8-2012

Statistical Software Properties: Definition, Inference and Monitoring

Javier A. Darsie
University of Nebraska-Lincoln, javier.darsie@gmail.com

STATISTICAL SOFTWARE PROPERTIES: DEFINITION, INFERENCE AND MONITORING

by

Javier Darsie

A THESIS

Presented to the Faculty of

The Graduate College at the University of Nebraska

In Partial Fulfilment of Requirements

For the Degree of Master of Science

Major: Computer Science

Under the Supervision of Professor Sebastian Elbaum

Lincoln, Nebraska

August, 2012

STATISTICAL SOFTWARE PROPERTIES: DEFINITION, INFERENCE AND MONITORING

Javier Darsie, M. S.

University of Nebraska, 2012

Adviser: Sebastian Elbaum

Software properties define how software systems should operate. Specifying correct properties, however, can be difficult and expensive as it requires deep knowledge of the system's expected behavior and the environment in which it operates. Automated analysis techniques to infer properties from code or code executions can mitigate that cost, but are still unable to go beyond state properties and the simplest patterns of temporal properties. This limitation renders properties that sacrifice fault detection power.

To address this problem, we introduce a new type of software properties called *statistical properties*, which characterize significant statistical relationships among the values of variables across program states. We define an approach to infer these relationships automatically and support their monitoring while controlling the tradeoffs between overhead and the precision and recall of the inferred properties.

We perform several experiments to assess the approach in the context of distributed robotics applications. Our findings indicate that the inferred statistical properties can be use to generate precise and cost-effective models capable of detecting faults in software systems while keeping the number of false positives close to zero and previous knowledge of the software system design and behavior unnecessary.

ACKNOWLEDGMENTS

My sincerest thanks to all those who have participated in the completion of this thesis. I found support and encouragement from a large list of persons, which contributed in many different ways, and I am sorry it is not possible to list them all on just one page. To my family, friends and colleagues, thank you. It would not have been possible without you.

I would like to specially thank the invaluable guidance of my advisor, Sebastian Elbaum. Any recognition to his support and dedication will fall short. I would also like to thank all the Computer Science faculty and staff, most specially those on my committee, Carrick Detweiler and Matthew Dwyer, for their feedback and suggestions.

My deepest thanks to my father, mother, siblings and my beloved wife Ana, who makes me the happiest person every single day. I would also like to thank my school and college best friends Atuk, Berna, Emi, Ernesto, Maju, Marito, Patri, Lucas, Juanma, Juancho, Mati, Mariano and Camilo. The students in the ESQuaReD and Nimbus lab deserve special acknowledgement, especially Adrian, Brady, Charlie, Elena, Katie, Rafael, Rahul, Ben, Brent, Dave, Hengle, John-Paul, Megan, Paul. They made university felt like home.

This work was supported in part by Air Force Office of Scientific Research through grant #9550-09-1-0687.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Software properties characterize how a system operates or should operate. Such characterization aims to capture different functional and non-functional attributes to support various forms of validation and verification. This aim in turn influences how the properties are specified and checked.

Consider for example, the Java construct $assert(x\,!=null)$. This construct specifies a property in the code, which may have been derived by the programmer based on his domain or program knowledge, or by an automated inference tool through the analysis of the program structure or of some of its execution traces. This property is represented by a boolean expression, which will be checked at run-time, raising an exception if the subset of the program state represented by variable x is $null$.

Now consider the property $\Box(Booting \rightarrow \bigcirc CheckMem)$ which asserts that it is always the case that $CheckMem$ occurs right after $Booting$. This is a richer property in that it operates over a set of states, but it is also more challenging to specify correctly as it includes richer semantics. It is also more expensive to check as it involves keeping track of data across multiple states and may require constructs that are not part of standard programming languages. Yet, in spite of its cost, such temporal properties are often critical for reactive systems that maintain an ongoing interaction with their environment rather than produce some final value upon termination [37].

These two sample properties illustrate some of the trade-offs to consider between properties involving one state (state-properties) versus properties involving multiple states (temporal-properties). Such trade-offs include the costs of correctly specifying and checking those properties and their benefits in terms of fault detection power.

In this work we propose a new type of properties, **statistical properties**, that provide an intermediate choice in this spectrum. These properties aim to capture significant statistical relationships between a window of values of variables across multiple program states which makes them interesting from several perspectives.

Statistical properties have several interesting attributes. First, they are a natural match for systems that have a distribution of outputs or events that can be statistically fitted such as those systems that employ control algorithms or planners, or that propagate messages across distributed and independent components in a somewhat consistent manner. Second, they can be inferred with similar effectiveness and level of automation as state-properties, avoiding some of the challenges involved in inferring the more complex temporal properties. Third, they offer simple and intuitive parameters that can be adjusted to meet monitoring costs, and can be easily extended by providing more statistical relationships.

Statistical properties, however, are not always appropriate or efficient. Properties that cannot be easily map to a distribution or statistical relationship may be more appropriate for other types of representation. Statistical properties are also not rich enough to capture relationships that include implicit system variables such as time or locality, and may become impractical in the presence of many events.

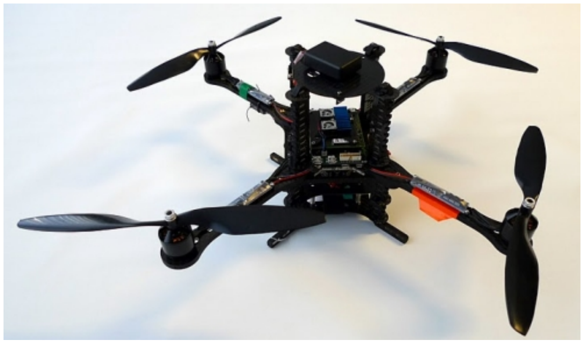

Figure 1.1: Asctec Hummingbird quad-rotor

In the rest of this chapter we define the problem in more detail, illustrate our approach, and detailed the contribution of the work.

1.1 Motivating Example

Consider a system that controls a quad-rotor helicopter like the one shown in Figure 1.1. The software system enables a pilot to move the quad-rotor using a joystick controller and it includes a height controller that automatically adjusts the quad-rotor thrust to maintain a target height without human intervention. Figure 1.2 shows a graph of the software components. Each vertex corresponds to a component of the distributed system and the edges specify the type of message sent between components. So, for example, the vertex */asctecjoy* corresponds to the component in charge of transforming raw data from the joystick controller into commands, and the edge with the label */TargetHeight* represents messages that set the target height.

Specifying and monitoring properties of this type of systems is difficult because their interaction with rich physical environments leads to many potential scenarios where properties can be easily over or under specified. Another difficulty is that the sensory information that the systems collect is often noisy, which is particularly problematic for tools trying to automatically infer properties from run-time data.

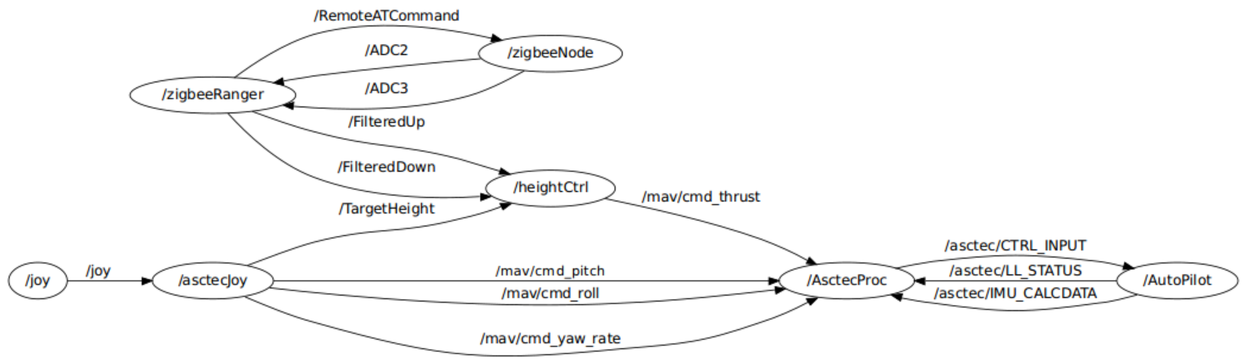

Figure 1.2: Graph representation of the distributed system under analysis

Consider the *zigbeeRanger* component (Figure 1.2) which constantly receives raw data about the quad-rotor's distance to the floor and publishes the filtered distance value. The process to produce the filtered distance is triggered whenever a message containing the raw distance, */ADC3*, is received (the content of messages with label */ADC2* is ignored by the system and does not have any impact in the component behavior).

Since changes in the environment can cause abrupt changes in the ranger's readings, the filtering process reduces the variance of the raw height data it receives. This process is as follows. First, *raw* distance readings are stored in a queue and the *average* of the raw readings received in the last s seconds is calculated. Then, for each *raw* value in the queue (starting from the most recent value), the absolute value of the difference between *average* and *raw* is calculated and compared against a constant *threshold*. If the difference is lower than *threshold*, *raw* is considered noise-free and it is used to compute the *filtered* distance, where *filtered* distance is the average of the latest 3 noise-free raw distances. Given this explanation, it seems that a strong property of the zigbee ranger component must capture the fact that *filtered* distance is a function of the raw height values, and *threshold*.

1.1.1 State Properties

State properties can characterize the set of valid program states of software systems by describing the correct values of its state variables and the relationship between them using propositional or first-order logic. They can be specified by someone with previous knowledge of the system or automatically inferred by program analysis tools that instantiate predefined property templates with the program variables and their values collected at run-time.

Consider for example a proposition like $raw - filtered < threshold$. This property would be effective in discovering outliers in the raw readings and could be generated by automatic inference generation tools like Daikon[21]. However, the effectiveness of this characterization relies heavily in the choice of *threshold* which may be problematic to determine automatically for systems like this where the data is noisy. Also, the relationships between variables may be more complicated and only hold for some scenarios (e.g., it may not hold if the quad-rotor is on the ground), which further challenges inference tools.

Last, state properties will be unable to capture anomalies that manifest across several states, like when the UAV oscillates without converging to the target height.

1.1.2 Temporal Properties

Temporal properties are those that can be described using first-order logic or temporal logic, where each proposition contains one or more temporal operators and one or more propositional variables which represent an event or state of the system. Temporal properties have an implicit notion of time that allows them to reason about the ordering of events or states.

These properties must be specified by someone with previous knowledge of the system or, for their simplest types (e.g., properties with just two events appearing in specific patterns), inferred from an execution trace of the system. In the context of the sample system under analysis, every message type could represent a propositional variable and every published message could represent an occurrence of that proposition.

Automatic inference techniques for temporal properties try to match the relationship between various types of events to a series of templates in the form of specification patterns. For example, an instance of the *Response* temporal pattern [19] could be automatically inferred for our example. This pattern states that after receiving the raw distance to the floor (event P), the system next action is to publish the filtered distance to the floor (event Q). Using linear temporal logic, the property specification is defined as $\square(P \rightarrow \bigcirc Q)$ and should be read as *it is always the case that an occurrence of event Q will happen immediately after an occurrence of event P*. While it is true that the automatically inferred temporal property holds, it can only judge the behavior of the *zigbeeRanger* component by the type of messages received and sent, but not by of the data they transmit, which hurts fault detection when the payload of the events is the key to the faults.

With some knowledge of the system, more meaningful properties can be defined by defining richer propositions that represent a monotonic relationship between the parameter values. Let's assume that it is expected that *filtered* values change in the same direction than *raw* values. The new propositions could represent the direction in which the value of a given parameter changes with respect to the previous event of the same type. Therefore, given the last occurrence of event P_i and the previous occurrence of event P_{i-1}, we could define as an occurrence of event type PE(P Equal) when the values of $P_i.value$ and $P_{i-1}.value$ are the same. If $P_i.value$ is lower than $P_{i-1}.value$, then we have an occurrence of PL(P Lower). Otherwise, the occurred event is PG(P Greater). Once we have defined this new family of propositions, properties stating that an occurrence of PE is immediately followed by an occurrence of QE can be defined to model the expected behavior. So, these new instances of the 'Response' pattern are specified as $\Box(PG \rightarrow \bigcirc QG)$, $\Box(PE \rightarrow \bigcirc QE)$ and $\Box(PL \rightarrow \bigcirc QL)$. The first one for example indicates that it is always the case ($\Box$)that an occurrence of PG (the new value of raw height is greater than the previous one) is immediately followed ($\bigcirc$) by an occurrence of QG (the new value of filtered height is greater than the previous one).

Even when the new propositions do not model the system perfectly, they are more accurate than the one insensitive to the message data. However, the definition of this new set of richer propositions requires someone with domain knowledge to map the relationship between parameters to new propositions since the state of the art in automatic inference tools cannot infer such complex properties.

1.1.3 Probabilistic Temporal Properties

Probabilistic temporal properties are described using probabilistic temporal logics. These logics extend temporal logics by adding discrete time constraints and the capability of describing the probabilistic nature of some software systems. As systems that define or require the specification of these behaviors are generally mission critical and the properties quite complex, system designers with previous domain knowledge are preferred rather than inference tools to take care of properties specification.

In the context of the sample system under analysis, let's now suppose that the system under analysis is considered to be more stable and the system's developers would like to define a less strict behavior so event Q to is not required to occur immediately after event P. In that case, they could specify a *soft deadline* or property that states that after receiving the raw distance to the floor (P) there is at least a 90% probability that the filtered distance to the floor will be published (Q) within 5 time units. Using probabilistic temporal logic, this property can be specified like this $\Box[(P \rightarrow \Diamond^{\leq 5}_{\geq .9} Q)]$.

As probabilistic temporal logics extend temporal logics, probabilistic temporal properties share the benefits and downsides of temporal properties against state properties. Comparing probabilistic temporal properties and temporal properties, probabilistic temporal properties offer more expressiveness capabilities because they provide more branching options. While temporal properties can specify that a sequence of event must occur in every path or at least one path, probabilistic temporal properties can define the exact percentage of executions path in which those events must occur. However, their specification requires a more accurate and detailed knowledge of the system under analysis and its environment.

1.1.4 Statistical Properties

We have briefly discussed state, temporal and probabilistic temporal properties and their trade-offs between inference cost and meaningful characterization. We believe that statistical properties are an intermediate alternative in that trade-off spectrum. While state properties rely on propositional formulae and ignore the execution history, temporal properties rely on abstract states derived using the execution history, and probabilistic temporal properties offer a larger expressiveness power at cost of little inference automation, statistical properties model the system behavior by finding significant statistical relationships between a window of values of a set of state variables.

In the same way as state and temporal properties automated inference tools try to find instances of predefined property templates using execution data, inference of statistical properties tries to find instances of statistical relationships between variables and their values. For example, the subsystem under analysis is a potential candidate for the *correlation* template property. To determine if raw and $filtered$ are an instance of this property, our approach keeps two windows of their values and calculates the correlation coefficient between those two windows every time new values are assigned to both of them. The property is discarded if a statistical test determines that it cannot be assured that the correlation relationship does not exist by chance.

Contrary to state property $raw - filter < threshold$, the power of the statistical property *correlation* is less dependent on the quality of the test suite and it can adapt easier to untested scenarios. For example, it is possible that $raw - filter$ becomes greater than $threshold$ by an insignificant amount after a new target height is set. While this is not a fault in practice, it will be considered a violation to the state property even if it happens once. A *correlation* property will be robust enough to handle that scenario because it considers the relationship between a window of values of raw and $filter$. Furthermore, a state property would not capture that the quadrotor is oscillating as long as each individual state does not violate a state property. A statistical property describing the correlation between these variables may expose this undesired behavior.

The robustness comes at a cost. The computation of the correlation coefficient is more costly than the proposition $raw - filter < threshold$, which increases the inferring and monitoring time of the *correlation* property. Still, in general, the inference is less costly for statistical properties than temporal properties because the latter have to keep track of an undetermined number of program states to discard

Attributes	**Property Type**			
	State	**Temporal**	**Probabilistic Temporal**	**Statistical**
Number of States	One	Multiple	Multiple	Bounded
Expressiveness	Low	High	High	Medium
Logic	Propositional	Temporal	Probabilistic	First Order
Automation Support	High	Medium	Low	High
Inference Cost	Low	High	High	Medium
Monitoring Cost	Low	High	High	Medium

Table 1.1: Property Types Comparison

an instance of a property template. The monitoring cost of temporal properties and probabilistic temporal properties is proportional to their complexity which is determined by the number of variables, states, and temporal operators. For statistical properties, the cost is fixed and adjustable, and most statistical functions can be computed incrementally.

In order to concisely position statistical properties in the software properties spectrum, Table 1.1 summarizes the main attributes and trade-offs of each type of software properties that we have discussed in this section. The column corresponding to statistical properties contains our claims about them and the other columns provide an approximation of the attributes of the other types of properties.

1.2 Research Contributions

In this work we present statistical properties that can be inferred and monitored automatically and cost-effectively. Concretely, our contributions are:

- Formal definition of statistical properties, their constituent elements, and the explanation of how overhead and fault detection power can be adjusted to satisfy the needs of their users.

- Development of a dynamic analysis approach to infer statistical properties, optimize their window size, and monitor them.

- The inference and monitoring approach, including a mechanism to build and refine models of software systems, and a framework to extend the approach by adding statistical properties defined by third parties.

- Study of the cost-effectiveness of statistical properties in generating models for 3 distributed robotics systems. The results indicate that precision, recall and overhead can be manipulated to achieve a desired range of performance. Comparison between the cost of inferring and the fault detection power of properties generated by our tool against Daikon, a state properties inference tool.

1.3 Thesis Overview

After introducing statistical properties and their trade-offs against state and temporal properties, the rest of the thesis is organized as follows. Chapter 2 presents related work pertaining to the inference and monitoring of software properties. In Chapter 3 we formally define statistical properties and provide the details of the inference and monitoring processes. We also present a tool that implements our approach. Chapter 4 presents the results of a study performed to evaluate if statistical properties are able to effectively model distributed robotics system by distinguishing correct and faulty behaviors. Chapter 5 discusses the conclusions of this work and outlines our plans for the future of this work.

Chapter 2

Background and Related Work

This chapter focuses on introducing the three types of software properties that have kept the attention of researchers: state properties, temporal properties and probabilistic temporal properties. These types of properties have being shown to be useful in characterizing the behavior of software systems in many situations, have been targeted for inference, and used extensively in the context of run-time verification. This chapter defines these types of properties and presents previous research efforts to infer and verify software properties.

2.1 Software Properties

To fully position statistical properties in the cost-effectiveness trade-off spectrum, we first introduce the definition of *program state* and *transition systems*, and then present state and temporal properties along with their benefits and limitations.

2.1.1 Program State

At a specific time t, a *state* s of a given program S that has a set of *state variables* X, is defined as the set of key-value pairs such that the *key* component refers to a state variable $x \in X$ and the *value* component $v(x)$ represents the current value of x. The set of state variables is frequently defined as a subset of all program variables and the criteria to select them depends heavily on the characteristics and/or behavior of each program.

Definition (Program State) Given system $S = (X, D, dom)$ where:

1. X is the finite set of state variables of S,
2. D or domain is a non-empty set of values,
3. dom is a mapping from X to the set of non-empty subsets of D. For each state variable $x \in X$, the set $dom(x)$ is called the domain for x.

then, state s of system S is a function $d : X \rightarrow D$ such that for every $x \in X$ and its value $v(x)$, $v(x) \in dom(x)$. [31]

The previous definition implies that every program, at every moment, is in one and only one state defined by the values of its state variables.

As an example of program state, if a given system has state variable $x \in X$ and domain $D \subseteq \mathbb{N}$, then it is possible to define state $s_t = \{(x, 1)\}$ if $x = 1$ at time t or state $s_u = \{(x, 0)\}$ if $x = 0$ at time u. Figure 2.1 illustrates those states.

2.1.2 Transition Systems

Transition or state-full systems need to reason about sequences of program states and how they change in time in order to achieve their goals and perform correct

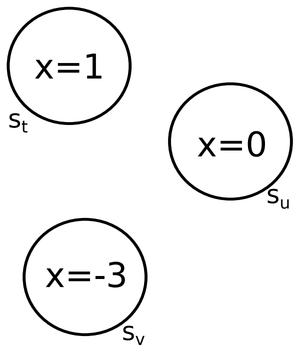

Figure 2.1: Program states of a stateless system

computations [37]. As their name suggests, transition systems define a set of pairs of states called *transitions* that specify correct sequences of program states.

Definition (Transition System) A transition system is a tuple $S = (X, D, dom, In, T)$ where:

1. X is a finite set of state variables,
2. D or domain is a non-empty set of values,
3. dom is a mapping from X to the set of non-empty subsets of D. For each state variable $x \in X$, the set $dom(x)$ is called the domain for x,
4. In is a set of states, called the initial states of S,
5. T is a finite set of transitions.

A state of a transition system S is a function $d : X \to D$ such that for every $x \in X$ and its value $v(x)$, $v(x) \in dom(x)$. A transition is a set of pairs of states. A transition t is applicable to a state s if there exists a state s' such that $(s, s') \in T$. [31]

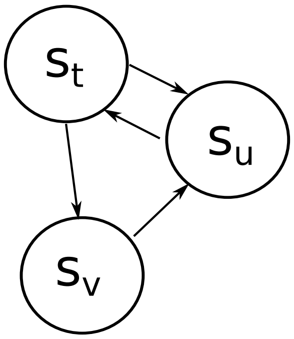

Figure 2.2: Program states of a transition system

From the previous definition it can be observed that a transition (s, s') determines that state s' is one of the valid next program states only if the current program state is s. For example, given a sample system with program states $s_t = \{(x, 1)\}$, $s_u = \{(x, 0)\}$ and $s_v = \{(x, -3)\}$, it is possible to define a potential set of valid transitions as $T = \{(s_t, s_u), (s_t, s_v), (s_u, s_t), (s_v, s_u)\}$. Figure 2.2 depicts that potential set T as a digraph where system states are represented by the graph vertices and transitions by the graph edges.

Usually, transition systems are able to differentiate distinct types of events. This last point is important because transition systems move from the current state to the next state as a consequence of internal or external events of various types. However, not every event always triggers a transition to a different state. In practice, the next program state is a function of the current state and the event type.

2.1.3 State Properties

The state properties of transition systems characterize their set of valid program states by describing the correct values of their state variables and the relationship between them. One way to perform these characterizations is by using propositional formulae whose propositions inquire about the value of the program state variables.

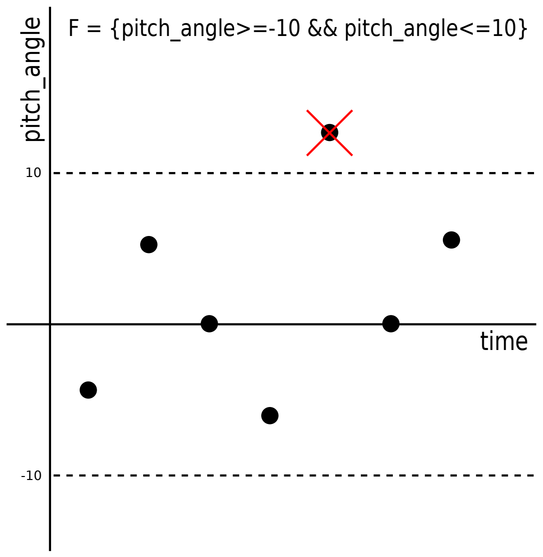

Figure 2.3: State properties representation

This is a practical way, for example, to describe invariants that should hold during the entire program execution. Some examples include defining ranges of state variables ($altitude \geq 0 \wedge altitude \leq 100$), expected values ($temperature \geq 50$, $object \neq null$) and relationships between variables ($door\_open = 1 \wedge altitude = 0$ or $power\_motor1 = power\_motor2$).

A more powerful option consists of defining properties using first-order logic, incorporating the universal quantifier $\forall$, the existential quantifier $\exists$ and quantified variables. These additions make it possible to operate over sets of variables like for example $\forall sensor \in Sensors : sensor.status =$ 'ok' or $\exists battery \in Batteries : battery > 0.20$.

Now, given a formula describing the valid program states we can check whether a state obides to it. For example, given a sample system that has state variable $pitch\_angle \in X$ and state property $s_{valid} = \{pitch\_angle > -10^{\circ} \wedge pitch\_angle <= 10^{\circ}\}$ which characterizes every valid program state, it is possible to define propositional formula $F = \{pitch\_angle >= -10 \wedge pitch\_angle <= 10\}$ to express the set of valid program states and define state property $StateProperty = \{s \in S \mid s \models F\}$. Figure 2.3 illustrates an execution of this system and its corresponding state property. As only one state variable, *pitch_angle*, is present in F, each dot in Figure 2.3 represents the system state (y axis) over time (x axis) and the dashed lines enclose the valid values of *pitch_angle*. The crossed out dot depicts an invalid state or failure as the value of *pitch_angle* is larger than $10°$, which is inconsistent with property $StateProp$.

2.1.4 Temporal Properties

While propositional logic is able to model systems by specifying state properties or valid program states, propositional formulae present some limitations when modeling complex temporal aspects of transition systems or properties that require some reasoning about the ordering of system transitions. Those properties, called temporal properties, can be represented using first-order logic or temporal logics [38]. In practice, both logics can express temporal properties, but quantified variables in terms of time quickly become cumbersome to interpret simple and readable than temporal formulae. Temporal logics are best suited to express temporal properties of transition systems because of their implicit notion of time.

To achieve the required expressiveness, temporal logic extends propositional logic by including several temporal operators. Hence, temporal formulae contain both temporal operators and propositional formulae representing a transition of the system. In the case of *linear temporal logic* (LTL [37]), these temporal operators are $\Box$, $\Diamond$, $\bigcirc$, U, R. The semantics of these operators given formulae A and B and an ordered sequence of states or path $\pi = \{s_0, s_1, s_2 \ldots\}$ where s_0 is the current state, are:

- $\Box$ (always) Temporal formula A hold at all states along path π.

- $\Diamond$ (eventually) Temporal formula A hold at some state on the path π.
- $\bigcirc$ (next) Temporal formula A holds at the next state in π, that is, at s_1.
- U (until) Temporal formula A U B holds if A holds until B occurs and if B does not occu
- R (release) Temporal formula A R B holds if whenever $\neg$B occurs at a state on path π, A occurs before. Or equivalently, either B holds globally on the path, or A occurs before the first state at which B is violated.

To illustrate a simple temporal property, let's consider a sample system with state variable $pitch\_angle \in X$, states $s_{move} = \{(pitch\_angle > 0 \wedge pitch\_angle <= 10) \vee (pitch\_angle >= -10 \wedge pitch\_angle < 0)\}$, $s_{no\_move} = \{pitch\_angle = 0\}$, initial state $In = \{s_{no\_move}\}$, transitions $T = \{\ (s_{move}, s_{no\_move}), (s_{no\_move}, s_{move})\}$ and able to process events of type *ObstacleDetected*. Then, we can define temporal formula $F = \Box(ObstacleDetected \rightarrow \bigcirc s_{no\_move})$ that expresses that it is always the case ($\Box$ operator) that an occurrence of event *ObstacleDetected* implies that the next state ($\bigcirc$ operator) is s_{no_move}. Then, given a sequence of states $\pi = s_0, s_1, s_2 \ldots$ we can define temporal property $TempProp = \{s_i \in \pi \mid s_i \models F\}$.

Figure 2.4 shows this last example where black circles represent the program state and the white circles the occurrence of event *ObstacleDetected*. The solid lines represent the fact that, according to the specified temporal property, an occurrence of *ObstacleDetected* has to be followed by a transition that sets the system state to s_{no_move}. The crossed out circle depicts a violation of the property.

Specifying temporal properties is hard [28]. To address that challenge, researchers have created various mechanisms to facilitate their development. For example, Dwyer et al. [19] have developed a framework of frequently appearing patterns in properties

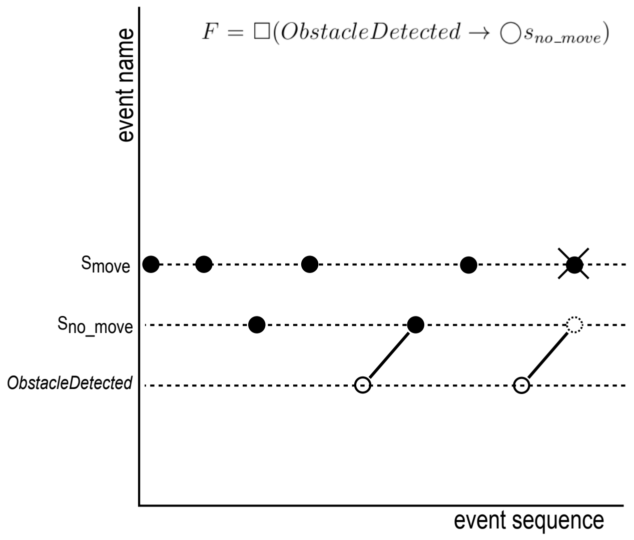
$F = \Box(ObstacleDetected \rightarrow \bigcirc s_{no\_move})$
event name
Smove
Sno_move
ObstacleDetected
event sequence

Figure 2.4: Temporal properties representation

specification focus on the matching of problem characteristics to solution strategies. Another example is the creation of specification languages [14] [33] which provide built-in higher-level operators or abstractions designed to facilitate the specification of temporal properties.

2.1.4.1 Computation Trees and Paths

A practical way to facilitate the analysis of the behavior of transition systems, and consequently their properties, is by constructing their corresponding *computation tree* [20]. A *computation tree* defines the set of all possible executions of a transition system by representing the states s of system S as the tree nodes and the applicable transitions $(s, s') \in T$ as its edges.

Consider the sample system again, we can construct its computation tree as shown in Figure 2.5. This graph allows us to understand that (R, Q), (R, P), (P, R), (Q, R) and (Q, P) are the applicable transitions T of the system. Not only that, it tells us that to reach P we need to go first through R and then through Q.

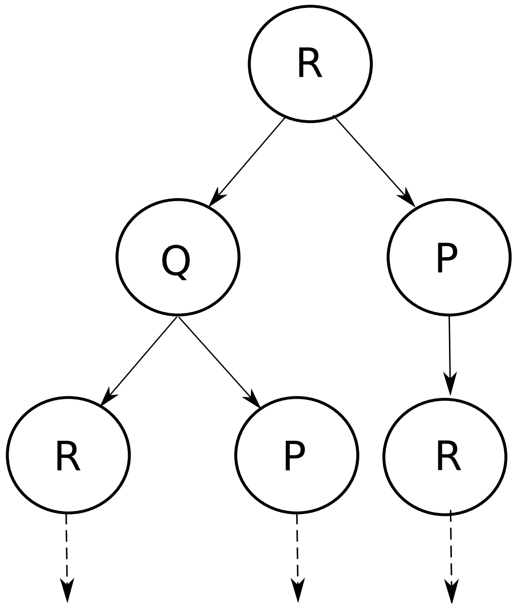

Figure 2.5: A sample computation tree

By constructing the computation tree of a transition system it is also possible to define every computation path of the system. A computation path is a sequence of nodes such that for all i we have $(s_i, s_{i+1}) \in T$. If the sequence is finite then there exists no state s such that $(s_n, s_{root}) \in T$. A computation path is any maximal sequence of states through which a computation may go by applying the transitions. Computation trees are a convenient representation of the possible temporal behaviors of transition systems, since assertions about possible temporal behaviors can be conveniently formulated as properties of computation paths or the states on those paths. In fact, linear temporal formulae express temporal properties of the system computation paths.

Figure 2.6 shows a sample computation path. In this case, the graph illustrates that the sequence of states that the system went through was (R, Q, P, R, Q, P).

While it is true that temporal formulae can define temporal properties of transition systems and computation trees can also be used to verify that a given temporal

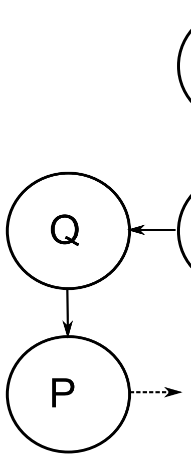

Figure 2.6: A sample computation path

formula holds during the system execution, in practice, a problem called 'state explosion' compromises the efficiency of validating temporal properties. This problem is caused by the fact that computation trees grow exponentially and even small systems can end up building extremely large transition systems. In the case of the most typical transition systems, reactive systems, this situation gets worse given the unpredictable ordering of external events that force the consideration of every single transition. In the systems that we study, such trees contain billions of states.

2.1.5 Probabilistic Temporal Properties

While temporal properties can express conditions that must hold in every or at least one path, they cannot state that a property must hold for a certain portion of the computations, for example, 50% of the system executions. Properties that encode the probability of making a transition between states instead of simply the existence of such a transition are called *Probabilistic Temporal Properties* and can be specified using probabilistic temporal logics such as *Probabilistic Computation Tree Logic* [26] (PCTL), PCTL* [10] or CSL (Continuous Stochastic Logic) [11] [9]. Probabilistic temporal logics usually extend a temporal logic. Therefore, besides the operators these logics define, probabilistic temporal formulae also include state propositions and temporal operators. In the case of PCTL, it also includes timing constraints that bound the occurrence of events. This logic treats time as a discrete unit where each time unit correspond to one transition along the execution path.

Probabilistic temporal properties are used to specify the behavior of real-time distributed systems where probability represents a tool to analyze their performance [42]. Some examples of probabilistic temporal properties are 'with at least 50% probability p will hold within 20 time units ($F^{\leq 20}_{\geq 0.5} p$)' and, 'with at least 99% probability q will hold continuously for 20 time units ($G^{\leq 20}_{\geq 0.99} q$)'. Probabilistic models are specified using *discrete-time Markov chains* (DTMCs) or *Markov decision processes* (MDPs). The first type specifies the probability $\pi(s, s')$ of making a transition from state s to some target state s', where the total probabilities of reaching the target state is 1. The latter type of models extends DTMCs by allowing non-deterministic behavior.

In spite of the more expressiveness power of probabilistic temporal properties, there are theoretical results indicating that the problem of learning transitions probabilities to automatically infer probabilistic temporal formulae is hard [6] [29]. Therefore, researchers have developed specification pattern system of common probabilistic properties to help practitioners formulate these properties correctly [24]. In addition, checking that probabilistic properties are being meet is also a challenging problem because there is no a binary acceptance condition.

2.2 Automatic Inference of Software Properties

Daikon [21] is the most prominent work on automatic inference of state properties. Its authors present it as a dynamic analysis approach to discover likely invariants from program executions. The technique consists of instrumenting the target program to record the values taken on by a set of variables of interest, executing the instrumented program over a test suite, and running an inference engine over the collected traces. The inference engine identifies properties by instantiating a set of possible invariants, testing them against the values captured from the variables of interest, and keeping only those invariants that are never falsified.

Daikon attempts to infer invariants located in what its authors called ‘program points’. By default, these points are procedure entries, procedures exits and loop heads. An instrumenter injects code that records in execution traces the value of every variable in scope when the program point is executed. The inference engine supports several types of invariants over 1, 2 or 3 variables such as constant value, range limits, linear relationship, functions or ordering comparison. The tool also creates ‘derived variables’ like *array[int]*, *sum(array)*, *min(array)*, *max(array)* or *size(array)* which are treated like hard coded variables when invariants are tested.

Programs instrumented by Daikon run an order of magnitude slower [21], the cost of the inference process is hard to predict as it depends on the number of variables in scope in each program point the size of the test suite, and the number of invariants discovered [21].

Diduce [25] is an online dynamic invariant detector and checker capable of identify the root of program crashes. Diduce instruments read/write operations and methods invocation of the target program to derive invariants (*tracked expressions*). Each instrumented program point has a set of associated expressions and a counter keeping track of the executions. Expressions are a function of the object or variable being accessed and invariants are created from this expressions. Diduce does not consider as a fault a single violation of an invariant. To determine faults, the invariant confidence is checked. This metric is the ratio between the number of times that the invariant is accessed and the number of times that it was accepted. Large drops of the coincidence indicate an ongoing violation. The reported overhead of the tool ranges from 8x and reaches 20x.

DySy [18] is a dynamic analysis approach that makes use of dynamic symbolic execution to discover likely invariants. Unlike other approaches, DySy does not falsify invariants produced by predefined templates. Instead, invariants are generated by combining concrete execution of test cases with the path conditions generated by symbolic execution tools. In this way, DySy reduces the size of the test suite required to obtain good invariants. While this technique is able to infer the majority of the interesting Daikon invariants, it does not capture all of them.

Techniques that extract temporal behavior from software systems can be classified into automaton and non-automaton based techniques. The first group of techniques generate finite state automata (FSA) from execution traces [36] [40] [8]. Lo et al. [36] for example presented a technique to steer the automata learner algorithm *kTail* [12] by denying merge operations which produce automata that do not satisfy temporal properties previously inferred.

Non-automaton based techniques instead infer the ordering of the system events or states [35] [41]. Perracotta [41] is an example of non-automaton based technique. It was inspired by Daikon and it uses a dynamic analysis approach for automatically inferring temporal properties of software systems. The instrumenter also injects code at all method entry and exit points and the recorded events consist of the threads identifier, method name, and a location (entering a method or exiting the method). Then, a set of 2 event temporal patterns are instantiated with the monitored event. The distinctive attribute of Perracotta is that it was the first temporal properties inference approach being able to find faults in large programs like JBoss or the Windows kernel. Its down-side was that it was only capable of inferring variations of the *Response* pattern [19].

Javert [22] is a dynamic analysis approach to infer general temporal properties from execution traces. Under the assumption that complex properties can be can formed by composing instances of small generic patterns, this approach specifies arbitrary size temporal properties by concatenating what the authors called *micropatterns*. Gabel it. al [23] presented an automatic dynamic technique for simultaneously learning and enforcing temporal properties over sequences of method calls. This is an on-line approach that operates over a short finite window of trace events and it considers the fact that method calls usually exhibit temporal locality. Under the assumption that common behavior represents correct behavior, its learning and enforcing strategies are tuned on-line by changing parameters, such as the window size, and by analyzing the effect of the change on an objective function that defines the tolerance to anomalies or violations. While this approach has being shown to be useful in finding defects and code-smells related to the wrong usage of APIs, it incurs a significant amount of overhead. Like Perracotta, this technique ignores the parameters of the methods invocations, losing information that encodes part of the system behavior.

2.3 Run-time Verification of Software Properties

The aim of run-time verification techniques is to check at run-time that software systems work in accordance to their specifications [16]. Run-time monitors are automatically generated pieces of code that check program executions against their specification. Monitors can be embedded in the program or operated remotely by receiving a stream of data with run-time information.

The work on run-time verification undertakes two key challenges: how to specify different types of properties for monitoring, and how to encode them to monitor them efficiently. We discuss some attempts that have received considerable attention.

MOP [16], monitoring-oriented programming, allows the specification of properties using different formalisms such as design-by-contracts approaches (JML [34]), temporal properties, and extended regular expressions, and generates monitors from the specified properties. The framework integrates those run-monitors into the target program together with custom code to handle violations to the specified properties. JavaMop [15] takes advantage of the MOP framework by transforming properties specified using MOP into AspectJ [30] aspects. Those aspects are synthesized into the target program and they will act as the run-time monitors that check the program specification. The different mechanisms to specify properties allows the verification of both state and temporal properties.

A different approach is used in JPaX [27] to monitor temporal properties. In order to allow the specification of complex behaviors, JPaX supports properties defined using custom logics developed with Maude [17]. This tool also instruments the target program, but instead of synthesizing monitors into the code, events are extracted from the running program and sent to an observer which decides whether requirements are violated or not.

Allan et al. [7] introduced the notion of *tracematches*, a run-time monitoring approach able to detect temporal properties of AspectJ join-points executions or patterns in the behavior of $freevariables$ whose values are bound to the AspectJ pointcuts of the program under analysis. *Free variables* allow *tracematches* to share a history of their values, so monitors are able to find patters in that history of values.

Bodden et al. [13] investigated the use of *Dependency State Machines* to facilitate the specification of typestate properties in a flow-sensitive manner. State machines are generally easier to understand for developers with some knowledge of state machines. Several tools [7] [15] [32] are able to automatically convert *Dependency State Machines* into flow-sensitive run-time monitors aware of the order in which events have to occur.

Chapter 3

Statistical Properties

This chapter moves forward by expanding the definition of statistical properties and detailing the types of statistical properties that we have developed. Through this chapter we also explain our approach to infer and monitor statistical properties and introduce a tool that implements the proposed approach.

3.1 Definition

Just like others types of software properties, the goal of statistical properties is to model the expected behavior of the system under analysis. Once a model is created, it can be used in many different ways. However, our main target is to check if the modeled system is behaving as expected by, for example, automatically generating monitors that can be checked at run-time.

Informally, we define a statistical property as a significant relationship computed over the values of some variables across program states. In other words, given a set of state variables and a window containing their current and previous values, a statistical property dictates that it is always the case that a significant statistical relationship exists between those values. The significance of the relationship is determined using statistical inference techniques that draw propositions of the collected values. The term *significant* denotes that the relationship between the values of the state variables does not exist by chance for a determined confidence level.

Four elements should be distinguished from the previous definition as they apply to every statistical property: a statistical relationship, a set of involved state variables, a window size, and a significance level. Any statistical relationship can be used to define statistical properties as long as its significance can be determined. A relationship is implemented through a statistical function that maps a set of collected variable values to a single statistic. Depending on whether the relationship is over single or multiple states variables, different statistical functions can be employed. For example, two state variables should be considered if the property aims to capture the existence of a correlation relationship, while only one is required if the property is checking that the variable belongs to a given distribution. The window size dictates how many previous values of the involved state variables are required to evaluate the statistical relationship. Finally, the significance level is used to determine if the relationship exists by chance.

Given this intuition of the definition of statistical properties, a more formal version is provided next.

Definition (Statistical Property)

Given program S and its set of state variables X where:

1. y is a subset of the state variables X,
2. ws is the window size,
3. s_n is the current state of S,

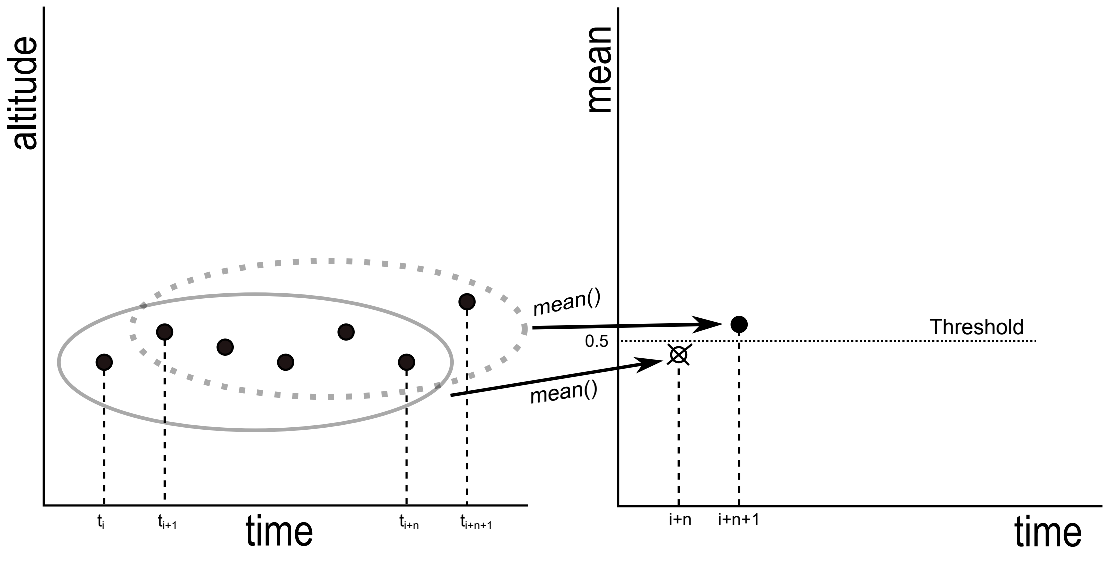

Figure 3.1: Conceptual Representation of Statistical Property over Single Variable

4. $\vec{y}_{ws} = \{y_{n-ws} \dots y_n\} \mid y_i \in s_i$,

5. $stat$ is a statistical function that operates over $\vec{y}_{ws}$ and returns a statistic

6. α is the significance level

7. $th = f(stat, ws, \alpha)$ is a threshold value, and

8. $F = \{stat(\vec{y}_{ws}) \geq th\}$

then $StatisticalProp$ is defined as $\{(s_{n-ws} \dots s_n) \models F\}$.

Figure 3.1 depicts how a statistical property over a single variable would be checked in practice. The property is defined over a single state variable called 'altitude', the used statistical function is 'mean', the windows size is 6 and the threshold value is 0.5. For example, the first observation in the right-hard graph is derived by computing the mean across the six values of variable altitude depicted in the left-hand graph from time t_i to t_{i+n}. The crossed out dot denotes a violation of the property because the statistical relationship is not significant for the first six elements as it is below the specified threshold, while the black dot indicates that the property was met for the next group of values of the state variable (dashed ellipse) containing observations t_{i+1} to t_{i+n+1}.

Given the space of properties defined by state and temporal properties, statistical properties seem to reside in between as they share characteristics with both of them. On the one hand, like state properties, statistical properties take into account the concrete value of state variables and they could be inferred efficiently using automatic techniques.

On the other hand, like temporal properties, the information provided by multiple program states can be used to check deeper aspects of the behavior of the system under analysis. They also share the ability to model systems composed of multiple components that communicate by sequences of messages. In general, whenever temporal properties can be applied, we conjecture that statistical properties can be applied as well.

Statistical properties also offer distinctive characteristics. One advantage is the possibility of tuning their overhead and strength by adjusting the significance level. However, there is a trade-off between overhead and property strength. Using lower significance levels will lead to the inference of weaker properties that require smaller window sizes. Typically, a small window size is desirable as it represents less memory consumption to store historical values and also a faster computation of the statistical relationships. However, inferring weaker properties means greater chances for an inferred property to be false. The opposite happens with high significance levels. Stronger relationships are inferred, with lower false positives, but larger window sizes are required to ensure their significance.

The following section explains how those concepts are taken into practice.

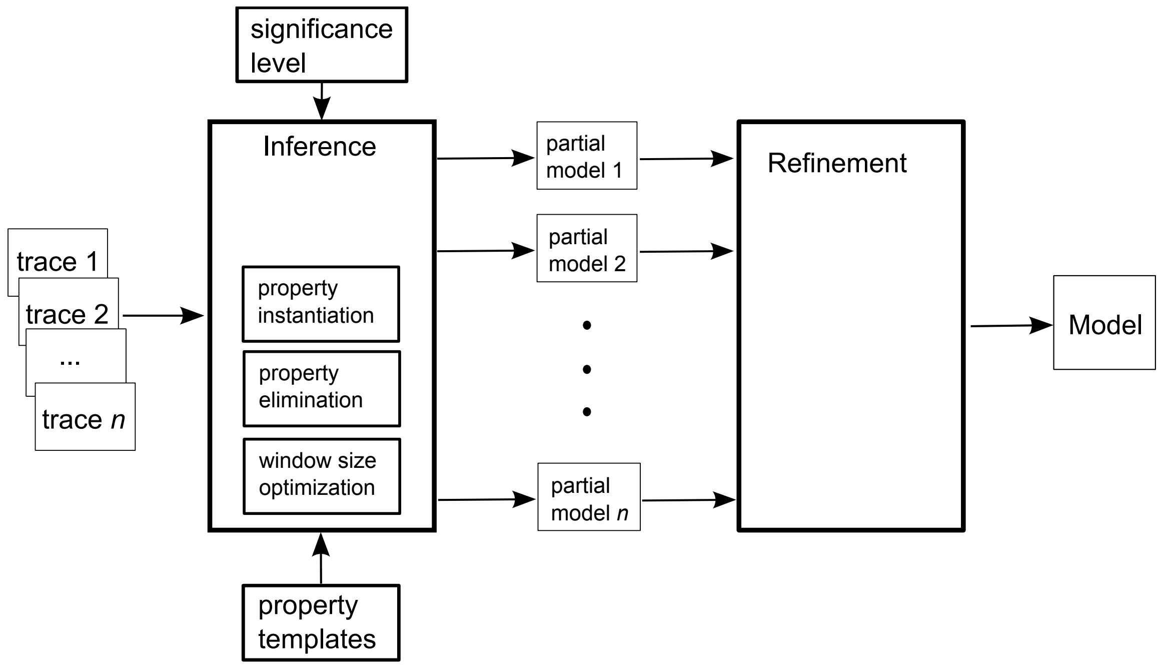

Figure 3.2: Inference Approach

3.2 Approaches for Inferring and Monitoring

Our approaches to infer and monitor statistical properties are illustrated in Figures 3.2 and 3.3. As depicted in Figure 3.2, the input of the inference process are execution traces, the desired significance level and a set of properties templates. Our approach assumes that each line of an execution trace is an event and the parameters of the events correspond to the state variables of the system. The figure shows two components. From each trace, the first component outputs a partial model that contains a set of inferred statistical properties with at least the significance level specified as an input of the process.

During the first step of the inference phase, *property instantiation*, every possible combination of variables are instantiated as a statistical property and their significance given the maximum windows size is determined. During the *property elimination* step, statistical properties that do not achieve the minimum significance

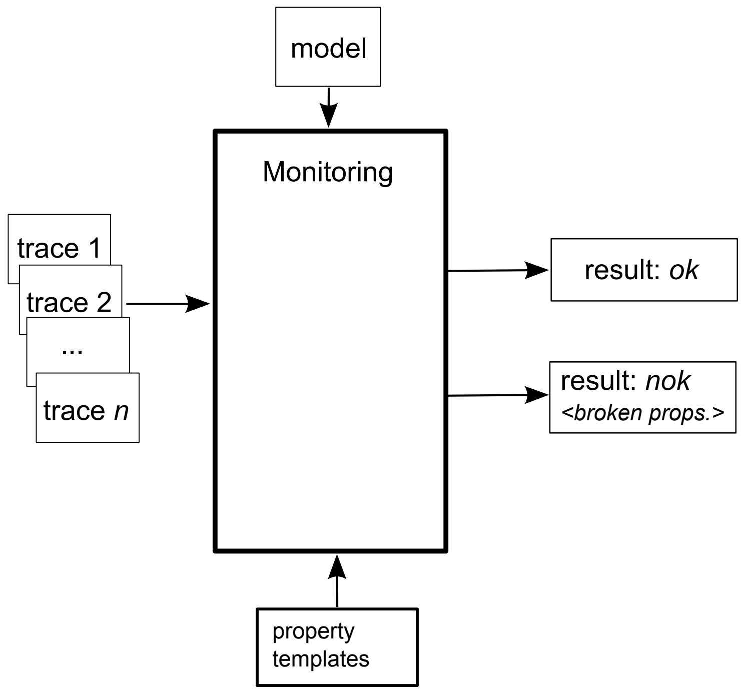

Figure 3.3: Monitoring Approach

level are identified and removed. Finally, the *window size optimization* step performs a binary search algorithm to find, for each statistical property, the smallest window that maintains the required significance and also a lower overhead.

The monitoring phase, illustrated in Figure 3.3, is in charge of checking that an execution trace under analysis respects every statistical property specified in the inferred model and to determine which properties are violated. The property templates are a required input as these templates specify the computation of each property statistic and the statistical test used to determine their significance. While the monitoring approach has been tested off-line, we think that it could be easily adapted to be used on-line by encoding the refined model into a monitor that can be checked at run-time.

While the main goal of this section is to provide a detailed explanation of the dynamic analysis approach that we have developed in order to infer and monitor statistical properties from execution traces, a few concepts are introduced first to facilitate their understanding. Firstly, the characteristics of the statistical relationships supported by our approach are listed followed by the statistical properties that we have already developed. After that, we explain how the concept of *aggregate state* helped us to deal with different update frequencies of the state variables.

3.2.1 Statistical Properties Templates

A property template should include the maximum window size, the minimum window size, the required conditions to update its aggregate state, a procedure to measure the statistical relationship and a procedure to check the significance of the relationship.

As we have mentioned in Section 3.1, any conclusion about the values of the target state variables that can be statistically tested is a candidate statistical relationship for our approach. In other words, it should be possible to determine if the relationship exists by chance or not. Some examples of potentially interesting relationships include: analysis of means, analysis of variances, correlation analysis, and covariance analysis, or determining if a given variable or set of variables belong to a determined distribution.

We have developed and evaluated two statistical relationships: *correlation* and *mean*. As we will show, our choice was meant to capture some of the properties we have informally observed in the type of robotics systems we were analyzing. The statistical relationship established between variables determines the name of the statistical property. For example, we use the term *correlation* property to refer to any statistical property that infers a correlation relationship between two state variables. The specifics of *correlation* and *mean* properties are detailed next.

3.2.1.1 Spearman's Rank Correlation Coefficient

Correlation coefficients can describe many behaviors of the robotics systems which are the target of our experiments. An example of correlated variables is *pitchangle* and the *velocity* of a quad-rotor. If the pitch angle of a quad-rotor increases, then its velocity increases as well.

There exist a number of correlation coefficients that measure the statistical dependence between two variables. This basically means that they can determine the likelihood that the values of two variables are ruled by a monotonic relationship. So, if one variable increases its value whenever the other increases its value, then the coefficient is positive. If one variable decreases its value whenever the other increases its value, then the coefficient is negative. In particular, we selected the Spearman's Rank correlation coefficient [39] because it is a non-parametric method less sensitive to non-normality in distributions. To calculate the Spearman's Rank correlation coefficient, the observations of each variable are ranked in ascending order and the differences between the ranks $diff\_rank$ of each observation on the two variables are calculated. Then, the coefficient is the result of applying the following formula:

$$\text{Correlation Coefficient} = 1 - \frac{6 * \sum \text{diff rank}^2}{\text{window size} * (\text{window size}^2 - 1)} \qquad (3.1)$$

To determine the significance of a correlation relationship, the threshold value for a given window and significance level is determined using the critical values of the Spearman's rank correlation [5]. These values indicate the minimum correlation coefficient to reject the null hypothesis that no correlation exists between the two variables. Table A.1 shows the critical values of the Spearman's rank correlation. For example, given a window size of 10 (row) and a significance level (alpha) of 0.05 (column), the minimum correlation coefficient to reject the null hypothesis that there is no correlation between the variables is 0.564. So, if the coefficient is higher than 0.564, the relationship is significant.

3.2.1.2 Mean

This property determines if, given the mean and standard deviation of a window of values of a single variable, the next value of the variable is within the confidence interval for the mean determined by the significance level [39]. In the context of robotics applications, this property could capture the behavior of many sensors. For example, range finders may return different readings even when the robot does not move. However, those readings should have similar values and are a potential target of the statistical relationship of the *mean* property. If later, during the monitoring phase, a value of the state variable is out of the confidence interval, it can be said that a fault or, at least, an unknown behavior was detected.

3.2.1.3 Setting Maximum and Minimum Windows

Besides defining the statistical relationship, our approach requires new property templates that define the maximum and minimum window size they are willing to deal with. Recall that a small minimum window size is desirable as it reduces the cost of calculating the statistical relationship. However, the minimum window size has to be as big as the minimum number of observations required by the statistical test to calculate the significance of the relationship. Decreasing the maximum window size can be used to adjust the overhead of our approach in several ways. First, the inference approach converges faster with a low difference between maximum and minimum window size. As we performs a binary search to find the optimal window size, less elements means less steps until the optimal is found. Second, weak properties are going to be discarded faster because smaller windows increase the value required by the significance test to reject the null hypothesis that no relationship exists between the variables. Third, the calculation of the statistics is also faster. Less data means less operations. However, choosing a maximum window size depends on each property.

3.2.1.4 Update Policies

Consider two variables *commanded_pitch* and *pitch* that are supposed to be correlated but whose update frequency is 10Hz and 1Hz respectively. Let's suppose a window size of 10 and a system where only the last value of *commanded_pitch* affects the value of *pitch*. Then, when *pitch* is changed by the 10th time, a real relationship between the window of *commanded_pitch* and *pitch* will not be detected, even if it really exists, as the windows of *commanded_pitch* and *pitch* contain values that may have been generated 10 times apart.

To accommodate such potential inconsistencies, we have developed the concept of *aggregate state*. This is an abstract state that stores just the relevant values, those that are going to be used to determine if a significant statistical relationship exists. Every statistical property has an aggregate state and each aggregate state stores a window of values per state variable involved with the statistical property. To determine what values are relevant, each statistical function must define a policy that specifies what values to add to the aggregate state. These policies are defined as predicates over the occurrence of events that update the value of the involved variables. Each property template has to indicate under what conditions the current value of the involved state variables are stored in the aggregate state.

Figure 3.4 provides a depiction of what state variables values are stored in the *aggregate state* of *correlation template properties*. This figure shows the two variables involved in the property, *a* and *b*, and their changes over the system execution. Each circle represents a program state and the suffixes represent the order in which they

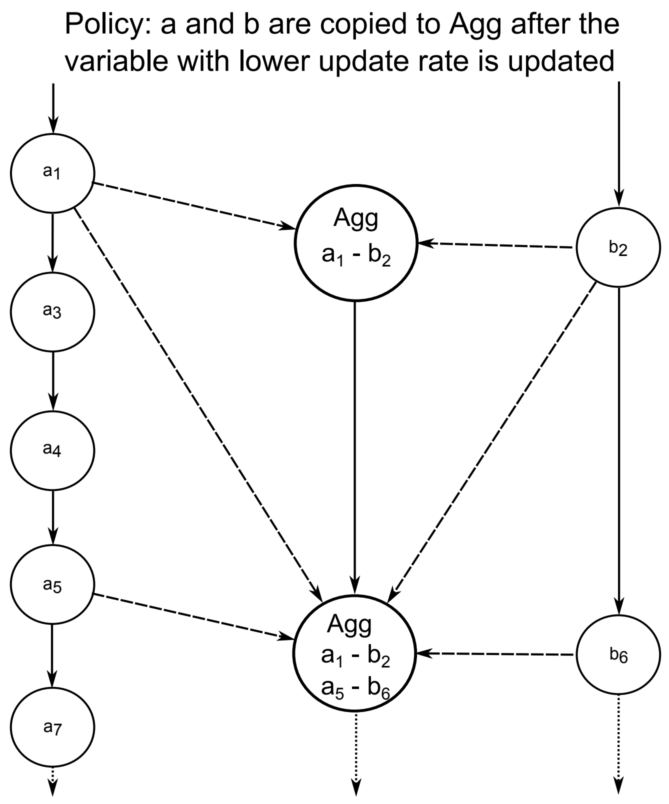
Policy: a and b are copied to Agg after the variable with lower update rate is updated
a1
a3
a4
a5
a7
Agg a1 - b2
Agg a1 - b2 a5 - b6
b2
b6

Figure 3.4: Aggregate state representation for correlation properties

occur. As seen in the figure, the current values of a and b are copied to the *aggregate state* only after both variables are updated.

The case of *mean properties* is different because the *aggregate state* is updated whenever the single state variable is updated, so no condition is set.

Every time, after all the new values are added to the *aggregate state*, as per the property policy, the significance of the statistical relationship associated to the statistical property is checked. So, if new property types are to be added to the inference process, those properties have to define the conditions under which the *aggregate state* is updated and the significance test that will check that the required significance is achieved.

3.2.2 Inference

The goal of the inference phase (Figure 3.2) is not only to identify statistical properties but also to determine the smallest window size that minimizes overhead. From a high level perspective, the way that our approach fulfills these objectives is by identifying every state variable in a trace, creating every possible statistical property from the set of state variables and the properties templates, and running an iterative process that evaluates the significance of the candidate statistical properties until all non-significant statistical properties are discarded and the optimal window size of the significant ones is found. The inference process is listed in Algorithm 1.

The inputs to this process are an execution trace, the significance level of the statistical properties and a set of statistical properties templates. The output of the inference process is the list of inferred statistical properties or model. For each inferred property, the model specifies the property type, state variables involved, window size and significance level.

The first step of the inference approach is to identify all the state variables of the program. Procedure $RetrieveStateVariables$ (line 3 of Algorithm 1) identifies the set of state variables by iterating over the trace events and storing their parameter names. Then, procedure $CreateStatisticalProp$ creates every possible statistical property from the set of state variables and properties templates (line 4 of Algorithm 1). The number of statistical properties created for each template is the k-combination of the state variables set, where k is the number of variables required by the template's statistical function. Procedure $CreateStatisticalProp$ also creates the *aggregate state* of each property and sets their window size to its maximum.

After that, an iterative process (lines 5-27) is launched to discard statistical properties that are not significant even when their maximum window is set and to op-

Algorithm 1 Inference Phase

```
Require: trace, property_templates, significance_level
 1: $first\_iteration$ = true
 2: $properties\_optimized$ = false
 3: state_variables = RetrieveStateVariables(trace)
 4: properties = CreateStatisticalProp(state_variables, property_templates)
 5: while $\neg properties\_optimized$ do
 6:   $properties\_optimized$ = true
 7:   ProcessTrace(trace, properties, significance_level)
 8:   for all property$_i$ in properties do
 9:     if $\neg$property$_i.significant$ then
10:       if $first\_iteration$ then
11:         properties = properties - property$_i$
12:         continue
13:       else
14:         property$_i.min\_window$ = property$_i.window$
15:       end if
16:     else
17:       property$_i.max\_window$ = property$_i.window$
18:     end if
19:     property$_i.window$ = (property$_i.max\_window$ + property$_i.min\_window$) / 2
20:     if property$_i.max\_window$ $\leq$ property$_i.min\_window$ then
21:       $properties\_optimized$ = false
22:     else
23:       property$_i.optimized$ = true
24:     end if
25:   end for
26:   $first\_iteration$ = false
27: end while
28: return properties
```

timize the window size of the rest. The statistical properties elimination process is performed only after the first iteration is completed. Procedure *ProcessTrace* (line 7) checks which of the properties created in line 4 are significant. The result of that operation is stored in the attribute *significant* of every property. After the first iteration, *ProcessTrace* will deem only the subset of instantiated significant properties. *ProcessTrace* is listed in Algorithm 2 and explained later.

After the *ProcessTrace* procedure is executed for the first time, any non-significant statistical property is discarded (line 11) and never used again. The remaining properties move to the window optimization process.

In order to determine the optimal window size for the surviving statistical properties, we use a binary-search-like algorithm (line 9 - 19). This optimization process starts after each iteration is completed. At that point, if the statistical property to optimize is significant, a reduction of the property window size is attempted. Otherwise, the window size is increased. To reduce the window size, max_window is set to the current window size and then the window size is set to $(max\_window + min\_window)/2$. To increase the window size, min_window is set to the current window size and then the window size is set to $(max\_window + min\_window)/2$.

The optimization process continues until $max\_window > min\_window$ in every statistical property (lines 21) as it indicates that the binary search algorithm is over. Otherwise, individual properties that reach their convergence point are marked as *optimized* and not processed any more. Once the set of relevant statistical properties is determined and their window sizes optimized, they are grouped together to define a model.

The procedure *ProcessTrace* (Algorithm 2) is in charge of checking, for a given significance level and the current window size of each statistical property in *properties*, if the statistical relationships are significant throughout *trace*. Whenever the statistical relationship of a determined statistical property is not significant, the attribute *significant* is set to *false*.

Algorithm 2 Process Trace Procedure

```
Require: trace, properties, significance_level
 1: for all event_i in trace do
 2:   for all parameter_j in event_i do
 3:     program_state[parameter_j.name] = parameter_j.value
 4:   end for
 5:   for all parameter_j in event_i do
 6:     param_properties = properties.get(parameter_j)
 7:     for all property_k in param_properties do
 8:       if property_k.significant && ¬ property_k.optimized then
 9:         if property_k.updateAggregatedState() then
10:           property_k.significant = property_k.checkSignificance(significance_level)
11:         end if
12:       end if
13:     end for
14:   end for
15: end for
```

To accomplish this, the events in *trace* are processed one by one. For each event, their parameters are retrieved and the program state updated (lines 2-4) as we match each event parameter to a state variable of the system. After that, each state variable or event parameter is processed.

First, the list of statistical properties where parameter$_j$ is an involved variable is retrieved (line 6). For each of those properties, if it was not marked as not significant (line 8) and it was not marked as optimized yet, then the conditions to update the *aggregate state* are checked (line 9). If affirmative, *checkSignificance(significance_level)* is invoked. This procedure, specified by each property template, evaluates the statistical relationship between the involved variables and returns *true* if the statistical relationship is significant or $false$ otherwise (line 10).

The complexity of procedure $ProcessTrace$ depends on the templates used in practice. Its template-independent complexity is $\mathcal{O}(ppsum * check)$, where *check* represents the complexity of procedure $template.checkSignificance(significance\_level)$ and *ppsum* is the number of times that procedure is invoked. The value of *ppsum* is calculated as $\sum_{i=1}^{n} \sum_{j=1}^{p} pp_{ij}$, where n is the number of events in *trace*, p is the number of parameters in n_i and pp_{ij} is the number of statistical properties where p_{ij} is involved. Therefore, in the case of the correlation template, as the complexity of calculating the Spearman's correlation coefficient is $ws * \log \mathrm{ws}$ [39], where ws is the window size, then the complexity of procedure $ProcessTrace$ is $\mathcal{O}(ppsum * ws * \log \mathrm{ws})$. If we consider the mean template, the complexity of procedure $ProcessTrace$ is $\mathcal{O}(ppsum * ws)$. The complexity of procedure *template.updateAggregatedState*() is constant and it is not considered in the computation of $ProcessTrace$'s complexity.

The complexity of the inference process also depends on the used templates. If we consider the correlation template, then the complexity of inferring statistical properties is $\mathcal{O}(\log(max - min) * (ws * \log \mathrm{ws} + \binom{vars}{2}))$, where max and min are the maximum and minimum window sizes defined by *template*, $vars$ the number of state properties in *trace*, $\binom{vars}{2}$ the number of statistical properties instantiated by procedure *CreateStatisticalProp*(), and $ws * \log \mathrm{ws}$ is the complexity of procedure $ProcessTrace()$. As our optimization process always lead to the worst case of the binary search algorithm, the $\log(max - min)$ factor represents that situation.

Our approach to filter out wrongly inferred statistical properties, listed in Algorithm 3, that could be encountered in some executions traces by coincidence, consists in the creation of a 'refined' model that contains only the properties found in every single model (line 11).

Algorithm 3 Model Refinement

```
Require: models
 1: refined_model = $\text{model}_1$;
 2: for all $\text{model}_j$ in models do
 3:   for all $\text{property}_i$ in $\text{model}_j$ do
 4:     if refined_model.$has(\text{property}_i)$ then
 5:       $new\_window = \text{property}_i.window$
 6:       $cur\_window = \text{refined\_model}.get(\text{property}_i).window$
 7:       if $new\_window > cur\_window$ then
 8:         refined_model.$put(\text{property}_i, new\_window)$
 9:       end if
10:     else
11:       refined_model.$delete(\text{property}_i)$
12:     end if
13:   end for
14: end for
15: return refined_model
```

It could be also the case that a statistical property is present in every model of the set but its window size is not the same across them. In those situations, the larger window size is kept (lines 5-11). Increasing the window size also increases the monitoring overhead of the property and makes it easier to achieve the desired significance as significance tests lower the threshold to which the relationship is compared against when the number of observations increase.

The complexity of the model refinement process is $\mathcal{O}(modprop)$, where $modprop$ is the summation of the number of properties in every model that belongs to $models$. Therefore, $modprop$ can be calculated as $\sum_{i=1}^{m}\sum_{j=1}^{p} 1$, where m is the number of considered models and p the number of properties in model m_i.

3.2.3 Monitoring

Our approach to monitor that a given execution trace does not violate the inferred statistical properties, Figure 3.3, is performed off-line. Algorithm 4 lists our approach.

The inputs of this process are the execution trace capturing the run-time behavior to monitor, the refined model created during the inference phase, and the properties templates.

The monitoring phase starts by creating monitors only for the statistical properties that were inferred (line 2) and then the optimal window size for each property, as specified in the model (lines 3-5). The monitors use the statistical function, significance test and update policy to compute the required statistics and verify that the significance level remains above the threshold. After that, the procedure *ProcessTrace* is invoked to check that the statistical relationship of each property is significant throughout the entire trace (line 5). In this case, only one iteration is needed. Finally, an error message is printed for every violated property.

Algorithm 4 Off-line Monitoring Phase

```
Require: model, trace, property_templates, significance_level
1: state_variables = RetrieveStateVariables(trace)
2: monitors = CreateStatisticalMonitor(model, state_variables)
3: for all $monitor_i$ in monitors do
4:     $monitor_i.window$ = model.$get$($monitor_i$).$window$
5: end for
6: ProcessTrace(trace, monitors, significance_level)
7: for all $monitor_i$ in monitors do
8:     if $monitor_i.significant$ == false then
9:         print $monitor_i$ violated
10:    end if
11: end for
```

Like the inference process, the complexity of the monitoring process depends on the used templates. In the case of the correlation template, the complexity of monitoring is $\mathcal{O}(monitors + (ws * \log \mathrm{ws}))$, where *monitors* is the number of properties defined by *model* and $(ws * \log \mathrm{ws})$ is the complexity of procedure *ProcessTrace*().

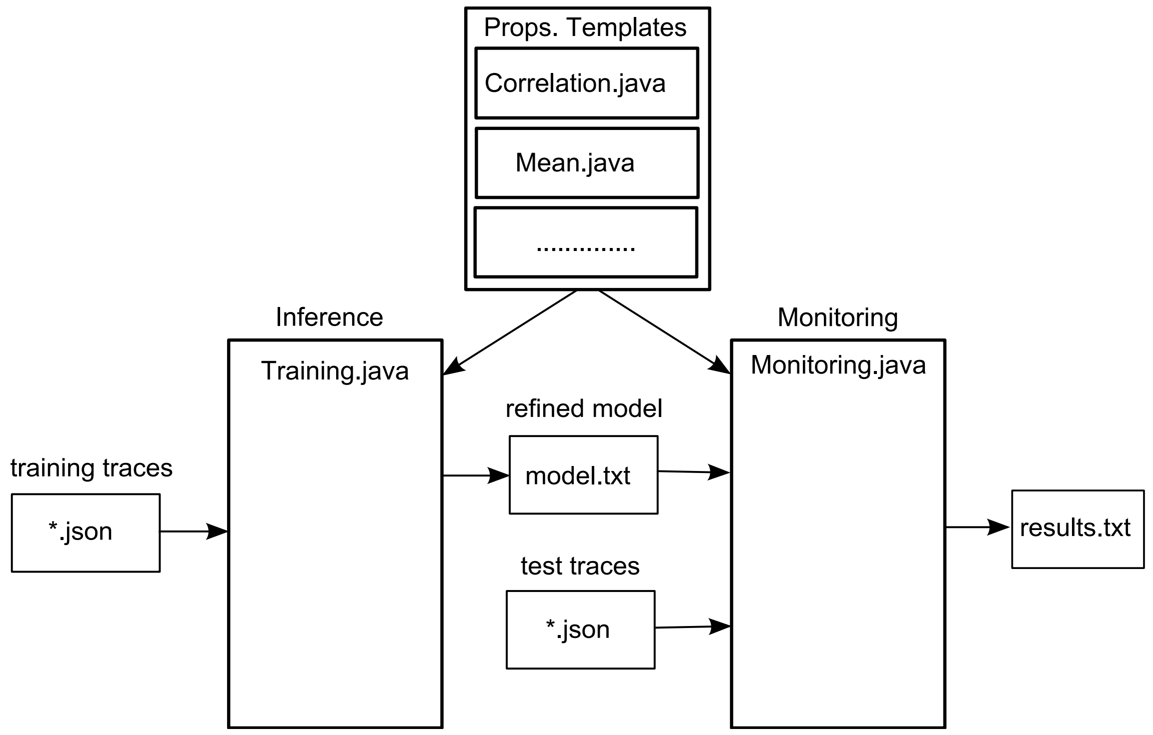

Figure 3.5: Architecture of Implementation

3.3 Tool

We have developed an infrastructure, using 1826 lines of Java code, in order to illustrate the effectiveness of our approach. Figure 3.5 shows the architecture that implements the inference and monitoring phases, and the mechanism we used to interface with them. As we already discussed these phases before, this section explains how these tool can be extended by adding new statistical properties and the format of the trace and model files.

3.3.1 Statistical Properties

In Figure 3.5, the box containing classes *Correlation* and *Mean* depict the two statistical properties that we have implemented. New statistical properties can be added

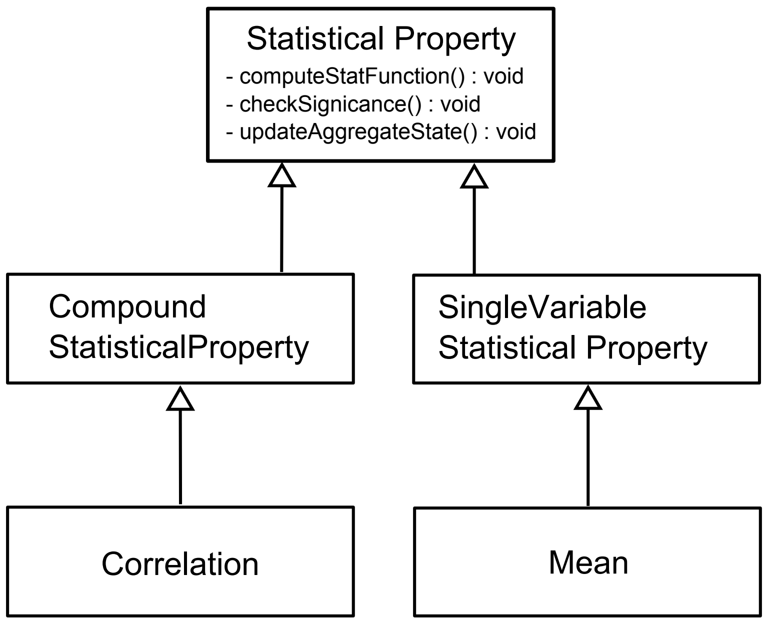
Statistical Property
- computeStatFunction() : void
- checkSignicance() : void
- updateAggregateState() : void
Compound StatisticalProperty
SingleVariable Statistical Property
Correlation
Mean

Figure 3.6: Statistical Properties Hierarchy

though a set of abstract classes that includes three core methods: *computeStatFunction*, *checkSignificance* and *updateAggregateState*. Custom classes should implement those methods. Method *computeStatFunction* calculates and returns the value of the statistical relationship under test. For example, in the case of the *correlation* property, this method return the Spearman's rank correlation coefficient. Method *checkSignificance* should determine the significance of the current value of the statistical relationship for the current window size and significance level and returns *true* if it is significant. Otherwise, it should return *false*. Finally, method *updateAggregateState* should return *true* if the conditions to update the *aggregated state* are met.

Figure 3.6 shows that class hierarchy of statistical properties that model the behavior of one variable should extend *SingleVariableStatisticalProperty* and statistical properties that model the relationship between two or more variables should extend *CompoundStatisticalProperty*. Classes *StatisticalProperty*, *SingleVariableStatistical-*

Property and *CompoundStatisticalProperty* take the responsibility of optimizing the window size, updating the aggregated state and invoking the methods implemented by custom statistical properties when necessary.

3.3.2 Execution Traces and Models Format

The content of the execution traces should be specified using JSON [3]. We selected JSON because is a popular light-weigh format and the high availability of libraries that read and create JSON files. This can simplify the process of consuming execution traces from third parties.

Listing 3.1 shows a JSON trace example where each line represents an event. The first element of each line is a numerical identifier used to distinguish each event and to define the ordering in which they are processed. The next element is the event type and, lastly, the event parameters are specified.

To define this format, we followed the conventions that JSON dictates. In a glance, JSON files are a collection of coma separated key:value pairs where strings are quoted, colons separate keys and values and brackets indicate nested key:value pairs.

Listing 3.1: JSON execution trace

```
{
'1': { '/motorSpeedLeft': {'data':0.35} },
'2': { '/motorSpeedRight': {'data':0.346595} },
'3': { '/leftRanger': {'data':308.0} },
'4': { '/rightRanger': {'data':0.0} },
'5': { '/rightRangerAvg': {'data':2.52380952381} },
'6': { '/leftRangerAvg': {'data':307.55} }
}
```

Models specify the inferred properties by enumerating their property type, state variables involved, window size and significance level as a set of comma separated values. Listing 3.2 shows two sample properties. The first line indicates a *correlation* relationship between the parameter *data* of the event *motorSpeedLeft* and the parameter *data* of the event *leftRander*, where the inferred optimal window size is 13 for a significance level of 0.05. The second line indicates a *mean* relationship between the values of parameter *data* of event *motorSpeedRight*, where the inferred optimal window size is 8 for a significance level of 0.05. In order to prevent having state variables with the same name, the name of the event is attached to them.

Listing 3.2: Model Format

```
Correlation ,motorSpeedLeft+data ,leftRanger+data ,13 ,0.05
Mean,motorSpeedRight+data ,8 ,0.05
```

3.3.3 Property Violation Report

The final output of the monitoring process is a text file that lists the violated statistical properties. Listing 3.3 show an example where the correlation property between variables *motorSpeedLeft+data* and *motorSpeedRight+data* was violated. The report file name results from the concatenation of the trace file name and string '.res'. After the entire test trace is monitored, some scripts are executed to aggregate the data of these reports. The output of those scripts are 2 ranks. The first one show the most effective property in detecting true negatives and the other the statistical properties responsible for the false positives.

Listing 3.3: Fault Report Format

```
Correlation ,motorSpeedLeft+data ,motorSpeedRight+data
Correlation ,leftRanger+data ,rightRanger+data
```

Chapter 4

Assessment

A set of three experiments have been conducted in order to evaluate if automatically inferred statistical properties are able to effectively characterize the behavior of distributed robotics systems. The effectiveness is determined by measuring the precision and recall of the inferred statistical properties at identifying successful and faulty executions of three artifacts. This study also assesses the effect of different significance levels, training set sizes, and the statistical functions used in the precision and recall of the generated models.

The cost of generating those models and the cost of monitoring them against other execution traces is also presented and the effect of significance, training set sizes, and the statistical functions is analyzed as well.

All studies share the same experimental setup, depicted in Figure 4.1, and the specific details of each experiment are discussed later in this chapter. In the figure, rectangles with thick lines represent processes, and rectangles with thinner lines represent data. The words in italic depicts the manipulated variables. Initially, and in accordance with the capabilities of each artifact, a particular task was chosen and two different scenarios were selected to affect the likelihood that the artifact could

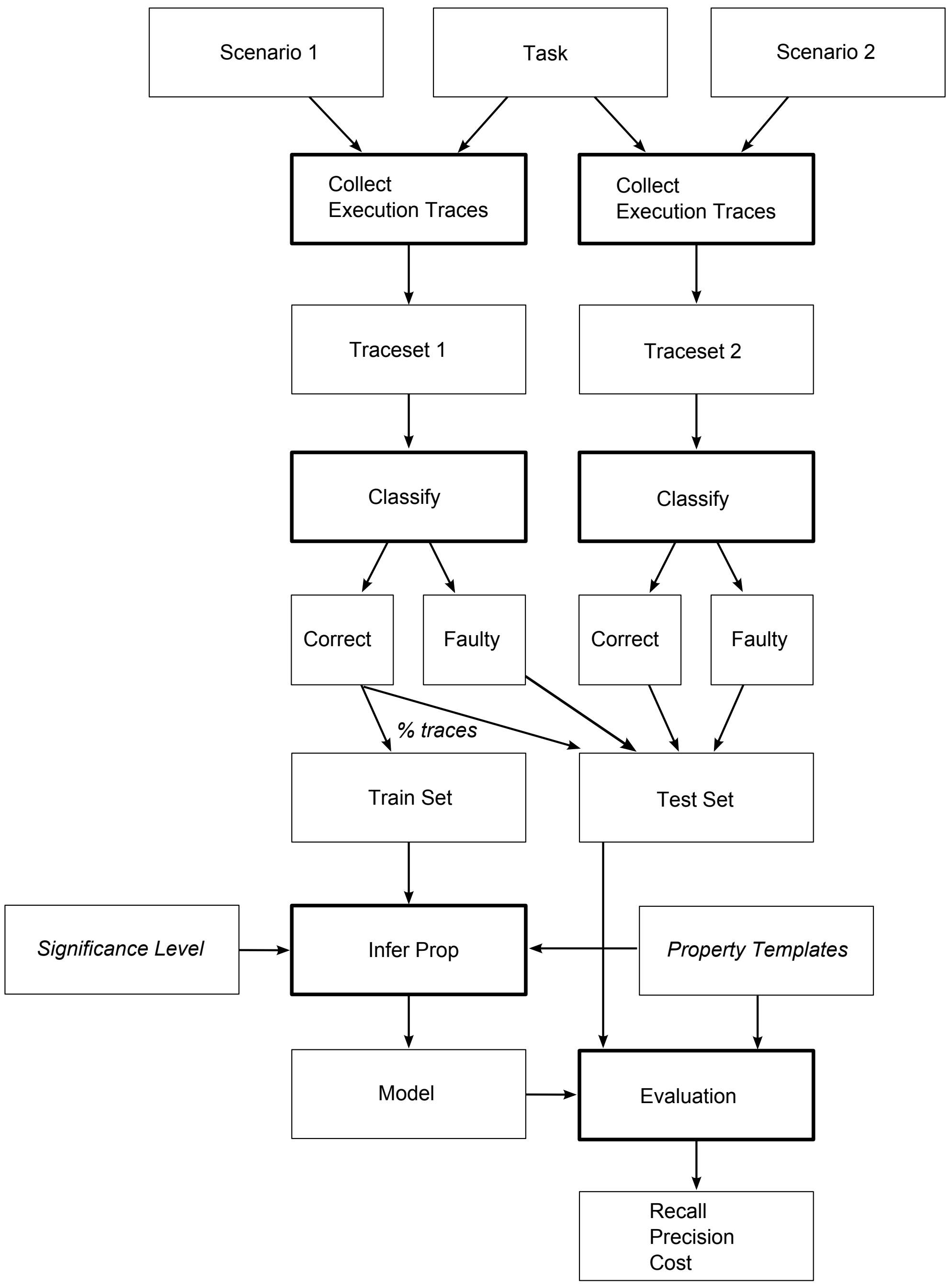

Figure 4.1: Experimental Design

accomplish the task successfully. While the tasks were executed by the artifacts, their execution traces were collected and later classified as successful or faulty according to whether or not the task objective was met. Once the execution traces of each artifact were collected and classified, a randomly selected subset of the successful executions was analyzed to automatically infer statistical properties of the artifact. Finally, the generated model was used to determine if the remaining successful traces were accepted by the derived properties and if the faulty traces broke the properties in a model. The time required to infer and monitor properties was collected to determine the cost of each phase.

The rest of this chapter is organized as follows. First, the three artifacts utilized in these experiments are described. Later, the experimental setup is further described along with the execution trace collection process. Finally, the results of the training and test phases are presented and analyzed, including a description of the generated models and a discussion of the inferred properties violated by the test traces.

4.1 Artifacts, Tasks and Scenarios

This section briefly introduces the distributed robotics systems used to evaluate the proposed approach, including both the hardware and software components that make the artifacts. Also the experiments' tasks and their outcome classification criteria are explained.

On the hardware side, the three artifacts used in this study are existing robotics platforms. These robotic platforms were acquired from two well-known robots vendors that design robots for researchers who focus on using them, not on building them. In this way, efforts could be concentrated on developing algorithms, controllers and new behaviors that extend the platforms' basic functionality. This is possible

Task	LOC	#Comp.	#Events	#Vars.
Wall following	1497	2	7	15
Ranger height controller	6006	16	23	134
Vision height controller	8387	17	15	98

Table 4.1: Experiments Artifacts

as both platforms provide mechanisms to retrieve sensory data and to control the robot actuators. These robotics platforms are: the Garcia Robot [1] (Figure 4.2), manufactured by Acroname Inc., and the AscTec Hummingbird [2] (Figure 1.1), from Ascending Technologies which is used in two of the three experiments.

All the software components of the artifacts were developed by researchers and graduate students of the University of Nebraska-Lincoln, as part of their daily work, who kindly let us use them to accomplish these experiments. These components were the artifacts under the analysis of our approach and provide the additional behaviors, which are not provided out-of-the-box by these robotics platforms, needed to accomplish the assigned tasks. We were not involved in the development of any of these components and no changes were performed to run the experiments.

The software components consist of a collection of distributed ROS-based components developed in C++. ROS [4], Robot Operating System, is a popular open source meta-operating system. It provides a rich library of features ranging from low-level device drivers to commonly used high-level functionality, and it wraps many popular open source libraries used by roboticists. It also provides a publish-subscribe architecture for software modules to communicate through well-defined messages, and a service construct for synchronous communication. This architecture hides the complexity of component communication which is realized through a name-server process.

From the point of view of our approach, we have considered that messages sent by ROS components are events that may cause a change in the system state and the value of the messages parameters are the system state variables and their values. We made this decision under the assumption that the information passed through messages by the system components provides an accurate overall representation of the systems behavior. Clearly, the variables of each component may provide additional information, but we conjectured that the key information eventually becomes part of messages to other components. Besides, considering internal variables increases the cost of the approach.

Table 4.1 summarizes the artifacts under analysis. The table lists their lines of code, the number of ROS components that conform the system, the number of messages types, and the total number of variables across all the messages. Table 4.2 shows the artifacts, the tasks, the scenarios, assertions used to classify their traces. More details are provided next.

We classify the executions programmatically through assertions in order to reduce experimental noise and costs. The assertions we selected were meant to coarsely mimic human observer who classifies executions as successful or faulty. Note that in practice, however, more sophisticated mechanisms would be needed to classify the system behavior.

4.1.1 Garcia Robot - Wall Following

The first artifact is built on top of the Garcia Robot, a ground customizable robot which has a set of base configurations available to facilitate the customization experience. The offered configurations differ in the type and quantity of included sensors. In particular, the robot used during our study is equipped with 6 IR range finders to measure the proximity to occasional obstacles, 4 servo motors to control the wheels and gripper, and wireless and serial communication interfaces.

Platform	**Task**	**Scenarios**	**Assertions**
Garcia	Wall following	Straight wall	$\lvert d_{current} - d_{target} \rvert < 8cm$
		Angled wall	
Hummingbird	Ranger height controller	Passive pilot	$\lvert h_{current} - h_{target} \rvert < 15cm$ $\lvert h_t - h_{t+0.5sec} \rvert < 5cm$ $\sum \lvert h_{target} - h_{current} \rvert < 5cm$
		Aggressive pilot	
Hummingbird	Vision height controller	Outdoors	$\lvert h_{current} - h_{target} \rvert < 15cm$ $\lvert h_t - h_{t+0.5sec} \rvert < 5cm$ $\sum \lvert h_{target} - h_{current} \rvert < 5cm$
		Indoors	

Table 4.2: Experiments Tasks

During the first experiment, the *wall following* task is assigned to the Garcia Robot. For this task, given points A and B in a wooden wall, the robot must move from point A to point B at a distance of 20 cm to the wall. A successful execution of this task is one in which the robot never deviates more than 8 centimeters from the target distance. Otherwise, the execution is considered as faulty. The maximum deviation was set to 8 centimeter according to the experience of the graduate student that developed this system.

The scenario that altered the probabilities of successful executions was a change in the physical environment in which the Garcia Robot performed the task. Figure 4.3 depicts this task and both scenarios. During the first scenario, the robot has to follow a straight wall (left part of the figure). During the second scenario the wall has a 45 degree angle, forcing the robot to change its direction (right part of the figure). The solid arrows in the figure show the expected behavior of the robot.

To perform this task, we reused a couple of ROS components developed by a graduate student of the University of Nebraska-Lincoln. This components add the required functionality through the Garcia API, which offers a high-level control frame-

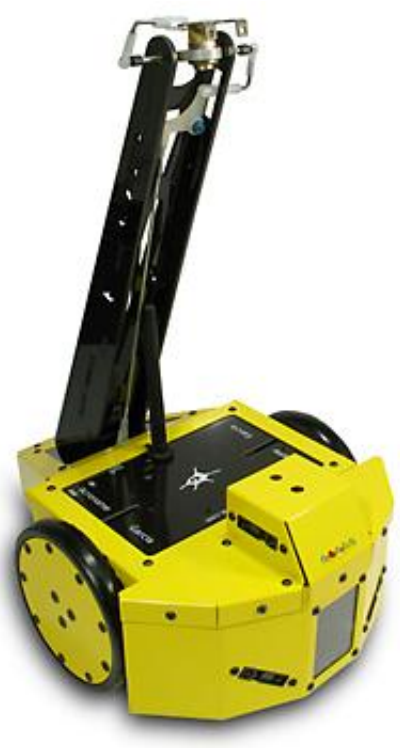
Figure 4.2: Garcia Robot

work to sends commands through a data streaming link to the network running on the Garcia robot.

Figure 4.4 provides a graph of the ROS components that conform the artifact under test and how these components interact. The artifact components are represented as ellipses connected by arrows whose labels represent the messages sent by each component. Each artifact component or ROS node is an individual operating process in charge of a specific functionality of the artifact. Their operations are usually triggered by messages of a determined type sent from other components and they send new messages to communicate the result of those operations. Each message type is a well-known structure by both the sender and the receiver that define a set of pair-value pairs.

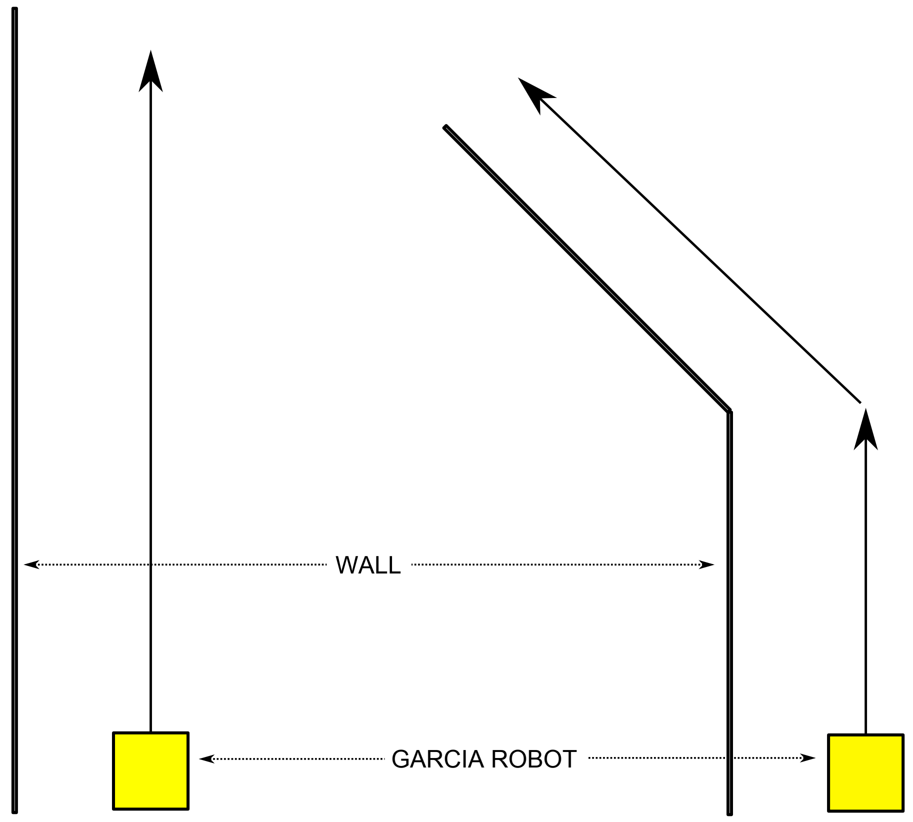

Figure 4.3: Wall Following: Training (Left) and Test (Right) Scenarios

In Figure 4.4 we can distinguish the two components that conform this artifact. The first component, *garciaControlNode*, holds the controller that calculates the speed of each wheel based on the IR range readings. The second component, *garciaInterfaceNode*, is an interface between the first component and the Garcia API. The interface node sends commands to the robot to set the wheels' speed and receives the information collected by the robot sensors (IR range readings). These commands do not appear in the graph as they are sent to the robot directly. The figure also depicts the exchange of messages between the components. It can be observed that the control node sends the adjusted speed of the wheels, *motorSpeedLeft* and *motorSpeedRight*, to the interface node and also that the interface node sends the range readings, *rightRanger* and *leftRanger*, to the control node.

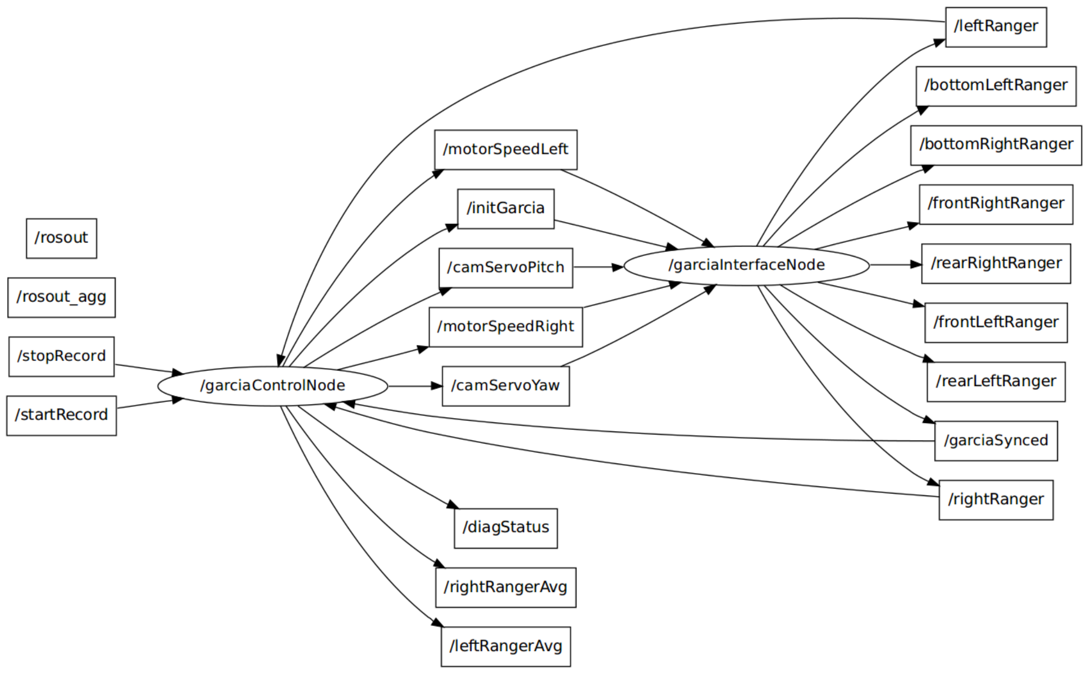

Figure 4.4: Wall Following System

4.1.2 Hummingbird - Ranger Height Controller

The second artifact is built on top of the AscTec Hummingbird (Figure 1.1), a quad-rotor whose vendor also offers a number of basic configurations. The basic configuration of this quad-rotor is equipped with MEMS gyro sensors, a GPS receiver, and a barometric altimeter. More advanced configurations include a 1.6Ghz Intel Atom processor for high computation processes such as image processing.

The task assigned to the second artifact is called *Ranger height controller*. The desired behavior in this case is that, given a target height set by a pilot, the quad-rotor should be able to maintain a target height using an ultrasound range finder attached to the rear arm of the quad-rotor as the height data source. It is important to point out that the pilot is still in control of the x and y position of the robot.

Once the target height is reached, a successful execution should fulfill three requirements. The first condition is that the difference between the target height and the actual height of the robot reported by the range finder is never greater than 15 cm. This condition checks that the main objective of the task is achieved. The second condition is that the difference between the actual height on time t and the actual height on time $t + 0.5sec$ is never greater than 5 cm. This condition checks if the quad-rotor goes up or down abruptly. The third condition is that the average difference between the target height and the current height is during the last 10 seconds less than 5 cm. The last condition checks if the robot is constantly oscillating. If any of these conditions is not met, the execution is considered a faulty one. These conditions were defined after we asked the researchers who developed this system how they determine if the system is working as expected.

The scenarios of the second experiment differed in that the quad-rotor pilot adopted contrary attitudes. During the first scenario, pilots were instructed to only perform slow and short maneuvers or to try to hover in place. During the second scenario, pilots were instructed to perform full accelerations and hard brakes. This last scenario was harder to handle by the ranger height controller because the ranger sensor was attached to an arm of the quad-rotor, which caused abrupt changes in the rotation angles, and consequently, in the height readings.

A total of 16 ROS components were developed for this artifact that can be grouped into 4 subsystems. The first subsystem (Figure B.1) is a PID controller for setting the required thrust level based on the ranger information and the target height set by the user. A second subsystem (Figure B.2) retrieves the ranger information, filtering it up and forwarding it to the controller. The third one (Figure B.3) processes the user commands. As mentioned before, pilots can set the target height and control the quad-rotor position by setting its pitch, roll and yaw angles. And the last subsystem

(Figures B.4 - B.5 - B.6) handles all the in-going and out-going serial communication from the remote computer to the quad-rotor.

4.1.3 Hummingbird - Vision Height Controller

The third artifact is also built on top of the Hummingbird platform and its task is called *Vision height controller*. The desired behavior is the same as the previous task: the quad-rotor should keep a target height without intervention of a pilot. Successful executions are determined using the same criteria as the previous task.

One of the differences with the second artifact is that this artifact has two height data sources. The first data source is the barometric pressure sensor included with the quad-rotor. The second data source is the result of a computer vision process that returns the radius of a purple ball located below the quad-rotor. A downward video camera had been attached to the aerial vehicle and connected to an Intel Atom processor in charge of processing the video stream in order to calculate the radius of the purple ball. Then, the inverse relationship between the ball radius and distance was used to calculate a estimate of the quad-rotor height and to adjust the commanded thrust. For this to work, the pilot was instructed to control the x and y position of the robot in order to locate it on top of the ball on the ground.

The two scenarios of the third experiment consisted of performing the experiment inside and outside where we expected for the different the light and pressure conditions to render different behaviors. The training executions were performed outdoors and the test executions indoors. The authors of this system noticed that more consistent results were observed in outdoor settings than indoor settings. Their analysis indicated that the reason of this problem are bursts of inaccurate readings of the pressure sensor.

The 17 ROS components developed for this artifact were grouped into 4 subsystems to make them more understandable. The first subsystem (Figure C.1) is a PID controller that calculates the required thrust by fusing the data obtained from the barometric pressure sensor and the image processing procedure. The second (Figure C.2) and third (Figure C.3) subsystems retrieve the pressure and radius information, filtering them up and forwarding them to the PID controller. And the last subsystem (Figures C.4 - C.5 - C.6), which was mostly reused from the previous artifact, handles the serial communication to and from the robot.

4.2 Experimental Setup

The experimental design is shared among the three performed experiments. It aims to evaluate if cost-effective models of statistical properties extracted from the execution traces of the artifacts could be built and monitored.

4.2.1 Measurements and Treatments

The effectiveness of the generated models is measured in terms of their precision and recall at predicting faults. Precision is defined as:

$$precision = \frac{TP}{TP + FP} \tag{4.1}$$

And recall is defined as:

$$recall = \frac{TP}{TP + FN} \tag{4.2}$$

where TP means *true positives* (number of successful executions that our approach classified as correct), FP means *false positives* (number of successful executions that

Indep. Var. Manipulated	**Model Name**	**Signif. Level**	**Training Set Size**	**Statistical Function**
Significance Level	A025-T10-Fc	0.025	10%	correlation
	A050-T10-Fc	0.050	10%	correlation
	A100-T10-Fc	0.100	10%	correlation
Training Set Size	A050-T05-Fc	0.050	5%	correlation
	A050-T10-Fc	0.050	10%	correlation
	A050-T25-Fc	0.050	25%	correlation
Stat. Function	A050-T10-Fc	0.050	10%	correlation
	A050-T10-Fm	0.050	10%	mean
	A050-T10-Fcm	0.050	10%	correlation & mean

Table 4.3: Generated models during the training phase

our approach classified as faulty), and FN means *false negatives* (number of faulty executions that our approach classified as correct). The cost of the approach is determined by the time required to infer statistical properties and monitor them.

Different combinations of the experiment's independent variables were used to evaluate their effect on the model effectiveness and cost. The independent variables of the experiments were the significance level of the statistical properties, the size of the training set, and the statistical functions used to identify statistical properties. The procedure to study the effect of a independent variable was to fix the value of the other two and to generate 3 different models using 3 different values of the target variable.

Table 4.3 shows how the independent variables were manipulated to generate 7 different models per artifact. Model A050-T10-Fc is repeated 3 times in the table to show how the independent variables were manipulated. Still, it was computed once. The significance levels studied were 0.025, 0.05 and 0.1, which lead to the generation of models A025-T10-Fc, A050-T10-Fc and A100-T10-Fc. In the case of the training set size, the values were 5%, 10% and 25% and the models were A050-T05-Fc, A050-T10-Fc and A050-T25-Fc. Finally, the statistical functions 'correlation', 'mean' and 'correlation-mean' (which means that both functions were used) ended up building models A050-T10-Fc, A050-T10-Fcm and A050-T10-Fm.

4.2.2 Execution Traces Collection

The same procedure was used to collect the execution traces of all the artifacts. They were collected using a ROS utility called *rosbag*. This tool is able to record every message sent during run-time by the components of a ROS system into *bag* files and save them to disk. These files are stored using a compressed binary format and an API is provided to retrieve raw information back.

An additional feature of this utility is the ability to split large bags into smaller ones. This feature was heavily used during experiments 2 and 3 because a flying session typically included several instances of the same task. In this way, we were able to generate multiple execution traces per task instance recorded during a single long flying session. A visualization tool called *rxbag*, also provided by ROS, was used to distinguish the different instances.

A Python script was developed to extract the execution traces information from *bag* files and to create new trace files using the JSON format accepted by our tool. This script makes use of the *rosbag* API.

Two factors constrained the number of traces we collected for the experiments on the Hummingbirds. First, as quad-rotors move in three dimensions and can achieve a high speed, a pilot is required to hold the radio controller to take control of the robot in case that something goes wrong. Second, batteries only last for a maximum of 15 minutes and the replacement process once they drain takes between 2 and 5 minutes. Still, we collected at least 100 traces per artifact. For the experiments on the Garcia, the limitation were the 90 seconds that took to run the task and to take the robot from the finish line to the start point.

In the end, every execution trace collected and used in the experiments contains at least 12 seconds and no more than 15 seconds of the artifact execution. This means that execution traces for the first artifact have an average of 5210 events, for the second 6471 events and 6091 events per execution trace of the third artifact.

After each scenario was executed, the traces were classified as successful or faulty using the objectives described for each artifact previously that were encoded as assertions in the code. Then, the training and test sets were defined. The traces of the training set were randomly selected successful traces from the training scenario. The test set consisted of every trace from the test scenario, the faulty traces from the training scenario, and the successful traces from the training scenario that were not included in the training set.

4.3 Results

The aim of this section is to present and discuss the results of the performed experiments. First, the statistical properties derived by our approach for each artifact are introduced. After that, the effects of the independent variables on the approach precision and recall are analyzed along with an explanation of the specific changes caused by each variable on the generated models. A discussion about the true negatives and false positives closes the analysis of the approach effectiveness. After that, the cost of inferring and monitoring properties are presented along with the effect caused by the independent variables.

Model Name	Wall Following	Ranger Height Controller	Vision Height Controller
A025-T10-Fc	5	34	24
A050-T10-Fc	5	35	26
A100-T10-Fc	5	38	27
A050-T05-Fc	5	36	27
A050-T10-Fc	5	35	26
A050-T25-Fc	5	33	22
A050-T10-Fc	5	35	26
A050-T10-Fm	3	42	39
A050-T10-Fcm	8	77	65

Table 4.4: Number of Inferred Statistical Properties

4.3.1 Artifacts Models

Table 4.4 shows the number of properties inferred by each treatment. We can observe that the number of properties decreases or remains the same with larger training sets and significance levels, suggesting that weaker properties were discarded. As expected, the number of inferred properties by model A050-T10-Fcm is the sum of model A050-T10-Fc and A050-T10-Fm.

For the Garcia Robot, our approach inferred the same 3 mean properties and 5 correlation properties regardless the value of the independent variables. This was expected as this artifact has only 15 variables and the task does not require a human operator, making the artifact behavior more consistent. The inferred properties capture the relationship between the state variables involved in the PID controller that sets the wheel's speed, the current distance to the wall, the error (current distance minus target distance) and the proportional (P) and derivative (D) values of the PID controller. Listing 4.1 presents the model $A025 - T10 - Fc$ and shows the specific properties generated during the training process. For example, the first line indicates that a correlation relationship exists between the error and the current distance to the wall. The window size of 5 of this property indicates a strong correlation which is obvious as $error = target - leftRanger$ and $target$ is a constant. The third line shows a strong correlation between error and P. This makes sense as the P term of a PID controller is proportional to the error. The larger window size in line number 4 shows a weaker correlation between the commanded speed of the wheel and the D term of the PID controller. The reason why is weaker is that the speed of the wheel is a function of several variables such as the difference between the previous and current $error$.

Listing 4.1: Model A025-T10-Fc of the Wall Following task

```
Correlation ,status+error ,leftRanger+data ,5 ,0.025
Correlation ,motorSpeedLeft+data ,status+sum,15 ,0.025
Correlation ,status+error ,status+P,5 ,0.025
Correlation ,motorSpeedLeft+data ,status+D,19 ,0.025
Correlation ,status+P,leftRanger+data ,5 ,0.025
```

Several more properties were inferred for the artifacts built on top of the Hummingbird as these are for more complex and larger systems. A group of properties inferred the correlation between the pilot commands and the navigation angles of the quad-rotor (e.g, when the commanded pitch increases, the pitch angle increases), other group captured the correlation relationship between speed and acceleration (e.g., if x acceleration increases, x speed increases), and another one captured the correlation between the variables that form part of the height PID controller. In the case of the ranger height controller, those variables are current height, thrust, P and D.

On the other hand, the statistical properties of the vision height controller includes the ball radius, the current pressure, thrust, P and D. We did not find a relevant pattern among the inferred mean properties except for the fact that 79.7% of the properties of this type had window sizes close to the maximum, indicating that they may be weaker properties.

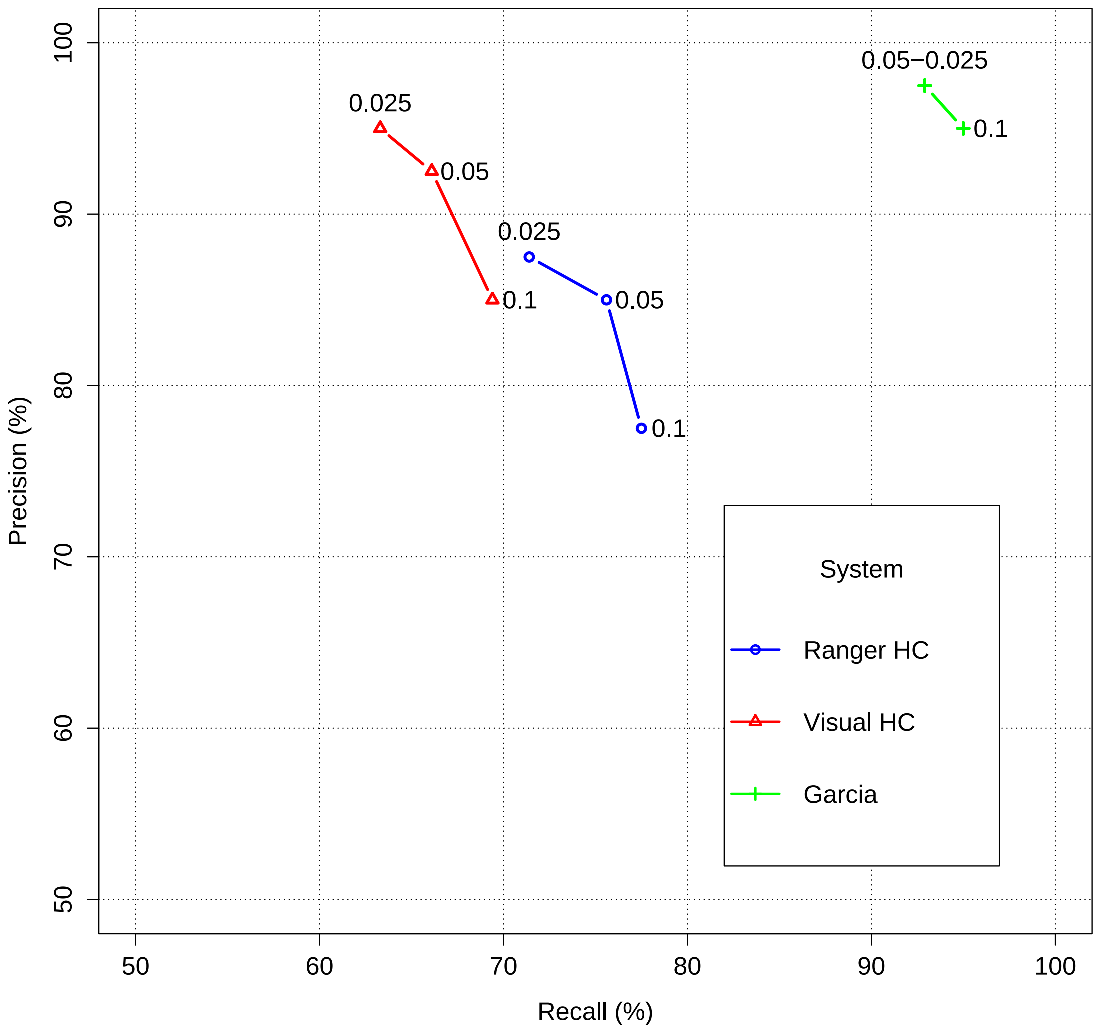

Figure 4.5: Effect of Alpha on Fault Detection Effectiveness

4.3.1.1 Effect of Alpha

Figure 4.5 shows how precision and recall change with different significance levels or alpha values. The x-axis describe the models recall and the y-axis their precision.

Each line corresponds to a particular artifact and the dots corresponds to a particular model of the artifact. From these graphs, it is possible to observe that with lower values of alpha, precision increases while recall decreases. This tendency can be explained by the fact that higher significance levels retain stronger properties and discard weaker ones. In some situations, a low alpha is not able to discard weak properties, but instead sets a window size close to the maximum. This makes weak properties hard to violate causing false negatives.

Figure 4.5 also shows that the models' precision never stays below 85% for significance levels of at least 0.05; however, the models' recall never reaches 80% for the second and third artifact. Between the models of these two artifacts, the Vision Height Controller ones have a higher precision. Our conjecture is that faults related to inaccurate sensors are easier to identify by our approach than those forced by the pilot because robots are built under the assumption that sensor have a minimum level of accuracy. The case of the Garcia robot is different as both precision and recall stay close to 100% for every value of alpha. As we mentioned before, we attribute this situation to the limited variability across the scenarios during the task execution.

4.3.1.2 Effect of the Training Set Size

Figure 4.6 shows how different sizes of the training set affect precision and recall. These figures depict a similar effect than the alpha value. Larger training set sizes increase precision while reducing recall. For example, the inferred models of the Vision Height Controller achieve a perfect precision if the training size is of 25%, while recall is reduced by 11% compared against the recall of training sets of 5%. This means the inferred models are able to retain most relevant properties when given more training.

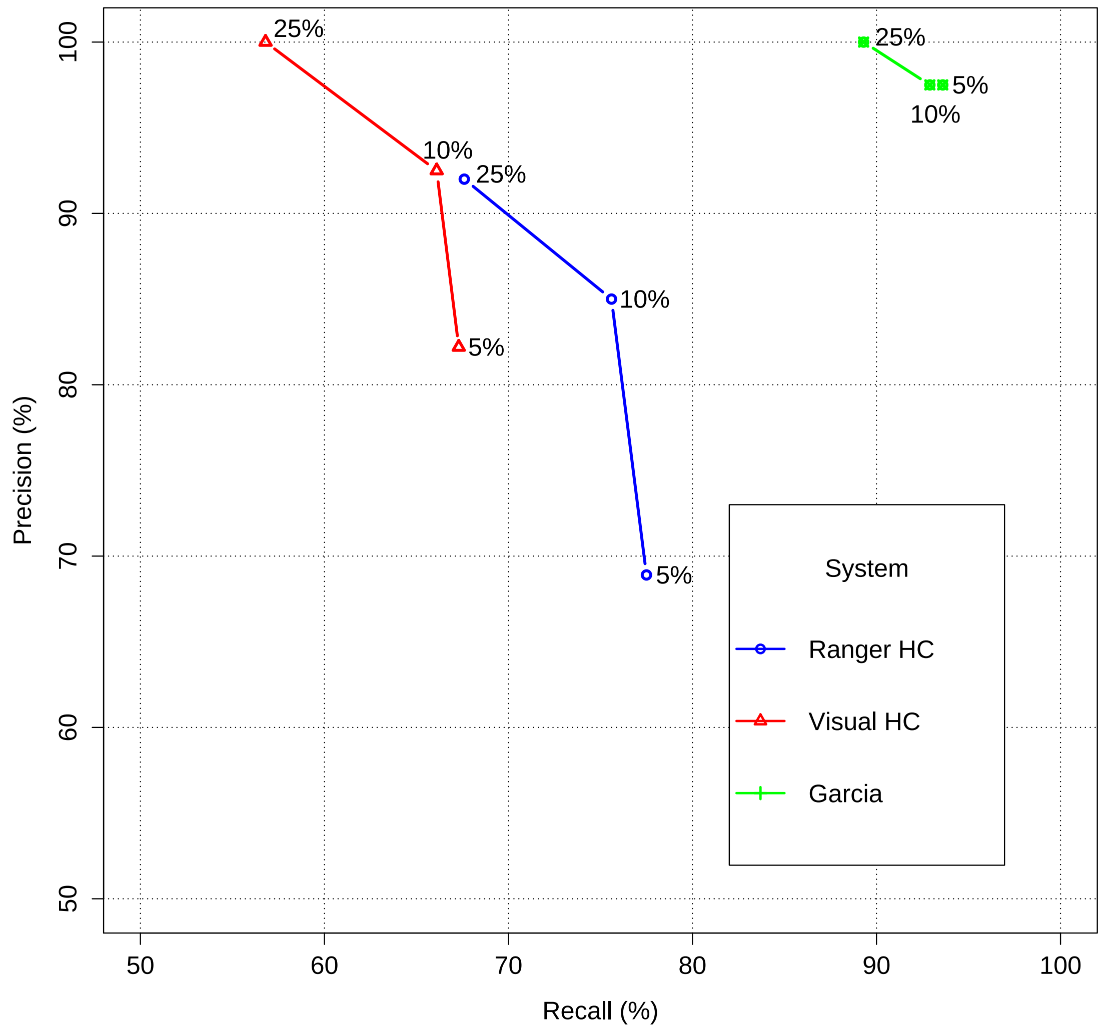

Figure 4.6: Effect of the Training Set Size on Fault Detection Effectiveness

Except for the Garcia robot artifact, small training sets are not capable of removing many weak properties, hurting the precision of the models. As more traces are added to the training set, weak properties are removed which improves precision.

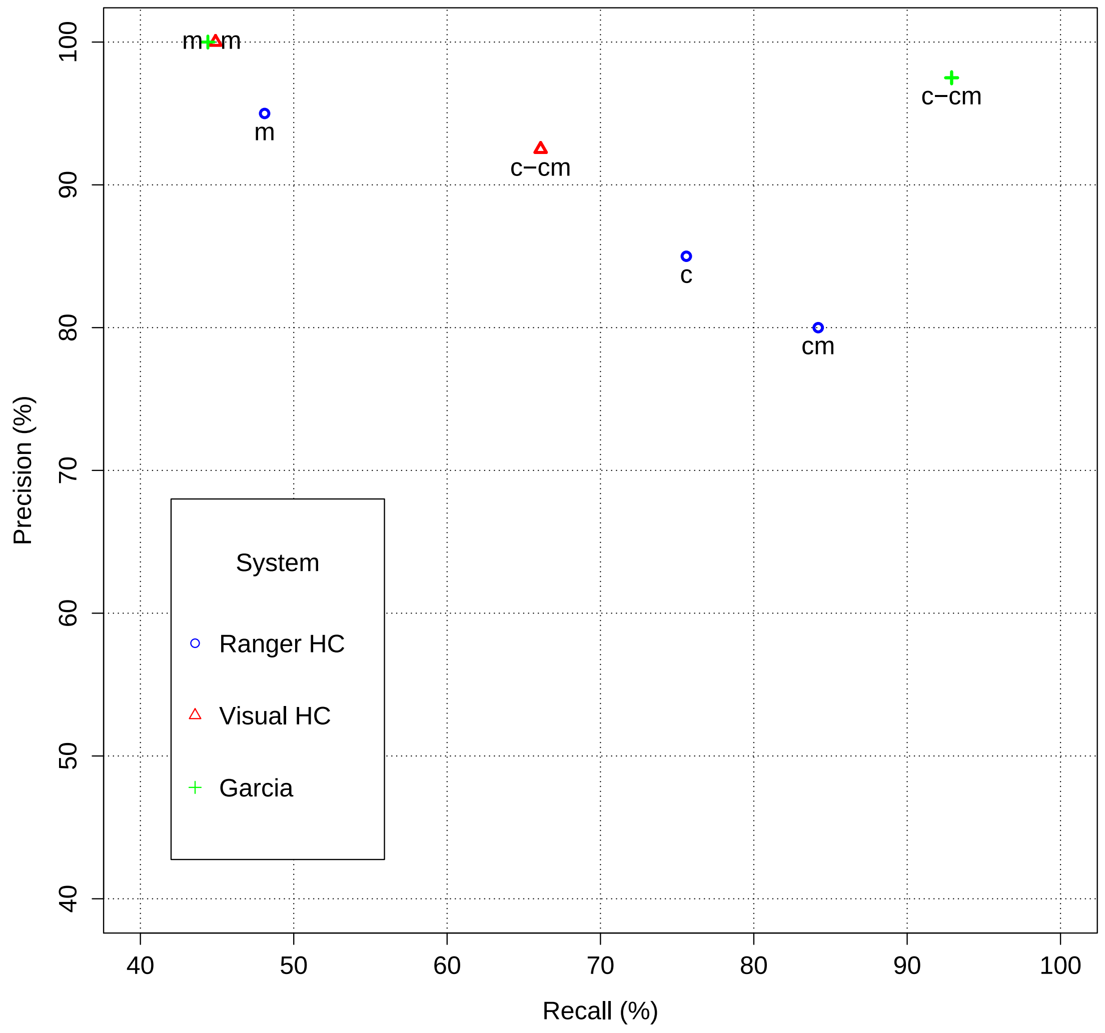

Figure 4.7: Effect of the Statistical Function on Fault Detection Effectiveness

4.3.1.3 Effect of the Statistical Functions

Figure 4.7 shows the effect of the used statistical functions on precision and recall where label *c* means correlation, label *m* means mean and label *cm* means correlation and mean. These plots show that the statistical function *mean* does not add much value. By its own, the mean function classified almost every execution as correct, leading to a almost perfect precision at the expense of a very low recall. When combined with the correlation function, the results are almost identical to only using the correlation function. The mean properties did not have any effect over the models of the Garcia and Vision Height Controller systems because they were never violated.

4.3.1.4 Noticeable True Negatives

The property that detected 90% of the true negatives or unknown behaviors for the first artifact was the one specified in the 4th row of Table 4.1. This property correlates the wheel's speed and the derivative term (D) of the PID controller. By inspecting the code, we found that the P term is a function of the current error, the D term is a function of the current error and the previous error and speed is a function of D and P. During training, as error was low, P and D changed together following a monotonic relationship. The test scenario breaks the property after the robot reaches the 45 degree angle in the wall. This cause and abrupt change in the error, and consequently, P and D changed in different directions.

In the case of the Ranger Height Controller artifact, the most effective statistical property at detecting violations of the task goals was the property that related the total acceleration of the quad-rotor and its speed over the z axis. This finding is correct but non-intuitive, and requires further explanation. During training, as the pilot performed smooth maneuvers or hovered, both total acceleration and z speed exhibited small variations apparently in same direction. This behavior created a strong correlation property with a window of 5. During the test scenario, the quad-rotor moved at full speed for a few seconds. This action is enough to unbalance the quad-rotor's arms. Thus, if the quad-rotor moved forward, then the rear arm would be higher than the target height, and consequently, the attached sensor ranger as well. Then, the height PID controller would reduce the drone thrust to adjust it to the correct height. However, even when it is moving at full speed, the artifact can handle this scenario successfully without violating any property. The violation actually happened when the pilot breaks the quad-rotor hard by moving full speed to the opposite direction. At that point, the total acceleration and speed over z still changed in the same direction, but at different intensities. As the window size was too small as determined by the training process, the strong correlation did not hold, and the property was violated.

Two properties revealed 82% of the failures of the Vision Height Controller artifact: the correlation between pressure and radius data and the correlation between pressure and thrust. These properties were inferred during the training scenario because they followed the monotonic relationship that they were expected to follow. If the radius increases, the pressure sensor indicates a lower altitude. However, during the test scenario, they were frequently violated because the pressure sensor operating indoors returned bursts of inconsistent data, altering the relationship inferred during training between radius and thrust.

4.3.1.5 Noticeable False Positives

One property caused 67% of the false positives in the Garcia Robot system. This property, which captures the correlation between motor speed and *sum*, is specified in the second line of Listing 4.1. The *sum* variable, probably created for debugging purposes by the system developer, is defined as $P + D$. We inspected the traces of these executions and the reason was that a few range messages were lost, causing the message that adjusts the wheel to not be sent as usual. When range messages were delivered at the usual rate, a bigger correction was applied with the consequences explained in 4.3.1.4.

During the training scenario of the Ranger Height Controller, a relationship between the commands pitch and roll was inferred when a training size of 5 was use. This is a property inferred by chance, which led to many false positives. The reason why it was inferred was the drift of the used Hummingbird. This drone, if no commands are sent, moves on it own backwards and to the left. This forced the pilot to apply a pitch and roll in the opposite directions in order to make the drone hover, originating the property. However, this was not the behavior of the pilot during the entire training sessions and the property was discarded with a training size of 10, reducing the number of false positives.

In the case of the Vision Height Controller, an example was the casual correlation relationship between pressure/radius and the commanded pitch. This property exists because during one of the training sessions the weather was windy, forcing the pilot to apply some pitch to keep the quad-rotor on top of the ball. Forcing small variations on pitch and radius in the same direction originated this casual property that caused a number of false positives when the smallest training set was used. We note that in both of the last two scenarios, larger training sets mitigated the false positives.

4.3.2 Cost of Training

We have defined the cost of training as the time required to create the 7 models specified in Table 4.3. The impact of each independent variable is analyzed separately. The results reported in this section were collected using the Unix command *time*.

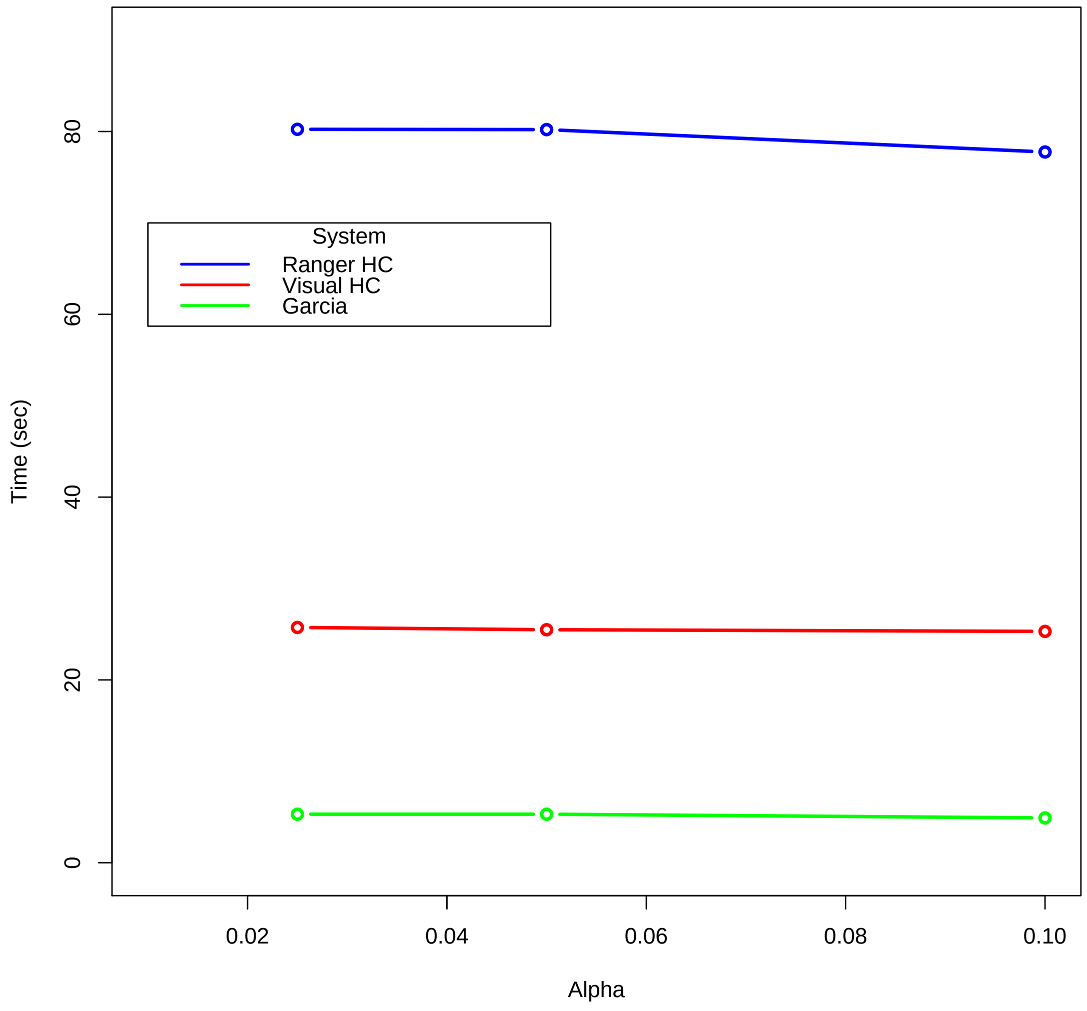

Figure 4.8: Effect of Alpha on Cost of Training

4.3.2.1 Effect of Alpha

Figure 4.8 depicts that the value of alpha has an insignificant effect on the time required to create the artifact's models. Three seconds is the improvement of the Ranger Height Controller artifact from the biggest to the smalles significance level.

The figure suggests that the cost for the Garcia robot artifact is lower because it has less variables, the executions traces have less events, and the number of properties in each model is lower than the other two artifacts.

Figure 4.8 also shows a considerable difference between the cost of training the Ranger and Vision Height Controller, even when these artifacts share a number of software components. One reason is that the models of the Ranger Height Controller have an average of 34 properties while the Vision Height Controller have an average of 25 properties per model.

It also appears that the number of iterations required to find the optimal windows size for each property is related to that cost given that the number of properties remains somewhat stable across the different alpha values. This means that the cost of the training phase can be adjusted by altering the maximum and/or minimum windows size or by changing the convergence condition.

4.3.2.2 Effect of the Training Set Size

Figure 4.9 shows the effect of the training set size. On the one hand, the artifact that incurs in the biggest cost is the Ranger Height Controller. This artifact generated larger traces and is the artifact with the larger number of properties. On the other hand, the Garcia robot generates the smallest traces and fewer properties. This indicates that the effect of the training set on the cost of training is proportional to the length of the execution traces and the number of properties. More data is need data to determine the specific impact of each variable. But overall we note that increases in processing time are proportionally smaller than increases in the training set size. Furthermore, we have several ideas on how to improve the performance of the approach by more closely connecting the inference and refinement cycle. We discuss those further in Chapter 5.

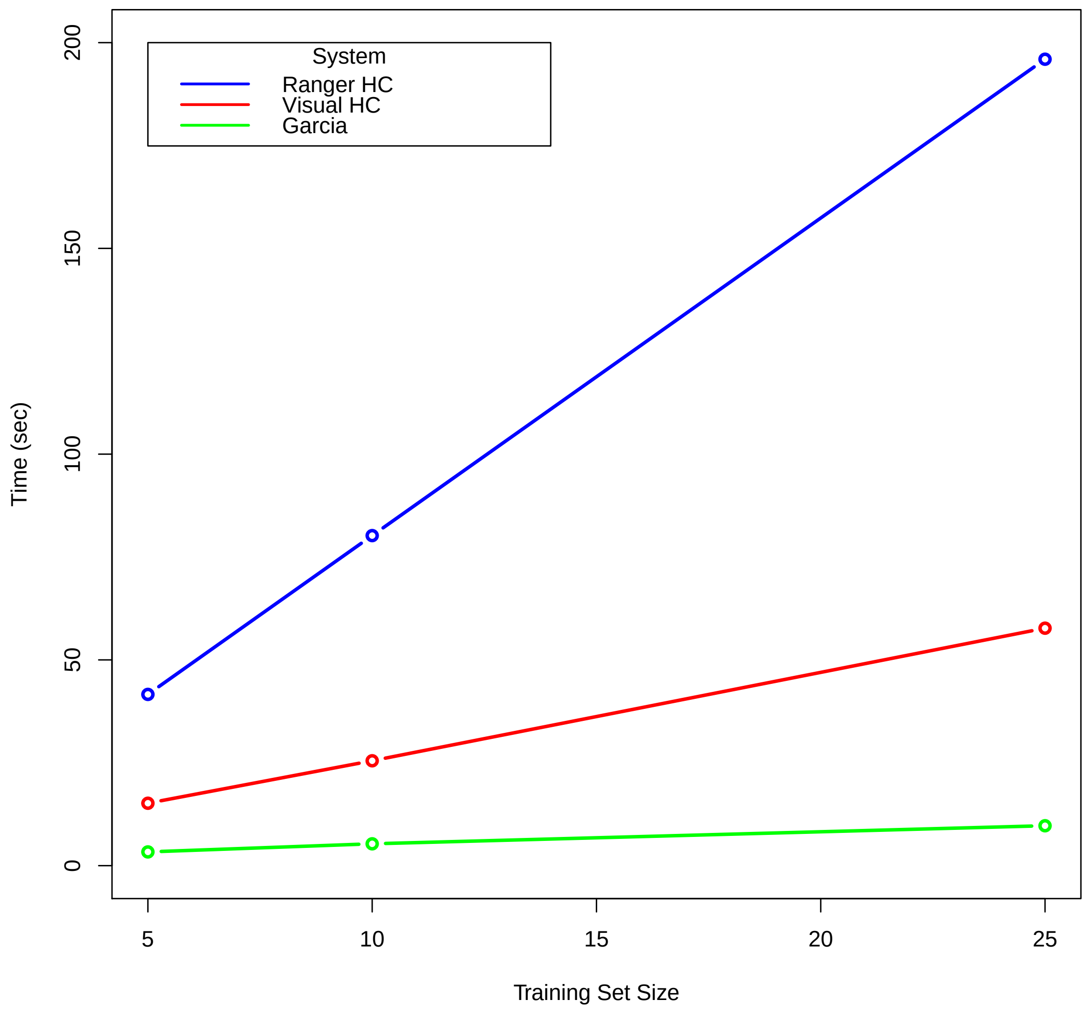

Figure 4.9: Effect of the Training Set Size on Cost of Training

4.3.2.3 Effect of the Statistical Functions

Figure 4.10 denotes that the cost of processing the mean function is less expensive than processing the correlation function by several orders of magnitude. For example, in the case of the Ranger Height Controller, the cost of inferring 77 mean properties

System	Garcia Robot	Ranger Height Controller	Vision Height Controller
Mean(sec)	0.10	0.11	0.13
SD(sec)	0.08	0.08	0.16

Table 4.5: Overhead

remains below 10 seconds while the cost of inferring 35 correlation properties is 81 seconds.

The figure also depict the cost of training properties with larger window sizes. The average window size of the correlation properties of the Ranger Height Controller is 17.5 and the average of the window size of the correlation properties of the Vision Height Controller is 11.6. While the difference is not a considerable, it seems to contribute significantly in the cost training, in addition to the number of properties. This can be explained by the fact that the binary-search optimization algorithm tested larger windows sizes when optimizing the properties of the Ranger Height Controller.

4.3.3 Cost of Monitoring

The cost of monitoring was defined as the total time required to execute the monitoring phase of our approach. The monitors were created using the models inferred in the previous experiment and we measured the time required to process the test against them. The time required to monitor the conformance of execution traces to a determined model is considerably lower than the cost of training mainly because only one iteration is performed during the test phase.

The average overhead of monitoring single traces and the average standard deviation are depicted in table 4.5. We can observe that the overhead does not present a high variability among artifacts, being higher when monitoring the Vision Height Controller artifact. An intuition of the reason is given later in this section.

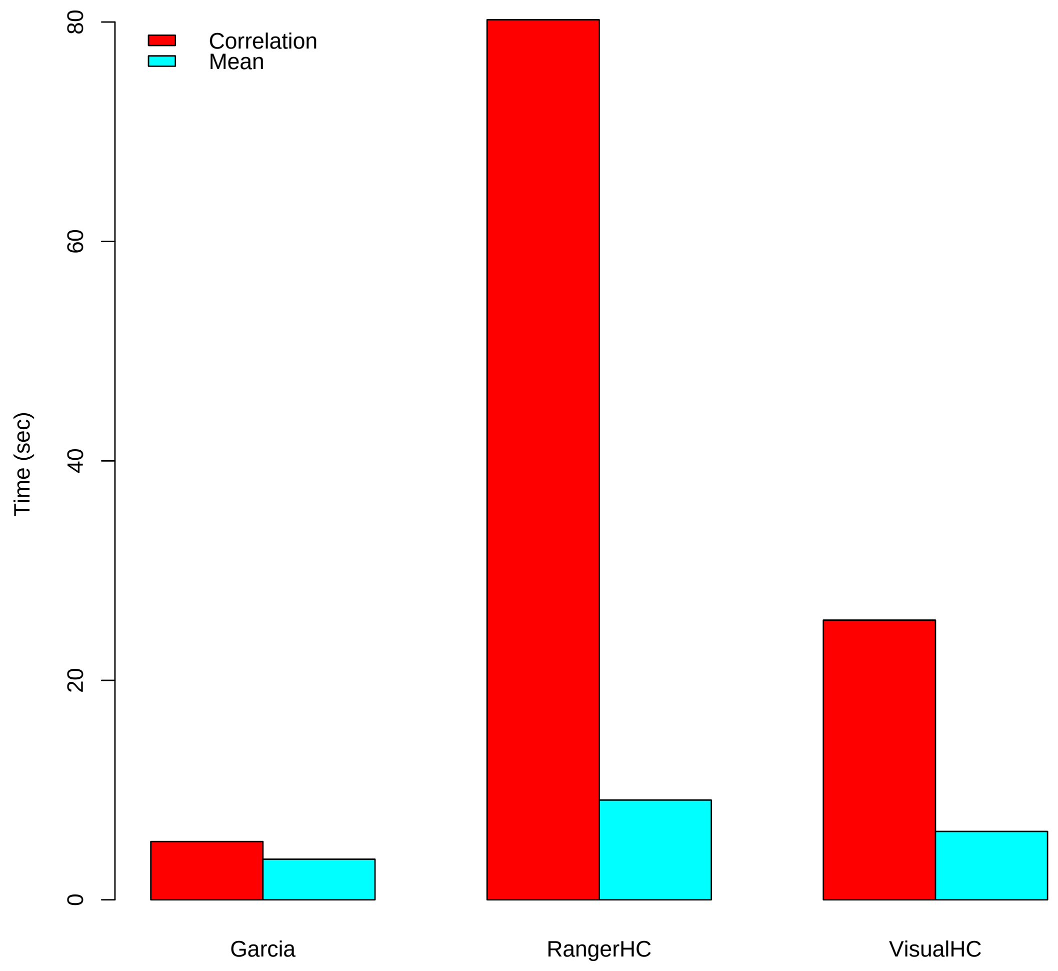

Figure 4.10: Effect of the Statistical Function on Cost of Training

4.3.3.1 Effect of Alpha

Figure 4.11 shows that alpha has little impact on the time required to monitor a set of traces. The difference between the Ranger and the Vision Controllers is in average 1.48 seconds. The cost of monitoring the Garcia Robot is only 7.2 seconds lower on average.

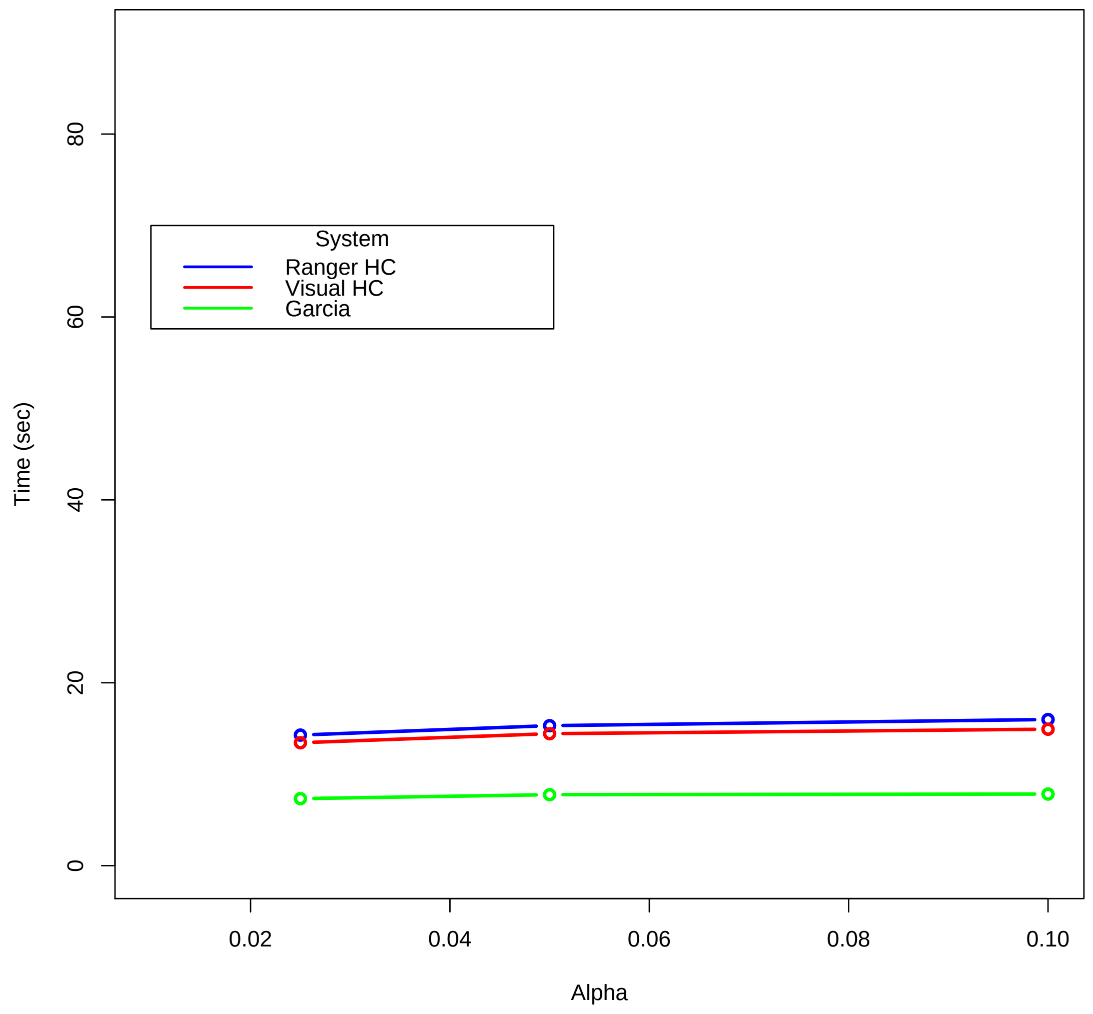

Figure 4.11: Effect of Alpha on Cost of Monitoring

Considering each system in isolation, it can be observed that the required time increases a slightly with higher values of alpha as this favors the inference of weak properties. The difference between the bigger and smaller alpha is 2.1 seconds on average.

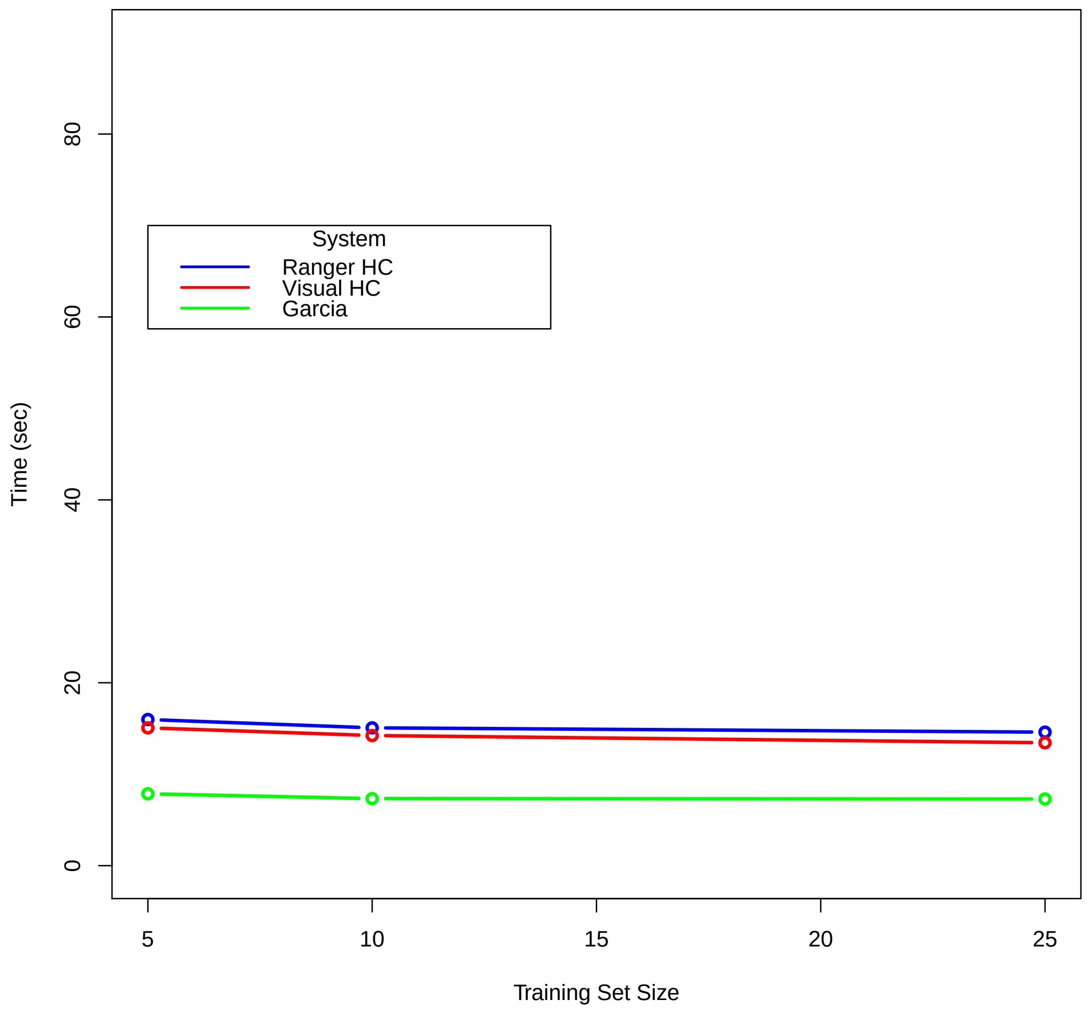

Figure 4.12: Effect of the Training Set Size on Cost of Monitoring

4.3.3.2 Effect of the Training Set Size

Figure 4.12 shows the effect of the training set size on the cost of monitoring. The required time barely increases with lower training set sizes because those models contain weaker properties while models generated with large training sets discard some of those properties, saving some monitoring time. The average difference between a training set of 5% and 25% for all systems is 0.89 seconds.

4.3.3.3 Effect of the Statistical Functions

Figure 4.13 denotes that the required time for monitoring correlation and mean properties is similar. A reason of this result is that the number of mean properties inferred is more than two times the inferred correlation properties. Another situation that balanced the costs between correlation and mean is that the mean properties are never violated, so the are monitored thoughout the entire process.

4.4 Alternative Methods

Using the same artifacts and the collected set of traces, we evaluated the fault detection power of properties generated by Daikon, a state properties inference tool, and compared it against statistical properties obtained with an alpha of 0.05 and correlation function. We faced several challenges in order to perform this comparison because there are not free automatic inferencing tools that can handle the distributed systems we are working with. As a result, we had to adapt Daikon to work at the message level as our tool does in order to achieve a fair comparison in terms of fault detection. Our adaptation required that the invariant inferencing and checking have to run separately for each component of the system. As Daikon does not have a parameter equivalent to the significance level of statistical properties, we have performed this comparison using just different training set sizes. Table 4.6 shows the number of invariants inferred by Daikon for each of the artifacts and training set sizes. The table shows that more training reduces the number of inferred state prop-

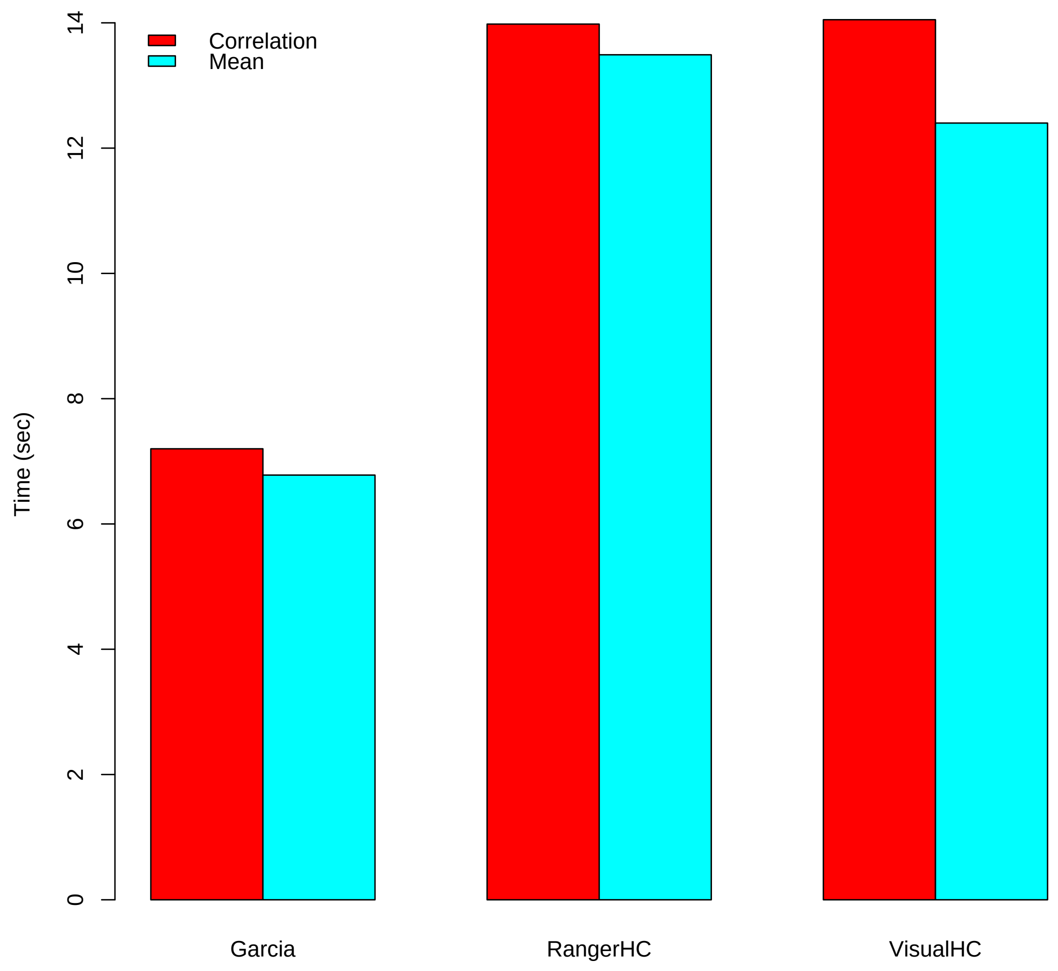

Figure 4.13: Effect of the Statistical Function on Cost of Monitoring

erties. However, the number of inferred state properties is considerably larger than the number of inferred statistical properties (Table 4.4).

Figure 4.14 shows the precision and recall of the state properties inferred with Daikon and the inferred statistical properties for the Wall Following task. In the figure it can be observed that Daikon was able to detect every faulty trace in the test set as faulty because the assertion that we used to classify them is an instance of Daikon's

Training Set Size	Wall Following	Ranger Height Controller	Vision Height Controller
5%	33	586	2392
10%	31	574	2040
25%	26	546	1830

Table 4.6: Number of Invariants Inferred by Daikon

templates. The inferred invariant was $target - leftRanger \leq 89$ and the assertion that we used to classify traces was $target - leftRanger \leq 100$. The invariant that caused most of the false positives with a small training set was the casual relationship $rightMotorSpeed \leq target$. We did not find any reason other than chance to explain why the relationship held for the smaller training set. The small difference in precision between the smallest and largest training set sizes suggests that statistical properties need less training data to achieve higher precision while recall remains comparable.

Figure 4.15 shows the results of the state properties inferred with Daikon and the statistical properties infered by our approach for the Ranger Height Controller task. The analysis of the violated invariants shows that the stronger state properties are those that included the speed of the quad-rotor. During training, the speed of the quad-rotor was maintained low while during test almost maximum speed was achieved. The lower precision of state properties was caused by those defining a maximum or minimum limit for altitude, pitch, roll, yaw, velocity or acceleration. They were violated easily when the pilot performed slightly different maneuvers during monitoring. Statistical properties seemed more tolerant to low quality test suites as they capture more fundamental relationships between variables and not just absolute values. As before, statistical properties required a smaller training set to achieve higher precision but their recall never reach 80%. State properties still keep perfect recall but precision barely passes 70% with the largest training set.

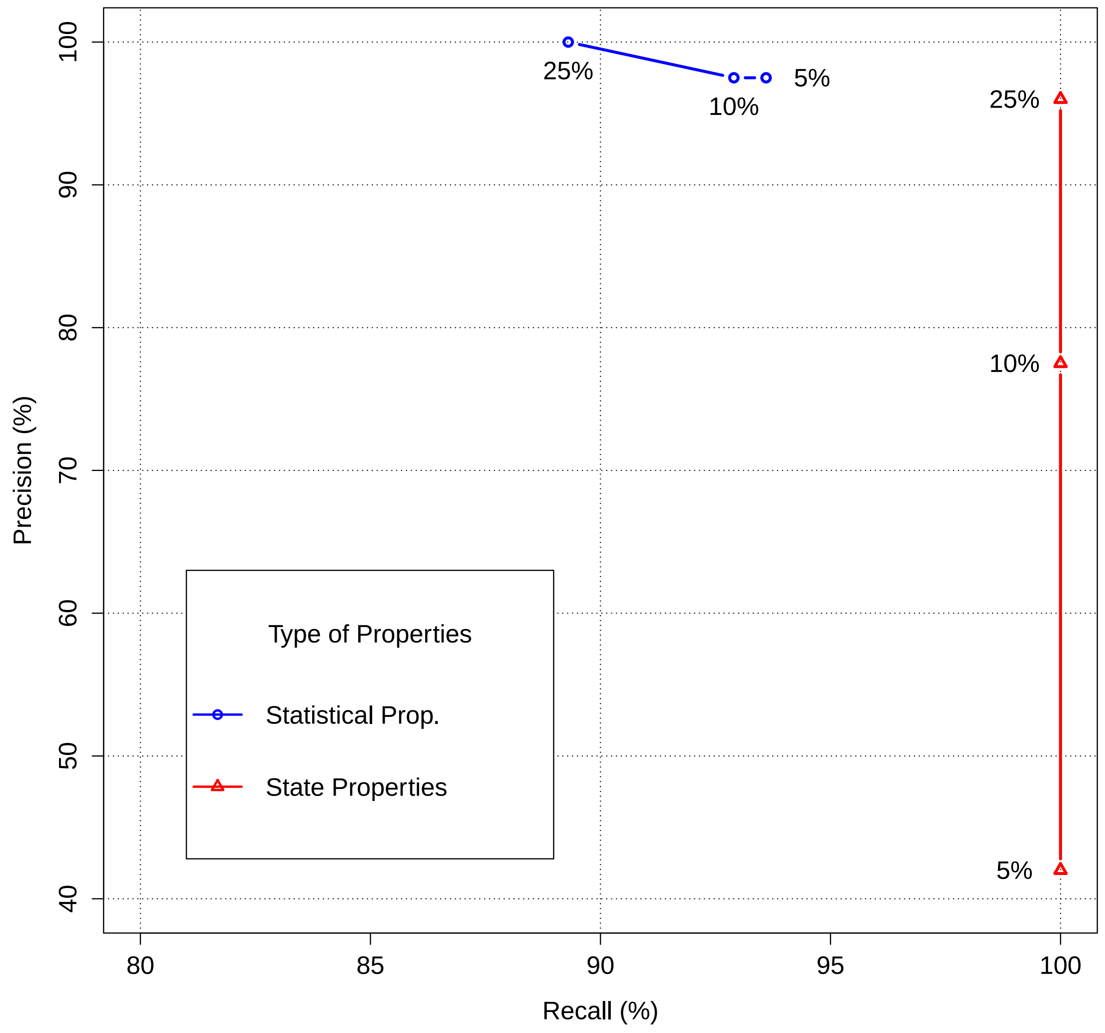

Figure 4.14: Wall Following - Comparison of inferred statistical properties against state properties inferred through Daikon using different training set sizes

Figure 4.16 shows similar results than the previous two figures except that the precision of the Daikon invariants are below 45%. This can be related to the large number of invariants inferred for this system, most of them uninteresting and highly dependent of the test suite. Invariants considering pressure sensor readings were the strong ones that classified the test set traces as faulty. False positives instead were

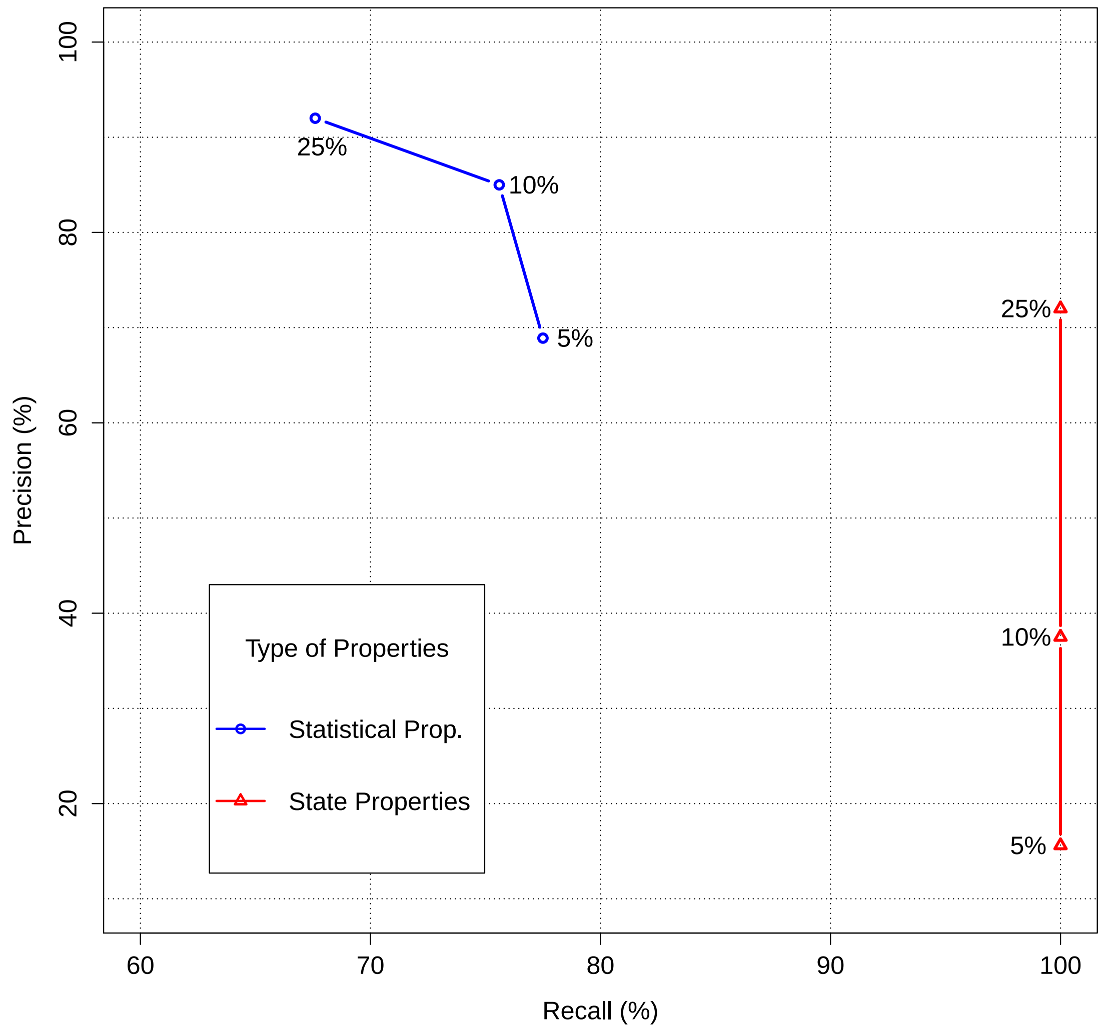

Figure 4.15: Ranger Height Controller - Comparison of inferred statistical properties against state properties inferred through Daikon using different training set sizes

caused by of invariants modeling ranges of navigation data as altitude, pitch, roll, yaw, velocity or acceleration.

Table 4.7 provides a comparison of the infering and monitoring cost of state and statistical properties. The cost of state properties should be viewed with caution as we used a version of Daikon that operates on message data instead of method entry

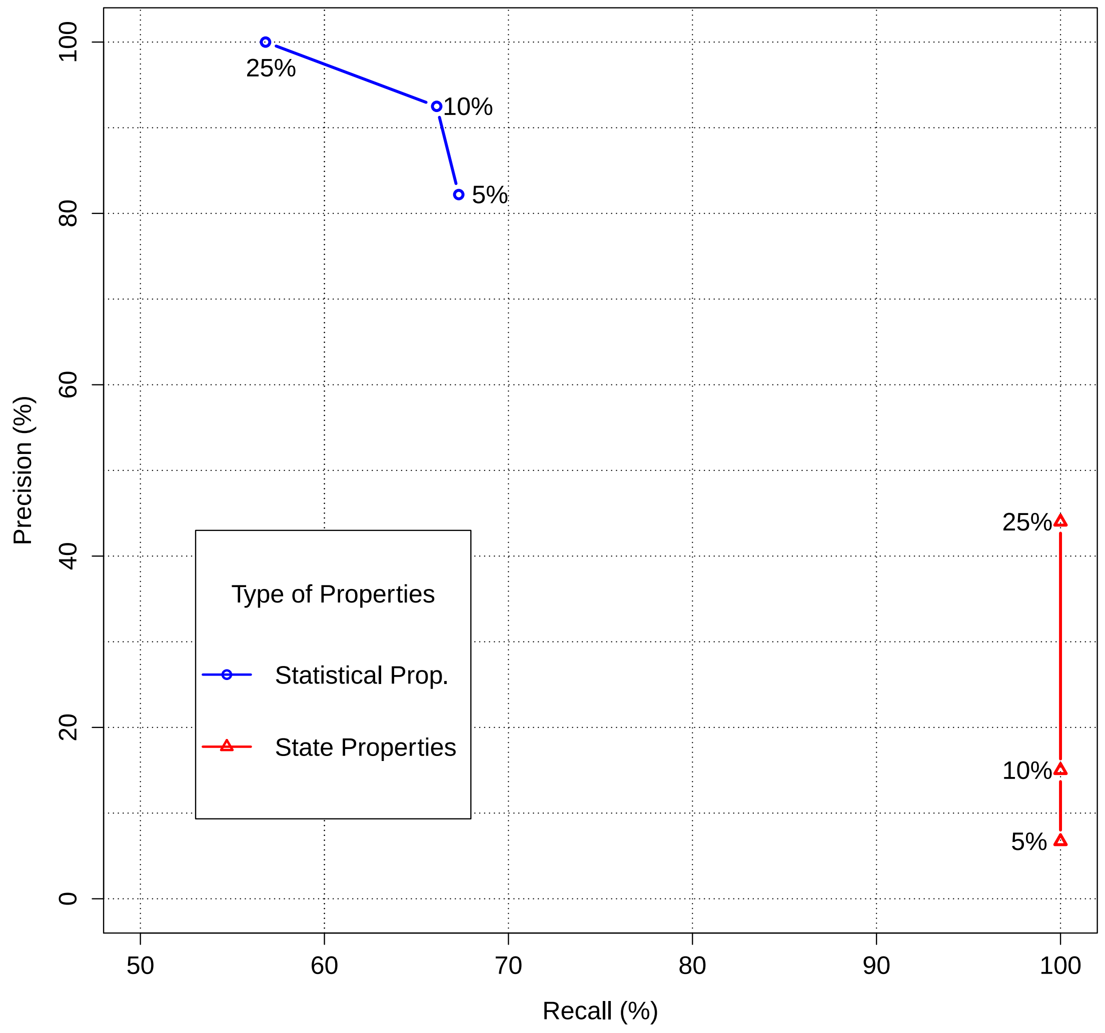

Figure 4.16: Visual Height Controller - Comparison of inferred statistical properties against state properties inferred through Daikon using different training set sizes

and exit data. These changes forced us to run the inference and monitoring process of state properties on individual nodes. For example, for the *Visual Height Controller* and a given training set size, we run the inference process 17 times, 1 per component, and the monitoring process another 17 times. The values shown in Table 4.7 for state properties are the sum of those 17 executions. Compared to our approach that only

Artifact	Prop. Type	Cost Training			Cost Monitoring		
		5%	**10%**	**25%**	**5%**	**10%**	**25%**
Wall following	Statistical	5.7	8.3	14.7	7.8	7.3	7.2
	State	5.3	7.5	13.8	135.5	151.1	152.2
Ranger Height Controller	Statistical	41.6	80.2	195.9	15.9	15.0	14.6
	State	10.1	20.2	50.6	953.0	857.7	810.1
Visual Height Controller	Statistical	15.1	25.4	57.7	15.0	14.2	13.4
	State	27.8	54.2	135.4	1015.2	913.7	862.9

Table 4.7: Cost of Inferring and Monitoring in Seconds

runs once for the entire system, this added a considerable extra overhead caused by the creation of new OS processes and I/O operations. This extra overhead and the considerable larger number of state properties made their cost of monitoring higher than monitoring statistical properties. The window optimization process incurs in a large overhead when inferring statistical properties making them more costly to infer than state properties. However, in the case of the *Visual Height Controller*, as the number of inferred state properties is so large (an average of 2087 for the three training set sizes) the inferring cost is indeed larger than inferring statistical properties. In spite of the relative differences in time, we can observe that in both cases the cost of inferring increases with the training set and the monitoring cost instead decreases as a consequence of the lower number of state properties to check.

4.5 Summary

The results of the performed experiments showed a number of interesting characteristics of our approach. In the first place, the results show that correlation properties achieved a precision of 91.1% on average across all artifacts with a training set of 10%, and recall levels of over 77.8%. Properties using the mean function, however, were not as effective. We also understand how weak properties hurt the precision of models and how they can be reduced by increasing the significance level and the size of the training set. Figures 4.5 and 4.6 depicts that these corrective techniques also caused a decrease of recall, but at a lower rate.

The study of the time incurred in creating models and checking properties suggests that the cost of training is a function of the number of traces in the training set, the size of those traces, the number of properties inferred, their windows and the condition that marks the end of the training phase. These are assumptions due to the lack of comparative data. We assume that the bottleneck of the inferencing process is the optimization procedure. During the first iteration, training phase has to deal with a large number of statistical properties. Concretely, n properties per $SingleVariableStatisticalProperty$ and $n*(n-1)$ per $CompoundStatisticalPropertey$.

As expected, the cost of monitoring is significantly lower than the cost of training. The fact that the significance level does seem to affect the cost of monitoring is encouraging because suggest that stronger properties do not incur in more overhead.

Compared to automatically inferred state properties, our experiments showed that statistical properties require less training data to achieve higher precision than state properties. Also, state properties are highly sensitive to the test suite quality while statistical properties seem to better tolerate this type of variation. Like statistical properties, the cost of infering and monitoring state properties is proportional to the training set and the number of inferred properties to check.

Chapter 5

Conclusions and Future Work

In this work, we have introduced a new type of software properties called *Statistical Properties* which characterize the behavior of software systems by identifying statistically significant relationships between a window of values of their variables. These properties can be inferred efficiently using automatic techniques and used to check deeper aspects of the behavior of the system under analysis. We have also developed approaches to automatically infer and monitor these statistical properties. The inference approach discovers instances of statistical properties specified in a set of pre-defined property templates and optimizes their window size so the incurred overhead is as low as possible while the required significance is maintained. Mechanisms to eliminate weaker properties and handle tuples of variables with different updates rates were developed as well. Our approach considers that the information passed through messages can represent the system behavior. If the traced variables do not contain fundamental information about the system operations, statistical properties inferred from those traces will have little fault detection power.

We have also implemented these approaches and two properties templates: *correlation* and *mean*. Third parties can use our implementation simply by formatting execution traces as a sequence of JSON objects and augment the set of property templates by extending an abstract class which takes care of interacting with our implementation.

We have assessed our approach against three distributed software systems that control robotic platforms. The approach was able to infer interesting properties for all the three systems we studied, and the assessment showed what factors contributed the most to its effectiveness. More specifically, the higher significance levels and larger training sets lead to the identification of strong correlation properties that characterize the systems operations with a higher precision with minimal reduction in terms of recall. The results also show that stronger statistical properties have smaller window sizes which implies a reduction in the monitoring cost.

In the future, we will expand this work in several directions. First, we need to conduct more extensive comparison of statistical properties against state properties, and also start a comparison versus temporal properties to allow us to more precisely locate statistical properties in the cost-effectiveness spectrum. The major challenge to accomplish this is to identify inference tools that are robust enough to work on the systems we are targeting.

Second, we will also define new statistical property templates and evaluate their efficiency. More specifically, we are considering properties that focus on the distribution of variables and properties that performs analysis of variances and covariances. We would also like to study the effect of the minimum and maximum window sizes defined by property templates since these parameters can affect the cost of inferring by reducing the number of iterations during the optimization process. Similarly, we would like to develop and study more complex update policies to accommodate variables with different frequencies and also occurring in different parts of the system. In addition, we would like to investigate the performance of our approach on different languages and architectures to better understand the generality of our approach. Furthermore, although the approach effectiveness was shown for systems whose behavior was dominates by controllers, it would be interesting to observe it on a broader kind of systems.

Third, we plan to adapt our tool to enable on-line verification of statistical properties. We are considering different options depending on whether the involved variables resides in the same program or are distributed across programs. Encoding monitors into the program is a viable option for single programs. A plausible approach for compound properties located across different processes is to create an observer process that subscribes to the topics containing the variables of interest. When moving the approach to operate on-line, we would like to explore how to do sampling to control the technique's overhead without losing effectiveness. We conjecture that sampling could be embedded into the policy structure to, for example, reduce the monitoring frequency of properties that are unlikely to fail.

Fourth, we plan to explore if statistical properties are able to predict faults, instead of just detecting the occurrence of faults, in order to take corrective actions to prevent them. Initially, we would like to prevent the last action that causes the fault. In a more advanced stage, the goal would be to learn a sequence of events or a tendency that leads to faults and take a corrective action earlier.

Appendix A

Statistical Tables

n	α = **0.05**	α = **0.025**	α = **0.01**	α = **0.005**
5	0.900			
6	0.829	0.886	0.943	
7	0.714	0.786	0.893	
8	0.643	0.738	0.833	0.881
9	0.600	0.683	0.783	0.833
10	0.564	0.648	0.745	0.794
11	0.523	0.623	0.736	0.818
12	0.497	0.591	0.703	0.780
13	0.475	0.566	0.673	0.745
14	0.457	0.545	0.646	0.716
15	0.441	0.525	0.623	0.689
16	0.425	0.507	0.601	0.666
17	0.412	0.490	0.582	0.645
18	0.399	0.476	0.564	0.625
19	0.388	0.462	0.549	0.608
20	0.377	0.450	0.534	0.591
21	0.368	0.438	0.521	0.576
22	0.359	0.428	0.508	0.562
23	0.351	0.418	0.496	0.549
24	0.343	0.409	0.485	0.537
25	0.336	0.400	0.475	0.526
26	0.329	0.392	0.465	0.515
27	0.323	0.385	0.456	0.505
28	0.317	0.377	0.448	0.496
29	0.311	0.370	0.440	0.487
30	0.305	0.364	0.432	0.478

Table A.1: Critical Values for Spearman's Rank Correlation Coefficients

Appendix B

Ranger Height Controller System

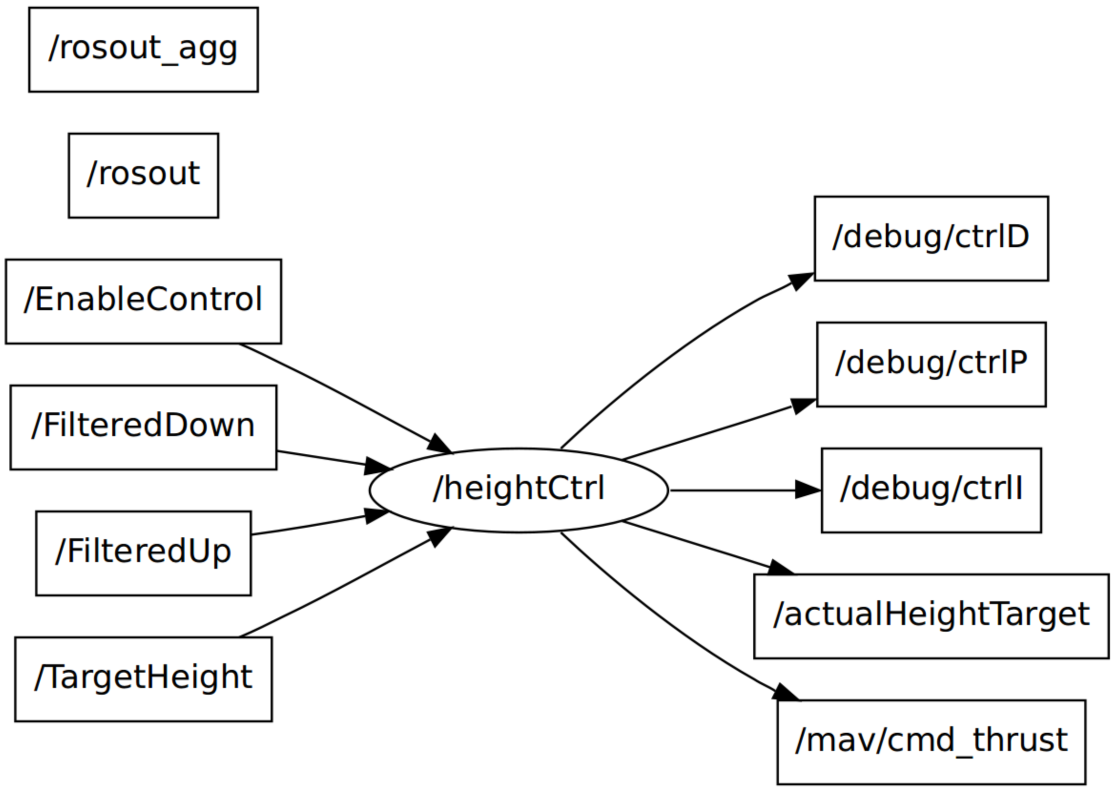

Figure B.1: PID Controller Subsystem

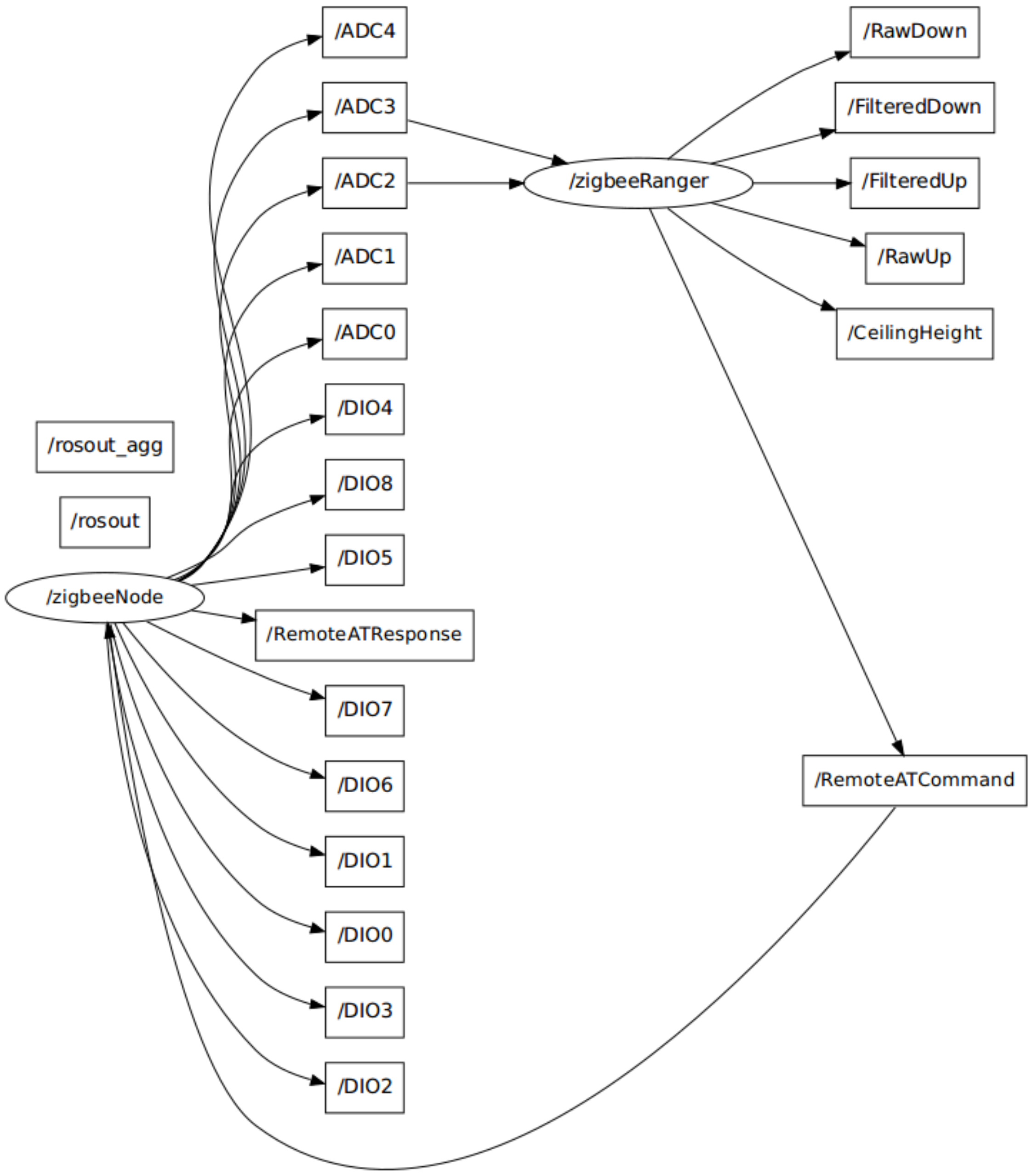

Figure B.2: Ranger Subsystem

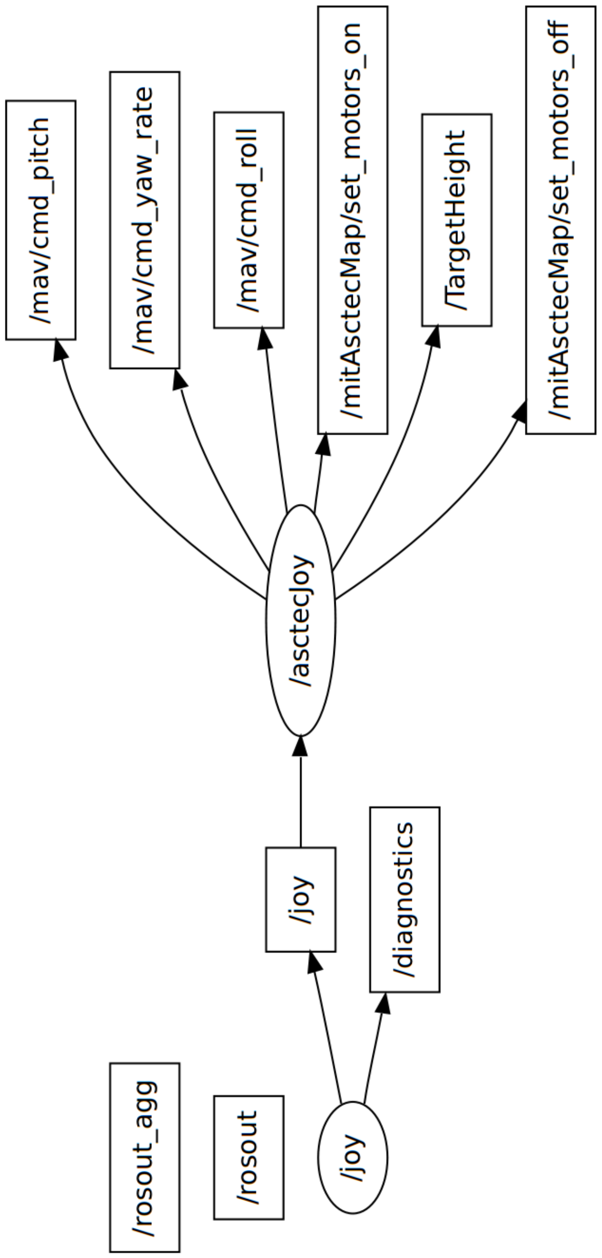

Figure B.3: User Commands Subsystem

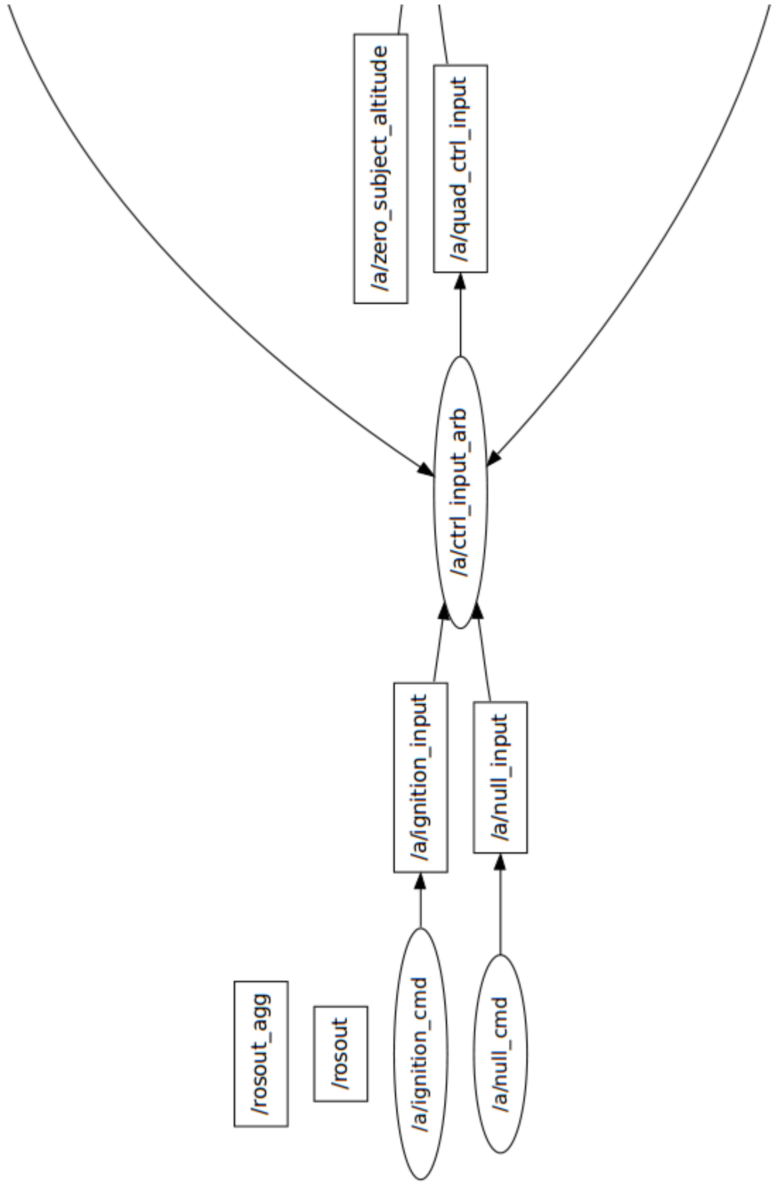

Figure B.4: Serial Communication Subsystem - Part 1

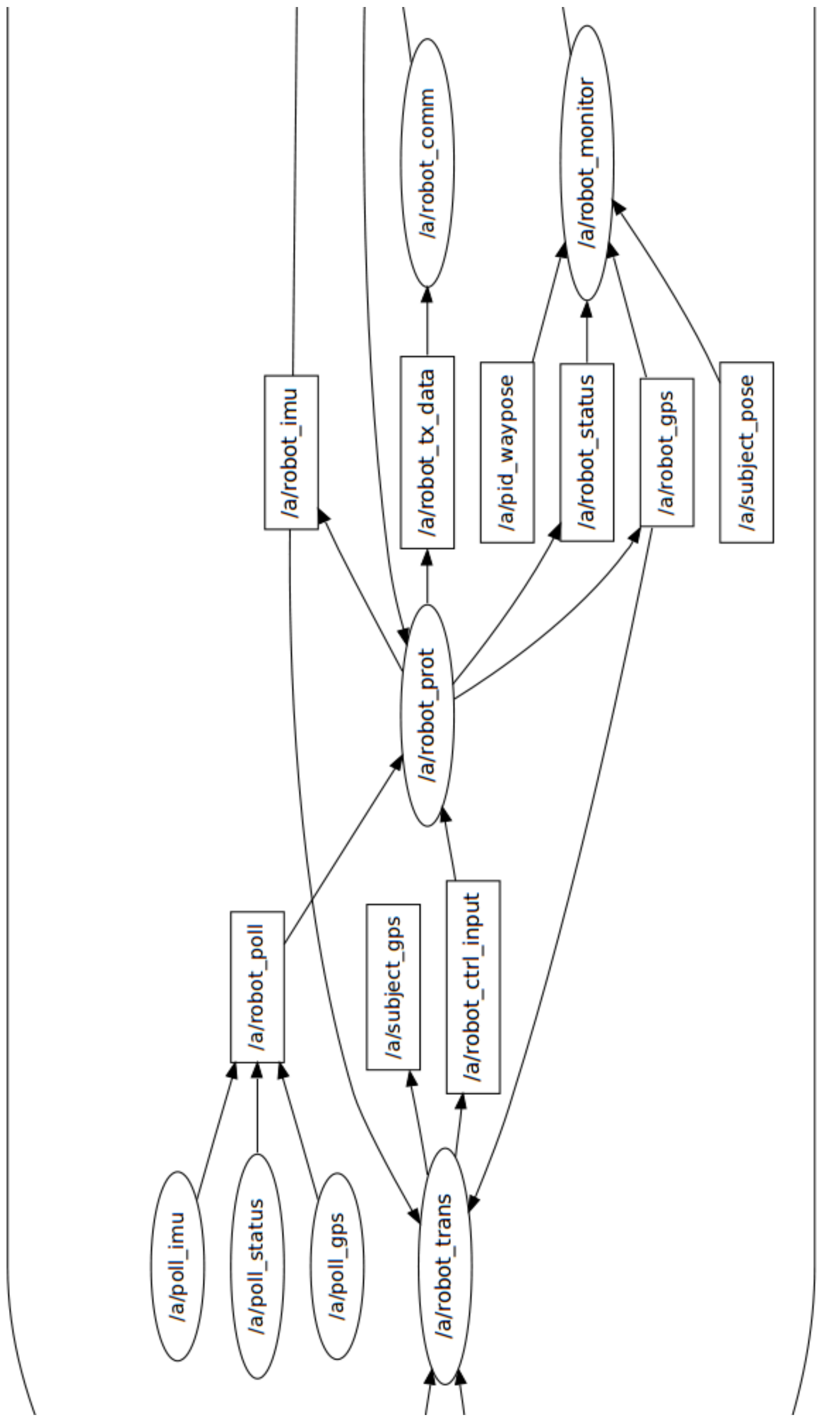

Figure B.5: Serial Communication Subsystem - Part 2

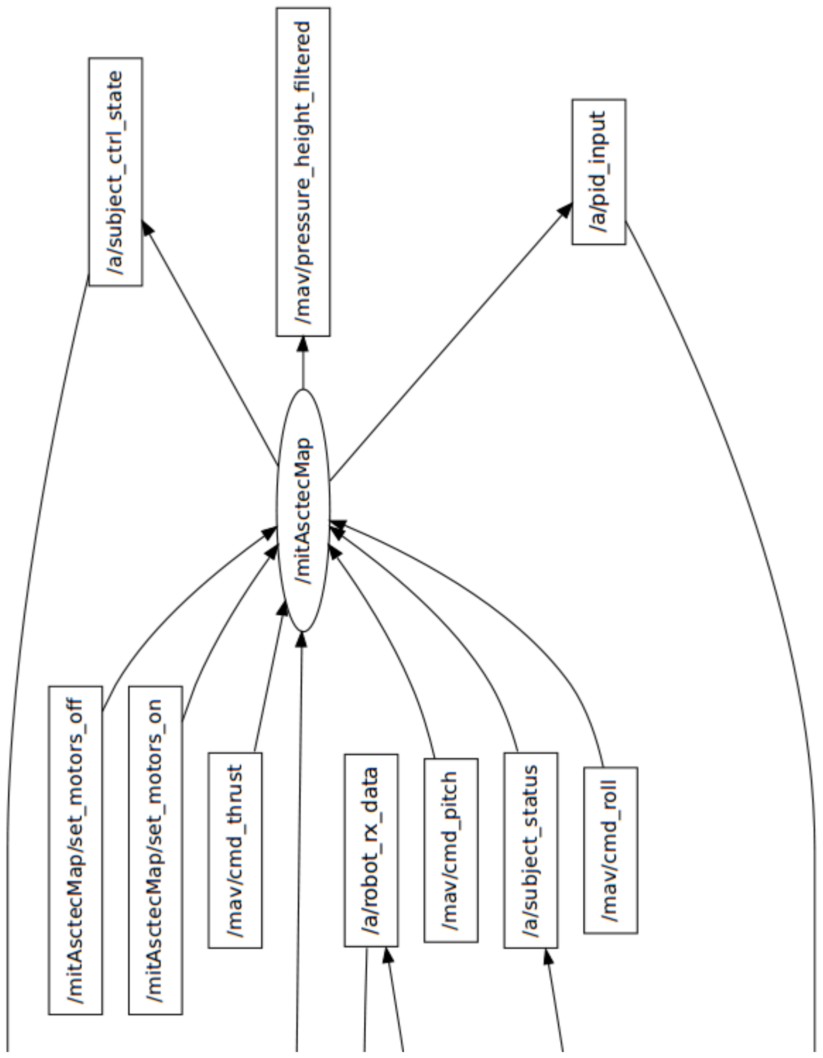

Figure B.6: Serial Communication Subsystem - Part 3

Appendix C

Vision Height Controller System

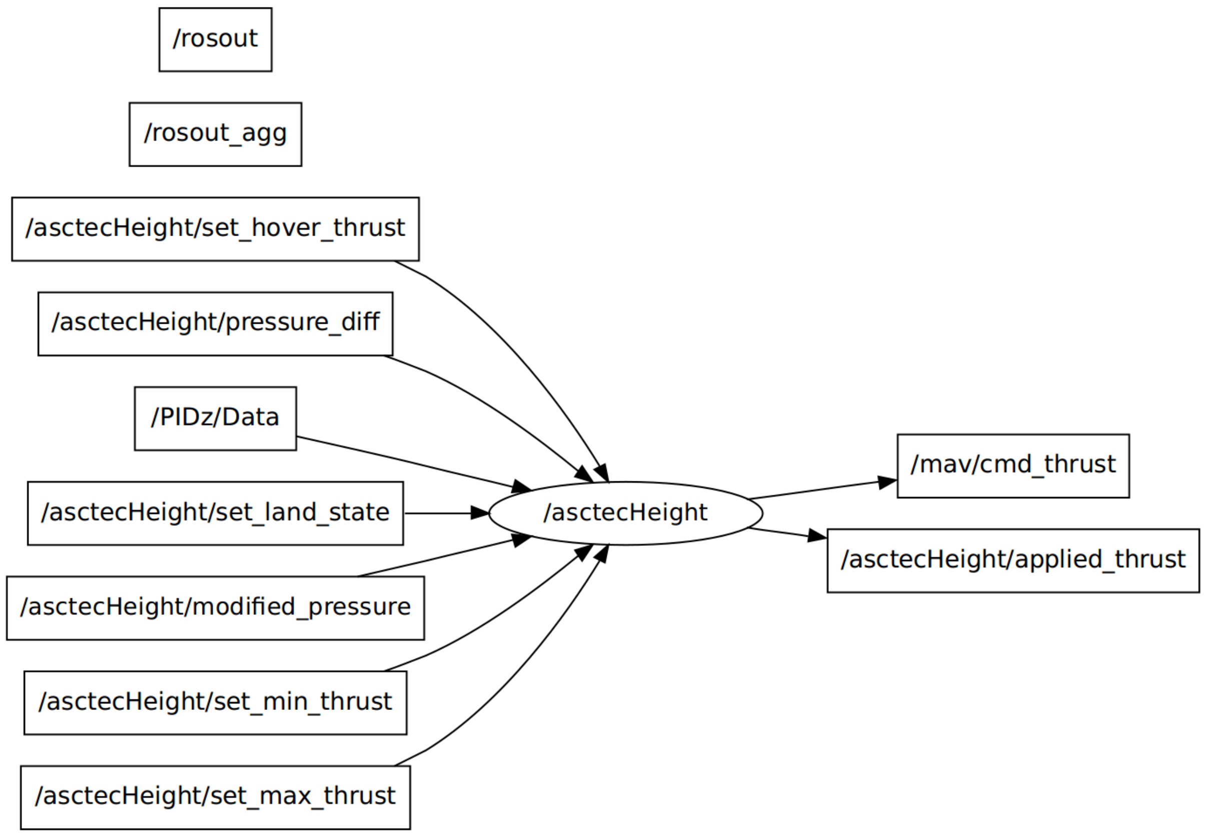
/rosout
/rosout_agg
/asctecHeight/set_hover_thrust
/asctecHeight/pressure_diff
/PIDz/Data
/asctecHeight/set_land_state
/asctecHeight/modified_pressure
/asctecHeight/set_min_thrust
/asctecHeight/set_max_thrust
/asctecHeight
/mav/cmd_thrust
/asctecHeight/applied_thrust

Figure C.1: PID Controller Subsystem

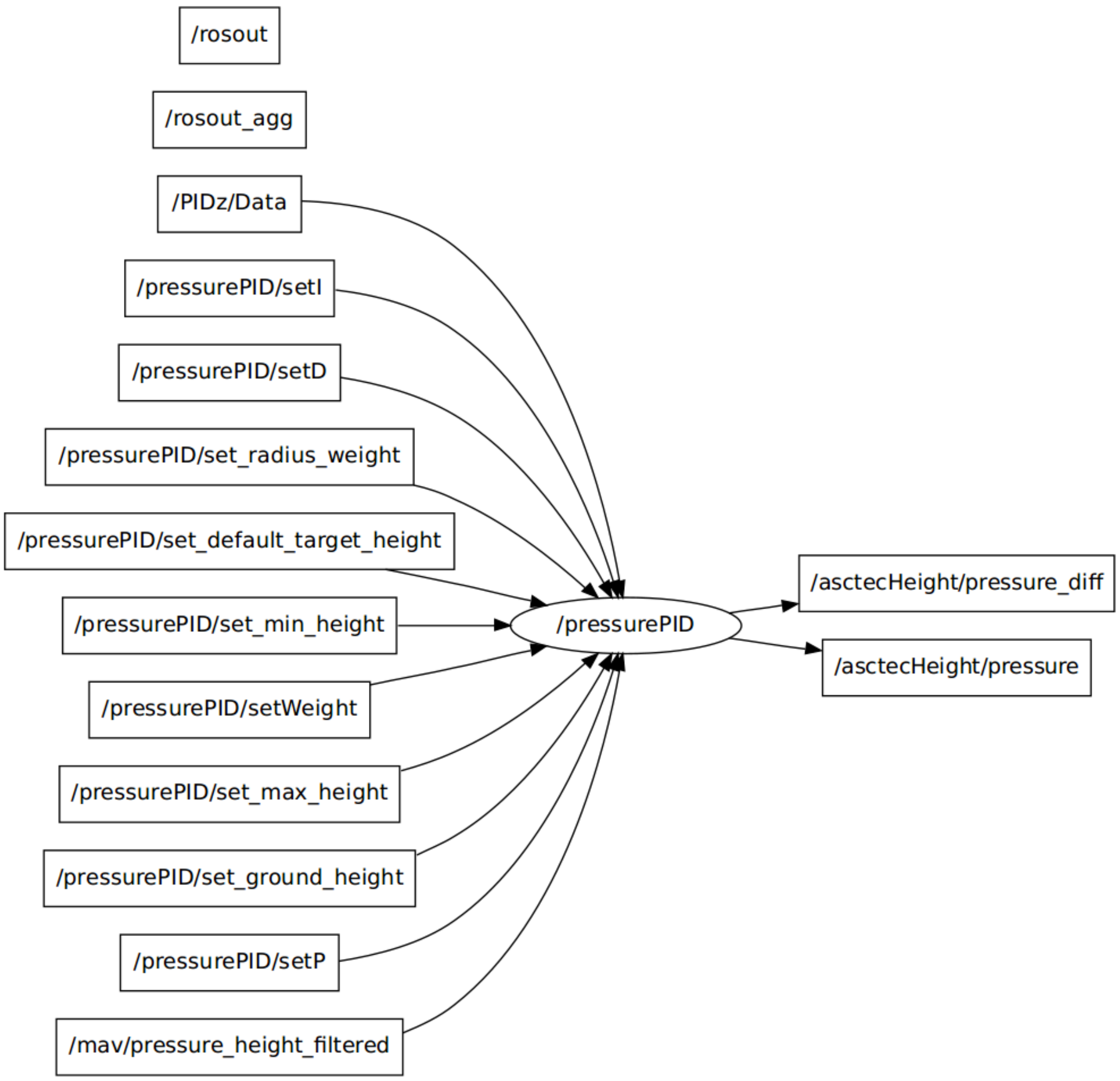
/rosout
/rosout_agg
/PIDz/Data
/pressurePID/setI
/pressurePID/setD
/pressurePID/set_radius_weight
/pressurePID/set_default_target_height
/pressurePID/set_min_height
/pressurePID/setWeight
/pressurePID/set_max_height
/pressurePID/set_ground_height
/pressurePID/setP
/mav/pressure_height_filtered
/pressurePID
/asctecHeight/pressure_diff
/asctecHeight/pressure

Figure C.2: Pressure Subsystem

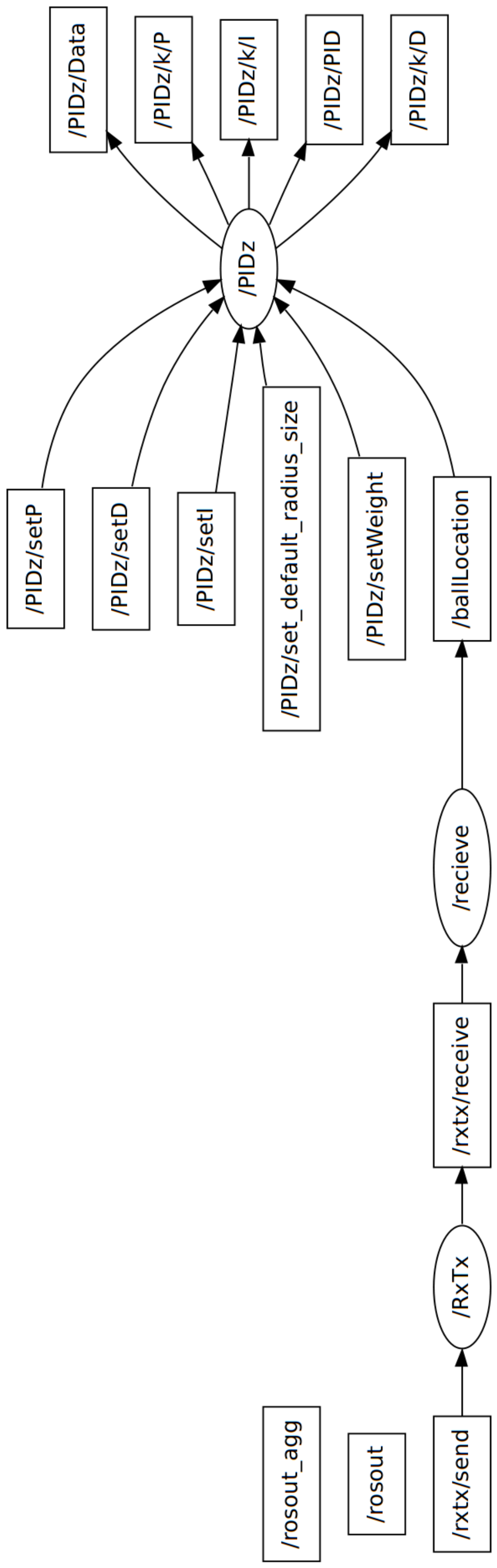

Figure C.3: Radius Subsystem

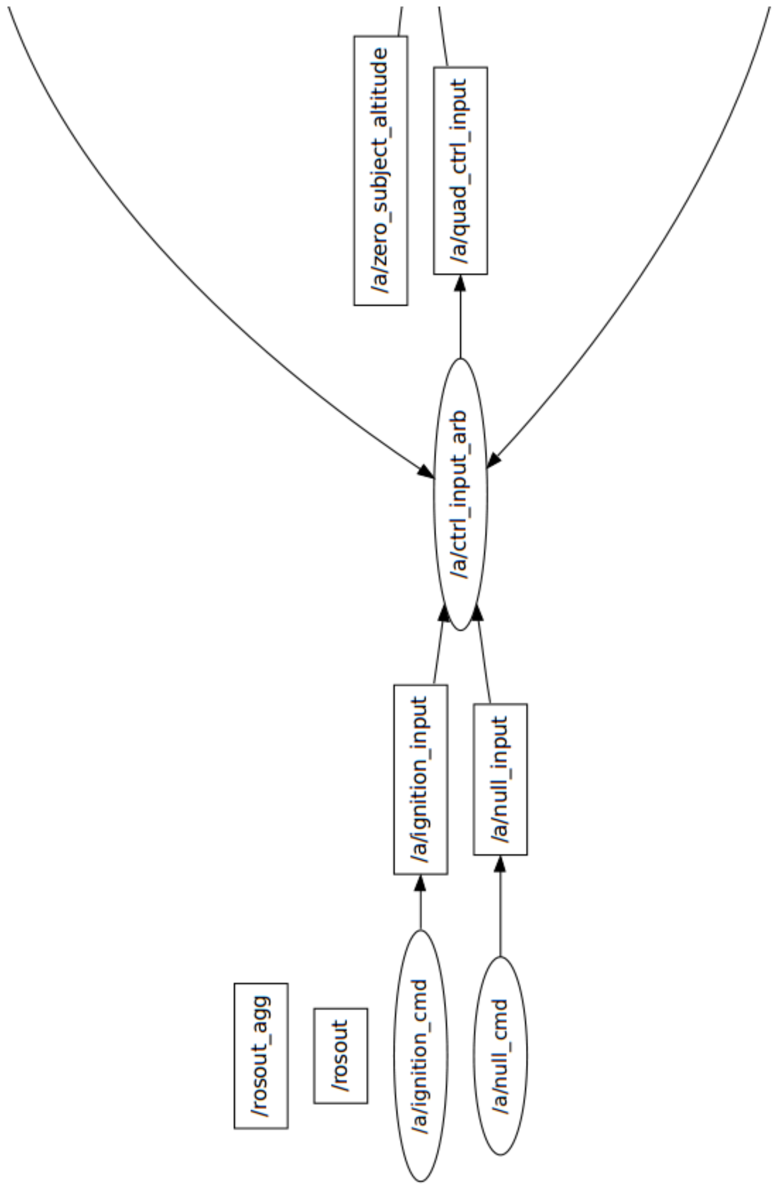

Figure C.4: Serial Communication Subsystem - Part 1

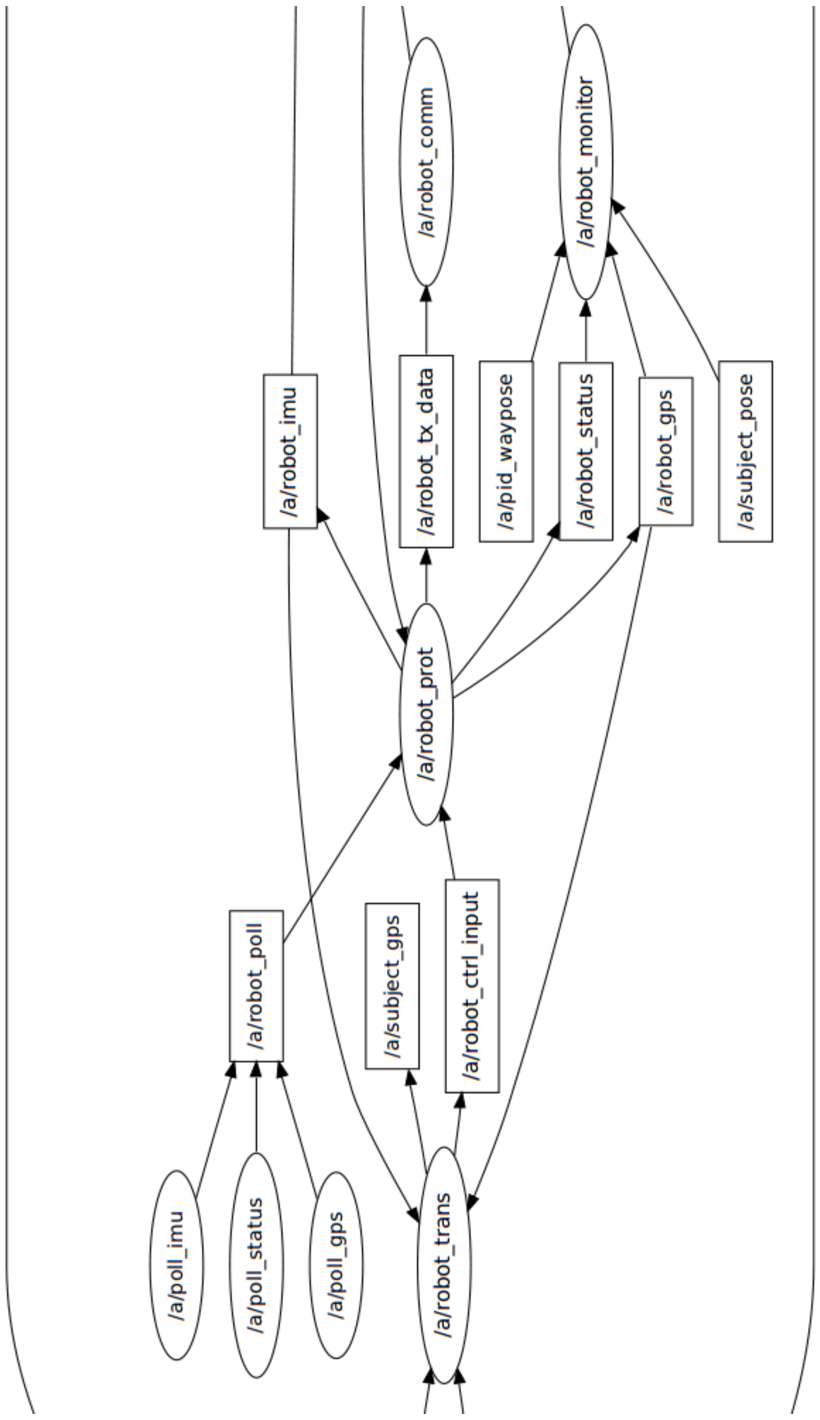

Figure C.5: Serial Communication Subsystem - Part 2

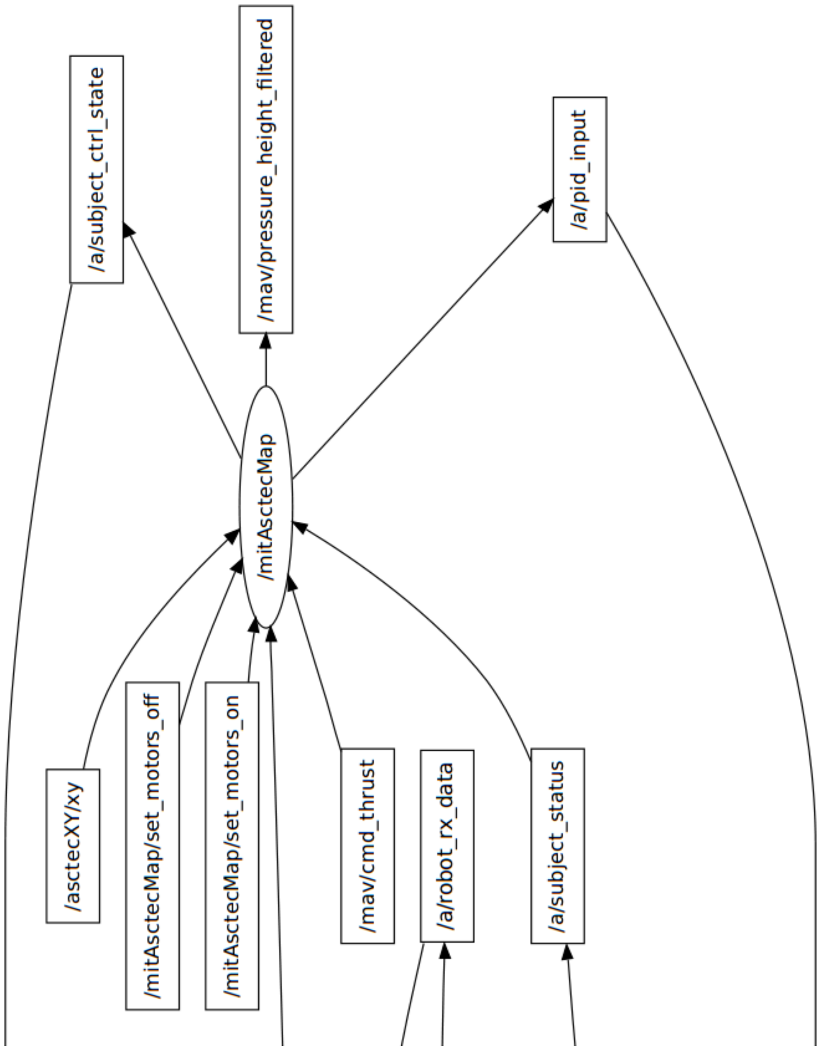

Figure C.6: Serial Communication Subsystem - Part 3

Bibliography

[1] Acroname garcia custom robot. http://www.acroname.com/garcia/garcia.html.

[2] AscTec hummingbird autopilot. http://www.asctec.de/asctec-hummingbird-autopilot-5/.

[3] JSON javascript object notation. http://www.json.org/.

[4] Ros. http://www.ros.org.

[5] *Probability & Statistics for Engineers & Scientists*. Pearson Education, 2007.

[6] Naoki Abe and Manfred K. Warmuth. On the computational complexity of approximating distributions by probabilistic automata. In *Proceedings of the third annual workshop on Computational learning theory*, COLT '90, pages 52–66, San Francisco, CA, USA, 1990. Morgan Kaufmann Publishers Inc.

[7] Chris Allan, Pavel Avgustinov, Aske Simon Christensen, Laurie Hendren, Sascha Kuzins, Ondřej Lhoták, Oege de Moor, Damien Sereni, Ganesh Sittampalam, and Julian Tibble. Adding trace matching with free variables to AspectJ. *SIGPLAN Not.*, 40(10):345–364, October 2005.

[8] Glenn Ammons, Rastislav Bodík, and James R. Larus. Mining specifications. In *Proceedings of the 29th ACM SIGPLAN-SIGACT symposium on Principles*

of programming languages, POPL '02, pages 4–16, New York, NY, USA, 2002. ACM.

[9] Adnan Aziz, Kumud Sanwal, Vigyan Singhal, and Robert Brayton. Verifying continuous time markov chains. pages 269–276. Springer, 1996.

[10] Adnan Aziz, Vigyan Singhal, Felice Balarin, Robert K. Brayton, and Alberto L. Sangiovanni-vincentelli. It usually works: The temporal logic of stochastic systems. pages 155–165. Springer, 1995.

[11] Christel Baier, Joost pieter Katoen, and Holger Hermanns. Approximate symbolic model checking of continuous-time markov chains (extended abstract), 1999.

[12] A. W. Biermann and J. A. Feldman. On the synthesis of finite-state machines from samples of their behavior. *IEEE Trans. Comput.*, 21(6):592–597, June 1972.

[13] Eric Bodden. Specifying and exploiting advice-execution ordering using dependency state machines. In *International Workshop on the Foundations of Aspect-Oriented Languages (FOAL)*, March 2010.

[14] P. Caspi, D. Pilaud, N. Halbwachs, and J. A. Plaice. Lustre: a declarative language for real-time programming. In *Proceedings of the 14th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, POPL '87, pages 178–188, New York, NY, USA, 1987. ACM.

[15] Feng Chen, Marcelo d'Amorim, and Grigore Roşu. Checking and correcting behaviors of Java programs at runtime with Java-MOP. *Electron. Notes Theor. Comput. Sci.*, 144(4):3–20, May 2006.

[16] Feng Chen and Grigore Roşu. Mop: an efficient and generic runtime verification framework. *SIGPLAN Not.*, 42(10):569–588, October 2007.

[17] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and J. F. Quesada. Maude: specification and programming in rewriting logic. *Theor. Comput. Sci.*, 285(2):187–243, August 2002.

[18] Christoph Csallner, Nikolai Tillmann, and Yannis Smaragdakis. DySy: dynamic symbolic execution for invariant inference. In *Proceedings of the 30th international conference on Software engineering*, ICSE '08, pages 281–290, New York, NY, USA, 2008. ACM.

[19] Matthew B. Dwyer, George S. Avrunin, and James C. Corbett. Patterns in property specifications for finite-state verification. In *Proceedings of the 21st international conference on Software engineering*, ICSE '99, pages 411–420, New York, NY, USA, 1999. ACM.

[20] E. Allen Emerson. Handbook of theoretical computer science (vol. b). chapter Temporal and modal logic, pages 995–1072. MIT Press, Cambridge, MA, USA, 1990.

[21] Michael D. Ernst, Jake Cockrell, William G. Griswold, and David Notkin. Dynamically discovering likely program invariants to support program evolution. In *Proceedings of the 21st international conference on Software engineering*, ICSE '99, pages 213–224, New York, NY, USA, 1999. ACM.

[22] Mark Gabel and Zhendong Su. Javert: fully automatic mining of general temporal properties from dynamic traces. In *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of software engineering*, SIGSOFT '08/FSE-16, pages 339–349, New York, NY, USA, 2008. ACM.

[23] Mark Gabel and Zhendong Su. Online inference and enforcement of temporal properties. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 1*, ICSE '10, pages 15–24, New York, NY, USA, 2010. ACM.

[24] Lars Grunske. Specification patterns for probabilistic quality properties. In *Proceedings of the 30th international conference on Software engineering*, ICSE '08, pages 31–40, New York, NY, USA, 2008. ACM.

[25] Sudheendra Hangal and Monica S. Lam. Tracking down software bugs using automatic anomaly detection. In *Proceedings of the 24th International Conference on Software Engineering*, ICSE '02, pages 291–301, New York, NY, USA, 2002. ACM.

[26] Hans Hansson and Bengt Jonsson. A logic for reasoning about time and reliability. *Formal Aspects of Computing*, 6:512–535, 1994. 10.1007/BF01211866.

[27] Klaus Havelund and Grigore Roşu. An overview of the runtime verification tool Java PathExplorer. *Form. Methods Syst. Des.*, 24(2):189–215, March 2004.

[28] Gerard J. Holzmann. The logic of bugs. In *Proceedings of the 10th ACM SIGSOFT symposium on Foundations of software engineering*, SIGSOFT '02/FSE-10, pages 81–87, New York, NY, USA, 2002. ACM.

[29] Michael Kearns, Yishay Mansour, Dana Ron, Ronitt Rubinfeld, Robert E. Schapire, and Linda Sellie. On the learnability of discrete distributions. In *Proceedings of the twenty-sixth annual ACM symposium on Theory of computing*, STOC '94, pages 273–282, New York, NY, USA, 1994. ACM.

[30] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William G. Griswold. An overview of AspectJ. In *Proceedings of the 15th European Conference on Object-Oriented Programming*, ECOOP '01, pages 327–353, London, UK, UK, 2001. Springer-Verlag.

[31] Fred Kröger and Stephan Merz. *Temporal Logic and State Systems (Texts in Theoretical Computer Science. An EATCS Series)*. Springer Publishing Company, Incorporated, 1 edition, 2008.

[32] Ingolf H. Krüger, Gunny Lee, and Michael Meisinger. Automating software architecture exploration with m2aspects. In *Proceedings of the 2006 international workshop on Scenarios and state machines: models, algorithms, and tools*, SCESM '06, pages 51–58, New York, NY, USA, 2006. ACM.

[33] Leslie Lamport. The temporal logic of actions. *ACM Trans. Program. Lang. Syst.*, 16(3):872–923, May 1994.

[34] Gary T. Leavens. Tutorial on JML, the Java modeling language. In *Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering*, ASE '07, pages 573–573, New York, NY, USA, 2007. ACM.

[35] David Lo, Siau-Cheng Khoo, and Chao Liu. Mining temporal rules for software maintenance. *J. Softw. Maint. Evol.*, 20(4):227–247, July 2008.

[36] David Lo, Leonardo Mariani, and Mauro Pezzè. Automatic steering of behavioral model inference. In *Proceedings of the the 7th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*, ESEC/FSE '09, pages 345–354, New York, NY, USA, 2009. ACM.

[37] Zohar Manna and Amir Pnueli. *The temporal logic of reactive and concurrent systems*. Springer-Verlag New York, Inc., New York, NY, USA, 1992.

[38] Amir Pnueli. The temporal logic of programs. In *Proceedings of the 18th Annual Symposium on Foundations of Computer Science*, pages 46–57, Washington, DC, USA, 1977. IEEE Computer Society.

[39] William H. Press, Saul A. Teukolsky, William T. Vetterling, and Brian P. Flannery. *Numerical Recipes 3rd Edition: The Art of Scientific Computing*. Cambridge University Press, New York, NY, USA, 3 edition, 2007.

[40] John Whaley, Michael C. Martin, and Monica S. Lam. Automatic extraction of object-oriented component interfaces. In *Proceedings of the 2002 ACM SIGSOFT international symposium on Software testing and analysis*, ISSTA ’02, pages 218–228, New York, NY, USA, 2002. ACM.

[41] Jinlin Yang, David Evans, Deepali Bhardwaj, Thirumalesh Bhat, and Manuvir Das. Perracotta: mining temporal API rules from imperfect traces. In *Proceedings of the 28th international conference on Software engineering*, ICSE ’06, pages 282–291, New York, NY, USA, 2006. ACM.

[42] Wlodzimierz M. Zuberek. Performance evaluation using extended petri nets. In *International Workshop on Timed Petri Nets*, pages 272–278, Washington, DC, USA, 1985. IEEE Computer Society.