Verification and testing of mobile robot navigation algorithms: A case study in SPARK

Piotr Trojanek and Kerstin Eder

Abstract— **Navigation algorithms are fundamental for mobile robots. While the correctness of the algorithms is important, it is equally important that they do not fail because of bugs in their implementation. Yet, even widely-used robot navigation code lacks proofs of correctness or credible coverage reports from testing. Robot software developers usually point towards the cost of manual verification or lack of automated tools that would handle their code. We demonstrate that the choice of programming language is essential both for finding bugs in the code and for proving their absence. Our re-implementation of three robot navigation algorithms in SPARK revealed bugs that for years have not been detected in their original code in C/C++. For one of the implementations we demonstrate that it is free from run-time errors. Our code and results are available online to encourage uptake by the robot software developers community.**

I. INTRODUCTION

In this paper we focus on the verification of code that implements robot navigation algorithms. These algorithms guide the robot towards a goal while avoiding obstacles within the range of on-board sensors. Comparing to path planning algorithms, they seem more important for the safety of the robot mission (if they fail then robot can hit the wall), more difficult to debug (if robot goes wrong it can be result of a problem that occurred few iterations of the algorithm before) and more challenging to prove safe (they operate on data from real sensors and not on idealized models of the environment).

Navigation algorithms are typically embedded within robot control software as modules with well-defined boundaries and implemented as drivers, components or nodes (depending upon the vocabulary preferred by authors of a particular robot software framework). Traditionally, they are considered as low-level control algorithms that—mostly for performance, but also for portability and reusability—should be implemented in C or C++. These languages, and C in particular, are well-known for being inherently unsafe. They enable the programmer to write code whose behaviour cannot be explained in terms of the language definition.

In our study we are interested in proving that none of the following occurs when executing robot navigation code:

- array out-of-bounds accesses,
- run-time exceptions, such as those related to the use of data containers,
- integer and floating-point arithmetic errors, e.g. divisions by zero, overflows and underflows,
- calls to mathematical functions, e.g. square root, with arguments out of their domains,
- null pointer dereferences,
- dynamic memory allocations, which may affect real-time behaviour of the robot,
- calls to potentially blocking operations, e.g. locking a mutex,
- accesses to uninitialized data and
- any operation with results not specified by the programming language definition.

Testing of robot control software, which is the most common approach to gain confidence in its reliability, is difficult to be done exhaustively. Robot navigation algorithms should be tested with all possible values of sensor data, goal poses, internal state variables and parameters, such as robot dimensions. Even with a simulator and an automated testing procedure some common bugs in C/C++, such as out-of-bounds array accesses, may remain undetected.

Instead of verifying existing C/C++ robot software we translated three open-source implementations of well-known robot navigation algorithms to SPARK—a programming language designed to develop high-reliability software. The goal of our case study is to demonstrate that the run-time safety of robot navigation algorithms implemented in SPARK is easier to test and prove. In Section II we introduce the reader to the origins of SPARK, compare it with C/C++ and demonstrate its suitability for verification of robot control software. In Section III we present bugs that we found after translating the original C/C++ code to SPARK and explain how we proved the absence of bugs in our code. In Section IV we discuss our approach in terms of effort and the performance of the resulting, verified code. We justify our choices and compare our results with related work in Section V. We conclude and outline possible extensions of this work in Section VI.

II. SPARK AND ROBOT NAVIGATION CODE

SPARK is an imperative, strongly-typed language designed for the development of reliable software. Technically, it is a subset of Ada—a general purpose programming language originally designed for use in military applications [1]. Ada offers many features that make it the perfect choice for embedded and real-time software. SPARK restricts the use of those constructs of Ada that make the code difficult to verify. While Ada is often compared to C++, SPARK is typically compared to MISRA-C—a set of guidelines for the use of C in critical systems.[1] SPARK was designed with soft-

This work was supported by the EPSRC grant EP/J01205X/1 RIVERAS: Robust Integrated Verification of Autonomous Systems. Both authors are with the Department of Computer Science, University of Bristol, UK. Email: {Piotr.Trojanek,Kerstin.Eder}@bris.ac.uk

[1]See `http://www.spark-2014.org/` for the detailed comparison of the latest versions of MISRA-C and SPARK.

ware reliability in mind; in contrast, MISRA-C attempts to eliminate those constructs of C that may result in unreliable code. Finally, the SPARK specification and its supporting tools are freely available (at least for the development of free and academic software). The MISRA-C specification must be purchased from the MISRA organization and freely available tools for the verification of MISRA-C code are less mature than those for SPARK.

The latest release of SPARK, 2014, and its supporting tools offer a unique approach to verification by combining static analysis and testing [2]. The GNATprove tool[2] checks conformance of the Ada source code with SPARK restrictions and generates a *verification condition* for each operation that may potentially result in a run-time error, an exception or a violation of an explicit assertion. Verification conditions are expressed in the language of the Why3 deductive verification platform [3]. They are discharged using one of several automatic provers and interactive proof assistants supported by Why3. The Ada compiler can automatically inject tests for both explicit assertions and run-time errors (such as array out-of-bounds accesses) into executable code.

In the following subsections we discuss which features of three open-source implementations of robot navigation algorithms, all implemented in C/C++, make them difficult to verify. We explain how we re-implemented and verified the same algorithms in SPARK. We analyzed code for VFH+ (Vector Field Histogram) [4], ND (Nearness Diagram) navigation [5] and SND (Smooth Nearness-Diagram) navigation [6] algorithms. This code is a part of the Player/Stage environment for mobile robot programming [7], which, before ROS [8], was a *de-facto* standard software platform in the mobile robotics research community. Our selection of code is not meant to be exhaustive and we intentionally do not focus on recently published software; rather, we are interested in exposing weaknesses in stable code that is supposed to be well-tested.

A. Annotations and pointers

A modular approach to software verification relies on explicit contracts, such as pre- and postconditions, that describe external interfaces of subprograms. For C code these contracts are typically expressed using annotations in ACSL (ANSI/ISO C Specification Language) [9]; no such annotations language exists for C++.

In practice, even simple C code requires relatively complex annotations. For example, a function from the VFH+ code that constrains a numerical value x to the interval $\langle min; max \rangle$ requires a number annotations to be placed in comments above the function's declaration:[3]

```
/* @requires min <= max
   @ensures  min <= *x && *x <= max
   @requires \valid(x) &&
             \unrelated(x, &min) &&
             \unrelated(x, &max)
   @modifies *x
*/
void ApplyLimits(float *x, float min, float max);
```

The essential information about this function, i.e. the relationship between the values of its arguments and result, is captured by the first two annotations; the other two annotations may seem superfluous, but are necessary. They specify that: the `x` pointer must not be NULL, must not alias other parameters, i.e. `min` and `max`[4], and that the value pointed to by `x` will be modified.

In SPARK, pointers (or access types in the Ada terminology) are forbidden and functions must be side-effect free, i.e. must not update their parameters nor global variables. SPARK procedures (which are equivalent to C/C++ void functions) are allowed to modify their parameters, but data that they modify must not be aliased. This greatly reduces the number of explicit annotations required to specify the externally visible behaviour of subprograms—only the essential information needs to be expressed explicitly. For example, in SPARK the same subprogram is declared as shown below.

```
procedure ApplyLimits(x : in out Float;
                      min, max : in Float)
with
  Pre  => min <= max,
  Post => min <= x and then x <= max;
```

B. Arrays and vector containers

Arrays in the robot navigation code primarily store fixed-size input data, e.g. sequences of sensor readings; vectors, on the other hand, primarily store intermediate results, e.g. sequences of obstacle-free regions around the robot. Accesses to array elements are cumbersome to verify in C, where array variables may lack information about their size. Another problem occurs if pointer arithmetic is used to access elements of C arrays (this was the case for the VFH+ code that we analyzed). Vectors, which are part of the C++ standard library, must be verified for accesses to non-existing elements and for not increasing their capacity at run-time, which may involve memory allocation. In practice, verification of array accesses in C requires even more verbose and complicated annotations than those already presented. For C++ vectors we found no tool that statically verifies that no memory allocation will occur at run-time.

In SPARK, arrays do carry information about their dimensions and so verification of array accesses requires no additional annotations. For sequences of data that vary in size we use *formal containers*—a SPARK library of generic data structures similar to the Standard Template Library (STL) of C++, but adapted to critical software development [10]. This library has been fully formalized, which makes it possible to be used in the verification process.

[2]GNAT is a free-software Ada compiler and names of several tools related to Ada are prefixed with GNAT.

[3]For the reader's convenience we translated the function name from Spanish.

[4]Similar information can be expressed using *restricted references*—a non-standard C++ extension, which allows compilers to generate more efficient executable code.

C. Numbers

Arithmetic errors, e.g. overflows, seem to be rare in robot navigation code, but are still unsafe should they occur. C and C++ offer very little protection against them. The typical solution involves the use of external libraries for integer arithmetic and explicit checks of results of floating-point operations [11]. In SPARK, arithmetic operations can be verified both statically and dynamically—by run-time checks automatically injected by the Ada compiler.

In practice, verification requires numeric variables to be annotated with ranges that constrains their values, e.g. to non-negative or positive numbers. In SPARK, numeric ranges can be easily defined using *subtypes*. For integers, we used only the standard-defined subtypes, i.e. `Positive` and `Natural`, which range respectively from 0 and 1 to the upper bound of the underlying `Integer` type. For floating-point data, we created our own subtypes. For example, non-negative numbers are represented by the following subtype, which ranges from 0.0 to the upper bound of the underlying `Float` data type.

```
subtype Non_Negative is Float
        range 0.0 .. Float'Last;
```

D. Mathematical functions

Robot navigation algorithms are typically based on geometric relationships between data, e.g. distances between points or angles between vectors. In C/C++, these relationships are calculated using functions from the standard mathematical library, e.g. `sqrt()` or `atan2()`. Most programmers do not routinely test for domain errors that may occur when calling such functions.

In SPARK, we use mathematical functions from the Ada standard library, which raise exceptions when domain errors occur. For static analysis, we created a wrapper package (an equivalent of namespace in C++) with mathematical functions annotated with preconditions that specify their domain, as defined in the Ada reference manual. To verify assertions about the code that rely on the properties of mathematical functions, we map each such function to its counterpart in the real-number theory. We use *external axiomatization*, which is a technique originally developed for the formal containers library [10], and verify the code based on mathematical axioms already known to theorem provers.

E. Threading and dynamic memory allocation

In robotics, algorithmic C/C++ code is often interweaved with calls to a threading library. This makes proving and debugging of such code much more difficult. SPARK simply forbids the use of concurrency and synchronization features of Ada; they are only allowed outside the clearly defined boundaries of purely algorithmic code.

Similarly, SPARK forbids the use of dynamic memory allocation, thus guarantees that no garbage collection, no memory leaks and no memory allocation errors will occur at run-time. Recursive calls, which are permitted in SPARK but may result in stack-overflow errors, are detected using GNATcheck—a tool for checking Ada coding standards.

TABLE I
CODE STATISTICS FOR C/C++ AND ADA IMPLEMENTATIONS (IN SLOC)

	Driver	**Algorithm**	
	C++	C/C++	Ada
VFH+	807	782	918
ND	828	1037	1426
SND	403	941	1183
Total	2038	2760	3527

F. Uninitialized data and undefined behaviours

Uninitialized variables in code may result in unpredictable robot operation and should be handled with the same care as other errors. Some C/C++ compilers, notably GCC, may warn about uninitialized data, but those checks are disabled in default compilation settings and limited in scope to local variables. In contrast, SPARK tools perform data-flow verification by default, including accesses to uninitialized global variables and individual components of record data.

Finally, many constructs in C/C++ code result in *undefined* or *implementation-defined behaviour*, i.e. their results depend on a compiler and not on the language definition. SPARK excludes all ambiguous constructs of Ada. This ensures that the program results are independent of the compiler or optimization settings. (Yet, some settings, such as precision of floating-point arithmetic, are platform-specific in SPARK.)

III. VERIFICATION

We have analyzed three open-source implementations of robot navigation algorithms. The algorithms process range data from an on-board sensor (typically a laser range-finder or an array of sonars) and produce translational and rotational velocity commands for the robot. Each implementation consists of several hundreds of source lines of code (SLOC) in C/C++. They are split between "driver" parts, which integrate the algorithms with the robot control framework, and "algorithm" parts, which process range data into velocity commands (Table I).

We manually translated the "algorithm" code to the SPARK subset of Ada and integrated it with the original "driver" code in C++. In many aspects C/C++ is very similar to Ada and the translation process for a large part of the code was straightforward. However, attention was required in those places where the languages differ; for example, there is no `continue` statement in Ada. On average our SPARK code is about 30% longer than the original C++ code (Table I). This is mainly because Ada has more verbose syntax, as it was designed to be more readable than other languages. Interfacing of code in Ada and C/C++ is part of the Ada standard and is well-supported by Ada compilers. In our setup, the Ada code is compiled into libraries and simply linked with the original "driver" code in C++.

We translated all pointers in C/C++ subprogram declarations to Ada parameter modes: const pointers to `in`, and other pointers to `out` or `in out` (depending on whether the pointed location was only written to or also read). The

ND code uses pointers and pointer arithmetic to access array elements; our implementation relies solely on array indices for this purpose. During the translation it became evident which parts of the original C/C++ code involve dynamic memory allocations at run-time. For example, the SND code allocates memory to store optional, intermediate results: it calls the `new` operator if the algorithm detects a safe passage towards the goal location. We translated this code using *discriminated records*—a native Ada data structure that stores values of one of several, possibly empty, fixed data types, and does not involve dynamic memory allocation.[5]

Vector data types in SPARK must explicitly specify their maximal capacity. In all of the analyzed code, vectors are accessed from within loops, so the maximal numbers of their elements can be determined from the ranges of the loop variables. In the VFH+ code some of the vectors are fixed in size, but they contain other vectors as elements. We translated these to multi-dimensional arrays, which are a native data structure in Ada.

The original SND code implements the "algorithm" part as a separate thread that communicates with the "driver" part using mutexes and condition variables. In our translation we refactored this code and entirely removed the need for inter-thread synchronization.

A. Run-time verification

After translation, the Ada code was compiled with all run-time checks enabled and validated using standard navigation scenarios from the Stage simulation environment. This already resulted in few run-time errors, most of which were related to calling mathematical functions with arguments out of their domains. Interestingly, in the VFH+ implementation these errors do not propagate to the final velocity commands, but—after a sequence of comparisons—affect only the internal operation of the algorithm. In the SND implementation, the relative distance to goal location is calculated using polar coordinates and ignoring the angular component of the result. It becomes a `NaN`, or *not a number*, each time when the robot reaches the goal (the origin of the coordinate system has no unique representation in polar coordinates). This also did not affect the final results, but such code, while "safe" in a sense, is quite difficult to understand. The meaning of program variables can be explained only by taking into account the details of floating-point arithmetic.

In the VFH+ code the run-time checks injected by the Ada compiler revealed also that every initialization of the algorithm involves accessing an array element at index `-1`. In C/C++ this error does not result in a run-time error, so probably the accessed memory contains some other data used by the same process.

B. Static analysis

To prove that no run-time error will ever occur when using robot navigation code we applied the static analysis tool GNATprove, which relies on deductive reasoning and the weakest-preconditions calculus rather than on testing [12]. In this approach, the information required for proofs needs to be propagated across the subprogram boundaries. For example, if a function uses its argument as a divisor then it needs to specify that this argument must be non-zero. Such information is typically expressed using explicit pre- and postconditions, but by using subtypes the code becomes more readable. In fact, we captured most of the required information using types that constrains the ranges of numeric variables.

Loop invariants, i.e. conditions that hold during loop executions, are well-known to be difficult to formulate when used to capture the state of the program variables at the loop exit. They turned out to be relatively easy to devise or even unnecessary when used only to prove the run-time safety of the loop bodies.

In several places we avoided explicit annotations (or substantially reduced their complexity) by refactoring the translated SPARK code. For example, the SND code in C++ searches for a pair of elements in a vector using two indices that are initially set to `-1`; clearly, either both of these indices must be `-1` (if no pair has been found yet) or both must point to valid vector elements. In SPARK we use a discriminated record that either is empty or contains a pair of valid vector indices. In general, it is possible to prove that such *refactorings for verification* (and the C++ to SPARK translation itself) preserve the semantics of the original code [13], but this was not the aim of our case study.

In other places extra annotations were inserted and the code was refactored to speed up the verification process. For example, code preceded by several (possibly nested) `if-then-else` statements can be time-consuming to prove, because all the possible paths that lead to its execution must be analyzed. We *cut* such code into sections using `Assert_and_Cut` annotations, which replace the exact knowledge about the state of program variables with expressions that carry only enough information to prove the subsequent code. Several parts of the code were refactored as nested procedures (available only as an extension to C/C++) with parameters of the mode `in` to indicate the data that are only accessed and not modified.[6]

SPARK tools are designed to prove only *partial correctness*, i.e. properties of a program's results at the program exit. The *total correctness*, which requires the program to also terminate, can be verified by prohibiting recursive calls, using `for` loops where possible (in Ada they require fixed bounds thus are guaranteed to terminate) and annotating the remaining `while` loops with *loop variants* (non-negative integer expressions whose values monotonically decrease with each loop iteration).

IV. RESULTS

We compared the performance of our implementations in SPARK with the original C/C++ code by averaging the time of

[5]In C++, a similar, but less flexible data structure is provided by the `boost::optional` library.

[6]Similar effect can be achieved using *ghost variables*, i.e. variables introduced only for the purpose of verification, but they are not yet supported by the SPARK tools.

TABLE II
AVERAGE TIMES OF SINGLE ITERATIONS OF THE ALGORITHMS IN µs AND RELATIVE PERFORMANCE OF THE IMPLEMENTATIONS IN SPARK

	C/C++	SPARK		
Run-time checks	No	Yes	No	No
Math library	C	Ada	Ada	C
VFH+	646	2567 (4.0)	892 (1.4)	861 (1.3)
ND	83	318 (3.8)	174 (2.1)	85 (1.0)
SND	165	752 (4.6)	493 (3.0)	449 (2.7)

iterations of the algorithms in typical navigation scenarios[7]. All results have been measured using a desktop PC with Intel Core i5-3230M 2.6 GHz CPU, 8GB RAM and 64-bit Ubuntu 12.04 Linux operating system. Both the SPARK and C/C++ code has been compiled with the GNAT GPL 2013 compiler, which is based on GCC 4.7.4, with "-O3" optimization settings.

In the worst case, a single iteration of the VFH+ algorithm took 2.6 ms, but the other algorithms can be executed at much higher update rates even with all run-time checks enabled (Table II). Heavy use of arrays and run-time assertions in VFH+ algorithm makes it up to four times slower than the original C++ code. The ND algorithm with run-time checks disabled performs even better in SPARK, but only when compiled with the C mathematical library, which implements many of its functions in software. Ada compiler uses the hardware floating-point unit for trigonometric functions, which is slower, but gives more accurate results (this accuracy is mandated by the Ada standard). The hot spots in the performance of the SND algorithm are the operations on vectors. The API of the SPARK formal containers library prevents the compiler from optimizing the code as effectively as for the C++ standard template library.

The explicit annotations for each of the analyzed implementations take less than 5% of the code, but complete proofs involve hundreds of verification conditions (Table III). GNATprove's effectiveness builds upon the use of Satisfiability Modulo Theories (SMT) solvers, which automatically decide whether a given verification condition holds in all configurations of the program variables [3]. We evaluated several SMT solvers supported by the Why3 platform and get the best results with Alt-Ergo (the default solver shipped with SPARK tools) and Z3 (Table IV).

The exact ranges and accuracy of floating-point values can be easily defined only for sensor data. Manual propagation of this information through the code to each intermediate result is difficult and error-prone. Without such information, SMT solvers typically reach a timeout limit when attempting to verify floating-point operations. To speed up the static analysis we skipped such proofs and (optimistically) assumed that floating-point operations are exact and do not overflow.

Trigonometric functions pose another difficulty, as SMT solvers do not handle them natively, but using generic procedures and a limited set of axioms. Surprisingly, we got better results by excluding some axioms from the default set, in particular those that define values of $\sin$ and $\cos$ for 0, π and $\frac{\pi}{2}$. Solvers that handle trigonometric functions natively might be more effective in our application if they were not limited only to the theory of real-number arithmetic [14]. Verification conditions resulting from our code are typically "polluted" with other theories, such as integers.

[7]The verified SPARK code and both the original and fixed C/C++ implementations, together with detailed instructions for reproducing our results, are available at `http://github.com/ptroja/spark-navigation`.

TABLE IV
NUMBER OF DISCHARGED VERIFICATION CONDITIONS AND THE RUNNING TIME OF STATIC ANALYSIS

	Alt-Ergo 0.96	Z3 4.3.1	Alt-Ergo & Z3 combined	**Total**
VFH+	633 11 min	699 37 min	701 48 min	**748**
ND	462 17 min	482 21 min	483 41 min	**540**
SND	350 29 min	366 6 min	366 36 min	**375**

A. VFH+

Part of the 6% of unproved verification conditions in the VFH+ implementation result from several assertions that we put as substitutes of a single type invariant. Type invariants are not yet supported by the current release of the SPARK tools. The remaining proofs depend on missing assumptions about the relationships between algorithm parameters. Their discovery requires further investigation and, ideally, a tool support for type invariants.

B. ND

The ND algorithm is implemented in C using a programming style that is particularly difficult to verify. For example, invalid array indices are indicated as -1 and code with several branches is put into a 200-lines long loop body. Other parts of the code directly call trigonometric functions instead of referring to geometric primitives such as points and angles.

Despite the disputable coding style of this algorithm, it was the only for which our experiments did not revealed any run-time error. Almost 90% of the verification conditions are proved. The remaining ones results from code with particularly high complexity metrics. Their proofs depend on invariants, which discovery requires further investigation.

C. SND

The SND algorithm is based on the ND algorithm, but the implementations differ substantially. The C++ code of SND is shorter and easier to understand than the code of ND. It uses vectors and cursors instead of arrays and pointers. Subprograms related to geometric concepts, i.e. positions, angles and poses, are encapsulated within a library and not mixed with the main code.

The clarity of the SND code turns out to be essential for proving its run-time safety: 97% of the verification conditions are proved automatically and the remaining ones

TABLE III
VERIFICATION CONDITIONS BY CATEGORY

	Explicit annotations					Implicit run-time checks						Total
	Pre-conditions*	Post-conditions	Loop invariants*	Loop variants	Assertions	Divisions	Integer overflows	Floating-point overflows	Subtype ranges	Array indices	Record discriminants	
VFH+	46 (3)	5	18 (9)	0	23	36	36	120	100	102	262	**748**
ND	83 (18)	10	8 (4)	2	3	54	23	254	53	50	0	**540**
SND	104 (9)	9	14 (7)	2	30	29	1	140	22	0	24	**375**

* Separate verification conditions are generated for each call to subprogram with precondition, and similarly for initialization and preservation of each loop invariant; the numbers of explicit annotations are given in brackets.

are easy to examine. Two are related to normalizing angles to intervals of the width 2π and require formalization of the floating-point remainder function. Another two are related to explicit assumptions about bounded vectors and are required by the current release of the formal containers library.

Two unproved assertions reveal hidden assumptions of the authors of the code about the relationships between algorithm parameters. Finally, three verification conditions require reasoning about the relationships between trigonometric functions. We checked them by pen and paper, but this can also be done more formally using an interactive theorem prover.

In addition to run-time safety, termination of the VFH+ and SND algorithms is verified by proving loop variants for all of the `while` loops in their code.

V. RELATED WORK

Before translating the analyzed code to SPARK we attempted to verify it using several state-of-the-art tools for C/C++. To avoid the complexity of parsing C++ source code, the majority of tools, e.g. CBMC[8] and KLEE[9], operate on intermediate compiler representation of the code, which lacks much of the information needed for the proofs. The notable exceptions are Frama-C[10] and VCC[11], which follow the same approach for C as GNATprove for SPARK, but do not support C++. According to our experiments, none of these tools is suitable for automated verification of "real-world" robot control software, at least not without substantial effort in annotating the source code.

SPARK is used in diverse applications including avionics, security and medical systems, but—to the best of our knowledge—it has not been yet used for verification of robot navigation software. In particular, we heavily rely on the features that have been introduced in the latest SPARK 2014 release of the language, such as the formal containers library.

Existing proofs of robot navigation algorithms, such as the popular *dynamic-window approach*, verify only their functional properties and not their implementations [15]. Our approach is most closely related to verification of software that prevents a mobile robot from colliding with static obstacles [16]. The authors developed a custom tool for static analysis of MISRA-C code and verified both their algorithm and its implementation. They relied solely on an interactive theorem prover—a tool that is very difficult to use by non-experts. In contrast, Ada and SPARK are quite similar to C++ and thus are much more straightforward to adopt.

[8] `http://www.cprover.org/cbmc/`
[9] `http://ccadar.github.io/klee/`
[10] `http://frama-c.com/`
[11] `http://vcc.codeplex.com/`

VI. CONCLUSIONS

Software verification is traditionally perceived as expensive and difficult to apply to "real-world" code. We demonstrated that this is not the case for robot navigation software, provided that it is implemented in a programming language designed with software-reliability in mind. We believe that the effort of learning and using a dedicated programming language is a justifiable investment for all those who are concerned about the possible disastrous effects of bugs in their code. Run-time assertions automatically injected by the Ada compiler and advanced tools for static analysis of SPARK code make detecting bugs and proving their absence much easier than what is currently possible with C/C++.

Directions for future work include verification of path-planning code and integrating SPARK with ROS [8]. Floating-point operations in our code can still result in overflow errors at run-time. We plan to apply complementary techniques, such as *abstract interpretation*, to prove their correctness. We are also interested in extending the scope of verification to multi-threaded software and using the RavenSPARK subset of Ada—a combination of SPARK and a restricted concurrency profile of Ada [17]. By publishing our results and code online we hope to encourage uptake by robot software developers, and that their future code releases will be accompanied with evidence of code correctness.

ACKNOWLEDGEMENT

We would like to thank Claire Dross for support in getting the formal containers library to work, the SPARK developers for their excellent tools, and the authors of the original C/C++ implementations for releasing their code online thus making our analysis possible.

REFERENCES

[1] J. Barnes, *SPARK: The Proven Approach to High Integrity Software*. Altran Praxis, 2012.

[2] C. Comar, J. Kanig, and Y. Moy, "Integrating formal program verification with testing," in *Proceedings of the Embedded Real Time Software and Systems conference, ERTS^2 2012*, Toulouse, Feb. 2012.

[3] J.-C. Filliâtre and A. Paskevich, "Why3 — where programs meet provers," in *Proceedings of the 22nd European Symposium on Programming*, ser. Lecture Notes in Computer Science, M. Felleisen and P. Gardner, Eds., vol. 7792. Springer, Mar. 2013, pp. 125–128.

[4] J. Borenstein and Y. Koren, "The vector field histogram-fast obstacle avoidance for mobile robots," *IEEE Transactions on Robotics and Automation*, vol. 7, no. 3, pp. 278–288, 1991.

[5] J. Minguez and L. Montano, "Nearness diagram (ND) navigation: Collision avoidance in troublesome scenarios," *IEEE Transactions on Robotics and Automation*, vol. 20, no. 1, pp. 45–59, 2004.

[6] J. W. Durham and F. Bullo, "Smooth nearness-diagram navigation," in *IEEE/RSJ International Conference on Intelligent Robots and Systems*, 2008, pp. 690–695.

[7] G. Biggs, R. B. Rusu, T. Collett, B. Gerkey, and R. Vaughan, "All the robots merely players: History of Player and Stage software," *Robotics Automation Magazine, IEEE*, vol. 20, no. 3, pp. 82–90, 2013.

[8] M. Quigley, B. Gerkey, K. Conley, J. Faust, T. Foote, J. Leibs, E. Berger, R. Wheeler, and A. Ng, "ROS: an open-source Robot Operating System," in *Proceedings of the Open-Source Software workshop at the International Conference on Robotics and Automation*, 2009.

[9] P. Baudin, P. Cuoq, J.-C. Filliâtre, C. Marché, B. Monate, Y. Moy, and V. Prevosto, "ACSL: ANSI/ISO C specification language," CEA LIST and INRIA, Tech. Rep. Revision 1.7, 2013.

[10] C. Dross, J.-C. Filliâtre, and Y. Moy, "Correct code containing containers," in *Tests and Proofs*, ser. Lecture Notes in Computer Science, M. Gogolla and B. Wolff, Eds. Springer, 2011, vol. 6706, pp. 102–118.

[11] R. C. Seacord, *Secure Coding in C and C++*, 2nd ed. Addison-Wesley Professional, Apr. 2013, ch. Integer Security.

[12] E. W. Dijkstra, *A Discipline of Programming*. Prentice Hall, 1976.

[13] X. Yin, J. C. Knight, E. A. Nguyen, and W. Weimer, "Formal verification by reverse synthesis," in *Computer Safety, Reliability, and Security*, ser. Lecture Notes in Computer Science, M. D. Harrison and M.-A. Sujan, Eds. Springer Berlin Heidelberg, 2008, vol. 5219, pp. 305–319.

[14] B. Akbarpour and L. C. Paulson, "Metitarski: An automatic theorem prover for real-valued special functions," *Journal of Automated Reasoning*, vol. 44, no. 3, pp. 175–205, 2010.

[15] S. Mitsch, K. Ghorbal, and A. Platzer, "On provably safe obstacle avoidance for autonomous robotic ground vehicles," in *Robotics: Science and Systems*, 2013.

[16] H. Täubig, U. Frese, C. Hertzberg, C. Lüth, S. Mohr, E. Vorobev, and D. Walter, "Guaranteeing functional safety: design for provability and computer-aided verification," *Autonomous Robots*, vol. 32, pp. 303–331, 2012.

[17] A. Burns, B. Dobbing, and T. Vardanega, "Guide for the use of the Ada Ravenscar Profile in high integrity systems," *ACM SIGAda Ada Letters*, vol. 24, no. 2, pp. 1–74, 2004.