Automated Extraction and Checking of Property Models from Source Code for Robot Swarms

Anonymous - Blind review
anon@anon.anon

ABSTRACT

As robots become a common presence in our everyday lives, ensuring the security and safety of robotic systems becomes an increasingly important and urgent challenge. Multi-robot systems, in particular, have the potential to revolutionize multiple industries—such as transportation and home care—where safety guarantees are a primary requirement. A known challenge for swarms and multi-robot systems is the gap between requirements and design, due to the need to translate swarm-level objectives into robot-level behaviors. In this paper, we focus on a less studied problem—the gap between requirements and implementation. As a case study, we use Buzz, that is a dynamic programming language designed for swarm robotics applications. Similarly to Python, Lua, and JavaScript, Buzz does not natively offer formal guarantees of correctness or safety. We propose an approach to automatically extract "as-implemented" models from Buzz programs, whose properties can then be formally analyzed and verified. Results obtained from the experiments performed on two medium-size open-source production-level systems for robotics research have also been reported. Our results show that the approach is feasible and is scalable to larger systems.

CCS CONCEPTS

• **Computer systems organization → Robotics**; • **Software and its engineering → Domain specific languages**.

KEYWORDS

Swarm Robotics, Software Engineering, Safety, Model Extraction, Model Checking, Domain-specific Languages

ACM Reference Format:
Anonymous - Blind review. 2022. Automated Extraction and Checking of Property Models from Source Code for Robot Swarms. In *RoSE 2022: Proceedings of the 4th International Workshop on Robotics Software Engineering, May, 2022, Pittsburgh, PA, USA.* ACM, New York, NY, USA, 8 pages. https://doi.org/TBD

1 INTRODUCTION AND MOTIVATION

Swarm robotics [Şahin et al. 2008] is a branch of multi-robot systems in which coordination among a large number of agents occurs purely through decentralized mechanisms. The spectrum of applications of swarm robotics is wide, and it includes search-and-rescue, planetary exploration, underground mining, and ocean restoration. Coordination in swarm systems is intended to be scalable, parallel, and resilient to faults in complex spatio-temporal dynamic tasks. However, these properties do not automatically arise as the product of decentralization; rather, they still require careful design and implementation by domain experts.

Swarm engineering [Brambilla et al. 2013] is an emerging discipline that studies design and implementation principles for real-world swarm systems. In this paper, we focus on one of the several existing hurdles towards sound and safe multi-robot systems: the issue of *programming* swarm-based solutions. While they share many of the typical issues of distributed systems, swarm systems also offer novel challenges:

- The computational units of a "swarm machine" are *robots*, i.e., machines that must sense and act in the real world, whose state is dynamic and unknown. Robots have intrinsic limitations such as strict energy budgets and the likelihood to experience noisy measurements and failures.
- The communication topologies of robot swarms are highly dynamic and volatile due to the agents' mobility.
- Coordination among individual robots occurs through local information and peer-to-peer interactions, making it impossible for any single robot to have awareness of the state of the entire swarm.

Decentralization makes swarm robotics radically different from more traditional multi-robot systems, such as warehouse automation systems. These latter often rely on powerful cloud infrastructure to offload complex computation and ensure consistency. In contrast, coordination in swarm systems is an emergent property that stems from the many local interactions among individuals. Some forms of emergent coordination are desirable, such as those leading to self-organized pattern formation, decision-making, and task allocation. However, undesirable emergent phenomena also exist, such as crowding and error cascades.

The general problem of engineering swarm robotics systems remains an open one. Among its challenges, the gap between swarm-level objectives and robot-level behaviors—often called the *global-to-local* problem—is one of the most researched.

In this paper, however, we argue that an equally important, but less studied problem, must be solved: the gap between swarm-level requirements and the concrete implementation of the system—in particular, its software layer. Our work is thus situated at the intersection between swarm robotics and software engineering.

Pinciroli and Beltrame proposed Buzz [Pinciroli and Beltrame 2016a], that is a programming language designed to offer a multi-paradigm approach to swarm programming. While successful in providing powerful and concise primitives for large collections of heterogeneous robotic platforms [Pinciroli and Beltrame 2016b; Pinciroli et al. 2016], Buzz lacks correctness and safety guarantees

RoSE 2022, May, 2022, Pittsburgh, PA, USA

ACM ISBN TBD…$TBD
https://doi.org/TBD

due to the dynamic nature of the language, which draws from Python, JavaScript, and Lua.

In this paper, we propose an approach to trace and analyze "as-implemented" Buzz scripts. This tool produces a model of the code that can be used for formal verification such as model checking. To the best of our knowledge, our work is the first to study this problem in the context of swarm robotics.

In the following, Section 2 presents the current state-of-art in validaton and verification of robotics applications. Sections 3 and 4 introduce the model extraction and model checking methods. Section 5 describes our experiments and reports on obtained results. Section 6 briefly introduces further research topics that are natural extension of this work. Finally, Section 7 presents our conclusions and perspectives.

2 VALIDATION AND VERIFICATION OF SWARM ROBOTIC SYSTEMS

Safety has always been a primary concern in robotics—and in automation in general—in the prospect of industrial and commercial applications. The public is keenly aware of any real or perceived risk related to self-driving cars or autonomous weapons, and the success of a technology depends on its widespread acceptance. As an example, the public has repeatedly questioned the safety of unmanned autonomous vehicles (UAVs, also known as *drones*) [Gorman et al. 2009; Shachtman 2011].

In general, public safety is addressed by enforcing policies and regulations (e.g., through the OSHA an FAA in the US). These standards require specific certifications to be granted before a system can be sold and used. We believe that, in the near future, new standards will arise to deal with swarm robotics systems, and new verification and certification methodologies will be needed[1].

Higgins et al. [2009a,b] reckon that the current rise of swarm robotics calls for the mitigation of its security and safety challenges as soon as possible. Some of these challenges are common to other domains—such as wireless sensor networks (WSNs), mobile ad-hoc networks (MANETs), multi-robot or multi-(software-)agent systems, and the internet of things (IoT). For a review of security challenges in the industrial IoT, for example, we refer the reader to the work of Sadeghi et al. [2015]. In general, wireless sensor networks (WSNs) [Ruiz et al. 2016; Saghar et al. 2010] does not consider the and other distributed systems research does not address the fundamental differences of robot swarms, i.e. the dynamics of the nodes. In fact, swarm robotics' peculiarities have unique repercussions on security and safety requirements [Vahidalizadehdizaj et al. 2015]. This urges for the meticulous design of their software architectures. Three, in particular, are the challenges highlighted in [Higgins et al. 2009b]: *(i)* complex and dynamic inter-communication among mobile robots; *(ii)* the special use of the concept of identity (or lack thereof); and, above all, *(iii)* the unpredictability of emergent group behaviors.

Some of the most promising application scenarios for swarm robotics are highly critical (e.g., to establish a communication infrastructure for first responders in the event of a natural disaster). The verification of conformity to policies is essential for such critical systems. Automating these verification steps has the potential to greatly accelerate the development of new applications.

Konur et al. [2012] proposed an automated formal model for a foraging swarm scenario to check if a swarm will behave as required. Since swarm behaviors usually embrace uncertainty and naturally depend on stochastic information, Kwiatkowska et al. [2011] have used a probabilistic model checker PRISM. Building on the work of Konur et al. [2012], Mikaël [2012] proposed a formal model to assess the flexibility of a foraging system. Flexibility was defined as the ability of a system to achieve a set of collective behaviors in various environments, without changing the behaviors of the individual agents. Similarly, Mikaël [2012] also used the PRISM model checker and validated their model by extensive simulations in ARGoS [Pinciroli et al. 2012]. Dixon et al. [2012] applied formal verification using temporal logic [Barringer et al. 2013] with the model checker NuSMV [Cimatti et al. 2002] on a particular swarm aggregation algorithm, namely Nembrini's alpha algorithm, for which the authors showed how formal temporal analysis can help refine the algorithm.

Winfield et al. [2005] focused on the alpha algorithm using temporal logic as well, but their emphasis was on specification rather than verification. Amin et al. [2017] used UPPAAL [Behrmann et al. 2006] to formally model and verify two algorithms: path planning and live human navigation through the robot swarm. However, Amin et al. [2017] did not take into consideration the stochastic behavior of the swarm or the temporal specification of the system. As part of the ASCENS project[2], Gjondrekaj et al. [2012] studied a collective transport behavior using the formal language Klaim [De Nicola et al. 1998] and related analysis tools. Particular attention has been dedicated to the stochastic aspect of the problem. However, their approach is scenario-specific and can involve only a limited number of robots.

Research exists on how to make secure-by-design programming languages for multi-robot systems [Desai et al. 2017; Ghosh et al. 2020] as well as runtime assurance systems [Desai et al. 2019]. These approaches require to redesign and recode the entirety of each robot behavior, along with the certification of underlying software. In contrast, we do not assume any previous guarantees, but look at ways to perform safety analysis of "as-implemented", dynamically-typed scripts that coordinate swarms of robots. Furthermore, each of the two systems used for the presented experiments in this paper is about 100 times larger than those used for experiments reported for the Koord language [Ghosh et al. 2020].

3 STATIC ANALYSIS AND MODEL EXTRACTION

3.1 Overview

Buzz [Pinciroli and Beltrame 2016a] is a simple, dynamically-typed language.[3] Buzz is open source, with an elementary compiler and virtual machine suitable for analysis. We propose to automatically extract property models from Buzz code using Pattern Traversal Flow Analysis (PTFA) [Letarte and Merlo 2009]. The extracted

[1] http://www.consilium.europa.eu/en/policies/drones/

[2] http://www.ascens-ist.eu/

[3] For a Backus-Naur form definition of the Buzz syntax, refer to https://the.swarming.buzz/wiki/doku.php?id=buzz_syntax_bnf.

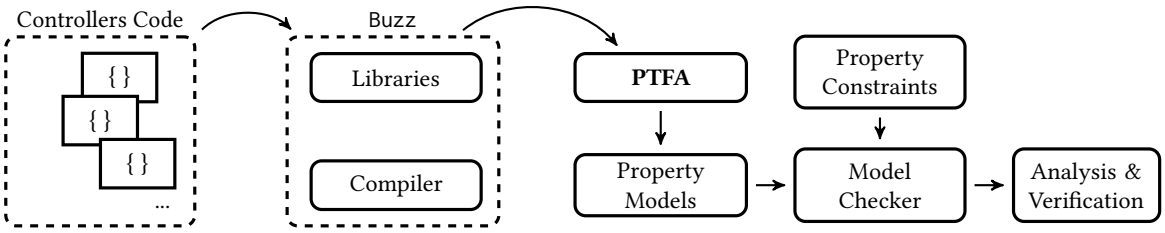

Figure 1: Flowchart of the overall verification architecture.

models can then be analyzed by a model checker to verify the required safety properties.

Figure 1 presents the flowchart of the verification process. The *front-end* analysis of Buzz language and compiler are language-dependent, while the whole process chain from PTFA to the verification is language independent.

3.2 Definite Reachability Analysis

PTFA computes the definite reachability of predicate satisfaction properties for all nodes in an inter-procedural Control Flow Graph (CFG), by propagating Boolean predicate satisfaction values.

PTFA analyzes and solves by dataflow analysis the dynamic calls, that are called "closure" in Buzz, performed by storing a function name into a program variable and calling the variable. PTFA also efficiently merges together the *property-satisfaction-equivalent* inter-procedural contexts. PTFA's asymptotic computational complexity is linear for one predicate, so the total complexity is proportional to the number of analyzed properties times the size of the CFG. In this paper, we use PTFA for the efficient extraction of definite reachability information from Buzz source code.

The predicate $definitely\_reaching(propName, v)$ is *True* if and only if there is at least one valid path reaching v and all paths to v satisfy $propName$.

$$\begin{gathered} definitely\_reaching(propName, v) = \\ (\exists p = (v_0, \ldots, x, y, \ldots, v) \mid (x, y) \in E_{CFG}) \wedge \\ ((\forall p = (v_0, \ldots, x, y, \ldots, v) \mid (x, y) \in E_{CFG}), \\ propValue(propName, v)) \end{gathered} \tag{1}$$

By default, *definitely_reaching* predicates are always assumed *False* for all properties, whenever no valid paths to v exist.

Considering the reverse paths composed of transposed CFG edges (E^T_{CFG}) from the exit CFG node v_M to a node w, *reachable properties* can be identified, as follows.

The predicate $definitely\_reachable(propName, w)$ is *True* if and only if all valid reverse paths from v_M to node w satisfy $propName$.

$$\begin{gathered} definitely\_reachable(propName, v) = \\ ((\exists q = (v_M, \ldots, y, x, \ldots, w) \mid (y, x) \in E^T_{CFG}) \wedge \\ (\forall q = (v_M, \ldots, y, x, \ldots, w), \\ propValue(propName, w))) \end{gathered} \tag{2}$$

3.3 Property Implication model

The property implication model captures the logical implication relation between all pairs of properties, as defined in Equation 3.

$$\begin{gathered} propertyImplicationModel = G_{IM} = (V_{IM}, E_{IM}) \\ V_{IM} = \{p_i \in PROPERTIES\} \\ E_{IM} = \{(p_j, p_k) \mid p_j \rightarrow p_k\} \end{gathered} \tag{3}$$

If a property p implies a property q, it means that for all possible executions corresponding to paths in the *CFG* reaching any node v in the *CFG*, the property satisfaction of p at node v logically implies that of q at the same node v.

Since the logical implication relation is transitive, if there exists a directed path in the property implication model G_{IM} from p to q, we can transitively infer that p implies q. In these cases, that may include single *CFG* edges, p is also called a necessary condition for property q and, conversely, q is a sufficient condition for p.

In general, since the property model is a directed acyclic graph (DAG), it may happen that for two properties there is neither a directed path from p to q, nor a path from q to p. In these cases, neither property is a necessary nor sufficient condition for the other. These properties are not related under the necessary or sufficient condition relation.

Generally speaking, the set of all necessary properties for a node v, representing a property p, is the set of all the properties traversed through paths in the property implication model that end into to node v. Similarly, the node v represents a sufficient condition for all the properties traversed in paths in the property implication model starting from v.

In Section 5, examples of real models extracted from source code are presented and discussed. In Figure 6, a black solid edge depicts an implication relation between a pre-condition and a following condition, while a blue dashed edge identifies an implication relation between a preceeding condition and a following post-condition.

In the presented models, certain properties p and q are linked by a directed path from p to q and, at the same time, there exist a directed path from q to p. When this happens, properties p and q are both mutual implying one another, that is, they mutually represent both necessary and sufficient conditions. When this happens, the properties are logically equivalent.

3.4 Model Extraction Algorithm

In Figure 2, an execution path containing one property-asserting transition is depicted.

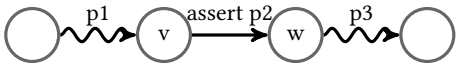

Figure 2: Assert implications

Since $((False \rightarrow True) = True)$ and $((False \rightarrow False) = True)$, the only constraints that must be verified to infer the property implication relations are defined in Equation 4, for property-asserting transitions.

$$\begin{gathered} \forall(v,\ w) \mid assert(v,\ w) = \text{“}assert\ p_2\text{”}, \\ (p_2 \rightarrow p_1) \leftrightarrow definitely\_reaching(p_1,\ v) \\ (p_2 \rightarrow p_3) \leftrightarrow definitely\_reachable(p_3,\ w) \end{gathered} \tag{4}$$

An implication relation exists between the granted property p_2 and all the *definitely_reaching* properties p_1 incoming into the granting transition (Figure 2 and Equation 4). Properties p_1 represent *pre-conditions* of p_2.

Additionally, an implication relation exists between the granted property p_2 and all the *definitely_reachable* properties p_3 encountered after the granting transition up to to the end of the execution (Figure 2 and Equation 4). Properties p_3 represent *post-conditions* of p_2.

The implication relations between properties can be computed by the algorithm reported in Figure 3 that directly computes the satisfaction of the constraints defined in Equation 4.

```
01   function computeModelEdges(defReaching,
                                defReachable) {
02     "initialize E_IM[p1, p2] = True"
03     forall p1 ∈ PRIV_SET {
04       forall (v, w) ∈ E_CFG |
                | assert(v, w) = "assert p2") {
05         E_IM[p2, p1] = (E_IM[p2, p1]∧
                        ∧ defReaching(p1, v))
06         E_IM[p2, p3] = (E_IM[p2, p3]∧
                        ∧ defReachable(p3, w))
       }
15     return(E)
   }
```

Figure 3: Model Implicaton Extraction Algorithm

The worst-case asymptotic complexity of the algorithm in Figure 3 is reported in Equation 5, where *ASSERT_SET* is the set of CFG edges that assert the satisfaction of a property p.

$$\begin{gathered} complexity = O(\ |PROPERTIES| \times \\ max(\ |ASSERT\_SET|, |PROPERTIES|\ )) \end{gathered} \tag{5}$$

Worst case complexity of the model extraction algorithm is quadratic on the number of tracked properties, when the number of property-asserting transitions is lower or equal the number of properties. The complexity is proportional to the number of properties times the number of property-asserting transitions, otherwise. The link between software size and model size is variable, because it depends on whether some additional pieces of software contain property-asserting transitions.

4 MODEL CHECKING

Extracted models are represented in JSON format and can be shared in an inter-operable way with other tools. Indeed, to experiment on model checking, we have converted the extracted models into an Alloy [4] problem as depicted in Figure 4 and into a Z3 [5] problem as show in Figure 5. The conversion is pretty straightforward and it is based on declaring the model nodes as facts and the model edges as implications relations.

Model checking is used to evaluate the implication transitivity across the models, to infer the validity of arbitrary pairs of property constraints. For the experiments, ROSBuzz and SwarmsRelays open-source developers suggested to us a subset of 20 significant property validation cases to be tested.

In figure 7 we can observe that the time required by Alloy to solve the one example problem is about 17 secs per pair of properties, while for the Z3 solver we observe an example of execution times of the order of 0.01 to 0.02 secs per pair of properties, as shown in Figure 8. Given this execution time difference, we decided to test the suggested property pairs using Z3 for the sake of feasibility and speed of execution.

5 EXPERIMENTS AND DISCUSSION

The robotics software used for our experiments, namely ROSBuzz[6] and SwarmRelays[7], have been taken from the MIST Laboratory Web site. They are both written to exploit the Buzz domain-specific language[8].

Buzz is a programming language for heterogeneous robots swarms [Pinciroli et al. 2015]. Buzz advocates a compositional approach, by offering primitives to define swarm behaviors both in a bottom-up and in a top-down fashion. Bottom-up primitives include robot-wise commands and manipulation of neighborhood data through mapping/reducing/filtering operations. Top-down primitives allow for the dynamic management of robot teams, and for sharing information globally across the swarm. Self-organization results from the fact that the Buzz run-time platform is purely distributed. The

[4] https://alloytools.org/
[5] https://github.com/Z3Prover/z3
[6] https://github.com/MISTLab/ROSBuzz
[7] https://github.com/MISTLab/SwarmRelays
[8] https://github.com/MISTLab/Buzz

```
abstract sig rosNode {
  impl : set rosNode
}

one sig LimitAngle_begin, LimitAngle_end,
add_obstacle_begin, add_obstacle_end,
...
unpackstatus_begin, unpackstatus_end,
vec_from_gps_begin, vec_from_gps_end extends rosNode{}

pred rosTopology {
  LimitAngle_begin.impl = LimitAngle_end +
                          gps_from_vec_begin
  LimitAngle_end.impl = LimitAngle_begin +
                        gps_from_vec_end
  add_obstacle_begin.impl = add_obstacle_end
  add_obstacle_end.impl = add_obstacle_begin
  barrier_allgood_begin.impl = barrier_allgood_end
  barrier_allgood_end.impl = barrier_allgood_begin
  ...
  vec_from_gps_begin.impl = vec_from_gps_end
  vec_from_gps_end.impl = convert_pt_end +
                          vec_from_gps_begin
}

assert rosImplication {
  rosTopology implies
  stattab_send_end in unpackstatus_end.*impl
}

check rosImplication for 1
```

Figure 4: Alloy code

```
; Defining property boolean variables

(declare-const LimitAngle_begin Bool)
(declare-const LimitAngle_end Bool)
(declare-const add_obstacle_begin Bool)
(declare-const add_obstacle_end Bool)
(declare-const barrier_allgood_begin Bool)
(declare-const barrier_allgood_end Bool)
...

(declare-const vec_from_gps_begin Bool)
(declare-const vec_from_gps_end Bool)

; Define the property implications

(assert ( => LimitAngle_begin LimitAngle_end))
(assert ( => LimitAngle_begin gps_from_vec_begin))
(assert ( => LimitAngle_end LimitAngle_begin))
(assert ( => LimitAngle_end gps_from_vec_end))
...
(assert ( => vec_from_gps_begin vec_from_gps_end))
(assert ( => vec_from_gps_end convert_pt_end))
(assert ( => vec_from_gps_end vec_from_gps_begin))

; The negation of the assertion we are trying to check

(assert (not
(=> barrier_ready_begin barrier_set_end)
))

(check-sat) ; the result should be unsat,
            ; because our assertion holds
```

Figure 5: Z3 code

language can be extended to add new primitives (thus supporting heterogeneous robot swarms) and can be laid on top of other robotics software frameworks, such as ROS.

Table 1 reports the characteristics of the two analyzed systems. The size of both is about 4 KLOC. In the same table, we report execution times (in seconds) for the parsing and CFG construction. We ran our experiments on a system powered by an i7950@3.07GHz CPU. We configured the OpenJDK Java VM version 1.8.0_131 to use up to 8GB of RAM.

Table 1: Analyzed Systems

	ROS	Swarm Relays
Size (LOCs)	4007	3490
Parsing and CFG Construction (secs)	26	34
Reachability Analysis and Model Extraction (secs)	53	71

The analyzed properties are the traversal of the entry and exit nodes of some relevant functions in the code.

The model obtained for ROSBuzz is shown in Figure 6, while the model obtained for SwarmRelays is similar but not shown for space sake. Characteristics of extracted models are reported in Table 2.

These results show that the models can be efficiently extracted for two examples of production level code for robotics research.

Table 2: Extracted Model Characteristics

	ROS	Swarm Relays
nodes	96	128
edges	202	266

The visualization of the extracted model allows designers and developers to compare the *as-implemented* models with the *as-designed* models in the requirements and design documents. This can be helpful to identify accidental errors or to monitor and take proper actions against the architectural drift, that often is observed during the evolution and modifications of a complex system. Furthermore, the extraction of property models can be used to help the documentation of a system code.

Model checking allows the verification of implementation constraint in the application domain. The satisfaction of some predicate that should never happen according to some policies is a violation. Think, for example, of taking-off without verifying the GPS location of the robot. On the other hand, the unsatisfaction of some properties that should always happen according to some other policies is also a violation.

Model checking results obtained on the suggested test property pairs have been manually evaluated by ROSBuzz and SwarmRelays open-source developers.

If developers suggested to check $q1: \ p1 \rightarrow p2$, we also checked the converse query $q2: \ p2 \rightarrow p1$. In 18 out of 20 suggested test

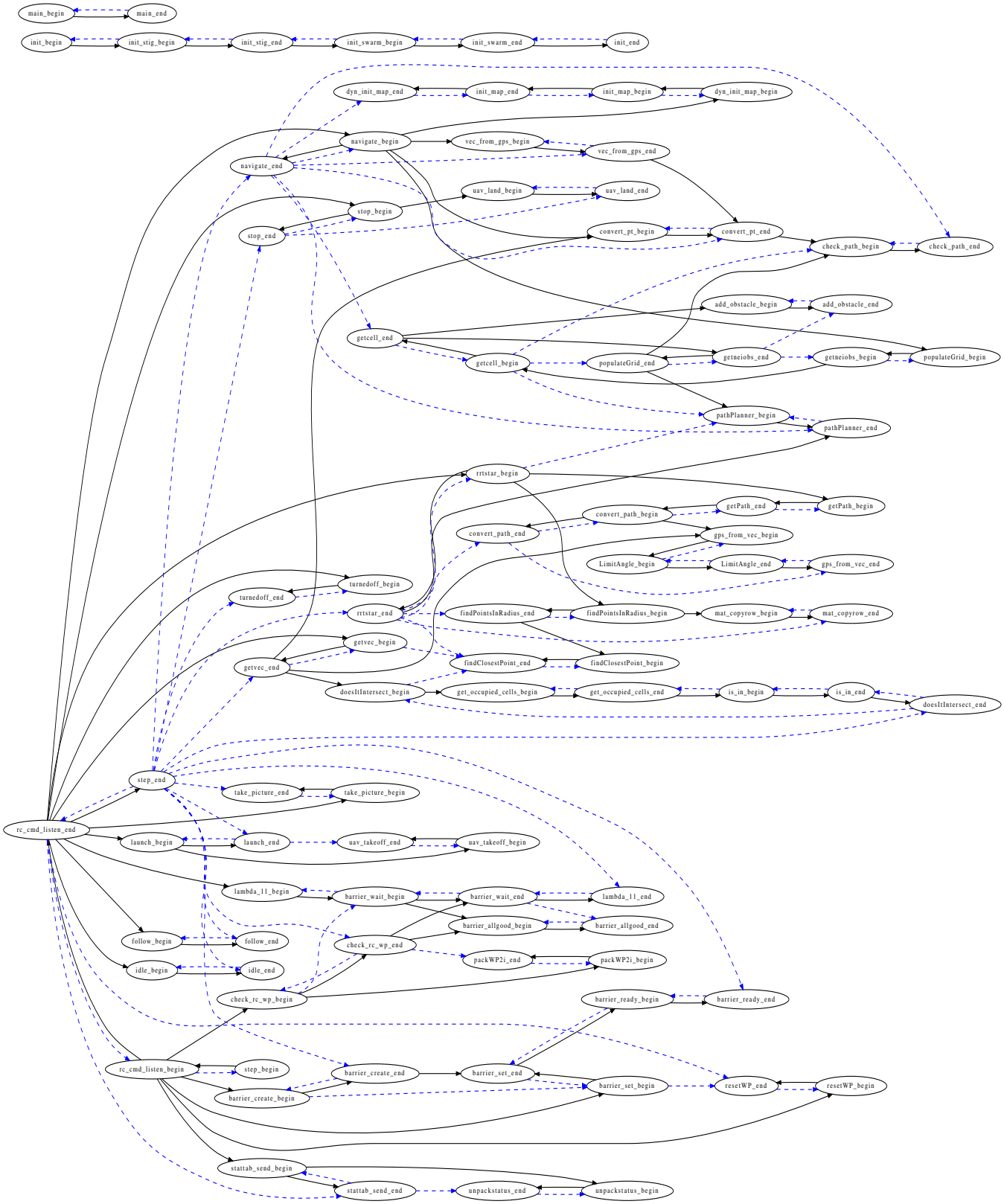

Figure 6: Implication model for ROS/mainRRT

```
Alloy Analyzer 5.1.0 built 2019-08-14T18:53:58.297Z

Executing "Check rosImplication for 1"
Solver=minisat(jni)
Bitwidth=4 MaxSeq=1 SkolemDepth=1 Symmetry=20
6298561 vars. 9216 primary vars. 8962075 clauses. 16939ms.
Counterexample found. Assertion is invalid. 612ms.
```

Figure 7: Alloy model checking results

```
ASSERT: (not (=> barrier_set_end barrier_ready_begin))
unsat
 :total-time   0.02)
ASSERT: (not (=> step_begin launch_begin))
unsat
 :total-time   0.01)
ASSERT: (not (=> step_begin idle_begin))
unsat
 :total-time   0.01)
ASSERT: (not (=> step_begin rrtstar_begin))
unsat
 :total-time   0.01)
ASSERT: (not (=> step_begin navigate_begin))
unsat
 :total-time   0.01)
ASSERT: (not (=> step_begin follow_begin))
unsat
 :total-time   0.02)
ASSERT: (not (=> step_begin take_picture_begin))
unsat
 :total-time   0.01)
ASSERT: (not (=> step_end launch_begin))
unsat
 :total-time   0.02)
ASSERT: (not (=> step_end idle_begin))
unsat
 :total-time   0.01)
ASSERT: (not (=> step_end rrtstar_begin))
unsat
 :total-time   0.02)
ASSERT: (not (=> step_end navigate_begin))
unsat
 :total-time   0.02)
ASSERT: (not (=> step_end follow_begin))
unsat
 :total-time   0.02)
ASSERT: (not (=> step_end take_picture_begin))
unsat
 :total-time   0.02)
ASSERT: (not (=> launch_begin uav_takeoff_begin))
unsat
 :total-time   0.02)
ASSERT: (not (=> launch_end uav_takeoff_end))
unsat
 :total-time   0.01)
ASSERT: (not (=> stop_begin uav_land_begin))
unsat
 :total-time   0.01)
ASSERT: (not (=> stop_end uav_land_end))
unsat
 :total-time   0.01)
ASSERT: (not (=> navigate_begin pathPlanner_begin))
unsat
 :total-time   0.01)
```

Figure 8: Z3 model checking results

cases, $q1$ satisfaction was $false$, while $q2$ was $true$, or the other way around. In these cases, the developers found the results in agreement with their expectation. In 2 cases out of the 20 cases, both the satisfactions of $q1$ and $q2$ were $true$. These cases were also in agreement with developers' expectation, but, indeed, they represented relations between *equivalent* properties $p1$ and $p2$, that were not explicit in the developers' minds before the experiments.

6 FURTHER RESEARCH AND DIRECTIONS

From the static analysis point of view, this paper investigates *definitely_reaching* and *definitely_reachable* flow information regarding all executions path. Further research should be devoted to integrate *possibly_reaching* and *possibly_reachable* flow analysis to extend model extraction to existentially quantified execution paths constraints.

The presented model extraction approach only deals with "property-asserting" transitions. For the purpose of generality, the approach should be extended and investigated to also include the analysis of "property-revoking" transitions, whenever appropriate. Nevertheless, these latter are not present in the presented experiments, since we cannot "revoke" the activation or the end of a function execution.

Several additional problems can be investigated through the means of model extraction. The use of the extracted model for model-based testing is one of those. Indeed, some criteria of model-based testing could include model-coverage based on nodes and edges.

The presented experiments involved a human-suggested test set of relevant properties. Indeed, the proposed approch can be used interactively, based on developers' requests. Further research could also be devoted to the analysis of all the properties involved in a system specifications, whenever specifications are available.

The analysis of deep causes of execution time differences between Alloy and Z3 model implementations are beyond the scope of this paper. Indeed, the differences may depend on the specific way code has been derived from models, in those environments. Further investigation should be required, for more general recommendations.

Future research will target the automated construction of justifications for the detected policy violations. By doing so, justifications and explanations of the reasoning that detected a policy violation in the code could be supplied to developers, to accelerate their comprehension of the reported violations.

Also, counter-examples may be automatically generated from PTFA results [Laverdière and Merlo 2017], when violations are detected. Further research would target the investigation of the effectiveness and usefulness of providing counter-examples to developers, to *harden* the existing code against policy violations and against *incorrect* model paths that may have happened because of design or programming errors.

Another area of investigation is the inventory and catalog of which safety properties are relevant to check in the code. Intuitively, these properties include time-outs and blocking functions. It may also be relevant to track safe mission termination policies. Graceful degradation of robot status, exceptions, and failure handling related to risk analysis and severity of failures are all potentially relevant

properties. Communication guarantees are of great importance for swarm-based systems. Topological properties of the robots should also be assessed. Cyber-security of the code is another important issue for certain application domains.

Finally, the approach presented here is based on static analysis. Certainly, an important contribution to safety analysis may be obtained by combining dynamic analysis of code execution during tests and simulation with the presented static models.

In the future, automated model-based verification of policies may become a part of formal code inspection for auditing or certification procedures. An advantage of the presented method is that it maintains traceability between models and code, since all results and violations can directly be traced to the code statements and paths.

The proposed model extraction can also be used to compare the models corresponding to two versions of the same system. In this scenario, one could detect the model changes between two versions and use these difference to assess the intended modifications.

Also, it will be interesting to investigate the developers' response during the development and evolution of robotics applications, to better assess the usability and effectiveness of the proposed approach.

7 CONCLUSIONS

We have presented a novel model extraction method of "as-implemented" *Definite Property Implications* and we have applied it to the analysis of systems written in the domain-specific programming language Buzz.

Experiments have been performed on two medium-size open-source production-level systems for robotics research, namely ROSBuzz and SwarmRelays. Results of model extraction have been reported.

Model checking of swarm robotics properties against the extracted models has been performed on suggested test pairs from ROSBuzz and SwarmRelays developers. Results have been succesfully manually validated and indicate that the approach is feasible and is also scalable to larger systems.

REFERENCES

Saifullah Amin, Adnan Elahi, Kashif Saghar, and Faran Mehmood. 2017. Formal modelling and verification approach for improving probabilistic behaviour of robot swarms. In *Applied Sciences and Technology (IBCAST), 2017 14th International Bhurban Conference on*. IEEE, 392–400.

Howard Barringer, Michael Fisher, Dov M Gabbay, and Graham Gough. 2013. *Advances in temporal logic*. Vol. 16. Springer Science & Business Media.

Gerd Behrmann, Alexandre David, Kim Guldstrand Larsen, John Hakansson, Paul Petterson, Wang Yi, and Martijn Hendriks. 2006. UPPAAL 4.0. In *Quantitative Evaluation of Systems, 2006. QEST 2006. Third International Conference on*. IEEE, 125–126.

Manuele Brambilla, Eliseo Ferrante, Mauro Birattari, and Marco Dorigo. 2013. Swarm robotics: a review from the swarm engineering perspective. *Swarm Intelligence* 7, 1 (01 Mar 2013), 1–41. https://doi.org/10.1007/s11721-012-0075-2

Alessandro Cimatti, Edmund Clarke, Enrico Giunchiglia, Fausto Giunchiglia, Marco Pistore, Marco Roveri, Roberto Sebastiani, and Armando Tacchella. 2002. Nusmv 2: An opensource tool for symbolic model checking. In *International Conference on Computer Aided Verification*. Springer, 359–364.

Rocco De Nicola, Gian Luigi Ferrari, and Rosario Pugliese. 1998. KLAIM: A kernel language for agents interaction and mobility. *IEEE Transactions on software engineering* 24, 5 (1998), 315–330.

Ankush Desai, Shromona Ghosh, Sanjit A Seshia, Natarajan Shankar, and Ashish Tiwari. 2019. SOTER: a runtime assurance framework for programming safe robotics systems. In *2019 49th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. IEEE, 138–150.

Ankush Desai, Indranil Saha, Jianqiao Yang, Shaz Qadeer, and Sanjit A Seshia. 2017. DRONA: A framework for safe distributed mobile robotics. In *Proceedings of the 8th International Conference on Cyber-Physical Systems*. 239–248.

Clare Dixon, Alan FT Winfield, Michael Fisher, and Chengxiu Zeng. 2012. Towards temporal verification of swarm robotic systems. *Robotics and Autonomous Systems* 60, 11 (2012), 1429–1441.

Ritwika Ghosh, Chiao Hsieh, Sasa Misailovic, and Sayan Mitra. 2020. Koord: a language for programming and verifying distributed robotics application. 4 (2020), 1–30. Issue OOPSLA. Publisher: ACM New York, NY, USA.

Edmond Gjondrekaj, Michele Loreti, Rosario Pugliese, Francesco Tiezzi, Carlo Pinciroli, Manuele Brambilla, Mauro Birattari, and Marco Dorigo. 2012. Towards a formal verification methodology for collective robotic systems. In *International Conference on Formal Engineering Methods*. Springer, 54–70.

Siobhan Gorman, Yochi J Dreazen, and August Cole. 2009. Insurgents hack US drones. *Wall Street Journal* (December 2009).

Fiona Higgins, Allan Tomlinson, and Keith M. Martin. 2009a. Threats to the Swarm: Security Considerations for Swarm Robotics. *International Journal on Advances in Security* 2, 2&3 (2009), 288 – 297.

F. Higgins, A. Tomlinson, and K. M. Martin. 2009b. Survey on Security Challenges for Swarm Robotics. In *2009 Fifth International Conference on Autonomic and Autonomous Systems*. 307–312. https://doi.org/10.1109/ICAS.2009.62

Savas Konur, Clare Dixon, and Michael Fisher. 2012. Analysing robot swarm behaviour via probabilistic model checking. *Robotics and Autonomous Systems* 60, 2 (2012), 199–213.

Marta Kwiatkowska, Gethin Norman, and David Parker. 2011. PRISM 4.0: Verification of probabilistic real-time systems. In *Computer aided verification*. Springer, 585–591.

Marc-André Laverdière and Ettore Merlo. 2017. Computing Counter-Examples for Privilege Protection Losses Using Security Models. In *Proc. IEEE 24th Int'l Conf. Software Analysis, Evolution, and Reengineering (SANER '17)*. 240–249. https://doi.org/10.1109/SANER.2017.7884625

Dominic Letarte and Ettore Merlo. 2009. Extraction of Inter-procedural Simple Role Privilege Models from PHP Code. In *WCRE '09: Proceedings of the 16th Working Conference on Reverse Engineering*. IEEE Computer Society, 187–191.

Lenaertz Mikaël. 2012. *Formal Verification of Flexibility in Swarm Robotics*. Ph.D. Dissertation. Citeseer.

Carlo Pinciroli and Giovanni Beltrame. 2016a. Buzz: a programming language for robot swarms. *IEEE Software* 33, 4 (2016), 97–100.

C. Pinciroli and G. Beltrame. 2016b. Swarm-Oriented Programming of Distributed Robot Networks. *Computer* 49, 12 (Dec 2016), 32–41.

Carlo Pinciroli, Adam Lee-Brown, and G. Beltrame. 2015. Buzz: An Extensible Programming Language for Self-Organizing Heterogeneous Robot Swarms. *ArXiv* abs/1507.05946 (2015).

Carlo Pinciroli, Adam Lee-Brown, and Giovanni Beltrame. 2016. A tuple space for data sharing in robot swarms. In *Proceedings of the 9th EAI International Conference on Bio-inspired Information and Communications Technologies (formerly BIONETICS)*. 287–294.

Carlo Pinciroli, Vito Trianni, Rehan O'Grady, Giovanni Pini, Arne Brutschy, Manuele Brambilla, Nithin Mathews, Eliseo Ferrante, Gianni Di Caro, Frederick Ducatelle, Mauro Birattari, Luca Maria Gambardella, and Marco Dorigo. 2012. ARGoS: a Modular, Parallel, Multi-Engine Simulator for Multi-Robot Systems. *Swarm Intelligence* 6, 4 (2012), 271–295.

M Carmen Ruiz, Hermenegilda Macià, José Antonio Mateo, and J Calleja. 2016. Formal analysis of an energy-aware collision resolution protocol for wireless sensor networks. *Procedia Computer Science* 80 (2016), 1191–1201.

A. R. Sadeghi, C. Wachsmann, and M. Waidner. 2015. Security and privacy challenges in industrial Internet of Things. In *2015 52nd ACM/EDAC/IEEE Design Automation Conference (DAC)*. 1–6. https://doi.org/10.1145/2744769.2747942.

Kashif Saghar, William Henderson, David Kendall, and Ahmed Bouridane. 2010. Formal modelling of a robust wireless sensor network routing protocol. In *2010 NASA/ESA Conference on Adaptive Hardware and Systems*. IEEE, 281–288.

Erol Şahin, Sertan Girgin, Levent Bayindir, and Ali Emre Turgut. 2008. *Swarm Robotics*. Springer Berlin Heidelberg, Berlin, Heidelberg, 87–100. https://doi.org/10.1007/978-3-540-74089-6_3

Noah Shachtman. 2011. Computer virus hits US drone fleet. *CNN.com* (2011).

M. Vahidalizadehdizaj, J. Jadav, and Lixin Tao. 2015. Security challenges in swarm intelligence. In *2015 6th International Conference on Computing, Communication and Networking Technologies (ICCCNT)*. 1–4. https://doi.org/10.1109/ICCCNT.2015.7395213

Alan FT Winfield, Jin Sa, Mari-Carmen Fernández-Gago, Clare Dixon, and Michael Fisher. 2005. On formal specification of emergent behaviours in swarm robotic systems. *International journal of advanced robotic systems* 2, 4 (2005), 39.