Interpolants in Nonlinear Theories over the Reals

Sicun Gao and Damien Zufferey[⋆]

MIT

Abstract. We develop algorithms for computing Craig interpolants for first-order formulas over real numbers with a wide range of nonlinear functions, including transcendental functions and differential equations. We transform proof traces from δ-complete decision procedures into interpolants that consist of Boolean combinations of linear constraints. The algorithms are guaranteed to find the interpolants between two formulas A and B whenever $A \wedge B$ is not δ-satisfiable. At the same time, by exploiting δ-perturbations one can parameterize the algorithm to find interpolants with different positions between A and B. We show applications of the methods in control and robotic design, and hybrid system verification.

1 Introduction

Verification problems of complex embedded software can be reduced to solving logic formulas that contain continuous, typically nonlinear, real functions. The framework of δ-decision procedures [19,21] establishes that, under reasonable relaxations, nonlinear SMT formulas over the reals are in principle as solvable as SAT problems. Indeed, using solvers for nonlinear theories as the algorithmic engines, straightforward bounded model checking has already shown promise on nonlinear hybrid systems [9,28]. Naturally, for enhancing performance, more advanced reasoning techniques need to be introduced, extending SMT towards general quantifier elimination. However, it is well-known that quantifier elimination is not feasible for nonlinear theories over the reals. The complexity of quantifier elimination for real arithmetic (i.e., polynomials only) has a double-exponential lower bound, which is too high for most applications; when transcendental functions are further involved, the problem becomes highly undecidable.

Craig interpolation provides a weak form of quantifier elimination. Given two formulas A and B, such that $A \wedge B$ is unsatisfiable, an interpolant I is a formula satisfying: (1) $A \Rightarrow I$, (2) $B \wedge I \Rightarrow \bot$, and (3) I contains only variables common to A and B. It has found many applications in verifications: as an heuristic to compute inductive invariant [30,33,35], for predicate discovery in abstraction

[⋆] Sicun Gao was supported by NSF (Grant CCF-1161775 and CPS-1446725). Damien Zufferey was supported by NSF (Grant CCF-1138967) and DARPA (Grant FA8650-15-C-7564).

refinement loops [32], inter procedural analysis [2,3], shape analysis [1], fault-localisation [17,10,39], and so on.

In this paper, we present methods for computing Craig interpolants in expressive nonlinear theories over the reals. To do so, we extract interpolants from proofs of unsatisfiability generated by δ-decision procedures [22] that are based on Interval Constraint Propagation (ICP) [6]. The proposed algorithms are guaranteed to find the interpolants between two formulas A and B, whenever $A \wedge B$ is not δ-satisfiable.

The framework of δ-decision procedures formulates a relaxed notion of logical decisions, by allowing one-sided δ-bounded errors [19,18]. Instead of asking whether a formula has a satisfiable assignment or not, we ask if it is "*δ-satisfiable*" or "unsatisfiable". Here, a formula is δ-satisfiable if it would be satisfiable under some *δ-perturbation* on the original formula [18]. On the other hand, when the algorithm determines that the formula is "unsatisfiable", it is a definite answer and no numerical error can be involved. Indeed, we can extract proofs of unsatisfiability from such answers, even though the search algorithms themselves involve numerical errors [22]. This is accomplished by analyzing the execution trace of the search tree based on the ICP algorithm.

The core ICP algorithm uses a *branch-and-prune* loop that aims to either find a small enough box that witnesses δ-satisfiability, or detect that no solution exists. The loop consists of two main steps:

- (Prune) Use interval arithmetic to maintain overapproximations of the solution sets, so that one can "prune" out the part of the state space that does not contain solutions.
- (Branch) When the pruning operation does not make progress, one performs a depth-first search by "branching" on variables and restart pruning operations on a subset of the domain.

The loop is continued until either a small enough box that may contain a solution is found, or any conflict among the constraints is observed.

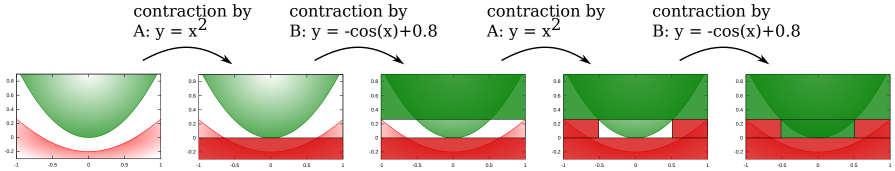


Fig. 1: Interval constraint propagation and interpolant construction where A is $y \geq x^2$ and B is $y \leq -\cos(x) + 0.8$ over the domain $x \in [-1, 1]$, $y \in [-1, 1]$. The A is shown in green and B in red. The final interpolant is the green part.

When a formula is unsatisfiable, the execution trace of the algorithm generates a (potentially large) proof tree that divides the space into small hypercubes and associating a constraint to each hypercube [22]. The interpolation algorithm can essentially traverse this proof tree to construct the interpolant. To each leaf in the proof, we associate $\top$ or $\bot$ depending on the source of the contradiction. The inner nodes of the proof tree correspond to case splits and are handled in a manner reminiscent of Pudlák's algorithm [37]. Common variables are kept as branching points and A,B local variables are eliminated. A simple example of the method is as follows:

Example 1. Let $A : y \geq x^2$ and $B : y \leq -\cos(x) + 0.8$ be two constraints over the domain $x \in [-1, 1]$, $y \in [-1, 1]$. A δ-decision procedure uses A and B to contract the domains of x and y by removing the parts that be shown empty using interval arithmetic. Figure 1 shows a sequence of contraction proving the unsatisfiability of the formula. As the contraction occurs, we color the region of the space by the color of the *opposite* formula. When the interval constraint propagation has finished, the initial domain is associated to either A or B. The interpolant I is composed of the parts corresponding to A. We will compute that I is $y \geq 0 \wedge (0.26 \leq y \vee (y \leq 0.26 \wedge -0.51 \leq x \leq 0.51))$.

We have implemented the algorithms in the SMT solver dReal [20]. We show examples of applications from various domains such as control and robotic design, and hybrid system verification.

Related work. Our algorithm is very similar to the algorithm for propositional interpolation studied by Pudlák [37]. Craig interpolation for real or integer arithmetic has focused on the linear fragment with LA($\mathbb{R}$) [31,38] and LA($\mathbb{Z}$) [8,24]. Dai et al. [15] present a method to generate interpolants for polynomial formula. Their method use semi-definite programming to search for a polynomial interpolant and it is complete under the *Archimedean* condition. In fact, the Archimedean condition imposes similar restrictions as δ-decidability, e.g., the variables over bounded domains and limited support for strict inequalities. Our method is more general in that it handles nonlinear fragments over $\mathbb{R}$ that include transcendental functions and solution functions of ordinary differention equations. Existing tools to compute interpolation such as MathSat5 [12], Princess [8], SmtInterpol [11], and Z3 [34] focus on linear arithmetic. We are the first to provide interpolation in nonlinear theories.

Outline. In Section 2, we review notions related to interpolation, nonlinear arithmetic over the Reals and δ-decision procedures. In Section 3, we introduce our interpolation algorithm. In Section 4, we present and evaluate our implementation. We conclude and sketch future research direction in Section 5.

2 Preliminaries

Craig interpolation [14]. Craig interpolants were originally defined in propositional logic, but can be easily extended to first-order logic. Given two quantifier-free first-order formulas A and B, such that $A \wedge B$ is unsatisfiable, a Craig interpolant I is a formula satisfying:

- $A \Rightarrow I$;
- $B \wedge I \Rightarrow \bot$;
- $fv(I) \subseteq fv(A) \cap fv(B)$ where $fv(\cdot)$ returns the free variables in a formula.

Intuitively, I provides an overapproximation of A that is still precise enough to exhibit its conflict with B. In particular, I involves only variables (geometrically, dimensions) that are shared by A and B.

Notation 1 *We use the meta-level symbol $\Rightarrow$ as a shorthand for logical implications in texts. In the proof rules that we will introduce shortly, $\vdash$ is used as the formal symbol with the standard interpretation as logical derivations.*

δ-Complete Decision Procedures. We consider first-order formulas interpreted over the real numbers. Our special focus is formulas that can contain arbitrary nonlinear functions that are *Type 2 computable* [40,7]. Intuitively, Type 2 computability corresponds to *numerical computability.* For our purpose, it is enough to note that this set of functions consist of all common elementary functions, as well as solutions of Lipschitz-continuous ordinary differential equations.

Interval Constraint Propagation (ICP) [6] finds solutions of real constraints using the *branch-and-prune* method, combining interval arithmetic and constraint propagation. The idea is to use interval extensions of functions to *prune* out sets of points that are not in the solution set and *branch* on intervals when such pruning can not be done, recursively until a small enough box that may contain a solution is found or inconsistency is observed. A high-level description of the decision version of ICP is given in Algorithm 1 [6,18]. The boxes, or interval domains, are written as $\boldsymbol{D}$ and c_i denotes the ith constraint.

Algorithm 1 ICP$(c_1, ..., c_m, \boldsymbol{D} = D_1 \times \cdots \times D_n, \delta)$

```
$S \leftarrow \boldsymbol{D}$
while $S \neq \emptyset$ do
    $\boldsymbol{D} \leftarrow S$.pop()
    while $\exists 1 \leq i \leq m, \boldsymbol{D} \neq_\delta$ Prune$(\boldsymbol{D}, c_i)$ do
        $\boldsymbol{D} \leftarrow$ Prune$(\boldsymbol{D}, c_i)$
    end while
    if $\boldsymbol{D} \neq \emptyset$ then
        if $\exists 1 \leq i \leq m, |\boldsymbol{D}| \geq \varepsilon$ then        ▷ $\varepsilon$ is some computable factor of $\delta$
            $\{\boldsymbol{D}_1, \boldsymbol{D}_2\} \leftarrow$ Branch$(\boldsymbol{D}, i)$
            $S$.push$(\boldsymbol{D}_1)$
            $S$.push$(\boldsymbol{D}_2)$
        else
            return sat
        end if
    end if
end while
return unsat
```

Proofs from constraint propagation. A detailed description of proof extraction from δ-decision procedure is available in [22]. Here, we use a simplified version. Intuitively, the proof of unsatisfiability recursively divides the solution space to small pieces, until it can prove (mostly using interval arithmetic) that every small piece of the domain contains no solution of the original system. Note that in such a proof, the difference between pruning and branching operations become blurred for the following reason.

Pruning operations show that one part of the domain can be discarded because no solution can exist there. Branching operations split the domain along one variable, and generates two sub-problems. From a proof perspective, the difference between the two kinds of operations is simply whether the emptiness in one part of domain follows from a simple properties of the functions (theory lemma), or requires further derivations. Indeed, as is shown in [22], the simple proof system in Figure 2 is enough for establishing all theorems that can be obtained by δ-decision procedures. The rules can be explained as follows.

- The Split rules divides the solution space into two disjoint subspaces.
- The theory lemmas (ThLem) are the leaves of the proof. They are used when the solver managed to prove the absence of solution in a given subspace.
- The Weakening rule extracts those conjunct out of the main formula.

We see that each step of the proof has a set of variables $\boldsymbol{x}$ with a domain $\boldsymbol{D}$ and F is a formula. We use vector notations in the formulas, writing $\boldsymbol{x} \in \boldsymbol{D}$ to denote $\bigwedge_i x_i \in D_i$. The domains are intervals, i.e., each D_i has the form $[l_i, u_i]$ where l_i,u_i are the lower and upper bounds for x_i. Since we are looking at unsatisfiability proofs, each node implies $\bot$. The root of the proof is the formula $A \wedge B$, and D covers the entire domain. The inner nodes are Split, and the proof's leaves are theory lemmas directly followed by weakening. To avoid duplication, we do not give a separate example here, since the full example in Figure 5 shows the structure of some proof trees obtained from such rules.

A proof of unsatisfiability can be extracted from an execution trace of Algorithm 1 when it returns unsat. The algorithm starts at the root of the proof tree and explores the proof tree depth-first. Branching (line 9) directly corresponds to the Split rule. Pruning (line 5), on the other hand, is a combination of the three rules. Let us look at $\boldsymbol{D'} = \text{Prune}(\boldsymbol{D}, c_i)$. The constraint c_i is selected with the Weakening. For each $D'_i = [l', u']$ which is strictly smaller than $D_i = [l, u]$, the Split and ThLem rules are applied. If $u' < u$ then we split on u' and a lemma shows that the interval $[u, u']$ has no solution. The same is done for the lower bounds l',l. Figure 3 shows a pruning step and the corresponding proof.

3 Interpolants in Nonlinear Theories

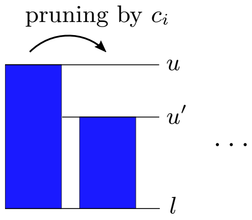
Intuitively, a proof of unsatisfiability is a partition of the solution space where each sub-domain is associated with a conjunct c from $A \wedge B$. c is a witness that shows the absence of solution in a given domain. The interpolation rules traverse the rules and selects which parts belong to the interpolant I. We now describe

$$\dfrac{}{\boldsymbol{x} \in \boldsymbol{D} \wedge c \ \vdash \ \bot}\,(\mathsf{ThLem})$$

$$\dfrac{C := c \wedge \bigwedge_k C_k \qquad \boldsymbol{x} \in \boldsymbol{D} \wedge c \ \vdash \ \bot}{\boldsymbol{x} \in \boldsymbol{D} \wedge C \ \vdash \ \bot}\,(\mathsf{Weakening})$$

$$\dfrac{x_i \in [l_i, p] \wedge \bigwedge_{j \neq i} x_j \in D_j \wedge C \ \vdash \ \bot \qquad x_i \in [p, u_i] \wedge \bigwedge_{j \neq i} x_j \in D_j \wedge C \ \vdash \ \bot}{x_i \in [l_i, u_i] \wedge \bigwedge_{j \neq i} x_j \in D_j \wedge C \ \vdash \ \bot}\,(\mathsf{Split})$$

Fig. 2: Proof rules for the ICP algorithm. We use the standard notations for sequent calculus. Also, when we write an interval $[a, b]$, we always assume that it is a well-defined real interval satisfying $a \leq b$.



$$\dfrac{\vdots \qquad \dfrac{\dfrac{}{x \in [u', u] \wedge c_i \ \vdash \ \bot}\,(\mathsf{ThLem})}{x \in [u', u] \wedge c_i \wedge \bigwedge_{k \neq i} c_k \ \vdash \ \bot}\,(\mathsf{Weakening})}{x \in [l, u] \wedge \bigwedge_k c_k \ \vdash \ \bot}\,(\mathsf{Split})$$

Fig. 3: Pruning operation and the corresponding proof. The pruning shrinks the domain of x from $[l, u]$ to $[l, u']$. The corresponding proof starts with a Split around u'. The interval $[u', u]$ is proved empty using a ThLem and Weakening step. The remaining $[l, u']$ interval is shown empty by further operations.

the algorithm for obtaining such interpolants for formulas A and B from the proof of unsatisfiability for $A \wedge B$.

3.1 Core Algorithms

Our method for constructing disjunctive linear interpolants takes two inputs: a proof tree and a labeling function. The labeling function maps formula and variables to either A, B, or AB. For each proof rule introduced in Figure 2, we associate some partial interpolants, written in square bracket on the right of the conclusion of the rule. Figure 4 shows these modified versions of the rules.

- At the leaf level (rule ThLem-I), the tile is in I if c is not part of A, i.e., the contradiction originates from B. If c is in both A and B then it can be considered as either part of A or B. Both cases lead to a correct interpolant.

$$\frac{}{\boldsymbol{x} \in \boldsymbol{D} \wedge c \ \vdash\ \bot \quad [l(c) \neq \textsc{a}]}\,(\mathsf{ThLem\text{-}I})$$

$$\frac{C = c \wedge \bigwedge_{k} C_k \qquad \boldsymbol{x} \in \boldsymbol{D} \wedge c \ \vdash\ \bot \quad [I]}{\boldsymbol{x} \in \boldsymbol{D} \wedge C \ \vdash\ \bot \quad [I]}\,(\mathsf{Weakening\text{-}I})$$

$$\frac{\begin{array}{c} x_i \in [l_i, p] \wedge \bigwedge_{j \neq i} x_j \in D_j \wedge C \ \vdash\ \bot \quad [I_1] \\ x_i \in [p, u_i] \wedge \bigwedge_{j \neq i} x_j \in D_j \wedge C \ \vdash\ \bot \quad [I_2] \end{array}}{x_i \in [l_i, u_i] \wedge \bigwedge_{j \neq i} x_j \in D_j \wedge C \ \vdash\ \bot \quad \begin{bmatrix} I_1 \vee I_2 & \text{if } l(x_i) = \textsc{a} \\ ite(x_i < p, I_1, I_2) & \text{if } l(x_i) = \textsc{ab} \\ I_1 \wedge I_2 & \text{if } l(x_i) = \textsc{b} \end{bmatrix}}\,(\mathsf{Split\text{-}I})$$

where $ite(x, y, z)$ is a shorthand for $(x \wedge y) \vee (\neg x \wedge z)$

Fig. 4: Interpolant producing proof rules

- The Weakening-I rule does not influence the interpolant, it is only required to pick c from $A \wedge B$.
- The Split-I is the most interesting rule. Splitting the domain essentially defines the bounds of the subsequent domains. Let x be the variable whose domain is split at value p and I_1, I_2 be the two interpolants for the case when $x < p$ and $x \geq p$. If x occurs in A but not B, then x cannot occur in I. Since x is in A then we know that A implies $x < p \Rightarrow I_1$ and $x \geq p \Rightarrow I_2$. Eliminating x gives $I = I_1 \vee I_2$. A similar reasoning applies when x occurs in B but not A and gives $I = I_1 \wedge I_2$. When x occurs in both A and B then x is kept in I and acts as a selector for the values of x smaller than p I_1 is selected, otherwise I_2 applies.

The correctness of our method is shown by the following theorem:

Theorem 1. *The rules Split-I, ThLem-I, Weakening-I generate a Craig interpolant I from the proof of unsatisfiability of A and B.*

Proof. We prove correctness of the rules by induction. To express the inductive invariant, we split the domain $\boldsymbol{D}$ into the domains $\boldsymbol{D}_A$ and $\boldsymbol{D}_B$ which contains only the intervals of the variables occurring in A, B respectively.

At any given point in the proof, the partial interpolant I is an interpolant for the formula A over $\boldsymbol{D}_A$ and B over $\boldsymbol{D}_B$. At the root of the proof tree we get an interpolant for the whole domain $\boldsymbol{D} = \boldsymbol{D}_A \wedge \boldsymbol{D}_B$.

At the leaves of the proof, or the ThLem-I rule, one of the constraints has no solution over the domain. Let's assume that this constraint comes from A. Then the partial interpolant I is $\bot$. We have that $A \wedge \boldsymbol{D}_A \Rightarrow I$ by the semantics of the ThLem rule ($\bot \Rightarrow \bot$). Trivially, $B \wedge \boldsymbol{D}_B \wedge I \Rightarrow \bot$ and $fv(I) = \emptyset \subseteq fv(A) \cap fv(B)$. When the contradiction comes from B, a similar reasoning applies with $I = \top$.

The $\mathsf{Weakening\text{-}I}$ only serves to select the constraint which causes the contradiction and does not change the invariant.

The $\mathsf{Split\text{-}I}$ rule is the most complex case. We have to consider whether the variable x which is split come from A, B, or is shared. For instance, if $x \in fv(A)$ then the induction step has $\boldsymbol{D}_{A1} = \boldsymbol{D}_A \wedge x < p$ and $\boldsymbol{D}_{A2} = \boldsymbol{D}_A \wedge x \geq p$ and $\boldsymbol{D}_B$ is unchanged. If $x \in fv(B)$ then $\boldsymbol{D}_B$ is affected and $\boldsymbol{D}_A$ is unchanged. If x is shared then both $\boldsymbol{D}_A$ and $\boldsymbol{D}_B$ are affected.

Let consider that $x \in fv(A)$ and $x \notin fv(B)$. We omit the case where x is in B but not A as it is similar. The induction hypothesis is

$$\begin{array}{c} A \wedge (\boldsymbol{D}_A \wedge x < p) \Rightarrow I_1 \\ A \wedge (\boldsymbol{D}_A \wedge x \geq p) \Rightarrow I_2 \\ B \wedge \boldsymbol{D}_B \wedge I_1 \Rightarrow \bot \\ B \wedge \boldsymbol{D}_B \wedge I_2 \Rightarrow \bot \end{array} \quad \text{which simplifies to} \quad \begin{array}{c} A \wedge \boldsymbol{D}_A \Rightarrow I_1 \vee I_2 \\ B \wedge \boldsymbol{D}_B \wedge (I_1 \vee I_2) \Rightarrow \bot \end{array} .$$

Finally, we need to consider $x \in fv(A)$ and $x \in fv(B)$. The induction hypothesis is

$$\begin{array}{c} A \wedge (\boldsymbol{D}_A \wedge x < p) \Rightarrow I_1 \\ A \wedge (\boldsymbol{D}_A \wedge x \geq p) \Rightarrow I_2 \\ B \wedge (\boldsymbol{D}_B \wedge x < p) \wedge I_1 \Rightarrow \bot \\ B \wedge (\boldsymbol{D}_B \wedge x \geq p) \wedge I_2 \Rightarrow \bot \end{array} \quad \text{and simplifies to} \quad \begin{array}{c} A \wedge \boldsymbol{D}_A \Rightarrow ite(x < p, I_1, I_2) \\ B \wedge \boldsymbol{D}_B \wedge ite(x < p, I_1, I_2) \Rightarrow \bot \end{array} .$$

□

Example 2. If we look at proof for the example in Figure 1, we get the proof annotated with the partial interpolants shown in Figure 5. The final interpolants I_5 is $0 \leq y \wedge (0.26 \leq y \vee (y \leq 0.26 \wedge -0.51 \leq x \leq 0.51))$.

Boolean structure. The method we presented explain how to compute an interpolant for the conjunctive fragment of quantifier-free nonlinear theories over the reals. However, in many cases formula also contains disjunctions. To handle disjunctions, our method can be combined with the method presented by Yorsh and Musuvathi [41] for building an interpolant from a resolution proof where some of the proof's leaves carry theory interpolants.

Handling ODE constraints. A special focus of δ-complete decision procedures is on constraints that are defined by ordinary differential equations, which is important for hybrid system verification. In the logic formulas, the ODEs are treated simple as a class of constraints, over variables that represent both state space and time. Here we elaborate on the proofs and interpolants for the ODE constraints.

Let $t_0, T \in \mathbb{R}$ and $g : \mathbb{R}^n \to \mathbb{R}$ be a Lipschitz-continuous Type 2 computable function. Let $t_0, T \in \mathbb{R}$ satisfy $t_0 \leq T$ and $\boldsymbol{x}_0 \in \mathbb{R}^n$. Consider the initial value

$$\dfrac{\dfrac{}{x \in [-0.51, 0.51] \wedge y \in [0, 0.26] \wedge B \ \vdash\ \bot \quad [\top]}\,(\textsf{ThLem-I})}{x \in [-0.51, 0.51] \wedge y \in [0, 0.26] \wedge A \wedge B \ \vdash\ \bot \quad [I_1 : \top]}\,(\textsf{Weakening-I})$$

$$\dfrac{\dfrac{\dfrac{}{x \in [0.51, 1] \wedge y \in [0, 0.26] \wedge A \ \vdash\ \bot \quad [\bot]}\,(\textsf{ThLem-I})}{x \in [0.51, 1] \wedge y \in [0, 0.26] \wedge A \wedge B \ \vdash\ \bot \quad [\bot]}\,(\textsf{Weakening-I}) \qquad \vdots\ [I_1]}{x \in [-0.51, 1] \wedge y \in [0, 0.26] \wedge a \wedge b \ \vdash\ \bot \quad [I_2 : x \leq 0.51]}\,(\textsf{Split-I})$$

$$\dfrac{\vdots\ [I_2] \qquad \dfrac{\dfrac{}{x \in [-1, -0.51] \wedge y \in [0, 0.26] \wedge A \ \vdash\ \bot \quad [\bot]}\,(\textsf{ThLem-I})}{x \in [-1, -0.51] \wedge y \in [0, 0.26] \wedge A \wedge B \ \vdash\ \bot \quad [\bot]}\,(\textsf{Weakening-I})}{x \in [-1, 1] \wedge y \in [0, 0.26] \wedge a \wedge b \ \vdash\ \bot \quad [I_3 : -0.51 \leq x \leq 0.51]}$$

$$\dfrac{\vdots\ [I_3] \qquad \dfrac{\dfrac{}{x \in [-1, 1] \wedge y \in [0.26, 1] \wedge B \ \vdash\ \bot \quad [\top]}\,(\textsf{ThLem-I})}{x \in [-1, 1] \wedge y \in [0.26, 1] \wedge A \wedge B \ \vdash\ \bot \quad [\top]}\,(\textsf{Weakening-I})}{x \in [-1, 1] \wedge y \in [0, 1] \wedge A \wedge B \ \vdash\ \bot \quad [I_4 :\ 0.26 \leq y \vee (y \leq 0.26 \wedge I_3)]}\,(\textsf{Split-I})$$

$$\dfrac{\dfrac{\dfrac{}{x \in [-1, 1] \wedge y \in [-1, 0] \wedge A \ \vdash\ \bot \quad [\bot]}\,(\textsf{ThLem-I})}{x \in [-1, 1] \wedge y \in [-1, 0] \wedge A \wedge B \ \vdash\ \bot \quad [\bot]}\,(\textsf{Weakening-I}) \qquad \vdots\ [I_4]}{x \in [-1, 1] \wedge y \in [-1, 1] \wedge A \wedge B \ \vdash\ \bot \quad [I_5 :\ 0 \leq y \wedge I_4]}\,(\textsf{Split-I})$$

Fig. 5: Proof of unsatisfiability where A is $y \geq x^2$, B is $y \leq -\cos(x) + 0.8$ along with the corresponding interpolant

problem

$$\frac{\mathrm{d}\boldsymbol{x}}{\mathrm{d}t} = \boldsymbol{g}(\boldsymbol{x}(t)) \text{ and } \boldsymbol{x}(t_0) = \boldsymbol{x}_0, \text{ where } t \in [t_0, T].$$

It has a solution function $\boldsymbol{x} : [t_0, T] \to \mathbb{R}^n$, which is itself a Type 2 computable function [40]. Thus, in the first-order language $\mathcal{L}_{\mathbb{R}_\mathcal{F}}$ we can write formulas like

$$\Big(||\boldsymbol{x}_0|| = 0\Big) \wedge \Big(\boldsymbol{x}_t = \boldsymbol{x}_0 + \int_0^t \boldsymbol{g}(\boldsymbol{x}(s))\mathrm{d}s\Big) \wedge \Big(||\boldsymbol{x}_t|| > 1\Big)$$

which is satisfiable when the system defined by the vector field $\boldsymbol{g}$ can have a trajectory from some point $||\boldsymbol{x}(0)|| = 0$ to $||\boldsymbol{x}(t)|| = 1$ after time t. Note that we use first-order variable vectors $\boldsymbol{x}_0$ and $\boldsymbol{x}_t$ to represent the value of the solution function $\boldsymbol{x}$ at time 0 and t. Also, the combination of equality and integration in the second conjunct simply denotes a single constraint over the variables $(\boldsymbol{x}_0, \boldsymbol{x}_t, t)$.

In the δ-decision framework, we perform interval-based integration for ODE constraints that satisfies the following. Suppose the time domain for the ODE constraint in question is in $[t_0, T]$. Let $t_0 \leq t_1 \leq \cdots t_m \leq T$ be a sequence of time points. An interval-based integration algorithms compute boxes $D_{t_1}, ..., D_{t_m}$ such that

$$\forall i \in \{1, ..., m\},\ \{\boldsymbol{x}(t) : t_i \leq t \leq t_{i+1}, \boldsymbol{x}_0 \in D_{\boldsymbol{x}_0}\} \subseteq D_{t_0}.$$

Namely, it computes a sequence of boxes such that all possible trajectories are contained in them over time. Thus, the ODE constraints can be handled in the same way as non-ODE constraints, whose solution set is covered by a set of small boxes. Consequently, the proof rules from Figure 4 apply directly to ODE constraints.

3.2 Extensions

For any two formulas A,B which conjunction is unsatisfiable, the interpolant I is not unique. In practice, it is difficult to know a priori what is a good interpolant. Therefore, it is desirable to have the possibility of generating and testing multiple interpolants. We now explain how to get interpolants of different logical strength. An interpolant I_1 is stronger than an interpolant I_2 iff $I_1 \Rightarrow I_2$. Intuitively, a stronger interpolant is closer to A and a weaker interpolant closer to B.

Parameterizing interpolation strength. The interpolation method that we propose uses a δ-decision procedure to build a Craig interpolant. I being an interpolant means that $A \wedge \neg I$ and $B \wedge I$ are both unsatisfiable. However, these formulas might still be δ-satisfiable.

To obtain an interpolant such that both $A \wedge \neg I$ and $B \wedge I$ are δ-unsatisfiable, we can weaken both A and B by a factor δ. However, A and B must be at least 3δ-unsatisfiable to guarantee that the solver finds a proof of unsatisfiability. Furthermore, we can also introduce perturbations only on one side in other to make the interpolant stronger of weaker. To introduce a perturbation δ, we apply the following rewriting to every inequalities in A and/or B:

$$\begin{array}{lcl} L = R & \mapsto & L \geq R - \delta \wedge L \leq R + \delta \\ L \geq R & \mapsto & L \geq R - \delta \\ L > R & \mapsto & L > R - \delta \end{array}$$

Changing the labelling. Due to the similarity of our method to the interpolation of propositional formulas we can adapt the labelled interpolation system from D'Silva et.al. [16] to our framework.

In the labelled interpolation system, it is possible to modify the A,B,AB labelling as long as it preserves *locality*, see [16] for the details. An additional restriction in our case is that we cannot use a projection of constraints at the proof's leaves. The projection is not computable in nonlinear theories. Therefore, the labelling must enforce that the leaves maps to the interpolants $\top$ or $\bot$.

4 Applications and Evaluation

We have implemented the interpolation algorithm in a modified version of the dReal SMT solver.[1] The proofs produced by dReal can be very large, i.e., gigabytes. Therefore, the interpolants are built and simplified on-the-fly. The full proof is not kept in memory. We modified the ICP loop and the contractors which are responsible for the pruning steps. The overhead induced by the interpolant generation over the solving time is smaller than 10%.

The ICP loop (Figure 1) builds a proof starting from the root of the proof tree and exploring the tree like a depth-first search. On the other hand, the interpolation rules build the interpolant starting from the proof's leaves. Our implementation modifies the ICP loop to keep a stack P of partial interpolants alongside the stack of branching points S. When branching (line 9), the value used to split $\boldsymbol{D}_1$ and $\boldsymbol{D}_2$ is pushed on P. The pruning steps (line 5) are converted to a proof as shown in Figure 3. When a contradiction is found (line 7, else branch), P is popped to the branching point where the search resumes and the corresponding partial interpolant is pushed back on P. When the ICP loop ends, P contains the final interpolant.

Interpolant sizes. The ICP algorithm implemented in dReal eagerly prunes the domain by applying repeatedly *all* the constraints. Therefore, it usually generates large proofs often involving all the constraints and all the variables. Interpolation can extract more precise information from the proof. Intuitively, an interpolant which is much smaller than the proof are more likely to be useful in practice. In this test, we try to compare the size of the proof against the size of the interpolants using benchmark from the Flyspeck project [25], certificates for Lyapunov functions in powertrain control systems [27] and the other examples presented in the rest of this section.

[1] Currently available in the branch `https://github.com/dzufferey/dreal3/`.

We run dReal with a 20 minutes timeout and generate 1063 interpolants. Out of these, 501 are nontrivial. In Figure 6 we plot the number of inequalities in the nontrivial interpolants against the size of the proof without the Weakening steps. For similar proofs, we see that the interpolants can be order of magnitude simpler than the proofs and other interpolants obtained by different partitions of the formula. The trivial interpolants still bring information as they mean that the only one side is part of the unsatisfiable core.

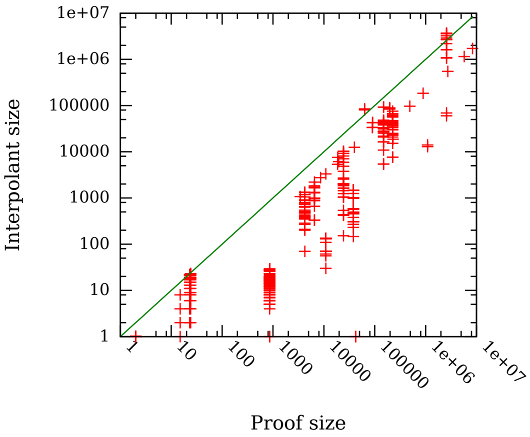


Fig. 6: Interpolants' size (number of inequalities) compared to the proofs' size.

Hybrid system verification. Our method can compute interpolants for systems of ODEs. For instance, we can check that two trajectories do not intersect. Figure 7a shows an interpolant obtained for the following equations:

$$A: \qquad x_t = x_0 + \int_0^t -x + cos(x)\, \mathrm{d}x \wedge x_0 = 3 \wedge 0 \leq t \leq 2$$

$$B: \qquad y_t = y_0 + \int_0^t -y + sin(y-1)\, \mathrm{d}y \wedge y_0 = 2 \wedge x_t = y_t$$

A large portion, 479 out of 1063, of our examples involves differential equations. These examples include: airplane control [5], bouncing balls, networked water tanks, models of cardiac cells [29], verification of the trajectory planning and tracking stacks of autonomous vehicle (in particular, for lane change maneuver [4]), and example from dReal regression tests. Table 1 shows statistics about the interpolants for each family of examples.

Robotic design. Often, hybrid system verification is used in model-based design. An expert produces a model of the system which is then analysed. However, it

Family	#tests	#flow	#var	proof size	interpolant size	time
Airplane control	53	[1,4]	[56,61]	[4213,24249]	[70,10260]	[57s,178s]
Apex	17	1	44	23	[0,22]	[5s,9s]
Bouncing ball	165	2	128	857	[0,28]	[1.6s,5.5s]
Cardiac cells	37	4	71	15	[0,1]	[15m,20m]
Water tanks	68	[4,8]	[18,30]	[6530,225099]	[331,92594]	[7s,12m]
Lane change	15	1	44	24	[0,23]	[19s,20s]
Other tests	142	1	5	2	[0,1]	[0.1s,1s]

Table 1: Results for the interpolation of ODEs. The [_,_] notation stands for intervals that cover the values for the whole families of examples. The first column indicates the family. The next three columns contains the number of tests in the family, the number of flows and variables in the tests. The last three columns shows the size of the proofs, interpolants, and the solving time.

is also possible to extract models directly from the manufacturing designs. As part of an ongoing project about co-design of both the software and hardware component of robots [42], we extract equations from robotic designs. In the extracted models, each structural element is represented by a 3D vector for its position and a unit quaternion for the orientation. The dimension of the elements and the joints connecting them corresponds to equations that relate the position and orientation variables. Active elements, such as motors, also have specific equations associated to them.

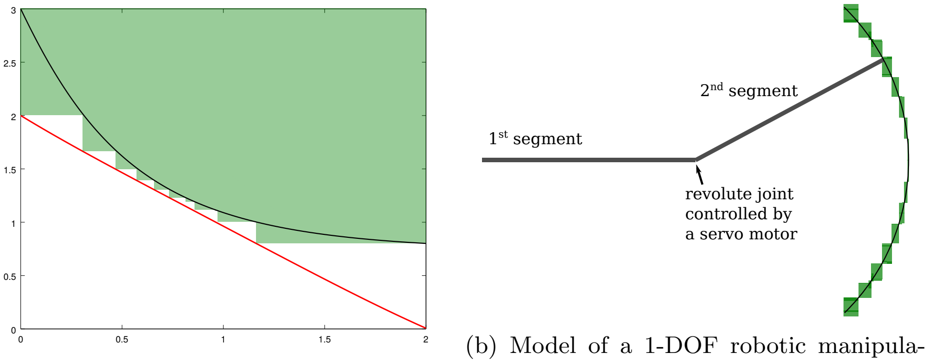
This approach provides models faithful to the actual robots, but it has the downside of producing large systems of equations. To verify such systems, we need to simplify them. Due to the presence of trigonometric functions we cannot use quantifier elimination for polynomial systems of equations [13]. However, we use interpolation as an approximation of quantifier elimination.

Let us consider a kinematic model, $\mathcal{K}(\boldsymbol{x}, \boldsymbol{y}, \boldsymbol{z})$ where $\boldsymbol{x}$ is a set of design and input parameters, $\boldsymbol{y}$ is the variables that represent the state of each component of the robot, and $\boldsymbol{z}$ is the variables that represent the parts of the state needed to prove the property of interest. For instance, in the case of a robotic manipulator, $\boldsymbol{x}$ contain the sizes of each element and the angles of the servo motors and $\boldsymbol{z}$ is the position of the effector. $\boldsymbol{y}$ is determined by the designed of the manipulator.

Fully controlled systems have the property that once the design and input parameters are fixed, there is a unique solution for remaining variables in the model. Therefore, we can create an interpolation query:

$$\begin{aligned} A: &\quad \mathcal{K}(\boldsymbol{x}, \boldsymbol{y}, \boldsymbol{z}) \wedge \\ B: &\quad \mathcal{K}(\boldsymbol{x}, \boldsymbol{v}, \boldsymbol{w}) \wedge (\boldsymbol{z} - \boldsymbol{w})^2 \geq \epsilon^2 \quad \text{where } \epsilon > \delta \end{aligned}$$

$\boldsymbol{y}, \boldsymbol{v}$ are two copies of the variables we want to eliminate. Since the kinematic is a function of $\boldsymbol{x}$ which is the same for the two copies $\boldsymbol{z}$ and $\boldsymbol{w}$ should be equal. Therefore, the formula we build has no solution and we get an interpolant $I(\boldsymbol{x}, \boldsymbol{z})$ which is an ϵ-approximation of $\exists \boldsymbol{y}.\, \mathcal{K}(\boldsymbol{x}, \boldsymbol{y}, \boldsymbol{z})$.



(a) Interpolant for system of nonlinear ODEs. The black and red curves are the trajectories described by A and B. The green area is the interpolant.

(b) Model of a 1-DOF robotic manipulator composed of two 100mm long segments. The black line shows the effector's reach and the green cubes are the approximation obtained by interpolation where ϵ is 10mm.

Fig. 7: Application of interpolation to nonlinear systems

Example 3. Consider the simple robotic manipulator show in Figure 7b. The manipulator has one degree of freedom. It is composed of two beams connected by a revolute joint controlled by a servo motor. The first beam is fixed.

The original system of equations describing this system has 22 variables: 7 for each beam, 7 for the effector, and 1 for the revolute joint. Using the interpolation we obtain a simpler formula with only 4 variables: 3 for the effector's position and 1 for the joint. Table 2 shows some statistics about the interpolants we obtained using different ϵ for a one and a two degrees of freedom manipulators.

5 Conclusion and Future Work

We present an method for computing Craig interpolants for first-order formulas over real numbers with a wide range of nonlinear functions. Our method transform proof traces from δ decision procedures into interpolants consisting of disjunctive linear constraints. The algorithms are guaranteed to find the interpolants between two formulas A and B whenever $A \wedge B$ is not δ-satisfiable. Furthermore, we show how the framework apply to systems of ordinary differential equations. We implemented our interpolation algorithm in the dReal SMT-solver and apply the method to domains such robotic design, and hybrid system verification.

In the future, we plan to expand our work to richer proof systems. The ICP loop produces proof based on interval pruning which results in large, "squarish" interpolants. Using more general proof systems, e.g. cutting planes and semi-definite programming [15], we will be able to get smaller, smoother interpolants. CDCL-style reasoning for richer theories, e.g., LA($\mathbb{R}$) [36] and polynomial [26], is a likely basis for such extensions. Furthermore, we are interested in investigating the link between classical interpolation and Craig interpolation over the reals.

1-DOF Model	#var	Theory	#th. atoms	time
original	22	polynomial deg. 2, trig. fct.	24	-
$\epsilon = 10$	4	linear	1073	0.3s
$\epsilon = 5$	4	linear	2757	0.6s
$\epsilon = 3$	4	linear	3307	0.8s
$\epsilon = 2$	4	linear	6137	1.3s
$\epsilon = 1$	4	linear	12485	2.6s

2-DOF Model	#var	Theory	#th. atoms	time
original	30	polynomial deg. 2, trig. fct.	32	-
$\epsilon = 10$	5	linear	45686	2m 7s
$\epsilon = 7$	5	linear	97068	3m 51s
$\epsilon = 5$	5	linear	184762	6m 41s
$\epsilon = 3$	5	linear	547558	19m 4s
$\epsilon = 2$	5	linear	1151454	41m 51s

Table 2: Comparison of the original model of a 1 and 2 degrees of freedom manipulator against approximations obtained using interpolation. For the size of the formulas we report the number of theory atoms in the formula. The last column shows the time dReal takes to compute the interpolants.

Using methods like spline interpolation and radial basis functions, it maybe possible to build smoother interpolants. We also to extend the our rules to compute interpolants mixed proofs with both integer and real variables.

References

1. A. Albarghouthi, J. Berdine, B. Cook, and Z. Kincaid. Spatial interpolants. In J. Vitek, editor, *ESOP*. Springer, 2015.
2. A. Albarghouthi, A. Gurfinkel, and M. Chechik. Whale: An interpolation-based algorithm for inter-procedural verification. In V. Kuncak and A. Rybalchenko, editors, *VMCAI*. Springer, 2012.
3. A. Albarghouthi, Y. Li, A. Gurfinkel, and M. Chechik. Ufo: A framework for abstraction- and interpolation-based software verification. In P. Madhusudan and S. A. Seshia, editors, *CAV*. Springer, 2012.
4. M. Althoff and J. M. Dolan. Online verification of automated road vehicles using reachability analysis. *Robotics, IEEE Transactions on*, 30(4):903–918, 2014.
5. K. Bae, J. Krisiloff, J. Meseguer, and P. C. Ölveczky. Designing and verifying distributed cyber-physical systems using multirate pals: An airplane turning control system case study. *Science of Computer Programming*, 103:13 – 50, 2015. Selected papers from the First International Workshop on Formal Techniques for Safety-Critical Systems (FTSCS 2012).
6. F. Benhamou and L. Granvilliers. Continuous and interval constraints. In F. Rossi, P. van Beek, and T. Walsh, editors, *Handbook of Constraint Programming*, chapter 16. Elsevier, 2006.
7. V. Brattka, P. Hertling, and K. Weihrauch. A tutorial on computable analysis. In S. B. Cooper, B. Löwe, and A. Sorbi, editors, *New Computational Paradigms*, pages 425–491. Springer New York, 2008.

8. A. Brillout, D. Kroening, P. Rümmer, and T. Wahl. An interpolating sequent calculus for quantifier-free presburger arithmetic. In J. Giesl and R. Hähnle, editors, *IJCAR*. Springer, 2010.
9. X. Chen, E. Ábrahám, and S. Sankaranarayanan. Flow*: An analyzer for non-linear hybrid systems. In N. Sharygina and H. Veith, editors, *CAV*. Springer, 2013.
10. J. Christ, E. Ermis, M. Schäf, and T. Wies. Flow-sensitive fault localization. In R. Giacobazzi, J. Berdine, and I. Mastroeni, editors, *VMCAI*. Springer, 2013.
11. J. Christ, J. Hoenicke, and A. Nutz. Smtinterpol: An interpolating SMT solver. In A. F. Donaldson and D. Parker, editors, *SPIN*. Springer, 2012.
12. A. Cimatti, A. Griggio, B. Schaafsma, and R. Sebastiani. The MathSAT5 SMT Solver. In N. Piterman and S. Smolka, editors, *TACAS*. Springer, 2013.
13. G. Collins and H. Hong. Partial cylindrical algebraic decomposition for quantifier elimination. In B. Caviness and J. Johnson, editors, *Quantifier Elimination and Cylindrical Algebraic Decomposition*, Texts and Monographs in Symbolic Computation, pages 174–200. Springer Vienna, 1998.
14. W. Craig. Linear reasoning. A new form of the Herbrand-Gentzen theorem. *J. Symb. Logic*, 22:250–268, 1957.
15. L. Dai, B. Xia, and N. Zhan. Generating non-linear interpolants by semidefinite programming. In N. Sharygina and H. Veith, editors, *CAV*. Springer, 2013.
16. V. D'Silva, D. Kroening, M. Purandare, and G. Weissenbacher. Interpolant strength. In G. Barthe and M. V. Hermenegildo, editors, *VMCAI*. Springer, 2010.
17. E. Ermis, M. Schäf, and T. Wies. Error invariants. In D. Giannakopoulou and D. Méry, editors, *FM*. Springer, 2012.
18. S. Gao, J. Avigad, and E. M. Clarke. Delta-complete decision procedures for satisfiability over the reals. In Gramlich et al. [23], pages 286–300.
19. S. Gao, J. Avigad, and E. M. Clarke. Delta-decidability over the reals. In *LICS*. IEEE Computer Society, 2012.
20. S. Gao, S. Kong, and E. M. Clarke. dreal: An SMT solver for nonlinear theories over the reals. In M. P. Bonacina, editor, *Automated Deduction - CADE-24 - 24th International Conference on Automated Deduction, Lake Placid, NY, USA, June 9-14, 2013. Proceedings*, volume 7898 of *Lecture Notes in Computer Science*, pages 208–214. Springer, 2013.
21. S. Gao, S. Kong, and E. M. Clarke. Satisfiability modulo odes. In *FMCAD*. IEEE, 2013.
22. S. Gao, S. Kong, and E. M. Clarke. Proof generation from delta-decisions. In F. Winkler, V. Negru, T. Ida, T. Jebelean, D. Petcu, S. M. Watt, and D. Zaharie, editors, *SYNASC*. IEEE, 2014.
23. B. Gramlich, D. Miller, and U. Sattler, editors. *Automated Reasoning - 6th International Joint Conference, IJCAR 2012, Manchester, UK, June 26-29, 2012. Proceedings*, volume 7364 of *Lecture Notes in Computer Science*. Springer, 2012.
24. A. Griggio, T. T. H. Le, and R. Sebastiani. Efficient interpolant generation in satisfiability modulo linear integer arithmetic. In P. A. Abdulla and K. R. M. Leino, editors, *TACAS*. Springer, 2011.
25. T. Hales, M. Adams, G. Bauer, D. Tat Dang, J. Harrison, T. Le Hoang, C. Kaliszyk, V. Magron, S. McLaughlin, T. Tat Nguyen, T. Quang Nguyen, T. Nipkow, S. Obua, J. Pleso, J. Rute, A. Solovyev, A. Hoai Thi Ta, T. N. Tran, D. Thi Trieu, J. Urban, K. Khac Vu, and R. Zumkeller. A formal proof of the Kepler conjecture. *ArXiv e-prints*, Jan. 2015.
26. D. Jovanovic and L. M. de Moura. Solving non-linear arithmetic. In Gramlich et al. [23], pages 339–354.

27. J. Kapinski, J. V. Deshmukh, S. Sankaranarayanan, and N. Arechiga. Simulation-guided lyapunov analysis for hybrid dynamical systems. In M. Fränzle and J. Lygeros, editors, *HSCC*. ACM, 2014.
28. S. Kong, S. Gao, W. Chen, and E. M. Clarke. dreach: δ-reachability analysis for hybrid systems. In C. Baier and C. Tinelli, editors, *TACAS*. Springer, 2015.
29. B. Liu, S. Kong, S. Gao, P. Zuliani, and E. M. Clarke. Parameter synthesis for cardiac cell hybrid models using δ-decisions. In P. Mendes, J. O. Dada, and K. Smallbone, editors, *CMSB*. Springer, 2014.
30. K. L. McMillan. Interpolation and sat-based model checking. In W. A. H. Jr. and F. Somenzi, editors, *CAV*. Springer, 2003.
31. K. L. McMillan. An interpolating theorem prover. In K. Jensen and A. Podelski, editors, *TACAS*. Springer, 2004.
32. K. L. McMillan. Lazy abstraction with interpolants. In T. Ball and R. B. Jones, editors, *CAV*. Springer, 2006.
33. K. L. McMillan. Interpolants and symbolic model checking. In B. Cook and A. Podelski, editors, *VMCAI*. Springer, 2007.
34. K. L. McMillan. Interpolants from Z3 proofs. In P. Bjesse and A. Slobodová, editors, *FMCAD*. FMCAD Inc., 2011.
35. K. L. McMillan. Widening and interpolation. In E. Yahav, editor, *SAS*. Springer, 2011.
36. K. L. McMillan, A. Kuehlmann, and M. Sagiv. Generalizing DPLL to richer logics. In A. Bouajjani and O. Maler, editors, *CAV*. Springer, 2009.
37. P. Pudlák. Lower bounds for resolution and cutting plane proofs and monotone computations. *J. Symbolic Logic*, 62(3):981–998, 1997.
38. A. Rybalchenko and V. Sofronie-Stokkermans. Constraint solving for interpolation. In *VMCAI*, 2007.
39. M. Schäf, D. Schwartz-Narbonne, and T. Wies. Explaining inconsistent code. In B. Meyer, L. Baresi, and M. Mezini, editors, *ACM SIGSOFT*. ACM, 2013.
40. K. Weihrauch. *Computable Analysis: An Introduction*. 2000.
41. G. Yorsh and M. Musuvathi. A combination method for generating interpolants. In R. Nieuwenhuis, editor, *CADE*. Springer, 2005.
42. D. Zufferey, A. Mehta, J. DelPreto, S. Sidiroglou-Douskos, M. Rinard, and D. Rus. Talos: Full stack robot compilation, simulation, and synthesis. Submitted to ICRA'16.