Probabilistic Conditional System Invariant Generation with Bayesian Inference

Anonymous Author(s)*

ABSTRACT

Invariants are a set of properties over program attributes that are expected to be true during the execution of a program. Since developing those invariants manually can be costly and challenging, there are a myriad of approaches that support automated mining of likely invariants from sources such as program traces. Existing approaches, however, are not equipped to capture the rich states that condition the behavior of autonomous mobile robots, or to manage the uncertainty associated with many variables in these systems. This means that valuable invariants that appear only under specific states remain uncovered. In this work we introduce an approach to infer conditional probabilistic invariants to assist in the characterization of the behavior of such rich stateful, stochastic systems. These probabilistic invariants take the form $P(Outcome|Givens)$, can encode a family of predefined patterns in either term, are generated using Bayesian inference to leverage observed trace data against priors gleaned from previous experience and expert knowledge, and are ranked based on their surprise value and information content. Our studies on two semi-autonomous mobile robotic systems show how the proposed approach is able to generate valuable and previously hidden stateful invariants.

KEYWORDS

invariant generation, bayesian inferenece, autonomous systems

ACM Reference Format:
Anonymous Author(s). 2020. Probabilistic Conditional System Invariant Generation with Bayesian Inference. In *ISSTA '20: International Symposium on Software Testing and Analysis, July 18–22, 2020, Los Angeles, U.S.A..* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/1122445.1122456

1 INTRODUCTION

Our community has built a large body of work on likely invariant generation from system traces. This body includes the inference of invariants of different types [23], from those attempting to characterize a variable range of values [4, 5, 13, 17, 22] to those infering temporal invariants [2, 11, 19–21, 26, 28], and utilizes a variety of approaches ranging from frequentist inference [5] to the generation of polynomial relations [22] to k-tail [20] to deep learning [19]. As we explored this body of work for its application to semi-autonomous mobile robots, however, we came to realize that these kinds of systems introduce a couple of unique attributes that existing invariant generation approaches were unable to fully capture.

Figure 1: Drone ISR scenario.

The first unique attribute is the extent to which different system states render distinct sets of invariants, as the behavior of these systems is conditioned not so much by typical programmatic structures (i.e., functions pre- and post-conditions), but rather by particular system states. For example, the sensors activated and the attitude of a drone is conditioned by different mission states such as takeoff, approaching a target, or tracking a target, while a self-driving vehicle's linear velocity bounds may change depending on whether the car is charging, parking, driving within a city, or driving on a highway. We argue and later show that ignoring this attribute greatly limits the potential of uncovering valuable invariants that only appeared under certain states.

The second distinctive attribute is the degree of uncertainty intrinsic to these systems. They may render different results under the same environmental conditions due to sensor noise, imperfect estimators, inaccurate actuators, and humans in the loop. Sensors like the onboard GPS for the Parrot BeBop 2 drone are ±3.0 meters[1], actuators like those associated with the PX4 flight stack are often capped below their max to avoid actuator saturation[2], and humans operators have a wide variety of reaction times. We argue and later show that failing to handle this attribute properly will make it difficult to judge the value of an inferred invariant.

The state of the art, however, does not support the generation of invariants that are conditioned by arbitrarily complex system states, nor does it support probabilistic invariants to better characterize the uncertainty associated with the exposed behaviors. The closest related work considers having two outcomes happening jointly (not conditionally) and ignores the prior probabilities by making assumptions about the data distribution [5].

In this work we address this challenge by building on the statistical structure of conditional probabilities, $P(Outcome|Givens)$, to

[1]https://www.parrot.com/global/support/products/parrot-bebop-2/faq-bebop-2
[2]https://dev.px4.io/v1.9.0/en/concept/mixing.html

uncover likely invariants that only manifest under particular program states. In doing so we aim to fulfill four requirements to make the approach practical. First, provided a high level specification of the potential variables to explore as part of the *Outcome* and *Givens*, the approach must systematically investigate the potential of relevant predicates on those variables as part of the *Outcomes* that are only likely under specific predicates on those variables as part of the *Givens*. Second, the approach must be able to uncover valuable conditional invariants without overfitting. Overfitting presents a challenge in that the addition of predicates into the *Givens* may render invariants that hold with high probability but are applicable to a very small number of instances. Third, the approach must be able to leverage prior knowledge as it becomes available, either from a trace or from a developer, to improve the probability estimates, without incurring in the cost of recomputing all invariants when new data is added. Fourth, the approach must avoid relying on arbitrary thresholds to determine what is and is not significant as the choice of such thresholds as highly dependent on the context.

To address the first requirement, we define a family of initial relevant predicates patterns for the robotics domain and a Bayesian invariant inference engine that implements conditional inference, and implement a domain-specific specification language and a tool pipeline to compute them. To address the second requirement, we incorporate a ranking mechanism that judges value based on how much an invariant probability changed from prior estimates to posterior findings, and use an information content metric to select an invariant per outcome that offers best fit with the least parameters. To address the third and fourth requirement, and also to further support the first, we shift the inference model from using the classical (frequentist) statistics employed by existing approaches [5, 13, 17, 28], to a Bayesian inference model that allows us to easily incorporate prior information from previous traces or developer's knowledge, and does not require the definition of arbitrary thesholds or the reliance on data distribution assumptions.

The contributions of this work are:

- Approach to infer a family of conditional invariants patterns from traces through Bayesian inference.
- Implementation that provides the mechanisms to specify the space of variable predicates to explore, launch the inference engine to systematically explore that space, and ranked the solutions based on surprise and content.[3]
- Assessment of the proposed approach through its application to two systems, a reconnaissance drone and a semi-autonomous simulated car. The findings indicate that the approach can uncover valuable invariants that cannot be generated by existing approaches.[4]

2 MOTIVATION

Consider a drone performing a surveillance mission over a field whose objective is to locate and confirm a target. Figure 1 shows a downsized version of this real-world scenario where a micro-drone is sweeping for QR code targets inside of a user-defined area.

This drone system contains a rich set of states that influence system behavior. Some of its states may be defined by the mission

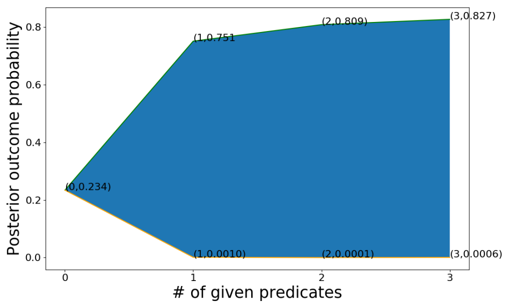

Figure 2: Maximum and minimum possible probabilities for Outcome predicate *x-velocityHigh* across number of predicates in Givens.

phase, such as $Machine = Hovering$, or by the level of critical resources such as $Warning = LowBattery$. Many of such states are encoded by design through specifications such as "When battery is low, drone must land". However, not all system states are explicitly specified or defined in the code.For example, ranges over the speed of the drone define states that affect the system behavior, such as the ability to recognize and track obstacles. To adequately characterize these systems, we must consider how such explicit and implicit states condition the system behavior.

Some of these states are tightly coupled with sources of stochasticism, such as mission states determined via operator-issued *UserCommands*, *Warning* states precipitated by a loss of localization connectivity, or speed-range states affected by a flight controller cap on acceptable oscillations. Furthermore, such stochasticity may be associated to the particular state. For example, assuming that speed impacts the noise of the drone's target sensors, the range of speeds while sweeping an area correlates to the drone's ability to find targets. While it may be correct to report that a drone's speed was between 0.0 and 1.0 whenever a target was detected, a probability distribution across those values gives a more granular view of detection success based on speed. These stochastic attributes point to the need to express these conditional invariants probabilistically.

Existing techniques [5, 11] are equipped to capture invariants such as *0.0 < drone Speed < 1.0* and *MissionState=PossibleTargetDetected* as preconditions when the drone enters its subroutine to query the user for how to adjudicate the target, or temporal invariants such as $\square(Sweeping) \rightarrow \diamond(PossibleTargetDetected)$. To determine these invariants, existing approaches might look at the parameters and return values of the "target detected" subroutine or other forms of program points, but would not necessarily inspect system variables that are not directly tied to the input/output of that subroutine, such as velocity or warning states. Moreover, existing approaches would capture the value of those variables, but not necessarily the probability of those values, much less the conditional probability of those values. As a result, existing approaches do not depict a sufficiently rich picture of the state space, especially of stochastic systems like the ones we are targeting.

While there is obvious value to invariants such as "a drone might recognize a target at velocities between 0.0 and 1.0 with 95% probability", existing approaches would not find that the drone had a much higher detection rate when velocity was closer to zero. Applied to this drone scenario, our approach not only indicates the

[3]github.com/anonymized/bayesInference/tool
[4]github.com/anonymized/bayesInference/data

range of speed values given a target has been detected, but also the probability of each range of speed values. To compute (and subsequently update) the probabilistic distributions associated with the outcomes at various system states, we use Bayesian inference to determine the conditional probability relating them.

Another way to understand the potential of our proposed approach is illustrated by Figure 2, which shows the probabilities for invariants of the form P(*x-velocityHigh*|*Given*$_1$, *Given*$_2$, ..., *Given*$_n$) where the x-axis displays the number predicates in the given. As shown by the upper bound of the blue area, selectively and incrementally conditioning the space by incorporating more predicates into the Givens can render invariants with higher probability. Increasing the number of predicates to consider, however, also increases the computation cost exponentially and there is a point of diminishing return where the invariants have higher probability but overfit the data. These tradeoffs between value and cost is one of the driving motivations for this work and one that studied later.

3 RELATED WORK

Ernst et. al. [5] established Daikon, one of the benchmark inference engines for detecting program invariants. Daikon's engine creates a field of potential invariants based on a set of predefined invariant templates and the values found in a trace. We follow a similar approach in that our predicates are basic patterns, but we incorporate then in the richer conditional probability structure. Daikon then evaluates the potential invariants according to whether there are sufficient samples to support them and no samples that violate them. This frequentist approach, prevalent among invariant inference engines, uses a confidence interval to ascertain that a predicate holds against some probability of random negation, determined by the number of samples supporting that predicate. One drawback of their frequentist approach is that, while new traces can be added to an existing sufficiently large set of samples, there is a significant cost associated with the trace accumulation and algorithmic complexity needed in order to reach that sufficiently large set without relaxing the confidence interval. Daikon does not compute conditional invariants, just joint probabilities, and does not support multiple predicates.

Jiang et al. [17, 18] extend the Daikon invariant library to patterns seen in robotic systems in order to derive monitors that can check system properties at runtime. While Jiang et al. introduce invariant templates tailored to robotic systems (e.g., bounded time differentials, polygonal relationships between spatial variables), their approach still relies on a traditional frequentist approach and the use of confidence intervals to retain viable instantiations of those templates for a set of traces. Aliabadi et al. [1] present a similar approach for cyberphysical system security.

Grunkse [12] focuses on probabilistic invariants as a qualitative expression of requirements. He introduces a rich set of specification patterns coupled with a structured english grammar to express bounded probabilistic behavior of a system, instead of in terms of absolute correctness, to incorporate expert knowledge and to sllow for it to be used for formal verification. Although this work did not pursue automated inference of invariants, its treatment of probabilistic patterns offers a roadmap for us to expand our work.

Perracotta [28] extracts temporal API specifications from traces through a mix of analysis, patterns, and heuristics. There have been many similar approaches since, but Perracota was among the first to recognize that traces, or trace content for that matter, can be noisy, so it incorporated mechanisms to ignore potential blips of aberrant behavior patterns as negligible in the context of the overall trend. Our search for probabilistic invariant generation is inspired by these kinds of challenges. In a similar line of work, Gabel et al. [11] developed a mining framework of temporal logic properties. Their approach is similar to many approaches in terms of combining patterns and incrementally encoding them as FSMs. However, their strategy to start with simple patterns that can be composed to generate much more complex ones is one that we have adopted in our approach.

We note that none of the approaches, although closely aligned with ours, produce the conditional probabilistic invariants with Bayesian inference that we are pursuing.

4 APPROACH

The goal of the proposed approach is to generate invariants that capture the probabilistic influence between system events as in $P(A|B)$. The next sections describe how, by building on conditional probabilities and Bayesian statistical inference, we can process system traces to produce invariants that meet that goal.

4.1 Overview

Figure 3 provides a high-level diagram of the approach. The approach takes three parameters: a trace, a file of invariant specifications processable by the grammar, and a set of prior distributions. The trace consists of a series of time-stamped variable-value pairs, the invariant specifications define the space of outcome and given predicates worth exploring as part of a conditional distribution $P(A|B)$, and the prior distribution $P(A)$ is a set of prior probabilities which reflect whatever knowledge we have on the outcome. Data wrangling is a preprocessing step that consists of trace interpolation and file checking, and configuration provides the inference engine with a space of predicates to explore. The core component of the approach is the inference engine, which utilizes Bayesian inference to compute conditional probabilities based on the predicate space and information in the trace. The engine produces such probability as per the Bayesian formula $P(A \mid B) = \frac{P(B|A)P(A)}{P(B)}$, where $P(B \mid A)$ and $P(B)$ are computed based on the trace. $P(B \mid A)$ tells us how likely is A to support B, and $P(B)$ is the total probability B conditioned upon outcome variable A. The generated conditional probabilities are then ranked based on the change from the prior probability $P(A)$ to the computed posterior $P(A \mid B)$, which reflects the chance of uncovering an overlooked invariants.

4.2 Invariant Specification Pattern

We capture the probabilistic influence between two dependent events as conditional probabilities of the form $P(A|B)$, generalized as the probability of A given B, *where A and B are boolean predicates evaluated over single or multiple states of a system.* B is commonly referred to as the "given" predicate, which defines the subspace where the probability of the A predicate is considered, and A as the outcome. This simple conditional pattern belies the richness of invariants it can encode, as these predicates can take many forms

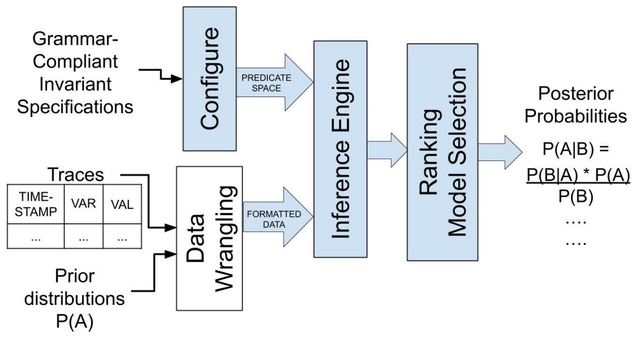

Figure 3: Approach Overview.

and can be composed to form arbitrarily complex descriptions of variable states.

In its simplest form, if B is TRUE, then $P(A \mid B) = P(A)$. This implies that any existing invariant pattern developed in the related work has the potential to be subsumed by the proposed invariant encoding. That includes from the simplest form of state invariants such as $battery > 0$ to complex metric temporal logic formulas such as $\square(Takeoff) \implies \diamond_{[0,3]}(altitude > 0))$. In its richest form and the focus of this effort, $P(A \mid B)$ lets us explore how the system behavior encoded in B could influence other parts of the behavior encoded in A. For example, continuing with the drone scenario described in the motivation, given $A : TargetDetected = TRUE$ and $B : VelocityHigh = TRUE$, we could consider the conditional probability *P(TargetDetected | VelocityHigh)* [5], which describes the probability of a drone sensor detecting features of a target, given that the drone is moving at high speeds.

The family of specification patterns we currently support is guided in part by the needs we observed in the systems we have developed and studied, and includes three basic types of pattern predicates: *Equality*, *Range*, and *Trend*. *Equality* predicates are of the form *var EqOP const*, where variable type can be $int|float|string|bool$, and *EqOP: == | !=*. This type of predicate encompasses earlier examples such as *TargetDetected=TRUE*, *Speed=0* and *DroneState=Hovering*. For non-integer *Equality* variables, we support fuzzy predicates with the addition of a threshold such that $variable\ EqOP\ const \pm \delta$, such as $Acceleration = 9.8 \pm 0.01$. Additionally, we can have a disjunction of *Equality* over a variable, such as *DroneState=Hovering* ∨ *DroneState=Translating*. This kind of predicate is effective at capturing explicit conditional states such as those embedded in the state machine of a system.

Range predicates are of the form *var OP const*, with $OP : |<|>|>= |<=$, and include conjunctions and disjunctions. *Range* predicates can capture implicit states encoded in variables values. For example, a variable *Latency* has values that can be partitioned indicating a fast response $Latency < 10$, a medium response $Latency \geq 10 \wedge Latency < 20$, or a slow response $Latency \geq 20$, and the system behavior may be conditioned differently across those ranges.

Trend predicates are different in that they involve state sequences, which is particularly valuable to capture tendencies over time. These predicates apply a function to a sequence of values within a configured number of timesteps, referred to here as a window. *Trend* predicates take the form of $f(var[window])\ OP\ const$. The function f could compute an average over the window, but it can also be more complex. For example, one f we use in our implementation calculates the derivative of the best fit quadratic polynomial function of the values in this series. Given a *window*, the predicate checks whether the derivative of a variable has increased in that window: $dvar_w > 0$, has decreased: $dvar_w < 0$, or remains constant: $dvar_w = 0$. For example, a predicate on the variable $TrustHumanOnSystem$ could check whether it is increasing, decreasing, or remains unchanged. Similarly, for whether a car is in *autonomous* or *manual* mode, the *changeMode* variable could encode the direction in which the mode changed.

The pattern specification grammar we built for developers to specify the space of predicates to explore is described in Section 4.5. For simplicity, the examples in this section present predicate expressions over single variables as *P(A | B)*. However, predicates may be composed into conjunctions to define more complex nested models such as *P(A | B, C, ...)*. For example, we can condition *MissionState=PossibleTargetDetected* upon *x-velocityHigh*, or upon *x-velocityHigh* ∧ *MachineState=Hover*. Compound predicates and model complexity are addressed in more detail in Section 4.4.

4.3 Inferring Invariants

Algorithm 1 shows the key steps in the inference process for two predicates A and B in $P(A|B)$. Given a trace, two predicates A and B, and a prior distribution for A, the algorithm starts by processing each record in the trace, and evaluating the predicates on the appropriate trace variables. Such evaluation is performed according to the type of predicate to render a *TRUE* if the predicate holds. If predicates A or B hold, then their corresponding frequency count is updated, and if both of them hold then their joint probability is also updated. Once the trace is processed, these frequency counts are used to compute the probabilities required by Bayesian inference: the total probability P(B), P($B \mid A$), and finally P($A \mid B$).

In practice, two aspects of the algorithm acquire additional complexity. First, the *eval* function must deal with predicates that require processing multiple trace records concurrently, peeking backward and forward in the trace to evaluate *trend* predicates. Second, compound predicates impose additional frequency tracking, and extended functions to compute the probabilities as they require to iterate over a larger number of combinations of predicates. We now illustrate these challenges through an example.

Consider the brief sample trace in Table 1. Let us assume a developer is interested in just exploring predicates $A : MachineState = PossibleTargetDetected$ and B : *y-velocityChange* < 0, where the y-velocityChange window size is 3 timesteps and its ranges are *y-velocity* < 0, *y-velocity* ≥ 0. [6] The developer also provides a $MachineState = PossibleTargetDetected$ prior of 0.3.

Algorithm 1 processes the trace to compute the frequency of $MachineState = PossibleTargetDetected$ and of the likelihood $MachineState = PossibleTargetDetected$ given *y-velocityChange* $<$ 0. Since the y-velocity window is 3 and ignoring windows extending outside the shown trace, the algorithm counts two instances where y-velocity decreases: from times 1-3 and from times 2-4. Checking

[5] For readability, we simplify the predicates that check a boolean variable by just specifying the variable name

[6] In practice, a developer may specify a much larger number of predicates to explore, and likely without constraining it to be part of just the outcome or the given.

```
Algorithm 1: Single Inference
   Input: trace, $A$, $B$, P($A$)
   Output: P($A|B$)
1  foreach record in trace do
2      if eval(A) then
3          $freqA \leftarrow freqA + 1$;
4          if eval(B) then
5              $freqAandB \leftarrow freqAandB + 1$;
6          end
7      end
8  end
9  P($B$ | $A$) ← $freqAandB$ / $freqA$ ;
10 P($B$) ← P($B$ | $A$) * P($A$) + P($B$ | ¬$A$) * P(¬$A$) ;
11 P($A$ | $B$) ← P($B$|$A$) * P($A$) + P($B$);
```

Table 1: Sample trace from the drone scenario.

Time	MachineState	y-velocity
1	Sweeping	0.1
2	Sweeping	0.2
3	Sweeping	0.05
4	PossibleTargetDetected	0.0
5	PossibleTargetDetected	0.1

the anchor index of these windows, times 3 and 4 have an instance of *MachineState=PossibleTargetDetected*. So, the frequency counts after processing the trace are: *MachineState=PossibleTargetDetected*: count = 2, $y\text{-}velocityChange < 0$: count = 2, and *y-velocityChange<0* | *MachineState=PossibleTargetDetected*: count = 1.

After the trace is processed, $P(B)$ is calculated as per the law of total probability, which is generally defined as such for a discrete set of all evaluations of predicate A: $P(B) = \sum_{i=1}^{n} P(B \mid A_i)P(A_i)$, where A_i denotes a value of predicate A and $P(A_i)$ is the prior probability of A_i. For now, since we are working on just a single predicate A, A_1 corresponds to $MachineState = PossibleTargetDetected$, and we use the complement *¬MachineState=PossibleTargetDetected* as A_2 to compute the total probability. If another predicate like $MachineState = Landing$ was defined by the developer, then that would constitute the new A_2, and the complement of those predicates' conjunction would be A_3.

We are computing $P(MachineState = PossibleTargetDetected \mid y\text{-}velocityChange < 0)$, so we would only need to calculate $P(y\text{-}velocityChange < 0 \mid MachineState = PossibleTargetDetected) + P(y\text{-}velocityChange < 0 \mid \neg MachineState = PossibleTargetDetected)$. Since the prior of $MachineState = PossibleTargetDetected$ is 0.3 and the prior of $\neg MachineState = PossibleTargetDetected$ is therefore 0.7, we have $\frac{1}{2} * 0.3 + \frac{1}{3} * 0.7 = 0.383$ as the total probability of $MachineState = PossibleTargetDetected$. The prior, likelihood, and total probabilities are then used to calculate the Bayesian probabilities of each generated invariant. Then, $P(MachineState{=}PossibleTargetDetected \mid y\text{-}velocityChange < 0)$ is $\frac{\frac{1}{2}*0.3}{0.383} = 0.392$.

Note that in this illustrating example we focus on predicates with expressions on just single variables. The above example easily generalizes to handle multiple given predicates, where the algorithm checks for the coincidence of *Outcome* and multiple *Given* predicates instead of a single one. In order to calculate $P(A|B,C)$, $freqAandBandC$ would have to be computed to produce $P(B,C|A)$ on line 9 in Algorithm 1, and line 10 would become *P(B, C) ← P(B, C | A) * P(A) + P(B, C | ¬A) * P(¬A).*

4.4 Ranking and Model Selection

As the potential space of predicates to explore grows, so does the number of invariants the approach must evaluate, and the ones a developer must analyze. As such, we found it essential to integrate mechanisms to highlight invariants that may surprise the developer while taking into consideration the invariant complexity.

The first mechanism is based on the surprise ratio [15], expressed as the ratio of posterior likelihood over prior probability of the outcome, expressed $\frac{P(A|B)}{P_{prior}(A)}$. Ranking by this ratio allows for the prioritization of invariants that show the greatest potential to change the likelihood of an outcome, and hence to highlight invariants that may have been hidden before conditioning was used.

The second mechanism is based on the Bayesian information criterion (BIC) [27]. This metric maximizes log likelihood while penalizing overfitting of the model through the formula $BIC = ln(n)k - 2ln(\hat{L})$, where n=number of observations, k=number of model parameters, and $\hat{L}$=maximum log likelihood. Given two invariants with the same outcome, BIC is particularly useful to determine whether an invariant with more predicates is worth it. For example, if *P(WarningState=BatteryLow | x-velocityHigh)* contains *P(WarningState=BatteryLow | x-velocityHigh y-velocityHigh)* and *P(WarningState=BatteryLow | x-velocityHigh MissionState=Complete)* and the nested models with more predicates show little fluctuation in the resulting likelihood of *WarningState=BatteryLow*, then the BIC will assign a more favorable score to the smallest model of the three, i.e. the model at the topmost nesting level.

4.5 Implementation

The implementation generalizes Algorithm 1 to accomodate multiple predicates, perfoming additional bookkeeping to optimize the evaluation of those predicates, requiring only one traversal of the trace. The implementation uses the Apache Commons Lang [8] and Apache Commons Math [7] packages to process original and intermediate trace files. The implementation also defines a language for developers to specify the space of predicates worth exploring, which is summarized by the next production rules. The grammar was implemented using the LALR-1 CUP parser generator [6] and has been abbreviated for clarity.

```
\<start> ::= OUTCOMES <pred-definition>* GIVENS <pred-definition>*
      CONSTRAINTS <constraint_definition>*
 <pred-definition> ::= <var name> ',' <type> ',' <threshold> ','
 <partitions> ',' <window>
 <constraint_definition> ::= 'P(' <var_name> '|' <var_name>* ')'
 <type> ::= `INT-Eq'| `DOUBLE-Eq'| `STRING-Eq'| `INT-Range'| `DOUBLE-
       Range'| `STRING-Range'| `INT-Trend'| `DOUBLE-Trend' | `STRING-
       Trend'
 <partitions> ::= <partitions> <exp> | EMPTY
 <exp> ::= <exp> ∧ <exp> | <exp> ∨ <exp> |
 <var name> <num-op> NUMBER | <var name> <string-op> STRING
 <num-op> ::= == | != | > | < | <= | >=
 <string-op> ::= == | !=
 <threshold> ::= NUMBER | EMPTY
 <window> ::= NUMBER | EMPTY
```

The engine can either use predicates as givens or outcomes as specified by the grammar, or combinatorically construct conjunctions of givens using any of the variable definitions. Constraints allow the engine to investigate only specific combinations of variables of interest. This enables developer-defined configurations of variables upon which outcomes should be conditioned, further reducing the possible combinations of variables. For example, if a config file defines the *Outcome* variables *x-velocity* and *y-velocity*, and the *Given* variables *MissionState* and *WarningState*, then adding the constraint *P(x-velocity | MissionState)* will only apply *MissionState Given* predicates to *x-velocity* predicates as defined in the constraints, as opposed to applying the predicates of all *Given* variables to the predicates of the *x-velocity Outcome* variable.

5 STUDY

The goal of this study is to better understand the value and cost of the generated conditional probabilistic invariants. More specifically, our research questions are:

- *What is the value-added of the generated conditional probabilistic invariants?* We judge value by ranking the invariants based on the surprise index, i.e. how much they changed from the prior, by interpreting the top generated invariants per outcome among those according to the lowest Bayesian information criteria, and by comparing the resulting sets of invariants against those generated by Daikon[7].
- *What is the cost of generating such invariants?* We assess cost in terms of the time to generate the invariants as a function of the number of predicates and trace length.

In the following sections, we first describe the setup of engine and scenarios, then address these two research questions.

5.1 Study Setup

To answer the research questions, we required systems that met three requirements. First, they had to exhibit stochastic and conditional behavior. Second, they had to be amenable to the proposed analysis in that they generated a trace and were accessible enough for us to interpret the findings. Third, they had to cover different domains in terms of the sources of uncertainty and type of systems to help us understand whether the results would generalize. We identified two systems and contexts that meet those criteria and are described in more detail in the next subsections: (1) a drone performing a reconnaissance mission, and (2) an autonomous driving car interacting with a human driver. Both systems and their executing scenarios were developed at [anonymized], they cover two distinct domains with different sources of uncertainty, with the ground system uncertainty caused primarily by the drivers and sensors, and the aerial system caused by the sensors.

For each system we prepared configuration files and converted their system traces into a standard format processable by the inference engine. As a general strategy, when building the configuration files we favored including all potentially interesting variables even when we could not clearly anticipate their relation to other variables. We were more conservative in defining ranges, favoring fewer ranges because smaller partitions of variable values were difficult to define meaningfully without significant empirical tuning. For example, the values of $reaction\_time$ were split into two ranges, $0 \leq reaction\_time < 7$ being fast and $reaction\_time \geq 7$ being slow. These ranges were determined through Jenks natural breaks optimization [16] with slight adjustments to the resulting breakpoints to create uniform intervals. Ranked invariants were evaluated by surprise ratio, calculated by comparison of the posterior over prior. Surprise ratios can be interpreted as the factor by which an outcome becomes more likely in a conditioned state. An invariant *P(x-velocityHigh | batteryLow)* with a surprise ratio of 2 indicates that the outcome *x-velocityHigh* is 2 times more likely when conditioned upon the given *batteryLow*.

Priors were calculated according to their frequency count in a subset of the traces for each system. The subset from which the priors were derived was disjoint from the subset of traces from which invariants were generated.

5.1.1 *Drone ISR.*

The drone scenario is designed to mimic a simple intelligence/surveillance/reconnaissance (ISR) mission. The drone scenario employs a DJI Tello [24] whose onboard computer is interfaced with a series of controllers implemented in (ROS) [10]. The drone navigates in a controlled indoor flying cage equipped with a Vicon localization system [25]. The ISR mission starts with the drone autonomously taking off at randomly defined "base" coordinates and approaching a predefined sweep area. Once the sweep area is reached, the drone sweeps for a "target" identifiable through a QR code. The waypoint navigation and sweep speed are adjusted by PID controllers. When a target is detected, the drone stops and queries the operator to confirm the target, which may require closer manual exploration. When a target is adjudicated to be the desired target, the drone returns to the home base coordinates. If the drone loses its localization services, it hovers and queries the user for a decision. The user may either request manual control to guide the drone back into the last localized position, in which case the user may return to autonomous control and the drone resumes sweeping where it left off, or the user may request an emergency landing. If the user does not respond within a set time, the drone performs an emergency landing. The targets were positioned outside of the localization area so, if the operator requested the drone to examine a target too closely, the drone would likely lose localization services.

A total of 34 runs containing 109 unique variables were collected using one operator and a variety of user inputs. Of those runs, 30 detected a possible target, 23 had the operator taking manual control, 20 had the operator issuing a closer examination command, 21 saw at least a partial loss of localization services, 20 saw a total loss of localization services, and 3 ended in emergency landings. 17 of these drone runs were randomly selected and their traces were used to compute the priors. Traces were captured using the ROS bagging [9], which produces a trace of timestamped variable-value pairs. Our data wrangling scripts transform those bags into traces that had a value for every variable at every timestep using interpolation and that were in the .csv format required by our implementation. Traces were on average 93.5s long per run. Priors traces comprised 49.0% of all traces used in this study.

[7]We note that Daikon does not consider prior probabilities, assumes certain thresholds and data distributions, and most important it only supports single-predicate joint probabilities (Daikon's mechanism to identify state partitions is called "conditional" splitting but it is really computing a "joint" probability between a variable holding a value and an invariant).

Table 2: Variables in the Drone Study.

Variable	Predicate Type	Meaning
UserCommand	String Equality	Command chosen by user from a predefined list. Can take values of: Default (No command), Hover, KeepSweeping, LookCloser, RequestAutoControl, RequestManualControl, ReturnHome, Land
MissionState	String Equality	Set of states describing mission status. Can take values of: Complete, InProgress, InsideSweepArea, OutsideSweepArea, PossibleTargetDetected, Suspended
MachineState	String Equality	Set of states describing drone status. Can take values of: FinishedBehavior, Hovering, Landing, LosingVicon, Manual, OutsideSweepArea, PossibleTargetDetected, Sweeping
x-velocity	Float Range	Magnitude of the x-velocity in m/s. Range: 0-1.0
y-velocity	Float Range	Magnitude of the x-velocity in m/s. Range: 0-1.0
reaction_time	Float Range	Time in seconds user took to give a command after being prompted. Range: 0.0-14.0
sensor.status	Integer Equality	Values target sensor takes depending on what it detects. 1=no target detected, 2=sensor ready to detect, 3=full target detected, 4=partial target detected

Table 3: Variables in the Autonomous Car Study.

Variable	Predicate Type	Meaning
Mode	String Equality	Driving mode that the simulated car is in. Can take values of: Autonomous, Manual
WheelChange	Float Trend	Rate of change of wheel angle of the simulated car per second in degrees. Range: -360−360
Throttle	Float Range	Throttle applied to the simulated car in percentage. Range: 0-100
Brake	Float Range	Braking pressure applied to the simulated car in percentage. Range: 0−100
VelocityChange	Float Trend	Rate of change of the velocity of the simulated car per second in m/s^2. Range: -7−5
Event	String Equality	Event detected by the sensor of the simulated car. Can take values of Pedestrian, Obstacle, Truck, Cyclist, False Alarm, None (None indicates nothing is detected by the sensor)
TrustChange	Integer Trend	Change of trust level towards the autonomous driving system. Can take values of -5, -4, -3, -2, -1, 0, 1, 2, 3, 4, 5

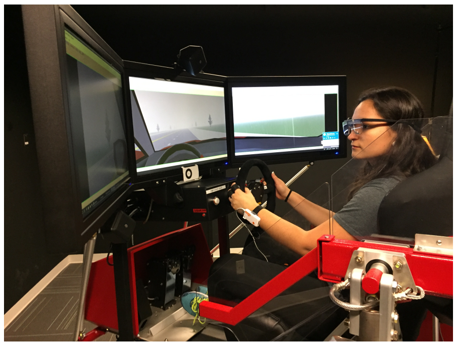

Figure 4: Autonomous driving scenario.

A total of 30 unique predicates were identified to comprise the outcomes and the givens to generate 72,347 invariants. Table 2 lists the variables appearing in the top ten invariants that will be reported in the next section.

5.1.2 Autonomous Driving. The autonomous driving study is designed to explore drivers reactions under 16 different scenarios and modes, on a simulated four-lane road for the participants to interact with a driving simulator (Force Dynamics CR 401 [3]). In each scenario, there are four potentially hazardous incidents: a pedestrian crossing the road, a cyclist riding slowly in the same lane, a stopped truck in the same lane, and an incoming truck in the other lane. The occurring incidents are randomized so that the participant cannot predict the next incident. The simulated car is equipped with a sensor to detect any events in the roadway within 40 meters. When detecting an incident, depending on the scenario, the car may send an auditory alarm to alert the driver. Among the 16 driving scenarios, eight of them are fully-autonomous scenarios, the wheel, throttle, and brake are controlled by a path-following controller that attempts to maintain a velocity, stay in the center of the lane, and avoid hazardous incidents. The other eight are semi-autonomous so the driver can switch between autonomous and manual driving mode. The system will not respond to any operation from the human driver.While driving, the subjects can adjust their trust level towards the autonomous driving system in a scale from one to five, with five being the highest trust level, by pressing buttons embedded on the steer to increase or decrease the trust level. The drivers can only change the trust level in the autonomous driving mode. Switching to manual driving mode will set trust level at 0. Switching from the manual driving mode to the autonomous driving mode will set trust to the default of three.

The study had 19 participants from [Anonymized]. Each participant had one training trial and 16 experimental trials. 9 participants were randomly selected and their driving traces (144 traces total) were used to compute the priors. Traces were captured using the PreScan software [14] which was integrated with the simulator. Each trial lasted 180 seconds resulting in a trace of 180 elements with a time step of 1 second. We recorded the vehicle dynamics, environment information, and user reactions through 32 variables. In total, 3,326 invariants were generated for the driving scenario. Table 3 shows variables appearing in the top ten invariants that will be reported in the next section.

5.2 Results on Value-added

We present the top ten invariants generatedfor each system in Tables 4 and 6 (the rest of the invariants are available in the repo). Invariants are presented in the form $P(Outcome \mid Givens)$. The prior probability is the probability of outcome A as calculated from the prior dataset. The surprise ratio shows the relation of posterior to prior, and the explanation is a straightforward description of the invariant defined formally in the first column.

Drone ISR. Table 6 shows the top ten invariants for the drone scenario, ranked in descending order by surprise ratio. A system developer might expect some of the invariants with high posterior probabilities, such as *P(MissionState=Complete | MachineState=Landing UserCommand=Default WarningState=Default)* on row 10. This is because in order for the drone to have completed its mission, it must make it back to its original starting point without incident and land. This invariant is strongly upheld by design, and so its probability will remain high across any traces supplied to the engine. Compare these high posterior probabilities to the respective 0.04574 prior probability of $MissionState = Complete$, which was calculated by frequency count across all state spaces in the prior dataset.

Some invariants had an unexpectedly high surprise ratio. Invariant *P(sensor.status=4 | MissionState=PossibleTargetDetected 0.01≤y-velocity<0.25 reaction_time=null)* with the highest surprise ratio on row 1 showed that when the drone was in a *PossibleTargetDetected* state and y-velocity was low, the sensor was only able to detect a partial target, perhaps indicating the need for a lower sweep speed in order to keep the full target in sensor range. The high surprise ratio of *P(UserCommand=RequestAutoControl | MissionState=InProgress MachineState=LosingVicon WarningState= LosingVicon)* on row 4 was unexpected because other mission states can be associated with *UserCommand=RequestAutoControl*, such as a suspension of the mission and use of manual control to return to Vicon connectivity following to a $NoVicon$ state, which can occur during any machine state. This invariant tells us that the user had difficulty perceiving when localization connectivity was re-established and tried using autonomous control when it was not possible, which could be an opportunity for improvement in a later version of the system. *P(reaction_time>7 | MissionState=OutsideSweepArea sensor.status=3 y-velocity>=0.25)* on row 7 was also unexpectedly high. The predicate *reaction_time>7* can be associated with any command, but it seems to be most closely associated with end-of-mission commands as the mission state it is most closely related to is *OutsideSweepArea*. The slow reaction time when the sensor is detecting a full target and the drone is outside the sweep area could be interpreted by the developer as an area for optimization, such as a need for sensor stabilization or a higher x-velocity at that time.

Predicates $flight\_data.battery\_low$, $y - velocity < 0.01$, and $0.01 \leq x - velocity < 0.25$ do not appear in the top ten invariants whatsoever. This shows that various subspaces within the trace show markedly different probabilistic behaviors than the priors characterizing the entire event space of similar traces, and which the naive approach discussed in the introduction could not capture. The wide gap in prior and posterior probabilities underlines the need to capture behavior on a stateful basis to more accurately represent system behavior.

The comparison of common predicates between invariants in Table 4 and priors in Table 7a, shows marked dissimilarities. No predicates appear in both the top ten priors and top ten invariants' posterior outcome predicates. Out of the 25 total unique predicates in Table 4 only 5 also appear in the top ten priors. This shows that stateful conditioning has a significant impact upon the probability of predicates.

We compare our approach to the perennially popular inference engine Daikon. Table 5 shows output for a tweaked version of Daikon that includes the invariant probabilities for single-predicate splitting and allows for expression of stateful behavior as a conjunction. Comparing the probabilities of the *UserCommand* predicates in Table 5 and Table 4, we see that the Daikon invariants capture probabilities closer to unconditioned prior probabilities, but give no indication of the circumstances under which those commands occur. In the *warning_state_change* entry point, we see that the range of *x-velocities* is similar to the Bayesian invariant *P(WarningState=LosingVicon | sensor.status=1 MachineState=LosingVicon x-velocity<0.01)* on row 6. However, this is not a conditional invariant but rather a conjunction of *sensor.status==1* ∧ *x-velocity* ≥ *-0.609* which holds with confidence ≥0.95 at this program point. Moreover, it is not possible to split this program point using more than one predicate at a time, thus overlooking most of the invariants in Table 4. While these invariants are informative, they do not reach the granularity and freedom of configuration that is achievable with our approach.

Overall, the generated probabilistic invariants confirm defined-by-design properties and expose properties of the drone ISR system that were heretofore unknown or not obvious and not captured by existing approaches.

Autonomous Driving. Similar to the drone scenario, some of the generated invariants with the highest posterior likelihood confirm our understanding of how the system operates. We here discuss invariants from the table selected for their high surprise ratios or which were of particular interest to the developers of the system. Invariant *P(Brake>0 | Mode=autonomous Throttle==0 Event=pedestrian detected TrustChange>0)* on row 1 had the highest surprise ratio, possibly due to the specific behavior the autonomous controller exhibited in the presence of a pedestrian and the trust-building effect it had on the human in the loop. Invariant *P(Throttle==0 | Mode = autonomous Event=pedestrian detected)* on row 2 had a similarly high surprise ratio, with posterior likelihood of 1.0 indicating that when the car is autonomously controlled and a pedestrian is in the roadway, the throttle is no longer engaged. This has a slightly higher probability than *P(Brake>0 | Mode=autonomous Throttle==0 Event=pedestrian detected TrustChange >0)*, likely because the throttle must be disengaged before brake can be engaged, and the brake may be engaged for multiple reasons, such as a different event or a curve in the road. The inclusion of predicate $TrustChange > 0$ in the givens was surprising as well, as it indicates that an increase in trust in conjunction with detecting a pedestrian is correlated with a subsequent application of the brakes. Note that this is not a causal relationship, as the autonomous driving algorithm does not react to changes in trust from the human in the loop.

Dealing with incidents on the road is essential to demonstrate realistic safety behaviors in the autonomous driving scenario. Invariants given *Event=Pedestrian detected* characterize the performance of the simulated car when handling the incident of pedestrian crossing the road. The car decreases the velocity by applying brake and easing the throttle, but no wheel change is closely associated according to the model selection algorithm. It fits the expectation that the car tends to slow down, rather than changing lanes to bypass the pedestrian. In larger models, both $WheelChange >= 20$ and $WheelChange < 20$ predicates present with no significant change to the posterior likelihood, showing that the posterior likelihood is more strongly conditioned on a change in $Throttle$. On the other hand, invariants given *Event=Cyclist detected* shows that the car performs a steep turn and unexpectedly accelerates in order to avoid the cyclist. Outside of the top ten, similar invariants can be found involving $Event = Truck$, which shows that the car again unexpectedly accelerates when passing the incoming truck on the other lane. These invariants present trends that the designers were unaware of and provide direct guidance on improving the system design when handling incidents.

Invariants related to *Mode* provide the information during manual driving or autonomous driving. Invariant *P(TrustChange>0 |*

Table 4: Drone Invariants

Invariant	Posterior	Original prior	Surprise ratio	Explanation
P(sensor.status=4 | MissionState=PossibleTargetDetected 0.01 ≤y-velocity<0.25 reaction_time=null)	0.29573	0.00185	159.85393	When a possible target has been detected, y-velocity is low, and no user reaction has been recorded, the sensor is likely detecting a partial target.
P(UserCommand=ReturnHome | MachineState=PossibleTargetDetected x-velocity>=0.25)	0.5213	0.00906	57.53832	When a possible target has been detected and x-velocity is high, user has likely just issued a command to return home.
P(UserCommand=Hover | MachineState=Sweeping x-velocity>=0.25)	0.02941	0.0006	49.01961	When drone is performing a sweeping task and x-velocity is high, user has likely just issued a command to hover.
P(UserCommand=RequestAutoControl | MissionState=InProgress MachineState=LosingVicon WarningState=LosingVicon)	0.27739	0.00604	45.92581	When mission is in progress, drone has detected unreliable Vicon connectivity, and a warning has been raised for unreliable Vicon connectivity, the user has likely just issued a command to give autonomous control to the drone.
P(UserCommand=Land | MachineState=Landing x-velocity<0.01)	0.02976	0.00121	24.59156	When machine is landing and x-velocity is low, the user has likely just issued a command to land.
P(WarningState=LosingVicon | sensor.status=1 MachineState=LosingVicon x-velocity<0.01)	0.3836	0.01853	20.70172	When sensor has not detected any target, drone has detected unreliable Vicon connectivity, and x-velocity is low, a warning has likely been raised for unreliable Vicon connectivity.
P(reaction_time>7 | MissionState=OutsideSweepArea sensor.status=3 y-velocity>=0.25)	0.23488	0.01183	19.85448	When drone is outside the sweep area, the sensor has detected a full target, and y-velocity is high, user reaction time is likely slow.
P(MachineState=FinishedBehavior | x-velocity>=0.25 UserCommand=ReturnHome)	0.68655	0.03845	17.85569	When x-velocity is high and user has issued a command to return home, the drone is likely finished its task.
P(MachineState=LosingVicon | sensor.status=1 x-velocity<0.01 WarningState=LosingVicon)	0.47021	0.02791	16.84733	When sensor has not detected any target, x-velocity is low, and a warning has been raised for no Vicon connectivity, the drone has likely detected loss of Vicon connectivity.
P(MissionState=Complete | MachineState=Landing UserCommand=Default WarningState=Default)	0.76471	0.04574	16.71854	When drone is landing, user has not issued a command, and no warning has been raised, mission is likely complete.

Table 5: Daikon Drone Invariants

Daikon Drone Invariants
/flight_data.battery_low one of { "False" (90.83%), "True" (9.17%) }
MissionState one of { "Complete" (4.63%), "InProgress" (9.14%), "InsideSweepArea" (37.34%), "OutsideSweepArea" (22.76%), "PossibleTargetDetected" (18.21%), "Suspended" (7.21%), "AbortingMission" (0.73%) }
UserCommand one of { "None" (1.00%), "Hover" (0.55%), "KeepSweeping" (2.51%), "Land" (0.73%), "LookCloser" (58.75%), "RequestAutoControl" (11.79%), "RequestManualControl" (13.85%), "ReturnHome" (10.83%)}
reaction_time ≥ 0.0
x-velocity ≤ 1.0
x-velocity ≥ -1.0
..warning_state_change():::ENTER;condition="sensor.status == 1" y-velocity <= 0.225656467772 x-velocity >= -0.609268306903

Brake==0 Mode=manual Throttle>0 Event=None) on row 5 shows that human drivers tend to increase their trust following "normal" operation in manual mode. This also indicates that they subsequently leave manual mode, as it is not possible to raise trust in manual mode.

Trust affects human drivers' reliance on the system. *P(TrustChange < 0 | Brake==0 Mode=autonomous Throttle>0 Event=None WheelChange >= 20)* on row 7 tells that when throttle is engaged and the wheel angle is changing quickly, the trust level is more likely to decrease. The absence of an *Event* seems to indicate this is a perceived safety issue on the part of the human in the loop. However, *P(TrustChange==0 | Brake==0 Mode=autonomous Throttle>0 Event=None WheelChange >= 20)* on row 10 shows us that the human in the loop is more likely to leave trust unchanged under those same conditions. This seems to indicate that there are latent variables not being accounted for, possibly in the system or on the part of the user. Understanding the trust evolution contribute to a trustworthy system.

The invariants in the driving scenarios confirmed expected performance and known system design attributes. Moreover, they helped to identify overlooked properties and better understand the system in the practical situation.

To judge the effect that given predicates have on the outcome likelihood, we compare our results in Table 6 to the prior probabilities in Table 7b. Of the 6 unique variables and 11 unique predicates shown in Table 6, the top ten priors and top ten invariants share all of those variables and 8 of those predicates. The top ten invariants contain 10 unique outcome predicates, half of which are absent from the priors. Posterior outcome predicates are predominantly *Range* and *Trend* types, whereas given predicates consistently include *Event* and *Mode* values. Only $Event = None$, $Event = Pedestriandetected$ and $Event = Cyclistdetected$ appear in these high-surprise, high-parsimony models. The only *Event* predicate in priors is $Event = None$. Additionally, of the outcome predicates, $TrustChange > 0$, $TrustChange == 0$, and $Brake > 0$ do not appear in the priors. Of the givens, $Throttle == 0$, $TrustChange > 0$, and all *Event* predicates excepting $Event = None$ do not appear in the priors. $Mode = Manual$, $WheelChange >= 20$, and $Brake == 0$ appear universally in all sets.

The insights gained from the conditional invariants and their marked dissimilarity from the priors show that we have uncovered invariants of value that were previously obscured in a context that did not consider state.

5.3 Results on cost

In this section we briefly explore a dimensions the space of invariants to explore and the associated inference cost. Figure 5 plots the runtime cost in system seconds when executing the inference engine on traces of three different lengths (25K, 50K, 100K) produced by the Drone ISR when three different sets of predicates are explored (5, 10 and 20 variables of *Range* type with two predicates each). Runtime tests were performed on a containerized Linux box with an x86_64 AMD FX-8120E 3.1GHz 8-core processor. As the graph shows, the runtime of the engine depends on both factors, but the influence of the number of variables being compared is dominant as the space to explore grows exponentially when the variables are considered as both outcomes and givens. As the number of variables grows, a developer can control this cost by specifying whether

Table 6: Driving Invariants

Invariant	Posterior	Original prior	Surprise Ratio	Explanation
P(Brake>0 | Mode=autonomous Throttle==0 Event=pedestrian detected TrustChange>0)	0.96674	0.04425	21.84719	When mode is autonomous, throttle is not engaged, a pedestrian is in the roadway, and trust has increased, brake is likely engaged.
P(Throttle==0 | Mode=autonomous Event=pedestrian detected)	1	0.0975	10.25641	When mode is autonomous and a pedestrian is in the roadway, throttle is likely not engaged.
P(Mode=manual | Throttle==0 Event=None)	0.94737	0.14803	6.39984	When throttle is not engaged and nothing is detected in the roadway, mode is likely manual.
P(WheelChange<20 | Mode=autonomous Event=None)	0.52775	0.496	2.91074	When mode is autonomous and nothing is detected in the roadway, wheel angle is likely changing slowly.
P(TrustChange>0 | Brake==0 Mode=manual Throttle>0 Event=None)	0.30904	0.115	2.68735	When brake is not engaged, mode is manual, throttle is engaged, and nothing is detected in the roadway, trust is likely increasing.
P(WheelChange>=20 | Brake==0 Mode=autonomous Throttle>0 Event=cyclist detected TrustChange<0)	0.90398	0.504	1.79362	When brake is not engaged, mode is autonomous, throttle is engaged, a cyclist is in the roadway, and trust increased, wheel angle is likely changing quickly.
P(TrustChange<0 | Brake==0 Mode=autonomous Throttle>0 Event=None WheelChange>=20)	0.14546	0.12	1.21214	When brake is not engaged, mode is autonomous, throttle is engaged, nothing is detected in the roadway, and wheel angle is changing quickly, trust is likely to decrease.
P(Throttle>0 | Mode=autonomous Event=None)	1	0.9025	1.10803	When mode is autonomous and nothing is detected in the roadway, throttle is likely engaged.
P(Brake==0 | Mode=autonomous Event=None)	1	0.95575	1.0463	When mode is autonomous and nothing is detected in the roadway, brake is likely not engaged.
P(TrustChange==0 | Brake==0 Mode=autonomous Throttle>0 Event=None WheelChange>=20)	0.76918	0.765	1.00546	When brake is not engaged, mode is autonomous, throttle is engaged, nothing has been detected in the roadway, and wheel angle is changing quickly, trust is likely to remain constant.

Table 7: Top Prior probabilities.

(a) Drone scenario

Predicate	Prior
UserCommand=Default	0.958
flight_data.battery_low=False	0.950
WarningState=Default	0.897
status.data=1	0.724
MachineState=Sweeping	0.427
y-velocity < 0.01	0.426
MissionState=InsideSweepArea	0.405
0.01 ≤ x-velocity < 0.25	0.390
x-velocity < 0.01	0.376
0.01 ≤ y-velocity < 0.25	0.354

(b) Driving scenario

Predicate	Prior
Brake=0	0.956
Event=None	0.918
Throttle>0	0.902
Mode=Autonomous	0.852
WheelChange≥20	0.504
WheelChange<20	0.496
VelocityChange>0	0.466
VelocityChange<0	0.197
Mode=Manual	0.148
TrustChange<0	0.120

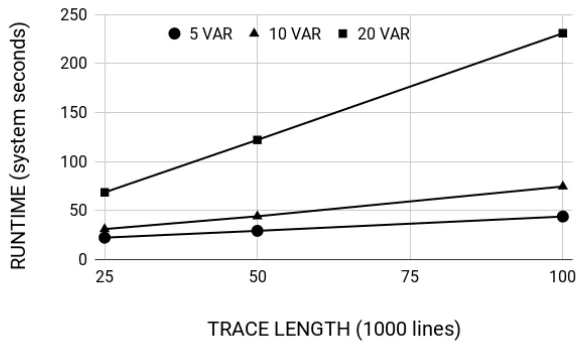

Figure 5: Runtime vs. trace length and number of predicates.

a variable is to be explored as a given or as an outcome, or more restrictively by aiming for particular pairs of variable predicates.

6 CONCLUSIONS

In this work we have introduced what we believe is the first automated approach to generate conditional probabilistic invariants leveraging Bayesian inference. Our study showed the viability and potential of the approach to capture rich behaviors in two distinct autonomous systems and contexts. While most invariant uses apply here, we believe that a probabilistic understanding of stateful behavior can quantitatively reaffirm system and safety properties, assist in the discovery of unexpected behaviors appearing only under certain contexts, and expose potential opportunities to optimize a system with greater context specificity. As for associated cost seen in Figure 5, the engine linearly scales to larger trace lengths and exponentially scales to larger variable spaces.

This work leaves a great deal of opportunity for future research threads. First, we will augment the inference engine by incorporating predicate definitions that include more complex operators such as temporal ones. Second, we hope to move past developer's suggested variable and variables ranges to an engine that can automatically explore that space through linear regression. Third, we want to further tap into the Bayesian capabilities of the engine, such as those for continuously updating the posterior and to incorporate multiple sources of information. Bayesian updating is one of the most attractive features of the Bayesian approach, where one after processing several traces and finding posterior probabilities for some invariant, one could use that posterior probability in the processing of further traces as a more accurate description of the predicate space of a trace. Moreover, as the trace is processed, the prior can be updated according to some given variable of choice. Say we want to know the probability of a true positive of detecting a target, i.e. $P(TargetDetected \mid DroneNearTarget)$. We would calculate the likelihood every step or several steps during the course of the trace and use the updated prior at the end, resulting in a more nuanced understanding of the overall probability of that prior for that particular trace.

REFERENCES

[1] Maryam Raiyat Aliabadi, Amita Ajith Kamath, Julien Gascon-Samson, and Karthik Pattabiraman. 2017. ARTINALI: Dynamic Invariant Detection for Cyber-Physical System Security *(ESEC/FSE 2017)*. Association for Computing Machinery, New York, NY, USA, 349âĂŞ361. https://doi.org/10.1145/3106237.3106282
[2] Glenn Ammons, Rastislav Bodı k, and James R. Larus. 2002. Mining Specifications *(POPL '02)*. ACM, New York, NY, USA, 4–16. https://doi.org/10.1145/503272.503275
[3] Force Dynamics. 2019. Force Dynamics CR 401. https://www.force-dynamics.com/.
[4] Michael D. Ernst, Adam Czeisler, William G. Griswold, and David Notkin. 2000. Quickly detecting relevant program invariants. In *ICSE*.

[5] M. D. Ernst, J. H. Perkins, P. J. Guo, S. McCamant, C. Pacheco, M. S. Tschantz, and C. Xiao. 2007. The Daikon system for dynamic detection of likely invariants. In *Science of Computer Programming*, Vol. 69. 35–45.

[6] C. Scott Ananian et. al. 2015. CUP Parser Generator for Java. https://www.cs.princeton.edu/~appel/modern/java/CUP/.

[7] Apache Software Foundation. 2016. Commons Math: The Apache Commons Mathematics Library. https://commons.apache.org/proper/commons-math/index.html.

[8] Apache Software Foundation. 2019. Commons Lang. https://commons.apache.org/proper/commons-lang/.

[9] Open Source Robotics Foundation. 2015. rosbag Package Summary. http://wiki.ros.org/rosbag.

[10] Open Source Robotics Foundation. 2019. Robot Operating System. https://www.ros.org/.

[11] Mark Gabel and Zhendong Su. 2008. Javert: fully automatic mining of general temporal properties from dynamic traces. In *SIGSOFT FSE*.

[12] L. Grunske. 2008. Specification patterns for probabilistic quality properties. In *2008 ACM/IEEE 30th International Conference on Software Engineering*. 31–40. https://doi.org/10.1145/1368088.1368094

[13] Sudheendra Hangal and Monica S. Lam. 2002. Tracking Down Software Bugs Using Automatic Anomaly Detection *(ICSE '02)*. ACM, New York, NY, USA, 291–301. https://doi.org/10.1145/581339.581377

[14] TASS International. 2019. PreScan Overview. https://tass.plm.automation.siemens.com/prescan-overview.

[15] L. Itti and P. F. Baldi. 2006. Bayesian Surprise Attracts Human Attention. In *Advances in Neural Information Processing Systems, Vol. 19 (NIPS*2005)*. MIT Press, Cambridge, MA, 547–554.

[16] G. F. Jenks. 1967. The Data Model Concept in Statistical Mapping. *International Yearbook of Cartography* 7 (1967).

[17] Hengle Jiang, Sebastian G. Elbaum, and Carrick Detweiler. 2013. Reducing failure rates of robotic systems though inferred invariants monitoring. In *2013 IEEE/RSJ International Conference on Intelligent Robots and Systems, Tokyo, Japan, November 3-7, 2013*. 1899–1906. https://doi.org/10.1109/IROS.2013.6696608

[18] Hengle Jiang, Sebastian G. Elbaum, and Carrick Detweiler. 2017. Inferring and monitoring invariants in robotic systems. *Auton. Robots* 41, 4 (2017), 1027–1046.

[19] Tien-Duy B. Le and David Lo. 2018. Deep specification mining. In *ISSTA*.

[20] Davide Lorenzoli, Leonardo Mariani, and Mauro Pezzè. 2008. Automatic generation of software behavioral models. *2008 ACM /IEEE 30th International Conference on Software Engineering* (2008), 501–510.

[21] ThanhVu Nguyen, Matthew B. Dwyer, and Willem Visser. 2017. SymInfer: Inferring program invariants using symbolic states. *2017 32nd IEEE /ACM International Conference on Automated Software Engineering (ASE)* (2017), 804–814.

[22] Thanhvu Nguyen, Deepak Kapur, Westley Weimer, and Stephanie Forrest. 2014. DIG: A Dynamic Invariant Generator for Polynomial and Array Invariants. *ACM Trans. Softw. Eng. Methodol.* 23, 4 (September 2014), 30:1–30:30. https://doi.org/10.1145/2556782

[23] Martin P. Robillard, Eric Bodden, David Kawrykow, Mira Mezini, and Tristan Ratchford. 2013. Automated API Property Inference Techniques. *IEEE Transactions on Software Engineering* 39 (2013), 613–637.

[24] Ryze Robotics. 2018. Tello User Manual v1.0. https://dl-cdn.ryzerobotics.com/downloads/Tello/20180212/Tello+User+Manual+v1.0_EN_2.12.pdf.

[25] Vicon Motion Systems. 2019. Vicon Motion Capture. https://www.vicon.com/motion-capture/.

[26] Neil Walkinshaw, Ramsay Taylor, and John Derrick. 2016. Inferring extended finite state machine models from software executions. *Empirical Software Engineering* 21, 3 (01 Jun 2016), 811–853. https://doi.org/10.1007/s10664-015-9367-7

[27] Ernst Wit, Edwin van den Heuvel, and Jan-Willem Romeijn. 2012. âĂŸAll models are wrong...âĂŹ: an introduction to model uncertainty. *Statistica Neerlandica* 66, 3 (Aug. 2012), 217–236. https://doi.org/10.1111/j.1467-9574.2012.00530.x

[28] Jinlin Yang, David Evans, Deepali Bhardwaj, Thirumalesh Bhat, and Manuvir Das. 2006. Perracotta: Mining Temporal API Rules from Imperfect Traces. In *Proceedings of the 28th International Conference on Software Engineering (ICSE '06)*. 282–291.