arXiv:cs.LO/0105021 11 May 2001

Solving Composed First-Order Constraints from Discrete-Time Robust Control

Stefan Ratschan[1]* and Luc Jaulin[2]

[1] Research Institute for Symbolic Computation, A-4040 Linz, Austria,
e-mail: stefan.ratschan@risc.uni-linz.ac.at

[2] Luc Jaulin, LISA, 62 avenue Notre Dame du Lac, 49 000 Angers, France,
e-mail: jaulin@univ-angers.fr

Abstract. This paper deals with a problem from discrete-time robust control which requires the solution of constraints over the reals that contain both universal and existential quantifiers. For solving this problem we formulate it as a program in a (fictitious) constraint logic programming language with explicit quantifier notation. This allows us to clarify the special structure of the problem, and to extend an algorithm for computing approximate solution sets of first-order constraints over the reals to exploit this structure. As a result we can deal with inputs that are clearly out of reach for current symbolic solvers.

1 Introduction

A discrete time system can be described by state-space equations of the form $\boldsymbol{x}_{k+1} = f(\boldsymbol{x}_k, \boldsymbol{u}_k, \boldsymbol{w}_k)$, where k is the discrete time, f is a non-linear real function, $\boldsymbol{x}_k$ is the state vector at time k, $\boldsymbol{u}_k$ is the control vector which can be chosen arbitrarily in a set U_k, and $\boldsymbol{w}_k$ is the perturbation vector which cannot be influenced by us but remains inside a known set W_k. In this paper we deal with the problem of computing the set of state vectors $\boldsymbol{x}_0$ for which we can set the controls in such a way that future state vectors $\boldsymbol{x}_1, \ldots, \boldsymbol{x}_n$ will belong to certain sets $X_1, \ldots, X_n$ chosen by us.

This problem is closely related to the problem of characterization of viability sets involved when studying the evolution of macro-systems arising in biology, economics, cognitive sciences, games, and similar areas, as well as in nonlinear systems of control theory. A good reference is the book of Aubin [4].

When trying to solve this problem one immediately runs into constraints that contain a large number of universally and existentially quantified variables. In this paper we report on ongoing research on a method for solving such constraints. This method is already able to compute (approximate) solutions that are far too hard to compute for current symbolic solvers (e.g., QEPCAD [12]).

We proceed by giving a recursive formulation of the problem which can be read as a program in a constraint logic programming language with explicit quantifier notation. This formulation will allow us to clarify the special structure

* Supported by the Austrian Science Fund FWF in the frame of the project SFB F1303

of the resulting first-order constraint (i.e., formula in the first-order predicate language with predicate symbols $=$ and $\leq$, function symbols $+$ and $\times$, and rational constants, all of them with their usual interpretation). We exploit this structure by breaking the constraint into parts, where each part corresponds to one stage of the system, and these parts are glued together by a syntactic entity called "function inversion quantifier". We show how to extend the method of approximate quantified constraint solving (AQCS [32,30]), as developed by one of the authors, to deal with such function inversion quantifiers. This will allow us to solve non-trivial instances of the mentioned problem from discrete-time robust control.

The resulting approach is also applicable to several other problems from discrete-time control, and even to the general case of first-order constraints with composition structure, that is, first-order constraints that have the form $C(f_n(\ldots f_1(\boldsymbol{x})\ldots))$. Since defining objects in hierarchies is ubiquitous in the human modeling and problem-solving process, and since such a composition structure very often is (implicitly) encoded in programs in CLP languages, we believe that according support in constraint solvers will become more and more important.

2 Problem Definition

A *discrete-time system with control and perturbation* (or short a *system*) is a tuple (f, n, X, U, W), where

- $f : \mathbb{R}^{a+u+w} \rightarrow \mathbb{R}^a$,
- $n \in \mathbb{N}$,
- $A \subseteq \{1, \ldots, n\} \times \mathbb{R}^a$,
- $U \subseteq \{1, \ldots, n\} \times \mathbb{R}^u$, and
- $W \subseteq \{1, \ldots, n\} \times \mathbb{R}^w$.

The function f is called *transition function*, the predicate A *allowed state*, the predicate U *allowed control*, the predicate W *possible perturbation*. Intuitively (see Figure 1), such a system models a process which starts with some value $\boldsymbol{x_0}$ (the *initial state*) and then applies the function f n-times to $\boldsymbol{x_0}$, where at each application additional user-definable input (control) $\boldsymbol{u}_k$ s.t. $U(k, \boldsymbol{u}_k)$, and perturbation $\boldsymbol{w}_k$ s.t. $W(k, \boldsymbol{w}_k)$ influence the function f — resulting in the *state at stage k*. In this paper we solve the *problem of computing the robust feasible initial set*: Given a discrete-time system with control and perturbation find an initial state such that for all following stages the state is allowed (i.e., the predicate A holds).

Given a system $S = (f, n, A, U, W)$ we can formalize this using the following predicate, which models the states $\boldsymbol{x}$ at stage k for which we can apply input

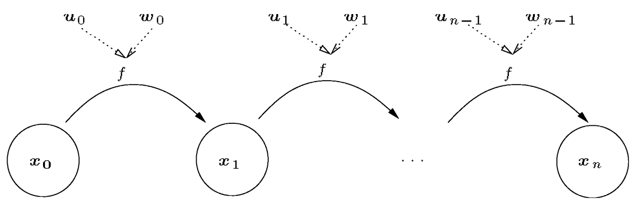


Fig. 1. Discrete Time System

such that for all possible perturbations the state at all future stages is allowed:

$$\begin{aligned} C_S(\boldsymbol{x}, k) \quad :\longleftrightarrow\ & A(\boldsymbol{x}, k) \wedge \\ & \quad k < n \rightarrow \\ & \qquad \exists \boldsymbol{u}\ U(\boldsymbol{u}, k) \wedge \\ & \qquad\quad \forall \boldsymbol{w}\ W(\boldsymbol{w}, k) \rightarrow C_S(f(\boldsymbol{x}, \boldsymbol{u}, \boldsymbol{w}), k+1) \end{aligned} \tag{1}$$

Our problem is to compute the robust feasible initial set $C_S(\boldsymbol{x}_0, 0)$. After expanding this query and writing the predicates U and W as sets, we get:

$$\begin{aligned} & A_0(\boldsymbol{x}_0) \wedge \exists \boldsymbol{u}_0 \in U_0\ \forall \boldsymbol{w}_0 \in W_0 \\ & \quad A_1(f(\boldsymbol{x}_0, \boldsymbol{u}_0, \boldsymbol{w}_0)) \wedge \exists \boldsymbol{u}_1 \in U_1\ \forall \boldsymbol{w}_1 \in W_1 \\ & \qquad A_2(f(f(\boldsymbol{x}_0, \boldsymbol{u}_0, \boldsymbol{w}_0), \boldsymbol{u}_1, \boldsymbol{w}_1)) \wedge \exists \boldsymbol{u}_2 \in U_2\ \forall \boldsymbol{w}_2 \in W_2 \\ & \qquad\quad \ldots \\ & \qquad\qquad A_n(f(f(\ldots f(\boldsymbol{x}_0, \boldsymbol{u}_0, \boldsymbol{w}_0), \boldsymbol{u}_1, \boldsymbol{w}_1), \boldsymbol{u}_2, \boldsymbol{w}_2)) \end{aligned} \tag{2}$$

This is an expression that is definitely too hard to solve for current symbolic solvers like QEPCAD [12] (for which the border between solvable and unsolvable problems is around 3-5 variables), and in its raw form also for the approximate solver developed by one of the authors [32,30]. Thus we have to exploit the special structure of such a constraint in order to be able to solve it.

3 Main Idea

Observe that in the recursive definition of C_S in Expression 1, we only have a fixed number of variables in each recursion step. When expanding the definition, each step maps the newly introduced variables into a variable vector of lower dimension by applying the function f. In the other direction, when collecting results, we can first compute the solution set of step n, plug the result into step $n-1$, compute this solution set, and so on.

Here we have to solve a first-order constraint in $a + u + w$ variables for each step. But even if we could compute the solution set of step n by a symbolic solver, the size of the result will blow up when plugging it into the preceding steps $n-1$, $n-2$, and so on. The reason is that the complexity of real quantifier elimination results from the size of its quantifier-free output [40,13]. So we use the same idea to compute approximate solutions instead.

For this we abstract from our concrete example and assume a constraint C for which we want to compute the solution set of $C(f(\boldsymbol{x}))$, where the dimension of $\boldsymbol{x}$ is higher than the dimension of $f(\boldsymbol{x})$. However, we want to avoid substituting $f(\boldsymbol{x})$ into the constraint because that would blow up the complexity by forcing us to solve a constraint with the additional variables $\boldsymbol{x}$.

Thus we try to deal with such a situation in a special, more efficient way. For this we write it by using *function inversion quantifiers*, that is, syntactical entities of the form $[f(\boldsymbol{x}) = \boldsymbol{z}]\ C(\boldsymbol{z})$. Semantically such a constraint is equivalent to $C(f(\boldsymbol{x}))$. Operationally it will be handled in a different, more efficient way. Now our problem reads as follows:

$$\begin{array}{l} A_0(\boldsymbol{x}_0) \wedge \exists \boldsymbol{u}_0 \in U_0\ \forall \boldsymbol{w}_0 \in W_0\ [f(\boldsymbol{x}_0, \boldsymbol{u}_0, \boldsymbol{w}_0) = \boldsymbol{x}_1] \\ \quad A_1(\boldsymbol{x}_1) \wedge \exists \boldsymbol{u}_1 \in U_1\ \forall \boldsymbol{w}_1 \in W_1\ [f(\boldsymbol{x}_1, \boldsymbol{u}_1, \boldsymbol{w}_1) = \boldsymbol{x}_2] \\ \quad\quad A_2(\boldsymbol{x}_2) \wedge \exists \boldsymbol{u}_2 \in U_2\ \forall \boldsymbol{w}_2 \in W_2\ [f(\boldsymbol{x}_2, \boldsymbol{u}_2, \boldsymbol{w}_2) = \boldsymbol{x}_3] \\ \quad\quad\quad \ldots \\ \quad\quad\quad\quad A_n(\boldsymbol{x}_n) \end{array} \tag{3}$$

Observe that in this constraint each sub-constraint beginning at a certain line has exactly one free-variable vector $\boldsymbol{x}_i$. Now we can view our problem of computing the *robust feasible initial set* as the problem of solving n constraints that are glued together by function inversion quantifiers. For $1 \leq i \leq n$, the i-th of these constraints only contains the variable vectors $\boldsymbol{x}_i$, $\boldsymbol{u}_i$, and $\boldsymbol{w}_i$.

For solving these constraints we use the method for approximate quantified constraint solving (AQCS) as developed by the first author [32,30]. In the rest of the paper we will describe how to glue together several instances of this solver by implementing function inversion quantifiers. However, the scope of this paper will only allow us to explain these details of AQCS that are absolutely necessary for understanding how we deal with function inversion quantifiers.

4 Function Inversion in Quantified Constraint Solving

In approximate quantified constraint solving we approximate solution sets by functions that assign "true" to elements that are guaranteed to be in the solution set, "false" to elements that are guaranteed to be out of the solution set, and that assign "unknown" otherwise:

Definition 1. *An* approximate set on $B \subseteq \mathbb{R}^n$ *is a function in* $B \to \{\mathbf{T}, \mathbf{F}, \mathbf{U}\}$. *The* error $\mathrm{err}(\tilde{S})$ of an approximate set $\tilde{S}$ on B *is the volume of the* $\boldsymbol{x}$ *such that* $\tilde{S}(\boldsymbol{x}) = \mathbf{U}$. *An approximate set* $\tilde{S}$ *on* B *is an* approximation of a set $S \subseteq B$ *iff for all* $\boldsymbol{x} \in B$, $\tilde{S}(\boldsymbol{x}) = \mathbf{T}$ *implies that* $\boldsymbol{x} \in S$, *and* $\tilde{S}(\boldsymbol{x}) = \mathbf{F}$ *implies* $\boldsymbol{x} \notin S$.

We delay the description of a concrete computer representation of approximate sets to the end of this section.

The basic idea of the algorithm for approximate quantified constraint solving [32], is to define for every syntactic element p (e.g., $\wedge$, $\forall$) a *propagation function* prop_p that computes an approximation of the solution set of a constraint of the form $p(C_1, \ldots, C_n)$ from approximations of the solution set of each sub-constraint $C_1, \dots, C_n$ [31]. For example, for the constraint $\exists x\ x^2 + y^2 \leq 1$ the propagation function $\text{prop}_\exists$ takes an approximation of the solution set of $x^2 + y^2 \leq 1$ and returns an approximation of the solution set of $\exists x\ x^2 + y^2 \leq 1$. We follow this approach by simply defining such a propagation function also for function inversion quantifiers. This means that we show how to compute an approximation of the solution set of $[f(\boldsymbol{x}) = \boldsymbol{z}]C(\boldsymbol{z})$ (i.e., $C(f(\boldsymbol{x}))$) from an approximation of the solution set of $C(\boldsymbol{z})$. In a first attempt we define for an approximate set $\tilde{S}$ on B, $\text{prop}_{[f(\boldsymbol{x})=\boldsymbol{z}]}(\tilde{S})$ to be $\lambda \boldsymbol{x} \in B.\tilde{S}(f(\boldsymbol{x}))$[1]. We will see later how to implement this in detail.

Now a naive approach would apply the idea from Section 3 by computing an approximate solution set of stage n, then using this information to compute an approximate solution set of stage $n-1$, and so on (this corresponds to a computation that follows the arrows in reverse order in Figure 1). The problem is that here we do not know how small we have to make the error of these approximate solution sets in order to reach a certain desired output error. Furthermore we do not know which parts of these approximate solution sets are needed for the end result. Thus we compute all of them on demand.

This is also how approximate solution sets are computed within AQCS. For this it uses a function refine that takes a constraint C and a set B, and returns an approximation of the solution set of C on B (an exact discussion of how large the error of this approximation is allowed to be is beyond the scope of this paper). The argument B is chosen on demand within the algorithm — it is usually just a small part of the solution set we want to compute, and during the algorithm execution the limit of its volume goes to zero. So we have to extend this refinement function for the case when the outermost symbol of C is a function inversion quantifier:

$$\begin{aligned}&\text{refine}([f(\boldsymbol{x}) = \boldsymbol{z}]C, B) \doteq \\ &\quad \text{prop}_{[f(\boldsymbol{x})=\boldsymbol{z}]}(\text{refine}(C, f(B)))\end{aligned}$$

The resulting overall function is recursive, the base case is reached when called with an atomic constraint (i.e., an equality or inequality). We will deal with the problem of how to compute $f(B)$ (i.e., the range of f on B) later.

However, this solution does not yet suffice to solve the complexity problem. The reason is that for each input set B it computes an approximate solution set of C on $f(B)$. In other words, it substitutes $f(\boldsymbol{x})$ into C at the level of solving instead of at the syntactic level, and we do not gain anything in terms of complexity.

This problem comes up because we never represent the approximate solution set of the sub-constraint C of $[f(\boldsymbol{x}) = \boldsymbol{z}]C$ explicitly. This means that, even if for many different $\boldsymbol{x}$ the value of $f(\boldsymbol{x})$ is the same, we recompute the solution set of C for different $\boldsymbol{x}$ that result in the same $f(\boldsymbol{x})$ by new calls to the refinement function. We solve this problem by remembering already computed parts of the

[1] The notation $\lambda \boldsymbol{x} \in B.f(\boldsymbol{x})$ denotes a function that takes an argument $\boldsymbol{x} \in B$ and returns $f(\boldsymbol{x})$ [5].

solution set of C. We do this by wrapping the above call to the refinement function into the following memoization function which keeps the current knowledge about the solution set of C by caching it in the global variable $\tilde{S}$, which we initialize by the everywhere unknown approximate solution set $\lambda x \in \mathbb{R}.\mathbf{U}$:

```
refineMemo$(C, B) \doteq$
    if $\mathrm{err}(\mathrm{restr}(\tilde{S}, B)) > \varepsilon$ then
        $\tilde{S} \leftarrow \mathrm{embed}(\tilde{S}, \mathrm{refine}(C, \mathrm{choose}(\tilde{S}, B)))$
    return $\mathrm{restr}(\tilde{S}, B)$
```

Here ε is a pre-defined real constant, $\mathrm{restr}(\tilde{S}, B)$ returns the restriction of the approximate set $\tilde{S}$ (as a function) to B, $\mathrm{choose}(\tilde{S}, B)$ returns a subset of B for which $\tilde{S}$ is $\mathbf{U}$, and $\mathrm{embed}(\tilde{S}, \tilde{S}')$ is an approximate set that is equal to $\tilde{S}$ except that it is equal to $\tilde{S}'$ on its domain of definition.

Up to now we have allowed approximate sets to be arbitrary functions in $B \to \{\mathbf{T}, \mathbf{F}, \mathbf{U}\}$. For computer representation we restrict this class to functions that are constant on finitely many floating-point boxes. We can represent these by a set of boxes for which the approximate set is $\mathbf{T}$, a set of boxes for which the approximate set is $\mathbf{F}$, and a set of boxes for which the approximate set is $\mathbf{U}$. Now we can easily implement the above functions using this representation. Especially, we can implement the computation of $f(B)$ in the refinement function by interval methods for computing an overestimation of the range of the function f on a box B [29].

The only problem lies in the implementation of the propagation function for function inversion quantifiers: First, the resulting approximate solution set can have an extremely complicated structure, which can be hard or even impossible to represent by finitely many floating point boxes. Second, all the approximate solution sets occurring in the algorithm have to fulfill the additional property of cylindricity [32,3] (the clarification of this notion is beyond the scope of this paper). To solve these problems remember that the second argument to the refinement function is a set B (i.e., a box), whose volume goes to zero during the algorithm execution. The smaller this box B is, the more probable it is that the argument to the propagation function in the refinement function is an approximate solution set that returns the same truth value everywhere. So we can remove both problems easily by just returning approximate solution sets that return the same truth value everywhere. This means that, given an approximate set $\tilde{S}$ on a set B, we define:

```
$\mathrm{prop}_{[f(\boldsymbol{x})=\boldsymbol{z}]}(\tilde{S}) \doteq$
    if for all $\boldsymbol{x} \in f(B),\ \tilde{S}(\boldsymbol{x}) = \mathbf{T}$ then return $\lambda \boldsymbol{x} \in B.\mathbf{T}$
    else if for all $\boldsymbol{x} \in f(B),\ \tilde{S}(\boldsymbol{x}) = \mathbf{F}$ then return $\lambda \boldsymbol{x} \in B.\mathbf{F}$
    else return $\lambda \boldsymbol{x} \in B.\mathbf{U}$
```

This propagation function always computes constant approximate sets which means that it returns the everywhere unknown function very often. In this case AQCS will bisect the resulting box and do further calls to the refinement function on the single pieces. This need for bisection instead of more intelligent box pruning methods is one of the weaknesses of the current approach.

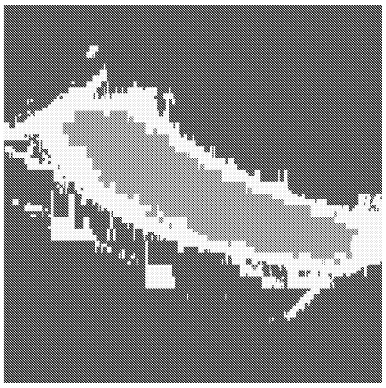
5 Example

We did timings on the following example:

- $f(x_1, x_2, u, w) = (3x_1x_2 + wu, 2x_2 + x_1)$
- $A(x_1, x_2, t) :\leftrightarrow -1 \leq x_1 \leq 1, -1 \leq x_2 \leq 1$
- $U(u, t) :\leftrightarrow -0.5 \leq u \leq 0.5$
- $W(w, t) :\leftrightarrow -0.1 \leq w \leq 0.1$

For $n = 0$ we simply need to solve $-1 \leq x_1 \leq 1, -1 \leq x_2 \leq 1$. For example for $n = 3$, the total number of involved variables is 12. We list timings for $n > 0$ in the table below. All numbers denote seconds needed on a 500MhZ Intel Celeron PC running Linux, where ∞ denotes that the run either needed more than 64MB of memory or more than 20min of time. The column with the title "err=0.2" shows the time for computing an approximate solution set of error 0.2 on the box $[-1, 1] \times [-1, 1]$, the column with the title "single" the time for computing one single true box. The last column lists the time for computing an exact symbolic solution with the solver QEPCAD [12]. For the latter we did not simply feed the input into the program — this would be definitely too hard to solve — but we applied the idea of Section 3 also to this case, by computing a solution of stage n, and then recursively back-substituting the result for computing earlier stages. The figure shows the computed approximate solution set for $n = 2$ with error 0.2 again on the box $[-1, 1] \times [-1, 1]$ (green=**T**, red=**F**, white=**U**).

n	err=0.2	single	QEPCAD
1	0.7	0.0	18.3
2	19.8	0.3	∞
3	∞	11.9	∞
4	∞	∞	∞



6 Related Work

The behavior of various algebraic objects, such as Gröbner bases [19] or subresultants [18], under composition is becoming an important topic in computer algebra. The composition structure of constraints has also been exploited for reuse of certain shared interval function evaluation in global optimization [25,21], and for computing good range overestimations of functions [11,37].

Inversion of functions on sets is done implicitly by every algorithm for solving systems of equations [29] — in this case the input set just contains one zero vector. It is mentioned explicitly mostly for computing the solution set of systems of inequalities [9,39].

First-order constraints occur frequently in control, and especially robust control. Up to now they either have been solved by specialized methods [1,41,6] or by applying general solvers like QEPCAD [12]. In the first case one is usually restricted to conditions like linearity, and in the second case one suffers from the high run-time complexity of computing exact solutions [40,34]. We know of only one case where general solvers for first-order constraints have been applied to discrete-time systems [28], but they have been frequently applied to continuous systems [24,15,14]. For non-linear discrete-time systems without perturbations or control, interval methods have also proved to be an important tool [23,26].

Apart from the method used in this paper [32], there there have been several successful attempts at solving special cases of first-order constraints, for example using classical interval techniques [35,36] or constraint satisfaction [7], and very often in the context of robust control [16,22,27,38].

7 Conclusion

We have designed a method for solving composed first-order constraints arising in discrete-time robust control by extending a solver (AQCS) developed by one of the authors. The result can solve problems that are far too hard to compute for state-of-the-art symbolic solvers. However it is still too slow for solving problems of practical interest. We believe that the method is promising also in other situations where composed first-order constraints occur.

In problems without function inversion quantifiers, AQCS spends most of the time on computing information for atomic constraints, using tightening [20] and range computation. However, for inputs containing function inversion quantifiers, this is not the case. Usually more than 90 percent of the time is spent on propagating (usually unknown) boxes in tasks like memo lookup or choosing boxes for further processing. We envision two basic approaches for dealing with this situation:

– Try to produce less unknown boxes.
– Try to handle the produced boxes more efficiently.

For producing less unknown boxes, the following approaches seem promising:

- Instead of bisection, develop a method similar to tightening [20] or to box consistency methods [17,8] on the level of function inversion quantifiers, for example by implementing the following specification: Given a box B, compute a box B' such that $f(B') \subseteq B$. Then the fact that B is in the solution set of a constraint $C(\boldsymbol{z})$ implies that B' is in $C(f(\boldsymbol{x}))$.
- Pruning boxes that do not contribute to the overall results (in a similar way as the monotonicity test in global optimization).

For handling the produced boxes more efficiently we can:

- Devise strategies for choosing boxes in the context of function inversion quantifiers [33].
- Design efficient algorithms for the memo lookup.
- Parallelize the method by putting each stage on a different processor.

References

1. J. Ackermann. *Robust Control.* Springer, 1993.
2. G. Alefeld, A. Frommer, and B. Lang, editors. *Scientific Computing and Validated Numerics — SCAN'95*, volume 90 of *Mathematical Research.* Akademie Verlag, 1996.
3. D. S. Arnon, G. E. Collins, and S. McCallum. Cylindrical algebraic decomposition I: The basic algorithm. *SIAM Journal of Computing*, 13(4):865–877, 1984. Also in [10].
4. J.-P. Aubin. *Viability Theory.* Birkhäuser, 1991.
5. H. P. Barendregt. *The Lambda Calculus.* North-Holland, 1981.
6. B. R. Barmish and H. I. Kang. A survey of extreme point results for robustness of control systems. *Automatica*, 29(1):13–35, 1993.
7. F. Benhamou and F. Goualard. Universally quantified interval constraints. In *Proc. of the Sixth Intl. Conf. on Principles and Practice of Constraint Programming (CP'2000)*, LNCS, Singapore, 2000. Springer Verlag.
8. F. Benhamou, F. Goualard, L. Granvilliers, and J. F. Puget. Revising hull and box consistency. In *Int. Conf. on Logic Programming*, pages 230–244, 1999.
9. M. Candev. On the application of an interval algorithm to set inversion. In Alefeld et al. [2], pages 140–146.
10. B. F. Caviness and J. R. Johnson, editors. *Quantifier Elimination and Cylindrical Algebraic Decomposition.* Springer, 1998.
11. M. Ceberio and L. Granvilliers. Solving nonlinear systems by constraint inversion and interval arithmetic. In *5th Int. Conf. on Artificial Intelligence and Symbolic Computation*, 2000.
12. G. E. Collins and H. Hong. Partial cylindrical algebraic decomposition for quantifier elimination. *Journal of Symbolic Computation*, 12:299–328, 1991. Also in [10].
13. J. H. Davenport and J. Heintz. Real quantifier elimination is doubly exponential. *Journal of Symbolic Computation*, 5:29–35, 1988.
14. P. Dorato. Quantified multivariate polynomial inequalities. *IEEE Control Systems Magazine*, pages 48–58, October 2000.
15. P. Dorato, W. Yang, and C. Abdallah. Robust multi-objective feedback design by quantifier elimination. *Journal of Symbolic Computation*, 24:153–159, 1997.

16. J. Garloff and B. Graf. Solving strict polynomial inequalities by Bernstein expansion. In N. Munro, editor, *The Use of Symbolic Methods in Control System Analysis and Design*, pages 339–352. The Institution of Electr. Eng. (IEE), 1999.
17. P. V. Hentenryck, D. McAllester, and D. Kapur. Solving polynomial systems using a branch and prune approach. *SIAM Journal on Numerical Analysis*, 34(2), 1997.
18. H. Hong. Subresultant under composition. *Journal of Symbolic Computation*, 23(4):355–365, 1997.
19. H. Hong. Groebner basis under composition I. *Journal of Symbolic Computation*, 25(5):643–663, 1998.
20. H. Hong and V. Stahl. Safe starting regions by fixed points and tightening. *Computing*, 53:323–335, 1994.
21. E. Hyvönen and S. D. Pascale. Shared computations for efficient interval function evaluation. In Alefeld et al. [2], pages 38–44.
22. L. Jaulin and É. Walter. Guaranteed tuning, with application to robust control and motion planning. *Automatica*, 32(8):1217–1221, 1996.
23. L. Jaulin and É. Walter. Global numerical approach to nonlinear discrete-time control. *IEEE Transactions on Automatic Control*, 42(6):872–875, 1997.
24. M. Jirstrand. Nonlinear control system design by quantifier elimination. *Journal of Symbolic Computation*, 24(2):137–152, 1997.
25. R. B. Kearfott. Decomposition of arithmetic expressions to improve the behavior of interval iteration for nonlinear systems. *Computing*, 47(2):169–191, 1991.
26. M. Kieffer, L. Jaulin, É. Walter, and D. Meizel. Robust autonomous robot localization using interval analysis. *Reliable Computing*, 3(6):337–361, 2000.
27. S. Malan, M. Milanese, and M. Taragna. Robust analysis and design of control systems using interval arithmetic. *Automatica*, 33(7):1363–1372, 1997.
28. D. Nešić and I. M. Y. Mareels. Dead beat controllability of polynomial systems: Symbolic computation approaches. *IEEE Transactions on Automatic Control*, 43(2):162–175, 1998.
29. A. Neumaier. *Interval Methods for Systems of Equations*. Cambridge Univ. Press, Cambridge, 1990.
30. S. Ratschan. Approximate quantified constraint solving (AQCS). `http://www.risc.uni-linz.ac.at/research/software/AQCS`, 2000. Software package.
31. S. Ratschan. Uncertainty propagation in heterogeneous algebras for approximate quantified constraint solving. *Journal of Universal Computer Science*, 6(9), 2000.
32. S. Ratschan. Approximate quantified constraint solving by cylindrical box decomposition. *Reliable Computing*, 2001. To appear.
33. S. Ratschan. Search heuristics for box decomposition methods. Technical Report 01-11, Research Institute for Symbolic Computation, Linz, 2001. Submitted.
34. J. Renegar. On the computational complexity and geometry of the first-order theory of the reals. *Journal of Symbolic Computation*, 13(3):255–352, March 1992.
35. S. P. Shary. Algebraic approach to the interval linear static identification, tolerance, and control problems, or one more application of Kaucher arithmetic. *Reliable Computing*, 2(1):3–33, 1996.
36. S. P. Shary. Outer estimation of generalized solution sets to interval linear systems. *Reliable Computing*, 5:323–335, 1999.
37. V. Stahl. *Interval Methods for Bounding the Range of Polynomials and Solving Systems of Nonlinear Equations*. PhD thesis, Research Institute for Symbolic Computation, Johannes Kepler University, A-4040 Linz, Austria, 1996.
38. J. Vehí, J. R. J., M. Á. Sainz, and J. Armengol. Analysis of the robustness of predictive controllers via modal intervals. *Reliable Computing*, 6(3):281–301, 2000.

39. E. Walter and L. Jaulin. Guaranteed characterization of stability domains via set inversion. *IEEE Transaction on Autom. Control*, 39(4):886–889, 1994.
40. V. Weispfenning. The complexity of linear problems in fields. *Journal of Symbolic Computation*, 5(1–2):3–27, 1988.
41. K. Zhou. *Essentials of Robust Control.* Prentice Hall, 1997.