Journal of Logical and Algebraic Methods in Programming 132 (2023) 100860

Contents lists available at ScienceDirect

Journal of Logical and Algebraic Methods in Programming

journal homepage: www.elsevier.com/locate/jlamp

# Verification of the ROS NavFn planner using executable specification languages

Enrique Martin-Martin[a], Manuel Montenegro[a], Adrián Riesco[a], Juan Rodríguez-Hortalá[b], Rubén Rubio[a,*]

[a] *Facultad de Informática, Universidad Complutense de Madrid, Madrid, Spain*
[b] *Independent researcher, Madrid, Spain*

ARTICLE INFO

*Article history:*
Received 8 January 2022
Received in revised form 8 February 2023
Accepted 9 February 2023
Available online 14 February 2023

*Keywords:*
Formal verification
Model checking
ROS
Maude
Navigation
Dafny

ABSTRACT

The Robot Operating System (ROS) is a framework for building robust software for complex robot systems in several domains. The *Navigation Stack* stands out among the different libraries available in ROS, providing a set of components that can be reused to build robots with autonomous navigation capabilities. This library is a critical component, as navigation failures could have catastrophic consequences for applications like self-driving cars where safety is crucial.

Here we devise a general methodology for verifying this kind of complex systems by specifying them in different executable specification languages with verification support and validating the equivalence between the specifications and the original system using differential testing techniques. The complex system can then be indirectly analyzed using the verification tools of the specification languages like model checking, semi-automated functional verification based on Hoare logic, and other formal techniques. In this paper we apply this verification methodology to the NavFn planner, which is the main planner component of the Navigation Stack of ROS, using Maude and Dafny as specification languages. We have formally proved several desirable properties of this planner algorithm like the absence of obstacles in the planned path. Moreover, we have found counterexamples for other concerns like the optimality of the path cost.

© 2023 The Author(s). Published by Elsevier Inc. This is an open access article under the CC BY license (http://creativecommons.org/licenses/by/4.0/).

## 1. Introduction

Autonomous robot systems [56], ranging from robotic arms to autonomous vacuum cleaners, are ubiquitous in our society. The tasks they are in charge of are increasingly complex; they might interact with or involve human beings (like delivering packages and autonomous cars, respectively), and they require a precision level that cannot be achieved by humans, leaving these tasks completely dependent of the robots' behavior. For these reasons, new tools and techniques for analyzing and verifying robot algorithms are required. In this paper we focus on *motion planning* [12], whose essence is finding a path between two spatial configurations. The particular definition of configuration depends on the robot; here we focus on ground mobile robots where a configuration is the combination of a position and an orientation in a 2D space. The

[*] Corresponding author.
*E-mail addresses:* emartinm@ucm.es (E. Martin-Martin), montenegro@fdi.ucm.es (M. Montenegro), ariesco@ucm.es (A. Riesco), juan.rodriguez.hortala@gmail.com (J. Rodríguez-Hortalá), rubenrub@ucm.es (R. Rubio).

https://doi.org/10.1016/j.jlamp.2023.100860
2352-2208/© 2023 The Author(s). Published by Elsevier Inc. This is an open access article under the CC BY license (http://creativecommons.org/licenses/by/4.0/).

search of this path is not static but interacts with the environment through the sensors of the robot, which might prevent some dangerous movements or reveal new obstacles, hence requiring the algorithm to find a new path.

The Robot Operating System [52,54] (ROS) is a framework for developing robot systems. It includes several tools and libraries for efficiently creating robust and complex robot software. Its foundational aim is to provide a collaborative environment to combine the most appropriate pieces for each particular task. In this way, experts in particular fields (e.g. computer vision or motion planning) can develop particular features and distribute them as software libraries extending ROS, while using the functionality implemented by others to complete their designs. The *Navigation Stack* [37] stands out among those software libraries. It provides a set of reusable components for programming ground robots with autonomous navigation capabilities, and it is the *de facto* standard for robot navigation in ROS.

The Navigation Stack is a safety-critical component for ROS: a navigation failure could lead to undesired situations, such as a robot vacuum cleaner falling down a flight of stairs; or even catastrophic, for example in ROS deployments into full size autonomous cars [1,34]. For that reason, we have worked on analyzing the planner component of the Navigation Stack—in particular its most mature and popular planner plugin NavFn—with the aim to get a high-confidence assurance about the safety and stability of the system. The Navigation Stack is mainly written in C++, making extensive use of macros, pointers, type coertions, and classes. C++ is quite a low-level language whose semantics are informally defined in international standards of more than a thousand pages, which nevertheless leave some undefined behavior that different compilers may resolve diversely. In addition to the lack of a formal semantics of reference, third-party formalizations do not cover all aspects of the language, and in particular those features we have mentioned to be present in the Navigation Stack code. Hence, it is hardly ever possible to make a purely formal analysis of a real C++ program. Moreover, to the best of our knowledge, there are not off-the-shelf tools able to verify arbitrary high-level properties of C++ programs through partial formalizations, unlike for other languages like Ada or C.

One possibility to verify the NavFn planner is using a mechanized semantics of (a subset of) C++ [48,53] suitable for proof assistants like Isabelle/HOL [47] or Coq [9], provided it covers all C++ features in our program. However, these semantics have been defined after a manual translation of the informal standard, with only the execution of a few selected examples as a sanity check. In particular, we do not know of any real piece of software that has been studied with these semantics. Moreover, this approach requires the verifier to understand the details of the particular formalization and to manually prove the intended properties step by step by using the proof assistant's tactics, which is a considerable amount of work that usually makes the verification effort impracticable. The $\mathbb{K}$ Framework [22] facilitates this approach by offering a dedicated formal language to specify operational semantics and several tools to run analysis on them like model checking, test case generation, theorem proving, etc. Its formalization of the C language has been systematically tested against the mainstream compilers by comparing their outputs using thousand of programs, and it has been already applied to analyze real software systems. However, no semantics for the C++ language is available in K, and the translation from C++ to C is not always trivial and implies an error-prone manual work. This translation is automatized by the Frama-Clang plugin[1] of the Frama-C tool [32], a set of interoperable analyzers for C programs including verification in Hoare-logic. Frama-Clang translates C++ to C using the Clang compiler infrastructure,[2] but the caveats in the plugin page state that it "is currently in an early stage of development" and "it is known to be incomplete and comes without any bug-freeness guarantees", which reinforces our view that the translation (manual or automated) is not as straightforward as it may seem. Moreover, a usual criticism from the $\mathbb{K}$ community to Frama-C is that their analyses are not based on a formal semantics of the language but on ad hoc implementations, which may not be as trustworthy. Finally, another possibility for the verification of the NavFn planner is classical *testing*, i.e. to build an extensive test suite for the original NavFn C++ planner and test the intended properties *in those test cases*. However, testing does not provide formal guarantees and its evidence may be strongly affected by the potential biases or incompleteness of the sample. In summary, we have not found any system, tool, or approach that perfectly fits our verification needs for the NavFn planner.

Combining the promising possibilities of the previously considered techniques, we have developed an alternative general verification methodology to verify systems like the NavFn planner, for which no tool completely solves the verification problems. In this general methodology we will produce a high-level executable specification of the original system, compared against that original system with an extensive test suite, and then apply native tools in the specification language to verify the intended properties, usually by means of deductive methods. The general verification methodology is composed by the following steps:

1. generate an extensive test suite to exercise the original system in a wide variety of situations,
2. specify the system in a high-level, executable specification language and corroborate that it is equivalent to the original system by means of differential testing using the test suite, and
3. verify different aspects of the specification easily at that abstraction level with the different tools provided by the specification language.

[1] https://frama-c.com/fc-plugins/frama-clang.html.
[2] https://clang.llvm.org.

We claim that our methodology is general in the sense that it imposes mild requirements on the system it is applied to. In particular we need to (1) be able to run the system as many times as needed, in order to execute each test case; and (2) be able to observe the system's behavior, in order to compare it with the behavior reported by the executable specification. The system's source code is not a requirement because the test suite can be tested as a black box. However, we can obtain higher confidence in the results if we apply white-box testing and compare not only the observable behavior, but also the intermediate results. Besides this, our methodology is generic also in the sense that it can be used with any verification framework, provided that the specification language of the latter is executable.

Note, however, that the proposed verification methodology does not provide absolute guarantees that the properties proved in the specification will be preserved in the original system. The handcrafted specification may not be completely accurate due to human errors or omissions, while the coincidence of specification and original system on an extensive test suite cannot provide actual certainty of their full equivalence. However, we think that the combination of both manual specification and testing provides a strong confidence that the properties proved in the high-level specification, usually using formal deductive methods, can be attributed to the original system. Let us explain it with a metaphor. While two arbitrary mathematical functions whose images coincide in several arguments are not likely the same, they must surely coincide if they are actually polynomials of an order surpassed by the number of samples. Likewise, there is some regularity in the original system and the specification satisfying the desired properties, which makes the matching by differential testing between them more significant than the mere satisfaction of the properties in the test cases. Stronger confidence is obtained with higher-quality test suites and simpler specifications. In this sense, our verification methodology can be considered as an alternative to the previously presented approaches for verifying complex systems.

Although the methodology can be instantiated with any executable specification language, in this work we have focused on two of them: Maude [14], a high-performance logical framework based on rewriting logic, and Dafny [36], an imperative and functional programming language supporting formal specification and semiautomated verification in Hoare logic, i.e. through preconditions, postconditions, and loop invariants. Although having several different specifications of the same planner is not required to verify properties, we have decided to apply the general verification to two unrelated specification languages like Dafny and Maude as an indication of the applicability of our approach. Note that this general approach could also be instantiated with other target languages not purely related with specification but having a support of verifying tools, as is the case of the C language with the previously mentioned $\mathbb{K}$ framework and Frama-C. However, we have decided to explore the possibilities of specification languages because they usually have a more powerful and extensive set of verification tools.

We have reimplemented the NavFn planner of the Navigation Stack both as a Dafny program and as a Maude system module. We have then employed differential testing techniques to check that they compute the same paths—up to a reasonable level of numerical precision—as the original C++ version. For this differential testing phase, we have elaborated a set of different maps and paths that thoroughly exercise the original code of the NavFn planner, according to different notions of code coverage. In general, differential testing cannot prove with an absolute guarantee that the two systems exhibit the same behavior, even if the tests reach a high coverage level, since they still cannot cover every possible input. However, our aim is to provide a high level of confidence in a context where the usage of more complete verification tools is not feasible. The more comprehensive our test suite is, the more confidence we have on the similarity of the two systems, and the more support when we extrapolate the conclusions we get from analyzing the functional behavior of the Maude and Dafny specifications to the original C++ implementation, even if we know that we cannot reach absolute certainty.

Along the way of differential testing, we found counterexamples for the optimality of the algorithm in terms of path cost, although this concern has been purposely slighted in the design of the algorithm in a compromise with speed.

We have also been able to prove some relevant properties of the NavFn planner, like the absence of collisions with obstacles in the generated paths, as well as some other intermediate properties. Like many path-planning algorithms, NavFn proceeds by first computing an auxiliary structure called *navigation function* and then using it to compute the desired path. We have formally proved using Hoare logic under Dafny that this navigation function is well-defined and appropriate for computing a path. Then, we have checked that the generated paths do not contain obstacles. Using Maude, we have model checked some temporal properties telling that the A* search that computes the navigation function never explores obstacle positions, and that no path is computed until the navigation function is defined in the origin of the path.

In summary, our main contributions are:

1. A general verification methodology for complex systems based on an auxiliary implementation in an executable specification language, checked for equivalence against the original system by differential testing.
2. An extensive test suite for the ROS NavFn planner with the greatest possible coverage of its C++ code regarding lines, regions, and conditions. These tests have revealed counterexamples attesting that NavFn does not always find optimal paths in terms of cost and smoothness.
3. Dafny and Maude executable specifications of the ROS NavFn planner. Using differential black-box and white-box testing, and comparing the intermediate navigation functions, we have obtained strong confidence that the Maude and Dafny specifications have a similar behavior to the original ROS NavFn implementation.
4. A formal proof using Hoare logic under the Dafny specification that paths produced by the navigation algorithm do not collide with obstacles. Moreover, we have also proved that the navigation function satisfies some desirable properties.

5. A formal assessment using the built-in Maude model checker that some invariants relative to the good behavior of the algorithm are satisfied for every case of the test suite.

The current article is an extension of the informal publication [39]. The novel contributions of this work are the generalization of the approach based on differential testing, the separation of the Dafny specification as a new instance of this general procedure independent of the Maude specification, the introduction of white-box testing and the comparison of intermediate results to check the internal equivalence of the implementations, the application of stronger coverage metrics to support the quality of the test suite, the extension of the test suite itself with new maps and new ways of arranging obstacles, and the automation of the model-checking analysis to be applied to all maps from the differential testing suite. Moreover, we have improved the presentation of the contributions, with more detailed explanations and the simplification of the unnecessary complexity of the original publication.

The rest of the paper is organized as follows: Section 2 presents the basics of the ROS system, the overall design of the NavFn navigation algorithm, and the executable specification languages used; Section 3 describes the general methodology for the verification of complex systems using executable specification languages; Section 4 presents the differential test suite developed for the NavFn planner, including its coverage analysis and the counterexamples found; Section 5 describes the Dafny implementation and its formal verification with regard to the obstacle avoidance property; Section 6 explains the Maude specification and the application of model checking on it; Section 7 discusses related work; Section 8 summarizes the main points to take into account when following this methodology; and Section 9 concludes and presents some topics of future work. The complete code of the Maude and Dafny implementations, integration, tests, and proofs is available at [40].

## 2. Preliminaries

In this section we introduce the Robot Operating System, explain the implementation details of the NavFn planner, and present the main features of the specification languages used in this work.

### 2.1. The Robot Operating System

The Robot Operating System (ROS) is a framework for developing robotic systems. ROS is not a true operating system, but a set of programs and libraries that run on existing operating systems. At its core, it is a middleware for concurrent and distributed computation, used to write small programs (called *nodes*) that communicate through asynchronous message passing, using a topic-based publish-subscribe messaging pattern.

One of the keys to the success of ROS is its modularity and extensibility. ROS nodes can be distributed as ROS packages, which can be published by any user to the public ROS index.[3] ROS applications can use the nodes available in those packages by connecting them together through topics, maximizing code reuse. This has led to a wide community both in academia and industry [26]. As already mentioned, in this paper we focus on the Navigation Stack. Its latest version is called *Navigation 2*, and it has already reached a certain maturity level, as demonstrated in [38] where two robots were able to navigate without issues for over 37 miles (ca. 60 km) in less than 23 hours through a university campus full of students.

Navigation 2 has a modular design in which the actual path computation is done by configurable components. The planner component has a simple interface with a procedure ComputePathToPose that, given a goal *pose*, i.e., a position together with an orientation, should return a sequence of poses that can be followed to reach the goal position from the current one. The planner also needs some additional contextual information that is obtained from other nodes in the ROS application: an estimate of the current pose of the robot, and a map of the environment where the robot is operating.

### 2.2. Navigation algorithm

The NavFn planner, based on the Global Dynamic Window approach [11], was the only planner available in ROS until very recently, and it still is the default one. Maps are described as discrete rectangular grids stored in bidimensional integer matrices, where every cell is associated with a traversing cost. Cost values range from 50 (COST_NEUTRAL) to 254 (COST_OBS), where only 254 represents a *lethal* or insurmountable obstacle. Given such a map, the start and goal positions of the requested path, and the length of a single step, the algorithm produces a path satisfying these requirements. As shown in the pseudocode of Algorithms 1 and 2, which is based on the navfn.cpp file of the Navigation 2 repository [55], navFn proceeds in two phases:

1. In a first phase, calcPotential, an auxiliary navigation function or potential array is calculated on the cells of the map, assigning lower potentials to positions closer to the path destination.
2. In a second phase, called calcPath, this array is traversed in the direction of its negative gradient to calculate the path.

[3] https://index.ros.org/packages.

The navigation function is computed by either the Dijkstra or the A* search algorithms at the user's choice, starting from the goal position until the origin is reached, so that the value of the function in each cell is ideally its distance to the destination.

**Algorithm 1** Pseudocode of the NavFn algorithm (potential).

```
1: function NAVFN($map$ : matrix⟨int⟩, $start$, $goal$ : pose, $potIterMax$, $pathIterMax$ : int) : list⟨pose⟩
2:     $navfn \leftarrow$ calcPotential($map$, $start$, $goal$, $potIterMax$)
3:     if $navfn[start] < \infty$ then
4:         return calcPath($map$, $start$, $goal$, $navfn$, $pathIterMax$)
5:     else
6:         return no path
7:     end if
8: end function
9:
10: function CALCPOTENTIAL($map$ : matrix⟨int⟩, $start$, $goal$ : pose, $ncycles$ : int) : matrix⟨float⟩
11:     $threshold \leftarrow d_2(start, goal) + \mathsf{COST\_OBS}$
12:     $navfn \leftarrow$ matrix with the same size as $map$ filled with $\infty$
13:     $curr, next, over \leftarrow$ empty queues of poses
14:
15:     $navfn[goal] \leftarrow 0$
16:     append the cardinal neighbor cells (in dir. N, S, E, W) of $goal$ to $curr$
17:     repeat at most $ncycles$ times
18:         while $curr$ is not empty do
19:             $p \leftarrow curr$.extractFirst()
20:             if $map[p] < \mathsf{COST\_OBS}$ then                                  ▷ skip obstacles
21:                 $v \leftarrow$ propagated potential ($\sim$distance) from neighbors
22:                 if $v < navfn[p]$ then
23:                     $navfn[p] \leftarrow v$
24:                     for every cardinal neighbor $p'$ of $p$ do
25:                         if $v + d_2(p, start) + 1/\sqrt{2} \cdot map[p'] < navfn[p']$ then
26:                             if $v + d_2(p, start) < threshold$ then
27:                                 append $p'$ to $next$
28:                             else
29:                                 append $p'$ to $over$
30:                             end if
31:                         end if
32:                     end for
33:                 end if
34:             end if
35:         end while
36:         swap $curr$ with $next$
37:         if $curr$ is empty then
38:             $threshold \leftarrow threshold + 2 \cdot \mathsf{COST\_NEUTRAL}$
39:             swap $curr$ with $over$
40:         end if
41:     until $navfn[start] < \infty$
42:     return $navfn$
43: end function
```

The implementation of A* in NavFn is not a textbook one. Its main peculiarity is that there is no priority queue for the nodes to be visited next, but a collection of FIFO queues that separate nodes whose total path estimation with the Euclidean distance is below and above a growing threshold. More precisely, these queues are *curr* for the positions to be expanded in the current iteration, *next* that will become the *curr* of the next iteration (line 36 of Algorithm 1), and *over* with the positions whose total path estimation exceeds the current threshold. When an iteration ends with an empty *next* queue (line 39), the threshold is stepped by a fixed amount and *over* becomes curr. When a position $p$ in *curr* is processed, its neighbors are added to *next* or *over* queues depending on whether the estimated distance is below or above the current *threshold* (line 26), respectively. Neighbors are only added to the queues if their potential improves, as specified in line 25. As we will see, the threshold approach does not guarantee the optimality of the distances in the potential map or the calculated paths, and not even a tight approximation, but it is fast and its behavior is satisfactory in most practical situations. If the origin is not reachable, the algorithm will stop after exploring all reachable cells or exceeding a given iteration limit.

We should point out that the condition in line 25 of Algorithm 1 is inconsistent because it compares the lengths of two essentially different paths. Indeed, the right operand is the previously calculated distance from $p'$ to *goal*, since this is the approximate meaning of $navfn[p']$. However, the left operand represents the estimated total cost of a path from *origin* to *goal* through $p'$, because it adds to $navfn[p']$ an estimation $d_2(p, start) + 1/\sqrt{2} \cdot map[p']$ of the rest of the path. In most cases, $navfn[p']$ will be the initial infinity value, and this would not make any difference.

Once the navigation function is computed, NavFn's calcPath (see Algorithm 2) slides down the potential map from the origin towards the destination, whose potential is zero. In each iteration, the algorithm either moves to the neighbor cell with the lowest potential (line 14) or advances a fractional distance stepLength in the direction of its gradient (line 24),

**Algorithm 2** Pseudocode of the NavFn algorithm (path).

```
 1: function CALCPATH($map$ : matrix⟨int⟩, $start$, $goal$ : pose, $navfn$ : matrix⟨float⟩, $ncycles$ : int) : list⟨pose⟩
 2:     $path \leftarrow$ empty list of poses
 3:     $p \leftarrow start$                                              ▷ current position (integer)
 4:     $dx, dy$ : float $\leftarrow 0, 0$                                ▷ current offset (fractional)
 5:
 6:     for $ncycles$ rounds do
 7:         if $navfn[p + \text{rounded } (dx, dy)] < \text{COST\_NEUTRAL}$ then
 8:             append $goal$ to $path$
 9:             return $path$
10:         end if
11:         append $p + (dx, dy)$ to $path$
12:         $oscilation \leftarrow |path| > 2 \wedge path[|path| - 1] = path[|path| - 3]$
13:         if $\exists$ neigh. $c$ of $p$ (inc. diagonals) s.t. $navfn[c] = \infty \vee oscilation$ then
14:             $c \leftarrow$ neighbor cell with the lowest potential
15:             $p, dx, dy \leftarrow c, 0, 0$
16:             if $navfn[c] = \infty$ then
17:                 return no path
18:             end if
19:         else
20:             $g \leftarrow -\nabla navfn[p + (dx, dy)]$                  ▷ interpolated gradient
21:             if $g = (0, 0)$ then
22:                 return no path
23:             end if
24:             $dx, dy \leftarrow (dx, dy) + \frac{\text{stepLength}}{\|g\|} \cdot g$
25:             normalize $p$, $dx$, and $dy$ so that $|dx|, |dy| \leq 1$
26:         end if
27:     end for
28:     return no path
29: end function
```

depending on the presence of nearby obstacles (line 12). Iterations are limited by a bound parameter, so the algorithm always terminates, but perhaps without finding a path. In principle, the navigation function is free of local minima where the search may get stuck, but oscillations around saddle or similar points are possible. The implementation attempts to detect oscillations by checking whether the current point coincides with the point two positions before in the path (see line 12), and then interrupts the oscillations by jumping to a neighbor cell with lower potential. However, this comparison uses exact equality between floating-point numbers and overlooks cycles with a longer period, so some oscillations may escape this method and consume all the iterations without reaching the destination. Moreover, the repeating points remain in the path until they are detected, unnecessarily increasing its length.

### 2.3. *Executable specification languages*

Executable specification languages are formal languages that allow for describing systems in a much higher level than a typical programming language, are executable or at least able to produce executable code, and usually come with tools for checking different properties of their specifications. Being executable is an essential requirement in the verification approach presented in Section 3 because a test suite must be run by the specifications. Two examples of these languages are Dafny and Maude.

Dafny [36] is an imperative language and verifier based on *Hoare logic*, whose programs must be annotated with preconditions, postconditions, and invariants to formally describe their semantics. These annotations induce proof obligations that are semi-automatically checked with a Satisfiability Modulo Theories (SMT) solver and perhaps with the help of some assertions and lemmas introduced by the user within the program. Verified programs can be compiled to executable code through transpilation.

On the other hand, Maude [14] modules correspond to specifications in *rewriting logic* [42]. This logic is constructed, in the case of Maude, as an extension of *membership equational logic* [10], an equational logic that, in addition to equations, allows for the statement of *membership axioms* characterizing the elements of a sort. Rewriting logic extends membership equational logic by adding rewrite rules, which represent transitions in a concurrent system and can be nondeterministic. Maude provides a built-in *search* command, which performs a breadth-first search using rewrite rules, and a model checker to analyze Linear-time Temporal Logic (LTL) properties in Maude specifications [21]. Maude's rewrite engine provides several commands to efficiently execute specifications, making Maude a powerful declarative programming language and analysis tool.

## 3. Verification methodology using executable specification languages

In this section we present the general verification procedure we will follow to verify the NavFn planner. This verification methodology only imposes a particular workflow that can be instantiated to any complex system and executable specification language. Here, by *complex system* we mean any hardware-software system for which there are not enough verification

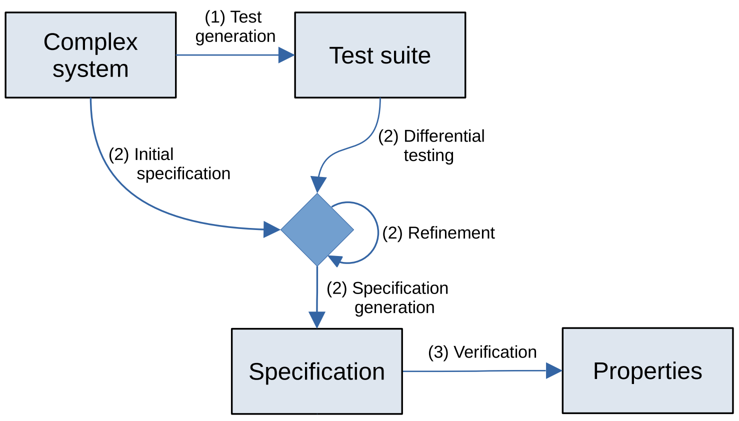

**Fig. 1.** Workflow of the verification methodology.

tools available in their native environment. This includes systems as the NavFn planner that is implemented in C++, a programming language without a well-established formal semantics or strong verification tools, but it also includes distributed systems composed by several components (software, hardware, or mixed) that cooperate to produce an observable behavior. This observable behavior of the system is essential because we need to elaborate a set of tests to support that the developed specification of the system does behave the same as the original system, so at least we have to obtain and compare this black-box observable output. However, having more knowledge about the inner workings of the complex system helps us create a more robust test set (v.g. considering code coverage of the original system code) and also a more complete testing process (v.g. by comparing intermediate results or using white-box testing to check that the internal states evolve the same both in the system and in the specification). The main phases in the verification methodology are the following, which are depicted in Fig. 1 and explained in detail in the rest of this section:

1. Generate a test suite to exercise the complex system in a wide variety of different scenarios. In order to check that tests are comprehensive, coverage analysis is highly recommended.
2. Produce a specification that mimics the behavior of the complex system using an executable specification language, and check they behave the same by using the test suite. The definition of the specification is an iterative process that benefits from the previously generated test suite, in what can be seen as a test-driven development [7].
3. Using the tools of the specification language, state and prove in the specification the expected properties of the original system.

Note that this verification methodology can be instantiated with several executable specification languages for the same complex system, as each of them can be better suited to handle a specific kind of properties. However, the whole process does not need to be replicated in every instance because, once a extensive and robust test suite is generated, it can be reused for any specification of the same complex system.

## *3.1. Generation of the test suite*

The first phase of the verification methodology is to generate a set of tests that will be used to later corroborate the specification behaves equivalently to the original complex system. This way, we would be able to reason on the specification defined in the specification language and extrapolate the results to the original system. Formally validating a specification w.r.t. the behavior of a complex system is a fundamental problem in software verification and it is usually impossible to provide a perfect solution for real situations. Concretely, we are considering complex systems where we do not have a formal semantics, but only its implementation in some programming language or even in binary executable format. Therefore, to validate our specification we will consider the original system as a *pseudo-oracle* [6,64,17] to check that the results obtained by the specification are the expected ones. This technique, called *differential testing* [41], has been used in large software systems to compare different versions of a program produced independently, as in our case. This is not a perfect solution for checking whether a specification behaves the same as the original system for every possible input, but in our opinion is the most complete technique we can apply in this situation where we lack a formal semantics and appropriate verification tools.

When generating the test suite for differential testing, it is essential to be as comprehensive as possible, producing tests that exercise every possible scenario of the complex system. For this purpose is very relevant to have an intimate knowledge of the problem that the system solves, as well as its usual and corner cases. After detecting the different scenarios, an effective approach is to automatically generate a great number of randomized variations of each one. If the source code or the design of the complex system is available, it can be carefully inspected in order to detect the maximum number of scenarios and exercise them in some tests. Additionally, if the source code is available, there are several code coverage metrics [45] that can be used to evaluate the completeness of the test suite:

- *Line coverage*: lines of source code executed by the test suite.
- *Statement coverage*: statements executed by the test suite. Note that in some programming languages several statements or even the whole program can be written in a single line, so this metric is more fine-grained than line coverage.
- *Region coverage*: regions in the control-flow graph [3] of the program executed by the test suite. Each region may include several statements, which are executed the same number of times, and an statement may include several regions, v.g. `return x || y && z`, which is equivalent to an *if-then-else* expression under the short-circuit semantics of C++. Hence, region coverage is more informative than statement coverage.
- *Branch coverage*, also known as *decision coverage*: the execution of program branches (such as those created by *if-then-else* or *switch* constructions) follows both paths in the tests. For example, in an *if-then-else* expression like `if (x >= 0 || y > 5) {...} else { ... }` this metric measures whether the tests cover situations where the condition is evaluated to *true* and also situations where it is evaluated to *false*.
- *Condition coverage*: the Boolean sub-expressions in conditions are evaluated to both *true* and *false* when running the test suite. For example, in a condition like `x >= 0 || y > 5` this metric will check whether `x >= 0` and `y > 5` are evaluated to both *true* and *false* in different tests. Note that this metric is more fine-grained than branch coverage as the value of a complex Boolean condition can be the result of many different combinations of the truth values of its sub-expressions.
- *Path coverage*: paths of the control flow graph of a piece of code executed by the test suite (modulo loops, which produce infinitely many paths). This is a very strong coverage measure, but usually too costly, and there are no general tools for evaluating it.

The goal therefore is to obtain a test suite with the maximum code coverage possible considering these criteria. It is well-known that a high level of coverage by itself does not guarantee that the test suite considers all the possible behaviors of the program or will pinpoint failures [28]. However, these criteria are tools that can objectively signal improvements in the test suite. Therefore, we encourage using code coverage metrics when possible and use their reports to create new tests.

### 3.2. Specification of the system and differential testing

The second step in the verification methodology is to produce an executable specification in some specification language that mimics the observable behavior of the complex system, replicating its successful and erroneous results. For this purpose, it is often useful to inspect the code of the system and its components, if available, and try to define a specification following the same implementation details, mainly in terms of inner data structures. If neither the code or details about the implementation are available, the best approach is to define a specification based on the expected behavior of the system and refine it iteratively when the tests of the previously created test suite fail.

The most important decision in this phase is selecting the appropriate executable specification language. This choice will be strongly affected by the expressive power of the verification language and the facilities it provides to state and prove the different properties of the original system. One possibility is using proof assistants like Coq [9], Isabelle-HOL [47], and Lean [43]. These assistants provide a functional language to define the specification and also constructions to state the properties and prove them manually by means of different strategies. They can also export these functional specifications into mainstream functional or imperative programming languages (OCaml, Haskell, and C) but their support for I/O operations is limited, so automating the execution of the test suite is usually a complex task.

The Dafny platform [36] is also a good choice for specification. Compared to proof assistants, it requires less manual effort to prove the properties as proofs obligations are automatically extracted from preconditions, postconditions, and invariants, and sent to an SMT solver for checking them. The user may need to provide assertions or lemmas to help the SMT solver in establishing the validity of the specification, but none or small human intervention is required in many cases. From the point of view of executability, Dafny includes source-to-source compilers for several languages like C#, Go, and Javascript. The generated code can be extended in the corresponding language to provide the missing library functions and input/output facilities that are required to load the test cases and communicate the results.

Another possible executable specification language to instantiate the methodology with is Maude, whose specifications are directly executable. It has a good support for I/O operations, which is enhanced by the Python bridge [57] that allows one to use Maude modules directly from a Python program. Maude is a declarative, rule-based, specification language, so reasoning about Maude specifications is easy because there are no mutable constructs. Besides this, it is possible to execute Maude specifications step by step, which allows performing white-box testing by inspecting the fine-grained evolution of the inner data structures. Inductive theorem proving is also possible through the Maude ITP [15] and other tools [60].

### 3.3. Verifying properties

Finally, the last step of the verification procedure is to use the specification that has been differentially tested against the original complex system to prove some properties on it. The kind of properties that can be stated and proved varies among the different specification languages, each of which supports a different set of techniques and tools. Some examples of suitable verification approaches to be considered for complex systems are the following:

- Functional verification using Hoare logic. In Hoare logic, the behavior of any piece of code $S$, from a basic statement to a function, is specified with a so-called Hoare triple $\{P\}\,S\,\{Q\}$ where $P$ and $Q$ are two predicates called precondition and postcondition. The triple means that whenever $S$ is executed from a state satisfying $P$ and terminates, its final state satisfies $Q$. From the axiomatically-defined meaning of the basic predicates and by calculating *weakest preconditions* through the program, we can obtain a collection of logical implications whose satisfaction implies the correctness of the program against the specification. These logical implications, called *proof obligations* or *verification conditions*, can be passed to an SMT solver to automatize their evaluation. However, for complex programs, some human intervention will be usually needed to cover the gaps that the SMT solver cannot fill by itself. This is the kind of verification Dafny supports.
- Model checking against temporal properties. While the proposed verification methodology is mainly aimed at deterministic systems, where branching-time properties are not meaningful, linear-time model checking can be useful to study the temporal behavior of those systems. Its main advantage is the high degree of automation, although it must be applied on fixed inputs. Maude includes a builtin model checker for Linear-time Temporal Logic (LTL) [21], and supports the $\mu$-calculus [58] and LTL of Rewriting (LTLR) [4] through extensions.
- Symbolic execution. Instead of running a program over terms with concrete (ground) values for all its input arguments we can execute terms using variables in some of them. In this way, it is possible to check properties that should be valid for all instantiations of these variables, generalizing the process and covering all cases in a systematic way. Symbolic execution, search, and model checking are supported in Maude via narrowing [19,2,5,20]. Note that these techniques, as the ones above, are highly automated, although symbolic techniques are limited to some theories where the underlying algorithms (e.g. unification) are decidable.

## 4. Test suite for differential testing the NavFn Planner

As explained in the previous section, the first step of our verification methodology is to define an extensive test suite to exercise all the possible behaviors of the system. In the case of the NavFn Planner, we want to create a wide set of tests covering different types of bidimensional maps with varying patterns of obstacles and costs and starting and goal cells to compute paths between them. We have tried to detect the different situations that can arise when computing paths for an autonomous robot moving in a 2D map. For every situation, we have applied a high degree of randomization to generate as many variants as possible. To evaluate the quality of our tests, we have applied code coverage criteria to conclude that all NavFn behaviors are exercised by our test suite. Note that this test suite is completely independent of the specifications that will be defined later for the NavFn planner, so it will be used to compare both the Dafny and the Maude specifications (see Sections 5 and 6, respectively).

To exercise the NavFn planner in a wide enough variety of situations, we have considered 101 different maps and computed more than 120,000 paths on them (see Table 3 and the directory src/testing/tests in [40] for details). Note that all the maps in the test suite have obstacles in the edges because the ROS Navigation framework automatically overwrites the border cells of the map with obstacles when preprocessing it. This way, the original NavFn planner and the Dafny/Maude specifications receive the same view of each map.

We have used two sets of maps to cover as many scenarios as possible. First, we have generated a set of 96 maps with different dimensions and properties separated in 9 categories as follows. Fig. 2 shows a sample map for each category, where red cells represent obstacles and gray-scale cells represent navigable cells that get darker as the navigation cost increases. Coordinates $(x, y)$ use the Cartesian pixel coordinate system with $(0, 0)$ in the center of the cell at the upper-left corner, $x$ and $y$ being the horizontal and vertical movement, respectively.

a. circus: maps with the typical oblong rectangular track of Roman circuses. These maps have been handcrafted to refute the optimality of the algorithm, with a high contrast of costs between their two longitudinal lanes (see Fig. 2-a).
b. columns: maps with custom-sized columns as obstacles organized uniformly in a grid. There are maps where all navigable cells have the lowest possible cost COST_NEUTRAL (we say that these cells are *free* or *empty*) and others where navigable cells have a random cost (as in Fig. 2-b).
c. free: maps where all cells are empty but the surrounding walls (see Fig. 2-c).
d. obstacle: some obstacle cells are randomly generated in the map, and the rest of navigable cells have a random cost. There are maps with a different ratio of obstacles, producing complex maps with isolated spaces and diagonal paths (as shown in Fig. 2-d).
e. radial: the cost of the cells increases uniformly with the distance from a center point, and there are no obstacles. The size of the map, the position of the center point, and the radius of the slope varies between maps (depicted in Fig. 2-e).
f. random: maps without obstacles where the cost of each cell is randomly generated (illustrated in Fig. 2-f).
g. spiral: this category contains maps with a single spiral path, where the navigable cells can be completely empty (as in Fig. 2-g) or with a random cost. These maps exercise the planner with very long paths compared with the dimensions of the map.

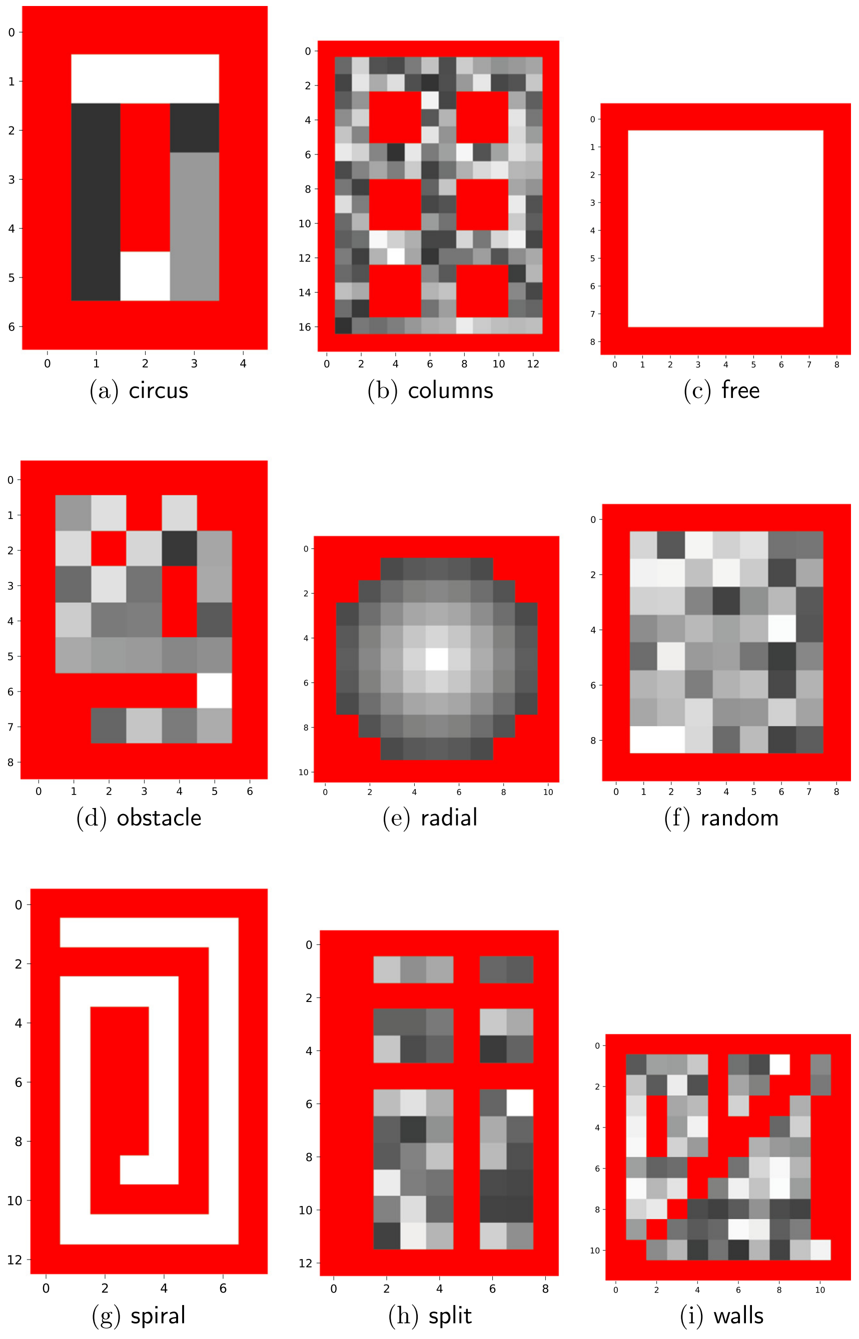

**Fig. 2.** Maps generated for validation (obstacles are marked in red, as can be observed in the web version of this article).

h. split: cells with random cost that are crossed by vertical and/or horizontal walls, therefore generating several isolated regions (shown in Fig. 2-h).
i. walls: maps that contain some random linear walls of obstacles of width 1. These walls do not need to go from border to border but can be included in inner parts of the maps. The high level of randomness, both in the walls and in the cost of navigable cells, generates complex maps with intricate paths (see Fig. 2-i).

These generated maps have both square and rectangular shapes, and for every map we have tried many of the possible paths between their cells. As the specification languages we consider (Dafny and Maude) are slower than C++, we have kept map dimensions small (at most $30 \times 30$) so that we could compute many paths. Since the algorithm is based on local decisions, there is no loss of generality by considering small-sized maps. However, we have also used the set of test maps

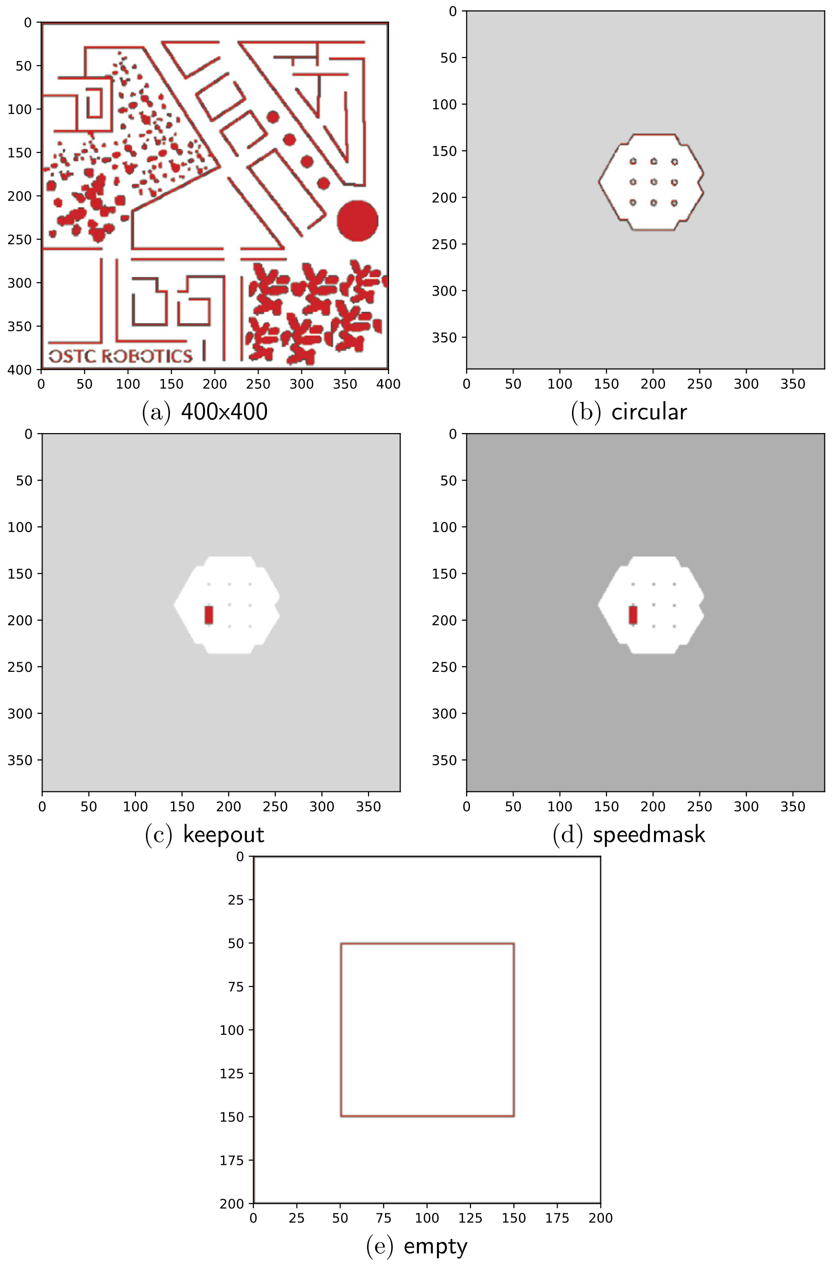

**Fig. 3.** ROS navigation2 test maps.

in the ROS Navigation 2 package,[4] which contains 5 maps of bigger dimensions (from 200 × 200 to 400 × 400), as can be seen in Fig. 3. They are one big and complex map (400x400), three versions of a hexagonal room containing 9 columns with different costs and obstacles (circular, keepout, speedmask), and one empty map with only one squared room (empty). For the same efficiency reasons as before, we have considered only 190 paths within them.

Table 1 summarizes the number of maps generated by category and the number of tested paths. For each category we have automatically generated between 10 and 20 maps of different sizes and thousands of paths. The only exception is the circus category, with only 3 maps and 5 paths. The reason of these small numbers is that this category was only created to pinpoint a non-optimal behavior of the navigation algorithm (see Section 4.1), so we crafted manually only those scenarios that show that desired behavior. As mentioned before, the ROS category contains 5 maps because these are the only maps contained in the ROS Navigation 2 repository for testing purposes.

[4] https://github.com/ros-planning/navigation2/tree/main/nav2_system_tests/maps.

**Table 1**
Number of maps and paths tested.

| Category | Number of maps | Number of paths |
|---|---|---|
| circus | 3 | 5 |
| columns | 10 | 17, 584 |
| free | 10 | 3, 708 |
| obstacle | 11 | 7, 717 |
| radial | 10 | 12, 145 |
| random | 12 | 9, 182 |
| spiral | 10 | 27, 212 |
| split | 10 | 15, 719 |
| walls | 20 | 27, 947 |
| ROS | 5 | 190 |
| **TOTAL** | 101 | 121, 409 |

**Table 2**
Code coverage of the A* methods of the navfn.cpp file under the test suite.

| Criterion | Raw | | Possible |
|---|---|---|---|
| | % | Fraction | % |
| Lines | 93.49 | 287/307 | 100 |
| Regions | 74.31 | 457/615 | 100 |
| Conditions | 70.30 | 232/330 | 100 |

As the C++ code of the NavFn planner is available, we have evaluated the completeness of the test suite considering different metrics of code coverage: line, region, and condition coverage, as explained in Section 3.1. Table 2 shows the coverage numbers and percentages obtained for each criterion. Even though 20 lines, 158 regions, and 98 conditions are uncovered, we claim that the test suite has the greatest possible coverage of the NavFn algorithm because the missing entities are unreachable due to invariants of the algorithm. For example, some methods that operate on a given cell check whether it is inside bounds, but this is ensured by the caller. Another method checks whether the given cell is an obstacle, but this is explicitly discarded by the caller. The full coverage report annotated with the reasons why visiting each uncovered entity is not possible is available at https://demiourgoi.github.io/ROS_navfn_verification/coverage. These results have been obtained with the source-based code coverage analysis of Clang and the llvm-cov tool.[5] Neither this tool nor other well-established coverage software we have tried support more sophisticated notions of coverage like path coverage to check that all possible paths of execution are executed at least once. However, we are confident that any specification passing these tests will exhibit the same behavior as the ROS NavFn, since region and condition coverage are quite fine-grained measures.

## 4.1. Counterexamples of the navigation algorithm

An interesting side-product of the generation of tests for differential testing is that we found concrete examples showing that the cost and regularity of the paths computed by the algorithm are not optimal (apart from the oscillations mentioned in Section 2.2, which were assumed). Two examples of these situations appear in Fig. 4, where the obtained paths are shown in blue:

- Fig. 4 (left) shows a path that is unnecessarily longer: after the third point at $(2.48, 2.47)$ it takes a diversion towards the point $(2.74, 1.96)$ that is then retraced with a drastic turn. Note that this non-optimal solution does not provide a "longer but smoother path", as it requires a turn of almost $180°$.
  This issue is an effect of the gradient-based traversal of the navigation function,[6] which may unnecessarily lead the path to a cheaper cell despite the global growth of the path length and cost. Obtuse angle turns occur in a 1.91% of the paths of the test suite (excluding oscillations, which appear in a 0.13% of the paths).
- Similarly, the map of Fig. 4 (right) shows two possibilities for going from the lower right cell $(3, 12)$ to the upper middle cell $(2, 1)$: using the left or right lanes. Even though the cells in the left lane have lower costs, the algorithm takes the right one and produces a path whose cost is 700 units higher (and here both paths are equally smooth).
  The cause of this issue is the fact that the algorithm expands its front of visited cells one layer at a time and stops as soon as the origin is reached, hence losing the opportunity to find a cheapest path that traverses more cells.

[5] https://clang.llvm.org/docs/SourceBasedCodeCoverage.html.
[6] The direction of the negative gradient is represented by means of green arrows in Fig. 4.

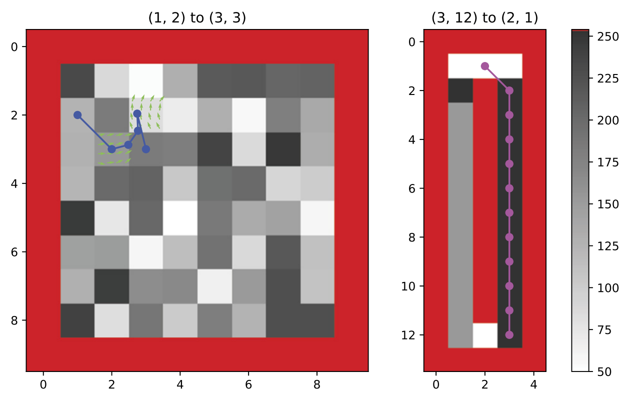

**Fig. 4.** Non-optimal paths found by the ROS navigation algorithm.

**Listing 1** Specification of navigation algorithm.

```
method AStar(start: Pose, goal: Pose, cm: CostMap, numIterations: nat) returns (error: bool, path: Path, navfn: PotentialMap)
  // Preconditions:
  requires 0 ≤ start.pos.row < cm.numRows
         ∧ 0 ≤ start.pos.col < cm.numCols
  requires 0 ≤ goal.pos.row < cm.numRows
         ∧ 0 ≤ goal.pos.col < cm.numCols
       // Start and goal poses lie within the cost map
  requires Open(cm, goal.pos.row, goal.pos.col)
       // Goal position is free of obstacles (Listing 2)
  requires ValidCostMap(cm)
       // Cost map has non−negative costs (Listing 2)

  // Postcondition:
  ensures ¬error ⟹  // If a path is found
    |path| ≥ 1
       // Path contains at least one position
    ∧ path[0] = RealPoint(start.pos.row as real, start.pos.col as real)
       // Path starts at the start position
    ∧ path[|path|−1] = RealPoint(goal.pos.row as real, goal.pos.col as real)
       // Path finishes at the goal position
    ∧ ValidPath(path, costMap)
       // All positions of the path are within cells without obstacles
       // (see Listing 3)
```

Nevertheless, we cannot consider these examples as failures of the algorithm because its design (explained in Section 2.2) is not oriented towards optimality but implements a compromise between the cost of the resulting path and the speed of its calculation. Moreover, the smoothness of paths is not a concern in the design of the planner.

## 5. Verification of NavFn using Dafny

In this section we explain how we have defined the specification of NavFn in Dafny, how we have applied the test suite for differential testing, and which properties of NavFn we have been able to prove with the Dafny specification.

### 5.1. Dafny specification

The Dafny implementation of the NavFn algorithm is an imperative program that imitates the original NavFn planner with a collection of similar methods (procedures) and functions (see folder src/dafny in [40]). The main one, AStar, specified in Listing 1, receives the input map and an initial position and produces a path (or an error flag) by computing the navigation function (navfn) and then computing the path from that function. The whole implementation is annotated with 94 preconditions, 53 postconditions, 13 invariants, 84 assertions, and 15 lemmas to prove their correctness in 31.78 seconds. While Dafny supports dynamic memory allocation, we have used immutable data types for the positions, paths, maps, and other structures of the implementation in order to abstract heap-related issues (such as framing) which would have been unavoidable if we had tried to verify the original C++ algorithm directly. The floating-point and integer values of the original implementation are respectively represented using the arbitrary-precision built-in types real and int.

Once verified, the Dafny code is compiled to a library or a standalone executable, if a Main method is defined. In our case, we do not provide a Main method in Dafny because the only support for input and output in the language is a print

**Listing 2** Auxiliary predicates.

```
// This predicate tells whether a cell is free of obstacles
predicate Open(costMap: CostMap, row: int, col: int)
{
  costMap.value(Point(row, col)) < obstacleCost
  // The value of the cost map at position (row, col) is lower
  // than the obstacleCost constant, which is set to the cost of
  // going through an obstacle (that is, a very high cost).
}

// This predicate holds iff all the cells of a given cost map have
// positive costs
predicate ValidCostMap(costMap: CostMap)
{
  ∀ i, j | 0 ≤ i < costMap.numRows ∧ 0 ≤ j < costMap.numCols ::
       costMap.value(Point(i, j)) > 0.0
}
```

**Listing 3** Specification of a valid path (i.e. a path that does not run into obstacles).

```
// This predicate tells whether all positions of a given path belong
// to cells without obstacles
predicate ValidPath(path: Path, costMap: CostMap) {
  ∀ p | p ∈ path ::
     ∃ i, j :: 0 ≤ i < costMap.numRows ∧ 0 ≤ j < costMap.numCols
          ∧ BelongsToCell(p, Point(i, j))      // (1)
          ∧ Open(costMap, i, j)                // (2)

  // For every position p in the path there is a cell (i, j) in the map
  // such that (1) p is contained within the surface of the cell
  //        and (2) the cell does not have any obstacles
}

// It tells whether a point is contained within the area of a given cell
predicate BelongsToCell(p: RealPoint, cell: Point)
{
     −0.5 ≤ p.row − cell.row as real ≤ 0.5
  ∧ −0.5 ≤ p.col − cell.col as real ≤ 0.5
}
```

statement. In order to read test cases, feed the AStar function with them, and output the results, glue code in an external programming language should be written. The traditional and main intermediate language for compilation in Dafny is C#, but other languages are supported. We have chosen Go[7] because of the maturity of its back-end in Dafny and small runtime dependencies of the generated programs. The generate-go.sh script in the src/dafny directory of the repository [40] generates an executable combining the verified Dafny-generated code with the input/output extension written in Go itself. Moreover, the Go code also provides some mathematical functions like the square root of a real number, which are not directly available in Dafny. In addition to these extensions, the script applies a patch to replace the implementation of the real type based on arbitrary-precision rationals (math/big.Rat in the Go standard library) with an implementation based on the machine single-precision floating-point type (float32 in Go). Otherwise, the execution times of the NavFn planner become impractical. Indeed, we have profiled the implementation based on rational numbers under a selection of test cases and almost a 90% of the execution time is spent normalizing the fractions to their irreducible forms. In order to perfectly adjust the floating-point arithmetic to the implicit C++ type conversions of the original implementation, some other small patches are applied. For example, the C++ expression

```
float v = −0.2301 ∗ d ∗ d + 0.5307 ∗ d + 0.7040;
```

has been copied verbatim to the Dafny code. However, it is equivalent in C++ to

```
float v = float(−0.2301 ∗ double(d) ∗ double(d) + 0.5307 ∗ double(d) + 0.7040);
```

where products and sums are evaluated using double-precision floating-point numbers and then the result is narrowed to single precision. On the contrary, the Go code generated by Dafny (where the Dafny type real is implemented by the Go type float32) always operates in single precision, yielding potentially different values that may produce qualitatively different

[7] https://go.dev/.

**Table 3**
Running times of differential testing for each implementation, in seconds.

| Category | Original | Maude | Dafny+Go |
|---|---|---|---|
| circus | 0.03 | 0.30 | 0.00 |
| columns | 0.31 | 143.71 | 22.68 |
| free | 0.13 | 18.64 | 3.18 |
| obstacle | 0.12 | 8.35 | 1.38 |
| radial | 0.22 | 82.84 | 14.31 |
| random | 0.22 | 94.87 | 14.84 |
| spiral | 0.41 | 169.67 | 25.19 |
| split | 0.19 | 71.35 | 10.38 |
| walls | 0.43 | 157.90 | 22.46 |
| ROS | 0.19 | 4449.63 | 194.12 |
| **TOTAL** | 2.25 | 5197.26 | 308.54 |

program outputs. Because these type conversions cannot be specified within Dafny, we have explicitly enforced them in the generated Go code (for the expression above, which is part of the potential propagation in line 21 of Algorithm 1, and the expressions involved in the gradient interpolation).

## 5.2. *Differential testing of the Dafny specification*

In order to check the conformance between the original ROS planner and its replica in Dafny, we have run the above explained Dafny+Go implementation on the test suite in Section 4. We have concluded that, for every test case,

- the paths produced by Dafny and ROS are identical,
- the auxiliary navigation functions computed by Dafny and ROS are identical, and
- the states of the A* searches in both implementations match iteration by iteration.

We have compared paths and potential matrices using strict equality between their single-precision floating-point numbers. In other words, the binary outputs of Dafny and ROS coincide bit by bit. For white-box testing the intermediate states of the potential calculation, we have run both implementations in sync while comparing their data structures. Thanks to the Python API of the LLDB debugger,[8] we have checked with breakpoints at the beginning of each iteration of the A* algorithm (line 18 of Algorithm 1) that the positions in the internal queues are the same.

Regarding performance, Table 3 shows the execution times of the original ROS, Dafny+Go, and Maude implementations on each category of tests.[9] The original C++ implementation is orders of magnitude faster than Dafny+Go, but running the whole test suite only takes 5 minutes with the latter.

## 5.3. *Verification of properties*

Let us recall that the navigation algorithm consists of two steps: the computation of a potential map and the computation of the path itself (see Section 2 for details). The potential map measures the proximity of each cell to the goal position. Initially all cells are assigned a special potential value $+\infty$, except for the goal itself, whose potential is zero. As the potential map is computed, some cells are assigned a finite potential value. If that happens for the cell containing the start position, then a path between the start and goal positions is guaranteed to exist. With regard to the properties involved in the computation of the potential map (specified in Listing 4), we have proved in Dafny that the cells whose potential become finite (1) do not contain obstacles (called *position safety*), and (2) for each of these cells, either it contains the goal position, or there is an adjacent cell with lower potential (called *progress*). The first property allows us to ensure that any path that only passes by cells with finite potential does not intersect with obstacles. The second property guarantees that whenever the cell containing the start position gets finite potential, there exists a path from that position to the goal. Both properties (see Listing 5) depend on a *cost map*, which specifies which locations of the environment are open (i.e. do not contain obstacles). The PositionSafe predicate checks the first property explained above, whereas Progress checks the second property.

Given the potential map, our Dafny implementation of the calcPath function of Algorithm 2 returns, besides the list of positions of the computed path, another list (named closest) with the cells of the cost map that contain each position of the path. The latter list is a *ghost result*, i.e., it is used only for verification purposes and it is not computed at runtime.

[8] See https://lldb.llvm.org/. Unlike for the ROS-Maude comparison, we have used LLDB instead of GDB because of its more sophisticated Python API that allows running two processes live in sync.
[9] The original NavFn planner, whose initialization times are larger, is faster in the ROS category despite its bigger maps because it consists of fewer maps and test cases.

**Listing 4** Specification of the computation of navigation function.

```
method ComputeNavFn(start: Pose, goal: Pose,
                    costMap: CostMap, numIterations: nat)
        returns (navfn: PotentialMap)
  // Preconditions:
  requires ValidCostMap(costMap)
      // Cost map has non−negative costs
  requires 0 ≤ start.pos.row < costMap.numRows
        ∧ 0 ≤ start.pos.col < costMap.numCols
  requires 0 ≤ goal.pos.row < costMap.numRows
        ∧ 0 ≤ goal.pos.col < costMap.numCols
      // Start and goal poses lie within the cost map
  requires Open(costMap, goal.pos.row, goal.pos.col)
      // Goal position is free of obstacles

  // Postconditions:
  ensures PositionSafe(navfn, costMap)
      // Position safety (see Listing 5)
  ensures Progress(navfn, goal, costMap.numRows, costMap.numCols)
      // Progress (see Listing 5)
  ensures navfn[goal.pos.row][goal.pos.col] ≠ Infinity
      // Goal position gets finite potential
```

**Listing 5** Property definitions in Dafny: position safety and progress.

```
// PositionSafe(pm, cm) holds iff pm and cm have the same size
//   and every position with finite potential in pm does not contain
//   any obstacles.

predicate PositionSafe(pm: PotentialMap, cm: CostMap)
{
  PotentialMapHasDimensions(pm, cm.numRows, cm.numCols)
  ∧ ∀ i, j | 0 ≤ i < cm.numRows ∧ 0 ≤ j < cm.numCols
      :: pm[i][ j ] ≠ Infinity ⟹ Open(cm, i, j)
}


// Progress(pm, goal, numRows, numCols) holds iff for every position (i, j)
//   different from goal, with finite potential in pm there is another
//   position (i', j') adjacent to (i, j) which lower potential.

predicate Progress(pot: PotentialMap, goal: Pose, rows: nat, cols: nat)
{
  PotentialMapHasDimensions(pot, rows, cols)
  ∧ ∀ i, j | 0 ≤ i < rows ∧ 0 ≤ j < cols ::
        i ≠ goal.pos.row ∧ j ≠ goal.pos.col ∧ pot[i][j] ≠ Infinity ⟹
          ∃ i', j' :: Adjacent(Pose(Point(i, j)), Pose(Point(i', j')))
               ∧ 0 ≤ i' < rows ∧ 0 ≤ j' < cols
               ∧ pot[i'][ j'] ≠ Infinity
               ∧ GreaterThan(pot[i][j], pot[ i '][ j '])
}
```

The FinitePotentialPath predicate (shown in Listing 6) relates both paths. This predicate asserts that each position in the computed path belongs to a cell with finite potential. Therefore, the position safety property allows us to conclude that the computed path does not run into obstacles.

## 6. Verification of NavFn using Maude

In this section we present the specification of NavFn in Maude, describe how we have applied the test suite for differential testing, and which properties of NavFn we have been able to prove with the Maude specification.

### 6.1. Maude specification

Maude is a general-purpose specification language that supports a flexible syntax and an ordered type system for describing the state and data structures of the model, equations and structural axioms (like associativity and commutativity) for specifying semantic equivalence between terms and programming auxiliary operations like in any functional language, and rules for defining the concurrent and nondeterministic transformation of the model. Hence, we have specified the data

**Listing 6** Definition of a path having finite potential.

```
predicate FinitePotentialPath(path: Path, closest: seq<Point>,
                  potentialMap: PotentialMap, numRows: nat, numCols: nat)
  requires PotentialMapHasDimensions(potentialMap, numRows, numCols)
{
  |path| == |closest|
    // Both lists have the same length
  ∧ (∀ i | 0 ≤ i < |path| :: BelongsToCell(path[i], closest[ i ])
    // Every position path[i] is located in its counterpart closest[i]
  ∧ (∀ i | 0 ≤ i < |closest| ::
        0 ≤ closest[i].row < numRows
        ∧ 0 ≤ closest[i].col < numCols
        ∧ potentialMap[closest[i].row][closest[ i ]. col] ≠ Infinity)
    // Every position in closest contains cells with finite potential
}
```

structures used by ROS (i.e., cost maps, poses, paths) with a high-level mathematical syntax. Moreover, because these data structures have been defined in independent modules, they can be used to implement other algorithms like Dijkstra's.

Equations have been used to specify auxiliary operations like get for obtaining the cost at a given position, and open? to check whether this position does not contain an obstacle:

```
op get : CostMap Nat Nat Nat → Float [special(id−hook SpecialHubSymbol)] .
ceq get(CM, X, Y, NC) = float(N')
 if N := (NC ∗ Y) + X ∧
    N' := skipN&Get(CM, N) .

op open? : CostMap Nat Nat Nat → Bool .
eq open?(CM, X, Y, NC) = get(CM, X, Y, NC) < 254.0 .
```

Notice that the representation of CostMap is a list of floating-point numbers and that getting the value at a given position implies walking linearly through N = (NC * Y) + X elements, as skipN&Get does, with a quadratic cost in the worst case. For obtaining a linear cost in the worst case, we could have packaged the matrix by rows, but instead we have used the Python bindings for Maude and its ability to define custom special operators to obtain random access to the map. The maude Python package is a general-purpose language binding for the specification language that represents Maude entities like terms, modules, and sorts as Python objects, and allows operating on them. This package communicates directly with the C++ implementation of Maude, whose objects and methods are called behind the scenes, allowing in particular the definition of new primitive operators implemented in Python. In this case, the special (id-hook SpecialHubSymbol) tells that the reduction of get must be handled by custom Python code, which will access the matrix in constant time when Maude is run under its Python library. Otherwise, when running Maude through the standard interpreter or when no handler is installed, the equation above will be used. On the contrary, the data structures for the potential and the gradient are represented as lists of lists of numbers, one list per row, to allow for linear-time access without external intervention.

The main functions of the algorithm are a*, getPotential, and computePath

```
***       init  goal          cols rows  iter bounds
op a∗ : Pose Pose CostMap Nat  Nat  Nat  Nat ~> Path .
*** idem
op getPotential : Pose Pose CostMap Nat Nat Nat → Potential .
***                          init  goal  step       cols rows max−iter
op computePath : Potential Pose Pose Float Gradient Nat  Nat  Nat ~> Path .
```

that respectively represent the whole NavFn algorithm, the calculation of the potential, and the calculation of the final path given a potential map. The a* function receives the initial and goal positions (two pairs of integers), the cost map (a list of real numbers, as explained before), the dimensions of the map, and the limits on the number of iterations for both the calculation of the navigation function and the path. In turn, the computePath function also receives the potential calculated in the previous step, the length of a path step, and an initially-empty gradient matrix. Both a* and computePath* are partial functions that return a path (a list of Poses) or an error term if, for example, the iteration limits have been exhausted. Looking at the original implementation of NavFn, getPotential and computePath have been specified to reproduce the steps in its calcNavFnAstar and calcPath methods, respectively. However, while getPotential is specified using equations, computePath is defined with rewrite rules. While using rewrite rules for a deterministic computation may look overkill, this enables doing model checking and white-box testing on the first phase of the algorithm, since it exposes the intermediate states to the Maude tools and commands. Moreover, rewriting with rules instead of equations does not noticeably affect performance for this particular specification, as observed empirically. This can be explained by the shape of our rules, which do not include rewriting conditions, are always applied on top, and delegate most of the work to equations.

As a final remark, we have enabled the integration of the Maude executable specification into the ROS Navigation 2 package. We have developed a framework where any call to the ComputePathToPose is delegated to the Maude planner through Python.[10] This works not only with the Maude clone of the NavFn implementation but also with any other planner programmed in Maude. Therefore, any robot using ROS could benefit of these Maude planners. As an example, a video showing a simulation using a textbook A* planner [27] is available at [59]. The Maude code of these implementations is available in the folder src/maude at [40].

## 6.2. Differential testing of the Maude specification

Once we have a Maude specification of the NavFn algorithm, the next step is applying differential testing with the test suite in Section 4 against the original implementation to check whether they behave the same. For every map and pair of positions (origin, destination), we have concluded that

- the paths obtained by Maude and ROS are the same,
- the auxiliary navigation function computed by Maude and ROS coincides, and
- the state of the A* search coincides iteration by iteration between the two implementations.

However, this comparison must take into account that ROS and Maude manipulate floating-point numbers with different precision, respectively the single (4 bytes) and the double (8 bytes) representation of the IEEE-754 standard, known as the float and double types in C++. Real numbers are used both to represent positions, values of the potential array, and intermediate results, so the outputs of the NavFn algorithm and the Maude specification can be slightly different because of the uneven precisions of the underlying types. Therefore, to consider that two positions are the same we have allowed a difference $\varepsilon$ in each axis depending on the map size, of at most $\varepsilon \leq 0.05$. This value has been selected empirically by inspecting the actual differences in the obtained paths. Another relevant aspect when comparing paths are oscillations (see Section 2.2), which are detected when the $i^{\text{th}}$ position of a path is the same as the $(i-2)^{\text{th}}$ position. The ROS implementation uses strict equality on the float type to compare these two positions, so it is very unlikely that the Maude version, which uses double numbers, would detect the oscillations at the same point of the path. Indeed, the Maude implementation usually requires more iterations to detect and interrupt oscillations, even if positions are not compared using strict equality but with a configurable tolerance level (which cannot be adjusted to always match the ROS behavior). Hence, we have compared the obtained paths ignoring any possible oscillation that may appear.

Regarding the intermediate result, the navigation function or potential map, we have checked that the potential maps produced by Maude coincide with those of the original implementation with a small numerical error. More precisely, the maximum difference between two cells is 1.0859 in the whole test suite and $1.3947 \cdot 10^{-2}$ in mean. The euclidean distance between every pair of functions, normalized by the map area, is $2.2567 \cdot 10^{-3}$ at most and $1.5799 \cdot 10^{-5}$ in mean.

Using the test suite, we have also checked the equivalence of the implementation in a white-box manner. Our white-box analysis has focused on the first phase of the algorithm, namely the calcPotential function shown in Algorithm 1. As explained in Section 2.2, this phase calculates a potential map using a custom A* search that expands the state space by taking positions from a collection of FIFO queues. We have compared the number of search iterations in both the NavFn planner and the Maude specification and the contents of these queues in every one of them (i.e., at line 18 of Algorithm 1). This has been achieved by running the NavFn code with the Python API of the GNU debugger and by advancing the Maude specification in sync with it, so that the execution states of both implementations can be obtained and compared on the fly. A debug build of the NavFn planner is available at the releases section of [40] to facilitate the reproduction of the results.

In order to automate the black-box tests, we have developed a Python script that invokes the ROS Navigation library and the Maude algorithm. The connection between Python and Maude is provided by the maude package [57].

In summary, the result of differential testing is that ROS and Maude obtain the same sequences of cells for all the 121,409 paths tested, considering the notion of path equivalence mentioned before. Moreover, both implementations execute the same number of iterations to calculate the potential, and the contents of their search queues coincide all along them, as follows from the white-box application of the same tests.

Regarding the efficiency of the Maude implementation, Table 3 shows the original C++ code is orders of magnitude faster. Moreover, Dafny+Go is also 16.84 times faster than Maude. However, Maude runs the whole test suite in less than 90 minutes (12 minutes if the biggest ROS maps are excluded), which is a reasonable delay ahead for this verification approach. There are several reasons why Maude can be slower than the original C++ and the Dafny implementations. The general ones are Maude being an interpreted language instead of a compiled one, pattern matching versus direct access to variables, the fact that Maude has to manage a dynamic collection of trees and DAG nodes in the heap for representing the terms while C++ simply deals with variables in the stack or already allocated arrays in the heap, etc. Indeed, we have profiled the execution of the Maude implementation under several test cases, and the most time-consuming operations are those related with the allocation and deallocation of terms, including garbage collection and matching. A more specific reason is

[10] The official programming languages of ROS are C++ and Python. We have chosen Python for its simplicity, since the computationally-expensive work of our planner is executed in Maude, and the performance gain of using C++ for the connection would be negligible.

that potential and gradient maps are accessed in constant time in ROS and Dafny but in linear time in Maude, due to their representation as lists. A more sophisticated version of the trick relying on Python that provides constant-time access to the map could also be applied to these other data structures for improving performance.

### 6.3. Verification of properties

As explained in Section 3.3, Maude provides an LTL model checker that lets us verify properties for particular maps and positions. This lets us prove on every case of the test suite that (1) the queues that keep the state of the search algorithm never include an obstacle (called *obstacle-free queues*), in LTL $\square\,(\neg\,\mathsf{wallInCurrent} \wedge \neg\,\mathsf{wallInNext} \wedge \neg\,\mathsf{wallInExcess})$; and that (2) no path is obtained until (and unless) a non-infinite potential is calculated for its origin (*no path without finite potential*), in LTL $(\neg\,\mathsf{PathComputed})\,\mathbf{W}\,\mathsf{NonInfinitePotAtInit}$. As usual in Maude, the atomic propositions are defined as predicates on the state terms

```
ops wallInCurrent wallInNext wallInExcess : → Prop [ctor] .
ops PathComputed NonInfinitePotAtInit : → Prop [ctor] .
```

specifed by equations like the following

```
eq a∗({X, Y}, GOAL, G, NC, NR, I, P) |= NonInfinitePotAtInit
= P @ [X, Y] <  infinite   .
```

Although somehow limited, this analysis sets the foundations for applying model checking into the ROS ecosystem; other temporal properties in this abstract model can be verified in the same way as we did with the properties above. Furthermore, notice that, although model checking allows us to prove properties for fixed states, in combination with our map generation framework, we are automatically checking them on every case of the differential testing suite, reaching a high level of confidence in the properties being analyzed.

## 7. Related work

We are not aware of any formal approach specifically aimed to analyze the ROS Navigation Stack. There are formal verification efforts focusing on verifying the ROS communication layer [63], where model checking is applied to the ROS scheduler to verify properties like deadlock-freeness, schedulability, and responsiveness of tasks. In [25] the authors propose a general methodology for extracting key parameters from the source code of ROS nodes, which are used to build a timed automata. Then, by model checking this timed automata, they can prove several real-time properties. However, this methodology is not automated but it must be done manually, which is error-prone, and focus mainly on the communication of the nodes, therefore being conceptually far from the domain-specific NavFn features we want to prove in this paper. There are also proposals to apply runtime verification in ROS, for example the systems ROSMonitoring [23] and ROSRV [30]. These tools automatically generate monitors from the temporal properties to check, which integrate in the ROS node hierarchy and inspect the messages sent and received at run time to detect when the properties fail. The same runtime verification approach through monitors has been extended to ROS-Based Robot Swarms [29]. These approaches are fundamentally different from ours, as we aim to verify statically some properties of the NavFn planner that must be fulfilled in every possible execution.

The differential testing technique we apply to check the specification w.r.t. the system has been widely used to validate C compilers [41,65,35]. As every C compiler must comply with the same standard, it is possible to generate several random input C programs with increasing quality levels (random strings, syntactically valid C programs, type-correct C programs, etc.) and compare the results of different compilers. This comparison can consider the messages produced while compiling and also the output produced by executing the compiled program. Compared to the generation of randomized maps and paths following certain patterns, as we do for the NavFn case, generating C programs is a more complex problem that requires stochastic grammars and even apply mutations on valid C programs previously selected. Another distinction with the canonical use of differential testing is that in our verification methodology any difference between the specification and the complex system is identified as a defect in the specification, as it has to mimic the system observed behavior. However, a difference between C compilers can imply a bug in some of them, in all of them or even in none of them (if tests containing undefined behavior, according to the C language standard, have not been previously discarded).

Differential testing has been also applied to detect defects in implementations of the Java Virtual Machine (JVM) [13]. In that work, they start from an initial set of seed class files to compare the behavior of different JVM implementations and they enlarge this set by randomly transforming these class files using different mutators (for example, adding local variables, adding or removing parameters, or inserting statements that throw exceptions), in a way that mixes *mutation testing* [31,49] and *fuzz testing* [33]. As this process of mutation can lead to a wide number of similar class files, they use a notion of *coverage uniqueness* to only accept a class file mutation if its statement and branch coverage over a reference JVM is significantly different from any previous considered class file. This mutation approach for enlarging the test suite could also be integrated in our generic verification methodology. However, it could be only applied for complex systems in which the source code is available, to be able to calculate a measure of coverage uniqueness of the generated tests. The NavFn planner would be a good candidate for it because the C++ source code is available, as well as robust tools for measuring different notions of code coverage. Apart from the coverage unique notion, the only additional requirement would be to define a set of mutators of our tests, that may include map rotation, symmetries, cell swap, among others.

Another area where differential testing have been successfully applied is in machine learning (ML) systems [66], i.e., systems that provide ML algorithms. As there are several implementations of the same ML algorithms in different systems, differential testing has been applied to detect defects in implementations of well-known ML algorithms like Naive Bayes and *k*-nearest neighbors [62], or even in libraries for deep learning [51] like TensorFlow or Theano. Moreover, these deep learning libraries contain a number of assertions based on differential testing to check if their computed values are close enough to the ones obtained by other similar systems [46], what are called *oracle approximation assertions*. Systems like DeepXplore [50] also use differential testing similarly as our verification methodology. It takes as input different Deep Neural Networks that solve the same problem and generates test cases that trigger different behaviors in each one. Moreover, DeepXplore tries to obtain test cases with high *neuron coverage*, i.e., tests that activated the majority of the neurons. This concept of coverage exemplifies that there are more natural coverage notions to measure the completeness of a test suite for a complex system, in combination with the code coverage proposed in Section 3.1.

Regarding the combination of ROS and external declarative languages, the single system similar to ours is roshask [16]. It is a Haskell library for ROS 1 that allows users to define nodes in Haskell that deal with the rest of the system transparently. The way we allow the integration into ROS of the Maude specification of the NavFn planner through a Python bridge, in fact, follows the same ideas and provides similar features. In our case, our approach is designed to work in the current version of ROS 2.

Finally, it is worth noticing other verification tools that have led to the detection of bugs in wide-spread real programs. One of the most successful stories is the detection of a bug in the Java implementation of the TimSort sorting algorithm in the OpenJDK [24], bug that also appeared in the original Python implementation from which it was ported. Using the system KeY [8], a tool to semi-automatically verify Java programs from a specification of its behavior defined in the Java Modeling Language that states preconditions, postconditions, and invariants, among others; the authors detected that under some circumstances the sorting algorithm incorrectly threw an ArrayIndexOutOfBoundsException exception. From this knowledge, they proposed a fixed version of the sorting algorithm that was verified using KeY and was integrated in the OpenJDK standard library. Notice that the use of KeY requires the availability of the program source code, which must have been written in Java (or C if KeY-C is used [44]), which is not the case of the C++ NavFn planner validated in this paper. Moreover, our verification methodology can still be applied even in complex systems compound by different interconnected modules in different programming languages, or even in systems where no source code is available but only some executable version used as a black-box pseudo-oracle for performing differential testing on the observed behavior.

## 8. Lessons learned

We discuss in this section the main issues that might appear when following the methodology described in this paper, according to our experience. On the one hand, verification is a difficult task and completely ensuring the soundness of a real system is often impossible in practice. When verifying real systems we need to take into account that they have been optimized to offer the best performance when applied in production, which poses a two-fold problem. First, obtaining the specification becomes harder because the same behavior might be hard-coded in different conditional branches to benefit from particular configurations, hence making the generalization process tougher for the specifier. In the same line, abstract concepts, which are usually loosely defined in most cases, need to be understood in order to represent them formally. In this work, an interesting example of such an abstract concept is that of a cell of the robot map, which is ubiquitous in the navigation algorithm but whose exact boundaries are not explicitly specified. In order to reach an appropriate formalization, in the absence of even an informal definition, clues should be obtained from all aspects of the system. A wrong understanding of the notion, as in our earlier attempts, can lead to conclusions that are meaningless in practice. Moreover, randomly generating the test cases might produce unfeasable configurations. In our case, this might happen when the values in adjacent cells are too different; in real scenarios this cannot happen because it would indicate that the floor changes suddenly, which is not expected in the context where these robots are used. Hence, errors found on this kind of configurations might not be relevant for developers. Nevertheless, using them on the differential testing phase can still be useful, because they show that the original system and the specification coincide even when pushed to the limit.

On the other hand, comparing the results obtained in the system and in the specification (even if this specification is correct) can be challenging. In our case, we discovered that low-level details may lead to high-level differences: as discussed in Section 6.2, the diverse precision used to represent real numbers in C++, Maude, and Dafny made traces different and required us to further refine the specification in order to represent more faithfully the behavior of the system. Hence, the refinement loop depicted in Fig. 1 is also vital at this stage, possibly not just for improving the specification but also for understanding how results must be compared.

## 9. Concluding remarks and future work

In this paper we have verified the soundness and some properties about the C++ implementation of the NavFn planner of ROS Navigation Stack using a general verification approach that can be applied to other complex systems. It consists in specifying the original implementation in an executable specification language with verification support, validating its equivalence with differential testing techniques using the original system as a pseudo-oracle, and then verifying properties on the specification, which is easier because of its higher abstraction level and the availability of formal deduction tools. Properties checked in the specification can be attributed with a high degree of confidence to the original system, backed by a extensive testing of equivalence between both implementations. This approach improves the manual specification and verification workflow by adding a differential testing phase that may detect errors in the specification and provides evidence to support its intended equivalence with respect to the original system. Regarding classical testing, we think that the development of a comprehensive human-written specification that matches the original behavior on a extensive test suite, where properties are usually proven using deductive methods, is a stronger guarantee than the one obtained by the direct assessment of these properties on the test cases. While the guarantees for the transferability of the properties to the original implementation are not absolute, we think that this is a reasonable procedure for handling complex systems where there are no other more direct verification tools available.

For the application of this procedure to the NavFn planner, we have developed two alternative specifications in the Dafny and Maude languages, and a extensive test suite with more than 100 maps and 120,000 test cases. Using this test suite, the two specifications have been differentially tested against the original planner using black-box and white-box testing techniques, finding that their behaviors and outputs are indistinguishable (except for small numerical precision errors in the case of Maude). We have then used Dafny to prove with Hoare logic that the paths generated by the NavFn algorithm do not collide with obstacles, as well as some other properties of the intermediate results. With the Maude specification, we have model checked several properties of the internal workings of the algorithm for every case of the test suite. Unlike the deductive methods used with Dafny, these checks are completely automated but they only work for specific inputs. Moreover, thanks to the test suite, we have incidentally found some counterexamples for the optimality –paths should have the lowest possible cost—of the NavFn planner and the smoothness of its paths.

In this process of verification of the ROS NavFn we have learned some useful insights. First, generating the executable specification and the complete test suite to corroborate its behavior is a non-trivial but worthwhile task, since any verification effort in Dafny or Maude is much simpler than in the source code. Second, from a practical point of view, executing a differential testing between a concrete system and an executable specification may be complex because of the different representation of basic datatypes like real numbers, which may require some low-level adaptations in the runtime environment or in the comparison process. Finally, having a parameterized unified methodology that can be instantiated with different executable specification languages for the same complex system is very helpful, as different properties can be addressed more easily in different tools and the test suite can be completely reused.

In the future, we are considering verifying some of the newer planners available in ROS, like the SmacPlanner [18], which has additional capabilities like support for Ackermann steering vehicles. The same methodology we used for NavFn could be applied here. For this analysis we could also use new notions of correctness and optimality that would have more practical applications than those considered in this paper. Specifically, a correctness notion that takes the robot volume and other physical constraints into account so that the formal absence of collisions actually translates to absence of robot crashes, and that allows determining which inflation parameters are required to warrant that the robot will never crash; and an optimality notion that allows us to express trade-offs between execution costs and path quality, e.g. path optimality under a limitation on the number of instructions that can be executed, that would act as a quota on the CPU cycles available.

We also plan to enhance Maude with support for specification in Hoare logic and the automated generation of verification conditions for an SMT solver, so that the semi-automated deductive verification available in Dafny can be also applied to Maude specifications. This will be facilitated by the reflective nature of Maude, its recent integration of SMT solvers, and its support for symbolic computation [19]. All these features are generic tools that could be useful for future unrelated verification tasks, but that have also been conceived working backwards from a concrete verification use case for a software component with industrial applications, which provides some evidence of their potential practical impact and usefulness. For instance, the Python interface to Maude [57], which was started for this work, has been also used to check CTL, CTL*, and $\mu$-calculus properties on standard and strategy-controlled Maude models through external tools [58], among other applications.

Finally, we would also like to explore metamorphic testing techniques [61] to automatically expand the number of tests in the suite used for differential testing. In the concrete case of the ROS NavFn planner we could apply metamorphic relations as map rotations, symmetries, and compose bigger maps from smaller ones; but similar techniques could also be applied to other complex systems. As generating a complete test suite with good coverage metrics is usually a nontrivial task that requires much work, considering metamorphic relations could possibly reduce the test generation effort and simplify the application of the proposed verification methodology.

## Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

## Data availability

Source code, test cases, and the results of experiments are publicly available on GitHub, and links are included in the manuscript.

## Acknowledgements

Work partially supported by the Spanish AEI project ProCode-UCM (PID2019-108528RB-C22), the Spanish MINECO project CAVI-ART-2 (TIN2017-86217-R), by Spanish MU grant FPU17/02319, and by Comunidad de Madrid as part of the program BLOQUES-CM (S2018/TCS-4339), co-funded by EIE Funds of the European Union.

## References

[1] E. Ackerman, Apex.AI does the invisible work that will make self-driving cars possible, IEEE Spectrum (2019), https://spectrum.ieee.org/cars-that-think/transportation/self-driving/apexai-does-the-invisible-critical-work-that-will-make-selfdriving-cars-possible.
[2] L. Aguirre, N. Martí-Oliet, M. Palomino, I. Pita, Conditional narrowing modulo SMT and axioms, in: W. Vanhoof, B. Pientka (Eds.), Proceedings of the 19th International Symposium on Principles and Practice of Declarative Programming, Namur, Belgium, October 09–11, 2017, ACM, 2017, pp. 17–28.
[3] F.E. Allen, Control flow analysis, ACM SIGPLAN Not. 5 (7) (1970) 1–19.
[4] K. Bae, J. Meseguer, Model checking linear temporal logic of rewriting formulas under localized fairness, Sci. Comput. Program. 99 (2015) 193–234.
[5] K. Bae, S. Escobar, J. Meseguer, Abstract logical model checking of infinite-state systems using narrowing, in: F. van Raamsdonk (Ed.), Proc. RTA 2013, Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2013, pp. 81–96.
[6] E.T. Barr, M. Harman, P. McMinn, M. Shahbaz, S. Yoo, The oracle problem in software testing: a survey, IEEE Trans. Softw. Eng. 41 (5) (2015) 507–525, https://doi.org/10.1109/TSE.2014.2372785.
[7] K. Beck, Test-Driven Development: By Example, Addison-Wesley Professional, 2003.
[8] B. Beckert, R. Hähnle, P.H. Schmitt (Eds.), Verification of Object-Oriented Software. The KeY Approach - Foreword by K. Rustan M. Leino, Lecture Notes in Computer Science, vol. 4334, Springer, 2007.
[9] Y. Bertot, P. Castéran, Interactive Theorem Proving and Program Development - Coq'Art: The Calculus of Inductive Constructions, Texts in Theoretical Computer Science. An EATCS Series, Springer, 2004.
[10] A. Bouhoula, J.P. Jouannaud, J. Meseguer, Specification and proof in membership equational logic, Theor. Comput. Sci. 236 (2000) 35–132, https://doi.org/10.1016/S0304-3975(99)00206-6.
[11] O. Brock, O. Khatib, High-speed navigation using the global dynamic window approach, in: 1999 IEEE ICRA, Detroit, Michigan, USA, May 10-15, 1999, Proceedings, IEEE Robotics and Automation Society, 1999, pp. 341–346.
[12] G. Carbone, F. Gomez-Bravo, Motion and Operation Planning of Robotic Systems: Background and Practical Approaches, Springer, 2015.
[13] Y. Chen, T. Su, C. Sun, Z. Su, J. Zhao, Coverage-directed differential testing of jvm implementations, in: Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '16, Association for Computing Machinery, New York, NY, USA, 2016, pp. 85–99.
[14] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, C. Talcott, All About Maude: A High-Performance Logical Framework, LNCS, vol. 4350, Springer, Berlin, Heidelberg, 2007.
[15] M. Clavel, M. Palomino, A. Riesco, Introducing the ITP tool: a tutorial, J. Univers. Comput. Sci. 12 (11) (2006) 1618–1650, https://doi.org/10.3217/jucs-012-11-1618.
[16] A. Cowley, C.J. Taylor, Stream-oriented robotics programming: the design of roshask, in: 2011 IEEE/RSJ International Conference on Intelligent Robots and Systems, IROS 2011, IEEE, 2011, pp. 1048–1054.
[17] M.D. Davis, E.J. Weyuker, Pseudo-oracles for non-testable programs, in: Proceedings of the ACM '81 Conference, ACM '81, Association for Computing Machinery, New York, NY, USA, 1981, pp. 254–257.
[18] D. Dolgov, S. Thrun, M. Montemerlo, J. Diebel, Practical search techniques in path planning for autonomous driving, Ann Arbor 1001 (48105) (2008) 18–80.
[19] F. Durán, S. Eker, S. Escobar, N. Martí-Oliet, J. Meseguer, R. Rubio, C.L. Talcott, Programming and symbolic computation in Maude, J. Log. Algebraic Methods Program. 110 (2020), https://doi.org/10.1016/j.jlamp.2019.100497.
[20] F. Durán, S. Eker, S. Escobar, N. Martí-Oliet, J. Meseguer, R. Rubio, C.L. Talcott, Equational unification and matching, and symbolic reachability analysis in Maude 3.2 (system description), in: J. Blanchette, L. Kovács, D. Pattinson (Eds.), Proc. IJCAR, in: LNCS, vol. 13385, Springer, 2022, pp. 529–540.
[21] S. Eker, J. Meseguer, A. Sridharanarayanan, The Maude LTL model checker, in: F. Gadducci, U. Montanari (Eds.), Proc. WRLA 2002, in: ENTCS, vol. 71, Elsevier, 2004, pp. 162–187.
[22] C. Ellison, G. Rosu, An executable formal semantics of C with applications, in: Proceedings of the 39th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '12, Association for Computing Machinery, New York, NY, USA, 2012, pp. 533–544.
[23] A. Ferrando, R.C. Cardoso, M. Fisher, D. Ancona, L. Franceschini, V. Mascardi, ROSMonitoring: a runtime verification framework for ROS, in: A. Mohammad, X. Dong, M. Russo (Eds.), Towards Autonomous Robotic Systems, Springer, 2020, pp. 387–399.
[24] S. de Gouw, F.S. de Boer, Fixing the sorting algorithm for Android, Java and Python, ERCIM News 2015 (102) (2015), http://ercim-news.ercim.eu/en102/r-i/fixing-the-sorting-algorithm-for-android-java-and-python.
[25] R. Halder, J. Proença, N. Macedo, A. Santos, Formal verification of ros-based robotic applications using timed-automata, in: 2017 IEEE/ACM 5th International FME Workshop on Formal Methods in Software Engineering (FormaliSE), 2017, pp. 44–50.
[26] M. Hansen, ROS2 Robot Dev Kit featuring Navigation2 overview, in: ROS-Industrial Conference 2019, Fraunhofer IPA, 2019, https://rosindustrial.org/events/2019/12/10/ros-industrial-conference-2019.
[27] P.E. Hart, N.J. Nilsson, B. Raphael, A formal basis for the heuristic determination of minimum cost paths, IEEE Trans. Syst. Sci. Cybern. 4 (2) (1968) 100–107, https://doi.org/10.1109/TSSC.1968.300136.
[28] H. Hemmati, How effective are code coverage criteria?, in: 2015 IEEE International Conference on Software Quality, Reliability and Security, 2015, pp. 151–156.
[29] C. Hu, W. Dong, Y. Yang, H. Shi, G. Zhou, Runtime verification on hierarchical properties of ros-based robot swarms, IEEE Trans. Reliab. 69 (2) (2020) 674–689, https://doi.org/10.1109/TR.2019.2923681.
[30] J. Huang, C. Erdogan, Y. Zhang, B. Moore, Q. Luo, A. Sundaresan, G. Rosu, Rosrv: runtime verification for robots, in: B. Bonakdarpour, S.A. Smolka (Eds.), Runtime Verification, Springer International Publishing, Cham, 2014, pp. 247–254.
[31] Y. Jia, M. Harman, An analysis and survey of the development of mutation testing, IEEE Trans. Softw. Eng. 37 (5) (2011) 649–678, https://doi.org/10.1109/TSE.2010.62.

[32] F. Kirchner, N. Kosmatov, V. Prevosto, J. Signoles, B. Yakobowski, Frama-c: a software analysis perspective, Form. Asp. Comput. 27 (3) (may 2015) 573–609, https://doi.org/10.1007/s00165-014-0326-7.
[33] G. Klees, A. Ruef, B. Cooper, S. Wei, M. Hicks, Evaluating fuzz testing, in: Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security, CCS '18, Association for Computing Machinery, New York, NY, USA, 2018, pp. 2123–2138.
[34] K. Korosec, Toyota taps Apex.AI for its autonomous vehicle operating system, https://techcrunch.com/2021/04/14/toyota-taps-apex-ai-for-its-autonomous-vehicle-operating-system, 2021.
[35] V. Le, M. Afshari, Z. Su, Compiler validation via equivalence modulo inputs, in: Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '14, Association for Computing Machinery, New York, NY, USA, 2014, pp. 216–226.
[36] K.R.M. Leino, Dafny: an automatic program verifier for functional correctness, in: 16th International Conference on Logic for Programming, Artificial Intelligence, and Reasoning, LPAR-16, in: LNCS, vol. 6355, Springer, 2010, pp. 348–370.
[37] S. Macenski, F. Martín, R. White, J.G. Clavero, The marathon 2: a navigation system, in: 2020 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS), IEEE, 2020, pp. 2718–2725.
[38] S. Macenski, F. Martín, R. White, J.G. Clavero, The Marathon 2: a navigation system, in: Proc. IROS 2020, IEEE, 2020, pp. 2718–2725.
[39] E. Martin-Martin, M. Montenegro, A. Riesco, J. Rodríguez-Hortalá, R. Rubio, Verification of ROS navigation using Maude, in: N.E. Martí Oliet (Ed.), PROLE 2021, SISTEDES, 2021, http://hdl.handle.net/11705/PROLE/2021/008.
[40] ROS_navfn_verification repository, https://github.com/demiourgoi/ROS_navfn_verification.
[41] W.M. McKeeman, Differential testing for software, Digit. Tech. J. 10 (1) (1998) 100–107.
[42] J. Meseguer, Conditional rewriting logic as a unified model of concurrency, Theor. Comput. Sci. 96 (1) (1992) 73–155, https://doi.org/10.1016/0304-3975(92)90182-F.
[43] L.M. de Moura, S. Kong, J. Avigad, The lean theorem prover (system description), in: A.P. Felty, A. Middeldorp (Eds.), Proc. CADE 2015, in: LNCS, vol. 9195, Springer, 2015, pp. 378–388.
[44] O. Mürk, D. Larsson, R. Hähnle, KeY-C: a tool for verification of C programs, in: F. Pfenning (Ed.), Automated Deduction – CADE-21, Springer Berlin Heidelberg, Berlin, Heidelberg, 2007, pp. 385–390.
[45] G.J. Myers, C. Sandler, T. Badgett, The Art of Software Testing, 3rd edition, John Wiley & Sons, 2011.
[46] M. Nejadgholi, J. Yang, A study of oracle approximations in testing deep learning libraries, in: 2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE), 2019, pp. 785–796.
[47] T. Nipkow, L.C. Paulson, M. Wenzel, Isabelle/HOL - A Proof Assistant for Higher-Order Logic, Lecture Notes in Computer Science, vol. 2283, Springer, 2002.
[48] M. Norrish, A formal semantics for C++, Tech. Rep., Technical report, NICTA, 2008.
[49] M. Papadakis, M. Kintis, J. Zhang, Y. Jia, Y.L. Traon, M. Harman, Chapter six - Mutation testing advances: an analysis and survey, in: Advances in Computers, vol. 112, Elsevier, 2019, pp. 275–378.
[50] K. Pei, Y. Cao, J. Yang, S. Jana, DeepXplore: automated whitebox testing of deep learning systems, in: Proceedings of the 26th Symposium on Operating Systems Principles, SOSP '17, Association for Computing Machinery, New York, NY, USA, 2017, pp. 1–18.
[51] H.V. Pham, T. Lutellier, W. Qi, L. Tan, Cradle: cross-backend validation to detect and localize bugs in deep learning libraries, in: 2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE), 2019, pp. 1027–1038.
[52] M. Quigley, B. Gerkey, W.D. Smart, Programming Robots with ROS, O'Reilly Media, 2015.
[53] T. Ramananandro, G. Dos Reis, X. Leroy, A mechanized semantics for C++ object construction and destruction, with applications to resource management, in: Proceedings of the 39th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '12, Association for Computing Machinery, New York, NY, USA, 2012, pp. 521–532.
[54] O. Robotics, The Robot Operating System (ROS), https://www.ros.org/, 2020.
[55] ROS2 navigation framework and system, https://github.com/ros-planning/navigation2.
[56] F. Rubio, F. Valero, C. Llopis-Albert, A review of mobile robots: concepts, methods, theoretical framework, and applications, Int. J. Adv. Robot. Syst. 16 (2) (2019), https://doi.org/10.1177/1729881419839596.
[57] R. Rubio, Maude as a library: an efficient all-purpose programming interface, in: K. Bae (Ed.), Proc. WRLA 2022, in: LNCS, vol. 13252, Springer, 2022.
[58] R. Rubio, N. Martí-Oliet, I. Pita, A. Verdejo, Strategies, model checking and branching-time properties in Maude, J. Log. Algebraic Methods Program. 123 (2021) 100700, https://doi.org/10.1016/j.jlamp.2021.100700.
[59] R. Rubio, Integration of a Maude-based $A^*$ planner into ROS, https://youtu.be/LaJMtna-6s8, Jul 2020.
[60] R. Rubio, A. Riesco, Theorem proving for Maude specifications using Lean, in: M. Zhang, A. Riesco (Eds.), ICFEM 2022, in: Lecture Notes in Computer Science, vol. 13478, Springer, 2022.
[61] S. Segura, G. Fraser, A.B. Sánchez, A.R. Cortés, A survey on metamorphic testing, IEEE Trans. Softw. Eng. 42 (9) (2016) 805–824, https://doi.org/10.1109/TSE.2016.2532875.
[62] S. Srisakaokul, Z. Wu, A. Astorga, O. Alebiosu, T. Xie, Multiple-implementation testing of supervised learning software, in: The Workshops of the Thirty-Second AAAI Conference on Artificial Intelligence, New Orleans, Louisiana, USA, February 2–7, 2018, in: AAAI Technical Report, vol. WS-18, AAAI Press, 2018, pp. 384–391, https://aaai.org/ocs/index.php/WS/AAAIW18/paper/view/17301.
[63] W. Wei, X. Li, Y. Guan, R. Wang, Q. Lu, J. Zhang, Model checking for the goal-feedback-result pattern in ROS, in: Proc. ISPA/BDCloud/SocialCom/SustainCom 2019, IEEE, 2019, pp. 640–645.
[64] E.J. Weyuker, On testing non-testable programs, Comput. J. 25 (4) (11 1982) 465–470, https://doi.org/10.1093/comjnl/25.4.465.
[65] X. Yang, Y. Chen, E. Eide, J. Regehr, Finding and understanding bugs in C compilers, in: ACM SIGPLAN PLDI 2011, PLDI '11, ACM, New York, NY, USA, 2011, pp. 283–294.
[66] J.M. Zhang, M. Harman, L. Ma, Y. Liu, Machine learning testing: survey, landscapes and horizons, IEEE Trans. Softw. Eng. 48 (2) (2022) 1–36, https://doi.org/10.1109/TSE.2019.2962027.