and K_{max} are determined from **Sys** *satisfies* G(**Q** ,C). GF is determine from eq. (10), and K_{max}, the number of DBN iteration steps is determined from the time behavior of **Sys** (Definition 21) and the time constraint on the mission.

B. Verification Examples

In this subsection, several examples are presented to illustrate how verification is accomplished using VIPARS and what verification results from this method look like. In [27] the issue of selecting a representation for PARS random variables is discussed and a *Mixture of Gaussian* model proposed. The example results presented here were calculated using that mixture model, but the representation issue is not addressed further here.

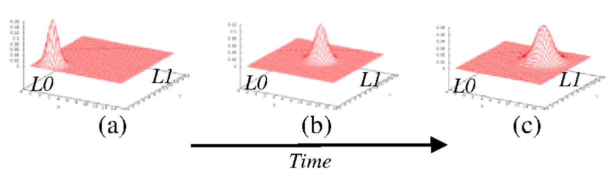

Figure 12. Three snapshots of the robot position distribution for increasing time, from motion start *L0* (a) to end *L1*(c)

Example 17 (Cont.) The robot controller in this example moves the robot from a point L0 to a point L1. The condition being verified is that the robot is at the point *L1* after some time $t<T_{max}$ with probability $p>P_{min}$.

Figure 12 shows the value of a position distribution at several steps during verification of that mission, that is, at several steps during the filtering per Definition 27. The robot position is a single peak distribution, and during filtering, the mean moves towards L1 and the variance expands (due to the influence of the noise in the robot model).

Figure 13 shows the value of the probability of the goal condition as a function of filtering iteration step (k in Definition 29). Figure 13(a) shows the case for this running example. The cumulative probability of being at L1 rises monotonically as the robot approaches L1. The initial low probabilities represent the cases when the robot motion error is so small that the robot arrives at the goal relatively quickly. Also output from

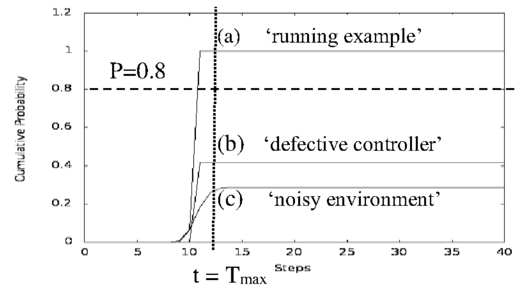

Figure 13. Cumulative probability of the Goal Condition versus DBN iteration step for three examples.

verification is the position distribution (e.g., that in Figure 12) for this mission at the iteration step where the probability of having arrived at L1 exceeds the (mission-designer) specified threshold; 80% in this example.

Example 18. The **MoveTo** process in the running example (Example 17) is replaced with a version in which the velocity calculated is δ to one side of the goal, $d(g\text{-}r)$ is replaced with $d(g\text{-}r+\delta)$. Figure 13(b) shows the cumulative probability for this logically defective controller, which never reaches the threshold probability of 80%.

Example 19. The **Robot** process in the running example is replaced with a version in which the noise Θ_h , Θ_v is increased. Figure 13(c) shows the cumulative probability for this (overly) noisy case, which again never reaches the threshold probability of 80%

Example 20. The PARS environment models need to be able to represent objects and obstacles when they are known. Figure 14 shows a Mixture of Gaussian (MoG) position distribution result for a waypoint mission through a narrow doorway and corridor. The MoG members are shown as shaded 1 Standard Deviation (SD) ellipses, the shading indicating the weight of the member. The smaller clusters to each side of the doorway in Figure 14(b,c) indicate the probability of missing the door and

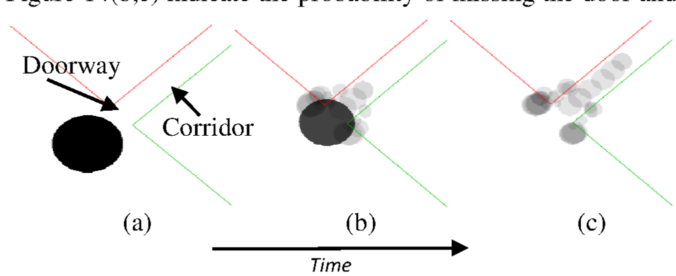

Figure 14. Position distribution snapshots for increasing time from start (a) through a door (b) and into a corridor (c).

hitting the wall. The member cluster smeared out in the corridor represents the 'safe' motion of the robot moving towards its goal. The environment and controller model for this example are outside the scope of this paper, but are presented in [27].

VII. VALIDATION RESULTS

It is not sufficient to demonstrate verification results for critical applications such as C-WMD robot missions; It is crucial to show also that the verification results correspond to the behavior of physical robots. In prior work [28], a series of measurements on a Pioneer 3-AT robot were conducted, so that the robot motion and sensing uncertainty distributions used in VIPARS could be calibrated for the Pioneer 3-AT robot on an indoor surface. The results of a validation of the performance predictions for the two missions described in Section III are now presented: first, for a multiple waypoint mission; and second, for an autonomous exploration mission. In each case, the details of the validation experiment are presented, then the PARS mission and VIPARS verification, and then the results are compared. Section IV.E describes the translation process from *MissionLab* to PARS; for each of the examples here, this procedure was followed manually.

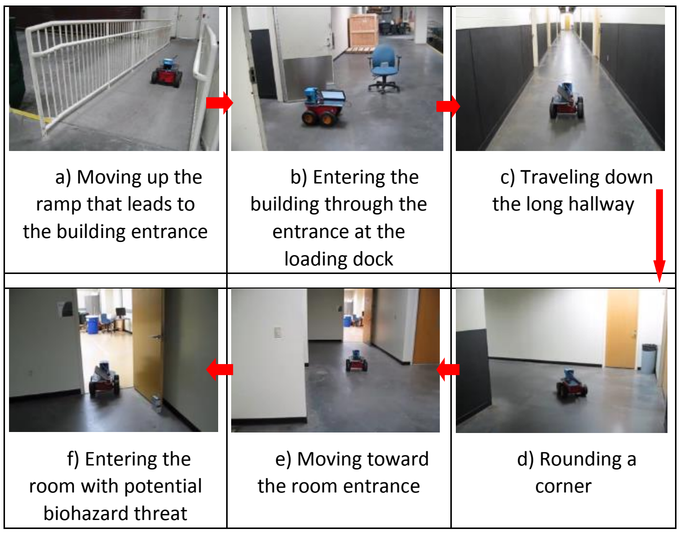

Figure 15. Snapshots of Pioneer 3-AT carrying out the multiple waypoint mission.

A. Multiple Waypoint Mission

The multiple waypoint mission was described in Section III.B.1 and the *MissionLab* FSA for that mission shown in Figure 3.

1) Validation Procedure

The mission was carried out with a Pioneer 3-AT robot (Figure 6). The mission area is approximately $60{\times}20$ meters. The robot started at the bottom of the ramp (Figures 2 and 15). Following the waypoints, the robot moved up the ramp that leads to the loading dock where an entrance to the building is located. The robot then entered the building and traveled down a hallway (approximately *40* meters in length), which leads to the room of interest located at the end of the hallway.

The performance criterion for the mission is whether the robot had gained access to the room of interest (i.e., reached the final waypoint, which resides in the room). The mission was run *40* times and the numbers of successes and failures were recorded. The result is shown in Table 2. No obstacle avoidance was active and only dead reckoning was used. Most failures observed were due to the robot being stuck at the corner near the third waypoint as in Figure 15d.

Table 2. Validation Result

# of Runs	# of Failures	# of Successes	P(Success)
40	12	28	70%

2) VIPARS Prediction

The waypoint *MissionLab* FSA of Section III, Figure 3 is translated to PARS as a sequence of behavior processes:

$$\begin{aligned}
&\textbf{Mission}\langle g1,g2,g3,g4\rangle\,(p,hi)(v,ho) = \\
&\quad \textbf{Turn}\langle g1\rangle\,(p,hi)(ho)\ ;\ \textbf{MoveToVC}\langle g1\rangle\,(p)(v)\ ; \\
&\quad \textbf{Turn}\langle g2\rangle\,(p,hi)(ho)\ ;\ \textbf{MoveToVC}\langle g2\rangle\,(p)(v)\ ; \\
&\quad \textbf{Turn}\langle g3\rangle\,(p,hi)(ho)\ ;\ \textbf{MoveToVC}\langle g3\rangle\,(p)(v)\ ; \\
&\quad \textbf{Turn}\langle g4\rangle\,(p,hi)(ho)\ ;\ \textbf{MoveToVC}\langle g4\rangle\,(p)(v)\ ; \\
&\quad \textbf{Turn}\langle g5\rangle\,(p,hi)(ho)\ ;\ \textbf{MoveToVC}\langle g5\rangle\,(p)(v)\ .
\end{aligned}$$

The mission is five instances of processes that turn the robot to face the goal **Turn**$\langle g1\rangle$, and then move the robot towards that goal, **MoveToVC**$\langle g1\rangle$. This information specifies the connections for the position input (p), the heading input (hi), the heading output (ho) and the velocity output (v). The system process is the parallel, communicating composition of the mission and environment processes. The **Robot** process is that used in Example 13 but with the information about heading and rotational uncertainty included. The process contains *no information* about walls or laser sensing to detect and respond to walls and obstacles and just moves the robot from waypoint to waypoint.

$$\begin{aligned}
\textbf{Sys} \;=\; &\textbf{Robot}\langle P0,H0\rangle\,(c2,c3)(c1,c4)\ \ | \\
&\textbf{Mission}\langle G1,G2,G3,G4\rangle\,(c1,c4)(c2,c3)
\end{aligned}$$

The capital letter variable values *P0, H0, G1,G2* and so forth are the initial conditions for the system: the initial position, heading, goal locations etc. The port connections $c1,\ldots,c4$ connect the position, heading and velocity ports on the mission to those in the environment model.

The performance criterion is the same as that Example 14, that the robot reach its final goal location by time T with a probability $p{>}P_{min}$. The system is analyzed by VIPARS using Definition 25, and keeping the subproblems independent as described in Note 6.

VIPARS reported a successful verification for this mission with final position distributions (in *mm*) shown in Table 3. We ran VIPARS several times with increasing values of P_{min} to determine a maximum value P_{max} for a successful verification, where P_{max} was calculated as largest probability threshold P_{min} (from Def. 28) where the mission still ended before the maximum time elapsed T_{max}). These are shown as the last column in Table 3 and the distribution data for the row is for that case. Since a failure could occur at any waypoint and the problems are independent, the probability for success is calculated as the product of success probabilities at each waypoint: P_{succ} =71.5%. The lowest P_{max} was for the third waypoint, with P_{max}*=81%.*

Table 3. VIPARS Waypoint Distributions. **WP#** is the waypoint number. **(μ_x, μ_y)** is the 2D mean position (mm) and **Σ** is the covariance for each waypoint. **P_{max}** is the largest P_{min} before T_{max}.

WP#	(μ_x, μ_y)	Σ	P_{max}
1	*(17468, 23585)*	*[2610, 0; 0,8830]*	***0.91***
2	*(17850, 21206)*	*[4675, 286; 286, 9449]*	***0.99***
3	*(59411,21639)*	*[14986, -608; -608, 48005]*	***0.81***
4	*(59092,33444)*	*[24717, -218; -218, 50625]*	***0.99***
5	*(60422, 39764)*	*[30051,-1048; -1048, 52273]*	***0.99***

3) *Comparison of Predicted and Measured Results*

Experiments show a success probability of 70% for this mission, given 40 runs with 12 failures. The predicted success rate is (rounding up) 72%. Predictions are statistically compared with the validation results using a *z-statistic* proportion test. The null hypothesis is H_0: $p_{succ}=0.72$ and H_a: $P_{succ}<0.72$. For applicability of the test, its necessary that $np_0=40\times0.72>10$. The z-statistic is calculated as $z=-0.28$, and $p(Z<-0.28)=0.3897$ from the standard distribution tables. Since $0.05<<0.3897$ this (emphatically) fails to reject H_0: $p=0.72$ at the 95% confidence level. The waypoint with the lowest P_{max} is the third waypoint. During validation it was observed that it was at this waypoint experimental trials most frequently had failures. A mission designer could leverage this information from verification, for example, to modify the motion behavior for the third waypoint to improve the probability of success for the overall mission.

B. *Autonomous Exploration Mission*

The autonomous exploration mission was described in Section III.B.2 and the *MissionLab* mission FSA shown in Figure 5.

1) *Validation*

For the *Biohazard Search* mission, the operating environment of the robot is a room of dimension approx. 10×12 meters, Figure 4. The room is covered with tile flooring and is well lit by florescent lights. The major area of the room is empty except some items along the walls (e.g., cabinets, storage crates).

The Pioneer 3-AT has a laser scanner for obstacle avoidance and a forward-facing camera for biohazard detection. The camera has a field of view of 39.6 degrees. The biohazard is represented by a red biohazard bucket, Figure 4. The color of the biohazard bucket is used for biohazard detection.

Table 4. Validation Result

Mission	# Trials	# Successes	Performance
Biohazard Search	106	88	83.0 %

The complete validation experiment consists of 106 trial runs of the Biohazard Search mission. The location of the biohazard was uniformly distributed with respect to the room, requiring a total of 106 trials. For each trial, the robot starts at the entrance of the room and proceeds to search the room with the control program described in Figure 5. Each trial is completed when the robot locates the biohazard. Mission success is defined by the performance criteria. For this mission, the criterion is that the robot needs to find the biohazard in 60 seconds. The time it takes for the robot to locate the biohazard is recorded for each trial. Table 4 shows result of the validation experiment.

2) *VIPARS Prediction*

The PARS representation of the *Biohazard Search* mission is:

$$
\begin{aligned}
\textbf{Mission} \quad = \quad & \textbf{NotDetected} \;;\; (\textbf{Detected} \;\#\; \textbf{Wander}) \;| \\
& \textbf{Detected} \;;\; (\textbf{Near} \;\#\; \textbf{MoveToward}) \;| \\
& \textbf{Near} \;;\; \textbf{Stop}
\end{aligned}
$$

The **Mission** process consists of trigger processes, such as **Detected** and **Near**, and behavior processes such as **Wander** and **MoveToward.** Some are implemented as basic processes and others as PARS networks, to replicate the equivalent *MissionLab* behaviors, as discussed previously in Section IV.E.

Different missions have different requirements that the robot has to meet. For the *Biohazard Search* mission, we are interested in time performance, successful detection of biohazard, and correct identification of the biohazard. These performance criteria are expressed in PARS as a performance specification network based on eq. (3):

$$
\begin{aligned}
\textbf{Q} \quad = \quad & \textbf{Delay}\langle t\rangle \;;\; (\,\textbf{At}\langle p\rangle \;|\; \textbf{Biohazard}\langle q\rangle\,) \\
& \textit{for some } t<T_{max}\,,\; P(|\,p-q\,|<\varepsilon)>P_{min}
\end{aligned}
$$

The **Biohazard**⟨p⟩ process indicates the location of the biohazard. Verification asks whether the mission will achieve this liveness condition for $t<T_{max}$ with at least probability P_{min}.

The robot model used in this verification is the same as for the previous. However, additional sensor and environment modelling is necessary.The sensor models are separated from the robot model for modularity; the same robot platform can be equipped with different external sensors. For this mission, the Pioneer 3-AT robot is equipped with a camera for biohazard detection and a SICK laser scanner for obstacle avoidance. The sensor model is a composite model of these sensors, which can be expressed in PARS as:

$$
\begin{aligned}
\textbf{Sensors} \quad &= \quad (\,\textbf{Camera} \,|\, \textbf{Laser}\,) \\
\textbf{Camera} \quad &= \quad (\textbf{In}\langle p\rangle\langle r\rangle \,|\, \textbf{In}\langle cs\rangle\langle c\rangle\,) \;;\; \textbf{Out}\langle cs, fc(r,c)\rangle \;;\; \textbf{Camera} \\
\textbf{Laser} \quad &= \quad (\textbf{In}\langle p\rangle\langle r\rangle \,|\, \textbf{In}\langle ws\rangle\langle sp\rangle) \;;\; \textbf{Out}\langle ls, fl(ws,sp)\rangle \;;\; \textbf{Laser}
\end{aligned}
$$

The structure of the sensor models are similar. They accept data from an environment model including the robot position (r), carrry out a sensor specific model function (*fc* for the camera model and *fl* for the laser model) on that data and make it available on a port which the Mission process can read.

The fundamental problem for the verification of robot behavior is the interaction between the robot and the environment. Undesirable robot behaviors might emerge through this interaction which might not have been foreseen by the robot programmer/operator. The targeted environment for the *Biohazard Search* mission is an indoor environment, Figure 4. The PARS model of the environment is:

$$
\begin{aligned}
\textbf{Env}\langle g,b,ws\rangle \quad = \quad & (\textbf{Out}\langle ps,ws\rangle \;\#\; \textbf{Out}\langle cs,b\rangle \;\# \\
& \textbf{In}\langle p\rangle\langle r\rangle \;\#\; \textbf{Biohazard}\langle b\rangle\,) \;; \\
& (\textbf{Inside}\langle g,r\rangle\langle fs\rangle \;| \\
& \textbf{Outside}\langle g,r\rangle\langle ws\rangle\,) \;;\; \textbf{Env}\langle g,b,ws\rangle
\end{aligned}
$$

Random variable values, such as the robot position (r) and the location of the biohazard (b) are represented as Gaussian Mixtures [27]. Where r is calculated by the **Robot** process, b is a constant that expresses what is known about the biohazard location: in this case, it's a uniform distribution within the room, which is directed to the camera sensor. The variable value g captures what is known about the room, which in this case is that it's an empty rectangle. There are no obstacles in this room; the problem being addressed is whether this controller, using a wander behavior will find its target within the time limit with a sufficiently high probability given the sensor and actuator probabilities. The **Env** process tests the robot position probability distribution and separates it into two mixtures: one representing the portion of the distribution that is inside the room (*fs*), and one that represents the portion that would collide with the room walls (*ws*) which is channeled to the sensors.

The PARS models of the control program, robot, sensors, and the environment form the System process **Sys**, which is the parallel, communicating network (dropping port connection labels for better clarity):

$$\mathbf{Sys} \quad = \quad \mathbf{Mission} \,|\, (\mathbf{Env} \,|\, \mathbf{Robot} \,|\, \mathbf{Sensors})$$

The **Sys** process is then analyzed by VIPARS to determine if it satisfies all the constraints specified by the property specification process network (i.e., the **Q** process).

3) Comparison of Verification and Validation Results

Verification of the *Biohazard Search* mission predicted an 85% mission success probability, while the validation experiments showed an actual robot succeeds 83% of the time based on 106 trials with 18 failures. Validation and verification results are compared using a z-statistic proportion test to determine if any statistical significant difference exists between these results. The null hypothesis is H_0: $p_{succ}=0.85$, and the alternative hypothesis is H_a: $P_{succ}<0.85$. The z-statistic is z = -0.58, which resulted in P(Z<-0.58)= 0.28 from the standard distribution table. Since 0.28>0.05, this fails to find any statistically significant difference between the verifier's performance prediction and the actual performance from the validation experiments. It is safe to conclude that the VIPARS' performance guarantee, the 85% probability of mission success with respect to the performance criteria for the *Biohazard Search* mission, is a valid prediction.

VIII. Conclusions

A novel approach to verification of performance guarantees for behavior-based robot programs was proposed in this paper. The approach differs from prior work in its avoidance of the concept of state via the use of a process algebra framework. The general case of software verification runs afoul of the halting problem. To address this fundamental limitation, most work therefore focuses on specific cases; this paper focused on a process algebra structure that captures behavior-based programming well: parallel interacting systems of tail recursive (TR) processes. TR processes have the useful feature that they easily allow the construction of recurrent flow-functions that capture how the TR processes transform variable values on each recursive step. The *SysGen* algorithm constructs a single *system period process* from the bodies of each component process, if one exists. The algorithm, *FloGen* that extracts the flow-function for the system period process by following and resolving communications over port connections between the processes in the system period process, was also presented.

This approach was developed to work with *MissionLab* so that the verification process could be completely automatic. This could be done because TR processes can be generated directly from the *MissionLab* behavior-based robot programs. We argue that other behavior-based robot programming approaches will also transfer fairly easily to the TR process representation, but other robot programing approaches and general purpose programs may not be as easy to map. Furthermore, MissionLab can generate software for a variety of robot software architectures.

To model uncertainty, which is a sine qua non for realistic robot results, the process algebra is extended to allow processes to have random variables. It is shown that the system flow function in this case can be mapped to a Dynamic Bayesian Network (DBN). The verification problem can then be recast as a DBN filtering problem.

Prior work [28] using the method described here reported a validation of one and two move missions for a Pioneer 3-AT robot in indoor conditions at various velocities. The results show strong statistical evidence of the predictive power of the approach. In this article, that validation is extended to a multiple waypoint mission and an autonomous exploration mission. Empirical testing of the multiple waypoint mission yielded a 70% success probability. The VIPARS prediction was 72%. The environmental model used in VIPARS for this example did not include walls or wall sensing. The second example, an autonomous exploration mission, did include these features as well as a more flexible control strategy. Empirical testing yielded an 83% success probability. The VIPARS prediction was 85%. Both experiments yielded statistically strong results.

There are two important extensions of this work in progress: dealing with multiple robots and dealing with environments that include obstacles. Although a C-WMD mission might involve single robot missions, multiple robot missions are also important. A crucial next stage in this work is to determine if it is effective for multi-agent as well as single-agent scenarios. Although Example 20 (Figure 14) showed interaction with obstacles/walls, the verification missions presented here did not include this work. Future missions will include environments that have uncertainty related to obstacle locations and terrain features as well as sensor and actuator uncertainty.

IX. Bibliography

[1] M. Hinchey and J. Bowen, High Integrity System Specification and Design, London: FACIT Series, Springer-Verlag, 1999.

[2] C. Baeir and J.-P. Katoen, Introduction to Model Checking, Cambridge MA: MIT Press, 2008.

[3] R. Jhala and R. Majumdar, "Software Model Checking," *ACM Computing Surveys,* vol. 41, no. 4, 2009.

[4] H. Kress-Gazit, E. Fainekos and G. Pappas, "Temporal Logic based Reactive Mission and Motion Planning," *IEEE Trans. on Rob.,* vol. 25, no. 6, pp. 1370-1381, 2009.

[5] M. Kloetzer and C. Belta, "Automatic Deployment of Distributed Teams of Robots from Temporal Logic Specifications," *IEEE Trans. on Rob.,* vol. 26, no. 1, pp. 48-61, 2010.

[6] D. MacKenzie, R. Arkin and R. Cameron, "Multiagent Mission Specification and Execution," *Autonomous Robots,* vol. 4, no. 1, pp. 29-52, 1997.

[7] T. Bolognesi and E. Brinksma, "Introduction to the ISO Specification Language LOTOS," *Computer Networks & ISDN Sys,* vol. 14, no. 1, pp. 25-59, 1987.

[8] S. Karaman, S. Rasmussen, D. Kingston and E. Frazzoli, "Specification and Planning of UAV Missions: A Process Algebra Approach," *Amer. Contr. Conf.*, St Louis MO, 2009.

[9] M. Steenstrup, M. Arbib and E. Manes, "Port Automata and the Algebra of Concurrent Processes," *JCSS,* vol. 27, no. 1, pp. 29-50, 1983.

[10] R. Ramadge and W. Wonham, "Supervisory Control of a Class of Discrete Event Processes," *SIAM J. Control and Optimization,* vol. 25, no. 1, pp. 206-230, 1987.

[11] G. Labinaz, M. Bayonmi and K. Rudie, "Modeling and Control of Hybrid Systems: A Survey," *IFAC 13th World Cngr.*, 1996.

[12] N. Napp and E. Klavins, "A Compositional Framework for Programming Stochastically Interacting Robots," *Int. Journal of Robotics Research,* vol. 33, no. 2, 2011.

[13] B. Johnson and H. Kress-Gazit, "Probabilistic Analysis of Correctness of High-Level Robot Behavior with Sensor Error," in *Robotics Science and Systems*, Los Angeles CA, 2011.

[14] D. Lyons, R. Arkin, P. Nirmal and S. Jiang, "Designing Autonomous Robot Missions with Performance Guarantees," in *Proc. IEEE/RSJ IROS*, Vilamoura Portugal, 2012.

[15] D. Lyons and R. Arkin, "Towards Performance Guarantees for Emergent Behavior," in *IEEE Int. Conf. on Rob. & Aut.*, 2004.

[16] R. C. Arkin, Behavior-Based Robots, Cambridge MA: MIT Press, 1998.

[17] J. Casper and R. R. Murphy, "Human-robot Interactions during the Robot-assisted Urban Search and Rescue Response at the World Trade Center," *IEEE Trans. on SMC Part B,* vol. 33, no. 3, pp. 367-385, 2003.

[18] R. Arkin, D. Lyons, S. Jiang, P. Nirmal and M. Zafar, "Getting it Right the First Time: Predicted Performance Guarantees from the Analysis of Emergent Behavior in Autonomous and Semi-autonomous Systems," in *Proceedings of SPIE. Vol. 8387.*, Baltimore MD, 2012.

[19] D. MacKenzie and R. Arkin, "Evaluating the Usability of Robot Programming Toolsets," *Int. Journal of Robotics Research,* vol. 4, no. 7, pp. 381-401, 1998.

[20] J. Baeten, "A Brief History of Process Algebra," *Elsevier Journal of Theoretical Computer Science – Process Algebra,* vol. 335, no. 2-3, 2005.

[21] D. Friedman and M. Wand, Essentials of Programming Languages, Cambridge MA: MIT Press, 2008.

[22] C. Boem and G. Jacopini, "Flow Diagrams, Turing Machines and Languages with only Two Formation Rules," *CACM,* vol. 9, no. 6, 1966.

[23] D. MacKenzie, "Configuration Network Language (CNL) User Manual," College of Computing, GATech, Atlanta GA, 1996.

[24] R. De Nicola, "Extensional Equivalences for Transition Systems,," *Acta Informatica,* vol. 26, no. 2, pp. 211-237, 1987.

[25] I. Ripoll and R. Ballester-Ripoll, "Period Selection for Minimal Hyperperiod in Periodic Task Systems," *IEEE Trans. Computers,* no. 62, pp. 1813-1822, 2013.

[26] S. Russel and P. Norvig, Artificial Intelligence, Prentice-Hall, 2010.

[27] D. Lyons, R. Arkin, T.-L. Liu, S. Jiang and P. Nirmal, "Verifying Performance for Autonomous Robot Missions with Uncertainty," in *IFAC Intelligent Vehicle Symposium*, Gold Coast Australia, 2013.

[28] D. Lyons, R. Arkin, P. Nirmal, S. Jiang and T.-L. Liu, "A Software Tool for the Design of Critical Robot Missions with Performance Guarantees," in *Conference on Systems Engineering Research (CSER'13)*, Atlanta GA, 2013.

Appendix I: Computational Complexity of *SysGen.*

Let n_p be the number of processes in the system. Let each process have at most n_{io} port communication operations. Let the communication map, *cm,* specify which ports are connected to which other ports. This map must obey the following constraints:

1. Input ports are connected *only* to output ports, and output ports are connected *only* to input ports.
2. Each port is connected to *at least* one other port in the system, with the worst-case fan-in or fan-out of connections being k_f.
3. No port is connected to another on the same process.

A port is considered ready to communicate only when during the execution of the process, an **In** or **Out** process is using the port for communication. *cm* and the system of processes are constrained for complexity reasons so that if a port is ready to communicate, then either

- at most one of its k_f connected ports is also ready, or
- it makes no difference to the computation which of the k_f connected ports receives the communication.

Without this constraint, all potential connected ports have to be checked, and backtracking may be necessary if a candidate connected port eventually results in a deadlock. This branching search introduces exponential complexity. However, introducing the constraint does not significantly restrict what can be represented: it just means that if, for example, the Robot process sends position information to a Laser and a Camera process, it does so in sequence (second bullet item above).

There are n_p* n_{io} communication operations in the system of processes. When a communication operation is ready, there is one, and at most one, other port with which it communicates. If that port is not ready to communicate, then *SysGen* identifies it. Otherwise, *SysGen* completes the system period process after making $(n_p * n_{io})/2$ connections. Since each port could have k_f possible partners, the worst case complexity is

$$k_f * (n_p * n_{io})/2. \tag{A1}$$

Notice that disabling (parallel-min) and parallel (parallel-max) compositions of communication operations y simply add choices for each of the $(n_p * n_{io})/2$ connections, but with k_f still being the maximum number of choices.

Appendix II: Computational Complexity of *Flogen*.

The complexity of *FloGen* depends on the number of component processes, n_p, and the number of variables of each, n_v, since each variable will generate one flow function. If there are n_{io} communication operations in each process, then in the worst case, each variable will be involved in every communication, and each communication will go through all n_p processes for substitutions. That will require

$$n_v * n_p * n_{io} \tag{A2}$$

substitutions in total in this worst case.