# Qualifying Examination Final Report: Implicit Invariants for Relational Data Structures

Meriel Stein

May 13, 2020

## Abstract

Robotics utilizes algorithms for control, localization, and navigation that rely heavily on complex data structures such as matrices and graphs. The values of these data structures are the result of interleaved controllers and complex interactions between equations that are difficult to parameterize as they relate to observed behavior. This makes the behavior, consistency, and maintenance of these data structures both difficult to track and highly critical to the successful operation of the system. Therefore, there is a demonstrable need to generate new types of invariants to better characterize the connection between control behavior and robustness. This work builds and generates novel invariants to capture the behavior of these complex data structures in both instantaneous and time-bounded contexts. Then, this invariant generation is applied to a testbed of simulated robotic system deployments. Capturing these invariants will enable the discovery of relations between member variables and model how data structures change over time.

## 1 Introduction

Program invariants describe truisms about a given system. When the system behaves unexpectedly, these truisms are often violated and the developer is tasked with the challenge of repairing the system to accommodate these surprise behaviors. This is especially true in the field of robotics, where behavior is often nondeterministic and developers cannot fully anticipate nor capture factors affecting operation.

Due to the multi-dimensionality of the space within which robots operate, their operation relies on high-dimensional data structures such as vectors, matrices, and graphs. These data structures are used in multiple places in the source code, collectively manipulated by multiple swarm agents, and/or passed between members and subsystems of the swarm.

The way these structures are operated upon can tell us much about the behavior of the system. Additionally, these data structures can tell us when the system's behavior is aberrant. Depending on the method with which invariants are inferred, invariants can leverage the way information is encoded into these data structures to tell us much about how the system is designed. One promising area for invariants is that of relational data structures.

This research seeks to explore implicitly encoded invariants for complex robotic systems whose runtime behavior is determined by control algorithms and to develop unique invariant patterns to effectively capture characteristics of their behavior. As these characteristics are often stored in data structures in the source code, these invariant patterns are meant to reveal characteristics of those data structures.

The goals of this project are to automatically infer such complex, subtle, but valuable invariants, which current approaches do not address. Current approaches are limited in their ability to infer relational characteristics within complex, arbitrarily large data structures in time series and to support the instantiation of patterns pertinent to those data structures. These data structures evolve with system behavior in relation to space and time, and so these invariants would be based on spatial structures and timely responses, something typical invariant inference techniques do not account for. Further detail is given in Section 3. I propose to explore new patterns for likely data structure invariants, and prototype inference tools to infer those patterns. To test these tools, I have applied them to existing cooperative systems to determine the prevalence of these patterns and determine whether they can be extracted from system traces and determined whether these patterns are effective at characterizing robustness and liveness properties by comparing the resulting inferred invariants across traces from successful and failed runs.

The hope is to expose invariants that point to system optimization, unaddressed vulnerabilities, and defensive strategies to increase robustness found through diffing invariants over successful and failed runs.

This paper presents the following contributions:

1. A frequentist inference engine to consume system traces and generate invariants for relational data structures.

2. A set of optimizations to that engine that reduce trace processing and abstract these data structures in a way that preserves their characteristics.

3. A library of patterns with demonstrable useful ap-

plication to relational data structures.

## 2 Motivation

Storing values together in a data structure implies a certain degree of interrelation between those values. This work aims to leverage that implied interrelation to characterize the system as a whole and to use the characteristics of these data structures to differentiate between successful and failed runs.

The intuition behind analysis of swarm algorithms is that relational data structures change more as the swarm is working to reach and/or maintain equilibrium, then see minimal change once the system is in a state of equilibrium.

Take the motivating example of a swarm meant to evenly disperse over a given area. After the swarm is randomly populated in the given environment, swarm members use attractive and repulsive force equations to determine the speed with which they actuate. Unless the swarm happens to be randomly populated in a configuration close to its final configuration, this speed will be faster at first and slow as the swarm is close to being evenly dispersed. This manifests in predictable rates of change in the distance matrix tracking the distance between members of the swarm. Additionally, all values in the distance matrix should be positive if they are Euclidean distances, and the matrix should be symmetric, as the distance from Robot 1 to Robot 6 is the same as the distance from Robot 6 to Robot 1. This distance matrix is collated from each member in the swarm calculating its distance from the swarm members it can sense.

These types of invariants cannot be inferred by existing invariant generation techniques, which tend to focus on single variables within linear programming paradigms.

As a collective, these invariants describe distance matrices, but can be applied to other structures. This work aims to develop a diverse library of invariant patterns for relational data structures and an engine capable of instantiating them for large traces.

### 2.1 Problem Space

The problem space was restricted to swarms for several reasons. One, swarms have highly interdependent components that can be readily observed and so an observer can gain an intuitive sense of the swarm's invariants and the patterns of behavior it exhibits over time. Additionally, those interdependent components are tracked in relational data structures whose values are constantly updating and the size of which are often scalable. Finally, swarms are capable of a range of potential behaviors, from simple to complex, that are sensitive to the environment in which they are deployed and thus have a variable propensity for failure.

I quantified the problem space for this work by collecting open source Robot Operating System (ROS) projects for swarms and cooperative robotic systems, then mined the source code for relevant data structures and explored potential commonalities. Putting together a set of swarm systems representative of the types of systems found "in the wild" confirmed that the motivating example generalizes to other swarms and heterogeneous cooperative robotic systems. These swarm projects were scraped from public Github repositories using permutations of three sets of keywords. The first set constricted the languages to python and C++, repository topics to ROS, and sorted by either stars, forks, or help wanted issues. The second list consists of terms pertaining to multi-robot systems: [{"multi", "cooperative"}+{"ugv", "turtlebot", "uav", "husky", "vehicle", "robot"}, "swarm"]. The last list configures results to prioritize swarm projects including simulations.

The scraped repositories were then pared down by maturity and how representative they are of swarms in deployment. Maturity was measured by number of commits, source lines of code, and the inclusion of a simulator. How representative they are is determined by the types of missions the system can perform, dependencies used, and whether the project cites any papers. The top 25 systems are listed in Table 1. System repositories are ordered by maturity. Repositories with high commits, high source lines of code (SLOC), and which use or include a demo with a simulation platform appear first.

To understand the behavior of complex robotic systems, I have investigated data structures in the source code which are closely tied to the behavioral specifications of the robot and thus critical to its functionality. Criticality was determined according to whether the data structure was published or updated at each cycle of the ROS control loop and whether its values could be used to determine whether the system had reached its goal. Matrices and point clouds containing sensing data, velocity vectors, and maps of the environment within which the robotic system operates would be counted as relevant because they are critical to the functionality of the robotic system. An array of strings corresponding to named states of the robotic system is less likely to be. Surveys on control algorithms tend to feature complex data structures in their implementations [14, 19]. These data structures are further complicated in later, more sophisticated robotics projects [4, 9], showing the ubiquity of these structures. Moreover, sensor data is often stored in relational data structures such as arrays and point clouds [7].

Preliminary analysis has been performed on the code bases in Table 1 whose control paradigms mirror early papers on ground and aerial swarm functionality. I collected all the numerically typed data structures present in these projects. This was accomplished by grepping for data types in C++ projects or constructor invocations in Python and tracing their usage and manipulation throughout the code. C++ data types were those of type $std :: vector$, $std :: list$, and $std :: array$ with numerical types, those same collections containing pointers to collec-

| GITHUB REPOSITORY | COMMITS | SLOC | LANG. | ROS? | SIM? | DATA STRUCTS OF INTEREST |
|---|---|---|---|---|---|---|
| yangliu28/swarm_robot_ros_sim | 198 | 3,152 | C++ | Y | Gazebo | 8 |
| xuefengchang/micros_swarm_framework | 182 | 8,990 | C++, python | Y | Rviz | 18 |
| lucascoelhof/voronoi_hsi | 76 | 1,484 | python | Y | Stage | 19 |
| or-tal-robotics/mcl_pi | 98 | 706 | python | Y | Rviz | 10 |
| terna/SLAPP3 | 145 | 5,772 | python | N | Turtle | 4 |
| hanruihua/slave_multirobot | 59 | 6,444 | C++ | Y | Matplotlib | 4 |
| raoshashank/ Multi-Robot-Decentralized-Graph-Exploration | 83 | 1,435 | C++, python | Y | Gazebo | 6 |
| USC-ACTLab/crazyswarm | 341 | 5,912 | C++ | Y | Cfsim | 10 |
| mehdish89/UR5_Cooperative_Transform | 582 | 4,238 | C++ | Y | Rviz, Gazebo | 12 |
| david-alejo/thermal_ws | 52 | 6,469 | python | Y | Marble | 17 |
| aarow1/cooperative_cable_transport_vision | 197 | 3,442 | C++ | Y | Rviz | 12 |
| awerenne/multi-robot-mapping | 75 | 3,667 | python | N | pygame | 5 |
| CARMinesDouai/ MultiRobotExplorationPackages | 104 | 56,504 | C++ | Y | Gazebo | 4 |
| Koll-Stone/Efficient_sche_cov | 47 | 774 | python | N | Matplotlib | 7 |
| ThomDietrich/multiUAV-simulation | 319 | 4,867 | C++ | Y | OMNet++ | 5 |
| afrl-rq/OpenUxAS | 706 | 464,238 | C++, python | Y | Amase | 25 |
| mrsd16teamd/MrdRRT | 103 | 1,770 | python | Y | matplotlib | 6 |
| correlllab/cu-droplet | 770 | 12,008 | C++ | N | Qt | 10 |
| aau-ros/aau_multi_robot | 102 | 15,834 | C++ | Y | N | 26 |
| gondsm/mrgs | 363 | 2,315 | C++ | Y | N | 10 |
| BasJ93/MinorAR_MultiRobot | 244 | 4,088 | C++, python | Y | N | 2 |
| mzahana/formation | 80 | 1,331 | python | Y | Gazebo | 8 |
| bramtoula/multi_robot_SLAM_separators | 189 | 4,694 | C++ | Y | N | 5 |
| jimjing/MandM | 65 | 2,831 | python | Y | Rviz | 8 |
| umass-rbr/multiagent-sas | 163 | 1,257 | python | Y | N | 7 |

Table 1: Table of swarm projects.

tions with numerical types, *std* :: *matrix*, *std* :: *map*, and project-specific object types with similar keywords in the class name. Python code was examined for variables containing a *dict*, *list*, or numpy object such as *numpy.array* or *numpy.matrix*. These data types are often used to instantiate ROS messages. C++ data structures *std* :: *set* and *std* :: *multiset* and Python data structure *set* were not considered as they were not common across systems.

The data structures were further analyzed for their role in meeting the behavioral specifications of the system. I examined the data structures present in these projects for their influence on sensing, kinematic behavior, swarm-wide knowledge and coordination, and control flow. This was done by grepping for ROS message types with fields using relational data structures and determining whether the data structures were used to instantiate any of these messages. Influence was scored according to whether they were used to compute new values in published messages, whether they were used to determine program path, and whether they were mutable or not. They were then pared down according to the relevance of the variable name and surrounding method names as they appeared in the source code. This returned a set of data structures likely to be worth instrumenting, tallied in the "Data structures of interest" column in Table 1. These projects comprise 248 data structures of interest over 624,222 source lines of code, averaging one data structure of interest being instantiated every 2,517 lines. The largest of these data structures was difficult to determine, as these depend on the scalable size of the swarm or the device driver populating laser scan values, but the smallest data structures were velocity and Euler angle orientation arrays of length 3 holding (x, y, z) or (roll, pitch, yaw) values. For one of the motivating example projects for this work, yangliu28/swarm_robot_ros_sim has 43 one-dimensional data structures of type std::vector, typed as double with the exception of 3 which are typed int_32t and 2 which are typed int. There were also 24 primitive one-dimensional arrays, 12 of type double and 12 of type int, and 28 two-dimensional arrays, 24 of type double and 4 of type int. These were predominantly used to store values relating to the individual swarm members, such as distance from one another, wheel velocities, and feedback from spring equations. By contrast, the raoshank/Multi-Robot-Decentralized-Graph-Exploration project used later in the study has 44 one-dimensional arrays and 8 two-dimensional arrays. Python is untyped but based on the operations performed on them they appear to consistently house doubles. Because the swarm is decentralized, these are used to house collated swarm-wide information, such as the SLAM-derived list of vertices and edges that comprises the map of the environment, and member-level information, such as velocity vectors and values keeping track of where the individual robot is within that map. These data structures of interest were published in a ROS message or encountered an opportunity to be updated every program loop.

# 3 Related Work

This work draws on several previous papers. Ernst et. al. [5, 6] 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 patterns and the values found in a trace. 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. We follow a similar approach in that our predicates are basic patterns instantiated by frequentist inference, but with the addition of epsilon equality comparison, linear temporal logic operators, and inference optimization techniques specific to relational data structures. Moreover, our instrumentation is dependent on time-series traces as opposed to method pre- and post-conditioning.

Hangal & Lam [11] created **DIDUCE** to perform instrumentation and invariant generation of Java bytecode for "tracked expressions" at various program points to better find the root of software bugs and to better tailor invariant generation to any given application. This work takes a similar approach, as instrumentation and subsequent invariant generation is limited to only data structures and program points determined to be of interest.

**Perracotta** [20] 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 approach incorporates some Perracotta techniques to accommodate noisy traces such as trace sampling, and has the potential to take inspiration from more techniques used by this tool in subsequent versions. In a similar problem space as Perracotta, Le et. al [16] use deep learning techniques to improve temporal specification model accuracy. While this work does not employ deep learning, it understands the potential for noise in temporal invariant instantiation. This work lumps in the need to account for noise in temporal invariants through trace sampling and other optimization techniques further explained in Section 4.3.

In a similar line of work as well, Gabel et al. [8] developed **Javert**, a mining framework of temporal logic invariants. Their approach is similar to many approaches in terms of combining patterns and incrementally encoding them as FSMs. Their technique of applying machine learning to instantiate their pattern library and their strategy to start with simple patterns that can be composed to generate much more complex ones could augment the approach proposed in this paper. This is discussed further in future work.

Diverging from temporal logics, Beschastnikh et. al's [2] 2014 invariant generation tool **CSight** puts forth techniques for handling pattern instantiation using timestamped logs from concurrent systems. These techniques produce a finite state model. This approach also leverages timestamped logs from concurrent systems to sample, maintain consistency, and validate linear temporal logics patterns.

Delving into invariants specifically for robotics, Jiang et al. [12, 13] extend the Daikon invariant library to patterns seen in robotic systems in order to derive monitors that can check system invariants at runtime. While Jiang et al. introduce invariant patterns tailored to robotic systems (e.g., bounded time differentials, polygonal relationships between spatial variables), their approach relies on Daikon to infer these invariants, and so encounters the same pitfalls.

Other methods of invariant generation were considered as reference materials for potential inspiration or future work. The 2014 invariant engine **DIG** developed by Nguyen et. al [18] generates nonlinear and geometric invariants over single variables and linear and reachability invariants for possibly-multidimensional arrays. However, it does not go beyond selective low-level linear relations on a element-wise basis, and does not necessarily extend these relations to account for all of the elements in the data structure. Nugyen et. al [17] apply similar inference techniques to symbolic traces in their 2017 tool **SymInfer**. While analyzing symbolic execution for implicit data structure invariants is beyond the scope of this work, it leads to compelling future work. Similarly, Csallner et al. [3] created **DySy**, a tool similar to Daikon with an orthogonal addition of symbolic execution. Symbolic execution plus concrete traces observed from executing test cases allows for the elimination of extraneous and/or redundant invariants that Daikon is susceptible to producing, which would benefit a future implementation of this Daikon-adjacent work. For example, an invertible matrix is also positive and so *isPositive* could be considered redundant and be removed from invariant output.

Grunkse [10] approaches invariants as a qualitative expression of requirements, introducing a rich set of specification patterns coupled with a structured English grammar to express bounded behavior of a system, instead of in terms of absolute correctness, in a way that incorporates expert knowledge of the system and that can be used for formal verification. Although this work did not pursue automated inference of invariants, its treatment of bounded patterns offered a jumping off point for this work. Grunske's subsequent work [1] on invariant generation offers a roadmap to expand this work into approximate pattern instantiation through machine learning techniques as well as a technique to cluster similar test cases.

Other invariant generation papers were considered as

potential jumping off points. Similar to claims in this paper, Kusano et al. [15] prove that Daikon produces inaccurate invariants for mutlithreaded programs. While this paper addresses that through its instrumentation technique, Kusano addresses it through invariant types.

# 4 Approach

Several goals must be met in order to discover invariants for these systems. The approach of this research is as follows:

1. Investigate potentially useful invariant patterns.
2. Expand upon existing inference techniques.
3. Optimize inference techniques for the domain of relational data structure invariants.

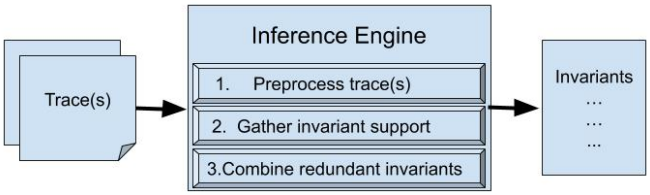



Figure 1: Overview of invariant generation.

An overview of the coalesced technique to generate invariants can be seen in Figure 1. Optimization is performed during the trace preprocess and in early breaks throughout gathering of invariant support. Support can be taken from multiple traces.

## 4.1 Invariant Patterns

While many different invariant patterns could be used to capture the qualities of relational data structures, these patterns were prioritized for their ability to capture interdependent features and behaviors of swarms. Criteria for potentially useful invariant patterns include but are not limited to patterns that characterize all elements of a data structure, a subset of elements within a data structure, or how the elements of the data structure change over time. Swarm member interdependence is often observable as subswarms, and data structure behavior tends to converge upon a single value or a cohesive distribution of values when the swarm is approaching equilibrium. Additionally, some persistent characteristics that hold for the entirety of the system run. For example, because the behavior of swarms tends to be bounded, e.g. members cannot be more than $x$ meters away from each other to maintain connectivity, bound and distribution invariants were used to capture these characteristics.

These patterns were also chosen for their potential to be arbitrarily complex. For example, conjunctions of linear

```
for(unsigned int i = 0; i < cloud_out.points.size(); ++i)
{
  uint32_t pt_index = cloud_out.channels[index_channel_idx].vals[i];
  tfScalar ratio = pt_index / ( (double) scan_in.ranges.size() - 1.0) ;

  tf::Vector3 v (0, 0, 0);
  v.setInterpolate3(start_transform.getOrigin(),
                                    end_transform.getOrigin(), ratio) ;
  cur_transform.setOrigin(v) ;

  tf::Quaternion q1, q2 ;
  start_transform.getBasis().getRotation(q1) ;
  end_transform.getBasis().getRotation(q2) ;

  cur_transform.setRotation( slerp( q1, q2 , ratio) ) ;

  tf::Vector3 pointIn(cloud_out.points[i].x,
                          cloud_out.points[i].y, cloud_out.points[i].z) ;
  tf::Vector3 pointOut = cur_transform * pointIn ;

  cloud_out.points[i].x  = pointOut.x();
  cloud_out.points[i].y  = pointOut.y();
  cloud_out.points[i].z  = pointOut.z();
}
```

Figure 2: Transformation from 2D LIDAR scan to 3D point cloud in CARMines-Douai/MultiRobotExplorationPackages

algebraic patterns can be used to express more complicated theorems that hold for a given data structure, and the distributions family of patterns can be extended to include other types of distributions, such as Poisson and geometric to capture probabilities associated with periodic equilibria or combined to be multivariate. These patterns come in a variety of forms, a full accounting of which can be found in Table 5:

**Distribution.** Distribution refers to the prevalence of member values in the matrix. For example, in a distance matrix for a swarm close to equilibrium for which the equilibrium distance is 2m, the density of values that are approximately a multiple of 2 in the distance matrix should be close to 100%. This can be useful in identifying subswarms or regions where the swarm has encountered an adversarial environment and can be defined as $\frac{data\_structure.count(value,\varepsilon)}{data\_structure.size}$, $mean == \mu$, and $variance == \sigma$. A low-variance distribution gives a better understanding of how the values are concentrated within the bounds invariants described below.

In the sample code for laser_geometry package's transformLaserScanToPointCloud() in Figure 2, a 2D LIDAR scan is transformed to a 3D point cloud taking into account the movement of the robot base during a scan. The point cloud is filled by interpolating the scan values using the transformation vector of the robot base from $start\_transform$ to $end\_transform$, then applying intermediate transformations to the interpolated point cloud points. For a structure like this, we want to verify that the point cloud is populated with values that have a distribution similar to the laser scan.

**Shape.** Shape refers to the placement of values in the data structure. For example, a distance matrix should be symmetric in shape and contain a zero diagonal, as the distance between a swarm member and itself is always zero, and the same distances apply between swarm member x to swarm member y and vice versa. These patterns characterizing shape can be de-

```
// 1.calculate distance between any two robots
double distance[robot_quantity][robot_quantity];
for (int i=0; i<robot_quantity; i++) {
    for (int j=i; j<robot_quantity; j++) {
        if (i == j) {
            // zero for the diagonal
            distance[i][j] = 0;
        }
        else {
            distance[i][j] = sqrt(
                pow(current_robots.x[i] - current_robots.x[j], 2)
                + pow(current_robots.y[i] - current_robots.y[j], 2)
            );
            // symmetrical matrix, copy the other side
            distance[j][i] = distance[i][j];
        }
    }
}
```

Figure 3: Update to swarm distance matrix for yangliu28/swarm_robot_ros_sim

```
void RuntimeHandle::insertOrUpdateNeighbor(int bot_id,
                                           float distance,
                                           float azimuth,
                                           float elevation,
                                           float x, float y,
                                           float z, float vx,
                                           float vy, float vz)
{
    upgrade_lock<shared_mutex> lock(neighbor_mutex_);
    std::map<int, NeighborBase>::iterator it =
                                        neighbors.find(bot_id);

    if(it != neighbors.end()) {
        NeighborBase new_base(distance, azimuth, elevation,
                                        x, y, z,vx, vy, vz);
        upgrade_to_unique_lock<shared_mutex> uniqueLock(lock);
        it->second = new_base;
    }
    else {
        NeighborBase new_base(distance, azimuth, elevation,
                                        x, y, z, vx, vy, vz);
        upgrade_to_unique_lock<shared_mutex> uniqueLock(lock);
        neighbors.insert(std::pair<int, NeighborBase>(bot_id,
                                                      new_base));
    }
}
```

Figure 4: Method to track swarm members entering, leaving, or adjusting their position and orientation within the swarm from ROS package repository xuefengchang/micros_swarm_framework.

fined as a set of boolean expressions, such as $diagonal == True$, $upper\_triangular == False$, $square == True$, $sparse == True$, and so on.

The sample code for yangliu28/swarm_robot_ros_sim in Figure 3 shows how the shape of a distance matrix is determined.

**Algebraic invariants.** Algebraic invariants are those such as invertability, value of the determinant, and matrix rank. These can be useful for determining dependencies within the matrix or whether computations are incorrect. These patterns can be defined as a set of boolean invariants as well, such as $positive == True$, $invertible == True$, $rank == 3$, and so on. For example, the distance matrix in Figure 3 must be positive semi-definite as all columns should be nonzero and all values should be nonnegative under normal operating conditions.

**Bounds.** This invariant pattern assigns a maximum and minimum to the values in a given data structure. This can be combined with density to determine if a swarm is close to equilibrium.

For a data structure such as the array of values obtained from a LIDAR scan $scan\_in$ in Figure 2, no value in $scan\_in$ should be greater than the maximum valid distance for which the LIDAR component is rated, and all values should be greater than or equal to zero.

**Changes over time.** This invariant pattern can be combined with the above patterns to capture the evolutionary characteristics of swarms. This can be broken up into regular time intervals or broken up according to significant events, such as encountering an obstacle or a vehicle leaving the swarm.

These invariant patterns are expressed in terms of linear temporal logics operators: next, eventually, and eventually always. As seen in the results tables in Section 5, next and eventually always provided the most valuable insights into swarm behavior.

The C++ $std :: map < int, NeighborBase >$ in xuefengchang/micros_swarm_framework in Figure 4 maps swarm members' runtime states to their identifiers. In a swarm tasked with dispersing evenly across a map, the velocities of the swarm members should decrease as the swarm reaches equilibrium and so the $vx$ and $vy$ members of all $NeighborBase$ objects in $neighbors$ should decrease over time.

**Subswarms.** Discovering subswarms is the finding of portions of a data structure that hold the same values within a user-specified epsilon ball for the lifetime of the swarm. Their values are not necessarily static but rather change together and so hold the same values as one another. Due to the high interdependence of swarm member behavior, these emergent patterns have the potential to show vulnerabilities and opportunities for optimization in swarm behavior.

## 4.2 Expanded Inference Techniques

The approach to learn likely invariants begins with an analysis of events in a set of traces. Events are developer-determined for the system under test according to how the code is instrumented. For example, if the code is instrumented after a method call manipulating a data structure, the event would be the change to that data structure. The frequency of an event is tallied over the course of the trace, and its confidence interval is computed. A typical confidence interval is computed as $1 - \frac{1}{n^2}$, $n$ being the event tally.

Likely invariants were generated from traces provided to a frequentist inference engine supplied with the patterns discussed in Section 4.1. Invariants that held with a confidence interval of 95% or greater were determined to be likely invariants. This confidence threshold of 95% is a standard in frequentist inference.

In order to handle inaccurate measurements and random noise, basic frequentist inference was expanded to

include fuzzy comparison by incorporating a parameterized epsilon term set by the user. Epsilon values for all projects was set to 0.1 to facilitate comparison of results across projects. Additionally, in order to handle trends over time, basic frequentist inference was expanded to include trace lookahead. "Heavy hitter" sampling allowed for additional space-saving measures by hashing the attributes of events that pass 30 samples (a threshold set by the Central Limit Theorem) for future comparison.

Algorithm 1 shows the basic frequentist instantiation and support of invariant patterns, which must be true throughout the trace. The trace is checked at all timesteps for support of the pattern, and the number of supporting samples is checked against the confidence interval of 0.95 for the invariant to be considered upheld and returned.

Algorithm 2 exemplifies the frequentist approach to instantiating temporal invariants. For each potential invariant pattern, the trace is processed to see if the invariant holds at a given timestep. If it does hold, it must hold for the rest of the trace or the instantiation is thrown away. If the trace is too close to the end for enough samples to be present to support this pattern, the instantiation is thrown away. This pseudocode assumes that the trace has already been preprocessed (i.e. sampled, rounded, etc.) according to the needs of the developer using optimization techniques discussed in Section 4.3

**Algorithm 1:** Inference for linear algebraic operators

```
   Input: trace, patterns
   Output: results
1  results = dict();
2  foreach pattern in patterns do
3      foreach record in trace do
4          if pattern.eval(record) then
5              results.put(pattern, True);
                 pattern.count++;
6          else
7              results.put(pattern, False); break;
8          end
9      end
10     if results.get(pattern) ∧ 1 − 1/(pattern.count²) <
         0.95 then
11         results.put(pattern, False);
12     end
13 end
14 return results;
```

**Algorithm 2:** Inference for "eventually always" temporal operator

```
   Input: trace, patterns
   Output: results
1  results = dict();
2  foreach pattern in patterns do
3      instantiated = False;
4      foreach record in trace do
5          highest_possible_conf_interval = 1 -
             1/((trace.length−trace.index(record))²);
6          if pattern.eval(record) then
7              instantiated = True;
8          else if not pattern.eval(record) then
9              instantiated = False;
10         end
11         if highest_possible_conf_interval < 0.95
             then
12             results.put(pattern, False); break;
13         end
14     end
15     results.put(pattern, True);
16 end
17 return results;
```

### 4.3 Optimization Techniques

Due to the massive scale of these traces, which produce two-dimensional data structures at an average rate of 335 milliseconds, traces containing more than 100 data structures were sampled at a parameterized rate of 20%. This is meant to handle the potentially nonlinear progression of swarm behavior due to noise as well as the problem of large traces.

An additional technique was explored to handle large relational data structures. Users have to option to abstract sequences of cells in the data structure by condensing them into one cell, by representing them as the average or maximum value. Average is useful in the case of similar readings, such as a LaserScan array, and maximum is useful in terms of discrete values, such as an OccupancyGrid matrix. While this technique can be applied to speed the processing of larger data structures, it can also be used to abstract smaller ones for readability.

Algorithm 3 shows the steps taken to reduce a matrix by a given factor. First, the size of the new matrix is computed according to approximate reduction by the factor and dimensions are computed according to the ratio of rows to columns in the old matrix. Then, according to whether the matrix is of a discrete type or not, the max or average of the $factor$ nearest cells is used to represent these cells in the new matrix.

## 5 Study

This section is meant to address the following research questions:

1. Are the posited invariant patterns for relational data structures upheld in practice by robotic systems with high interdependence (swarms)?

**Algorithm 3:** Optimization pseudocode

```
Input: trace, factor
Output: new_trace
size = trace[0].rows × trace[0].columns / factor;
ratio = round(trace[0].rows / trace[0].columns);
rows = size / (factor×ratio);
cols = size / (factor/ratio);
new_trace = [ ];
foreach matrix in trace do
    new_matrix = [rows][cols];
    foreach cell in new_matrix do
        nearest_cells = get_nearest_cells(cell.row,
         cell.col, factor, matrix);
        if matrix.type is int ∨ matrix.type is bool
         then
            cell = max(nearest_cells);
        else
            cell = avg(nearest_cells);
        end
    end
    new_trace.append(new_matrix);
end
return new_trace;
```

2. Can these invariant patterns be used to differentiate between successful and failed behaviors of these swarms?

3. Do the findings for the above two questions suggest the need for further patterns or further expansion of inference techniques?

4. What is the cost associated with generating the posited families of invariants?

Three ROS projects from Table 1 were chosen for their high level of system maturity and diversity of mission objectives. These projects' demos were run as intended by their developers, then the physical environment was adversarially perturbed in order to force a failure and the demos were re-run. Traces were collected from successful and adversarially perturbed runs. Success and failure is a system-specific metric based on code inspection and high-level system specifications. Then, likely invariants were generated, checked against system specifications and observable behavior to determine if they were sound, and compared over successful and failed runs. The gathered systems were run in simulation in order to analyze swarm deployments by manipulating the operating environments for noise and fault induction through changes such as spawning location, friction coefficients, and manipulation via obstacles.

## 5.1 yangliu28/swarm_robot_ros_sim[1]

Because this project employs a centralized controller, ROS message passing is limited to velocity commands between the controller and Gazebo with little understanding of how those velocities are generated under the hood. Analysis of this project focused on data structures within the controller, specifically a distance matrix and a nearest neighbor matrix. Results for the 8 by 8 cell distance matrix are discussed here.

Generated invariants for a distance matrix can be seen in Table 2. These invariants are from one 97.6-second run of a successful line-formation trace, which are then compared to a 120-second run in an adversarially perturbed environment containing two obstacles directly in the path of line formation in the fourth column, and a 60-second run in and adversarially perturbed environment derived from a satellite laser scan of a natural environment containing mountains, valleys, and generally uneven terrain in the fifth column. Of the 19 invariants upheld by the trace, 12 are of the linear algebraic type. This is likely because the operations performed on the data structure inform the linear algebraic invariants it exhibits. These invariants can be further interpreted through intuitive understanding of linear algebra, such as the norm indicating the area in space that the matrix describes.

Bound and distribution invariants are more easily upheld, because the entire trace can be used as samples to support the invariant. Therefore, if the trace is long enough, these invariants will always be upheld, though the values may change from trace to trace.

Temporal invariants for this swarm were expected in the successful case. The distance between each neighbor in the line was set to 0.7, so the mode of the data structure should eventually always settle close to 0.7 in the successful case. However, it is surprising that the determinant settled to be -56.96 in both the successful case and mildly perturbed case, but was violated in the mountain environment. As the determinant is intuitively thought of as the standard variance of the elements in the matrix, this becomes a bit more expected and understandable. The same goes for the norm eventually always being equal to 28.30, which is intuitively thought of as the size of the subspace described by the matrix.

Although there were subswarms that emerged for significant portions of the trace, frequentist inference requires that all samples exhibit an invariant for that invariant to be upheld. Therefore, no subswarms were detected. This type of invariant could be uncovered by different types of inference techniques or in combination with other types of invariant patterns, such as Bayesian inference or temporal logics patterns.

Moreover, it is worth reiterating that these patterns make no domain-specific assumptions about the data structures they receive. For example, if a distance of $-1$ were to indicate that the robot in question is not

[1] https://github.com/yangliu28/swarm_robot_ros_sim

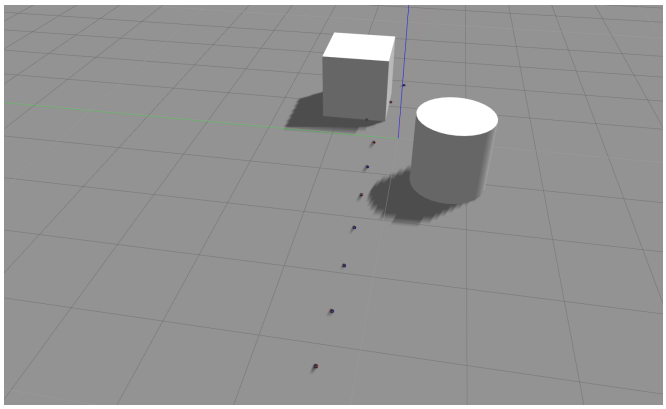


Figure 5: Unsuccessful end condition of run of yangliu line formation swarm with two obstacles.

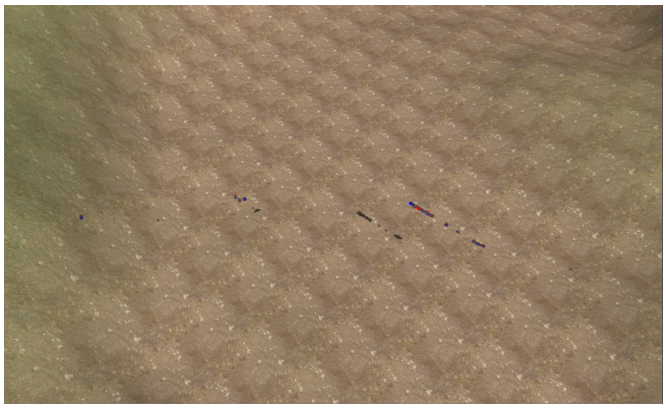


Figure 6: Unsuccessful end condition of run of yangliu line formation swarm in mountainous environment.

detectable, that logical jump is not encoded into the inference engine. For this reason, the patterns available to the inference engine are as generic as possible.

## 5.2 raoshank/Multi-Robot-Decentralized-Graph-Exploration[2]

This project is a relatively mature implementation of a decentralised graph exploration algorithm in Gazebo. A walled environment is mapped by two robots whose yaw is randomly generated between -45 and 45 degrees and whose forward motion is a constant 0.1m/s. LaserScan readings from both robots are incorporated into a shared map comprised of a set of vertices describing the exploration space. The 1000-cell LaserScan arrays are discussed here to provide a study for one-dimensional data structures employed in similar tasks.

Inferred invariants for LaserScan messages in this project are shown in Table 3. The linear algebra invariants are to be expected, given the shape of the one-dimensional array and its purpose as storing a sequence

[2]https://github.com/raoshashank/Multi-Robot-Decentralized-Graph-Exploration

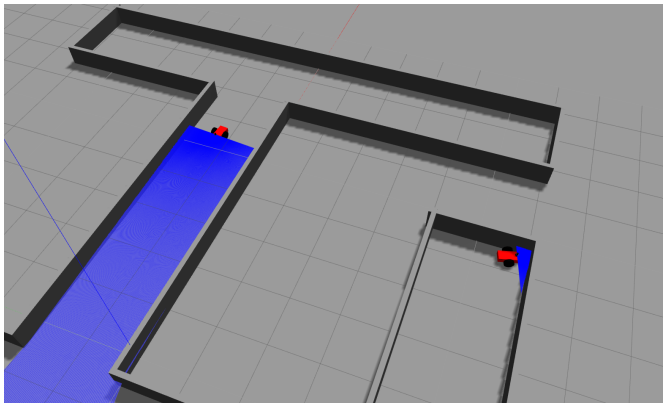


Figure 7: Multi-Robot-Exploration-Graph after one robot has become stuck in a corner of the walled environment.

of real numbers. The range of the norm is higher than expected, given the mean of the mean value is 3.357. The temporal next invariant is expected, given the rank does not change.

Due to the constant randomized motion and apparent lack of a stopping condition, there was not enough support for the possible temporal invariants with eventually always operators in a successful run because the yaw determining the values of the LaserScan messages does not stabilize.

It is possible to force unsuccessful runs for this project through manipulation of the environment. The movements of both robots are constrained to random forward movements. If a robot finds its way into a corner, it cannot make its way out because its software preempts random backwards motion, so it cannot back out of the corner. Setting one of the robots in a beginning position against a wall or too close to a wall will result in that robot eventually being disabled. This situation was determined to an unsuccessful run and was the metric against which invariants were determined to be violated or not.

This project suggests useful domain-specific extensions to the invariant pattern library, such as a linear temporal logic operator "always eventually" that can capture periodic equilibria.

## 5.3 lucascoelhof/voronoi_hsi[3]

The voronoi_hsi project uses a Voronoi-based strategy to autonomously disperse a scalable number of ground vehicles to cover non-convex environments. It accomplishes this through operating upon a universally scoped scalable matrix mapping the coverage environment with a default size of 80 by 80 cells, represented as a ROS OccupancyGrid message. The demo simulation performs this task with an 8-vehicle ground swarm.

[3]https://github.com/lucascoelhof/voronoi_hsi

| Invariant Type | Invariant | Explanation | Obstacle Violated? | Mountain Violated? |
|---|---|---|---|---|
| Shape | isSquare==True | Data structure is a square matrix. | N | N |
| Lin. Alg. | isSymmetric==True | Data structure is a symmetric matrix. | N | N |
| Lin. Alg. | isInvertible==True | Data structure is invertible. | N | N |
| Lin. Alg. | isPositive==True | Data structure is positive, i.e. contains only positive values. | N | N |
| Lin. Alg. | isLinearlyIndependent==True | Data structure has linearly independent columns. | N | N |
| Lin. Alg. | 7.52<= norm <=28.30 | The norm of the data structure falls between these values. | N | Y |
| Lin. Alg. | norm.gaussian(24.28, 5.31) | Norm values of the data structure follow a gaussian distribution with a mean of 24.28 and standard deviation of 5.31 | N | Y |
| Lin. Alg. | rank == 10<br>rank.gaussian(10.0, 0.0) | Data structure has rank==10. This makes the gaussian invariant redundant. | N | Y |
| Lin. Alg. | trace == 0.0<br>trace.gaussian(0.0, 0.0) | Data structure has trace==0. This makes the gaussian invariant redundant. | N | N |
| Lin. Alg. | isHermitian==True | Data structure is a Hermitian matrix. | N | N |
| Lin. Alg. | -56.96<= determinant <=-0.01 | Data structure has determinant between these values. | N | Y |
| bound | 0.66<= mean <=2.30 | Mean of data structure falls between these values. | N | Y |
| distribution | mean.gaussian(1.98, 0.42) | Mean of data structure follows a gaussian distribution with a mean of 1.98 and a standard deviation of 0.42. | N | Y |
| bound | 0.655<= median <=2.09 | Median value of data structure falls between these values. | N | Y |
| distribution | median.gaussian(1.78, 0.40) | Median of data structure follows a gaussian distribution with a mean of 1.78 and a standard deviation of 0.40. | N | Y |
| bound | 1.33<= maximum <=6.28 | Maximum value of data structure falls between these values. | N | Y |
| distribution | maximum.gaussian(5.44, 1.13) | Maximum of data structure follows a gaussian distribution with a mean of 5.44 and a standard deviation of 1.13. | N | Y |
| bound | minimum == 0.0<br>minimum.gaussian(0.0, 0.0) | Minimum value of the data structure is zero. This makes the gaussian invariant redudant. | N | N |
| subswarm | subswarm: (0,0) (1,1) (2,2) (3,3) (4,4) (5,5) (6,6) (7,7) (8,8) (9,9)] | Subswarm with (x,y) entries found. | N | N |
| temporal | ◯ next: norm@t ≤ norm@t+1 | Data structure is eventually always not sparse. | N | N |
| temporal | ◇□ isSparse==False | Data structure is eventually always not sparse. | N | N |
| temporal | ◇□ norm==28.30 | Norm is eventually always 28.30. | N | Y |
| temporal | ◇□ determinant==-56.96 | Determinant is eventually always -56.96. | N | Y |
| temporal | ◇□ mode==0.69 | Mode value in data structure is eventually always 0.69. | Y | Y |

Table 2: Invariants generated for distance matrix in yangliu control loop over one successful run.

This was a challenging project in which to force a nontrivial failure because of the higher level of specification accompanying the project and the mission itself. As is described on the Github page and accompanying references, the fact that it is meant to function in nonconvex environments, uses nonholonomic vehicles whose controllers exercise 6 degrees of freedom, and has full knowledge of the environment it is meant to cover, this system does not present the opportunity for a nontrivial failure. A trivial failure caused by an adversarial manipulation such as completely blocking access to a part of the environment essentially results in a new environment. Assuming a fully known environment, the human supervisor of the swarm would not issue that command to cover an unreachable area in the first place. Moreover, the simulation was in Rviz, a low-fidelity 2D environment specifically for testing ground vehicle controllers. More subtle adversarial manipulations such as the mountain environment in Section 5.1 were not possible without considerable changes to the original simulation that could invalidate the original intention of the system design. Despite applying a number of adversarial models, it was able to run to completion. Due to the highly specific system configuration and fully known potentially nonconvex environment, there was little opportunity to create a nontrivial failure in Rviz without altering the source code.

Invariants for this project can be seen in Table 4. The linear algebra category of invariants in the table is expected, given the 80 by 80 cell environment and the binary set of values {0, 100} denoting unoccupied and occupied, respectively.

Bound and distribution of rank is more surprising. A more sensitive/configurable for the temporal logics operators would be interesting to see how this progresses over time and whether rank increases or decreases over time.

The subswarm invariant shows $(x, y)$ coordinates of the OccupancyGrid that hold values equal to one another throughout the lifespan of the swarm run. In this case, cross-referencing with Figure 8, these seem to be cells which never become occupied and so their values never change. In future work with the incorporation of emergent obstacles into the environment, subswarm invariants could be further utilized to identify vulnerable parts of the swarm that are affected by these obstacles.

## 5.4 Cost to Infer

Figure 9 illustrates the time to compute invariants for a trace containing 2500 instances of 8-cell by 8-cell matrices of type double, organized by the type of invariants being

| Invariant Type | Invariant | Explanation | Violated? |
|---|---|---|---|
| Lin. Alg. | isPositive==True | Data structure is positive, i.e. contains only positive values. | N |
| Lin. Alg. | isSymmetric==False | Data structure is not a symmetric array. | N |
| Lin. Alg. | isComplex==False | Data structure contains to complex numbers. | N |
| Lin. Alg. | isHermitian==False | Data structure is not a Hermitian matrix. | N |
| Lin. Alg. | isSparse==False | Data structure is not sparse. | N |
| Lin. Alg. | rank==1 | Data structure rank is 1. | N |
| bound | 99.104 $\leq$ norm $\leq$ 123.333 | Norm of the data structure falls between these values | Y |
| distribution | norm.gaussian(111.480, 7.964) | Norm values of the data structure follow a gaussian distribution with a mean of 111.480 and standard deviation of 7.964 | Y |
| bound | 3.034 $\leq$ mean $\leq$ 3.670 | Mean of data structure falls between these values | Y |
| distribution | mean.gaussian(3.357, 0.205) | Means of the data structure follow a gaussian distribution with a mean of 3.357 and standard deviation of 0.205 | Y |
| bound | 2.505 $\leq$ median $\leq$ 2.638 | Medians of data structure fall between these values | Y |
| distribution | median.gaussian(2.567, 0.040) | Medians of the data structure follow a gaussian distribution with a mean of 3.357 and standard deviation of 0.205 | Y |
| bound | 8.060 $\leq$ maximum $\leq$ 11.210 | Maxima of data structure fall between these values | Y |
| distribution | maximum.gaussian(10.536, 1.105) | Maxima of the data structure follow a gaussian distribution with a mean of 10.536 and standard deviation of 1.105 | Y |
| bound | 0.837 $\leq$ minimum $\leq$ 0.873 | Minima of data structure fall between these values | Y |
| distribution | minimum.gaussian(0.857, 0.006) | Minima of the data structure follow a gaussian distribution with a mean of 0.857 and standard deviation of 0.006 | mean N, std Y |
| temporal | $\bigcirc$ rank@t == rank@t+1 | Next rank is equal to the preceding rank. | N |

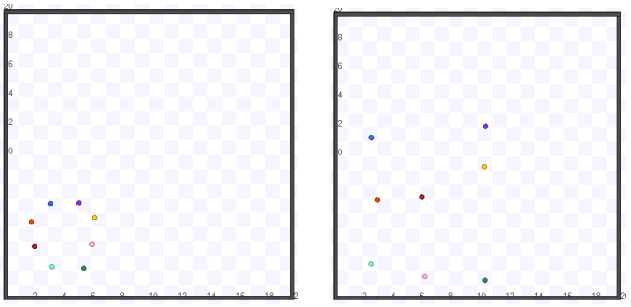
Table 3: Invariants from Multi-Robot-Exploration-Graph project for LaserScan array.

| Invariant Type | Invariant | Explanation |
|---|---|---|
| Lin. Alg. | isSquare==True | Data structure is a square matrix. |
| Lin. Alg. | isPositive==True | Data structure is positive, i.e. contains only positive values. |
| Lin. Alg. | isLinearlyIndependent==False | Data structure has linearly independent columns. |
| Lin. Alg. | isHermitian==False | Data structure is not a Hermitian matrix. |
| Lin. Alg. | isSparse==True | At least half of values in data structure are zero. |
| bound | 8 $\leq$ rank $\leq$ 45 | Rank of data structure falls between these values. |
| distribution | rank.gaussian(44.026, 5.923) | Rank of data structure follows a gaussian distribution with a mean of 44.026 and a standard deviation of 5.923. |
| bound | 1897.367 $\leq$ norm $\leq$ 4602.173 | Norm of data structure falls between these values. |
| distribution | norm.gaussian(4530.994, 432.966) | Norm of data structure follows a gaussian distribution with a mean of 4530.994 and a standard deviation of 432.966. |
| bound | 5.625 $\leq$ mean $\leq$ 33.094 | Mean of data structure falls between these values. |
| distribution | mean.gaussian(32.371, 4.397) | Mean of data structure follows a gaussian distribution with a mean of 32.371 and a standard deviation of 4.397. |
| bound | 2.505 $\leq$ median $\leq$ 2.638 | Median of the data structure falls between these values. |
| bound | median == 0.0 | Median of data structure is 0.0. |
| bound | maximum == 100.0 | Maximum of data structure is 100.0. |
| bound | minimum == 0.0 | Minimum of data structure is 0.0. |
| subswarm | (0-1, 0-79), (2-9, 0), (2-9, 1), (2-9, 78), (2-9, 79), (10-16, 0-1), (10, 48-60), (11, 46-62), (12, 45-63), (13, 44-64), (14, 44-65), (15, 43-66), (16, 43-66), (10-16, 78-79) | Subswarm emerged with these (x,y) entries in data structure. |

Table 4: Invariants from voronoi_hsi project OccupancyGrid matrix.

computed. The optimized curve shows the runtime in seconds with threading and sampling, and the unoptimized curve shows runtime without.

Previous works reference time complexity as the most significant cost to computing invariants for relational data structures. This is supported by the findings in this work, which found time to instrument arbitrary according to user preference and which previous work shows is possible to automate. As is evident from the figure, optimization has a significant impact on time to compute. On further tests, sampling had the most significant effect on reducing the time to run for distribution, linear algebraic, and bounds invariant families by one order of magnitude, whereas threading had the most significant effect on re-



(a) Initialized (b) 20 seconds

(c) 1 minute (d) Finished

Figure 8: voronoi_hsi swarm of 8 covering an 80 by 80 cell OccupancyGrid environment

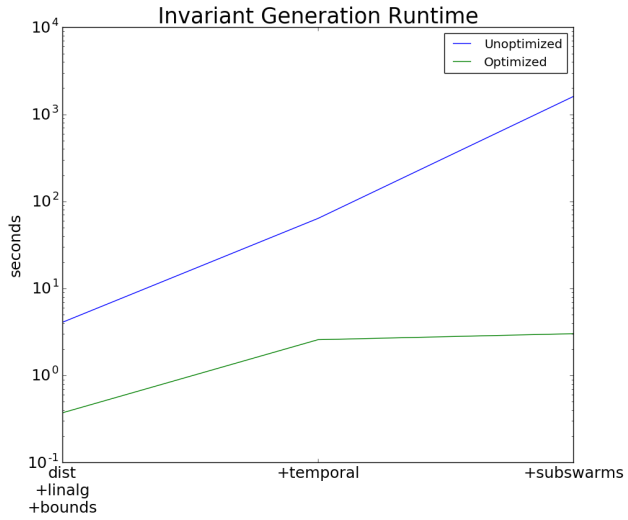



Figure 9: Runtime to compute invariants for 2501 2D data structures derived from a 93-second trace.

ducing the additional time to compute subswarms.

# 6 Conclusions

This paper explored the potential uses of invariants for relational data structures as they relate to cooperative robotic systems. The engine itself can operate on 1D and 2D relational data structures with the potential for extension. It is purposefully not enabled to do so in an elementwise fashion such as Daikon or in a reachability sense such as DIG. We assume that individual elements of the data structure are less interesting to characterize than the entirety. Experimentation with three real-life systems shows that this approach is effective at identifying and instantiating patterns that can at least partly characterize high level system behavior through low level data structures.

Due to length of traces and size of data structures, the inference engine is limited to analysing one data structure at a time. Without hardware upgrades (specifically CPU memory and speed) or significant parallelization efforts, invariants involving multiple relational data structures would be prohibitively time-consuming.

## 6.1 Future Work

These results present several opportunities for future work. One such avenue is the application of these patterns to similar systems that are similarly dependent on relational data structures, such as heterogeneous cooperative systems or neural networks.

Another is the extension of the pattern library (see Section A) to more patterns anticipated to be useful. This would go hand in hand with extension of the inference technique, for example if we want to capture approximate pattern instantiation. Capturing approximate classification is something better suited to machine learning techniques than straightforward linear algebra or elementwise comparison, and so the inference techniques would need to be augmented.

Overall, these patterns and inference methods show promise with the systems they have been applied to here, and could provide further value in future work.

# Appendix A Invariant Patterns

As seen in Table 5, a library of patterns was developed according to what might be useful to capture swarm behavior. These patterns are described in Section 4.1.

# Appendix B Focus Readings


1. Ivan Beschastnikh, Yuriy Brun, Michael D. Ernst, and Arvind Krishnamurthy. Inferring models of concurrent systems from logs of their behavior with csight. In Proceedings of the 36th International Conference on Software Engineering, ICSE 2014, page468–479, New York, NY, USA, 2014. Association for Computing Machinery.

2. 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, page 281–290, New York, NY, USA, 2008.Association for Computing Machinery.

| Invariant Type | Pattern | Explanation |
|---|---|---|
| Lin. Alg. | isSquare | x-dimension of data structure == y-dimension of data structure. Only applicable to 2D data structures. |
| Lin. Alg. | isSymmetric | Value at (x,y) equals value at (x,y) of transposed data structure. Applicable to 1D and 2D structures. |
| Lin. Alg. | isUpperTriangular | Data structure only contains nonzero values above the diagonal. Only applicable to 2D data structures. |
| Lin. Alg. | isDiagonal | Data structure only contains nonzero values on the diagonal. Only applicable to 2D data structures. |
| Lin. Alg. | isInvertible | For data structure A, $AA^{-1} = I$. Only applicable to 2D data structures. |
| Lin. Alg. | isPositive | All values in data structure are nonnegative. |
| Lin. Alg. | isLinearlyIndependent | Data structure has full rank. |
| Lin. Alg. | norm | Invariants on the norm values of a data structure. Must be combined with a bound, distribution, or temporal operator. |
| Lin. Alg. | eigs | Must be combined with a bound, distribution, or temporal operator. |
| Lin. Alg. | rank | Must be combined with a bound, distribution, or temporal operator. |
| Lin. Alg. | trace | Must be combined with a bound, distribution, or temporal operator. |
| Lin. Alg. | isComplex | Data structure contains complex values. |
| Lin. Alg. | isHermitian | Data structure is a Hermitian matrix. Only applicable to 2D data structures. |
| Lin. Alg. | determinant | Must be combined with a bound, distribution, or temporal operator. |
| distribution | A.gaussian($\mu$, $\sigma$) | x values follow a gaussian distribution with mean $\mu$ and standard deviation $\sigma$. |
| distribution | max == x | Maximum value of data structure is equal to x. |
| distribution | min == x | Minimum value of data structure is equal to x. |
| distribution | mean == x | Mean value of data structure is equal to x. |
| distribution | median == x | Median value of data structure is equal to x. |
| distribution | mode == x | Mode value of data structure is equal to x. Mode must occur more than once in individual data structures. |
| bound | A == B | A is equivalent to B within a user-defined epsilon ball. |
| bound | A $\leq$ B | A is less than or equal to B within a user-defined epsilon ball. |
| bound | A $\geq$ B | A is greater than or equal to B within a user-defined epsilon ball. |
| bound | A $<$ B | A is less than B within a user-defined epsilon ball. |
| bound | A $>$ B | A is greater than B within a user-defined epsilon ball. |
| subswarms | {($x_1$, $y_1$), ... ($x_n$, $y_n$)} | The (x,y) values in the set hold the same values within an epsilon ball at all steps in trace. |
| temporal | $\bigcirc$ A *op* B | *op* holds for current A and next B at all steps in the trace. |
| temporal | $\Diamond$ A | Eventually, value A appears in trace. Must be combined with a bound or distribution operator. |
| temporal | $\Diamond\Box$ A | Eventually, value A always appears in trace. Must be combined with a bound or distribution operator. |

Table 5: Currently supported invariant patterns.


3. Michael D. Ernst, Jake Cockrell, William G. Griswold, and David Notkin. Dynamically discovering likely program invariants to support program evolution. IEEE Transactions on Software Engineering, 27(2):99–123, February 2001.

4. Mark Gabel and Zhendong Su. Javert: fully automatic mining of general temporal properties from dynamic traces. In SIGSOFT FSE, 2008.

5. Thanhvu Nguyen, Deepak Kapur, Westley Weimer,and Stephanie Forrest. Dig: A dynamic invariant generator for polynomial and array invariants. ACM Trans. Softw. Eng. Methodol., 23(4):30:1–30:30, September 2014.

6. 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, 2006.

# Appendix C Background Readings

1. Tien-Duy B. Le, David Lo, Claire Le Goues, and Lars Grunske. A learning-to-rank based fault localization approach using likely invariants. In Proceedings of the 25th International Symposium on Software Testing and Analysis, ISSTA 2016, page177–188, New York, NY, USA, 2016. Association for Computing Machinery.

2. L. Grunske. Specification patterns for probabilistic quality properties. In 2008 ACM/IEEE 30th International Conference on Software Engineering, pages 31–40, May 2008.

3. Tien-Duy B. Le and David Lo. Deep specification mining. In Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2018, page 106–117, New York, NY,USA, 2018. Association for Computing Machinery.

# References


[1] Tien-Duy B. Le, David Lo, Claire Le Goues, and Lars Grunske. A learning-to-rank based fault localization approach using likely invariants. In *Proceedings of the 25th International Symposium on Software Testing and Analysis*, ISSTA 2016, page 177–188, New York, NY, USA, 2016. Association for Computing Machinery.

[2] Ivan Beschastnikh, Yuriy Brun, Michael D. Ernst, and Arvind Krishnamurthy. Inferring models of concurrent systems from logs of their behavior with csight. In *Proceedings of the 36th International Conference on Software Engineering*, ICSE 2014, page 468–479, New York, NY, USA, 2014. Association for Computing Machinery.

[3] 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, page 281–290, New York, NY, USA, 2008. Association for Computing Machinery.

[4] D. V. Dimarogonas and K. J. Kyriakopoulos. Connectedness preserving distributed swarm aggregation for multiple kinematic robots. *IEEE Transactions on Robotics*, 24(5):1213–1223, Oct 2008.

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

[6] Michael D. Ernst, Jake Cockrell, William G. Griswold, and David Notkin. Dynamically discovering likely program invariants to support program evolution. *IEEE Transactions on Software Engineering*, 27(2):99–123, February 2001.

[7] Open Source Robotics Foundation. Robot operating system: Sensor messages. http://wiki.ros.org/sensor_msgs, 2019.

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

[9] N. Goddemeier, K. Daniel, and C. Wietfeld. Role-based connectivity management with realistic air-to-ground channels for cooperative uavs. *IEEE Journal on Selected Areas in Communications*, 30(5):951–963, June 2012.

[10] L. Grunske. Specification patterns for probabilistic quality properties. In *2008 ACM/IEEE 30th International Conference on Software Engineering*, pages 31–40, May 2008.

[11] 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, page 291–301, New York, NY, USA, 2002. Association for Computing Machinery.

[12] Hengle Jiang, Sebastian G. Elbaum, and Carrick Detweiler. 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*, pages 1899–1906, 2013.

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

[14] C. A. Klein and C. Huang. Review of pseudoinverse control for use with kinematically redundant manipulators. *IEEE Transactions on Systems, Man, and Cybernetics*, SMC-13(2):245–250, March 1983.

[15] M. Kusano, A. Chattopadhyay, and C. Wang. Dynamic generation of likely invariants for multithreaded programs. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, volume 1, pages 835–846, 2015.

[16] Tien-Duy B. Le and David Lo. Deep specification mining. In *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA 2018, page 106–117, New York, NY, USA, 2018. Association for Computing Machinery.

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

[18] Thanhvu Nguyen, Deepak Kapur, Westley Weimer, and Stephanie Forrest. Dig: A dynamic invariant generator for polynomial and array invariants. *ACM Trans. Softw. Eng. Methodol.*, 23(4):30:1–30:30, September 2014.

[19] C. W. Wampler. Manipulator inverse kinematic solutions based on vector formulations and damped least-squares methods. *IEEE Transactions on Systems, Man, and Cybernetics*, 16(1):93–101, Jan 1986.

[20] 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, 2006.