Geometrical Specification using Dependent Types in PVS

Mauricio Ayala-Rincón,* Valnei A. Fernandes and André L. Galdino

[1]Grupo de Teoria da Computação, Departamento de Matemática
Universidade de Brasília, 70910-900 Brasília D.F., Brasil

{ayala,galdino,valnei}@mat.unb.br

Abstract. *We point out the importance of the use of the refined dependent types available in the proof assistant* PVS *for the correct specification of a geometrical approach to solve the problem of air traffic conflict detection and resolution (between two aircrafts in mid air). A correct* PVS *specification which completely solves this problem giving a triplet which consists of* $\langle$*escape trajectory, recovery trajectory, switching time*$\rangle$ *and whose first component allows exclusively changes in a unique of the parameters ground speed, heading and vertical speed is partially extended in such a way that the formulated escape maneuvers may combine changes in two of these parameters: ground speed and heading.*

Keywords: *Type Theory, Dependent Types,* PVS, *Geometric Specification, Air Traffic Conflict Detection and Resolution.*

1. Introduction

Dependent types have been shown of great usefulness in programming languages for detecting errors of programs by type checking even during compilation. In fact dependent types can express properties such as "arrays having the same length", "lists of empty length", etc. allowing for the detection of errors such as computing the dot product of vectors of different size, selecting the first element of an empty list, etc. In the context of algebraic and geometrical correct specification and verification, which is central for the applications presented in this paper, also dependent types play a relevant role. In fact, non trivial algebraic properties which are relevant for modeling real-time systems in proof assistants can be encapsulated as type dependencies within the specification of complex systems: for instance, the function "roots of quadratic equations" can be defined as a total function in the sort of triplets of reals (a, b, c coefficients of the quadratic equation $ax^2 + bx + c = 0$) with a non negative discriminant. Programming language prototypes with dependent types include DML which is a refinement of the type system of ML [Xi and Pfenning 1999, Xi 2001] and Cayenne in which language expressions can be used for indexing types losing decidability of type-checking, but enlarging the class of typable programs [Augustsson 1999]. Also, dependent types are the basis of logical frameworks such as CLF [Watkins et al.] as well as of important proof assistant environments [Barendregt and Geuvers 2001] from which we remark PVS [Owre et al. 1998], that is the proof assistant used in our experiments.

We show how the power of dependent types in PVS is applied in the specification and proof of correctness of geometrical solutions to the problem of air traffic conflict detection and resolution for two aircrafts in mid air. We start from the systems KB3D and its successor RR3D[Geser et al. 2002, Maddalon et al. 2004], developed by the Formal Methods Group at

*Corresponding author.

the National Institute of Aerospace, in which solutions for the reference aircraft are computed relatively to the intruder aircraft as triplets $\langle \vec{v'}, t', \vec{v''} \rangle$ whose components are the escape velocity $\vec{v}$ and a switching time t' to the recovery velocity $\vec{v''}$. These triplets give "good" escape and recovery maneuvers to avoid a detected conflict which is understood as the predicted intrusion in the protected zone of the intruder aircraft whenever the original trajectory of the reference aircraft remains unchanged. In [Geser et al. 2002, Maddalon et al. 2004] solutions are restricted in such a way that a unique of the parameters ground speed, vertical speed and heading can be modified to obtain the escape velocity. This restriction is interesting from the technical point of view, because automatic and manual control can be simplified when a sole of these parameters has to be changed and also from the computational point of view because the resulting equational systems are relatively easy to be efficiently solved, giving rise to solutions easily to be implemented even in hardware. Obviously, optimal solutions are not obtained under this restriction: escape trajectories which require minimal effort combine changes in these three parameters.

We extend the solutions provided by RR3D partially specifying and proving correctness in PVS of escape velocities which can combine two of these parameters: ground speed and heading. This is done (in four specific situations) maintaining the simplicity of the equational systems of the specification. Escape velocities computed in this way define escape trajectories closer to the optimal one(s).

In the following sections we give the necessary notions on PVS making emphasis in the use of dependent types. Then in the third section we briefly describe the solutions to the conflict detection and resolution problems computed in RR3D. For these two sections we use notations and examples from [Rushby et al. 1998] and [Geser et al. 2002, Maddalon et al. 2004]. Then, in section four, we present our specification and proofs of the specification of less restricted escape trajectories than the ones computed in RR3D. This is done for a sole case presenting firstly its analytical solution and then briefly (and partially) describing its proof in PVS. Then, we conclude. Additionally, for the benefit of the review process, we have included two appendices containing details of our proofs in PVS which can be dropped in a final version of the paper.

2. Background

Type-checking is relevant in specification environments because it helps to discard wrong and inconsistent specifications. Dependent typing is an extension of the simple theory of types [Church 1940] which admits dependence between types expressed as conditions over a logical theory independent of the theory of types. This makes type-checking powerful (but undecidable in general). For example, the type description of the function min over naturals can be given as

$$min(s: \text{ set of nat}): \text{ nat} \in s$$

Notice that the type of the range of the function *depends* on the value of its domain. In this way, dependent types are used (to try) to avoid erroneous specifications where min is applied to an empty set, for example. In PVS type-checking makes part of the theorem proving framework, which makes it possible to prove type conditions that are central for correctness proofs of the specifications.

Predicate subtypes express mathematical ideas such as naturals are contained in integers, constraints which determine whether a number is even, odd, prime, etc. Predicate subtypes generate a series of *Type Correctness Conditions*, called TCC in PVS, whose demonstrations depend on the extra logic embedded in these predicates instead on conventional type-checking.

A TCC or *proof obligation* for min, as previously specified, is the predicate asserting the existence of a total function having the given signature, expressed as

$$\exists(f : (s : \text{ set of nat }) \rightarrow \text{ nat} \in s) : True$$

This TCC is false, because it is impossible to select an element in the range when s is empty. Totality of functions provides well-known advantages in theorem proving simplifying the manipulation of abstract considerations over non interpreted functions. Detecting this problem, allows for a reformulation of the description of min as

$$min(s : \text{ nonempty set of nat}) : \text{ nat} \in s$$

for which the proof obligation is given as

$$\exists(f : (s : \text{ nonempty set of nat }) \rightarrow \text{ nat} \in s) : True$$

This TCC can be discharged in PVS via a built-in choice function for nonempty sets for instantiating f. The proof of this TCC in PVS looks like

```
min_TCC1 :

  |-------
{1}   EXISTS (f: [s: (nonempty?[nat]) -> (s)]): TRUE

Rule: (inst + "lambda (s:(nonempty?[nat])): choose({x:(s)|true})")
Instantiating the top quantifier in + with the terms:
 lambda (s:(nonempty?[nat])): choose({x:(s)|true}),
this simplifies to:
min_TCC1 (TCC):

  |-------
{1}   FORALL (s: (nonempty?[nat])): nonempty?[(s)]({x: (s) | TRUE})

Rule: (grind)
member rewrites member(x_1, {x: (s) | TRUE})  to TRUE
empty? rewrites empty?({x: (s) | TRUE})  to FORALL (x_1: (s)): FALSE
nonempty? rewrites nonempty?[(s)]({x: (s) | TRUE})
                                    to NOT (FORALL (x_1: (s)): FALSE)
member rewrites member(x, s!1)  to s!1(x)
empty? rewrites empty?(s!1)  to FORALL (x: nat): NOT s!1(x)
nonempty? rewrites nonempty?[nat](s!1)
                                to NOT (FORALL (x: nat): NOT s!1(x))
Trying repeated skolemization, instantiation, and if-lifting,
Q.E.D.
```

Observe that after the instantiation via the choice function the proof of the original existential TCC reduces to the proof of a universal quantified TCC: any nonempty set includes one element.

Also notice that the proof of this TCC is worthwhile for guaranteeing the correctness of the specification of the function min which states that the minimum is a member of the given (nonempty) set of naturals that is not larger than any other member of the set. This simplifies our specification of min to its description of types jointly with the following axiom:

$$\texttt{basic\_axiom\_for\_min} \;:\; \texttt{AXIOM}\ \forall(s:\ \text{nonempty set of nat})\forall(n:\ \text{nat} \in s) : min(s) \leq n$$

When we define the typing rules for dependent types we need to combine dependent typing with predicate subtyping. This combination is very important because it captures the relationship between the outputs and the inputs of functions, making simpler specification avoiding partial functions. For instance, the specification of min uses dependent types because its range are natural numbers which correspondingly belongs to its domain sets of naturals.

With this refined typing, relevant algebraic properties, useful in our geometric modeling, such as whether roots of quadratic polynomials are reals can be specified by total functions. In fact, let ax^2+bx+c and $root(a, b, c)$ denote a polynomial and its real root(s). The key dependence between the coefficients (types of the domain of $root$), which guarantees the existence of real solutions, is to have a non negative discriminant. This can be specified in PVS (namely, as it is done in RR3D) as

```
discr(a:nonzero_real,b:real,c:real) : real = sq(b) - 4*a*c

root(a:nonzero_real,b:real,c:real | discr(a,b,c)>=0,eps:Sign):real
=    (-b + eps*sqrt(discr(a,b,c)))/(2*a)
```

The fourth parameter of `root`, `eps` of type `Sign` and which ranges over $\{1, -1\}$, discriminates between the two possible respective real roots:

$$\frac{-b+\sqrt{b^2-4ac}}{2a} \text{ and } \frac{-b-\sqrt{b^2-4ac}}{2a}$$

Notice that the description of types of $root$ includes subtyping: a is a non null real as well as dependent typing condition for the non negative discriminant: `discr(a,b,c)>=0`. It follows the proof of the corresponding TCC in PVS.

```
root_TCC1 :

  |-------
{1}   FORALL (a: nonzero_real, b: real, c: real | discr(a, b, c) >= 0):
        discr(a, b, c) >= 0

Rerunning step: (subtype-tcc)
sq rewrites sq(b)  to b * b
discr rewrites discr(a, b, c)  to b * b - 4 * (a * c)
sq rewrites sq(b!1)  to b!1 * b!1
discr rewrites discr(a!1, b!1, c!1)  to b!1 * b!1 - 4 * (a!1 * c!1)
Trying repeated skolemization, instantiation, and if-lifting,
Q.E.D.
```

Totality of $root$ will simplify subsequent specifications and proofs of properties such as the unicity of solutions specified by the lemma quadratic_eq_0 as it is done in RR3D.

```
quadratic_eq_0 : LEMMA (a*sq(x) + b*x + c = 0 IFF
    discr(a,b,c) >= 0 AND (x = root(a,b,c,1) OR x = root(a,b,c,-1)))
```

Typing rules in PVS include the rules of the simple type theory and rules for adding subtypes as well as rules for dependent types. The possibility of having arbitrary type predicates for dependent types makes type checking undecidable. Using notations consistent with the ones in [Owre and Shankar 1997] we illustrate application of the typing rule for dependent product types in deciding that the type of root

$$[[\mathtt{x}:[\mathtt{nonzero\_real},\mathtt{real}],\{\mathtt{c}:\mathtt{real}|\mathtt{discr}(\mathtt{p1}(\mathtt{x}),\mathtt{p2}(\mathtt{x}),\mathtt{c})>=0\}],\mathtt{Sign}]\longrightarrow\mathtt{real}$$

is a well formed type. In this notation [_,_], x:A, pi, $\{$ x:A | Cond$\}$ denote respectively a product type, a type variable x of type A (which is used for expressing dependence predicates), the i^{th} projection of a product type ($i = 1, 2$) and a dependent type whose predicate Cond is expressed in terms of previous variables of type. Observe that $\{$c: real | discr(p1(x), p2(x), c) >= 0$\}$ is a dependent type whose predicate depends on the type variable x. According with notations in [Owre and Shankar 1997], suppose τ denotes the type-checking function such that $\tau(\Gamma)(A)$ returns TYPE when A is a well formed type under context Γ. Suppose that $\tau(\Gamma)(\mathtt{real}) = \tau(\Gamma)(\mathtt{Sign}) = \mathtt{TYPE}$, then we should verify that

$$[\mathtt{x}:[\mathtt{nonzero\_real},\mathtt{real}],\{\mathtt{c}:\mathtt{real}|\mathtt{discr}(\mathtt{p1}(\mathtt{x}),\mathtt{p2}(\mathtt{x}),\mathtt{c})>=0\}]$$

is well formed under context Γ: τ returns TYPE for it under context Γ.

In PVS, the type rule for dependent product types $[x : A, B]$ is given by

$$\begin{array}{lcll}\tau(\Gamma)([x:A,B]) & = & \mathtt{TYPE} & \text{if} \quad \Gamma(x) \text{ undefined and}\\ & & & \quad\ \tau(\Gamma)(A)=\mathtt{TYPE} \text{ and}\\ & & & \quad\ \tau(\Gamma,x:VAR\ \ A)(B)=\mathtt{TYPE}\end{array}$$

Thus, supposing $\Gamma(x)$ undefined,

$$\tau(\Gamma)([\mathtt{x}:[\mathtt{nonzero\_real},\mathtt{real}],\{\mathtt{c}:\mathtt{real}|\mathtt{discr}(\mathtt{p1}(\mathtt{x}),\mathtt{p2}(\mathtt{x}),\mathtt{c})>=0\}])=\mathtt{TYPE}$$

because $\tau(\Gamma)([\mathtt{nonzero\_real},\mathtt{real}]) = \mathtt{TYPE}$ since $\tau(\Gamma)(\mathtt{nonzero\_real}) = \tau(\Gamma)(\mathtt{real}) = \mathtt{TYPE}$, and

$$\tau(\Gamma,\mathtt{x}:\mathtt{VAR}\ \ [\mathtt{nonzero\_real},\mathtt{real}])(\{\mathtt{c}:\mathtt{real}|\mathtt{discr}(\mathtt{p1}(\mathtt{x}),\mathtt{p2}(\mathtt{x}),\mathtt{c})>=0\})=\mathtt{TYPE}$$

since $\tau(\Gamma)(\mathtt{real}) = \mathtt{TYPE}$ and

$$\tau(\Gamma,\mathtt{x}:\mathtt{VAR}\ \ [\mathtt{nonzero\_real},\mathtt{real}],\mathtt{c}:\mathtt{VAR}\ \mathtt{real})(\mathtt{discr}(\mathtt{p1}(\mathtt{x}),\mathtt{p2}(\mathtt{x}),\mathtt{c})>=0)=\mathtt{bool}$$

where $\tau(\Gamma,\mathtt{x}:\mathtt{VAR}\ \ [\mathtt{nonzero\_real},\mathtt{real}])(\mathtt{c})$ is supposed to be undefined too.

3. Detection and resolution of air traffic conflicts

Conflict Detection and Resolution (for short CD&R) algorithms warn about potential loss of air traffic *separation* and output avoidance maneuvers to be flown by a reference aircraft called the *own-ship*. The RR3D CD&R algorithm can be classified as a 3-dimensional algorithm which produces an alert if a conflict is detected and produces minimal resolution trajectories, which only affect one parameter at a time, to maintain separation to one *intruder* aircraft. The information that RR3D requires is the position and velocity of the own-ship and intruder, respectively denoted by $\vec{v_o}, \vec{s_o}, \vec{v_i}$ and $\vec{s_i}$ in $\mathbb{R}^3$. Coordinates of a 3-dimensional vector $\vec{s}$ are denoted by s_x, s_y and s_z. Additionally, it requires the time t'' of arrival at the final point $\vec{s''}$ (of the current trajectory).

Aircrafts are represented by a kinematic particle model with the center of gravity as the coordinate point of the particle. Trajectories are assumed to be composed of linear segments where the speed is constant and acceleration from one segment to another is supposed to be instantaneous. The intruder aircraft is surrounded by a cylindrical *protected zone* P of diameter $2D$ and height $2H$, where D is the required horizontal separation and H the vertical separation. A *conflict* is an intrusion of the own-ship in the intruder's protected zone. Escape and recovery maneuvers that are tangential to the intruder's protection zone are obviously the better solutions.

The considered CD&R algorithms work on a relative coordinate system where the intruder is fixed at the origin. In this system the relative position and velocity are given by $\vec{s} = \vec{s_o} - \vec{s_i}$ and $\vec{v} = \vec{v_o} - \vec{v_i}$. The target point of the trajectory (relative final point) is given by $\vec{s''} = \vec{s} + t''\vec{v}$.

Escape and recovery maneuvers for the own-ship (see Figure 1) are triplets $\langle \vec{v'_o}, t', \vec{v''_o} \rangle$, where $\vec{v'_o}$ is the escape velocity, t' is the switching time and $\vec{v''_o}$ is the recovery velocity. In RR3D, escape and recovery maneuvers are constrained in such a way that both $\vec{v'_o}$ and $\vec{v''_o}$ satisfy one of the following conditions.

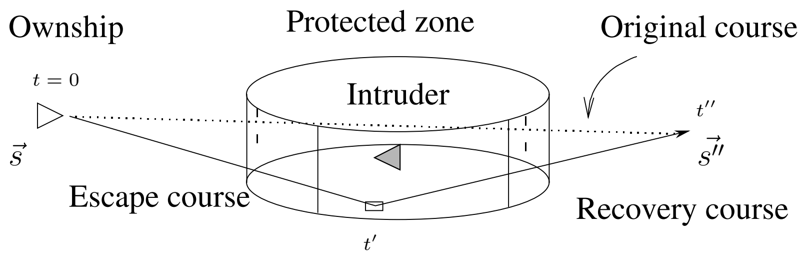

Figure 1. Escape and recovery maneuvers (case line-line)

- Exclusive change of vertical speed: $v'_{ox} = v''_{ox} = v_{ox}$ and $v'_{oy} = v''_{oy} = v_{oy}$.
- Exclusive change of ground speed: $v'_{oz} = v''_{oz} = v_{oz}$ and there exist $j, k > 0$ such that $v'_{ox} = kv_{ox}$, $v'_{oy} = kv_{oy}$, $v''_{ox} = jv_{ox}$ and $v''_{oy} = jv_{oy}$.
- Exclusive change of heading for the escape trajectory: $v'_{oz} = v_{oz}$, $v'^2_{ox} + v'^2_{oy} = v^2_{ox} + v^2_{oy}$ and $v''_{oz} = v_{oz}$.

In the third case the recovery trajectory should allows for changes in heading and ground speed, because an exclusive change in heading for the escape trajectory implies by the triangle inequality in the necessity to increment ground speed for reaching the target point at time t''.

Since we consider escape and recovery maneuvers that are tangential to the protected zone, the possible solutions can be classified according to the part of the surface of the protected zone P that is touched during the escape and recovery courses. Ten cases are identified in RR3D. When during one trajectory the lateral surface of the cylinder is touched we use the notation "line" and when the cylinder is touched at its superior or inferior border we use notation "circle". Thus, a solution "line-line" is one that touches tangentially the lateral surface of the cylinder during its escape as well as recovery maneuvers; a solution "line-circle" touches the lateral surface and then, during the recovery maneuver, the border of the cylinder. What justifies this characterization according to tangential solutions is the fact that tangential courses require the least effort to correct the original trajectory, where effort can be measured by the difference between the original velocity vector and the escape velocity vector. Also observe that the less the change from the original velocity vector to the escape velocity vector the less the necessary change for a recovery velocity vector.

4. Resolution of conflicts combining ground speed and heading

Escape trajectories which combine ground speed and heading are closer to the optimal geometric solutions than the ones specified in KB3D and RR3D [Geser et al. 2002, Maddalon et al. 2004] and mentioned in the previous section. The main problem is to express complete solutions (i.e. triplets of escape, recovery trajectories and switching times) with simple (in)equational systems. Escape trajectories which combine both ground speed and heading changes are presented here for the cases line-line and line-circle. Also, combined escape trajectories for the cases circle-line and circle-circle are considered in [Fernandes 2004]. The improved solution for the case line-line is analytically presented before its correctness proof in PVS is described.

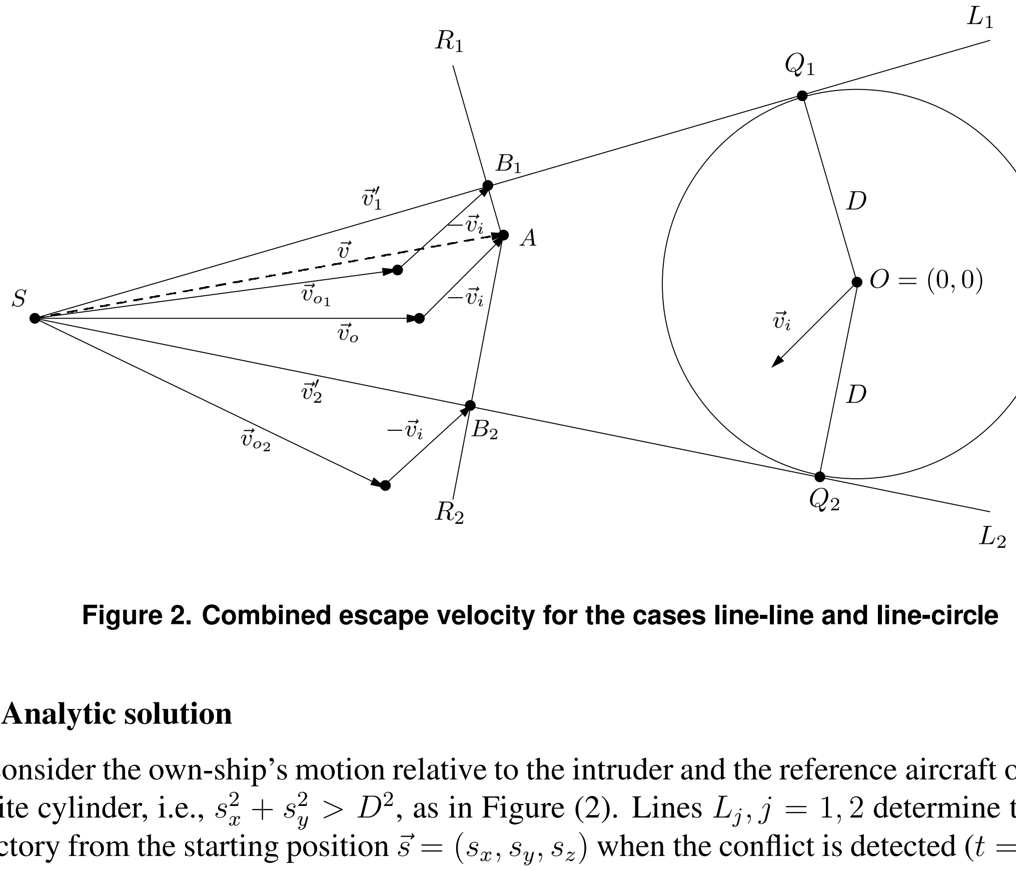

Figure 2. Combined escape velocity for the cases line-line and line-circle

4.1. Analytic solution

We consider the own-ship's motion relative to the intruder and the reference aircraft outside the infinite cylinder, i.e., $s_x^2 + s_y^2 > D^2$, as in Figure (2). Lines $L_j, j = 1, 2$ determine the escape trajectory from the starting position $\vec{s} = (s_x, s_y, s_z)$ when the conflict is detected ($t = 0$). Lines R_j are respectively perpendicular to $L_j, j = 1, 2$ through the extremity point of the vector of relative velocity $\vec{v}$, given by point A. Let B_j the intersection points between R_j and $L_j, j = 1, 2$, respectively.

Knowing A and B_j, escape relative velocities $\vec{v'_j}$ can be determined and, consequently, escape velocities $\vec{v'}_{o_j}$ can be determined ($\vec{v'_j} = \vec{v'}_{o_j} - \vec{v_i}$), $j = 1, 2$, respectively . The selected escape velocity will be the one associated with the minor distance between points A-B_1 and A-B_2.

A system for determining the tangential points $Q_j, j = 1, 2$, which provide the escape trajectories can be formulated from the equation of the circumference and the equation describing a line through S and tangent to the circumference as illustrated in Figure 3.

By the Theorem of Pythagoras, we obtain

$$s_x^2 + s_y^2 = (s_x - x)^2 + (s_y - y)^2 + x^2 + y^2$$

which gives the equation

$$D^2 = s_x x + s_y y \tag{1}$$

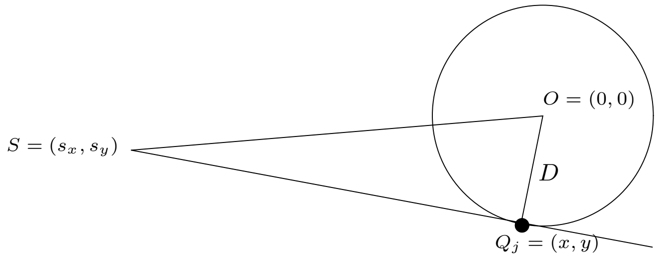

Figure 3. Alternative presentation of the equational system

The resulting system is given by this equation and the equation of the circumference.

$$\begin{cases} x^2 + y^2 = D^2 \\ s_x x + s_y y = D^2 \end{cases} \tag{2}$$

The resolution of the system (2), gives the quadratic equation (3):

$$y^2(s_x^2 + s_y^2) + y(-2s_y D^2) + (D^4 - s_x^2 D^2) = 0 \tag{3}$$

Since $s_x^2 + s_y^2 > D^2$, the system (2) has real solutions. In fact, notice that since $D > 0$ the equation of the circumference is well-defined and the discriminant of the previous quadratic equation is non-negative:

$$s_y^2 D^4 - s_x^2 D^4 - s_y^2 D^4 + s_x^4 D^2 + s_y^2 s_x^2 D^2 = -s_x^2 D^4 + s_x^4 D^2 + s_y^2 s_x^2 D^2 = s_x^2 D^2 (s_x^2 + s_y^2 - D^2) \geq 0$$

From the equation (3) we obtain

$$y_j = \frac{s_y D^2 \pm s_x D \sqrt{s_x^2 + s_y^2 - D^2}}{s_x^2 + s_y^2}, \quad j = 1, 2 \tag{4}$$

Changing y_j, $j = 1, 2$ in the second equation of the system (2), it reduces to

$$s_x x_j = D^2 - s_y \left[\frac{s_y D^2 \pm s_x D \sqrt{s_x^2 + s_y^2 - D^2}}{s_x^2 + s_y^2} \right], \quad j = 1, 2$$

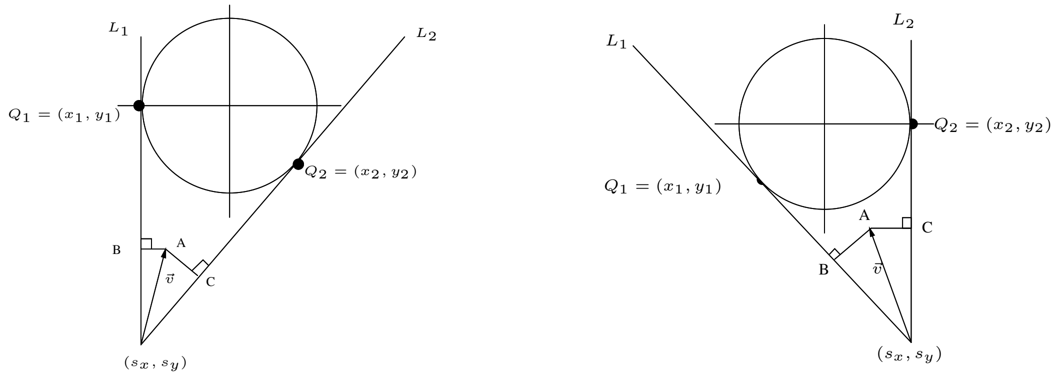

Figure 4. Case L_j vertical

Since $s_x^2+s_y^2 > D^2$, these solutions are possible and, consequently, the systems (2) has solution.

Two cases have to be considered according to whether $s_x \neq 0$ or $s_x = 0$. This treatment is done in the following theorem which compiles all previous results giving the constructive solution of the tangential points $Q_j, j = 1, 2$.

Theorem 4.1 *Suppose $s_x^2 + s_y^2 > D^2$. The system (2) has the following solutions.*

1. *Case $s_x \neq 0$:*

$$\begin{aligned} y_j &= \frac{s_y D^2 \pm s_x D\sqrt{s_x^2 + s_y^2 - D^2}}{s_x^2 + s_y^2}, \quad j = 1, 2 \\ x_j &= \frac{D^2}{s_x} - \frac{s_y}{s_x}\left[\frac{s_y D^2 \pm s_x D\sqrt{s_x^2 + s_y^2 - D^2}}{s_x^2 + s_y^2}\right], \quad j = 1, 2 \end{aligned}$$

2. *Case $s_x = 0$: $y_j = \dfrac{D^2}{s_y}$ and $x_j = \pm\sqrt{D^2 - y_j^2}, j = 1, 2$*

Tangential points of lines $L_j, j = 1, 2$ with the protection zone (Figure 2) are defined as $Q_j = (x_j, y_j), j = 1, 2$. Once points Q_j are found, points B_j of intersection between lines $R_j, j = 1, 2$ and $L_j, j = 1, 2$, resp., can be found (Figure (2)). $B_j, j = 1, 2$ give the magnitude of the relative escape velocity vectors $v'_j, j = 1, 2$, resp., which define combined changes in heading and ground speed.

In order to find $B_j, j = 1, 2$, it is necessary to solve the linear systems of the equations of lines R_j and $L_j, j = 1, 2$.

Denote the x-y components of the tangential points as $Q_j = (x_j, y_j)$. Then the linear equations of L_j ($j = 1, 2$) are given by

$$L_j : \begin{cases} y = \left(\dfrac{y_j - s_y}{x_j - s_x}\right) x + s_y - \left(\dfrac{y_j - s_y}{x_j - s_x}\right) s_x, & \text{case } s_x \neq x_j \\ x = s_x, & \text{otherwise} \end{cases}$$

The second case happens when the x-coordinate of the original position has magnitude D (see Figures 4).

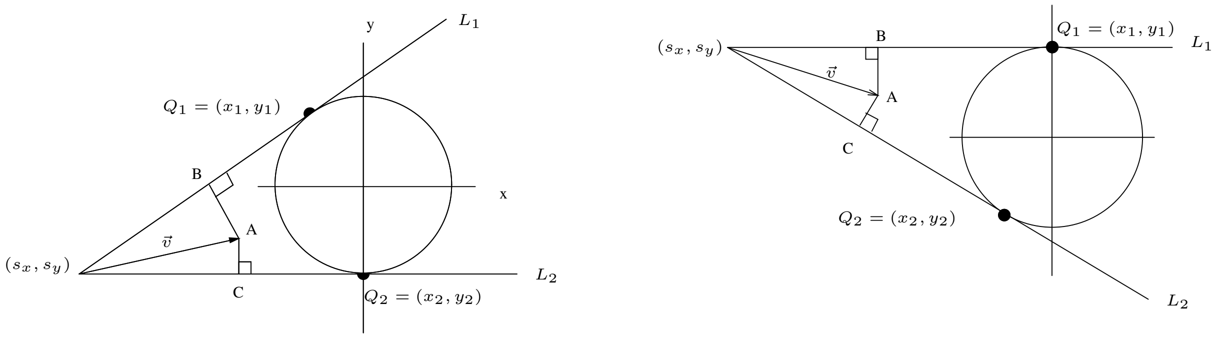

Figure 5. Case L_j horizontal

R_j are lines through the point A which are perpendicular to lines L_j, $j = 1, 2$, respectively. Thus, their angular coefficients r_j are given by $r_j = -\dfrac{1}{l_j}$, where l_j is the angular coefficient of L_j.

$$r_j = -\frac{x_j - s_x}{y_j - s_y}$$

Notice that this angular coefficient is defined only when $y_j \neq s_y$; in other words, whenever $|s_y| \neq D$ (see figure 5).

Since R_j, $j = 1, 2$, are lines through A, the linear equations of R_j are given by

$$R_j : \begin{cases} y = \left(-\dfrac{x_j - s_x}{y_j - s_y}\right) x + \left[(s_y + v_y.1) - \left(-\dfrac{x_j - s_x}{y_j - s_y}\right)(s_x + v_x.1)\right], & \text{case } s_y \neq y_j \\ x = s_x + v_x \cdot 1 & \text{otherwise} \end{cases}$$

Since we work with the position vector, coordinates of the velocity vector are multiplied by one (unity of time) to obtain A (Figure (2)).

Points B_j are obtained after resolving the two systems of intersection points between lines L_j and R_j for $j = 1, 2$, respectively:

$$\begin{cases} L_j : \begin{cases} y = \left(\dfrac{y_j - s_y}{x_j - s_x}\right) x + s_y - \left(\dfrac{y_j - s_y}{x_j - s_x}\right) s_x, & \text{case } s_x \neq x_j \\ x = s_x, & \text{otherwise} \end{cases} \\ R_j : \begin{cases} y = \left(-\dfrac{x_j - s_x}{y_j - s_y}\right) x + \left[(s_y + v_y.1) - \left(-\dfrac{x_j - s_x}{y_j - s_y}\right)(s_x + v_x.1)\right], & \text{case } s_y \neq y_j \\ x = s_x + v_x \cdot 1 & \text{otherwise} \end{cases} \end{cases} \tag{5}$$

Some particular situations for obtaining escape velocity vectors are related with Figures 5 and 4. Observe the Figure 5. If $s_y = y_j$, L_j goes through $\vec{s}$ and Q_j, parallel to the x-axis and the $\vec{v_j'}$ and $B_j, j = 1, 2$ are given by

$$\vec{v_j'} = (v_x, 0) \quad \text{and} \quad B_j = (s_x + v_x.1, s_y)$$

Suppose $s_x = x_j$. L_j goes through $\vec{s}$ and Q_j, parallel to the y-axis (Figure 4). The vector $\vec{v'_j}$ and B_j are given by

$$\vec{v'_j} = (0, v_y) \quad \text{and} \quad B_j = (s_x, s_y + v_y.1)$$

If none of the previous special cases happen then both $B_j, j = 1, 2$ are obtained as general solutions of the two systems (5).

The selection of the point which gives the better escape relative velocity vector is determined by the minor distance between A, B_1 and A, B_2. Since A and $B_j, j = 1, 2$ can be expressed as

$$A = (s_x + v_x.1, s_y + v_y.1) \quad \text{and} \quad B_j = (s_x + v'_{x_j}.1, s_y + v'_{y_j}.1)$$

where the two escape relative velocities which can resolve the conflict are denoted as $\vec{v'_j}, j = 1, 2$, such as in the Figure 2. The distance between these points are given by

$$\begin{aligned} |\overline{AB_j}| &= \sqrt{((s_x + v_x.1) - (s_x + v'_{x_j}.1))^2 + ((s_y + v_y.1) - (s_y + v'_{y_j}.1))^2} \\ &= \sqrt{((v_x - v'_{x_j}).1)^2 + ((v_y - v'_{y_j}).1)^2} \end{aligned}$$

4.2. Proof of correctness in PVS

A brief description of the correctness proof is presented. In this proof we have applied the tools and strategies in Field [Muñoz and Mayero 2001] and Manip [Vito 2003] for manipulation of field theories and algebraic simplification, respectively.

Proof of Theorem 4.1 is completed after proving two lemmas for the two cases: $s_x = 0$ and $s_x \neq 0$. The most complex case ($s_x \neq 0$) is proved by algebraic manipulation of the equational system (using strategies in Field and Manip) and then application of a the lemma `quadratic_eq_0` proved in RR3D and presented in the Section 2. Notice that this lemma uses between the proof the condition on (dependent) types given in the definition of root as explained in section 2.

The Theorem 4.1 can be written in PVS as follows.

```
tangentialpoints: THEOREM
   ( sq(x(s)) + sq(y(s)) >  sq(D) AND
     sq(x) + sq(y) = sq(D) AND
     x(s)*x + y(s)*y = sq(D) ) IMPLIES
    (y = (y(s) * sq(D) +   sqrt(sq(y(s)) * sq(sq(D)) - (sq(x(s)) +
         sq(y(s))) * (sq(sq(D))-sq(x(s))*sq(D))))/ (sq(x(s)) + sq(y(s)))
     AND
     x = sqrt(sq(D) - sq(y)) OR x = -sqrt(sq(D) - sq(y)))
   OR
    ((y = root(sq(x(s)) + sq(y(s)), -2*y(s)*sq(D),
                                             sq(sq(D))-sq(x(s)) * sq(D), 1)
      AND
      x = (sq(D) - y(s) * y) / x(s) )
   OR
     (y = root(sq(x(s)) + sq(y(s)), -2*y(s)*sq(D),
                                            sq(sq(D))-sq(x(s)) * sq(D), -1)
      AND
      x = (sq(D) - y(s) * y) / x(s)  ))
```

Observe the use of the functions `root` and mainly the fact that its use and its well-definedness makes part of the proof of this theorem.

Here we present the proof of the lemma for the simplest case: $s_x = 0$. The case: $s_x \neq 0$ and the proof of Theorem 4.1 are included in the appendices A and B, respectively.

We use the structure of three dimensional vectors `Vect3` from RR3D. Then in this proof, the x- and y-components of the initial position, s_x and s_y, are the first and second components of `s`, denoted by `s(0)` and `s(1)`, respectively. The case (2) of Theorem 4.1, denoted as lemma `combGSHL12`, is written as

```
combGSHL12 :

  |-------
{1}   FORALL (s: Vect3, x, y: real):
        (s(0) = 0 AND
          sq(s(0)) + sq(s(1)) > sq(D) AND
           sq(x) + sq(y) = sq(D) AND s(0) * x + s(1) * y = sq(D))
         IMPLIES
         y =
          (s(1) * sq(D) +
            sqrt(sq(s(1)) * sq(sq(D)) -
                    (sq(s(0)) + sq(s(1))) * (sq(sq(D)) - sq(s(0)) * sq(D))))
           / (sq(s(0)) + sq(s(1)))
          AND (x = sqrt(sq(D) - sq(y)) OR x = -sqrt(sq(D) - sq(y)))
```

Notice that the `y` in the consequent of this sequent is not given as in the Theorem 4.1 in the simplified form: $y_i = \dfrac{D^2}{s_y}$ because this is the general form used in the proof of the Theorem. The first step to prove this sequent is to eliminate the universal quantifier by Skolemnization. Skolemnized variables are then written with the symbol “!” and the sequent reorganized.

```
Rerunning step: (skosimp*)
Repeatedly Skolemizing and flattening,
this simplifies to:
combGSHL12 :

{-1}  s!1(0) = 0
{-2}  sq(s!1(0)) + sq(s!1(1)) > sq(D)
{-3}  sq(x!1) + sq(y!1) = sq(D)
{-4}  s!1(0) * x!1 + s!1(1) * y!1 = sq(D)
  |-------
{1}   y!1 =
       (s!1(1) * sq(D) +
         sqrt(sq(s!1(1)) * sq(sq(D)) -
                (sq(s!1(0)) + sq(s!1(1))) *
                 (sq(sq(D)) - sq(s!1(0)) * sq(D))))
        / (sq(s!1(0)) + sq(s!1(1)))
       AND (x!1 = sqrt(sq(D) - sq(y!1)) OR x!1 = -sqrt(sq(D) - sq(y!1)))
```

Secondly, the condition `s!1(0) = 0` (i.e., $s_x = 0$) is replaced in the whole sequent and simplified with the commands `replace` and `grind`. After, this we have two prove two cases because of the `AND` in the consequent.

```
Rerunning step: (replace -1)
```

```
Replacing using formula -1,
this simplifies to:
combGSHL12 :

[-1]  s!1(0) = 0
{-2}  sq(0) + sq(s!1(1)) > sq(D)
[-3]  sq(x!1) + sq(y!1) = sq(D)
{-4}  0 * x!1 + s!1(1) * y!1 = sq(D)
  |-------
{1}   y!1 =
       (s!1(1) * sq(D) +
         sqrt(sq(s!1(1)) * sq(sq(D)) -
                (sq(0) + sq(s!1(1))) * (sq(sq(D)) - sq(0) * sq(D))))
        / (sq(0) + sq(s!1(1)))
       AND (x!1 = sqrt(sq(D) - sq(y!1)) OR x!1 = -sqrt(sq(D) - sq(y!1)))

Rerunning step: (grind)
...
Trying repeated skolemization, instantiation, and if-lifting,
this yields  2 subgoals:
combGSHL12.1 :

[-1]  s!1(0) = 0
{-2}  s!1(1) * s!1(1) > D * D
{-3}  x!1 * x!1 + y!1 * y!1 = D * D
{-4}  s!1(1) * y!1 = D * D
  |-------
{1}   x!1 = sqrt(D * D - y!1 * y!1)
{2}   x!1 = -sqrt(D * D - y!1 * y!1)
```

The first case, denoted `combGSHL12.1` is proved by applying an auxiliary lemma `combLaux7`, which basically states that: `comLaux7: LEMMA x*x = y IMPLIES x=sqrt(y) OR x=-sqrt(y)`. `use-with` is a strategy of Manip which adapts lemmas for their immediate (obvious) application, which is not straightforward in algebraic manipulation. A further application of `assert` concludes this case of the proof.

```
Rerunning step: (use-with "combLaux7")
[Manip.use-with]  No formulas or preferred terms selected.

Using combLaux7 with formula preferences for instantiation,
this simplifies to:
combGSHL12.1 :

{-1}  x!1 * x!1 = D * D - y!1 * y!1 IMPLIES
       x!1 = sqrt(D * D - y!1 * y!1) OR x!1 = -sqrt(D * D - y!1 * y!1)
[-2]  s!1(0) = 0
[-3]  s!1(1) * s!1(1) > D * D
[-4]  x!1 * x!1 + y!1 * y!1 = D * D
[-5]  s!1(1) * y!1 = D * D
  |-------
[1]   x!1 = sqrt(D * D - y!1 * y!1)
[2]   x!1 = -sqrt(D * D - y!1 * y!1)

Rerunning step: (assert)
```

```
Simplifying, rewriting, and recording with decision procedures,

This completes the proof of combGSHL12.1.
```

The proof is concluded by proving the second case, denoted `combGSHL12.2`, by applying an auxiliary lemma `sqrt_0`, which states that the square-root of zero is zero, before asserting.

```
combGSHL12.2 :

[-1]  s!1(0) = 0
{-2}  s!1(1) * s!1(1) > D * D
{-3}  x!1 * x!1 + y!1 * y!1 = D * D
{-4}  s!1(1) * y!1 = D * D
  |-------
{1}   y!1 = (sqrt(0) + s!1(1) * (D * D)) / (s!1(1) * s!1(1))

Rerunning step: (use-with "sqrt_0")
[Manip.use-with]  No formulas or preferred terms selected.

Using sqrt_0 with formula preferences for instantiation,
this simplifies to:
combGSHL12.2 :

{-1}  sqrt(0) = 0
[-2]  s!1(0) = 0
[-3]  s!1(1) * s!1(1) > D * D
[-4]  x!1 * x!1 + y!1 * y!1 = D * D
[-5]  s!1(1) * y!1 = D * D
  |-------
[1]   y!1 = (sqrt(0) + s!1(1) * (D * D)) / (s!1(1) * s!1(1))

Rerunning step: (assert)
Simplifying, rewriting, and recording with decision procedures,

This completes the proof of combGSHL12.2.

Q.E.D.
```

5. Conclusion

We illustrate why application of the refined dependent types available in the proof assistant PVS are useful in algebraic and geometrical specifications. In particular, we make evident this fact showing the specification in RR3D of a total function *root* over a dependent domain of triplets of reals which encapsulates the conditions of existence of real roots of quadratic polynomials. Further use of this total function (and many others) and its associated proof obligations in PVS were useful in mechanical verification of the system RR3D as well as of our partial extension of the solutions presented in these systems. Our partial extension of the solutions in this systems allow for escape trajectories which combine changes in ground speed and heading, giving rise in this way to escape trajectories closer to the optimal solutions than the ones presented in RR3D where only exclusive changes in a sole of these parameters are allowed. For the case line-line complete analytical solutions (triplets including escape and recovery velocities as well as switching time) were totally characterized. In these complete solutions the possible recovery trajectories are specified in a similar way to escape trajectories and the switching time is determined by intersecting these two trajectories.

Joint work in progress with the Formal Methods group at NIA will describe PVS proofs of the correctness of complete escape/recovery maneuvers and a proof of the optimality of the escape maneuvers from the point of view of the horizontal 2 dimensional space. In addition, we are exploring the available typing techniques and logical power of PVS from other point of view: we develop a translation tool from rewriting specifications written in ELAN (mainly hardware specifications [Ayala-Rincón et al. 2002, Jacobi et al. 2005, Ayala-Rincón et al. 2006]) into the language of PVS [Ayala-Rincón and Sant'Ana 2006]. This translation includes mechanisms for generating critical pairs (of term rewriting systems) whose joinability is checked in PVS. Including other general strategies for dealing with proving techniques from rewriting theory, one will have a powerful proving mechanism for checking integrity of rewriting based specifications. In this way we are advancing in the formulation of extraction mechanisms of correct and secure code for implementing and obtaining software and hardware without flaws.

Acknowledgments The authors would like to thank César Muñoz for his constructive suggestions and valuable assistance with the use of PVS, Manip and Field.

References

Augustsson, L. (1999). Cayenne - A Language with Dependent Types. In *Advanced Functional Programming, Third International School, 1998, Revised Lectures*, volume 1608 of *Lecture Notes in Computer Science*, pages 240–267. Springer Verlag.

Ayala-Rincón, M., Hartenstein, R., Neto, R. M., Jacobi, R. P., and Llanos, C. (2002). Architectural Specification, Exploration and Simulation Through Rewriting-Logic. *Colombian Journal of Computation*, 3(2):20–34.

Ayala-Rincón, M., Llanos, C., Jacobi, R. P., and Hartenstein, R. W. (2006). Prototyping Time and Space Efficient Computations of Algebraic Operations over Dynamically Reconfigurable Systems Modeled by Rewriting-Logic. *ACM Transactions on Design Automation of Electronic Systems*, 11(2):31 pages. In press.

Ayala-Rincón, M. and Sant'Ana, T. M. (2006). SAEPTUM: Verification of ELAN Hardware Specifications using the Proof Assistant PVS. In *19th Symp. on Integrated Circuits and System Design*, page 6 pages. Accepted.

Barendregt, H. and Geuvers, H. (2001). Proof-Assistants Using Dependent Type Systems. In Robinson, J. A. and Voronkov, A., editors, *Handbook of Automated Reasoning*, volume 2, chapter 18, pages 1149–1238. Elsevier and MIT Press.

Church, A. (1940). A formulation of the simple theory of types. *Journal of Symbolic Logic*, 5:56–68.

Fernandes, V. (2004). Detecção e Resolução Formal de Conflitos de Tráfego Aéreo. Master's thesis, Departamento de Matemática, Universidade de Brasília, 70910-900 Brasília D.F., Brazil.

Geser, A., Muñoz, C., Dowek, G., and Kirchner, F. (2002). Air Traffic Conflict Resolution and Recovery. ICASE Report 2002-12, ICASE, Langley Research Center.

Jacobi, R. P., Ayala-Rincón, M., Carvalho, L. G. A., Llanos, C., and Hartenstein, R. (2005). Reconfigurable Systems for Sequence Alignment and for General Dynamic Programming. *Genetic and Molecular Research*, 4(3):543–552.

Maddalon, J., Butler, R., Geser, A., and Muñoz, C. (2004). Formal verification of a conflict resolution and recovery algorithm. Technical Report NASA/TP-2004-213015, NASA Langley Research Center, NASA LaRC,Hampton VA 23681-2199, USA.

Muñoz, C. and Mayero, M. (2001). Real automation in the field. ICASE Interim Report 39 NASA/CR-2001-211271, NASA Langley Research Center, NASA Langley Research Center.

Owre, S., Rushby, J., Shankar, N., and Stringer-Calvert, D. (1998). PVS: An Experience Report. In *Applied Formal Methods-FM-Trends 98*, volume 1641 of *Lecture Notes in Computer Science*, pages 338–345.

Owre, S. and Shankar, N. (1997). The Formal semantics of PVS. Technical report, SRI-CSL-97-2, Computer Science Laboratory, SRI International, Menlo Park, CA.

Rushby, J., Owre, S., and Shankar, N. (1998). Subtypes for Specifications: Predicate Subtyping in PVS. *IEEE Transactions on Software Engineering*, 24(9):709–720.

Vito, B. D. (2003). *Manip User's Guide, Version 1.1*. NASA Langley Research Center, Hampton, Virginia.

Watkins, K., Cervesato, I., Pfenning, F., and Walker, D. CLF: A Dependent Logical Framework for Concurrent Computations. Unpublished.

Xi, H. (2001). Dependent Types for Program Termination Verification. In *Proceedings of 16th IEEE Symposium on Logic in Computer Science (LICS '01)*.

Xi, H. and Pfenning, F. (1999). Dependent Types in Practical Programming. In *Proceedings of Symposium on Principles of Programming Languages (POPL '99)*, pages 214–227.

We remark that the two appendices included are unessential for a final version of the paper and they are only presented for benefit of the refereeing process.

A. Proof of case $s_x \neq 0$

```
combGSHL13p :

  |-------
{1}   FORALL (s: Vect3, x, y: real):
        (s(0) /= 0 AND
          sq(s(0)) + sq(s(1)) > sq(D) AND
           sq(x) + sq(y) = sq(D) AND s(0) * x + s(1) * y = sq(D))
         IMPLIES
         (y =
           root(sq(s(0)) + sq(s(1)), -2 * s(1) * sq(D),
                sq(sq(D)) - sq(s(0)) * sq(D), 1)
           AND x = (sq(D) - s(1) * y) / s(0))
          OR
          (y =
            root(sq(s(0)) + sq(s(1)), -2 * s(1) * sq(D),
                 sq(sq(D)) - sq(s(0)) * sq(D), -1)
            AND x = (sq(D) - s(1) * y) / s(0))
```

Notice that in the hypotheses we have the system (2) and we want to obtain the quadratic equation (3). For this, we will do some algebraic manipulation using some strategies of Manip, Field and by applying some auxiliary lemmas. Firstly, we use the strategies `skosimp*, cancel-by, move-terms, expand, group`. Secondly, we apply the auxiliary lemmas `minus_add, combLaux5, combLaux8, sq_rew`, which basically states that: `combLaux5: LEMMA (j * t) + (k * t) = (j + k) * t; combLaux8: LEMMA -(j * k) = -j * k; sq_rew: LEMMA a*a = sq(a)`. After this, we obtain:

```
combGSHL13p :

[-1]  sq(D) * sq(D) = sq(sq(D))
[-2]  -(2 * (s!1(1) * sq(D))) = -2 * (s!1(1) * sq(D))
[-3]  -((2 * (s!1(1) * sq(D))) * y!1) = -(2 * (s!1(1) * sq(D))) * y!1
[-4]  (sq(s!1(0)) + (sq(s!1(1)))) * sq(y!1) - (2 * (s!1(1) * sq(D))) * y!1
       =
       (sq(s!1(0)) + (sq(s!1(1)))) * sq(y!1) +
        -((2 * (s!1(1) * sq(D))) * y!1)
[-5]  (sq(s!1(0)) * sq(y!1)) + ((sq(s!1(1))) * sq(y!1)) =
       (sq(s!1(0)) + (sq(s!1(1)))) * sq(y!1)
[-6]  s!1(0) * s!1(0) = sq(s!1(0))
[-7]  s!1(1) * s!1(1) = sq(s!1(1))
[-8]  y!1 * y!1 = sq(y!1)
{-9}  ((sq(s!1(0)) + (sq(s!1(1)))) * sq(y!1) + -2 * (s!1(1) * sq(D)) * y!1
        + sq(sq(D))
        - sq(s!1(0)) * sq(D)
        = 0)
[-10] (x!1 = sq(D) / s!1(0) - (y!1 * s!1(1)) / s!1(0))
[-11] sq(s!1(0)) + sq(s!1(1)) > sq(D)
  |-------
[1]   1 / s!1(0) = 0
[2]   1 / s!1(0) = 0
[3]   s!1(0) = 0
[4]   s!1(0) = 0
[5]   y!1 =
       root(sq(s!1(0)) + sq(s!1(1)), -2 * (s!1(1) * sq(D)),
            sq(sq(D)) - sq(s!1(0)) * sq(D), 1)
[6]   y!1 =
       root(sq(s!1(0)) + sq(s!1(1)), -2 * (s!1(1) * sq(D)),
            sq(sq(D)) - sq(s!1(0)) * sq(D), -1)
```

A further application of `quadratic_eq_0 : LEMMA (a*sq(x) + b*x + c = 0 IFF discr(a,b,c) >= 0 AND (x = x1(a,b,c) OR x = x2(a,b,c)))` and `assert` concludes this proof.

```
Rerunning step: (lemma quadratic_eq_0
                 ("x" "y!1" "a" "(sq(s!1(0)) + (sq(s!1(1))))" "b"
                  "-2 * (s!1(1) * sq(D))" "c"
                  "sq(sq(D)) - sq(s!1(0)) * sq(D)"))
Applying quadratic_eq_0 where
  x gets y!1,
  a gets (sq(s!1(0)) + (sq(s!1(1)))),
  b gets -2 * (s!1(1) * sq(D)),
  c gets sq(sq(D)) - sq(s!1(0)) * sq(D),
this simplifies to:
combGSHL13p :

{-1}  ((sq(s!1(0)) + (sq(s!1(1)))) * sq(y!1) + -2 * (s!1(1) * sq(D)) * y!1
        + (sq(sq(D)) - sq(s!1(0)) * sq(D))
        = 0
        IFF
        discr((sq(s!1(0)) + (sq(s!1(1)))), -2 * (s!1(1) * sq(D)),
              sq(sq(D)) - sq(s!1(0)) * sq(D))
```

```
          >= 0
          AND
          (y!1 =
            root((sq(s!1(0)) + (sq(s!1(1)))), -2 * (s!1(1) * sq(D)),
                 sq(sq(D)) - sq(s!1(0)) * sq(D), 1)
            OR
            y!1 =
             root((sq(s!1(0)) + (sq(s!1(1)))), -2 * (s!1(1) * sq(D)),
                  sq(sq(D)) - sq(s!1(0)) * sq(D), -1)))
[-2]  sq(D) * sq(D) = sq(sq(D))
[-3]  -(2 * (s!1(1) * sq(D))) = -2 * (s!1(1) * sq(D))
[-4]  -((2 * (s!1(1) * sq(D))) * y!1) = -(2 * (s!1(1) * sq(D))) * y!1
[-5]  (sq(s!1(0)) + (sq(s!1(1)))) * sq(y!1) - (2 * (s!1(1) * sq(D))) * y!1
       =
       (sq(s!1(0)) + (sq(s!1(1)))) * sq(y!1) +
        -((2 * (s!1(1) * sq(D))) * y!1)
[-6]  (sq(s!1(0)) * sq(y!1)) + ((sq(s!1(1))) * sq(y!1)) =
       (sq(s!1(0)) + (sq(s!1(1)))) * sq(y!1)
[-7]  s!1(0) * s!1(0) = sq(s!1(0))
[-8]  s!1(1) * s!1(1) = sq(s!1(1))
[-9]  y!1 * y!1 = sq(y!1)
[-10] ((sq(s!1(0)) + (sq(s!1(1)))) * sq(y!1) + -2 * (s!1(1) * sq(D)) * y!1
        + sq(sq(D))
        - sq(s!1(0)) * sq(D)
        = 0)
[-11] (x!1 = sq(D) / s!1(0) - (y!1 * s!1(1)) / s!1(0))
[-12] sq(s!1(0)) + sq(s!1(1)) > sq(D)
   |-------
[1]   1 / s!1(0) = 0
[2]   1 / s!1(0) = 0
[3]   s!1(0) = 0
[4]   s!1(0) = 0
[5]   y!1 =
       root(sq(s!1(0)) + sq(s!1(1)), -2 * (s!1(1) * sq(D)),
            sq(sq(D)) - sq(s!1(0)) * sq(D), 1)
[6]   y!1 =
       root(sq(s!1(0)) + sq(s!1(1)), -2 * (s!1(1) * sq(D)),
            sq(sq(D)) - sq(s!1(0)) * sq(D), -1)

Rerunning step: (assert)
Simplifying, rewriting, and recording with decision procedures,
Q.E.D.
```

B. Proof of Theorem 4.1

```
tangentialpoints: THEOREM
     ( sq(x(s)) + sq(y(s)) >  sq(D) AND
       sq(x) + sq(y) = sq(D) AND
       x(s)*x + y(s)*y = sq(D) ) IMPLIES
      (y = (y(s) * sq(D) +   sqrt(sq(y(s)) * sq(sq(D)) - (sq(x(s)) +
           sq(y(s))) * (sq(sq(D))-sq(x(s))*sq(D))))/ (sq(x(s)) + sq(y(s))) AND
       (x = sqrt(sq(D) - sq(y)) OR x = -sqrt(sq(D) - sq(y)))) OR
      ((y = root(sq(x(s)) + sq(y(s)), -2*y(s)*sq(D), sq(sq(D))-sq(x(s)) * sq(D), 1)
                                                                         AND
       x = (sq(D) - y(s) * y) / x(s) ) OR
       (y = root(sq(x(s)) + sq(y(s)), -2*y(s)*sq(D), sq(sq(D))-sq(x(s)) * sq(D), -1)
                                                                         AND
       x = (sq(D) - y(s) * y) / x(s)  ))
```

The proof is given by

```
tangentialpoints :

  |-------
{1}   FORALL (s: Vect3, x, y: real):
        (sq(s(0)) + sq(s(1)) > sq(D) AND
          sq(x) + sq(y) = sq(D) AND s(0) * x + s(1) * y = sq(D))
         IMPLIES
         (y =
           (s(1) * sq(D) +
             sqrt(sq(s(1)) * sq(sq(D)) -
                   (sq(s(0)) + sq(s(1))) * (sq(sq(D)) - sq(s(0)) * sq(D))))
            / (sq(s(0)) + sq(s(1)))
           AND (x = sqrt(sq(D) - sq(y)) OR x = -sqrt(sq(D) - sq(y))))
          OR
          ((y =
             root(sq(s(0)) + sq(s(1)), -2 * s(1) * sq(D),
                  sq(sq(D)) - sq(s(0)) * sq(D), 1)
             AND x = (sq(D) - s(1) * y) / s(0))
            OR
            (y =
              root(sq(s(0)) + sq(s(1)), -2 * s(1) * sq(D),
                   sq(sq(D)) - sq(s(0)) * sq(D), -1)
              AND x = (sq(D) - s(1) * y) / s(0)))

Rerunning step: (skosimp*)
Repeatedly Skolemizing and flattening,
this simplifies to:
tangentialpoints :

{-1}  sq(s!1(0)) + sq(s!1(1)) > sq(D)
```

```
{-2}  sq(x!1) + sq(y!1) = sq(D)
{-3}  s!1(0) * x!1 + s!1(1) * y!1 = sq(D)
  |-------
{1}   (y!1 =
        (s!1(1) * sq(D) +
          sqrt(sq(s!1(1)) * sq(sq(D)) -
                 (sq(s!1(0)) + sq(s!1(1))) *
                  (sq(sq(D)) - sq(s!1(0)) * sq(D))))
         / (sq(s!1(0)) + sq(s!1(1)))
        AND (x!1 = sqrt(sq(D) - sq(y!1)) OR x!1 = -sqrt(sq(D) - sq(y!1))))
{2}   (y!1 =
        root(sq(s!1(0)) + sq(s!1(1)), -2 * s!1(1) * sq(D),
             sq(sq(D)) - sq(s!1(0)) * sq(D), 1)
        AND x!1 = (sq(D) - s!1(1) * y!1) / s!1(0))
{3}   (y!1 =
        root(sq(s!1(0)) + sq(s!1(1)), -2 * s!1(1) * sq(D),
             sq(sq(D)) - sq(s!1(0)) * sq(D), -1)
        AND x!1 = (sq(D) - s!1(1) * y!1) / s!1(0))

Rerunning step: (case "s!1(0)=0")
Case splitting on
   s!1(0) = 0,
this yields  2 subgoals:
tangentialpoints.1 :

{-1}  s!1(0) = 0
[-2]  sq(s!1(0)) + sq(s!1(1)) > sq(D)
[-3]  sq(x!1) + sq(y!1) = sq(D)
[-4]  s!1(0) * x!1 + s!1(1) * y!1 = sq(D)
  |-------
[1]   (y!1 =
        (s!1(1) * sq(D) +
          sqrt(sq(s!1(1)) * sq(sq(D)) -
                 (sq(s!1(0)) + sq(s!1(1))) *
                  (sq(sq(D)) - sq(s!1(0)) * sq(D))))
         / (sq(s!1(0)) + sq(s!1(1)))
        AND (x!1 = sqrt(sq(D) - sq(y!1)) OR x!1 = -sqrt(sq(D) - sq(y!1))))
[2]   (y!1 =
        root(sq(s!1(0)) + sq(s!1(1)), -2 * s!1(1) * sq(D),
             sq(sq(D)) - sq(s!1(0)) * sq(D), 1)
        AND x!1 = (sq(D) - s!1(1) * y!1) / s!1(0))
[3]   (y!1 =
        root(sq(s!1(0)) + sq(s!1(1)), -2 * s!1(1) * sq(D),
             sq(sq(D)) - sq(s!1(0)) * sq(D), -1)
        AND x!1 = (sq(D) - s!1(1) * y!1) / s!1(0))

Rerunning step: (use-with "combGSHL12")
[Manip.use-with]  No formulas or preferred terms selected.

Using combGSHL12 with formula preferences for instantiation,
this simplifies to:
tangentialpoints.1 :

{-1}  (s!1(0) = 0 AND
        sq(s!1(0)) + sq(s!1(1)) > sq(D) AND
         sq(x!1) + sq(y!1) = sq(D) AND s!1(0) * x!1 + s!1(1) * y!1 = sq(D))
       IMPLIES
       y!1 =
        (s!1(1) * sq(D) +
          sqrt(sq(s!1(1)) * sq(sq(D)) -
                 (sq(s!1(0)) + sq(s!1(1))) *
                  (sq(sq(D)) - sq(s!1(0)) * sq(D))))
         / (sq(s!1(0)) + sq(s!1(1)))
        AND (x!1 = sqrt(sq(D) - sq(y!1)) OR x!1 = -sqrt(sq(D) - sq(y!1)))
[-2]  s!1(0) = 0
[-3]  sq(s!1(0)) + sq(s!1(1)) > sq(D)
[-4]  sq(x!1) + sq(y!1) = sq(D)
[-5]  s!1(0) * x!1 + s!1(1) * y!1 = sq(D)
  |-------
[1]   (y!1 =
        (s!1(1) * sq(D) +
          sqrt(sq(s!1(1)) * sq(sq(D)) -
                 (sq(s!1(0)) + sq(s!1(1))) *
                  (sq(sq(D)) - sq(s!1(0)) * sq(D))))
         / (sq(s!1(0)) + sq(s!1(1)))
        AND (x!1 = sqrt(sq(D) - sq(y!1)) OR x!1 = -sqrt(sq(D) - sq(y!1))))
[2]   (y!1 =
        root(sq(s!1(0)) + sq(s!1(1)), -2 * s!1(1) * sq(D),
             sq(sq(D)) - sq(s!1(0)) * sq(D), 1)
        AND x!1 = (sq(D) - s!1(1) * y!1) / s!1(0))
[3]   (y!1 =
        root(sq(s!1(0)) + sq(s!1(1)), -2 * s!1(1) * sq(D),
             sq(sq(D)) - sq(s!1(0)) * sq(D), -1)
        AND x!1 = (sq(D) - s!1(1) * y!1) / s!1(0))

Rerunning step: (assert)
Simplifying, rewriting, and recording with decision procedures,

This completes the proof of tangentialpoints.1.

tangentialpoints.2 :

[-1]  sq(s!1(0)) + sq(s!1(1)) > sq(D)
[-2]  sq(x!1) + sq(y!1) = sq(D)
[-3]  s!1(0) * x!1 + s!1(1) * y!1 = sq(D)
```

```
  |-------
{1}   s!1(0) = 0
[2]   (y!1 =
        (s!1(1) * sq(D) +
          sqrt(sq(s!1(1)) * sq(sq(D)) -
                 (sq(s!1(0)) + sq(s!1(1))) *
                  (sq(sq(D)) - sq(s!1(0)) * sq(D))))
         / (sq(s!1(0)) + sq(s!1(1)))
        AND (x!1 = sqrt(sq(D) - sq(y!1)) OR x!1 = -sqrt(sq(D) - sq(y!1))))
[3]   (y!1 =
        root(sq(s!1(0)) + sq(s!1(1)), -2 * s!1(1) * sq(D),
             sq(sq(D)) - sq(s!1(0)) * sq(D), 1)
        AND x!1 = (sq(D) - s!1(1) * y!1) / s!1(0))
[4]   (y!1 =
        root(sq(s!1(0)) + sq(s!1(1)), -2 * s!1(1) * sq(D),
             sq(sq(D)) - sq(s!1(0)) * sq(D), -1)
        AND x!1 = (sq(D) - s!1(1) * y!1) / s!1(0))

Rerunning step: (use-with "combGSHL13p")
[Manip.use-with]  No formulas or preferred terms selected.


Using combGSHL13p with formula preferences for instantiation,
this simplifies to:
tangentialpoints.2 :

{-1}  (s!1(0) /= 0 AND
        sq(s!1(0)) + sq(s!1(1)) > sq(D) AND
         sq(x!1) + sq(y!1) = sq(D) AND s!1(0) * x!1 + s!1(1) * y!1 = sq(D))
       IMPLIES
       (y!1 =
         root(sq(s!1(0)) + sq(s!1(1)), -2 * s!1(1) * sq(D),
              sq(sq(D)) - sq(s!1(0)) * sq(D), 1)
         AND x!1 = (sq(D) - s!1(1) * y!1) / s!1(0))
        OR
        (y!1 =
          root(sq(s!1(0)) + sq(s!1(1)), -2 * s!1(1) * sq(D),
               sq(sq(D)) - sq(s!1(0)) * sq(D), -1)
          AND x!1 = (sq(D) - s!1(1) * y!1) / s!1(0))
[-2]  sq(s!1(0)) + sq(s!1(1)) > sq(D)
[-3]  sq(x!1) + sq(y!1) = sq(D)
[-4]  s!1(0) * x!1 + s!1(1) * y!1 = sq(D)
  |-------
[1]   s!1(0) = 0
[2]   (y!1 =
        (s!1(1) * sq(D) +
          sqrt(sq(s!1(1)) * sq(sq(D)) -
                 (sq(s!1(0)) + sq(s!1(1))) *
                  (sq(sq(D)) - sq(s!1(0)) * sq(D))))
         / (sq(s!1(0)) + sq(s!1(1)))
        AND (x!1 = sqrt(sq(D) - sq(y!1)) OR x!1 = -sqrt(sq(D) - sq(y!1))))
[3]   (y!1 =
        root(sq(s!1(0)) + sq(s!1(1)), -2 * s!1(1) * sq(D),
             sq(sq(D)) - sq(s!1(0)) * sq(D), 1)
        AND x!1 = (sq(D) - s!1(1) * y!1) / s!1(0))
[4]   (y!1 =
        root(sq(s!1(0)) + sq(s!1(1)), -2 * s!1(1) * sq(D),
             sq(sq(D)) - sq(s!1(0)) * sq(D), -1)
        AND x!1 = (sq(D) - s!1(1) * y!1) / s!1(0))

Rerunning step: (assert)
Simplifying, rewriting, and recording with decision procedures,
this simplifies to:
tangentialpoints.2 :

{-1}  y!1 =
       root(sq(s!1(0)) + sq(s!1(1)), -2 * (s!1(1) * sq(D)),
            sq(sq(D)) - sq(s!1(0)) * sq(D), 1)
       OR
       y!1 =
        root(sq(s!1(0)) + sq(s!1(1)), -2 * (s!1(1) * sq(D)),
             sq(sq(D)) - sq(s!1(0)) * sq(D), -1)
[-2]  sq(s!1(0)) + sq(s!1(1)) > sq(D)
[-3]  sq(x!1) + sq(y!1) = sq(D)
[-4]  s!1(0) * x!1 + s!1(1) * y!1 = sq(D)
  |-------
[1]   s!1(0) = 0
{2}   (y!1 =
        (sqrt(sq(s!1(0)) * sq(s!1(0)) * sq(D) - sq(s!1(0)) * sq(sq(D)) +
               sq(s!1(0)) * sq(s!1(1)) * sq(D))
          + s!1(1) * sq(D))
         / (sq(s!1(0)) + sq(s!1(1)))
        AND (x!1 = sqrt(sq(D) - sq(y!1)) OR x!1 = -sqrt(sq(D) - sq(y!1))))
{3}   y!1 =
       root(sq(s!1(0)) + sq(s!1(1)), -2 * (s!1(1) * sq(D)),
            sq(sq(D)) - sq(s!1(0)) * sq(D), 1)
{4}   y!1 =
       root(sq(s!1(0)) + sq(s!1(1)), -2 * (s!1(1) * sq(D)),
            sq(sq(D)) - sq(s!1(0)) * sq(D), -1)

Rerunning step: (assert)
Simplifying, rewriting, and recording with decision procedures,

This completes the proof of tangentialpoints.2.
Q.E.D.
```