Available online at www.sciencedirect.com
ScienceDirect
Electronic Notes in Theoretical Computer Science
ELSEVIER
Electronic Notes in Theoretical Computer Science 238 (2009) 57–70
www.elsevier.com/locate/entcs

The Importance of Being Formal

Udo Frese[1] Daniel Hausmann[1] Christoph Lüth[1,2]
Holger Täubig[1] Dennis Walter[1]

FB 3 — Mathematics and Computer Science, Universität Bremen
Deutsches Forschungszentrum für Künstliche Intelligenz
Bremen, Germany

Abstract

This paper presents work in the context of the certification of a safety component for autonomous service robots, and investigates the potential advantages offered by formally modelling the domain knowledge, specification and implementation in a theorem prover in higher-order logic. This allows safety properties to be stated in an abstract manner close to textbook mathematics. The automatic proof checking alleviates correctness concerns, and provides a seamless development process from high-level safety requirements down to concrete implementation. Moreover, the formalisation can be checked for correctness automatically, and the certification review process can focus on the correctness of the specification and safety cases.

Keywords: software certification, formal methods, robotics, safety, Isabelle

1 Introduction

In this paper, we report on work being done in the context of the SAMS project where a safety component for autonomous mobile service robots is being developed and certified as SIL-3 compliant. The purpose of the safety component is to calculate a safety zone for the moving robot, and stop the robot when an obstacle enters the safety zone, thus protecting both the robot and the obstacle from a collision. The safety properties are formulated at a very abstract level: e.g. that the robot does not collide with any obstacles is formulated in terms of geometry (polygons and sets of points changing over time), which is very different from the actual code. We keep the modelling of the domain mathematically close to what is found in textbooks on geometry and physics (providing the foundations of robotics), and formalise it in higher-order logic in a theorem-prover (in our case, Isabelle [16]). We further

[1] Research supported by BMBF Research Grant SAMS 01 IM F02
[2] Email: christoph.lueth@dfki.de (corresponding author)

doi:10.1016/j.entcs.2009.09.006

model the implementation in the theorem prover as well, achieving a seamless design process with a clear and formally proven relationship between abstract safety properties and implementation. We believe this approach is generally useful in a setting aiming for a certification: The formal modelling and proof makes explicit all hidden assumptions which may have been used in specifications, implementation or proofs, and focuses the certification process on examining the safety specification and the explicit assumptions, as the proofs can be checked automatically.

The paper is structured as follows: Sect. 2 gives an overview of the project. We consider how to specify safety properties in Sect. 3, using a motivating example, and show how to prove them correct with respect to an implementation in Sect. 4. Sect. 5 gives a simplified account of the whole development, demonstrating the seamless development cycle, and Sect. 6 concludes.

2 The SAMS Project

The SAMS project aims at developing and certifying a safety component for autonomous mobile service robots. The component will provide a self-contained, configurable, state-of-the-art collision avoidance service for mobile robots such as automated guided vehicles or mobile service robots. The sensor input will be provided by a safety laser scanner. This device determines the distance to obstacles by sending out a laser pulse and measuring the time until the reflection is received. It scans a planar slice of the environment by rotating the laser beam (similar to a lighthouse) while taking measurements. The scanner itself is certified, so we trust the data it provides.

Industrial solutions for safe collision avoidance are available but suffer from inflexibility. They are usually based on safety laser-scanners with a few preconfigured safety zones stopping the vehicle whenever an obstacle is detected in the safety zone. The disadvantage is the small number of static safety zones. This requires that one safety zone covers many different velocities, and thus necessarily overestimates most of them, leading to over-cautious motion.

Our solution is to compute a safety zone in real-time based on the vehicle's current velocity and angular velocity. It is defined as the area covered by the vehicle until it comes to a full stop plus a safety margin (Fig. 1). When an obstacle is detected in this safety zone, the vehicle is stopped. This way, the vehicle can always travel as fast as safe.

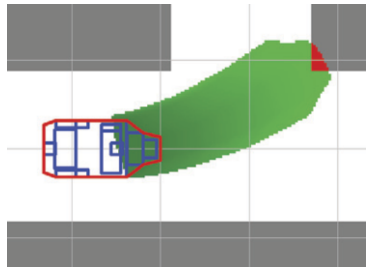

Fig. 1: Safety zone for vehicle taking a slight left turn.

Solutions like this are well-known in the scientific robotics community [12,8,13] but none of these has ever been certified according to norms as required by the machinery directive (2006/42/EC) for use of automated machines in the workplace (and soon for service robots as well). Thus, our objective of certifying a state-of-the-art collision

avoidance algorithm under IEC 61508 (the relevant norm) as a SIL-3 compliant safety component is novel, from the robotics point of view as well as from the certification point of view. Safety certification in an industrial setting typically involves hardly any complex algorithms and norms such as DIN EN ISO 10218 emphasise hardware-based solutions, e.g. physical separation by a cage or a light curtain. What is novel in our project is that the safety function to be certified is based on software and sophisticated algorithms.

3 Specification of Safety Properties

Typical safety requirement place specific constraints on control parameters (e.g. this valve should not be open longer than a specified time) or comprise generally understood high-level requirements (e.g. a concurrent system should not deadlock). In contrast, robotics is a rich and complex domain. It needs geometry and physics to describe the behaviour of the system, and the requirements are formulated in terms of this complex domain. In our case, the main safety requirement is that the robot does not collide with obstacles, which is formulated in terms of geometry and physics, rather remote on a conceptual level from any code. Moreover, the algorithms used to calculate the safety zones are fairly sophisticated, as they are parametric over the braking model of the robot, which at the same time needs to be simple, so it can be established with very few measurements, accurate, so the safety zones are not too large, and conservative, so the safety zones are not too small.

The problem is exacerbated by the fact that robotics has no previous history of formal methods, so there are no set of best practices, established specification languages or well-developed tools specific to the domain available. Thus, the most important question is how to specify safety properties. For specifications to be credible for an external certification authority, they should be formulated as comprehensible as possible and abstractly, not too close to an implementation.

For example, we model the contour of the robot as a polygon, and obstacles as arbitrary connected sets of points. A collision occurs if a point is both inside an obstacle and the polygon. Thus, we have to formalise the property of a point being inside a polygon. Fig. 2 shows an efficient textbook implementation which determines whether a point `p` is inside a convex polygon `poly[0..n-1]`. It can be easily specified, once we define the inside $\mathrm{I}_1(p_1, \ldots, p_n) \subset \mathbb{R}^2$ of a polygon. A straightforward way would be to define it as those points which are left of all edges, using the index i modulo n:

$$\mathrm{isLft}(p; a, b) \Leftrightarrow (p-a) \cdot (b-a)^{\perp} \leq 0, \text{ with } \binom{x}{y}^{\perp} = \binom{-y}{x} \tag{1}$$

$$\mathrm{I}_1(p_1, \ldots, p_n) = \left\{ p \in \mathbb{R}^2 \middle| \forall i : \mathrm{isLft}(p; p_i, p_{i+1}) \right\} \tag{2}$$

From a practical point of view this is a good specification. It requires only elementary arithmetic and corresponds almost directly to the efficient implementation in Fig. 2, making code verification easy. However, since it is so close the same conceptual error could spoil both implementation and specification (e.g. mistakenly comparing for larger than zero). Hence, although proven the code is not as credible

```
bool isLeftOf (Point p, Point a, Point b)
{ return -(p.x-a.x)*(b.y-a.y) + (p.y-a.y)*(b.x-a.x)>=0; }

bool isInside (Point p, Point* poly, int n) {
  int i;
  for (i=0; i<n; i++)
     if (!isLeftOf(p, poly[i], poly[(i+1)%n])) return false;
  return true;
}
```

Fig. 2. A typical C implementation to determine whether a point `p` is inside a convex polygon `poly[0..n-1]`.

as it should be. Furthermore, it is not immediately obvious that (2) defines what one would intuitively call the interior.

By contrast, our philosophy is to avoid these problems by starting from an abstract but mathematically intuitive definition. We first define the polygon outline as the union of all edges

$$\mathrm{L}(p_1,\ldots,p_n) = \bigcup_{i=1}^{n} \left\{ (1-\lambda)p_i + \lambda p_{i+1} \middle| \lambda \in [0\ldots 1] \right\} \tag{3}$$

and following the Jordan curve theorem [1] define the inside of any outline $L \subset \mathbb{R}^2$ as the union of L with all bounded connected-components of $\mathbb{R}^2 - L$.

$$x \sim_A y \Leftrightarrow \exists \phi : [0\ldots 1] \xrightarrow{\text{continuous}} A : \phi(0) = x,\ \phi(1) = y \tag{4}$$

$$\mathrm{CC}(A) = \left\{ \{y \in A | x \sim_A y\} \middle| x \in A \right\} \tag{5}$$

$$\mathrm{bnd}(C) \Leftrightarrow \exists d \in \mathbb{R} : \forall x \in C : |x| \leq d \tag{6}$$

$$\mathrm{I}_2(L) = L \cup \bigcup \{ C \in CC(\mathbb{R}^2 - L) | \, \mathrm{bnd}(C) \} \tag{7}$$

$$\mathrm{I}_2(p_1,\ldots,p_n) = \mathrm{I}_2(\mathrm{L}(p_1,\ldots,p_n)) \tag{8}$$

Intuitively, A is the complement of the outline L, so $x \sim_A y$, iff there is a connection between x and y not crossing the outline (4). $CC(A)$ defines equivalence classes under this relation (5) and I_2 unites those classes not extending to infinity (7). The definition is very general, since it is defined not only for convex polygons but for any polygon and even any outline. And it is very abstract yet intuitive, corresponding well to the established mathematical theory. This definition can be reviewed by an external expert who is not a programmer, thus introducing crucial redundancy into the review process. But it needs substantial theory to be formulated and to be related to an algorithmic formulation such as (2), and is hence more prone to errors unless our proofs are checked *automatically*. Fortunately, tools for this have reached a state where such a project is feasible.

4 Formal Proof and Modelling

By formalising the domain model inside a theorem prover, we can specify safety properties at an abstract level in terms of the domain and not the implementation. The modelling, being close to conventional textbook mathematics, is easier and

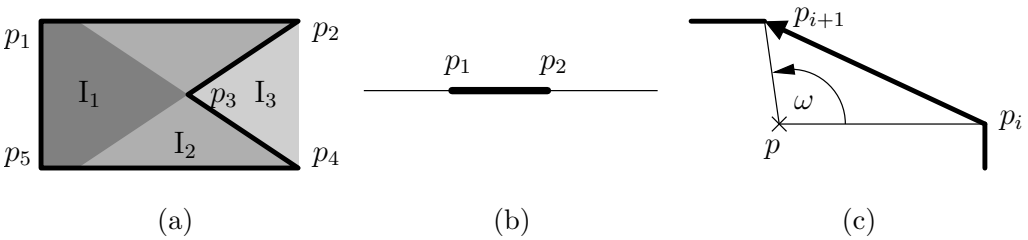

Fig. 3. (a) For a general polygon, the areas defined by I_1, I_2, I_3 form a chain $I_1 \subset I_2 \subset I_3$. For a convex polygon, they are all equal. (b) A degenerate polygon with two points which is excluded by the $\neg$ dgn precondition. (c) The definition of the winding number $\omega(p; p_i, p_{i+1})$ for a single edge. The winding number for the whole polygon is $\in \mathbb{Z}$, for a single edge it is fractional, proportional to the angle.

more readily understood than formalisations in a specification or modelling language such as the UML — an important aspect for the certification process where specifications have to be presented to, and discussed with an independent reviewer (the certification agency). In the context of certification, the formal aspect is particularly valuable, as it alleviates the need to check proofs for correctness — this can be done automatically by Isabelle. All that remains to be checked is that the specifications adequately model the safety properties, and that all assumptions are met. Thus, the formal modelling makes explicit any hidden assumptions that may underlie a specific implementation or specification. For tool support, we have chosen Isabelle, because we know it well; other possible tools are Coq [2] or PVS [17].

Picking up on the two definitions of the interior of a convex polygon above, we will present a proof that the abstract definition (8) is equivalent to the algorithmic definition of (2) at a level suitable to be formalised in Isabelle, showing our approach at work.

4.1 Definition of convexity

Fig. 2 determines whether a point is inside a convex polygon, so to specify it we need a notion of convexity as well. Pragmatically, following (2) one would define a polygon to be convex iff all vertices are to the left of all edges.

$$\mathrm{conv}_1(p_1, \ldots, p_n) \Leftrightarrow \forall i, j : \mathrm{isLft}(p_j; p_i, p_{i+1}) \tag{9}$$

Again, the definition of convexity is close to the implementation, not immediately obvious and may require discussion with the certification authority. Even worse, strictly speaking, convexity is a property of the polygon's inside. With (9), this connection is completely lost, because (2) does only make sense for convex polygons, so it is impossible to argue about polygon that have a non-convex inside. Overall, (2) and (9) make the definitions very technical and the overall verification less credible.

On the other hand, building on (8) it is possible to define convexity of arbitrary sets $C \subset \mathbb{R}^2$ and then call a polygon convex iff its inside is convex.

$$\mathrm{conv}_2(C) \Leftrightarrow \forall x \in C, y \in C, \lambda \in [0 \ldots 1] : (1 - \lambda)x + \lambda y \in C \tag{10}$$

$$\mathrm{conv}_2(p_1, \ldots, p_n) \Leftrightarrow \mathrm{conv}_2(\mathrm{I}_2(p_1, \ldots, p_n)) \tag{11}$$

Before we go into the details of the proof, we add three more notions important for verifying routines operating on convex polygons. First, we technically need the notion of a degenerate polygon where all vertices are collinear (Fig. 3b).

$$\mathrm{dgn}(p_1,\ldots,p_n) \Leftrightarrow \forall i,j : (p_j - p_i)\cdot(p_{i+1}-p_i)^{\perp} = 0 \tag{12}$$

Second, the convex hull of a set $A \subset \mathbb{R}^2$ is the smallest convex superset

$$\mathrm{CH}(A) = \bigcap\{B \mid A \subset B \subset \mathbb{R}^2,\ \mathrm{conv}_2(B)\}. \tag{13}$$

With this notation a third way of defining the inside of a convex polygon is possible, namely as the convex hull of all vertices.

$$\mathrm{I}_3(p_1,\ldots,p_n) = \mathrm{CH}(\{p_1,\ldots,p_n\}) \tag{14}$$

4.2 Proving the equivalence to the algorithmic definitions

We now have an easily comprehensible, abstract definition from which we want to derive the more algorithmic definitions as lemmas, namely overall

$$\mathrm{conv}_1 \wedge \neg\,\mathrm{dgn} \Rightarrow \mathrm{conv}_2 \wedge \mathrm{I}_1 = \mathrm{I}_2 = \mathrm{I}_3 \tag{15}$$

where for brevity we omit the argument $(p_1,\ldots,p_n)$. We assume $\neg\,\mathrm{dgn}$ and proceed by showing $\mathrm{I}_1 \subset I_2 \subset \mathrm{I}_3$ and $\mathrm{I}_3 \subset \mathrm{I}_1$, if conv_1 holds (Fig. 3a).

The first inclusion $\mathrm{I}_1 \subset \mathrm{I}_2$ is the most difficult to prove. We adapt an argument of [15, §7]. Let $p \in \mathbb{R}^2 - \mathrm{L}$ be a point. We consider the *winding number*, i.e. the number of times the polygon "winds around" p and observe

(i) The winding number is constant in each connected component of $\mathbb{R}^2 - \mathrm{L}$.

(ii) The winding number is 0 for p with sufficiently high $|p|$.

(iii) The winding number is ≥ 1 for all $p \in \mathrm{I}_1 - \mathrm{L}$, if $\neg\,\mathrm{dgn}$.

From these three properties it follows that all $p \in \mathrm{I}_1$ lie in a bounded connected component of $\mathbb{R}^2 - \mathrm{L}$ and hence in I_2. In detail, we define the winding number of a single edge as the signed angle that edge has with p (Fig. 3b) divided by 2π. The winding number of a polygon is the sum for all edges.

$$\omega(p;a,b) = \frac{1}{2\pi}\,\mathrm{atan2}\left((b-p)\cdot(a-p)^{\perp}, (b-p)\cdot(a-p)\right) \tag{16}$$

$$\omega(p) = \omega(p;p_1,\ldots,p_n) = \sum_{i=1}^{n}\omega(p;p_i,p_{i+1}) \tag{17}$$

We now show (i), (ii), (iii).

(i) Each $\omega(p;p_i,p_{i+1})$ is continuous in p if p is not on the edge p_ip_{i+1}, since atan2 is continuous except on $\{0\}\times\mathbb{R}_0^-$. Hence $\omega(p)$ is continuous on $\mathbb{R}^2 - \mathrm{L}$. Since the polygon is a loop of edges, $\omega(p) \in \mathbb{Z}$, justifying the name winding *number*. So

$\omega(\phi(\lambda))$ is constant on any path ϕ in $\mathbb{R}^2 - \mathrm{L}$ and hence constant on every connected component of $\mathbb{R}^2 - \mathrm{L}$.

(ii) It is $|\omega(p;a,b)| \leq \frac{1}{2\pi} \operatorname{asin}\left(\frac{|b-a|}{|p-a|}\right) \leq \frac{|b-a|}{4|p-a|}$. So, for fixed $a, b \in \mathbb{R}^2$ and $|p| \to \infty$, $\omega(p;a,b) \to 0$. Since $\omega(p) \in \mathbb{Z}$, $\omega(p) = 0$ for sufficiently large $|p|$.

(iii) The condition $\mathrm{isLft}(p;a,b)$ implies $\omega(p;a,b) \geq 0$ since p is not on the edge ab. So $\omega(p;p_i,p_{i+1}) \geq 0$ for all i and $\omega(p) \geq 0$. Could all $\omega(p;p_i,p_{i+1})$ be 0? Then p would be on the infinite line through $p_i p_{i+1}$ for all i. But then the lines $p_i p_{i+1}$ and $p_{i+1}p_{i+2}$ would have two distinct points in common, namely p_{i+1} and p. So they were identical and by induction all lines would be identical contradicting $\neg\,\mathrm{dgn}$. So, at least one $\omega(p;p_i,p_{i+1}) > 0$ and so is $\omega(p) > 0$.

For $\mathrm{I}_2 \subset \mathrm{I}_3$ imagine $p \in \mathrm{I}_2$ and a line through p. This line must leave I_2, thus intersecting L at one point on each side of p, because otherwise I_2 would be unbounded. Now $\mathrm{L} \subset \mathrm{I}_3$, since $\mathrm{L} = \bigcup_{i=1}^{n} \mathrm{CH}(\{p_i, p_{i+1}\})$. As p lies on a line between points in I_3 and I_3 is convex, it is in I_3 itself.

For $\mathrm{conv}_1 \Rightarrow \mathrm{I}_3 \subset \mathrm{I}_1$, assume conv_1 holds and consider one polygon edge $p_i p_{i+1}$. By (9) all p_j are on the left of this edge. The set of points left of an edge is convex. So the convex hull of $\{p_1, \ldots, p_n\}$, i.e. I_3, is completely left of that edge too. Since this holds for all edges, $\mathrm{I}_3 \subset \mathrm{I}_1$. So, $\mathrm{I}_1 = \mathrm{I}_2 = \mathrm{I}_3$ and since I_3 is by definition convex, conv_2 holds.

Some final remarks: conv_1 and conv_2 are not equivalent for technical reasons, since conv_1 requires a counter-clockwise ordering of the polygon. The degenerate polygons, for which dgn holds are also well handled by I_2 and I_3 but must be excluded, because I_1 defines an infinite line then. Overall, for non-degenerated convex polygons, $(\mathrm{conv}_1 \wedge \neg\,\mathrm{dgn})$, the definitions of "inside" are equivalent ($\mathrm{I}_1 = \mathrm{I}_2 = \mathrm{I}_3$) and the inside is convex (conv_2).

4.3 Formal proof: is it worth it?

The formalisation of the above pen-and-pencil proof is subject to ongoing work. We have formulated the necessary definitions ($\mathrm{I}_1, \mathrm{I}_2, \mathrm{I}_3, \mathrm{conv}_1$ and conv_2) in Isabelle. The main inclusion theorems have been proven using several intermediate lemmas; however, some of the more technical lemmas (mainly concerned with non-linear arithmetics) still need to be proven formally, so the proof is still incomplete. [3]

As pointed out above, once the proof is fully formalised, there is no need to discuss it any further with the certification authority, and reviews can focus on the specification. Further, we claim this approach increases the credibility of the specification. When using (2) as a definition, there are potential errors in the specification of `isInside` (Fig. 2) that would not be found in the whole software verification process. Imagine we had forgotten to specify conv_1 as a precondition for `isInside`. Using I_2, this error is found since the implementation does not comply with the specification. However, if I_1 is used, `isInside` still complies with the specification, since both are identical. The error would not have been uncovered

[3] The current version of the proof scripts is available at http://www.dfki.de/sks/sams/download/safecert08.tgz.

during verification, because if I_1 is used in the overall specification the program is still formally correct. So if a user entered a non-convex robot contour such as in Fig. 3a, the system would implicitly assume a smaller robot, namely the dark area in Fig. 3a. This is of course a critical safety fault.

A second error was actually uncovered by the formalisation: we had not realized that $\neg$ dgn is needed as a precondition. Without it, a two point polygon (a, b) would be interpreted as an infinite line by $I_1(a, b)$ (Fig. 3b). This is also critical, e.g. if the polygon represents free space.

Therefore, formalising proofs for well-known theorems may actually help in the certification. In mathematical textbooks, there are often contextual or tacit assumptions, and definitions may vary subtly from textbook to textbook. This may lead to special cases, such as dgn above, being overlooked. However, if the proof is formalised in Isabelle, all preconditions and assumptions are made explicit, eliminating this potential source of errors.

5 A Safety Case for a Simplified Robot

We will now demonstrate the seamless transition from a high-level, formal specification of safety properties to low-level correctness assertions of the software on a control system. The scenario we consider in this paper is a simplified version of the one used in the SAMS project.

5.1 Modelling the world

Our goal is to prove that a robot equipped with our obstacle avoidance control system does not collide with any objects in its environment. This safety requirement should be formulated at a high level without any reference to internals of the robot, such as the software running on its control system. We therefore create a formal model of the robot's environment (the *world*) and state the required safety assertions therein. Since the robot's sensors (odometry and laser scanner) only provide two-dimensional information, the world is modelled as the real plane $\mathbb{R}^2$. Objects in the world are simply connected sets of points. In this simplified world, lines denote walls or other boundaries, while circles denote all other obstacles. The robot itself is modelled as a single point with an orientation. Because objects, including the robot, may move or change their shape over time, they are modelled as a function over time of type $\mathbb{R}_0^+ \to \alpha$ (with $\mathbb{R}_0^+$ interpreted as time and α any type), called *behaviours*. E. g., $robot : \mathbb{R}_0^+ \to \mathbb{R}^2$ yields the robot's position for each point in time. Appropriate specifications of these functions restrict their behaviour. Higher-order logic is particularly well suited for specifying properties of functions; e. g. we can define a predicate $is\_cont : (\mathbb{R}_0^+ \to \mathbb{R}^2) \to \textbf{bool}$ that characterises exactly the continuous functions. The specifications of functions representing objects can roughly be separated into two parts: (a) properties that generally hold, like continuous movement (we exclude teleportation: $is\_cont(robot)$), finite size, or connectedness (objects with 'gaps' are naturally interpreted as distinct objects), and (b) object-specific properties, like maximum velocity of the robot ($\forall t.\ |D(robot)(t)| < v_{max}$,

where D is the differential operator), its braking deceleration or reaction times.

The main safety requirement is very easy: The robot shall never touch any of the surrounding objects. Since the control software applied in this case can only guarantee this requirement w.r.t. a static environment in which objects are not moving, we add a premiss fixing the world to be static. A preliminary formalisation of the safety requirement looks thus

$$\left[\forall t,\, t'.\ \forall o \in \mathit{Obs}.\ o(t) = o(t')\right] \Longrightarrow \forall t.\ \forall o \in \mathit{Obs}.\ \mathit{robot}(t) \notin o(t) \tag{18}$$

where *Obs* is the set of all functions representing objects in the world, each of which is of type $\mathbb{R}_0^+ \to \mathcal{P}(\mathbb{R}^2)$.

5.2 Tying together world and control system

Obviously, general physical specifications about *robot*, like continuous movement and adherence to some maximum velocity, do not suffice to prove statement (18). We have to express the fact that the robot is influenced by its control software, which in our case is a reactive system. This means that a safety routine is called in a cyclic fashion (every N milliseconds), receiving input from the robot's sensors and producing a simple output which indicates whether to initiate an emergency stop or to continue driving. The cycle length is in this case given by the scan frequency of the laser scanner. In each cycle, fresh data from the scanner and the odometry form the basis of the routine's output. This is modelled by introducing *discrete behaviours* of type $\mathbb{N} \to \alpha$, which are functions that yield a value at discrete points in time, i. e. at each cycle. The translation from cycle number to point in time is given by $d2c : \mathbb{N} \to \mathbb{R}_0^+$. We distinguish between discrete *input* and *output* behaviours: input behaviours model data that is available at each run of the safety routine, and depend on the state of the real world model; output behaviours represent a program and its output, and may depend on the real world *only* through input behaviours. This models the restriction that the only information about the world available to a program is through its discrete inputs.

The most important discrete behaviours of our example are:

- An input behaviour $\mathit{scanner} : \mathbb{N} \to \mathit{List}(\mathbb{N})$ modelling the distance measurements of the scanner
- An input behaviour $\mathit{odo} : \mathbb{N} \to \mathbb{Z}^2$ yielding the discrete (angular and translational) velocity approximations by the odometry.
- An output behaviour $\mathit{stop} : \mathbb{N} \to \mathbf{bool}$ representing the control software.

Specifications relating to the discrete behaviour *scanner* describe properties of the employed laser scanner. From this basic input behaviour we derive a behaviour that yields a list of points in the world instead of distance measurements; this is a simple geometric transformation requiring the robot's current position and orientation as well as the measured distances:

```
scanner_pts :: nat -> List(real)
scanner_pts n =  let t = d2c(n)
 in dist_to_pts(scanner(n), robot(t), orientation(t))
```

Given this definition, a simple way to model scanner precision is by saying that for each point yielded by *scanner_pts*, there must be an object in the vicinity (here: δ length units) that caused this measurement:

$$\forall n.\ \forall p \in \mathit{scanner\_pts}(n).\ \exists o \in \mathit{Obs}.\ |o(\mathit{d2c}(n)) - p| < \delta$$

Even more important is the assertion that for each measured distance d there are no obstacles inside the area covered by the laser beam up to d. This area takes the shape of an acute-angled cone, as the beam slightly diverges over distance.

$$\forall n. \forall p \in \mathit{scanner\_pts}(n).\ \forall o \in \mathit{Obs}.\ \mathit{cone}(p, \mathit{robot}(\mathit{d2c}(n))) \cap o(\mathit{d2c}(n)) = \emptyset$$

This example also shows how discrete input and continuous behaviours are linked: cycle n happens at time $\mathit{d2c}(n)$, and the output of *scanner_pts* (and thereby the output of *scanner*) depends on the current position of the obstacles in the world. Hence, the input to the control software is no longer arbitrary, but in correspondence with the environment, modulo scanner imprecision. Similar specifications must be given for the odometry.

For reasons of efficiency and tool support, the safety routine is implemented in C using the MISRA guidelines [14]. Roughly, the safety routine has to ensure that the robot may continue driving only if the sensors' data indicate that in no possible case an emergency stop will be necessary until the next cycle of the program. In particular, it must incorporate all possible latencies of the concrete system. If the routine cannot give this guarantee, it must halt the robot. This is a very general (and standard) technique for safeguarding with reactive systems.

The following statement expresses the fact that the *stop* behaviour which represents the control software actually restricts the possible motion of the robot. When an emergency stop is signalled ($\mathit{stop}(n) = \mathit{True}$), then after a certain reaction time T_{react} the robot will decelerate with at least the specified minimum amount of a_{brk}.

$$\begin{aligned} \forall n\ t.\ \mathit{stop}(n) \wedge \mathit{d2c}(n) + T_{react} \leq t \wedge 0 < |v_{robot}(t)| \\ \Longrightarrow a_{robot}(t) \leq a_{brk} < 0 \end{aligned} \tag{19}$$

5.3 Tying together control program and its implementation

To create a formal connection between the *stop* function representing the control software and the aforementioned C program implementing it, we formalised the C semantics in the logic. A type S models the state of C programs (values of global and local variables), data types represent C programs and expressions, and a semantic function *sem_c* maps program terms to state transformers $st : S \to \alpha \times S$, where α stands for the possible return value of the expression or program. In particular, a state transformer $sr : S \to \mathbf{bool} \times S$ represents the implemented safety routine. To capture the fact that this routine is called in each cycle, we define an embedding function that turns a state transformer into a discrete output behaviour. The definition is shown in Fig. 4; it is parametric over the state transformer *phi*, so that *embed_st*(*sr*) yields the desired output behaviour. *stop* is then simply defined

```
embed_st :: (S -> (Bool, S)) -> nat -> (Bool, S)
embed_st phi 0 = (False, initial_state)
embed_st phi n = let (pre_b, pre_s) = embed_st phi (n-1)
                 in phi (update (pre_s, scanner(n), odo(n)))
```

Fig. 4. Turning a state transformer `phi` into a discrete behaviour.

```
/*@ requires \valid_ptr(v, 0, len) &&
  @ ensures (\forall int i; 0 <= i && i < len --> v[i] <= \result) &&
  @         (ctr == \old(ctr) + 1)  */
  int upper_bound(int *v, int len);
```

Fig. 5. Simple example specification of a C function. The returned value is specified to be larger than any element of the given array `v`.

as its first projection. In cycle 0, no emergency stop is signalled; *initial_state* corresponds to the state the safety routine is started in. In all other cycles, the input behaviours are evaluated for the current cycle, and the state that resulted from the last execution is updated with the new inputs: $update(pre\_s, si, oi)$; this new state is then fed into the safety routine state transformer, its value yielding the behaviour's new state. The Boolean output is also determined by the state transformer.

Discrete output behaviours as defined here are reminiscent of functions in the synchronous programming language Lustre [10,4]. We have not investigated the connection deeply, but we require similar restrictions on what an allowed output behaviour is. For example, to model memory constraints an output behaviour may only use a finite number of input behaviour values (say, from the last N cycles) in its definition. Moreover, for causality reasons an output behaviour may only depend on past and present inputs.

5.4 Specification of C Programs

To state and prove properties of C functions in our framework, we use a specification language that annotates the actual code, rather than specifying the resulting state transformer afterwards. This way, specification and code are developed side by side. We follow existing approaches like JML [3] or Caduceus [7]. Fig. 5 shows a simple example with pre- and post-conditions attached to a function (*ensures* and *requires* clause, respectively). The pre-/post-conditions are specified in first-order logic, where a formula's free variables are identified with program variables. So in the ensures clause of Fig. 5, `v` is interpreted as `upper_bound`'s parameter, while `ctr` is assumed to be a global variable. Operator `\old` in the postconditions refers to the value of a variable in the state before the function was executed.

Similar to the code, these specifications are also translated into the logic, where their semantics is given as *state predicates*, i. e. functions $pre : S \to \mathbf{bool}$ for pre-conditions and $post : (S \times S) \to \mathbf{bool}$ for post-conditions, which are in fact predicates about both the initial and the final state due to the presence of the `\old` predicate. Very classically, a function is then considered to satisfy its specification if, when run in a state that satisfies the pre-condition, the function's interpretation

as a state transformer maps that state to one that satisfies the post-condition. A weakest-precondition style calculus [5,18] allows us to conveniently prove programs correct.

Once we have proved that all functions of the safety routine satisfy their specifications, we can use this fact to prove properties of the behaviour obtained by applying *embed_st* to the safety routine's state transformer. In particular, we can prove that this output behaviour yields *True* when the safety routine determines an emergency stop is necessary. Together with the assertion (19) that the robot will actually halt when the control software signals an emergency stop, we can then prove the overall safety assertion, namely that the robot does not collide with its environment.

5.5 Summary

The simplified scenario demonstrated how it is possible to tie together the high-level safety requirements of a whole system, and the low-level specification of control system inputs and the software running on that control system. All this is being done within the Isabelle prover, avoiding large gaps that in most other approaches would have to be overcome by a leap of faith, even though in the presentation above we had to omit many details.

However, we also made some simplifying assumptions. Firstly, we do not have a notion of computation time within the logic. This is not a huge oversimplification, because the only timing requirement is that every run of the control program must finish execution in one cycle and this must be taken into account as a latency. Secondly, the system hardware is idealised. We do not take random or systematic hardware failures into account. This is in accordance with the certification, which in our project focuses on software and assumes that hardware faults can be dealt with in a standard manner.

6 Conclusions

This paper has presented the approach to software certification as employed by the SAMS project. Our methodology is summed up as follows:

- Safety properties are stated as abstract, and as early in the development process as possible, using a formalisation of concepts from the foundations of robotics (geometry and physics);
- all proofs are done formally in Isabelle;
- the implementation is done in C and modelled in Isabelle, allowing a seamless formal development from the abstract specification down to the code.

The approach has three main advantages: because the specification is stated very abstractly, with an emphasis on mathematical concepts rather than computation, its review can focus on the main aspects, and uncover conceptual errors which would have gone unnoticed otherwise (see Sect. 4.3). Because the actual proofs are fully

formalised, they can be checked by machine and do not need to be discussed or reviewed further; the review process can concentrate on the specifications. Also, during the course of the formal proof, hidden assumptions in the specification are uncovered which can then be made explicit.

There is of course a huge body of related work on formal methods using theorem proving. Not many of these address certification; an exception is the recent certification of Gemalto's implementation of Java Card technology according to the Common Criteria (CC) [9], using the Coq proof assistant. The SCADE suite [6] can generate code from models, and is certified for use with all SILs, but does not use a theorem prover. In robotics, the KoSePro project [11] explicitly addresses certified safety, but in the more traditional vein adumbrated in Sect. 2. In summary, the combination of certification, formal proof and robotics in SAMS is quite unique.

The approach is not without drawbacks, of course. Firstly, formal proof is hard, and in particular the automatic proof support for real-valued arithmetic statements often occuring in robotics is not good as one might hope for. Hence, a formalisation on the scale undertaken here is not industrially viable yet; however, once concepts have been formalised and proven they can be reused for other projects in this area (e. g. the domain modelling of Sec. 4). Further, the concept of formal proof is alien to most industrial certification agencies. For the SAMS project, we are currently in discussions with a German certification authority, TÜV Süd; once this methodology has been established, others will be able to benefit. The certification with the TÜV is explicitly focusing on the software, with the aim of establishing our project as a prototypical certification.

Presently, the SAMS project is in the second of its three years. Our software is currently about 13.5 kloc of C code, and the proof scripts amount to 4.5 kloc of Isabelle proof script for the formalisation of the implementation model, and 1.5 kloc for the domain model.

For future work, we plan to certify the correctness of an algorithm which detects collisions in three dimensions, which has been developed recently, in extensions of the two-dimensional case presented in this paper.

References

[1] Alexander, J., *A proof of Jordan's theorem about a simple closed curve*, Annals of Mathematics **21** (1921), pp. 180–184.

[2] Bertot, Y. and P. Castéran, "Interactive Theorem Proving and Program Development: Coq'Art: The Calculus of Inductive Constructions," EATCS Texts in Theoretical Computer Science, Springer, 2004.

[3] Burdy, L., Y. Cheon, D. Cok, M. Ernst, J. Kiniry, G. T. Leavens, K. R. M. Leino and E. Poll, *An overview of JML tools and applications*, International Journal on Software Tools for Technology Transfer **7** (2005), pp. 212– 232.

[4] Caspi, P., D. Pilaud, N. Halbwachs and J. Plaice, *Lustre: a declarative language for programming synchronous systems*, in: *14th ACM Conf. on Principles of Programming Languages* (1987).

[5] Dijkstra, E. W., *Guarded commands, nondeterminacy and formal derivation of programs*, Commun. ACM **18** (1975), pp. 453–457.

[6] Esterel Technologies, *The SCADE suite*, Web site at `http://www.esterel-technologies.co/`.

[7] Filliâtre, J.-C. and C. Marché, *Multi-Prover Verification of C Programs*, in: *Sixth International Conference on Formal Engineering Methods (ICFEM)*, Lecture Notes in Computer Science **3308** (2004), pp. 15–29.

[8] Fox, D., W. Burgard and S. Thrun, *The dynamic window approach to collision avoidance*, IEEE Robotics and Automation Magazine **4** (1997), pp. 23–33.

[9] Gemalto N.V., *Gemalto achieves major breakthrough in security technology with JavaCard highest level of certification*, Press release at http://www.gemalto.com/php/pr_view.php?id=239.

[10] Halbwachs, N., P. Caspi, P. Raymond and D. Pilaud, *The synchronous data flow programming language LUSTRE*, Proceedings of the IEEE **79**, 1991, pp. 1305–1320.

[11] *Wirtschaftliche, adaptronische und sichere Schlüsselkomponenten für die Servicerobotik in der Produktion*, Web site at http://www.kosepro.de.

[12] Lankenau, A. and T. Röfer, *A safe and versatile mobility assistant*, Reinventing the Wheelchair. IEEE Robotics and Automation Magazine **8** (2001), pp. 29 – 37.

[13] Minguez, J. and L. Montano, *Nearness diagram (ND) navigation: Collision avoidance in troublesome scenarios*, IEEE Transactions on Robotics and Automation **20** (2004), pp. 45–59.

[14] *MISRA-C: 2004. Guidelines for the use of the C language in critical systems.* (2004).

[15] Needham, T., "Visual Complex Analysis," Oxford University Press, 1998.

[16] Nipkow, T., L. C. Paulson and M. Wenzel, "Isabelle/HOL — A Proof Assistant for Higher-Order Logic," Lecture Notes in Computer Science **2283**, Springer, 2002.

[17] Owre, S., J. M. Rushby, and N. Shankar, *PVS: A prototype verification system*, in: D. Kapur, editor, *11th International Conference on Automated Deduction (CADE)*, Lecture Notes in Artificial Intelligence **607** (1992), pp. 748–752.

[18] Reynolds, J. C., "Theories of Programming Languages," Cambridge University Press, Cambridge, England, 1998, 149–154 pp.