Engineering Safety in Swarm Robotics

Giovanni Beltrame
Polytechnique Montreal
Montréal, QC, Canada
giovanni.beltrame@polymtl.ca

Ettore Merlo
Polytechnique Montreal
Montréal, QC, Canada
ettore.merlo@polymtl.ca

Jacopo Panerati
Polytechnique Montreal
Montréal, QC, Canada
jacopo.panerati@polymtl.ca

Carlo Pinciroli
Worcester Polytechnic Institute
Worcester, MA, United States
cpinciroli@wpi.edu

ABSTRACT

Robotics, artificial intelligence, and the Internet-of-Things are driving current research and development for the technology sector. Robotic and multi-robot systems are becoming pervasive and more and more lives rely on their proper functioning in transportation, medical systems, personal robotics, and manufacturing. Assuring the security and safety of these systems is of primary importance to guarantee the real-world applicability of current research, and we argue that it should be an integral part of system design. Current software standards for safety and security for critical systems (e.g. industrial and aerospace) are not directly applicable to the large distributed systems that are envisioned for the near future. In this paper, we propose to address safety and security of swarm robotics systems at the programming language level. We propose to extend the Buzz multi-robot scripting language with constructs and code analysis that allow the verification of safety and security during development. We believe that detecting and correcting issues with what are inherently emergent systems—i.e. where collective behavior might not be immediately apparent from a single robot's code—during development would allow for a more effective advancement of swarm robotics.

CCS CONCEPTS

• **Security and privacy** → **Domain-specific security and privacy architectures**; • **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:
Giovanni Beltrame, Ettore Merlo, Jacopo Panerati, and Carlo Pinciroli. 2018. Engineering Safety in Swarm Robotics. In *RoSE'18: RoSE'18:IEEE/ACM 1st International Workshop on Robotics Software Engineering , May 28-June 28 2018, Gothenburg, Sweden.* ACM, New York, NY, USA, 4 pages. https://doi.org/10.1145/3196558.3196565

1 INTRODUCTION AND MOTIVATION

Swarm robotics [12] is a branch of robotics that focuses on decentralized coordination of large teams of agents. Swarm robotic systems promise scalable, parallel, and resilient solutions to problems that involve non-trivial space-time dynamic constraints. Swarm-based solutions are envisioned in diverse applications such as search-and-rescue, planetary exploration, underground mining, and ocean restoration, just to name a few.

Swarm engineering [1] is an emerging field that aims at the creation of methodologies for sound design of swarm robotics systems. In this paper, we argue that fundamental insight in this discipline can be gathered when swarm robotics is analyzed from a software engineering perspective. Swarm robotics systems can be seen as a "programmable machine" composed of large quantities of computational units. Swarm robotics systems are substantially different with respect to classical distributed systems:

(1) The computational units are *robots*—machines that must interact with the real-world and deal with changing working conditions, noise, failures, and partially observable states.
(2) Due to the robots' mobility, the communication network topology is dynamic and often partitioned in disconnected clusters; communication can also be prone to message loss.
(3) The complete state of the swarm is not available to individual robots which, hence, must rely on local information.

In addition to these design challenges, it must be noted that the dynamics of swarm robotics systems also displays *emergent dynamics*—properties and behaviors that are not explicitly encoded nor designed for, but that arise from the interactions of the robots among each other and with the environment. Some emergent properties 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 systematically designing swarm robotics systems remains open. A necessary step in this direction—we think—is the creation of tools that allow researchers to develop, share, and compare swarm algorithms. These tools would allow the field to mature, by revealing best practices and crystallizing methodologies. As a step towards this vision, Pinciroli and Beltrame created Buzz [10], a multi-paradigm programming language for robot swarms. Buzz is designed to create concise and composable programs to run across large collections of heterogeneous robotic platforms.

In this paper, we argue that, in addition to the ability to share, reuse, and compose software for swarms, Buzz must also offer primitives and constructs to encode and automatically analyze the *safety* of a given robot- or swarm-behavior.

2 SAFETY IN 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") [4, 13].

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. [5, 6] reckon the current rise of swarm robotics and the potential value of addressing its security and safety challenges early on. 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, please refer to [11]. Nonetheless, swarm robotics' peculiarities have unique repercussions on security and safety requirements [14]. This urges for the meticulous design of their software architectures. Three, in particular, are the challenges highlighted in [6]: (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.

We propose to integrate safety and security primitives in Buzz through keywords, built-in functions, and libraries. An application-dependent library of primitives can also be internally authored by the safety and security team of a development organization. Figure 1 presents the flowchart of the envisioned verification process.

We want to design these primitives to represent traceable "syntactic patterns" in the code such that formal analysis and model checking of safety/security policies become feasible. Policies might concern the behavior of individual robots as well as their interaction. In addition, the formal analysis should also be easy to compute, efficient, and as precise as possible, with respect to a reduced set of conservative false positives and false alarms. Furthermore, we want to complement static analysis by integrating in the Buzz Virtual Machine (VM) the implementation of dynamic safety checks. For example, dynamic checks may be required whenever the conservative approximation of static analysis is unsatisfactory. It is important to observe that, since Buzz is a recently developed language, the implementation of these ideas is a lot less arduous than what it would be in more established programming languages. Buzz is relatively simple, does not have any dependencies, and can be modified without strong repercussions on a large user base.

In the following, Section 3 provides a hint of the necessary background theory on control patterns. Section 4 presents the Pattern Traversal Flow Analysis [8] that extracts a safety and security model by tracing Buzz primitives in the code. In Section 5, we outline how to exploit this model for automated checking. Finally, Section 6 concludes our work summarizing the next practical steps that we will undertake to make Buzz—and swarms—safer tools for the robotics community.

3 PRIVILEGE CHECKING PATTERNS

As a first safety-oriented modification to Buzz, we propose to embed privilege checking patterns and routines directly into the language. Privilege checking routines are used to assert whether a robot owns a given privilege before performing restricted operations. More precisely, these routines verify that a robot is of at least one type that owns the required privileges.

As an example, in Listing 1, a robot can broadcast a message to its neighbors only if it has successfully passed the test of ownership of the privilege BROADCAST_ALL. Similarly, Listing 2 shows an example of a preemptive stop action: it forbids a robot from taking off, if the necessary authorization is not owned—as a simplified example, this is checked by evaluating the ownership of the privilege TAKE_OFF.

Listing 1: Example of Buzz access control routine.

```
if (safety.has_privilege(BROADCAST_ALL)) {

  # the current robot has the
  # BROADCAST_ALL privilege
  # send message to all robots

  neighbors.broadcast("key", value)
}
```

Listing 2: Another example of access control routine.

```
include "flight.bzz"

  if (!safety.has_privilege(TAKE_OFF)) {

    # stops the execution
    # if the current robot
    # doesn't have the TAKE_OFF
    # privilege

```

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

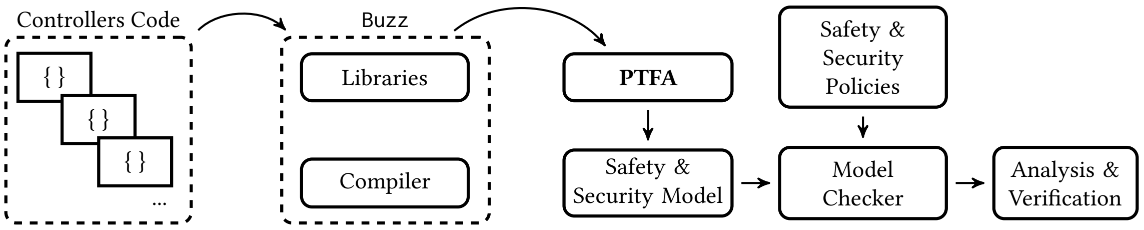

Figure 1: Flowchart of the overall verification architecture.

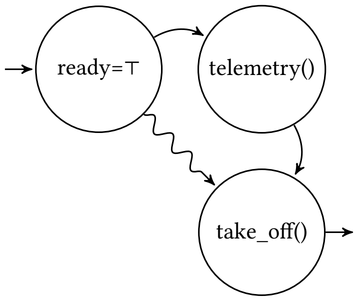

Figure 2: Unsafe execution model for a drone's take-off.

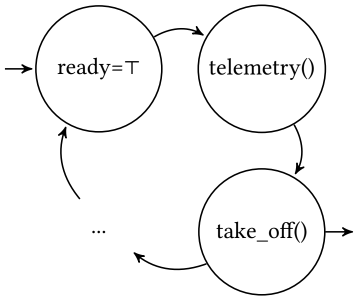

Figure 3: Safe execution model for a drone's take-off.

```
10        exit
11    }
12    take_off(altitude)
```

Privilege checking routines receive a privilege (or a set of privileges) as an argument and return a Boolean value asserting whether or not the robot is allowed to perform the corresponding action. We propose to add Privileges as special types in Buzz, that do not interfere with other language types such as strings, pointers, or character arrays. Static analysis of privileges is feasible—and easy—in an uncluttered language like Buzz, thus avoiding issues such as the analysis of data propagation involving strings, pointers, and arrays.

Figure 2 shows a stripped-down example of a safety model regarding telemetry and take-off operations. If we suppose that a safety policy requires telemetry to always be checked prior to take-off, it is clear that the model in Figure 2 violates such a policy. On the other hand, the model in Figure 3 satisfies it.

4 MODEL EXTRACTION AND INTER-PROCEDURAL ASPECTS

Pattern Traversal Flow Analysis (PTFA) [3, 8] extracts privilege satisfaction models from the source code of an application. Newly designed Buzz security and safety primitives that perform privilege verification checks will be identified as PTFA syntactic patterns.

We propose to produce an inter-procedural Control Flow Graph (CFG) through the Buzz parser. Then, PTFA can be used to transform it into an automaton M suitable for model checking as follows:

$$M = (Q_M, L_M, T_M, q_0, V_M, G_M, A_M) \tag{1}$$

where Q_M is a finite set of states; L_M is a finite set of labels applied on the states; $T_M \subseteq Q_M \times Q_M$ is a set of transitions; q_0 is the initial state; V_M is a set of variables used as "guards" and "assignments"; G_M is a set of "guards" that are logical propositions over V_M and are associated with transitions; and A_M is a set of assignments that modify the value of variables and are also associated with transitions.

In the privilege satisfaction model, states associated to a node w in the CFG represent the existence of a path from the beginning of the CFG to node w, in which a privilege *priv* has been granted. Safety and liveness properties of privileges can be verified by traversing the model. Policies can be expressed in Linear Temporal Logic (LTL) queries and verified against the model.

In the perspective of inter-procedural analysis, PTFA performs a *privilege satisfaction* sensitive analysis. Unlike context-sensitive analyses, that distinguish every calling context, PTFA only distinguishes calling contexts with different privilege satisfaction Boolean values. The possible privilege satisfaction contexts are thus intra-procedurally limited to $true$ and $false$, depending on whether the privilege was previously granted or revoked. Since a procedure can only be called from a $true$ or $false$ privilege satisfaction context, this limits to four the number of distinguishable contexts in which a model state can be for a single privilege. This constitutes a finite upper bound to the PTFA model size.

Guards and assignments in the model merge inter-procedural contexts, while preserving privilege satisfaction precision. We can often reduce the number of contexts by merging some of these results through a conservative approximation in the outcome of the analysis. By merging together contexts and propagating an equivalent privilege satisfaction value, PTFA reduces the number of inter-procedural contexts to be considered during analysis. Nevertheless, it preserves the full precision of privilege satisfaction information with no approximation, while reducing the number of caller/callee contexts to be processed. PTFA runs in linear time and memory with respect to the number of intra and inter-procedural edges in the CFG [8].

5 MODEL CHECKING OF SAFETY AND SECURITY PROPERTIES

Software model checking [2] is the algorithmic analysis of programs to prove properties of their executions. While originating from the logic and theorem proving fields, it has now evolved as a hybrid technique, simultaneously making use of analysis classified as theorem proving, model checking, or data-flow analysis [7].

A well-known limitation of model checking techniques is the combinatorial "state space explosion problem" forcing the model checker to explore a combinatorial number of states in the system under study. Several papers proposed exploration strategies, heuristics and specialized data structures to circumvent this problem and analyze increasingly large systems.

As previously mentioned, PTFA models are finite and upper bounded to four times the number of distinct privileges times the size of the corresponding CFG. Furthermore—while PTFA property models can be queried using arbitrary queries written in LTL that offer different execution time complexity—many useful safety and security queries are simple and can be efficiently computed by "ad-hoc" reachability algorithms as in [8].

A major challenge towards the validation of swarm safety is the evaluation of those properties that are *emerging*. These properties are a distinctive trait of swarms and they are not easily recognizable within the code of individual robots. Complex formations [9] are sometimes implemented by seemingly trivial controllers. To mitigate this problem, again, we propose to intervene at the programming language level, through new Buzz constructs (see Table 1). In our vision, the swarm programmer should be able to forfeit the implementation of low-level safety checks and use Buzz primitives and best practices instead. In particular, Buzz offers "swarm-oriented" programming [10] through three constructs: `swarm`, with which a developer can assign tasks to group of robots based on tags (i.e. the id of one or more "swarms") associated with the robots; `neighbors`, used to perform spatial computation; and `virtual stigmergy`, a shared tuple space among all robots of a swarm. We believe that adding model checking to models generated by PTFA on the these top-down constructs can control the behavior of the swarm as a single programmable machine, and provide safety from unwanted emergent behavior.

Another current limitation of Buzz is that any predicate-preserving function at runtime and at the swarm level must be coded by the developer. Providing an automatic swarm-level assertion mechanism would reduce errors and maintain desirable safety properties at runtime. (Table 1, `preserve` construct).

6 SUMMARY AND FUTURE WORK

This position paper presents our vision for the development of "safer-by-design" large multi-robot systems. In particular, we frame the problem in the context of the—increasingly popular—swarm robotics research field. We argue that the successful development of highly mobile, autonomous systems will only be possible with built-in security. To do so, we propose to intervene at the programming language level, enriching the domain-specific Buzz language with safety keywords and libraries. Thus, we enable the coupled exploitation of PTFA and model checking for the automated verification of policy compliance within a robot's controller.

We believe that the proposed approach can transform Buzz—that is explicitly intended for swarm robotics and currently in its early deployment stages—into the ideal platform for the software engineering of security in multi-robot systems.

Table 1: Buzz swarming functions and safety keywords

Examples of implemented domain-specific constructs and functions	
`stigmergy.{create(), put, get()}`	swarm-shared data-structure
`swarm.{create(), join(), ..}`	swarm management primitives
`neighbors.{broadcast(), filter(), ..}`	local robot-to-robot com.
Examples of envisioned security-specific keywords and functions	
`safety.{has_privilege(), ..}`	authorization verification
`behavior.{has_converged(), ..}`	swarm consensus primitives
`preserve(CONNECTIVITY, ..)`	swarm asserts

REFERENCES

[1] 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
[2] Edmund Clarke. 1997. Model checking. In *Foundations of Software Technology and Theoretical Computer Science*. Lecture Notes in Computer Science, Vol. 1346. Springer Berlin / Heidelberg, 54–56.
[3] F. Gauthier, D. Letarte, T. Lavoie, and E. Merlo. 2011. Extraction and comprehension of Moodle's Access Control Model: A case study. In *PST 2011*. 44 –51.
[4] Siobhan Gorman, Yochi J Dreazen, and August Cole. 2009. Insurgents hack US drones. *Wall Street Journal* (December 2009).
[5] Fiona Higgins, Allan Tomlinson, and Keith M. Martin. 2009. Threats to the Swarm: Security Considerations for Swarm Robotics. *International Journal on Advances in Security* 2, 2&3 (2009), 288 – 297.
[6] F. Higgins, A. Tomlinson, and K. M. Martin. 2009. 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
[7] R. Jhala and R. Majumdar. 2009. Software model checking. *Comput. Surveys* 41, 4 (2009), 1–54.
[8] 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.
[9] J. Panerati, L. Gianoli, C. Pinciroli, A. Shabah, G. Nicolescu, and G. Beltrame. 2010. From Swarms to Stars: Task Coverage in Robot Swarms with Connectivity Constraints. In *2018 IEEE International Conference on Robotics and Automation (ICRA)*. [to appear].
[10] C. Pinciroli and G. Beltrame. 2016. Swarm-Oriented Programming of Distributed Robot Networks. *Computer* 49, 12 (Dec 2016), 32–41.
[11] 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.
[12] 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
[13] Noah Shachtman. 2011. Computer virus hits US drone fleet. *CNN.com* (2011).
[14] 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