

Experiences in Applying Formal Verification in Robotics^{*}

Dennis Walter, Holger Täubig, and Christoph Lüth

Deutsches Forschungszentrum für Künstliche Intelligenz
Bremen, Germany

{Dennis.Walter,Holger.Taeubig,Christoph.Lueth}@dfki.de

Abstract. Formal verification efforts in the area of robotics are still comparatively scarce. In this paper we report on our experiences with one such effort, which was concerned with designing, implementing and certifying a safety function for autonomous vehicles and robots. We outline the algorithm which was specifically designed with safety through formal verification in mind, and present our verification methodology, which is based on formal proof and verification using the theorem prover Isabelle. The necessary normative measures that are covered are discussed. The algorithm and our methodology have been certified for use in applications up to SIL 3 of IEC 61508-3 by a certification authority. Throughout, issues we recognised as being important for a successful application of formal methods in the domain at hand are highlighted. These pertain to the development process, the abstraction level at which specifications should be formulated, and the interplay between simulation and verification, among others.

1 Introduction

While in some areas such as avionics, formal verification is well established in the development process, in other areas its use is still rare. One such area is robotics, in particular service robotics. This paper reports on our experiences when applying formal verification techniques to the certification of an algorithm calculating dynamic safety zones for an autonomous vehicle or robot which prevent it from colliding with stationary obstacles. Robotics as an application area offers its own challenges. Algorithms, often based on approximations and heuristics and implementing rather sophisticated computations such as the area covered by a braking robot in our case, play a central role. This leads to an increase in the importance of functional correctness. Further, the development process should be flexible, and allow us to iteratively develop both algorithms and specifications from the overall safety requirements. This can be contrasted to a rigid V-model with its strict separation of development phases which makes it expensive to ‘go back’ to earlier phases of the development when one discovers that the implemented algorithm is, in fact, safe but unusable in practice.

^{*} This work was funded by the German Federal Ministry of Education and Research under grants 01 IM F02 A and 01IS09044B.

Our methodology for specifying and verifying functional properties emphasises proving as well as testing. We demonstrate which measures can be covered by formal proof, even though the relevant standard IEC 61508-3 [10] focuses on testing, and argue that formal proof, on the other hand, allows us to relax the bureaucratic elements of the development process without losing reliability.

This paper is structured as follows: in Sec. 2, we give an overview over the project, showing the actual algorithm, the formal domain model, and our approach to specification and verification. In Sec. 3, we review our experiences made during the certification of the algorithm, pertaining to formal verification in the robotics domain and the development process we used, and highlight benefits and limitations of our approach.

2 The SAMS Project

2.1 The Safety Algorithm for collision avoidance

The algorithm which has been verified in SAMS is a collision avoidance algorithm, which protects a vehicle moving in a plane, e.g. an automated guided vehicle or service robot, from colliding with static obstacles. For that purpose, a safety zone is computed using the algorithm described in this section and then checked via a laser scanner whether there is an obstacle inside the safety zone. If so, the moving vehicle has to stop, otherwise it can safely continue its movement. The purpose of the verified algorithm is to compute a *safety zone* that is a superset of the *braking area* covered by the vehicle during braking (Fig. 1a).

Input. The algorithm takes as input intervals $[v_{min}, v_{max}]$ and $[\omega_{min}, \omega_{max}]$ which safely cover the measured translational and rotational velocities v and ω of the vehicle, a set of points $[R_i]_{i=1}^n$ which define the robots shape as their convex hull, and a list $(v_1, s_1), \dots, (v_m, s_m)$ of braking measurements for straight forward movements of the vehicle. Each pair consists of a velocity v_j and the corresponding measured braking distance s_j . We assume there is at least one measurement, taken at maximum speed. Furthermore, a latency t is given which parameterises the time the vehicle continues to drive with velocity (v, ω) before it starts to brake; it comprises the sum of the safety functions cycle time as well as any latency in the input data and the reaction time of the brakes.

Assumptions. First and foremost, we assume the robots braking trajectory to be a straight line or a circular arc; in other words, the steering of the vehicle remains fixed until the vehicle has completely stopped. A second assumption of the braking model is an energy consideration. It allows to transfer the braking distance measurement from straight motion to motion along a circle or even turning on the spot. Together, both assumptions establish the main braking model computation, which takes the initial velocity vector (v, ω) and delivers the so called *braking configuration* (s, α) . The braking configuration (s, α) consists of arc length s and angle α of the robots circular braking trajectory. In the case of $\omega = 0$, the angle α becomes zero and the braking configuration (s, α) describes a

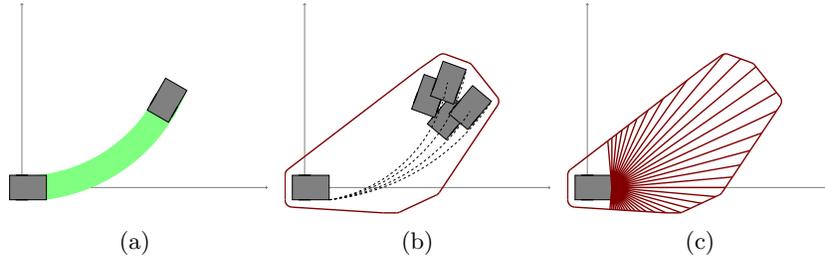


Fig. 1. Calculating the safety zones: (a) area covered by the vehicle during braking with a single braking configuration (s, α) ; (b) safety zone covering all braking configurations $[s_{min}, s_{max}] \times [\alpha_{min}, \alpha_{max}]$; (c) safety zone transformed into a laser-scan.

straight line of length s . Finally, besides static obstacles we assume the braking behaviour of the vehicle to be time-independent and location-independent.

Algorithm. First, conservatively transform $[v_{min}, v_{max}] \times [\omega_{min}, \omega_{max}]$, the velocity configuration area, into the braking configuration area $[s_{min}, s_{max}] \times [\alpha_{min}, \alpha_{max}]$. The transformation *braking-configuration* (v, ω) describes a movement consisting of moving for time t with velocity (v, ω) , and then braking on a circular arc that retains the radius defined by v and ω .

[Step 1] For (v, ω) in $\{v_{min}, v_{max}\} \times \{\omega_{min}, \omega_{max}\}$, compute the braking configuration (s, α) as follows, and determine minimum and maximum s_{min} , s_{max} , α_{min} , and α_{max} of the four results:

$$(s, \alpha) = \text{braking-configuration}(v, \omega) \quad (1)$$

Then, compute the safety zone in terms of a finite set of points $[P_k]_{k=1}^K$ and a buffer radius q (Fig. 1b). The safety zone is an area $A^+([P_k]_{k=1}^K; q)$, given by the union of the convex hull of $[P_k]_{k=1}^K$ and the set of all points having distance of at most $q > 0$ to any point of that convex hull:

$$A^+([P_k]_{k=1}^K; q) = \{P+Q \mid P \in \text{conv} \{[P_k]_{k=1}^K\}, |Q| \leq q\} \quad (2)$$

[Step 2a] To compute points P_k : For all $(s, \alpha) \in \{s_{min}, s_{max}\} \times \{\alpha_{min}, \alpha_{max}\}$, compute the safety zone for the single braking configuration (s, α) in terms of a set of $n \cdot (L + 2)$ points¹ as the convex hull of

$$H_{s,\alpha} = \{[U_{i,s,\alpha}^1, U_{i,s,\alpha}^2, V_{i,s,\alpha}^0, \dots, V_{i,s,\alpha}^{L-1}]_{i=1}^n\},$$

where $U_{i,s,\alpha}$ and $V_{i,s,\alpha}$ are given as follows (for i in $1, \dots, n$):

$$\begin{aligned} U_{i,s,\alpha}^1 &= R_i & U_{i,s,\alpha}^2 &= T\left(\frac{s}{L}, \frac{\alpha}{L}\right) \cdot R_i \\ V_{i,s,\alpha}^0 &= U^1 + Q\left(\frac{\alpha}{L}\right)\frac{1}{2}(U^2 - U^1) & V_{i,s,\alpha}^j &= T\left(\frac{j \cdot s}{L}, \frac{j \cdot \alpha}{L}\right) \cdot V^0 \end{aligned} \quad (3)$$

¹ The parameter L determines the number of auxiliary points when computing the convex hull of an arc.

with $T(s, \alpha) = \begin{pmatrix} \cos \alpha & -\sin \alpha & s \operatorname{sinc} \frac{\alpha}{2} \cos \frac{\alpha}{2} \\ \sin \alpha & \cos \alpha & s \operatorname{sinc} \frac{\alpha}{2} \sin \frac{\alpha}{2} \\ 0 & 0 & 1 \end{pmatrix}$ and $Q(\alpha) = \begin{pmatrix} 1 & \tan \frac{\alpha}{2} & 0 \\ -\tan \frac{\alpha}{2} & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}$.

Now, $[P_k]_{k=1}^K$ is the result of a standard convex hull algorithm like Graham scan applied to the union of the $H_{s,\alpha}$ for all $(s, \alpha) \in \{s_{\min}, s_{\max}\} \times \{\alpha_{\min}, \alpha_{\max}\}$.

[Step 2b] The buffer radius q includes a conservative error approximation for the algorithm, and is given as

$$q = \frac{1}{6} \left(\frac{\alpha_{\max} - \alpha_{\min}}{2} \right)^2 \max\{|s_{\max}|; |s_{\min}|\} + \left(1 - \cos \frac{\alpha_{\max} - \alpha_{\min}}{2}\right) \max_{1 \leq i \leq n} \{|R_i|\} \quad (4)$$

[Step 3] Finally, $[P_k]_{k=1}^K$ is transformed into scanner coordinates, and the safety zone $A^+([P_k]_{k=1}^K; q)$ is sampled into a laser-scan like representation (Fig. 1c).

Output and guarantees. If the vehicle satisfies the assumptions described above and if the input parameters are correct or at least conservative, the algorithm guarantees the correctness of the safety zone. This means that the vehicle will always be able to stop within the area defined by the braking zone. More precisely, no part of the vehicle will leave that area at any time while first driving with constant velocity (v, ω) for time t and then braking down to standstill.

The guarantee given consists of two major properties: first, correctness of the braking model computation (Step 1), and second, the correct and strictly conservative computation of the area $A^+([P_k]_{k=1}^K; q)$ (Step 2). Both properties have been formally proven in the Isabelle theorem prover.

2.2 Formalising the Domain Model

In order to be able to state and verify safety properties about the algorithm introduced in Sec. 2.1, we need a formal model of the *domain* of the algorithm, that is the world of two-dimensional moving objects. This model is used for the specification and verification of the concrete source code. Therefore, by formal model we mean a collection of theories of the theorem prover Isabelle [14], and not merely a pen-and-paper formalisation.

The contour of the robot, for instance, is modelled as a convex polygon, and obstacles are simply connected sets of points. The main safety property we will need to formalise is that the area traversed by the robot while braking from velocity (v, ω) is covered by the safety zone calculated for that velocity, given by equation (2) above. In the notation of Sec. 2.1, we require the following to hold:

$$\text{braking-area}(v, \omega) \subseteq A^+([P_k]_{k=1}^K; q) \quad (5)$$

Isabelle provides a rich base of theories concerned with concepts such as real numbers, polynomials, or set theory. In the process of domain formalisation this base is extended with the concepts and theorems relevant to our concrete model. For example, the following shows the definition of the property of convex sets of points; this definition has been copied verbatim from the corresponding Isabelle theory file ($*_R$ denotes scalar multiplication):

definition

```
convex :: "Point set  $\Rightarrow$  bool"
```

where

```
"convex K  $\equiv$  ( $\forall x \in K. \forall y \in K. \forall t. (0 \leq t \wedge t \leq 1) \longrightarrow$   

  ( $t *_R x + (1-t) *_R y \in K$ )"
```

Or in words, a set of points is convex iff for any two points x, y all points on the line between them are in the set as well. We can now define the convex hull of a set of points X as the intersection of all convex sets K containing X :

definition

```
convex_hull :: "Point set  $\Rightarrow$  Point set"
```

where

```
"convex_hull X =  $\bigcap \{K . convex K \wedge X \subseteq K\}$ "
```

These definitions are almost identical to what one can find in mathematical textbooks. This is particularly valuable in a certification context, as theory files can be reviewed without an in-depth knowledge of Isabelle syntax. The Isabelle equivalent of (5) involves slightly too many concepts whose definitions we must elide for reasons of brevity. To give the reader a taste of a more involved model concept, we present a theorem about the approximation of an arc by the convex hull of three points:

lemma arcpoint_in_convex_hull':

```
"[|  $|\varphi| < \pi$ ;  $0 \leq t \wedge t \leq 1$ ;  

  Q = arcendpoint s  $\varphi$  P; K = convexpoint P Q  $\varphi$  ]  

 $\implies$  arcendpoint (t*s) (t* $\varphi$ ) P - P  $\in$  convex_hull {K-P, Q-P, 0}"
```

If Q is the endpoint of a circular arc (defined by its length s and angle φ), starting at P , then the convex hull of P, Q and a third point K (whose computation via *convexpoint* is irrelevant here) will contain every point on the arc (computed via *arcendpoint* by scaling s and φ).

Our domain modelling consists of 11 theory files, containing about 110 definitions and 510 lemmas and theorems. It was developed in about five months by a researcher with a good background in mathematics, but no previous knowledge of Isabelle, supporting the claim that mathematics is the key, and the technicalities of Isabelle do not distract one unduely from the actual formal development.

2.3 Specification & Verification

Certifying a software module involves verification on several levels: design requirements need to be traced back to system (safety) requirements, code needs to be verified according to the V&V plan and against the design specification, and during integration there are several verification activities based on requirements set forth in the corresponding specification phase. For each of these levels a well-defined procedure for specifying requirements is needed. In the SAMS project we stressed *functional correctness*, the verification of the functional behaviour of concrete code. This comprises both the absence of runtime errors like array-out-of-bounds or division-by-zero (a property we call *program safety*), and correctness of the results of computations, as defined by formal specifications.

We consider functional correctness important because it is mandated by standards like IEC 61508-3 to ensure program safety properties on the code level, and moreover, because robotics algorithms as the one described in Sec. 2.1 involve very complex computations whose correct implementation is hard to verify by a mere code review. In the terminology of Heitmeyer et al. [8] they deserve a thorough and detailed analysis which is not possible in *operational models* (e. g. state machine models) and for which an *axiomatic* approach like the one presented here is well suited.² We consider a purely operational analysis insufficient, because it is equally important to analyse the data-related and the control-related aspects of software systems in robotics, where operational models focus on the latter. The former lends itself ideally to a declarative, ‘axiomatic’ specification. Moreover, models for robotics require non-discrete data as measurements of real world entities are involved.

As an example from our own code: one subroutine of the algorithm of Sec. 2.1 approximates the arc along which the vehicle brakes by a polygon. The complete specification (roughly stating that the arc is included in the area covered by the polygon) invariably leads to the use of mathematical concepts like convex sets of points, set intersection, etc. Furthermore, the correct execution of the system’s safety function crucially depends on the correct design and implementation of this approximation. In particular, a flaw in this subroutine can be very hard to detect. This is an important difference to, e. g., program crashes (which an external watchdog may detect) or Boolean circuits (whose input domain can be tested much more thoroughly).

To express the functional properties of interest we designed a formal language for the high-level specification of the functional behaviour of C programs. The language lies in the tradition of design by contract languages like JML [4] or ACSL [2], where program functions are annotated with preconditions, post-conditions and a modification frame limiting the effect of function execution on memory changes. Our language additionally allows to include higher-order logic expressions in the syntax of the theorem prover Isabelle in specifications. This gain in language expressivity is the crucial ingredient for allowing more abstract specifications in which program values are put in relation to their corresponding domain values. The desired properties of functions are then expressed in terms of the domain language as it was formalised in Isabelle.

An example specification is given in Fig. 2: it uses the concepts *is-RT* and *RT* from the domain formalisation as well as the function composition operator \circ of Isabelle/HOL to concisely express that `comp_transform` is an operation that composes two rigid body transforms. *RT* is what we call a representation function, which lifts a C value of type `RigidTransform` into its domain equivalent. *is-RT* is a predicate that recognises all C values that actually represent rigid body transforms. (The internal representation of `RigidTransform` are 3×3 ma-

² We observe a slight collision of terminologies here: in the theorem proving community, an axiomatic approach is distinguished from a definitional approach, in which theories are derived from first principles and new concepts are built on top of existing ones. We do not use this interpretation of ‘axiomatic’ in this paper.

```

/*@
  @requires  $\$!is\_RT(a2b) \ \&\& \ \$!is\_RT(b2c)$ 
  @memory \valid{a2b, b2c, a2c} &&
    *a2c <*> (*a2b <+> *b2c)
  @ensures  $\$!is\_RT(a2c) \ \&\&$ 
     $\$\{ \hat{RT}\{a2c\} = \hat{RT}\{b2c\} \circ \hat{RT}\{a2b\} \}$ 
  @modifies *a2c
  @*/
void comp_transform( const RigidTransform * a2b,
                    const RigidTransform * b2c,
                    RigidTransform * a2c );

```

Fig. 2. An example specification of a C function, directly using the domain vocabulary as defined by the formalisation in Isabelle/HOL. The **@memory** annotation requires that `a2c` is not aliased with `a2b` nor with `b2c` and that all three are valid pointers.

trices, which hence include other transformations, too.) Further details about the specification language and how functions can be proven correct in Isabelle w. r. t. their specification have been described in a previous paper [12].

We now sketch the steps that are taken in the specification and verification workflow. To reiterate our setting: To attach specifications to code, there must be code; while in the classic V-model, code enters rather late in the process, to specify we merely need the function interfaces, not the complete implementation. Moreover, in our case implemented prototypes are available early in the development, a point we elaborate on in Sec. 3.2. Assuming that the code for those parts of the program that are put under formal scrutiny are available, and that the domain model has been sufficiently formalised so that at least the relevant definitions that will be used in specifications exist, our workflow looks as follows (see also Fig. 3): (1) C function interfaces (declarations) are annotated with their respective specifications: those functions that implement operations with an analogue in the domain, like geometric transformations, are specified with the help of the domain vocabulary given by the formalisation; (2) In specification reviews both the completeness of the specification w. r. t. more high-level, natural language specifications as well as implementation-related issues are discussed. The latter include issues like the restructuring of code to ease verification, or the elimination of language constructs that are not supported by the verification environment; (3) After the functions have been implemented, they are checked for obvious deviations from their specified behaviour in code reviews. At least one programmer, one specifier or domain expert and one verifier takes part in these meetings. Such meetings give the verifier an understanding of how the code works, which is crucial for the verification to succeed. (4) The translation of C functions into the Isabelle formalisation is done modularly: to translate function `foo` for a verification attempt, only its source code and the specifications of all functions called by `foo` as well as obviously that of the function itself are required. A front-end parses and analyses these entities and emits Isabelle terms

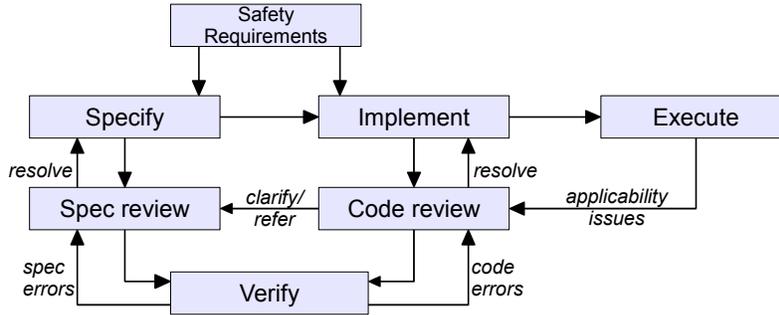


Fig. 3. The specification and verification workflow, not enforcing a temporal dependency between design specification and code, which is ensured by the final verification.

representing their abstract syntax; (5) Making use of the domain formalisation and a couple of automatic procedures written in Isabelle, a human verifier interactively proves these functions correct. This is a labour-intensive part, and the core of formal verification. (6) In the case that a function cannot be verified, a specification or code review is called and as soon as the error has been understood, modifications on either the design specification, the code, or the proof script are undertaken, leading to a reiteration of the process just described.

3 Lessons Learnt

3.1 Formal Verification in the Robotics Domain

Challenges. The functionality of robotics systems is mainly implemented in software, making use of increasingly complex algorithms. With growing system functionality the difficulties of ensuring safety increase, in particular in the face of moving towards mixed human-robot work places instead of physically separating machine operators and robotic devices. Functional correctness occupies a growing fraction of the overall integrity of systems. This requires deep analytical safety considerations and makes formal verification an eligible tool in the robotics domain. Characterised from a safety point of view, the domain stands out by its rich specifications, which usually are not as simple as “temperature never exceeds 90 °C”, and its rich domain involving high level concepts from mathematics, kinematics, and other areas of physics.

In practice, applying formal verification in the robotics domain faces the conflict between real-world applications involving unstructured environments and inaccurate sensors, and their idealised modeling in specifications and the formalised domain. Addressing this problem is not unique but especially important for projects applying formal verification in real world applications. Safety requires that reality conforms to the assumptions made in the model.

Another challenge in the robotics domain is the conflict between safety and practical issues like availability. It is quite easy to design a safe algorithm which

is unusable in practice. For example, a service robot will not be able to navigate through doors anymore if its safety zones are calculated too large. Avoiding these safe but non-applicable results as well as identifying the aforementioned kind of conflicts between models and reality is mainly done by evaluating parts of the implementation in practice. Occuring conflicts may result in changes of the design specifications. For that reason, our development process is very iterative and code-centric. Availability is not verified, it is tested in practice and in simulations, so it is important to be able to run the algorithm early in the development process.

Successful design and verification of robotics algorithms. Two concepts that proved helpful were the explicit use of intervals to accommodate for imprecision, and algorithms and representations from computational geometry.

To overcome the discrepancy between the real world and the idealised domain, and to account for imprecise measurements, our algorithm calculates safety zones for sets of velocities $[v_{min}, v_{max}] \times [\omega_{min}, \omega_{max}]$ instead of single ones.

Another benefit came from the representation of objects as sets of points, which not only led to efficient computations, but also allowed for mathematically pleasing proofs for major parts of the algorithm. This seems to hold true for many representations and algorithms from computational geometry. Of course, other proofs were pure grind work, such as the proof of the following (for $\omega \neq 0$):

$$\sqrt{\left(\frac{\omega - \sin \omega}{\omega^2} - \frac{1 - \cos \omega}{\omega}\right)^2 + \left(-\frac{\sin \omega}{\omega} + \frac{1 - \cos \omega}{\omega^2} - \frac{\omega - \sin \omega}{\omega^3}\right)^2} \leq \frac{2}{3}$$

Domain. Robotics is well suited for formal verification. Formalising high-level concepts is admittedly very time-consuming. Nevertheless, much can be taken directly from textbooks so that the formalisation in Isabelle went rather smoothly. Moreover, the effort is worthwhile, as it allows simpler specifications and verification. The domain modelling is reusable for other projects, independent of a reuse of the implementation.

3.2 Specification Process

Verification as a joint effort. One aspect of formal verification is that because correctness relies on formal proof, it is not that crucial anymore to strictly separate the roles of tester and implementer. In contrast, the close cooperation between the verifier and the implementer boosted productivity in our case: verification became a joint effort. Writing specifications which validate the safety requirements, and can be formally verified, is not easy; it requires an understanding of the implementation, the domain model, and how the verification works. It is easy to specify something which is correct but cannot be verified; on the other hand, it is also a temptation to write low-level specifications which just restate what the code is doing in elementary terms without the abstraction required to state useful safety properties.

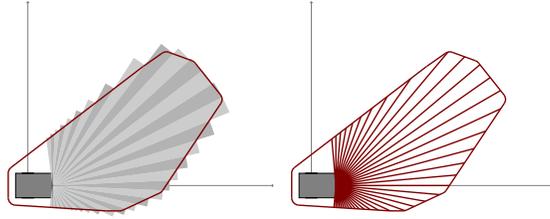


Fig. 4. Two ways of converting a buffered polygon into a laser scan representation.

A somewhat unusual example of a close collaboration between implementer and verifier is a change of the implementation induced by verifiability considerations. The function `abtasten` converts the buffered polygon into a sequence of vectors corresponding to a laser scan (Step 3 from the algorithm presented in Sec. 2.1; see Fig. 4). Initially, the specification interpreted the resulting sequence as the rays of an idealised laser scanner. We switched both specification and implementation to a sector-based interpretation, in which each result describes the whole area of a sector. This fitted in well with the other specifications and allowed us to specify the result simply as a superset of the actual safety zone, and was easier to verify formally.

Code-centric specification and verification. We experienced an interesting interplay between specification, implementation and application: at first, the specification required that if the speed of the vehicle exceeded the maximum speed for which a braking distance was measured (cf. Sec. 2.1), an emergency stop should be initiated. However, this turned out to be too restrictive: in typical applications, the measured maximum velocity v_m may be exceeded occasionally by a small margin, and initiating an emergency stop in these situations would severely reduce availability. Hence, the breaking distance for speeds larger than v_m was safely overapproximated, and the specification amended accordingly.

The importance of being formal. Formal specification necessitates to state requirements precisely. A beneficial side effect is that it focuses discussions and manifests design decisions. Besides the well-known issue of the ambiguities in natural language specifications, it turned out to be easier for specifiers *and* implementers to use the vocabulary of the domain formalisation to state these requirements and to reach agreement on their respective meaning. For quick sanity checks of specifications written down or modified during meetings, we provide tool support for the type-checking of specifications. This pertains both to code-related specification expressions (e. g., types of program variables) as well as Isabelle expressions used in code specifications. A typical specification meeting would end with a function specification reviewed and typechecked.

Beyond that, formal verification can uncover hidden assumptions, both in the specification and even in the domain (see also [6]). As an example, when verifying the overapproximation for speeds beyond maximum speed, it turned

out that the initially defined quadratic approximation was not enough, and a cubic one was necessary to formally prove the relevant properties.

3.3 Formal Verification in a Certification Process

V-model. The V-model of IEC 61508-3 asks for traceability between adjacent phases on the downward leg, i. e. from the system safety requirements down to the code, as well as ‘horizontal’ verification on the upward leg from code to the integrated and validated system, where appropriate tests ensure the satisfaction of all requirements. The model somewhat neglects model-based analysis and does not assign it a specific level; it might be considered part of the software architecture, but in any case has a direct link to the safety requirements. A definite strength of our methodology is the very strong link between this analysis level and the concrete source code (at the bottom of the V-model): Formal code verification in our methodology ensures both traceability between code and module design, *and* between module design and the analysis level. The main reason for this is the high level of abstraction of code specifications, in which the domain formalisation is directly embedded. For example, take the basic function computing a polygonal approximation of the curve described by a single point of the vehicle’s contour during an emergency stop. Its specification directly expresses that the area described by the returned polygon completely contains the braking curve in the two-dimensional environment model.

Modularity. Modular verification on a function-by-function basis allowed us to focus formal verification on those functions which are crucial to functional correctness; other functions may contain constructs that our tool cannot reason about, or may not pertain to global correctness (e. g., logging), and can be treated more adequately by manual review or informal proofs.

Open-minded authorities. To our surprise the external reviewers from the certification authority were quite open-minded towards the use of expressive (higher-order) formal logic for specifications and an interactive theorem prover for doing the actual verification. In our case this was Isabelle/HOL, but its specifics did not play an important role and HOL4 or Coq or any other well-known prover with an active research community, proper documentation and a large enough number of global usage hours would have worked.³

Certification of the tool itself. Even though there are indications that structured specification and verification actually increase cost-effectiveness [1], their use is most often induced by the external requirement of a safety certification. A convincing argument for a tool like ours is that its use covers several items on the list of required design and verification measures. Concretely, to claim compliance with IEC 61508-3, the measures listed in its Annexes A and B have to

³ We actually estimated the number of hours that Isabelle has been in serious use (as $2 \cdot 10^6$ hrs). This technique of showing that a tool is ‘proven in use’ is commonly applied for non-certified compilers.

be considered. As confirmed by the certification authority, our tool covers several of these, which we will now briefly discuss. With regard to software design and development (A.4), four out of six measures are covered: the use of formal methods, of computer-aided design tools, of design and coding guidelines, and of structured programming. Missing are defensive programming and modularisation. The standard interprets modularisation structurally, and our tool does not apply code metrics.⁴ In contrast, the modularity we do achieve is of a more *behavioural* nature: the effect of a function is summarised in its interface specification, even though the function body might be of arbitrary size and complexity.

Concerning software verification measures (A.9), we cover formal proofs and static analysis. The latter includes the measures marginal value analysis, control as well as data flow analysis and symbolic execution. Whereas our Hoare-logic style verification resembles a symbolic execution, many properties that are derived from the other analyses, like ensuring that only initialised variables are read, are also subsumed by formal verification.

However, most of the work in a verification effort goes into testing, so one would require that the overall amount of functional testing can be reduced in a development process using formal verification. In our case, the only tests that had to be performed on the module level were related to over-/underflow and numerical stability. No functional testing had to be performed for the formally verified units, due to the level of detail at which both specifications and the programming language are modelled.

Limitations. Our tool focuses on functional correctness, and does not consider aspects like execution time analysis and bounds, resource consumption, concurrency, and the interface between hardware and software. This is a clear separation of concerns, as it is becoming common consensus that only the use of multiple, specialised tools and methodologies can achieve a high level of confidence in software [9]. There are further limitations in the realm of functional properties and run-time errors. Like other formalisations, we idealise the numerical domains that programs work on from bounded integers and floating-point numbers to mathematical integers and real numbers, which may in exceptional cases result in undetected run-time errors (see [12] for an example). The price we had to pay to obtain a formalisation in which interesting, abstract, functional properties can be proved with tolerable effort was a slight mismatch between the actual and the formal semantics.

Some more notes on practical formal verification. There are of course also problems with using formal verification as described here. A major annoyance is the fragility of proofs, i. e. their lack of robustness w. r. t. changes in source code. This particularly hurts in the face of interactive verification: proofs are not generated automatically by a push-button tool, but proofs scripts are written by humans – even if they sometimes only consist of a sequence of calls to automatic proof

⁴ In practice, functions which can be formally verified with tolerable effort adhere to these structural properties anyway.

tactics. We easily support ‘regression verification’, i. e. the automatic checking of all existing proofs of correctness against modified source code as well as modified specifications. Unfortunately, however, many proofs ‘break’ even under only minor modifications like the rearrangement of statements or a semantics-preserving rewriting of expressions, so that the proof scripts need to be adapted manually.

4 Conclusions

The SAMS project is an example of the successful application of formal verification in a certification context. The algorithm and implementation have been certified as conformant to the requirements of IEC 61508-3 Software SIL3 development by TÜV Süd. The same goes for the verification environment, which has also been confirmed as according to IEC 61508-3, covering various measures in the appendix as elaborated above. Both the tool and the domain modelling can be reused in other projects.

Related Work. We are not aware of many other formalisations in the robotics domain, except for specific, idealised algorithms like Graham scan [13]. Other certification efforts using formal verification in our sense include [1], which also use pre-/postconditions, but in a discrete domain; cf. also recent work concerning the verification of operating systems using Isabelle [16], or the VCC framework [5]. Most of these results idealise floating-point numbers to reals; an exception are Boldo and Filliâtre [3], who verify floating-point computations with exact error margins, something which in the robotics domain would be particularly valuable. Peleska [15] integrates formal approaches and testing, using abstract interpretation. It would be interesting to reuse the results of analyses like these for the formal verification, in particular to discharge program safety proof obligations. Haddadin et al. [7] perform a systematic evaluation of safety in human-robot interaction, quantifying injury risks based on experiments and classifying contact scenarios. Their work is upstream to ours, as it contributes important data for a hazard analysis and helps to improve safety by construction, while our approach assumes the criticality of collisions and aims at avoiding them. Krishna et al. [11] claim to develop provably safe motion strategies for mobile robots, but provide validation merely in terms of classical simulation and experimental results.

Summary. This paper has presented our experiences when conducting the formal verification and certification of a robotics algorithm. To close, we would like to recap our three main points. Firstly, we have argued that *functional correctness* is a key aspect of system integrity in robotics applications. Secondly, it is important to have a *strong link* from safety concepts down to the executable code. This was achieved by including domain model concepts directly in the specifications. Thirdly, because the correctness of all proofs are checked by Isabelle, we could relax some of the formalities of the development process to the benefit of all. Instead of using a rigid V-model, we had a convergence of both design specifications and implementation down to verified implementation in an *iterative process* starting from the initial safety requirements.

Besides the tool itself, we hope that the experiences laid out in this paper might be of use to other researchers and practitioners. We envisage a similar approach, using our tool or similar ones, to be applicable in all areas concerned with functional correctness to the degree that robotics is.

References

1. J. Barnes, R. Chapman, R. Johnson, J. Widmaier, D. Cooper, and B. Everett. Engineering the tokeneer enclave protection software. In *ISSSE'06*. IEEE Computer Society, 2006.
2. P. Baudin, J.-C. Filiâtre, C. Marché, B. Monate, Y. Moy, and V. Prevosto. ACSL: ANSI C specification language. http://frama-c.cea.fr/download/acsl_1.4.pdf, Oct. 2008. Version 1.4.
3. S. Boldo and J.-C. Filiâtre. Formal verification of floating-point programs. In *ARITH18*, Montpellier, France, June 2007. IEEE Computer Society.
4. L. Burdy, Y. Cheon, D. R. Cok, M. D. Ernst, J. R. Kiniry, G. T. Leavens, K. R. M. Leino, and E. Poll. An overview of JML tools and applications. *Int. J. STTT*, 7(3):212–232, June 2005.
5. E. Cohen, M. Dahlweid, M. Hillebrand, D. Leinenbach, M. Moskal, T. Santen, W. Schulte, and S. Tobies. VCC: A practical system for verifying concurrent C. In *TPHOLs 2009*, volume 5674 of *LNCS*. Springer, 2009.
6. U. Frese, D. Hausmann, C. Lüth, H. Täubig, and D. Walter. The importance of being formal. In *SafeCert'08*, ENTCS. Elsevier Science, 2008.
7. S. Haddadin, A. Albu-Schaffer, and G. Hirzinger. Requirements for safe robots: Measurements, analysis and new insights. *Int. J. Robot. Res.*, 28(11–12):1507–1527, 2009.
8. C. Heitmeyer, R. Jeffords, R. Bharadwaj, and M. Archer. RE theory meets software practice: Lessons from the software development trenches. In *RE'07*, pages 265–268. IEEE Computer Society, 2007.
9. C. Hoare. Viewpoint retrospective: an axiomatic basis for computer programming. *Commun. ACM*, 52(10):30–32, 2009.
10. IEC. *IEC 61508 – Functional safety of electrical/electronic/programmable electronic safety-related systems*. IEC, Geneva, Switzerland, 2000.
11. K. M. Krishna, R. Alami, and T. Simeon. Safe proactive plans and their execution. *Robot. Auton. Syst.*, 54(3):244–255, 2006.
12. C. Lüth and D. Walter. Certifiable specification and verification of C programs. In *FM 2009*, volume 5850 of *LNCS*, pages 419–434. Springer, 2009.
13. L. I. Meikle and J. D. Fleuriot. Mechanical theorem proving in computational geometry. In *Automated Deduction in Geometry*, volume 3763 of *LNCS*, pages 1–18. Springer, 2006.
14. T. Nipkow, L. C. Paulson, and M. Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, volume 2283 of *LNCS*. Springer, 2002.
15. J. Peleska. A unified approach to abstract interpretation, formal verification and testing of C/C++ modules. In *ICTAC'08*, volume 5160 of *LNCS*, pages 3–22. Springer, 2008.
16. H. Tuch. Formal verification of C systems code. *J. Autom. Reasoning*, 42(2–4):125–187, Apr. 2009.